TUTOR INTELIGENTE PARA A PROGRAMAÇÃO EM LOGICA - IDEALIZAÇÃO, PROJECTO E DESENVOLVIMENTO

por

ROSA MARIA VICCARI

Dissertação apresentada a doutoramento em Engenharia Electrotécnica, especialidade de Informática, pela Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

COIMBRA / 1989

Inteligência artificial. SBC/II
Tutores inteligentes

AGRADECIMENTOS

Agradeço à Universidade Federal do Rio Grande do Sul por permitir o meu afastamento para a realização do doutoramento. A Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa de estudos e por acreditar no valor científico de meu trabalho. Ao Laboratório Nacional de Engenharia Civil pelo acolhimento e apoio no desenvolvimento desta investigação. Ao Projecto Minerva e, em particular à Escola no. 1 dos Olivais, nas pessoas dos prof. Cristina Zambujo, José Manuel Almeida por incentivar e acatar a realização das observações práticas. Ao meu orientador Doutor Eng. Helder Coelho (LNEC), pela orientação científica do trabalho. Ao Doutor Eng. Ernesto Costa (UC), pelo acompanhamento e interesse demonstrado durante toda a investigação. Ao meu amigo Gabriel Lopes (UNL), pelo apoio científico e principalmente pela amizade que desenvolvemos durante todos estes anos. A Embaixada do Brasil, pelo apoio quando do surgimento dos primeiros problemas burocráticos.A Ermelinda Ressurreição pelo dedicado auxílio prestado durante o desenvolvimento deste trabalho. Ao Léo, pela compreensão, e pela revisão e adaptação do Português. E, a todos os que de uma ou de outra forma contribuiram para a realização deste trabalho.

SUMARIO

LISTA DAS FIGURAS

RESUMO

Este trabalho de investigação situa-se na área dos Tutores Inteligentes ("Intelligent Tutor"), para o ensino da Programação em Lógica, envolvendo figuras de programação (decisão, repetição e recursão) e estruturas de representação da informação. O Tutor-Prolog ensina um subconjunto da linguagem de programação Prolog através da geração automática de exemplos organizados em níveis de complexidade dentro de um contexto instrucional. Estes exemplos são apresentados simultaneamente em Português e em Prolog.

O ensino de figuras básicas da programação é introduzido através de exercícios orientados pelo Tutor-Prolog. O mesmo ocorre na introdução das estruturas para a representação da informação (listas, árvores e enquadramentos). Todo o processo de ensino é conduzido pelo Tutor-Prolog, com base no modelo do aluno e no modelo da interacção. Portanto, os métodos utilizados são o ensino por indução (tutorial) e por tentativa e erro (depuração).

O modelo de ensino é suportado por conhecimentos e por hipóteses que representam o conhecimento que o aluno possuia ou adquiriu trvés do tutorial.

O Tutor-Prolog aprende através de modelos (do aluno e da interacção) resultantes da aplicação de hipóteses geradas com

base no conhecimento existente a cada momento. Em grande parte dos casos as hipóteses são utilizadas conjuntamente com exemplos. A capacidade de aprendizagem do Tutor é limitada a determinadas áreas de sua intervenção, como a selecção das estratégias de ensino (ao nível do tutorial), os programas que o aluno escreve (ao nível da depuração) e o vocabulário e a sintaxe que utiliza para as consultas (ao nível da interface em língua natural escrita [WEN 87]).

O presente trabalho apoiou-se em experimentação prática realizada com alunos do ensino secundário de uma escola de Lisboa, tendo o Tutor-Prolog resultado de diversas observações feitas com os três protótipos desenvolvidos.

1. INTRODUÇÃO

O Tutor-Prolog é um sistema inteligente, composto por vários programas escritos na linguagem de programação (Arity/)Prolog, capaz de ensinar a programação em lógica a alunos do ensino secundário, na faixa etária dos 14-16 anos.

O Tutor-Prolog pode ser classificado de Tutor Inteligente (TI), pois cria à sua volta um ambiente de interacção cooperativo, ensina e adapta-se de acordo com o modelo do aluno, estabelecendo uma relação amigável com os seus utilizadores. Adopta um modelo de ensino guiado por estratégias, que são utilizadas para a apresentação do material instrucional e para a geração do diagnóstico durante a correcção das falhas detectadas.

O Tutor-Prolog foi concebido numa perspectiva de engenharia, sendo o resultado de um esforço de integração de técnicas e ferramentas de diversas áreas nucleares da IA, com o objectivo de propor um produto industrial (protótipo) a adoptar (posteriormente) em escolas portuguesas e brasileiras. Durante o seu desenvolvimento houve sempre a preocupação de o comparar com seus congéneres de Lisp (LISP ITS da ACT) e Prolog (APT, experiências do Imperial College e do Weizmann Institut), e de avaliar o seu desempenho com grupos de alunos. E, por isso, em vez de se adoptar uma perspectiva de investigar uma zona reduzida de conhecimento, preferiu-se idealizar, projectar e desenvolver um sistema

que fosse útil a curto prazo, retirando-se da experiência com os seus utilizadores lições para alterar e melhorar a sua concepção. Esta perspectiva experimental, frequentemente adoptada em IA e denomidada de empírica, foi defendida, pelos japoneses, desde o início do programa dos Sistemas de Quinta Geração. Também mostrou-se ser a mais útil na área de aplicação dos TI's, considerando as dificuldades várias que impedem o progresso de algumas técnicas da IA necessárias aos TI's.

Do ponto de vista de **originalidade**, o Tutor-Prolog inclui técnicas de aprendizagem ao nível da interpretação da linguagem Prolog (sintaxe, semântica e pragmática - corrige os exercícios do aluno de acordo com a história da sessão) e da interface em Língua Natural (modifica suas próprias regras num processo semelhante aos sistemas TEAM [GRO;APP;MAR;PER 87] e TELI [BAL;STU 84] e [BAL 86]). Nos processos de adaptação e aprendizagem faz recurso às hipoteses (uma espécie de crenças), na linha do proposto por Self [SEL 88] e por Genesereth e Nilsson [GEN;NIL 87], em oposição ao recurso tradicional ao conhecimento. Emprega o modelo "overlay", proposto por Goldstein [GOL 77], que é uma das actuais técnicas para construir os modelos dos utilizadores. A componente de ensino, organizada de modo hierárquico em três níveis de dificuldade, garante a dinâmica do processo de ensino, pois o aluno pode iniciar a interacção a partir de qualquer nó da árvore de lições e não apenas a partindo da raiz, como acontece tradicionalmente. A organização do material instrucional permite o fornecimento do resumo de cada bloco e ainda a geração automática

de exemplos (frases em Português e cláusulas Prolog), de acordo com o modelo do aluno. A proposta de exercícios adoptada no Tutor-Prolog é também defendida na proposta de um projecto Alvey "Learning Prolog", defendida por Brna e Bundy em 1988 no Reino Unido.

Assim, os principais tópicos aprofundados e desenvolvidos durante a investigação foram os seguintes:

- Investigação de ambientes tutoriais para o ensino de linguagens de programação.
- Observação de formas de interacção homem-máquina (menus, ícones e língua natural escrita). O Tutor-Prolog representa uma evolução ao nível dos TI no que se refere às interfaces correntes, pois introduz a aprendizagem como forma de flexibilizar a comunicação em língua natural escrita. Os sistemas tradicionais como o NEOMYCIN [CLA 84], SCHOLAR [CAR 70] e WHY [STE;COL 77], apenas aceitam construções sintácticas e semânticas correctas e não possuem flexibilidade para a aprendizagem ao nível da língua natural.
- Flexibilização do Tutor-Prolog ao nível da utilização (facilidades de controle, adaptação dinâmica às necessidades de cada aluno e modularização do protótipo). A flexibilização envolveu a modelação do aluno, a aprendizagem e o uso de características exploratórias.
- Geração automática e dinâmica de exemplos para a adaptação do material instrucional às deficiências e às necessidades de cada aluno.

Defendemos que para ser inteligente um tutor deve ser **flexível** e ter capacidade para **aprender** com o meio-ambiente. Deve poder actualizar periodicamente o seu conhecimento, recorrendo a operações de particularização ou de generalização, ao comparar o conhecimento representado nos modelos dos diferentes alunos com o seu próprio conhecimento.

De acordo com as nossas observações, a flexibilidade de um tutor é o resultado de vários factores como a capacidade de reconfiguração automática ao longo de uma sessão de trabalho com o aluno (arquitectura dinâmica); a capacidade para acompanhar (perceber) o raciocínio e a estratégia que determinado aluno está utilizando na solução de um problema; e a capacidade de adaptação à linguagem do aluno.

Ainda, do ponto de vista de originalidade, o Tutor-Prolog possui um conjunto de características que o distinguem dos outros TI's estadados, embora tenha aproveitado e melhorado os aspectos fulcrais e centrais desta área de investigação. Em resumo, as nossas contribuições originais são as seguintes:

(a) **tratamento da flexibilidade do sistema**, incluindo arquitectura, controle, comunicação, adaptação ao aluno e adaptação do material instrucional;

(b) **navegação ao longo dos conteúdos instrucionais**, recorrendo à sua taxonomia inicial, a planos de ensino dinâmicos e à exploração do raciocínio;

(c) **inclusão de técnicas de aprendizagem**, para adequar o sistema ao estilo do aluno, ao nível da interacção Prolog (corrige os exercícios do aluno) em língua natural (modifica as suas próprias regras);

(d) **uso de metaconhecimento e de crenças**, para resolver situações não previstas nas regras que descrevem o conhecimento do Tutor;

(e) **tratamento do modelo do aluno**, através do uso de "overlays" misturados com uma componente de adaptação constituída de hipóteses; e,

(f) **recurso à detecção de falhas e à sua depuração**, para amplificar a flexibilidade e suportar a resolução conduzida de problemas.

As limitações do presente trabalho relacionam-se com o tipo de equipamento e de "software" escolhidos para o desenvolvimento e a aplicação: microcomputadores tipo IBM PC (16 bits), com 20 Megabytes de memória em disco, e a linguagem de programação Arity/Prolog. A escolha deve-se ao facto de este tipo de equipamento ser mais acessível, em termos financeiros, às escolas secundárias a que o Tutor-Prolog se destina (Portugal - Projecto MINERVA e Brasil).

E de ressaltar que, desde o princípio, o trabalho visou a realização de experiências num ambiente real e com utilidade imediata. A investigação foi baseada em observações práticas, sendo as várias versões do Tutor dirigidas sempre ao mesmo grupo de utilizadores. O primeiro protótipo foi definido com base em conhecimentos que possuíamos sobre o ensino da linguagem Prolog e em hipóteses sobre as exigências dos potenciais utilizadores [VIC 86]. Depois adoptámos as técnicas utilizadas em sistemas para o ensino assistido por computador inteligente, tais como o "ensino dirigido", a condução do aluno na direcção de um objectivo, a aceitação de perguntas, o fornecimento de explicações e a simulação das hipótese de correcção de falhas e da execução dos programas [VIC 84a,85]. Evoluímos para o desenvolvimento de um terceiro protótipo, já apoiado em conceitos precisos de Tutores Inteligentes: depuração de programas, diagnóstico orientado, capacidades para a aprendizagem, diálogos e regras estratégicas de ensino [VIC 87,88]. A evolução visou sempre a adequação e a solução das carências que os sistemas apresentavam em relação às

exigências dos utilizadores do Tutor-Prolog.

Com o Tutor-Prolog o aluno aprende fazendo ("learning by doing"), isto é, resolvendo os exercícios propostos e criando os seus próprios problemas. Para isso, delimitámos, através da pesquisa e da observação, uma organização didáctica adequada ao ensino da linguagem Prolog para uma classe bem definida de alunos do ensino secundário (14-16 anos), a saber:

- **factos** com a introdução da noção de átomos;
- **objectivos** com a introdução do uso de variáves e relações;
- **regras** com a utilização de estruturas de representação da informaçao e a apresentação de figuras de programação;
- **listas** e dentro deste tópico a introdução, em particular, da noção de procedimentos recursivos.

Esta organização é o modelo padrão de ensino do Tutor-Prolog, e por conseguinte serve para ensinar o aluno a gerar, consultar e explorar pequenas bases de dados, onde os dados são representados por um conjunto de factos e as consultas são feitas através de objectivos. A exploração evolui para o recurso a regras, adoptando a representação de alguns conjuntos de dados em listas.

1.1 Evolução e contexto histórico da área de intervenção

A utilização dos computadores no ensino remonta aos anos 60 com os sistemas para a **Instrução Assistida por Computador** ("Com-

puter Assisted Instruction") - IAC. Desde então a área tem evoluido. Os primeiros sistemas tinham poucas capacidades cognitivas e obrigavam o aluno a actuar de forma passiva, isto é, a sua participação resumia-se à selecção de alternativas. O modelo da apresentação não podia ser alterado: o sistema agia da mesma forma com todos os alunos (ensino programado linearmente). Estes sistemas, baseados no paradigma behaviorista (estímulo/resposta), consideram que a aprendizagem se dá através de fenómenos externamente observáveis. Este paradigma evoluiu com a introdução da noção do reforço (recompensa ou castigo) para as respostas dadas pelo aluno [THO 32] e [SKI 36]. Neste sentido, a preocupação central de Skinner foi a de procurar a melhor forma de implementar os reforços a fim de obter do aluno o comportamento desejado.

Com Piaget [PIA 76], surge uma nova proposta em que o conhecimento resultaria de um processo de construção activa do sujeito cognitivo a partir do seu sistema de representação e transformação. Com base nesta ideia tiveram origem aos chamados "ambientes de descoberta" (ambientes de programação). Nestes ambientes o aluno pode construir sua própria simulação, dos quais o mais conhecido é a linguagem de programação LOGO. As principais deficiências deste método estão relacionadas com a necessidade de acompanhamento constante por parte de um agente externo para auxiliar na correcção das falhas, tanto na programação, como da aplicação que o aluno está a desenvolver, e com a necessidade de se criar constantemente os objectos de aprendizagem.

Uma das formas de evolução dos modelos anteriores deu origem aos sistemas de Instrução Assistida por Computador Inteligente ("Intelligent Computer Assisted Instruction") [SEL 74], [HEW 77],[BAR;BEA;ATKI 76], [BAR;FEI 82], [SLE;BRO 82], [OSH;SEL 82], [AND 85], em que a IA desempenha um papel de relevo por permitir não só uma maior flexibilidade, mas também possibilitar a participação activa do aluno e do sistema, gerando um ambiente cooperante para o ensino e para a aprendizagem. Os sistemas IAC inteligentes, que aprendem, são também chamados de Tutores Inteligentes - TI - ("Intelligent Tutor") [GRE 87]. A manifestação de um comportamento inteligente por parte da máquina é defendida pela corrente cognitivista. Assim, para Sleemán e Brown, um tutor inteligente assume a forma de programas que acompanham a escrita de programas, de programas guias, de programas laboratórios de ensino e de programas que aceitam consultas. Ainda para os mesmos autores, um tutor inteligente é um programa de computador que utiliza as técnicas da IA para representar o conhecimento e para conduzir a interacção com o estudante [SLE;BRO 82].

Os TI's, desenvolvidos em linguagens de alto nível, como o Lisp e o Prolog, possuem capacidade para simular a resolução de problemas, para gerar o modelo do aluno (o que possibilita adequar as estratégias de ensino às necessidades de cada aluno) e, ainda, para depurar os programas. Em geral, estes sistemas utilizam o modelo de ensino socrático ([CAR 70], [STE;COL;GOL 82]), que prevê situações de diálogos a partir de um facto conhecido pelo aluno, levando-o a aperfeiçoar este conceito através da

exploração de contradições e da formulação de inferências correctas a partir do conhecimento inicial. Outro modelo de ensino que pode ser adoptado, é aquele em que o tutor conduz o aluno no decorrer do curso (exploração directa do assunto), através da geração de um ambiente interactivo (ver, por exemplo, [BUR;BRO 76], [GOL 82], [MIL 82], [LAN;BRE;FAR 83]). O Tutor-Prolog também recorre a este modelo de ensino.

Para melhor localizarmos as áreas de intervenção e as diferenças entre os sistemas IAC e TI, reproduzimos na figura 1.1 o diagrama apresentado por Kearsley no seu livro "Artificial Intelligence & Instruction" [KEA 87].

Figura 1.1 Domínio dos sistemas TI

Os sistemas TI's procuram explorar as técnicas da IA e tem a

sua base teórica nas Ciências Cognitivas. A estrutura destes sistemas é dinâmica e, frequentemente, são utilizados vários métodos e técnicas na sua organização. Por exemplo, na análise das respostas dos alunos a depuração é feita com base no conhecimento do perito. O modelo do aluno é sempre qualitativo. Por isso os sistemas TI's impõem recursos (programas e máquina) específicos, por causa do uso das técnicas e metodologias da IA.

Assim, tendo em vista o actual desenvolvimento da IA, no que se refere à aprendizagem dos programas, é perfeitamente viável pensarmos num ambiente tutorial computacional para a resolução de problemas de modo interactivo e inteligente. Num tal ambiente é possível a compreensão da intervenção do aluno, aprender a dialogar com ele, orientá-lo, diagnosticar as suas deficiências, e obter informações de forma amigável e coerente.

Como foi apresentado, o desenvolvimento de tutores tem sido, nos últimos anos, alvo de investigação multidisciplinar. Contudo, na área da Programação em Lógica, em particular envolvendo o ensino da linguagem de programação Prolog, existe apenas um produto comercial, o ATP - Active Prolog Tutor. Desde o início da década de 80 foram realizadas experiências no Imperial College [KOW 82], [ENN 81, 81a, 81b] e no Weizmann Institut [SCH;NAL;SHA 87] com a linguagem Prolog, a fim de introduzir os computadores no ensino primário. No caso da linguagem Lisp, para além dos vários tutores desenvolvidos por Anderson e pela sua equipa [AND 1985], existe também um produto comercial, o tutor LISP - The

LISP Intelligent Tutoring System da Advanced Computer Tutoring. No Anexo 2 apresentamos um resumo destes trabalhos.

1.2 Objectivos da investigação

O Tutor-Prolog visa criar um **ambiente para a resolução de problemas** e para isso ensina o paradigma da Programação em Lógica através do uso de um subconjunto da linguagem de programação Prolog direccionado a alunos do ensino secundário. Teve como campo de experimentação e avaliação dos resultados, a escola no. 1 dos Olivais [ZAM 86, 87a, 87b, 87c, 88] que está integrada ao projecto MINERVA (Meios Informáticos no Ensino: Racionalização/Valorização/Actualização), pólo de Coimbra. Com a finalidade de alcançar este objectivo geral foram definidos os seguintes objectivos específicos:

(a) De ensino

Capacitar o Tutor-Prolog a ensinar o aluno a explorar e a especificar problemas. A programação é ensinada através da especificação, que envolve o problema a ser resolvido. Para transcrever um problema em Prolog o aluno necessita dominar a sua especificação. Este aspecto incentiva a melhor exploração dos conhecimentos necessários para a obtenção da solução de um problema proposto. Para isso, são abordados problemas que propiciam

o exercício do raciocínio. O aluno é levado a interagir com programas que envolvem o uso da decisão, da repetição e da recursão, e a representar informações através da utilização de listas, de árvores, de redes semânticas e de enquadramentos.

Ensinar um subconjunto da linguagem Prolog através da comparação com situações da língua natural. Este princípio foi também utilizado nas já citadas experiencias do Imperial College e do Weizmann Institut. O ensino da linguagem Prolog tem situações comparáveis ao ensino de uma língua natural. Numa língua natural e numa linguagem de programação existem regras de escrita a serem observadas. Em ambas o pensamento é organizado de maneira a produzir frases semanticamente correctas. O relacionamento lógico entre as frases permite definir com precisão o universo do problema ao qual se está referindo. Tanto na linguagem de programação Prolog como na língua natural existe uma forma estrutural própria e, para haver boa comunicação, é fundamental que se raciocine de acordo com a estrutura inerente à forma em questão.

Os conteúdos instrucionais apresentados vão ensinar a abstrair as informações necessárias e relevantes, para a obtenção da especificação lógica de um problema, a partir da sua definição em Português. Pretende-se que a partir da semântica declarativa da linguagem Prolog o aluno desenvolva o problema proposto e o entenda, ao invés de apenas memorizar as fórmulas para resolvê-lo.

(b) **De aprendizagem**

Desenvolver mecanismos para a aprendizagem. Com esta capacidade o Tutor-Prolog pode evoluir para adequar-se melhor ao estilo de cada aluno, tornando a interacção mais inteligente, agradável e dinâmica. O Tutor-Prolog aprende a adaptar-se ao aluno através da interacção, isto é, quando o aluno o consulta, quando escreve exemplos e durante a solução de problemas. A adaptação refere-se, principalmente, à linguagem e ao ritmo do aluno no acompanhamento do tutorial.

A aquisição de conhecimento como forma de adequar o Tutor-Prolog ao aluno. O Tutor-Prolog adquire conhecimentos a respeito de seu utilizador, a respeito do uso que este faz do sistema e a respeito do domínio da sua especialidade. Os conhecimentos sobre o aluno permitem ao Tutor-Prolog adequar melhor as suas estratégias de ensino e gerar exemplos de acordo com as necessidades de cada aluno. Os conhecimentos do uso que o aluno faz do Tutor permitem definir o grau de auxílio que o aluno deve receber e ajudam a especificar os diagnósticos relativos às falhas ou às confusões feitas pelo aluno.

(c) **De desenvolvimento**

Organização de um sistema com base em hipóteses. A flexibilidade e a adequação do Tutor-Prolog a cada utilizador

baseia-se na geração de hipóteses, que procuram descrever o que o aluno conhece e o que o Tutor-Prolog supõe que o aluno conhece (modelo do aluno). As hipóteses, geradas durante a interacção, possibilitam o uso de estratégias de ensino, a correcção automática ou orientada de erros ocorridos durante a escrita de exemplos, a condução do diálogo e a aquisição de conhecimentos pelo Tutor-Prolog. A aprendizagem ocorre quando um conjunto de hipóteses é verificado, levando o Tutor-Prolog a alterar as regras existentes, ou a definir novas regras, ou, ainda, a eliminar as regras ou hipóteses antigas.

1.3 Comparação entre o Tutor-Prolog e outros sistemas

Resta ainda fazer uma breve comparação entre o Tutor-Prolog e os trabalhos que mais influenciaram a sua idealização, e que são apresentados no Anexo 2. Serão destacados os aspectos comuns, as diferenças e as adaptações, em virtude das peculiaridades do Tutor-Prolog e dos seus utilizadores (alunos do ensino secundário com idade entre os 14 e os 16 anos).

(a) O Tutor-Prolog e as experiências do Imperial College

Da mesma forma que o trabalho do Imperial College ([KOW 82] e [ENN 81a,81b,81c]), o Tutor-Prolog é uma das formas possíveis para a introdução dos computadores nas escolas. Eles têm em comum

o facto de adoptarem a introdução gradual dos conceitos da linguagem Prolog, que se concretiza na utilização de níveis de complexidade diferentes para o ensino e na divisão didáctica dos conceitos da linguagem em blocos instrucionais, partindo, em ambos os casos, da transcrição de frases escritas em língua natural para o Prolog. O Tutor-Prolog utiliza, igualmente, como motivação, a apresentação de problemas que envolvem um grau de complexidade crescente. Os problemas podem ser, também, propostos pelo aluno que, na busca da solução, deverá conhecer determinados conceitos de programação.

Na nossa experiência foram utilizados todos estes recursos, mas com as seguintes diferenças fundamentais:

- o recurso aos manuais é substituído pelo apoio do tutor inteligente;
- o ensino das estruturas de representação da informação engloba as listas, os enquadramentos e as árvores;
- o ensino da exploração de bases de dados é realizado através da construção de regras;
- a ligação de programas Prolog com outros programas aplicativos (gráficos, editores); e
- a comparação do desempenho dos alunos que desenvolveram um conjunto de actividades com o Tutor-Prolog e de outros que recorreram a programas de outro tipo ("Folha de Cálculo" [ZAM 87]).

Verificámos que a taxonomia proposta pelo Imperial College, nos conteúdos referentes ao ensino dos átomos, variáveis, factos e objectivos Prolog, adaptou-se com sucesso às nossas metas.

(b) O Tutor-Prolog e a experiência do Instituto Weizmann

Os resultados do programa desenvolvido pelo Instituto Weizmann, para o ensino da linguagem Prolog em escolas primárias, são passíveis de serem comparados com os resultados obtidos com o Tutor-Prolog. Segundo o relatório do grupo de trabalho deste Instituto [SCH;MAL;SHA 87], o programa obteve sucesso na introdução do conceito de regras para alunos do ensino primário. Nas experiências por nós realizadas visando a concepção do material instrucional a ser utilizado para as diferentes faixas etárias, o Tutor-Prolog não obteve resultados positivos no tocante às regras e às listas para este nível de escolaridade. Daí, existir no Tutor-Prolog uma regra de alerta para os casos de alunos com idade inferior ou igual a doze (12) anos. Quando tal ocorrer, o curso avança normalmente até ao bloco instrucional relativo aos objectivos e tenta apresentar o conteúdo seguinte (regras). Caso o aluno não responda satisfatoriamente, o curso deve ser encerrado. Os pontos comuns entre as duas experiências dizem respeito à transcrição da língua natural para o Prolog e do princípio de que é necessário criar **ambientes**, diferentes do oferecido pelo interpretador Prolog, para a introdução do ensino da linguagem de programação Prolog.

(c) Geração de modelos cognitivos

No que se refere ao uso de modelos cognitivos, o Tutor-

Prolog é comparado com os tutores desenvolvidos por Anderson e a sua equipa ([AND 85a] e [REI 87]). Assim, a modelagem do aluno, que realizamos no Tutor-Prolog, visa a geração do diagnóstico e a correcção automática ou orientada dos erros e desvios cometidos pelo aluno durante a interacção. Serve também para definir a estratégia de ensino a ser utilizada em cada instante. As possíveis situações de erro referem-se à sintaxe, à semântica e à pragmática da linguagem de programação Prolog, e algumas delas são descritas no Capítulo 4 deste trabalho. A forma da definição e utilização das estratégias de ensino no Tutor-Prolog é a nossa contribuição para as investigações em áreas como a construção de modelos e a utilização de estratégias pedagógicas.

Para além das regras e dos conhecimentos que representam o modelo do aluno, no modelo ideal de ensino existem planos que descrevem problemas, como nos tutores desenvolvidos por Anderson. No entanto, o auxílio para a construção dos programas descritos nos planos é fornecido de acordo com os conhecimentos que vão sendo adquiridos dinamicamente durante a interacção com o aluno.

Nos planos, representados através de enquadramentos, encontram-se informações (referentes à ordem e ao número de argumentos das cláusulas) que permitem ao Tutor-Prolog fornecer uma análise detalhada ao aluno.

Se as regras escritas no programa do aluno estiverem correctas, isto é, de acordo com um dos planos existentes, o Tutor-Prolog não interfere. Se, no entanto, as regras estiverem

em desacordo, isto é, se o código gerado pelo programa do aluno não corresponder ao código esperado pelo Tutor-Prolog, este intervém para explicar o erro. A intervenção só é imediata nos casos de erros de natureza sintáctica e semântica declarativa. Nos demais casos (semântica operacional e pragmática), o Tutor-Prolog só intervem quando não consegue acompanhar ou representar o raciocínio do aluno. Nestes casos optamos por uma estratégia oposta a de Anderson, pois o Tutor-Prolog procura acompanhar o raciocínio do aluno e só intervém quando tem conhecimentos que lhe permitam a reorientação do aluno ou a correcção do erro. Esta estratégia também é adoptada na interacção que envolve a utilização da língua natural escrita.

Se o aluno desistir da busca da solução correcta para o problema, o Tutor-Prolog expõe o passo correcto ou invoca o módulo de planeamento que demonstra como o algorítmo deveria ter sido programado. Este modelo é semelhante ao usado no tutor para o ensino da Geometria, desenvolvido por Anderson [AND 1985a]. Assim, as estratégias utilizadas nos tutores construídos por Anderson podem ser caracterizadas como processos inteligentes que monitoram as actividades do aluno. Ou seja, o tutor ajuda o estudante a representar as estruturas mentais do seu raciocínio.

Os tutores de Anderson, por outro lado, são capazes de gerar mensagens tutoriais de acordo com uma sequência particular de aplicações de regras. Esta capacidade também é largamente utilizada no Tutor-Prolog.

(d)Diagnóstico de erros em programas

Abordamos no Tutor-Prolog a correcção orientada (método que tem semelhanças com o utilizado por Shapiro [SHA 82] e [STE;SHA 87]) ou automática de falhas de natureza sintáctica, semântica (declarativa e operacional) e pragmática, que ocorrem em programas de alunos iniciados. A principal diferença em relação ao método de Shapiro é que o processo de correcção e orientação é sempre feito de acordo com o passado da interacção. Logo, o Tutor-Prolog aprende com o decorrer da interacção e faz ilações com o conhecimento adquirido em função do contexto geral da interacção, visando emitir o melhor diagnóstico possível para a falha detectada.

Utilizamos, ainda, o retrocesso inteligente para consultas que não utilizem construções que envolvam o uso de variáveis livres. A implementação desta característica segue a metodologia apresentada por [PER;POR 79]. Todo o nosso trabalho é voltado, particularmente, para a geração e a consulta de pequenas bases de dados criadas pelo aluno. A definição deste universo de trabalho é importante para a compreensão da semântica operacional da linguagem Prolog.

As investigações práticas em contexto escolar demonstraram que, para a população a que nos dirigimos, as estratégias como as utilizadas por Shapiro [SHA 82] não eram adequadas e/ou não surtiam efeito. Ou seja, os alunos das escolas secundárias neces-

sitam de diagnósticos mais detalhados, apoiados em demonstrações gráficas (visuais), controlados pelo Tutor-Prolog, e que dependam minimamente do aluno. Assim, as estratégias semelhantes às utilizadas por Brayshaw e Eisenstadt [BRA;EIS 88], apoiadas em demonstrações gráficas, mostraram-se mais eficientes. Assim, o trabalho de Shapiro serviu como ponto de referência para a implementação (escrever programas meta-interpretadores claros e eficientes). Consideramo-lo importante, especialmente para futuras versões do Tutor-Prolog, no campo da detecção de falhas em programas recursivos.

Para o diagnóstico de falhas sintáctico-semânticas utilizámos mensagens que, associadas às regras e ao conhecimento que o Tutor-Prolog adquiriu sobre a interacção, geram o diagnóstico apropriado a cada erro. A adaptação do diagnóstico e da correcção do erro ao contexto da utilização foi um dos aspectos da depuração investigado. Para a orientação do aluno na escrita dos programas propostos pelo Tutor-Prolog são utilizados planos que descrevem estes programas, como no sistema PROUST [SOL 83] e [JOH;LEW;SOL 85], só que os planos existentes no tutor estão representados em enquadramentos.

(e) Construção da interface - língua natural

O módulo de interface em língua natural, desenvolvido no Tutor-Prolog, foi contruído de forma a que as regras sintácticas

e o dicionário linguístico que o compõe pudessem evoluir a partir da interacção com o aluno. A evolução visa a adaptação da sintaxe e do vocabulário, existentes no núcleo inicial do Tutor, ao estilo e aos conhecimentos do aluno. Para isso, foram estudados sistemas como o T.E.L.I. ([BAL;STU 84] e [BAL 86]) e o T.E.A.M. ([GRO;APP;MAR;PER 87]).

No sistema T.E.L.I. o conhecimento linguístico, e o da própria estrutura interna do sistema, é bastante maior do que o solicitado pelo T.E.A.M.. Por outro lado, nota-se que no T.E.L.I. o conhecimento está mais acessível a quem com ele comunica (ver Anexo 2). No Tutor-Prolog há a preocupação de seguir a linha adoptada no T.E.A.M. no que se refere aos conhecimentos exigidos ao utilizador e de permitir, tal como no T.E.L.I., que o conhecimento adquirido pelo sistema esteja o mais acessível possível ao utilizador.

No que se refere á aquisição de conhecimento, uma vez que não iremos efectuar um tratamento no nível semântico, o processo de aquisição realizado no Tutor-Prolog é bastante mais simples do que o efectuado no T.E.A.M., e diferente do seguido no T.E.L.I.. Adoptámos o ponto de vista segundo o qual a presença de um vocábulo desconhecido não representa apenas uma nova palavra pertencente a uma estrutura fraseológica já conhecida (como é feito no T.E.L.I.) mas também pode implicar a aprendizagem de toda uma nova estrutura fraseológica. Ou seja, a divisão bastante marcada que existe no T.E.L.I. não se verifica no Tutor-Prolog,

em parte devido ao tipo de utilizadores a que se destina, isto é, utilizadores não especializados e muito pouco familiarizados com conceitos linguísticos e ferramentas computacionais.

Como característica comum entre todos os sistemas, note-se a divisão lógica existente entre as diferentes componentes do discurso. Assim cada componente apenas pode ser de classe "aberta" ou "fechada", só sendo permitida a aprendizagem dos vocábulos pertencentes a componentes de "classe aberta". No Tutor-Prolog cada componente é ainda caracterizado pelo facto de poder, ou não, ser **coberto pelo vazio**, ou seja, ser considerado como componente opcional numa estrutura fraseológica.

Note-se ainda que, nem no T.E.A.M., nem no T.E.L.I., são oferecidas facilidades de detecção e correcção de erros ortográficos, as quais estão presentes no Tutor-Prolog porque são necessárias para uma interacção amigável com os alunos.

Ainda, é importante refir que a viabilidade prática de sistemas com as características do T.E.A.M. e do T.E.L.I. motivou o desenvolvimento do estudo que é descrito no Capítulo 5.

(f) Reescrita de regras

Na reescrita das regras, utilizámos aspectos comuns à geração automática de programas [GAR;VER 84a,84b,84c]. Para reescrever uma regra, o Tutor-Prolog faz abstracções utilizando o

seu conhecimento interno e o conhecimento externo fornecido pelo aluno. Assim, partindo de uma frase escrita em Português, que não possa ser interpretada directamente pelas regras actuais definidas no interpretador, a capacidade de aprendizagem permite que a confrontação das informações sintácticas contidas na frase em análise com as regras existentes, gere um novo modelo, mais abrangente, através da reescrita pré e pós-fixada das regras iniciais.

Como foi possivel observar, a construção do Tutor-Prolog envolveu a utilização e a interligação de diferentes técnicas da IA, nomeadamente: representação do conhecimento, condução da interacção para o ensino, condução de interacções de aprendizagem e capacidades de adaptação do sistema ao utilizador (sensibilidade do sistema ao desempenho do aluno). Envolveu, também, conhecimentos de informática, nomeadamente: engenharia de "software" (construção de meta-interpretadores), sistemas operativos (simulação e tratamento de excepções) e estruturas de representação da informação (listas, matrizes, árvores, pilhas e enquadramentos).

1.4 Organização do texto

No Capítulo 2 são apresentados os conceitos que fundamentam o trabalho, a arquitectura e o resumo dos principais módulos que compõem o Tutor-Prolog.

No Capítulo 3 são tratados a geração do modelo do ensino, o processo de comparação entre o modelo de ensino do Tutor-Prolog e as necessidades do aluno, aspectos que envolvem o uso de estratégias de ensino, e o conhecimento pedagógico.

No Capítulo 4 é apresentado o especialista Prolog. São tratados aspectos da meta-interpretação da linguagem, envolvendo a análise sintáctica, a semântica (declarativa e operacional), a pragmática e a simulação das hipóteses de correcção dos erros ocorridos no programa do aluno e na respectiva execução.

No Capítulo 5 são tratados aspectos da interface, no que se refere à comunicação com o aluno através da língua natural escrita.

No Capítulo 6 é apresentado o trabalho futuro. Mais especificamente, os caminhos para o alargamento do Tutor-Prolog visando o ensino profissional e a sua transformação num produto comercial.

No Capítulo 7 apresentamos as nossas conclusões a respeito da experiência adquirida no desenvolvimento de Tutores Inteligentes para o ensino da Programação em Lógica e em particular de um subconjunto da linguagem de programação Prolog.

No Anexo 1 são apresentados os resultados das observações práticas efectuadas com os protótipos.

No Anexo 2 são apresentadas as experiências afins que

influenciaram o nosso trabalho nas áreas do ensino e da aprendizagem, no desenvolvimento prático e na interface com o utilizador.

E, finalmente, no Anexo 3 encontra-se a listagem dos programas correspondentes à última implementação do protótipo.

2. DESCRIÇÃO GERAL DO TUTOR-PROLOG

Neste Capítulo é apresentada a arquitectura do Tutor-Prolog e, em particular, os aspectos da sua exploração envolvendo o processo de comunicação no ambiente do Tutor, o processo geral de formação e utilização do modelo do aluno, e uma síntese de como está organizado o processo de ensino. É também apresentado o fluxo de informações e de controle que ocorre entre os módulos do Tutor-Prolog intimamente relacionados com o ambiente de ensino.

O Tutor-Prolog ensina através de exemplos ([WIN 77], [MIC;CHI 80], [LEN 76], [DEI;MIC 83], [MIC;KO;CHE 87]) e aprende[1] através da experiência[2] representada nos modelos do aluno e da interacção. O processo de aprendizagem é, em muitos casos híbrido, pois além da experiência são utilizados exemplos[3], os quais

---

1 Assumiremos o conceito adoptado por Herbert Simon [SIM 86], para quem a aprendizagem é todo o processo através do qual um sistema melhora o seu desempenho. Esta melhoria pode ser alcançada, quer aplicando novos métodos, quer melhorando os existentes, não podendo, contudo, ser levada a cabo sem que se verifique alguma alteração no sistema.

2 Aprendizagem por experiência, ou seja, construção de descrições, hipóteses ou teorias acerca de uma determinada colecção de factos ou experiências ocorridas durante a interacção.

3 Inferir uma descrição geral de um conceito a partir de exemplos ou de contra-exemplos desse conceito: forma de aprendizagem indutiva.

são apresentados simultaneamente em Português e na linguagem de programação Prolog. Para a exploração da representação directa do significado dum texto escrito em Português, num programa Prolog, a semântica declarativa da linguagem Prolog foi bastante utilizada na geração das lições.

Os modelos são utilizados para a realização do diagnóstico cognitivo. O **modelo ideal** de ensino está organizado segundo uma hierarquia ([GAG 67], [RES 73], [RES;WAN;KAP 73]) pré-determinada e corresponde ao que se pretende seja o conteúdo que um aluno do ensino secundário deva adquirir. Quando ocorre um erro, o Tutor-Prolog apresenta uma sequência instrucional alternativa de acordo com a hierarquia prevista para o caso ("overlay model") [CAR;GOL 77]. Contudo, a hierarquia existente é sensível ao desempenho do aluno e os reforços instrucionais são apresentados até o aluno responder satisfatoriamente ao sistema (de acordo com o modelo ideal) ou até o tutor gerar hipóteses a respeito do aluno, que lhe permitam adequar o modelo ideal às necessidades do aluno (**modelo do aluno**). A capacidade de adaptação é a característica fundamental do Tutor-Prolog que procura não só aproximar os conhecimentos do aluno aos conhecimentos do seu modelo ideal mas, em muitos casos, procura também aproximar os conhecimentos do Tutor aos conhecimentos do aluno através da geração do modelo da interacção.

A capacidade de **adaptação** encontra-se ligada a vários conceitos de **inteligência**, citamos por exemplo, Wechsler para

quem a inteligência é "a capaciade agregada ou global de um indivíduo agir adequadamente, de pensar racionalmente, e de se adaptar eficientemente ao ambiente" e Piaget [PIA 54] que define a inteligência como "a capacidade de adaptação ao ambiente físico e social".

Durante a interacção entre o aluno e o meta-interpretador Prolog (ver Capítulo 4), quando os erros envolvem a sintaxe, a semântica e a pragmática da linguagem Prolog, é estabelecido um diálogo enriquecido pelo processo de orientação e correcção dinâmica de erros. Isto é, o Tutor-Prolog aprende a respeito do programa que o aluno está escrevendo e assim pode particularizar a orientação e a correcção dos erros semânticos e pragmáticos que ocorrem durante a escrita dos programas. O processo de adptação é conduzido através da geração de hipóteses sobre os conhecimentos do aluno, o seu desempenho e as suas acções. As hipóteses permitem conduzir o ensino ao longo de diferentes níveis de complexidade, de acordo com os conhecimentos que vão sendo adquiridos durante a interacção (ver Capítulo 3).

No que se refere à interacção através da utilização da língua natural escrita, a forma de aprendizagem que foi testada neste protótipo está dependente do aluno e deve orientá-lo na inclusão de novos sintagmas ou de variantes estruturais. Verificámos que os menus, que são activados pelo interpretador gramatical, e que contenham exemplos com possíveis situações para a acomodação das novas palavras às regras, facilitam e orientam

mais adequadamente o aluno. Os menus são seleccionados de acordo com as hipóteses que o tutor vai gerando para a análise de cada frase. As hipóteses do Tutor são geradas a partir do conhecimento e das regras disponíveis em cada interacção, enquanto que as do aluno dependem do seu conhecimento linguístico. Portanto, a aprendizagem resulta da análise destes dois conjuntos de hipóteses.

Por último, o modelo da simulação[4] da execução dos problemas [NEW;SIM 72] é utilizado na apresentação dos exercícios previstos em cada módulo instrucional (ver Capítulo 4).

Deste modo, procurámos desenvolver um tutor que possuisse capacidades para o ensino e a depuração de programas Prolog, que aprendesse sobre determinadas áreas da sua intervenção e que percebesse o desempenho do aluno[5].

O Tutor-Prolog pode ser explorado ao longo de duas direcções principais:

---

4 O termo simulação é utilizado para designar o comportamento do Tutor-Prolog durante a execução dos programas.

5 Consideramos que tudo aquilo que é do nosso conhecimento é o resultado das nossas interacções com o meio-ambiente. Assim, podemos dizer que o objectivo da aprendizagem é captar ou extrair informações do ambiente e organizá-las de maneira apropriada.

a) O **ensino** - onde o utilizador aprende não só conceitos básicos da programação em lógica, mas também como armazenar e explorar informações recorrendo à linguagem PROLOG. Os conceitos de programação lógica e o subconjunto da linguagem Prolog contidos no Tutor são apenas os necessários para a introdução do uso dos computadores ao nível do ensino secundário.

Nesta direcção o Tutor-Prolog encoraja o aluno a gerar uma solução para cada problema. Isso envolve a resposta a erros, a sugestões e a ensinamentos para a clarificação dos motivos que provocaram os erros ocorridos.

b) **A depuração** - onde o utilizador escreve sua própria base de dados e os seus programas para a exploração destes dados. Neste módulo são utilizadas técnicas para a correcção automática de erros sintácticos, para a detecção e a orientação na correcção de erros semânticos, para a explicação do processo de execução dos comandos escritos pelo utilizador e para a orientação do aluno em aspectos da pragmática da linguagem de programação Prolog.

Nesta segunda direcção o aluno desenvolve a sua própria aplicação. O Tutor-Prolog somente intervém na actividade do aluno quando ocorrem falhas (sintácticas, semânticas ou pragmáticas), ou seja, na sua depuração. No processo de escrita de programas é o aluno quem detém o controle da interacção, passando o Tutor-Prolog a agir de forma menos tutorial. Os erros sintácticos são, na maior parte, corrigidos automaticamente mediante a simples apresentação da mensagem de alerta. Já, no que se refere à interpretação semântica e pragmática, o processo é mais lento e envolve directamente a participação do aluno. Ou seja, o Tutor-Prolog analisa e interpreta programas, simula execuções, apresenta diagnósticos que visam ajudar na depuração dos programas, executa

correcções, com intervenção directa ou não do aluno, e emite mensagens de alerta para possíveis confusões que o aluno possa cometer durante a escrita dos seus programas.

Os exemplos de programas propostos e programados pelo aluno, durante as observações práticas, são apresentados no Anexo 1.

Para além dessas duas direcções principais de exploração, existe ainda, a possibilidade de consulta à bases de dados através de perguntas escritas em língua natural.

2.1 Arquitectura

A arquitectura do Tutor-Prolog que foi adoptada inspira-se na organização tradicional [SEL 74], pois inclui o **modelo do domínio**, o **modelo do aluno** e as **estratégias de ensino**. A reflexão sobre esta organização, de acordo com a aplicação a que cada tutor se destina, torna natural outras alterações. E, o Tutor-Prolog não fugiu à regra. Assim, foram acrescentados novos componentes externos aos módulos centrais, pois procurámos transformar este Tutor num ambiente cooperante com o meio ambiente[6]. No Capítulo 6 sobre o trabalho futuro voltaremos a este assunto para defendermos uma nova arquitectura, a qual é o resultado da nossa investigação.

---

6 Por meio-ambiente entendemos o aluno, os programas aplicativos em geral e o sistema operativo.

Nas arquitecturas de outros TI's como, por exemplo, a dos sistemas NEOMYCIN e GUIDON, Clancey utilizou, para além dos três componentes já citados, uma base de dados, um interpretador e um bloco para as estratégias de ensino [CLA 82]. Já, para Park, Perez e Seidel, uma arquitectura geral para sistemas ICAI deve contemplar os seguintes módulos: as regras de ensino, a base de conhecimentos, o modelo do aluno, o material instrucional, a solução do perito e as regras para o diagnóstico [PAR;PER;SEI 87]. Para Nawrocki os componentes de um sistema ICAI são: o domínio do perito, o modelo do aluno, o comparador dos modelos, o controle do ensino, o conhecimento genérico e o conhecimento específico do perito, a interface em língua natural e as estratégias de ensino [NAW 87]. Para Harmon, a arquitectura de um sistema ICAI deve contemplar o conhecimento do perito, o conhecimento sobre o domínio de aplicação, o modelo do aluno e um bloco responsável pela integração do sistema [HAR 87].

Como podemos analisar, as especificações podem ser mais ou menos detalhadas, mas existe em todas as arquitecturas, um núcleo **básico** (miolo) constituído pelo modelo do aluno, pelo perito e pelas estratégias de ensino. Assim, cada autor tenta acrescentar novos módulos, como por exemplo Wenger [WEN 87] com o modelo da interface. Em nossa opinião, este novo módulo enriquece o núcleo básico da arquitectura de um TI, e por isso dedicámos o Cápitulo 5 desta tese à discussão das nossas experiências nesta área. Como investigámos apenas alguns aspectos envolvidos na geração do modelo da interface, não explicitámos tal componente na arquitec-

tura do Tutor-Prolog, isto é, optámos pela manutenção do módulo denominado "Interface", ao contrário do que acontece com o "Perito Prolog" que é desmembrado em vários módulos. Assim, o Tutor-Prolog está organizado como se apresenta na figura 2.1.

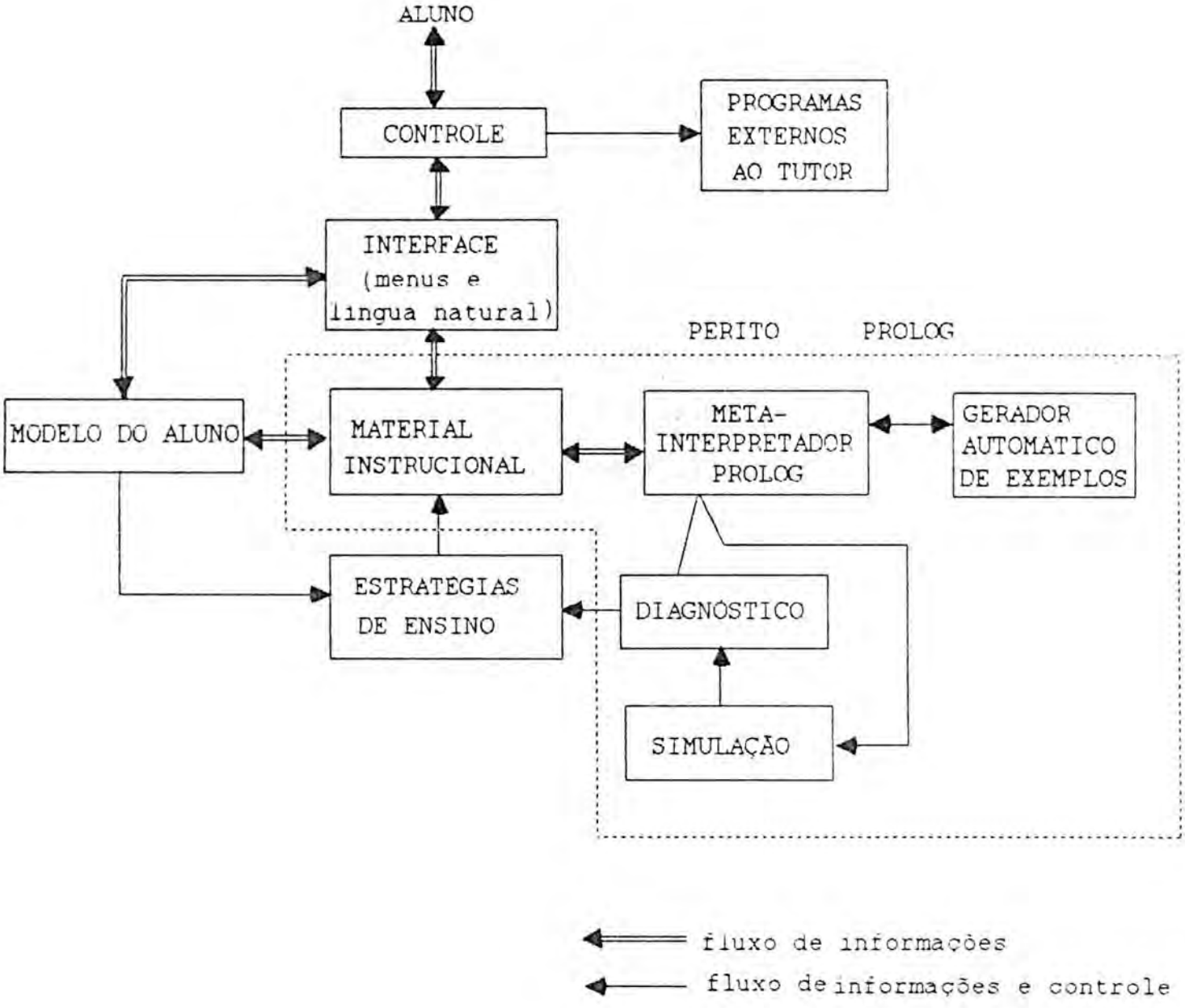


Figura 2.1 Arquitectura Geral do Tutor-Prolog

O dimensionamento e a configuração funcional do sistema depende da planificação do tutorial, que é feita dinamicamente para cada utilizador. A modularização e a troca de conhecimentos entre os vários módulos criam condições para que o sistema possa ser executado sem grandes prejuízos devido a restrições de memória e/ou de desempenho. Para isto, são utilizados algoritmos de controle, possibilidades de utilização dinâmica da memória e mecanismos de reaproveitamento do espaço de memória.

No Tutor-Prolog estão definidos três ambientes de trabalho que defendemos serem necessários para o ensino da programação em Lógica (figura 2.2). O aluno tem, ainda, acesso a programas exteriores ao ambiente tutorial e ao interpretador Arity/Prolog.

Figura 2.2 Ambientes para a programação no Tutor-Prolog

As limitações de memória e de processador não permitem que os três ambientes do Tutor-Prolog estejam activos simultaneamente no ambiente computacional seleccionado. Através do programa de controle, que faz a gestão das informações que são globais ao Tutor-Prolog, é possível a adequação e a delimitação dos módulos necessário para cada utilizador, como é apresentado na figura 2.3.

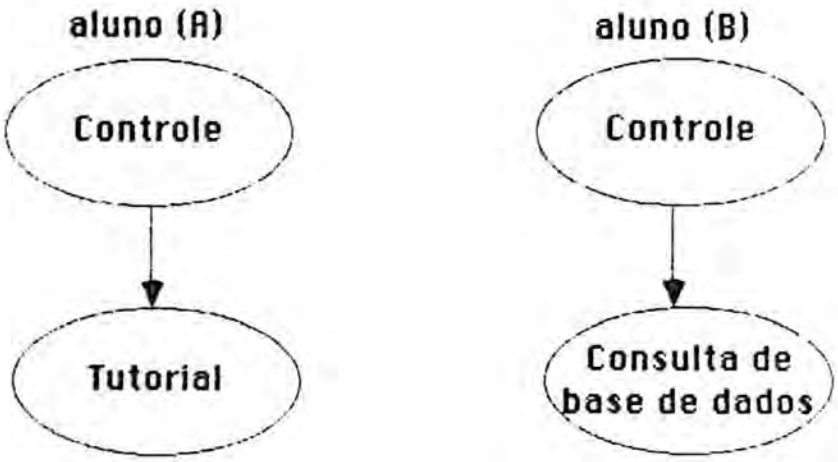


Figura 2.3 Ambiente adequado às necessidades de cada aluno

A passagem de um ambiente para o outro é executada pelo Tutor-Prolog de acordo com o desempenho do aluno ou é por este ordenada. A construção dos módulos foi bastante dinamizada com o uso do conceito de "mundos" da Arity Prolog, os quais servem para separar o código dos dados, isto é, são bases de dados que contém o código ou os dados.

Toda a interacção é activada pelo módulo de controle e

recebida ou apresentada através da interface. A interacção tutor-aluno é um dos aspectos mais importantes da investigação, pois o êxito de um programa tutorial depende grandemente da forma como este interage com o aluno. O aluno constrói exemplos em Prolog de acordo com a orientação apresentada no material instrucional (lições e exercícios). Os exemplos são analisados pelo perito Prolog, que fornece ao aluno o diagnóstico e/ou a simulação resultante do processo de interpretação do exemplo. Em função da análise e do diagnóstico obtido, são geradas hipóteses sobre o conhecimento que o aluno tem a respeito do assunto apresentado. Estas hipóteses, juntamente com a história do comportamento do aluno, definem a estratégia de ensino que melhor se adequa para dar seguimento à interacção (ensino ou aprendizagem).

No perito Prolog estão incluídos os módulos relacionados como o ensino da linguagem de programação, a sintaxe, a semântica (declarativa e operacional) e a pragmática; os exemplos e os exercícios sobre as figuras básicas de programação, como o controle (decisão, repetição e recursão) e as estruturas de representação da informação (cadeias, listas, árvores, regras de produção, e enquadramentos).

No caso da linguagem, estes conhecimentos estão representados na forma de regras DCG, de tabelas (de variáveis, de predicados/número de argumentos e de instâncias das variáveis) que são geradas durante a interpretação interactiva. Os exercícios estão representados através de enquadramentos. Além das

regras DCG, das tabelas e dos enquadramentos, existem ainda, as hipóteses que, neste caso, procuram detectar o motivo das falhas encontradas nos programas. As hipóteses são geradas a partir das regras (de ensino, da linguagem Prolog ou do subconjunto da língua natural interpretado no Tutor), das informações contidas nas tabelas do meta-interpretador e das informações que representam o modelo da utilização.

O **material instrucional** está organizado em seis blocos (árvores) que contêm ensinamentos sobre a Programação em Lógica e, em particular, sobre a linguagem de programação Prolog.

O processo de **diagnóstico** encontra-se, do ponto de vista da implementação, distribuído dentro dos vários módulos que compõem o Tutor-Prolog. O objectivo é orientar o aluno na correcção das falhas e/ou demonstrar qual é a melhor maneira de resolver um problema, de acordo com o conhecimento que o Tutor-Prolog possui em cada instante da interacção (ver Capítulo 4). O processo de aprendizagem contribui para o aperfeiçoamento e a especificação dos diagnósticos.

As **estratégias de ensino** corespondem a planos que permitem apresentar o material instrucional em três níveis de complexidade diferentes[8] [VIC;COE;COS 88]. A opção pela utilização de um

---

8 Os níveis de complexidade utilizados na apresentação do material instrucional, de acordo com as estratégias de ensino, são os seguintes: o modelo ideal (desempenho médio), o modelo detalhado (desempenho fraco) e o modelo avançado (desempenho bom).

plano depende do modelo que o Tutor-Prolog constrói para cada aluno, com base em conhecimentos estáticos e dinâmicos.

2.2 Controle do Sistema Tutor-Prolog

O módulo de controle é o responsável pela gestão da interface (menus e língua natural), pela troca de informações entre os diversos módulos que compõem o sistema e pela comunicação entre o Tutor-Prolog e outros programas aplicativos (Programas Externos ao Tutor-Prolog).

2.2.1 Comunicação com os programas utilitários

O Tutor-Prolog pode comunicar com outros programas, como, por exemplo, programas gráficos, editores de texto e outros interpretadores. A gestão destas operações de comunicação é feita através de um ficheiro tipo MS/DOS "BAT"[9] (ver Anexo 3). E este programa que controla a passagem de ficheiros de um ambiente para o outro, a activação, a paragem da execução dos programas e o fornecimento dos resultados. Estes aspectos são importantes se levarmos em conta a faixa etária dos estudantes a que o Tutor se destina.

---

9 Esta comunicação poderá tornar-se mais eficiênte se for utilizado um sistema operativo com possibilidades para a execução de múltiplas tarefas. Por exemplo, o sistema operativo AWARD DOS (MS DOS compatível).

2.2.2 Comunicação entre os módulos do Tutor-Prolog

O Tutor-Prolog realiza a troca de conhecimentos entre os vários módulos instrucionais, libertando o aluno de actividades paralelas, tais como guardar ou ler ficheiros [VIC 85]. Tudo o que se passa durante a interacção é transmitido automaticamente de um módulo para o outro e de uma sessão de trabalho para a outra. Isto é, o Tutor-Prolog grava a história e as alterações feitas durante uma sessão num ficheiro (modelo do aluno e modelo da interacção), as quais são informações globais para o sistema (área de memória comum a todo o sistema tutorial chamada "mundo PRINCIPAL"). Assim, se numa mesma sessão o aluno percorrer vários blocos instrucionais[10] (cada bloco corresponde a um mundo), o processo de activar um mundo e desactivar outro, não interfere nas informações de carácter global. Logo, durante uma sessão encontram-se activos no mínimo dois mundos, o principal e um temporário.

2.2.3 Comunicação com o aluno

Num ambiente interactivo entre um agente inteligente (aluno) e outro, que se pretende inteligente (tutor), a situação dos

---

10 Cada bloco instrucional corresponde a um conjunto de lições sobre um assunto no domínio do ensino.

diálogos deixa de ser a tradicional, em que apenas o agente humano tinha condições de mudar o tópico do diálogo, limitando-se o agente máquina a procurar acompanhar "inteligentemente" o discurso. No caso da interacção aluno/Tutor-Prolog ambos podem alterar o tópico do diálogo, ou seja, uma abordagem de iniciativa mista [COS 86a]. Os desvios podem ocorrer nas seguintes situações quando o aluno discorda das opiniões do Tutor-Prolog e passa a conduzir de forma diferente (de acordo com as suas hipóteses) o plano de ensino; quando o aluno resolve trocar de contexto, isto é, passar, por exemplo, do módulo de ensino, para a consulta a base de dados, ou mesmo para o interpretador Prolog; quando o Tutor-Prolog percebe que o aluno tem dificuldades em compreender determinada lição ou exercício e opta por retroceder para lições pertencentes a blocos instrucionais diferentes daquele que está em uso; e, quando o Tutor-Prolog necessita adquirir conhecimentos directamente do aluno.

Resumindo, podemos ter as seguintes situações de diálogos durante uma interacção:

- o aluno pergunta e o Tutor-Prolog responde;
- o Tutor-Prolog pergunta e o aluno responde;
- o Tutor-Prolog ensina o aluno; e
- o aluno ensina o Tutor-Prolog.

Todas estas situações são apresentadas no decorrer deste trabalho que tem como um dos seus objectivos testar formas de comunicação entre o Tutor-Prolog e o aluno. Assim, a comunicação é feita através de menus, símbolos, códigos e frases escritas em

Português. O processo de comunicação é dinamizado e organizado através do uso de cores - por exemplo, as hipóteses do Tutor-Prolog são escritas sempre em verde - destaques e divisão da tela em janelas. O objectivo é verificar se os alunos apresentam algumas preferências e se estas se mantêm, apesar dos diferentes graus de instrução ou de desempenho no tutorial.

Os menus colocam à disposição do aluno as operações que podem ser realizadas num determinado domínio. Toda a comunicação realizada entre o utilizador e o sistema é apoiada em menus, inclusive nas situações em que o aluno escreve suas frases em língua natural. A apresentação dos menus pode ser activada para iniciar um diálogo ou prosseguí-lo, ou devido à detecção de um "conflito", isto é, um contexto desconhecido pelo Tutor-Prolog.

O menu principal serve para orientar o aluno sobre as opções de trabalho, ou seja, o módulo de ensino, o módulo de auxílio na escrita de programas, o interpretador Prolog e o módulo de consulta a base de dados. A sua configuração é apresentada na figura 2.4.

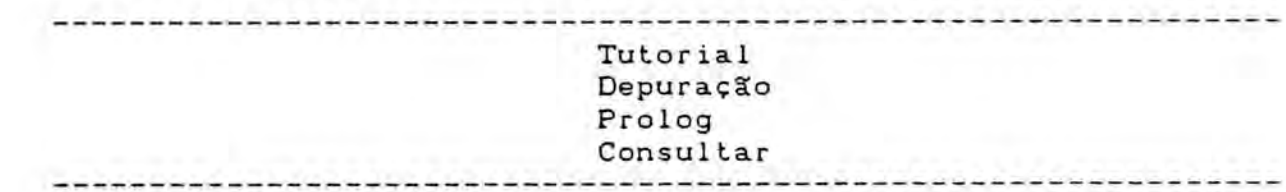


Figura 2.4 Níveis de trabalho no Tutor-Prolog

O plano de ensino gerado pelo Tutor-Prolog pode corresponder ao modelo do Tutor ou ter sido gerada especificamente para o

aluno com base nos conhecimentos existentes no modelo do aluno. O menu contendo o plano tutorial é apresentado no início de cada sessão ou antes do início da apresentação de um novo bloco instrucional, como por exemplo na figura 2.5.

```
------------------------------------------------------------
                          átomos
                          variáveis
                       >> factos
                          objectivos
                          regras
                          listas
------------------------------------------------------------
```

Figura 2.5 Plano de ensino proposto pelo Tutor-Prolog

Através dos menus o aluno pode optar também por um desenvolvimento do tutorial diferente do plano estabelecido pelo Tutor-Prolog.

O Tutor-Prolog adquire conhecimentos de duas formas: a interna e a externa. A forma interna é apresentada no Capítulo 3, onde é tratada a geração de hipóteses de ensino, que procuram verificar os conhecimentos, dominados ou adquiridos pelo aluno sobre a linguagem de programação Prolog. E, com base nas hipóteses e conhecimentos, o Tutor-Prolog conduz o processo de ensino. A forma externa de aquisição de conhecimentos relaciona-se com o conhecimento linguístico que o Tutor-Prolog necessita adquirir, para interpretar as consultas escritas pelo aluno. A forma externa é discutida no Capítulo 5, que trata da geração de hipóteses a respeito de vocábulos ou de construções sintácticas que o Tutor desconhece. Nesse caso, o Tutor solicita a ajuda do

aluno para confirmar ou abandonar as suas hipóteses.

A interacção entre o aluno e o Tutor-Prolog dá-se do seguinte modo. O Tutor-Prolog possui um conjunto de regras Rt (modelo do tutor), que descrevem determinado conhecimento. Durante a interacção, Rt é activado para interpretar a descrição de um problema escrito pelo aluno Da. Nesta interpretação é produzido um conjunto de hipóteses Rta, que representam as diferenças (conflitos) existentes entre o modelo do Tutor-Prolog Rt e o modelo do aluno Da. Caso o Tutor-Prolog consiga verificar a coerência de Rta, o conjunto de regras Rt deverá ser reescrito para passar a representar a situação Rt e a situação Rta. Se, no entanto, Rta não for verificado, as hipóteses que o compõem darão origem a novas hipóteses, que representarão o desconhecimento do aluno sobre o problema, ou a uma acção especifica do Tutor-Prolog para aproximar o conhecimento do aluno ao seu. Assim, o Tutor-Prolog adequa-se ao estilo do aluno (aprende), ou usa as hipóteses geradas para planear o próximo passo tutorial. Isto é, procura aproximar o modelo real do aluno ao modelo do Tutor-Prolog (ensina).

Pode acontecer que Rta esteja correcto e o Tutor-Prolog não possua os conhecimentos suficientes para verificar a coerência. Perante este caso, o Tutor-Prolog procura adquirir conhecimento externo que lhe permita classificar Rta. Caso a aquisição não se concretize, o Tutor-Prolog cria uma categoria "desconhecida" onde são guardados os novos conhecimentos apresentados.

O processo de "evolução" do Tutor-Prolog através da sua adaptação ao aluno pode ser resumido nos seguintes passos:

- determinação da zona da regra onde ocorreu o conflito;
- geração de hipóteses para a reescrita da zona de conflito;
- teste das hipóteses com o auxílio do aluno;
- nova designação da zona de conflito;
- substituição, na regra inicial, da zona de conflito pela sua nova designação;
- escrita das novas regras, tendo como cabeça a designação da zona de conflito e como corpo a zona de conflito ou as hipóteses geradas visando eliminar o impasse;

O processo de ensino do Tutor-Prolog, através do qual o conhecimento do aluno se adapta ao conhecimento do Tutor, pode ser resumido nos seguintes passos:

- determinação da zona da cláusula onde ocorreu o conflito;
- geração das hipóteses para a reescrita da zona de conflito com base nas regras da linguagem Prolog e nos conhecimentos adquiridos durante a interacção;
- teste das hipóteses com o auxílio do aluno;
- substituição, na cláusula inicial, da zona de conflito pela nova composição;
- reinterpretação da cláusula.

2.3 Processo de ensino

O processo de ensino do Tutor-Prolog está organizado em redor do material instrucional e das acções que o aluno realiza durante a interacção. O material instrucional é composto por explicações, exemplos, diagnósticos, simulações, hipóteses e conhecimentos. Este material pode ser apresentado ao aluno de maneiras distintas. A cada problema estão ligadas explicações alternativas, que podem ser solicitadas ou activadas por iniciativa do Tutor-Prolog, no caso de vários erros sucessivos cometidos pelo aluno envolvendo um determinado conteúdo.

O Tutor-Prolog possui dois tipos de ensino: o genérico e o individualizado. No genérico estão os ensinamentos básicos necessários para se ensinar o subconjunto da linguagem de programação Prolog, as estruturas de representação da informação e as figuras de programação. A definição destes conteúdos está distribuída em lições divididas entre os vários módulos instrucionais e ligadas a exemplos. Estas lições estão organizadas numa taxonomia inicial, de acordo com um grau de dificuldade considerado crescente pelo Tutor-Prolog. O ensino genérico está organizado de acordo com a taxonomia que melhor resultado obteve durante a observação exprimental na escola dos Olivais. Já o modelo de ensino individualizado actua por solicitação explícita do aluno ou do sistema. Ambos poderão alterar a taxonomia inicial das lições, de acordo com as informações qualitativas e quantitativas obtidas no modelo

do aluno.

Se, por exemplo, o Tutor-Prolog apresentar o conteúdo instrucional na forma "A" (maior grau de dificuldade inicial) e o aluno não apresentar um bom desempenho, o Tutor-Prolog apresenta o mesmo conteúdo na forma "B" ou "C" (parte do menor grau de dificuldade envolvido pelo contexto, visando chegar até "A"). Logo, a apresentação do material instrucional também é feita de forma dinâmica e contribui para a formação do modelo do aluno.

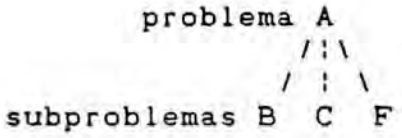
O método heurístico da decomposição de problemas em sub-problemas foi utilizado pela primeira vez no sistema GPS ("General Problem Solving sistem") Newell e Simon [NEW;SIM 63;72]. A sua importância para os tutores foi sugerida mais tarde por Ohlsson [OHL 87].

Durante a interacção o Tutor-Prolog apresenta exemplos, definições e exercícios, que o aluno procura entender, simulando a execução dos exemplos. Num passo imediatamente posterior, o aluno apresenta os seus próprios exemplos ao Tutor-Prolog, que os analisa para aferir o grau de conhecimento que o seu utilizador passou a deter sobre o assunto. Em função das informações adquiridas, o Tutor-Prolog faz o planeamento inicial do material instrucional, que pode ser alterado conforme for o desempenho do aluno. Ou seja, as hipóteses tanto podem ser confirmadas como tornarem-se falsas. O grau de dificuldade do curso aumenta em cada nova interacção, a medida que os problemas vão sendo solucionados e os novos conhecimentos vão sendo adquiridos, tanto

pelo Tutor-Prolog, como pelo aluno.

Pretende-se que o aluno possa acompanhar, passo a passo, através da simulação, o processo de resolução de qualquer problema [SEL;BRO 82], e que o Tutor-Prolog crie situações para ilustrar determinado conteúdo ou para se certificar da real compreensão do aluno sobre um conteúdo instrucional.

Consideremos a apresentação do problema "A", o qual se divide em vários subproblemas:



Se o aluno falhar na resolução do problema "A", a falha deve-se ao facto de não possuir conhecimentos que lhe permitam resolver o conjunto dos subproblemas "B","C" e "F". Supondo que a falha ocorreu pelo facto do aluno não saber resolver o sub-problema "F", o Tutor-Prolog passa a decompor "F" em novos sub-problemas e a ensinar os conteúdos instrucionais que estão associados à sua resolução. O Tutor-Prolog considera que os conhecimentos envolvidos pelos sub-problemas "B" e "C" são do conhecimento do aluno. Se noutra circunstância o aluno não souber resolver problemas que contenham as situações "B" ou "C" e se uma dessas situações estiver relacionada com o provável motivo do erro, os conteúdos instrucionais envolvidos por "B" e/ou "C" serão revistos.

Assim, se o aluno não entender a simulação do problema "A", o Tutor-Prolog partirá da hipótese de que o aluno está com dificuldades apenas nos novos conteúdos envolvidos no problema "A". Por exemplo, no caso de um novo conceito de programação, passa a desenvolver melhor este conceito. Esta é a primeira estratégia que o Tutor-Prolog recorre no ramo da 'árvore' correspondente aos conteúdos alternativos. Se isso não resultar, ocorre um retrocesso para os conteúdos instrucionais envolvidos no problema (linguagem Prolog). Observe-se que o nível de retrocesso que pode ser alcançado depende exclusivamente do grau de complexidade do problema, ou seja, da sua distância (número de nós existentes) da "folha" da "árvore instrucional". Ora, o Tutor-Prolog pode retroceder para vários "ramos" e chegar a várias "folhas".

No modelo do Tutor o problema "A" possui um grau de dificuldade "N" e os subproblemas "B", "C" e "F" são de dificuldade N-1. Se o aluno resolver satisfatoriamente "A", o Tutor-Prolog gera hipóteses de que "B", "C" e "F" são conhecimentos de domínio do aluno e apresenta um novo problema de complexidade A+1. Este novo problema utiliza, para a sua resolução, os conhecimentos envolvidos em "A".

A técnica da decomposição de um problema em subproblemas com menores graus de complexidade (simplificação), na tentativa de aproximar o modelo real do aluno ao modelo do Tutor-Prolog, é utilizada porque não abordamos a geração de hipóteses específicas

sobre os motivos que levaram o aluno a cometer um erro. Isto é, possuimos hipóteses e conhecimentos sobre o que o aluno sabe. Estas hipóteses, subtraidas dos conhecimentos do modelo do Tutor, resultam nos conhecimentos que o aluno supostamente não adquiriu, mas não possuímos hipóteses sobre o porquê da ocorrência do erro. Logo, as explicações alternativas devem ser exaustivas, embora sejam geradas dentro de um contexto instrucional[12] bem determinado.

2.4 Modelo do aluno

O Tutor-Prolog necessita de conhecer o seu utilizador e o modo como utiliza o sistema. O modelo representa o conhecimento que o aluno tem sobre determinado tópico da linguagem Prolog e as crenças que o Tutor-Prolog gera a respeito do conhecimento do aluno.

No modelo do aluno guarda-se toda a informação sobre o aluno reconhecido pelo Tutor-Prolog: informações de carácter estático, obtidas através de um questionário inicial, e o conhecimento de caráter dinâmico, obtido durante as diferentes interacções com o aluno. Consideramos os dados pessoais como informações estáticas,

---

12 Por contexto instrucional entendemos o assunto que compõe cada lição (átomos, factos, objectivos, etc.) e o grau de dificuldade em que as lições e os exemplos estão sendo formados.

ao contrário do que acontece no sistema MYCIN [CLA;87] pois no Tutor-Prolog estes dados servem fundamentalmente para gerar a primeira imagem do aluno e não possuem uma influência decisiva no processo de ensino. As informações dinâmicas correspondem ao desempenho do aluno durante o tutorial e originam um conhecimento quantitativo que representa aquilo que o aluno já sabe sobre o assunto que está sendo ensinado. O conhecimento é organizado através de hipóteses, que definem as estratégias de ensino ou de aprendizagem, podendo ser verdadeiras ou falsas. A formação e a estrutura das hipóteses[11] são tratadas no Capítulo 3. Estes dois tipos de informação são utilizados no processo de planeamento das diferentes acções de ensino e na realização da aprendizagem ao nível da utilização que o aluno faz do Tutor-Prolog como um todo, isto é, interagindo através da linguagem Prolog e através da língua natural escrita.

Para Carr e Goldstein [CAR;GOL 77], o modelo do aluno possui o estado do seu conhecimento sobre determinado assunto. Este estado é obtido através da comparação entre o conhecimento do perito e o conhecimento do aluno durante a execução de determinada tarefa ("overlay model"). Este é o tipo de modelo que foi adoptado no Tutor-Prolog.

---

11 Tudo aquilo em que o Tutor acredita, ou consegue verificar a respeito do conhecimento do aluno, é representado através de um conjunto de frases escritas na lógica de primeira ordem e constitui o conhecimento ou as hipóteses do Tutor a respeito do aluno.

Já, para Brown, Burt e Sleeman [BRO;BUR 78] e [SLE 82], o modelo do aluno é o conjunto de regras utilizadas pelo aluno na tentativa de obter a solução para um problema. Este conjunto representa as variações às regras definidas no tutor e é rotulado como um conhecimento inadequado. Discordamos deste conceito no que se refere à afirmação "é rotulado como um conhecimento inadequado". Somos da opinião de que um tutor inteligente também precisa de aprender com o aluno, pois o aluno, muitas vezes, possui conhecimentos a respeito do domínio de aplicação, que também estão correctos. Para além disso, cada aluno tem um estilo próprio de solucionar os problemas. A solução pode estar correcta, mesmo não sendo a melhor, e não estar contemplada nas regras do tutor.

Para obter o modelo do aluno, o Tutor-Prolog segue as técnicas propostas por Clancey [CLA 82], tanto nas interacções ao nível do ensino, como ao nível da escrita de programas ou da consulta à basé de dados. Os indicadores básicos para a obtenção deste modelo são a realização de perguntas directas ao aluno que involvem informações não utilizadas directamente em decisões (nome, sexo, ...) e informações utilizadas na planificação e selecção da estratégia de ensino a ser utilizada (idade, nível escolar, conhecimento de outras linguagens de programação, ...) e, a apresentação de conteúdos instrucionais gerados de acordo com a experiência adquirida pelo aluno e com as dificuldades observadas durante a interacção (desempenho).

- O progresso do desempenho do aluno observado pelo Tutor-Prolog

Em nossa opinião este indicador apenas permite a classificação do aluno. Para obtê-lo, utilizamos somatórios de erros, o contexto em que estes ocorrem (lição) e o percurso realizado pelo aluno dentro de cada bloco instrucional. Estas informações são representadas através de factos Prolog, como se apresentam na figura 2.6.

```
-------------------------------------------------------------
              tipo_do_erro(ocorrencia,erro,lição)
-------------------------------------------------------------
```

Figura 2.6 Somatórios de erros

Como vimos, o material instrucional está organizado em forma de árvores binárias. O percurso consta das lições das árvores instrucionais utilizadas e é obtido através do retrocesso aos nós (lições) visitados. O mecanismo de retrocesso é activado através das indicações contidas no apontador apresentado na figura 2.7 a) e b). Em b) temos a referência à última lição apresentada. Esse indicador é utilizado no regresso ao tutorial, isto é, quando o aluno interrompe uma sessão de trabalho ou troca o contexto de trabalho. A forma genérica corresponde a passou(x,y,h), onde "x" representa o bloco instrucional por onde deve ser reiniciado o tutorial, "y" representa o nível de complexidade para a formação das lições e "h" representa o apontador para a lição que deve ser apresentada de acordo com na forma a).

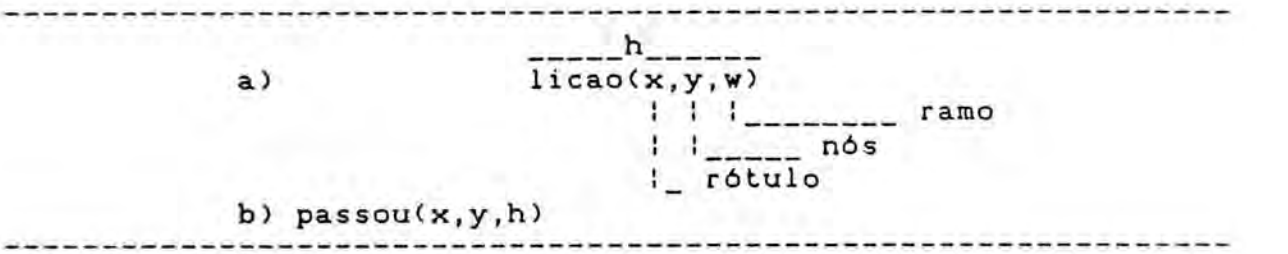


Figura 2.7 Estrutura do apontador para as lições

Este apontador faz parte (argumento) de uma estrutura maior, gerada de acordo com as hipóteses que o Tutor-Prolog utiliza ao optar por determinada estratégia de ensino.

- Perguntas directas dirigidas ao aluno

As perguntas ajudam a classificar o aluno e a perceber o grau de conhecimento que este possui sobre determinado assunto, guiando o Tutor-Prolog de forma directa e objectiva. Utilizamos este recurso para obter a classificação inicial do aluno, visando estabelecer o primeiro plano de ensino, e, também, para obter as informações que permitissem guiar o Tutor-Prolog em operações intermediárias na apresentação do material intrucional, isto é, para os casos em que o Tutor-Prolog ainda não teve oportunidade de observar o desempenho e obter a representação dos conhecimentos que o aluno possui a respeito de um determinado problema (perguntas 1, 2, 3, 4 e 8). As perguntas 5, 6, 7, 9 e 10 são utilizadas com o objectivo de seleccionar grupos de alunos para fins de avaliação (ver anexo 2 e, para maiores detalhes [ZAM 89]). As perguntas directas em geral possuem um carácter estático (factos) e são obtidas através do questionário inicial,

figura 2.8. No ambiente de consulta à base de dados o Tutor-Prolog utiliza as perguntas para a aquisição de conhecimentos linguísticos.

```
-----------------------------------------------------------------
  Qual é o teu nome?
  Qual é a tua idade?
  Qual é a tua turma?
  Reprovaste no último ano?
  Qual foi a nota que obtiveste no último ano em Português?
  Qual foi a nota que obtiveste no último ano em Matemática?
  Qual foi a nota que obtiveste no último ano em Física?
  Conheces alguma linguagem de Programação? (Qual?)
  Qual é a profissão do teu pai?
  Qual é a profissão da tua mãe?
-----------------------------------------------------------------
```

Figura 2.8 Questionário inicial

- **Apresentar conteúdos instrucionais que são gerados de acordo com a experiência que o aluno vai adquirindo e com as dificuldades observadas durante a interacção**

Este aspecto introduz a utilização de informações dinâmicas obtidas durante a interacção. O contexto histórico do desempenho apresentado pelo aluno é utilizado pelo Tutor-Prolog para a geração automática de exemplos Prolog, para a geração das frases-exemplos e para a geração dos problemas-exemplos.

As informações dinâmicas são as que permitem a análise qualitativa do aluno. Elas representam o modelo dinâmico do aluno. A obtenção e utilização destas informações são tratadas nos Capítulos 3, 4 e 5.

Resumidamente, as informações dinâmicas podem ser classificadas em três tipos: as que representam o desempenho do aluno, as

que constituem os planos de ensino gerados pelo Tutor-Prolog e as de caráter operacional.

O primeiro caso contempla as informações que o aluno é levado a introduzir durante a primeira sessão - substantivos e verbos (Anexo 1) - e que são utilizados pelo Tutor-Prolog aquando da geração dos exemplos. Fazem também parte deste caso os exemplos e os exercícios que o aluno escreve no decorrer do tutorial e, os erros cometidos durante estas operações (Capítulo 4). Ainda, nas consulta a base de dados o aluno pede vir a introduzir novos vocábulos ao dicionário linguístico ou alterar alguma estrutura sintática existente no interpretador (Capítulo 5).

No segundo grupo encontram-se as informações utilizadas, pelo Tutor-Prolog, para a organização do processo de ensino particular a cada aluno de acordo com a sua actuação. O plano tutorial é organizado ao redor dos conhecimentos e das hipóteses, que o Tutor-Prolog gera, a respeito do que o aluno já sabia ou aprendeu e, de como o processo de ensino deve ter prosseguimento (Capítulo 3).

Finalmente, são acrescentadas, ao meodelo do aluno, as informações de controle, utilizadas pelo Tutor-Prolog, para os casos em que o aluno troca o contexto de trabalho ou interrompe a sessão.

Logo, após a primeira interacção, o Tutor-Prolog utiliza o

modelo inicial do aluno adquirido durante a sessão. Com o decorrer da interacção, o Tutor-Prolog vai formando sequências-padrões a partir das actuações do aluno frente a situações que ocorrem em contextos idênticos ou semelhantes. Organiza, portanto, uma rotina interna de funcionamento de acordo com cada utilizador.

O procedimento realizado pelo Tutor para obter o modelo do aluno pode ser resumido nos seguintes passos:

(a) guardar novas informações obtidas através da interacção;

(b) procurar ligações entre as novas informações e os factos que formam a descrição da situação já existente; e,

(c) usar estas informações gerais para conduzir a utilização do sistema (plano), tornar a interacção mais dinâmica, aproximar o conhecimento do aluno do conhecimento do Tutor-Prolog e adequar os conhecimentos do Tutor-Prolog ao estilo do aluno.

Assim, pela apresentação de exemplos e definições, o Tutor-Prolog tenta aproximar ao máximo o modelo real do aluno (o que o aluno faz), do modelo do Tutor (o que o Tutor espera que o aluno faça).

A articulação de comandos de controle e de informações, que são realizadas pelo Tutor-Prolog durante o processo de ensino, encontra-se esquematizada na figura 2.9, que se apresenta em seguida:

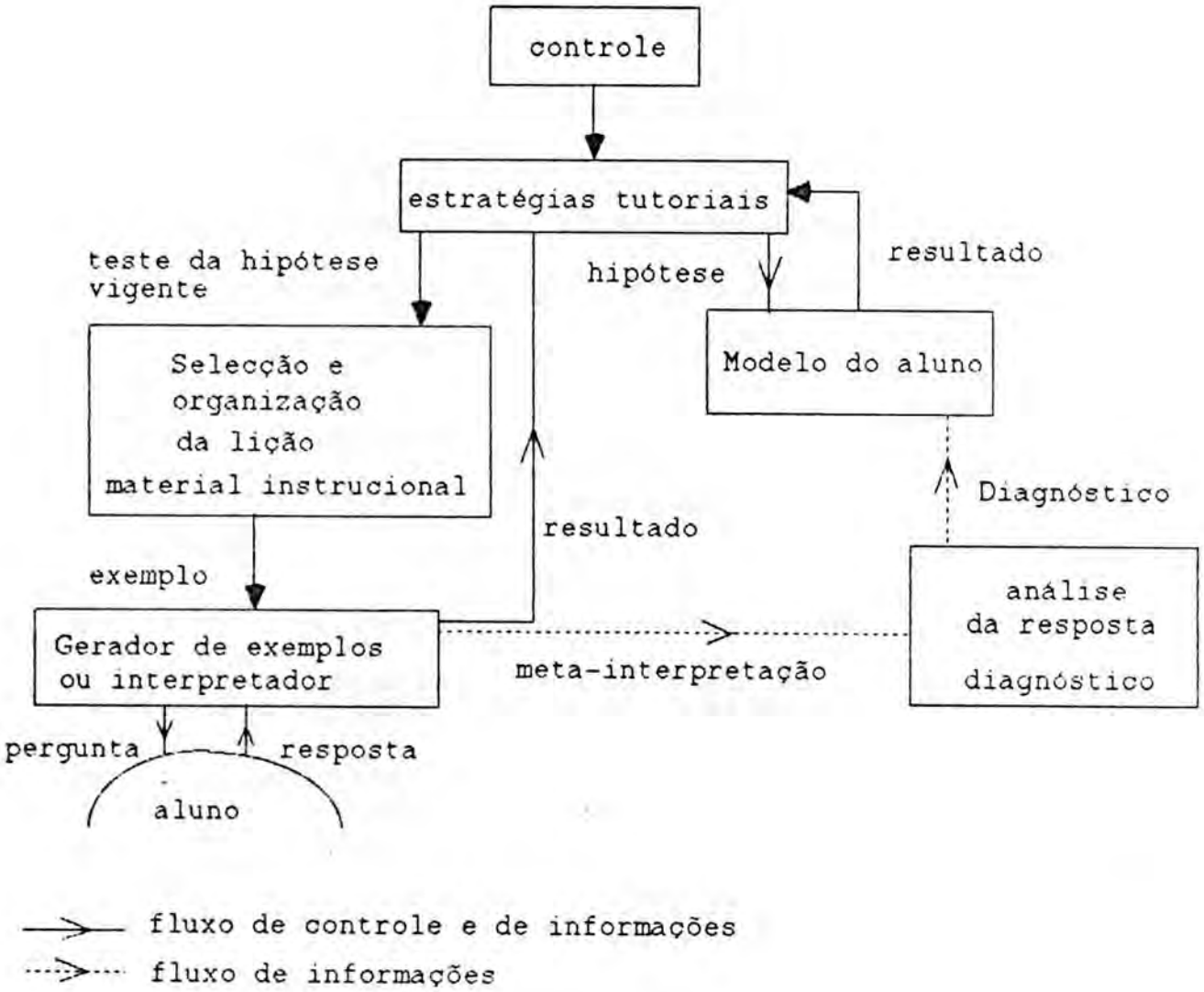


Figura 2.9 Diagrama dos componentes de ensino do Tutor-Prolog

A figura 2.9 mostra que o módulo de controle coordena o fluxo de informações que são trocadas entre os módulos envolvidos directamente no processo de ensino. O aluno emite uma pergunta a respeito de determinado contexto instrucional ou responde a uma pergunta do Tutor, apresentada pelo gerador automático de exemplos. A resposta é analisada pelo meta-interpretador Prolog, que emite um diagnóstico. O diagnóstico é passado ao modelo do aluno que o utiliza para actualizar a imagem que possui sobre os

conhecimentos do aluno. O resultado é a selecção de uma estratégia de ensino para direccionar o material instrucional a ser apresentado ao aluno.

Após esta breve introdução à arquitectura e à exploração do Tutor-Prolog passaremos a descrever em detalhe os seus principais módulos, com destaque no Capítulo 3 para o material instrucional e o modelo do ensino.

3. MODELO DO ENSINO

Neste Capítulo é apresentada a geração do modelo do aluno, no que se refere à organização e orientação do ensino, que tem por finalidade representar a imagem que o Tutor-Prolog vai formando a respeito dos conhecimentos que o aluno possui sobre a linguagem de programação Prolog.

O modelo do aluno gerado no Tutor-Prolog caracteriza-se por ser do tipo individual (um por aluno). A sua composição está directamente relacionada com a apresentação do material instrucional que segue um modelo de comunicação do tipo perguntas e planos ("goals and plans") [ALL;PER 80], [GER 81] e [GOO 85;86]; com a escolha do nível de complexidade em que cada lição deve ser gerada, que segue um modelo do tipo preferência ("attitudes" ou "preference") [RIC 79], [MOR;ROL 85] e [MOR 88]; e, com o processo de selecção da estratégia de ensino a ser adoptada, de acordo com um modelo do tipo conhecimento e crença ("knoledge and belief") [KOB 84;85] e [GEN;NIL 87].

O processo de planeamento do tutorial frente ao modelo do aluno pode ser representado pelas seguintes etapas:

1) Para a primeira interacção de um aluno:

(a) diálogo visando obter as informações mínimas necessárias e selecção e formalização destas informações (modelo inicial estanque do aluno que contém informações de carácter decisório);

(b) gerar o planeamento inicial do material instrucional de acordo com o modelo obtido;

(c) apresentar exemplos;

(d) avaliar a reacção do aluno frente aos exemplos apresentados; e

(e) incorporar as novas informações geradas com a apresentação dos exemplos.

2) Para as demais intervenções:

(a) carregar a descrição existente para o aluno;

(b) actualizar o planeamento do material instrucional de acordo com o modelo real obtido (modelo dinâmico do aluno);

(c) apresentar o material instrucional de acordo com a sequência do modelo do aluno; e

(d) actualizar o modelo do aluno de acordo com as novas informações obtidas.

A diferença entre o modelo do tutor (o que o Tutor-Prolog espera que o aluno faça) e o modelo real (o que o aluno faz) permite gerar o modelo que representa o conhecimento do aluno sobre o problema. Este modelo é utilizado no planeamento e na determinação das estratégias de ensino a serem utilizadas.

Todas estas informações devem ser concatenadas a fim de se obterem os **pontos comuns** da actuação do aluno durante um intervalo de tempo (uma sessão, um bloco instrucional, ...) e os pontos **comuns** entre o conhecimento do aluno e o conhecimento do Tutor-

Prolog. Os pontos comuns representam o que o aluno sabe em relação ao conhecimento do Tutor-Prolog. As diferenças representam o que o aluno não sabe, o que está equivocado, ou o que o Tutor-Prolog deve aprender.

Portanto, o Tutor-Prolog tem duas estratégias de acção frente às diferenças que surgem entre o conhecimento do perito e o conhecimento do aluno:

- ensinar o aluno, isto é, aproximar os conhecimentos do aluno aos conhecimentos do Tutor-Prolog;
- aprender com o aluno, isto é, adequar o modelo do Tutor-Prolog ao modelo do aluno.

3.1 Material instrucional

a) Organização do material instrucional

O material instrucional está organizado em quatro blocos, que contêm ensinamentos sobre um subconjunto da linguagem de progamação Prolog, tal como é indicado na figura 3.1. Cada bloco instrucional é organizado com uma taxonomia interna[1], que mantém um nível de complexidade crescente para a apresentação dos conteúdos.

---

1 A taxonomia assumida baseia-se nos resultados obtidos durante várias experiências no ensino da linguagem de programação Prolog [VIC 87],[ENN 81a], [KOW 82], [CLO;MEL 81] [STE;SHA 87] [COE;COT 88].

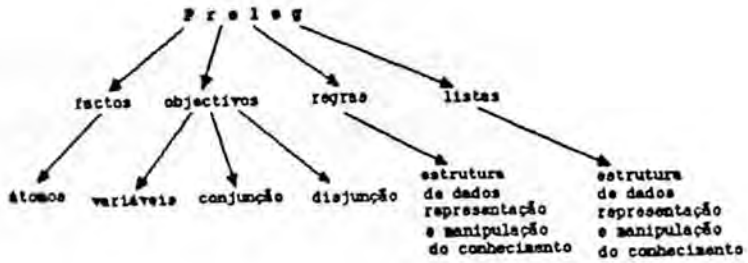


Figura 3.1 Blocos instrucionais

Cada bloco instrucional é composto por um conjunto de lições D_L, organizadas numa estrutura em árvore. O número de lições que compõem cada bloco instrucional depende da complexidade do assunto que está sendo abordado. Por exemplo, para ensinar objectivos, temos lições para o uso de átomos, de variáveis, da conjunção (','), da disjunção (';'), e ainda para a observação do sucesso ou falha nas operações de consulta a bases de dados.

Cada lição é formada dinamicamente de acordo com as informações obtidas durante a interacção, o índice de desempenho obtido nas lições já apresentadas[2], os exemplos gerados automaticamente, os exemplos escritos pelo aluno, as mensagens e os diagnósticos produzidos durante o processo de interpretação e simulação, como se apresenta na figura 3.2.

---

2 Nem sempre o Tutor-Prolog dispõe do índice de desempenho. O seu cálculo só é possível quando o aluno recebe lições, realiza simulações e escreve exemplos.

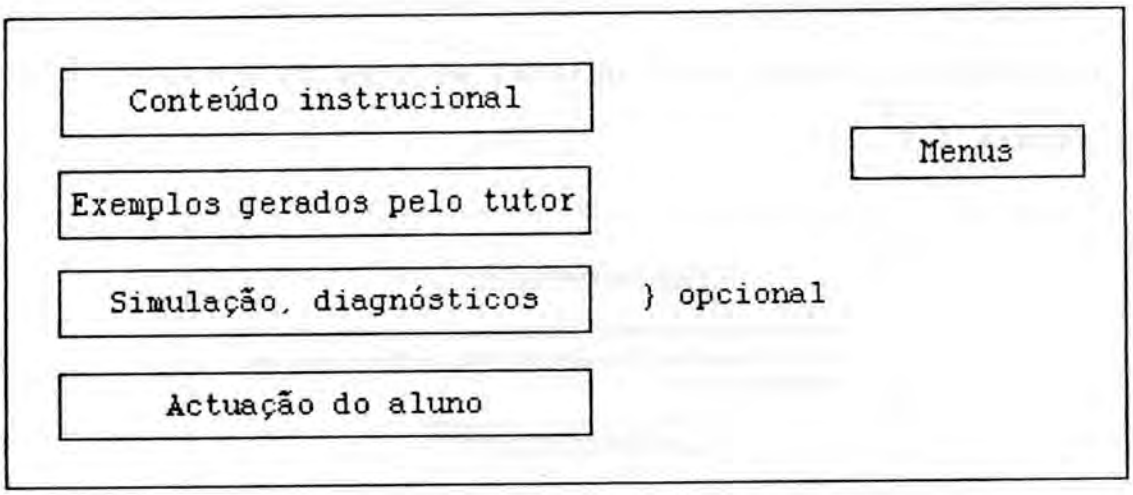


Figura 3.2 Estrutura das lições

O Tutor-Prolog pode recorrer, ainda, no decorrer da interacção, a um conjunto de lições alternativas D_{La}. Este conjunto é gerado em função do desempenho do aluno numa determinada lição de D_L. As lições alternativas actuam de acordo com as solicitações do aluno ou do Tutor-Prolog. A necessidade pode ser expressa pelo aluno, que diz não entender uma lição, ou pelo Tutor-Prolog, que detecta a apresentação de exemplos inadequados em um determinado contexto instrucional.

As lições de cada D_L estão organizadas em forma de árvores binárias, onde o ramo esquerdo corresponde ao sucesso e o ramo direito, ao insucesso. Quando o Tutor-Prolog necessita gerar uma sequência de várias lições alternativas para uma mesma lição de um D_L, a árvore gerada a partir do primeiro nó alternativo possui apenas o ramo direito. Quando o aluno demonstra que entendeu o conteúdo em questão, ocorre o retrocesso até a próxima lição de D_L (próximo nó esquerdo).

Vejamos um exemplo prático do percurso feito na árvore de organização do bloco instrucional através do exemplo apresentado na figura 3.3 [3].

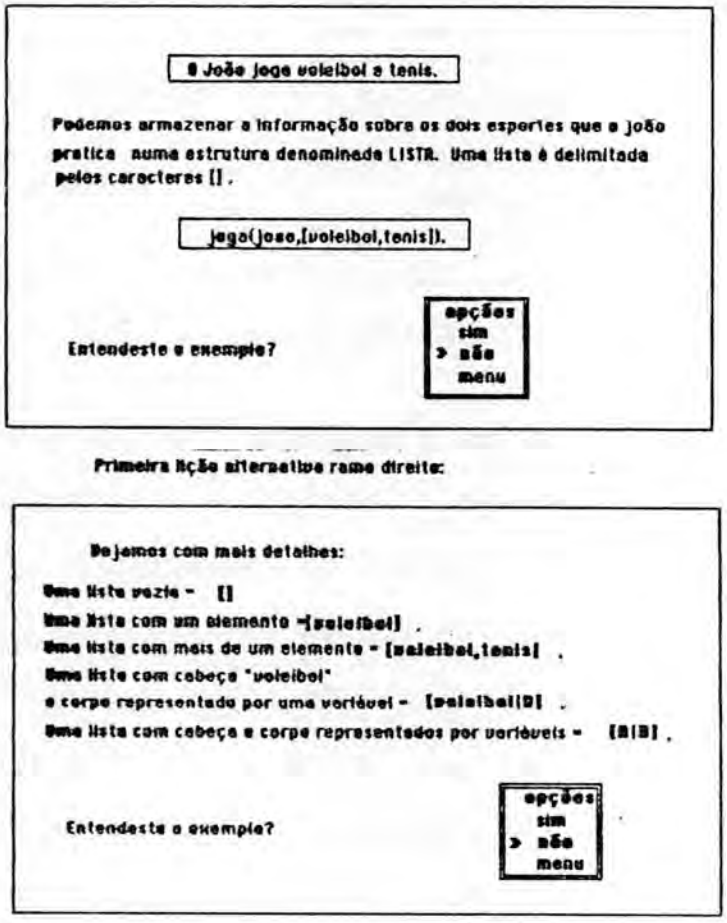


Figura 3.3 Apresentação de lições alternativas (1)

---

3 Na tentativa de clarificação o Tutor-Prolog pode utilizar simulações pré-definidas.

O meta-interpretador aceita como válida a construção [Lista].

Como o aluno, nesta etapa do curso, ainda não conhece o conceito de "unificação", optamos por substituí-lo pela palavra "casamento", que se mostrou mais acessível.

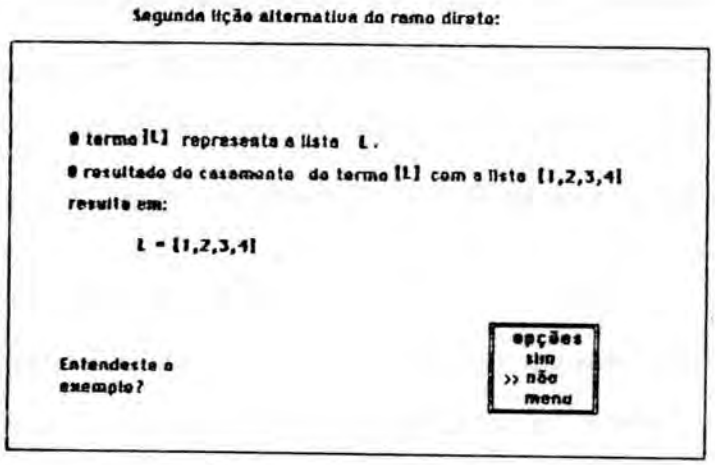


Figura 3.3 Apresentação de lições alternativas (2)

b) Apresentação do material instrucional

Segundo Ohlsson [OHL 87], um mesmo conteúdo instrucional pode ser apresentado sob várias formas. Por exemplo, a expressão:

$$\frac{a \times N}{b \times N} = \frac{a}{b}$$

- como um princípio ("uma fracção onde o numerador e o denominador compartilham um factor");
- como uma fórmula matemática (como neste exemplo);
- como um algorítmo ("primeiro procure os factores do numerador, então ..."");
- como uma demonstração de como funciona o algorítmo com um problema prático;
- como uma colecção de exemplos de problemas solucionados através do uso do algorítmo; e
- como a prova matemática de que o algorítmo está correcto.

A utilização de uma ou de várias formas para ensinar um determinado conteúdo depende das estratégias de ensino definidas no Tutor-Prolog. É natural que as estratégias estejam de acordo com a população a que o Tutor-Prolog se destina.

Caso o aluno persista na resposta negativa, o Tutor-Prolog passa a um nível menos complexo e explica o conceito de listas de uma segunda forma prevista neste nível. Caso contrário, é apresentada a próxima etapa prevista, que pode ser outra lição ou a solicitação de exemplos.

Como foi apresentado na figura 3.2 e nos exemplos da figura 3.3, fazem parte de uma lição frases exemplos escritas em Português e em Prolog. Assim, a geração automática de exemplos está organizada segunda essas duas vertentes, as frases em Prolog e as frases em Português. Ambas destinam-se a ilustrar o corpo das lições, pois são geradas de acordo com o contexto instrucional que as solicita e a partir de substantivos e verbos informados pelo aluno.

Os exemplos correspondentes à linguagem de programação Prolog são gerados com base nos mesmos princípios utilizado na gramática que realiza a análise sintáctica, semântica e pragmática do programa fornecido pelo aluno (Capítulo 4). Se o programa receber como entrada uma cláusula, realiza a sua análise. Se a entrada for a solicitação de uma cláusula, gera-a. Neste último caso o processo é simplificado, pois como é necessário apenas gerar uma cláusula, esta é gerada correctamente, sem que seja

preciso activar o processo de detecção e correcção de erros. A geração de uma cláusula é feita de forma ascendente, da esquerda para a direita.

Os objectivos e as relações utilizadas na geração dos exemplos são obtidas na base de dados do aluno. Caso o aluno não informe um contexto, isto é, inicie o curso a partir de um ponto qualquer, o Tutor-Prolog assume o do modelo ideal ou, em último caso, os caracteres ASCII. Os exemplos são utilizados para ilustrar os conteúdos teóricos sempre que existir um desvio entre o modelo ideal de ensino e o modelo real do aluno [CAR;GOL 79].

A cláusula-exemplo (Prolog) gerada, deve relacionar-se com a frase em Português que o antecede. Assim, os objectos e as relações (substantivos e verbos) são passados, do gerador de exemplos da língua natural, para o gerador de exemplos Prolog, como parâmetros. A cláusula gerada pode ser do tipo: átomos, variáveis, cadeias, factos, objectivos, listas, regras e funções (cláusulas compostas). A comunicação entre os dois módulos obedece às seguintes formas gerais:

---

termo(termo_prolog,<parâmetros>,tipo)

frase(tipo,<parametros>,frase_potuguês)

As frases-exemplos em Português são formadas a partir de regras e dos conhecimentos linguísticos do dicionário que o

Tutor-Prolog gera no decorrer da interacção e, portanto parte do modelo do aluno. Encontram-se divididas em dois grupos, que correspondem às frases afirmativas positivas e às frases interrogativas. Em ambos os casos as frases podem envolver conjunções.

No caso da geração de exemplos Prolog, a frase gerada está sempre inserida dentro do contexto que a solicitou. O tipo de frase, que deve ser gerada, é indicado de acordo com o contexto da lição que a solicita. Se a lição possuir um texto explicativo em que já foram utilizados objectos e relações (substantivos e verbos), estes também são enviados como contexto. Caso contrário, são seleccionados aleatoriamente. A coerência da frase gerada é obtida, como já foi visto, pelo contexto que a solicitou e pelas informações referentes à concordância armazenadas no dicionário.

Vejamos o processo de geração de exemplos a partir de um exemplo. Supondo que este tenha sido solicitado pelo contexto instrucional de ensino dos objectivos de acordo com a estratégia de ensino adoptada. Assim deve ser apresentada uma lição sobre a utilização de variáveis na construção de objectivos. Supondo ainda que os parâmetros recebido tenham sido os seguintes: um substantivo (João) e um verbo (gostar).

```
frase(interrogativa,[joao],[gostar],F)
```

Onde temos, respectivamente, o tipo da frase que deve ser gerada,

os argumentos e a frase final. O número de argumentos informados na lista de objectos e de relações está directamente relacionado com o tipo de frase solicitado. O resultado "F" é a frase:

O João gosta de quem?

As informações recebidas pelo programa gerador de frases em Português são transmitidas ao programa gerador de frases Prolog. Este último deve gerar um objectivo composto por um objecto (joao), por uma relação (gostar) e por uma variável.

```
entrada:
termo(T,[],[joao],[gostar],2,objectivo,[])

saída:
termo(?-gostar(joao,X),[X],[joao],[gostar],2,objectivo,[X:_0028])
```

A lista de variáveis eventualmente já utilizadas no texto da lição, a lista de objectos, a lista das relações, o número de argumentos que a cláusula gerada deve ter, bem como o seu tipo são recebidos como parâmetros. A lista das variáveis, que eventualmente sejam geradas, e a cláusula gerada são devolvidas.

?- gostar(joao,X).

Por último, as lições podem conter exercícios simulados e

orientados pelo Tutor-Prolog, com o objectivo de introduzir o aluno nos conceitos relativos à Programação em Lógica. Estão previstos uma série de exercícios orientados e relacionados com árvores regras de produção e enquadramentos. Os exercícios são descritos através de um texto em Português e são inspirados nos objectos e nas relações existentes na base de dados do aluno. Estão previstas duas situações: os problemas podem já estar programados, cabendo aos alunos apenas fornecer os dados para a sua simulação ou, então, o Tutor-Prolog apresenta a definição do problema em Português e o aluno deve representá-la em Prolog. A execução simulada pode ser acompanhada através do procedimento de retrocesso inteligente e do comando espiar (Capítulo 4). O aluno conta, em ambos os casos, com a orientação do Tutor-Prolog.

O objectivo principal dos exercícios é fixar os conceitos dos conteúdos apresentados e dar uma noção sobre os procedimentos recursivos na Programação em Lógica. Os exercícios correspondem às últimas lições de cada bloco instrucional. Na figura 3.4 é apresentado um exemplo de exercício de nível 3 previamente definido no Tutor-Prolog. O objectivo do exercício é que o aluno interaja com um programa recursivo e, ao mesmo tempo, simule a pesquisa de elementos armazenados em estruturas como as listas.

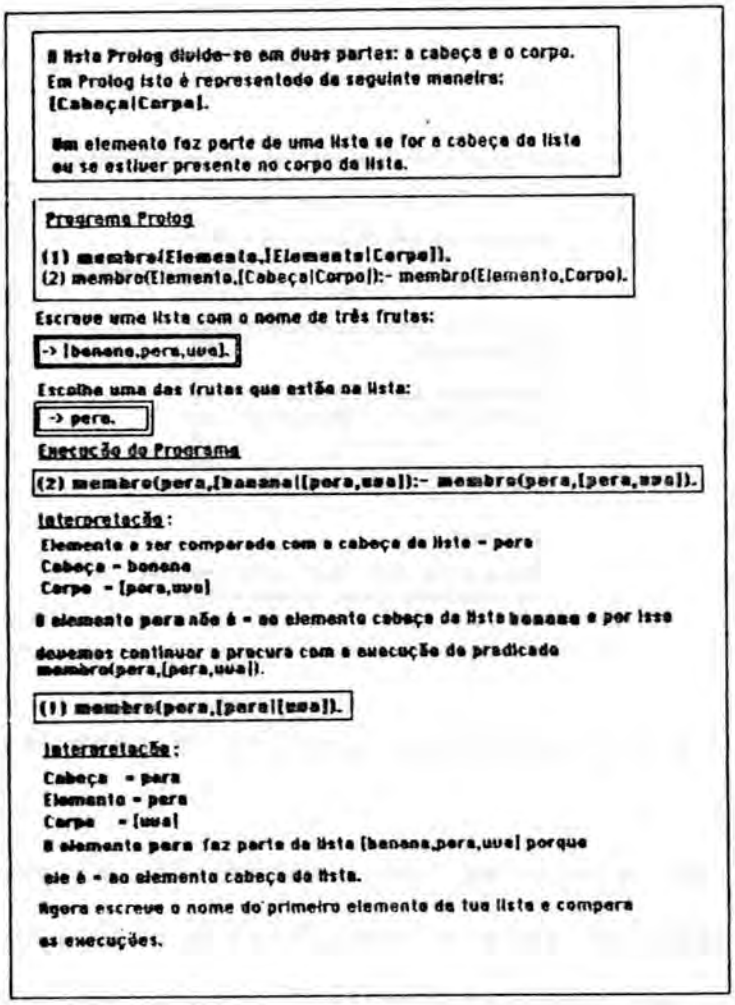


Figura 3.4 Exercício de nível 3

Outra opção para o acompanhamento da execução é a apresentação gráfica dos elementos pesquisados. Neste caso, o aluno pode observar a movimentação dos elementos do corpo para a cabeça da lista. Ou seja, a tentativa de unificação do elemento procurado com o elemento que é a cabeça da lista. O processo de procura de um elemento é sempre animado com o uso de cores. Apesar das limitações gráficas, na figura 3.5 é feita uma tentativa de apresentação deste processo. Naturalmente, a

impossibilidade da reprodução a cores empobrece a ideia original.

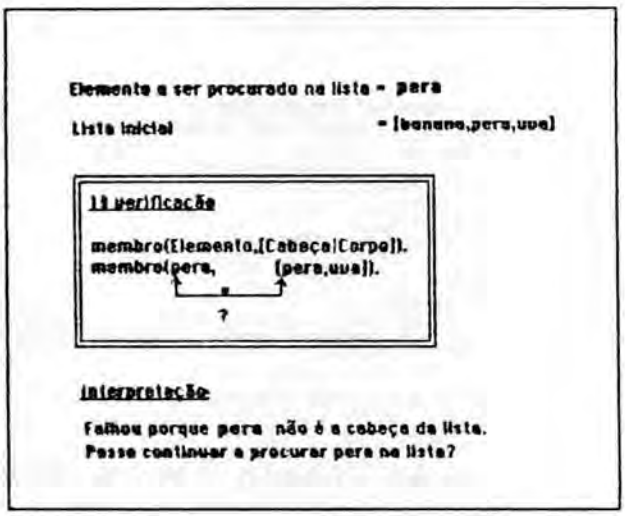


Figura 3.5 Apresentação gráfica da pesquisa em listas

Quando um aluno dá por encerrada a exploração de um exercício, são-lhe feitas perguntas a respeito da heurística envolvida na solução apresentada. As respostas servem para avaliar o grau de compreensão que o aluno atingiu sobre o conteúdo.

Por exemplo, em relação ao exercício anterior serão feitas perguntas como:

(1) Em que linha (1 ou 2) do programa membro é obtida a resposta à pergunta sobre se um elemento pertence ou não à lista.

(2) Quantas vezes o procedimento membro deveria ser executado, para chegar à conclusão de que o elemento d não pertence a lista [a,b,c].

O Tutor-Prolog é ainda capaz de resumir o conteúdo dos

blocos instrucionais. Essa capacidade, é de grande importância pois permite que os alunos, que já tem algum conhecimento sobre a linguagem Prolog, solicitem apenas o resumo de um determinado bloco instrucional e passem para o estudo dos assuntos que desconhecem.

O resumo é gerado com base no modelo ideal de ensino (nível 2). Mais especificamente, a partir do "esqueleto" das lições chaves, isto é, as que introduzem novos conceitos dentro de cada bloco. Na figura 3.6, é apresentado o resumo gerado para o bloco instrucional dos objectivos.

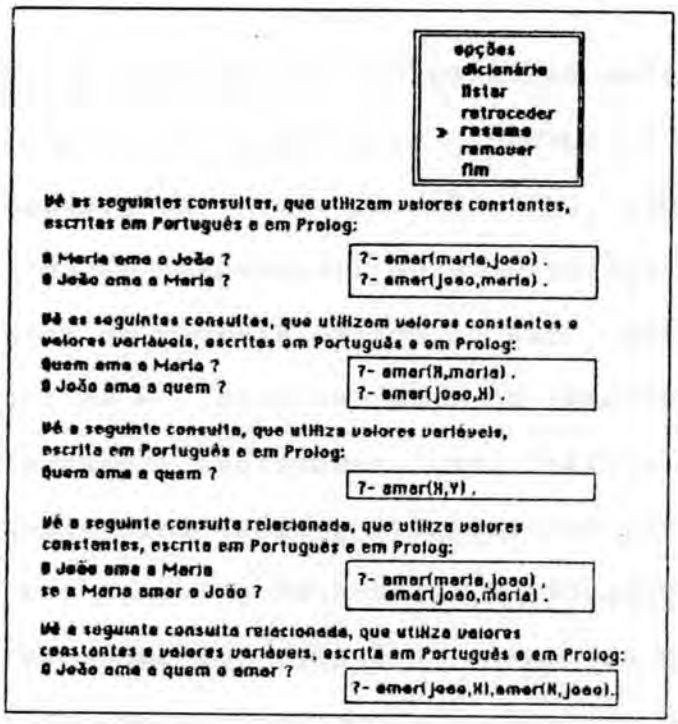


Figura 3.6 Resumo do conteúdo instrucional

3.2. Estratégias de ensino

As quatro estratégias de ensino utilizadas pelo Tutor-Prolog [VIC;COE;COS 88] são: (1) apresentar um mesmo material instrucional em diferentes níveis de complexidade de acordo com as necessidade de cada aluno; (2) apresentar o conteúdo instrucional de acordo com uma taxonomia (a utilização de pré-requisitos hierárquicos na educação foi investigada por educadores e psicólogos como [GAG 62], [RES 73], [RES,WAN;KAP 73]) formada a partir dos conhecimentos existentes no modelo do aluno; (3) ensinar a representação dos problemas em Prolog a partir da sua definição em língua natural escrita; e, (4) fornecer diagnósticos, mensagens, alertas e ajudas ao aluno durante a interacção;

As informações estratégicas são obtidas durante as actuações do aluno, isto é, quando o aluno escreve exemplos ou questiona o Tutor-Prolog. Das actuações do aluno são guardadas informações, como o grau de dificuldade na compreensão de um exemplo gerado pelo Tutor-Prolog, que é obtido através do percurso utilizado durante a apresentação de cada conceito, e as informações sobre as dificuldades sintácticas, semânticas e pragmáticas observadas durante a escrita dos exemplos gerados pelo aluno (Capítulo 4). Com estas informações o Tutor-Prolog avança com maior rapidez, conforme o plano do modelo do Tutor, ou detém-se em detalhes na apresentação do conteúdo instrucional de acordo com as necessidades do aluno.

O conteúdo do modelo do aluno, que permite a tomada de decisão a respeito da estratégia a ser utilizada em determinado instante, é constituído por hipóteses e conhecimentos estratégicos.

Como já foi dito, o Tutor-Prolog é o resultado de trabalho experimental, tendo sido construído a partir da observação prática, e a sua organização não foi baseada em modelos lógicos. O resultado obtido, principalmente no que se refere ao modelo do ensino, aproxima-se bastante, no aspecto expressivo, das formas da Lógica Modal apresentada por [GEN;NIL 87].

Acreditamos que os modelos dos TIs devam evoluir para uma representação que utilize a Lógica Modal (Self [SEL 88] aponta essa direcção de trabalho futuro).

No trabalho de Genesereth e Nisson, as crenças (B) e os conhecimentos (K) têm a forma de frases da Lógica de 1a. ordem. Estes operadores modais **B** e **K** possuem, por definição, dois argumentos: o primeiro, é um termo que denota o indivíduo (actor $_1$) que crê ou que conhece [HIN 62]; o segundo, é uma fórmula que representa o que o actor$_1$ crê ou conhece.

Num diálogo é necessário representar as crenças ou conhecimentos que um actor forma a respeito do conhecimento do interlocutor, como no caso da interacção com o Tutor-Prolog, pelo que interessa estabelecer o que o actor$_1$ crê ou conhece a respeito do actor$_2$.

No estado actual do Tutor-Prolog, o actor $_1$ pode representar tanto o Tutor-Prolog como o aluno (1). Já, no caso das crenças embricadas (2), o actor representa sempre o Tutor-Prolog e o actor $_1$ representa, apenas, o aluno. Isto equivale a dizer que apenas $_2$ estão representadas as crenças que o Tutor-Prolog gera a respeito do conhecimento do aluno. No entanto, um modelo do tipo conhecimentos e crenças poderá tornar-se recursivo, isto é, o modelo do aluno poderá incluir informações sobre o que o aluno acredita que o tutor acredita a seu respeito, sobre o que o aluno acredita que o tutor acredita que o aluno acredita a respeito do tutor, e assim por diante.

No primeiro caso (1), a sintaxe da linguagem é definida da seguinte forma:

(a) Todas as fórmulas bem formadas do Cálculo de Predicados (CP) de 1a. ordem;

(b) Se ϕ é uma fórmula bem formada do CP 1a. ordem, sem variáveis livres, e se a é um termo instanciado, então B(a,ϕ) é uma fórmula bem formada;

(c) Se ϕ e ψ são fórmulas bem formadas, então $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$, $\phi \Leftrightarrow \psi$ e not ψ são fórmulas bem formadas da CP 1a ordem.

O que equivale a expressões com a seguinte forma:

B(R1,P(A)) (1)

No segundo caso (2), linguagem difere da anterior, pois em (b) é permitido que ϕ seja qualquer fórmula bem formada da CP 1a. podendo conter o operador B.

O que equivale a expressões com a seguinte forma:

$$B(R1,B(R2,P(A))) \quad (2)$$

A diferença fundamental entre as crenças e os conhecimentos reside no facto de as crenças poderem ser falsas, enquanto que os conhecimentos são sempre verdadeiros, isto é:

$$K(a,\phi) \equiv B(a,\phi) \wedge true\ (\phi)$$

Notar que os sistemas de crenças associados aos estados mentais de professores e alunos são, por natureza, incompletos e, geralmente, inconsistentes. Do ponto de vista dos TI's apenas é importante que a axiomática destes sistemas seja psicologicamnte válida.

No Tutor-Prolog, optamos pela designação de "hipóteses" e "conhecimentos", e não "crenças" e "conhecimentos" (os operadores lógicos B e K respectivamente). O motivo para esta opção é que o objectivo deste trabalho não é o de estudar formalmente o sistema lógico associado aos TI's.

a) Organização do ensino

O Tutor-Prolog é capaz de apresentar um mesmo material instrucional em diferentes niveis de complexidade, de acordo com as necessidades de cada aluno, encontrando-se actualmente

definidos três níveis [VIC;COE;COS 88], como se indica na figura 3.7. Esta organização do ensino passa a ser agora detalhada.

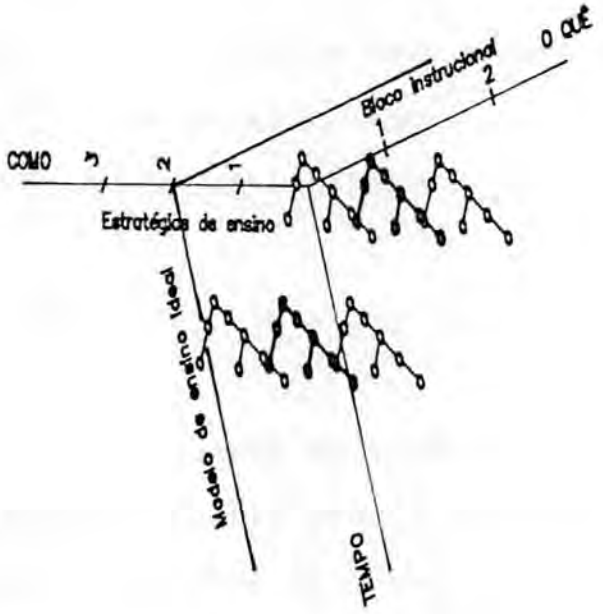


Figura 3.7 Organização do ensino

Os números do eixo (o quê) representam o início de cada bloco intrucional que, por sua vez, são compostos por lições. Cada lição pode conter mensagens, diagnósticos, simulações e exercícios. Os número do eixo (como) representam as estratégias de ensino, ou seja os níveis de complexidade (1,2 e 3).

O Tutor-Prolog é, também, sensível ao desenvolvimento da sessão, sendo capaz de manter um diálogo flexível com o aluno. O processo de aprendizagem do Tutor-Prolog é guiado pelo contexto (domínio) onde se desenvolve a interacção. Os objectivos da aprendizagem são a particularização ou a generalização do conhecimento existente. O contexto é organizado ao longo da

interacção através dos agentes do diálogo:

- "quem" --> actores;
- "o que" --> conteúdo instrucional;
- "onde" --> num determinado nível de complexidade;
- "quando" --> num determinado instante.

O caminho percorrido pelos actores durante a apresentação do conteúdo instrucional, dentro dos três níveis de complexidade, (figura 3.6) permite a obtenção da história da sessão ("background"). Esta história é obtida através do retrocesso aos diferentes nós da árvore (percurso), que representam o material instrucional já apresentado, e é uma das componentes do modelo do aluno que temos vindo a apresentar.

A selecção do conteúdo que compõe uma lição depende da imagem que o Tutor-Prolog tem, onde um dos componentes é o desempenho obtido pelo aluno durante a interacção com o conjunto de lições **lições(x,y,h)** anteriores e, em particular, durante a interacção com um determinado bloco de lições relevantes para a composição e apresentação do próximo bloco instrucional. No termo **lição(x,y,h)**, **x** indica o bloco instrucional actual, y representa o nível de complexidade em que o conteúdo instrucional da lição deverá ser apresentado e h é o apontador para uma lição específica dentro do bloco, instrucional em questão (indicando a árvore instrucional, o nó e o ramo). O conjunto de **lições(x,y,h)** representa o passado do aluno e a evolução do conhecimento. Estas informações são utilizadas pelo Tutor-Prolog para a definição das suas estratégias de ensino.

[i]

b) Obtenção do desempenho do aluno

Para obter o desempenho do aluno em cada bloco instrucional, o Tutor-Prolog observa a sua actuação em tarefas como a resolução de problemas e a escrita de exemplos. Assim, o Tutor-Prolog verifica, através da observação do desempenho do aluno nos vários blocos instrucionais, ou num determinado conjunto de lições apresentadas [i], o que o aluno requer. Se, por exemplo, ele requer o nível de complexidade y para entender melhor um determinado conteúdo instrucional, o Tutor-Prolog passa a apresentar o conteúdo dos demais blocos, preferencialmente, de acordo com o modelo do nível y.

Supondo que o conhecimento que o Tutor-Prolog possui no modelo do aluno corresponda às respostas do questionário inicial, de acordo com o exemplo de interacção apresentado no Anexo 1, isto é, com o aluno iniciando o tutorial, temos o conjunto de lições percorridas pelo aluno, como se apresenta na figura 3.8.

```
licao(factos,2,licao(factos,1,sim)).
licao(factos,2,licao(factos,1,nao)).
licao(atomos,1,licao(atomos,1,sim)).
licao(factos,2,licao(factos,2,sim)).
licao(factos,2,licao(factos,3,sim)).
              |     |          |___ apontador para a lição
              |     |
              |     |____ Nível de complexidade
              |
              |_____ Bloco Instrucional
```

Figura 3.8 Conjunto de lições percorridas num intervalo de tempo

Ao analisar este conjunto de lições percorridas pelo aluno no seu primeiro contacto, o Tutor-Prolog gera a hipótese de que a próxima lição (primeira do bloco dos objectivos) deverá ser formada de acordo com os conteúdos e a taxonomia prevista no nível de complexidade 2, pois foi este o preferido pelo aluno durante o estudo do bloco instrucional referente aos factos (generalização). Permite ainda gerar conhecimentos a respeito dos conceitos que o aluno adquiriu dentro do contexto instrucional dos factos, isto é, permite ao Tutor-Prolog concluir que o aluno conhece factos Prolog (sintaxe, semântica e pragmática) e que conhece a utilização de átomos Prolog no contexto instrucional dos factos.

No caso do exemplo da figura 3.8, a história da interacção indica que o aluno recebeu lições específicas sobre átomos. No entanto, mesmo que esse facto não estivesse presente na história, se o aluno aprendeu o conceito de factos, e como o conceito de átomos é parte do conceito de factos, então o conhecimento dos factos implica o conhecimento dos átomos nesse contexto instrucional (se um aluno conhece p e se q é um subconjunto de p então conhecer p => conhecer q num determinado contexto).

As hipóteses referem-se às acções que o actor$_1$ realizará guiado pelo conhecimento que possui sobre actor$_2$, ou devido aos conhecimentos que o actor$_2$ diz ter. As hipóteses assumem uma das seguintes formas:

```
-----------------------------------------------------------------
hipótese(actor ,<acção,conhecimento resultante da acção>) (1)
             1

hipótese(actor ,hipótese(actor ,<acção,
             1                  2
                conhecimento que provocou a acção>))     (2)
-----------------------------------------------------------------
```

O conhecimento, no caso do modelo do ensino, refere-se às acções do actor e é sempre verdadeiro em função de um contexto instrucional.
2

```
-----------------------------------------------------------------
          conhecimento(actor2,<conhecimento>))    (3)
-----------------------------------------------------------------
```

Em (1) e em (2) o actor representa o Tutor-Prolog e o actor , o aluno. A forma (1) é lida da seguinte maneira: o Tutor-Prolog tem uma hipótese de que deve tomar uma determinada acção e recebe o desempenho do aluno. Já a forma (2) tem a seguinte leitura: o Tutor-Prolog tem uma hipótese onde o segundo argumento é outra hipótese que descreve a acção, que o Tutor vai executar, em função dos conhecimentos que o aluno diz ter. Nesse segundo caso, a origem do conhecimento é externa.
1
2

As acções do Tutor-Prolog são o resultado da articulação entre as regras estratégicas e os conhecimentos existentes no modelo do aluno e no perito Prolog. Estas podem ser de:

- planear o ensino de acordo com o modelo do aluno;
- compor lições e exemplos de acordo com um nível de

complexidade, com a localização da lição ou do exemplo no plano, com o contexto instrucional e com os conhecimentos que representam as acções anteriores do aluno;

- guiar o aluno durante o processo de solução dos problemas;
- analisar as respostas e os exemplos escritos pelo aluno;
- adquirir novos conhecimentos e reescrever as regras de acordo com os conhecimentos adquiridos;
- simular problemas;
- resumir a apresentação dos conteúdos dos blocos instrucionais;
- desfazer as acções do aluno quando este o solicitar;
- salvaguardar os ficheiros de trabalho e
- interromper o tutorial se a idade e o desempenho do aluno for considerada insuficiente.

As acções do aluno podem ser as seguintes:

- alterar o plano de ensino de acordo com a sua vontade;
- inserir remover ou alterar dados;
- responder as questões propostas pelo Tutor-Prolog;
- escrever exemplos;
- solucionar os problemas propostos pelo Tutor-Prolog;
- realizar consultas e
- interromper o tutorial sempre que o desejar.

Assim, as hipóteses são utilizadas para decidir a respeito da aplicação das estratégias tutoriais ou para adequar o modelo do Tutor-Prolog ao modelo do aluno e vise-versa. As hipóteses, assim como os conhecimentos, também são escritos sob a forma de frases da Lógica de 1a. ordem.

---

hipótese(actor ,<<lição(x,y,h),conhecimento(c,o,n)>>) (1)
1

hipótese(actor ,hipótese(actor ,<<lição(x,y,h),
1 2
conhecimento(c,o,n)>>) (2)

A forma (1) é lida, genericamente, da seguinte maneira: "o actor
1
(tutor) tem uma hipótese de que necessita apresentar uma lição, que descreva os ensinamentos para um determinado conteúdo instrucional (x) e que deva ser formada de acordo com o nível de complexidade (y), que se encontra esquematizado no nó da árvore instrucional indicado por (h), para dar prosseguimento ao tutorial. O resultado da apresentação da lição é obtido através de **conhece**, que tem duas formas: a primeira refere-se a conhecimentos relativos ao desempenho no conteúdo tutorial e a segunda, a conhecimentos relativos ao desempenho na escrita de exemplos Prolog. No primeiro caso, **c** corresponde ao contexto instrucional (um dos 4 blocos instrucionais que compõem o tutorial), o corresponde à origem do conhecimento, isto é, se o conhecimento foi obtido através do tutorial ou se foi gerado a partir de informações externas ao tutorial, e **n**, ao nível onde o aluno aprendeu o conceito (1,2 ou 3). Caso a origem do conhecimento seja externa, este argumento ficará vazio.

Os argumentos **x** e **c**, e **y** e **n** podem ser diferentes. Isso ocorre sempre que, para a explicação de um determinado conteúdo, o Tutor-Prolog necessite navegar para outro bloco instrucional e,

eventualmente, para outro nível de complexidade. Por exemplo, se ao apresentar o conteúdo instrucional referente ao ensino dos objectivos o Tutor-Prolog necessite lembrar ao aluno o conceito de variáveis. Nesse caso, a configuração de "lição" pode, por exemplo, ser a seguinte:

lição(objectivos,2,lição(objectivo,3,não)

e a de "conhece", por exemplo,

conhece(variável,interna,1)

o que significa que o aluno aprendeu uma forma de utilização de variáveis através dos conteúdos instrucionais existentes no módulo relativo ao uso de variáveis e que, para isso, foram necessárias explicações detalhadas sobre o assunto. O nível de complexidade 1 representa o recurso à explicações pormenorizadas na apresentação de um conteúdo instrucional.

No segundo caso os argumentos do parâmentro "conhece" são os seguintes: exemplo escrito pelo aluno, erro detectado no exemplo e o texto contendo o diagnóstico. No Tutor-Prolog o diagnóstico contém as orientações para a correcção de um erro, que pode ser corrigido pelo Tutor-Prolog ou pelo aluno (ver Capítulo 4).

Os argumentos lição e conhece da forma (2) são lidos da mesma maneira que na forma (1). A diferença, como já foi dito, é que em (2) o Tutor-Prolog tem uma hipótese cujo segundo argumento é outra hipótese, que descreve a opinião do actor 2 a respeito do desenvolvimento do tutorial.

Internamente as hipóteses são organizadas em forma de pilhas e, quando um conjunto de hipóteses é verificado, dá origem a um conhecimento sobre o que o aluno já aprendeu ou já sabia, ou, então, gera uma nova hipótese de ensino e é removido.

Defendemos que durante uma sessão de ensino o planeamento e a composição das lições dependem do contexto histórico, isto é, das acções do aluno frente ao conteúdo já apresentado. Para isso introduzimos a função f (figura 3.9) que descreve a operação de união da história gerada durante a apresentação de um conjunto de lições(x,y,h) e que obteve como resultado um conjunto de conhecimentos$_{[i]}$(c,o,d) . O resultado da função é a forma (indicadores) de como a nova lição$_{[i]}$, a ser apresentada, deve ser seleccionada e composta.

$$f\left(\bigcup_{1\le i\le n} \left(\left\langle\left\langle \text{lição}(x,y,h), \text{conhece}(c,o,d)\right\rangle\right\rangle_{[i]}\right)\right) = \text{lição}(x,y,h)_{n+1}$$

Figura 3.9 Definição da próxima acção

O conjunto das lições e conhecimentos representam o desempenho do aluno ao longo dum determinado conjunto [i] de lições.

A função apresentada é necessária para o planeamento do próximo passo instrucional, que pode ser a revisão de um conteúdo anterior ou a apresentação de maiores detalhes a respeito dos conteúdos actualmente em estudo, e para a definição do nível em

que o próximo passo deve ser mapeado.

Como exposto, a apresentação do conteúdo instrucional pode ser guiada pelo Tutor-Prolog através das estratégias de ensino, mas pode, também, ser guiada directamente pelo aluno.

O aluno pode acompanhar o seu próprio desempenho no decorrer do curso, podendo optar por deter-se mais em determinados conteúdos e omitir módulos instrucionais relativos a outros conteúdos que julgue conhecer. Pode, assim, alterar, de forma directa o plano inicial do Tutor-Prolog, através dos menus que contém o mapa da evolução prevista para o tutorial. Por exemplo, se o aluno optar por estudar o conteúdo envolvido no bloco dos objectivos, omitindo o bloco dos factos, o Tutor-Prolog gera o seguinte conjunto de hipóteses:

```
hipotese(tutor,hipotese(aluno,p(licao(objectivos,2,
        licao(objectivos,1,sim)),conhece(atomos,externa,_)))) (1)

hipotese(tutor,hipotese(aluno,p(licao(objectivos,2,
        licao(objectivos,1,sim)),conhece(factos,externa,_)))) (2)
```

No exemplo, que é apresentado na figura 3.10, a acção do Tutor-Prolog traduz-se numa lição formada de acordo com o conteúdo instrucional para o ensino dos objectivos e com a complexidade que ele crê que o aluno esteja preparado para receber, de acordo com as hipóteses e os conhecimentos que possui naquel instante da interacção. No entanto, o aluno responde ao Tutor-Prolog num nível que este considera ser mais complexo. Esta

interacção vai resultar na alteração do modelo do aluno através da geração de uma nova hipótese a respeito do que o Tutor-Prolog pensa que é o conhecimento actual do aluno sobre Prolog e, consequentemente, resulta também na mudança do plano estratégico de ensino do Tutor-Prolog.

Segundo a vontade do aluno, a lição que deve ser apresentada corresponde ao ramo esquerdo do nó 1 da árvore dos objectivos.

Vê a seguinte frase interrogativa escrita em Português.

O João joga voleibol?

Vê, agora, a representação, desta mesma frase, em Prolog:

?- joga(joao,voleibol).

Entendeste o exemplo? >> opções: **sim** / não / executar / menu

Como a resposta foi "sim", o Tutor-Prolog passa a verificar a real compreensão do aluno através da solicitação de exemplos (através de analogias).

Escreve, agora, os teus exemplos. Quando não tiveres mais dúvidas escreve FIM.

-> ?- joga(joao,X).

Figura 3.10 Alteração da estratégia

O Tutor-Prolog está a espera de uma construção, no domínio dos objectivos, com argumentos do tipo átomo. Entretanto, o aluno escreveu um exemplo válido para o contexto esperado (objectivos), mas com um argumento do tipo "variável". Esta acção do aluno

provoca a geração de uma nova hipótese, que envolve a noção do uso de variáveis. A nova hipótese pode conter conhecimentos com origem diferentes consoante a história da interacção. Assim, se na história existirem referências ao ensino da utilização de variáveis, a origem do conhecimento é interna como em (4). Se, no entanto, não forem encontradas referências acerca da apresentação deste material instrucional, a origem será externa, como em (4').

```
hipotese(tutor,p(licao(objectivos,2,licao(objectivos,4,sim)),   (4)
                 conhece(variaveis,interna,2))))

hipotese(tutor,hipotese(aluno,p(licao(objectivos,2,              (4')
         licao(objectivos,4,sim)),conhece(variaveis,externa,_))))
```

Esta hipótese, independentemente da forma (4) ou (4'), causa a supressão de parte da apresentação do conteúdo instrucional no âmbito dos objectivos. Ou seja, ocorre uma nova mudança no plano inicial de ensino destinada a verificar a hipótese do Tutor-Prolog. Logo, o Tutor passa a testar os conhecimentos do aluno:

```
------------------------------------------------------------------
Represente a seguinte frase em Prolog:

                        Quem joga ténis?
------------------------------------------------------------------
```

Caso o aluno responda favoravelmente, a hipótese (4), pode transformar-se num conhecimento:

```
conhecimento(aluno,conhec(variável,interna,1,objectivos)))   (5)
```

onde o parâmentro "conhece", utilizado nas hipóteses, adquire

mais um argumento que denota o contexto instrucional em que o conhecimento foi originado.

Todavia, no decorrer da sessão a hipótese (4 ou 4') pode vir a ser considerada falsa, se a escrita de "X" pelo aluno, apesar de coincidir com uma forma sintáctico-semântica correcta em Prolog, for casual. Se isso ocorrer, nas futuras interacções o Tutor-Prolog perceberá que o aluno, na realidade, não conhece este conceito de variáveis no contexto dos objectivos e retrocede ao plano inicial de ensino.

Todo este processo de geração e verificação de hipóteses e de retrocesso na árvore instrucional provocará a geração de uma lição visando clarificar a utilização de variáveis dentro deste contexto, como é apresentado na figura 3.11.

Em Português temos frases interrogativas que admitem mais de uma resposta. Em Prolog acontece o mesmo. Vê o exemplo:

O que joga o João ?

```
?- joga(joao,X).
```

Entendeste o exemplo?

Figura 3.11 Retrocesso na árvore industrial

Resumindo, todo o processo de geração e manutenção do modelo do aluno ocorre, de forma esquemática, como apresentado na figura 3.12.

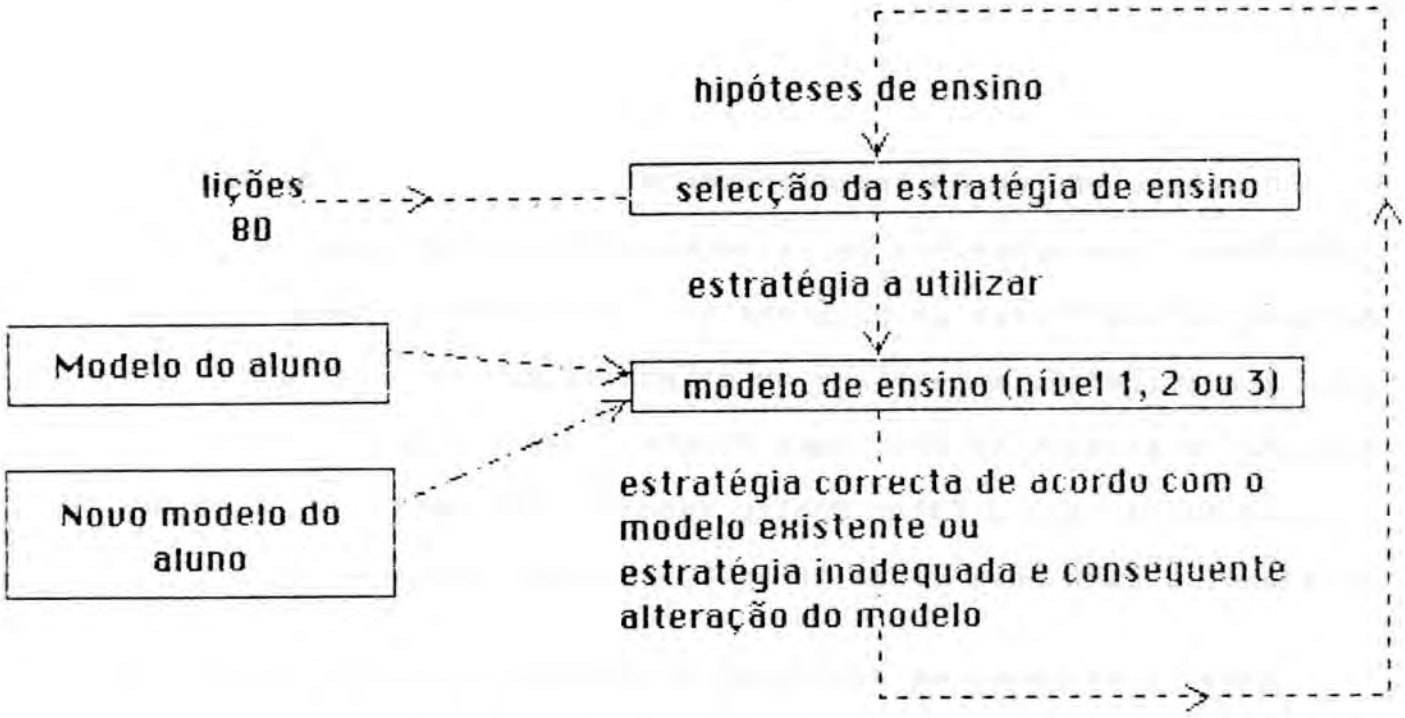


Figura 3.12 Operação de ensino do Tutor-Prolog

Até aqui o tempo esteve sempre implicitamente presente através do apontador h das lições e da referência ao conjunto de acções e resultados [i]. No entanto, o tempo necessita assumir uma forma explícita nas operações de resolução de um problema pelo Tutor-Prolog ou no processo de solução dos exercícios pelo aluno.

Assim, o diagrama descrito na figura 3.7 tem, ainda, uma outra dimensão (z), que representa o tempo necessário para a execução de uma ou mais operações responsáveis pela mudança de estado. Este tempo depende da forma como o aluno ou o Tutor-Prolog tentam solucionar os problemas pois num diálogo entre o Tutor e o aluno as hipóteses são dinâmicas, ou seja, uma hipótese é válida num determinado instante daí a necessidade de localizar as hipóteses no tempo.

O tempo é usado principalmente na detecção de falhas e confusões, nas operações de retrocesso e na orientação do aluno durante os exercícios de programação. Por exemplo, para a detecção e a correcção automática ou orientada das falhas, ocorridas durante a escrita de programas Prolog, são levados em conta os conhecimentos que o Tutor-Prolog adquire a respeito do programa do aluno, a cada nova linha de código por ele escrita.

Todo o processo de depuração é baseado nos instantes em que as acções acontencem, ou seja, num conjunto de estados. Cada estado é representado por um conjunto de proposições verdadeiras para aquele estado. Os estados são delimitados por uma operação predecessora e por uma operação sucessora. Assim, um intervalo de tempo é definido pelo conjunto de possíveis sequências de estados. Uma proposição que representa um evento associa a ocorrência deste evento com o estado em que ele ocorreu.

```
-------------------------------------------------------------------
          [ [<E1,op1,E2>] [<E2,op2,E3>] ...]
-------------------------------------------------------------------
```

Quando ocorre uma falha, o sistema localiza-a num determinado instante e verifica todas as operações geradas no intervalo de tempo, visando alterar o que for necessário. Verifica, ainda, a propagação dos efeitos gerados pela falha (ver exemplos 4.6, 4.8, 4.10 e 4.12, no Capítulo 4). Este conceito é importante para a depuração de programas.

No Capítulo 4 é tratada a meta-interpretação da linguagem

Prolog e são apresentados exemplos práticos de como as regras, as tabelas e as hipóteses são utilizadas na detecção de falhas ocorridas nos programas dos alunos.

No Capítulo 5 descreve-se como as regras, as tabelas e as hipóteses são utilizadas na análise das frases escritas pelos alunos.

4. INTERACÇÃO EM PROLOG

Neste Capítulo é apresentada a interacção em Prolog, com particular destaque para o ensino da sintaxe, da semântica (operacional e declarativa) e dos aspectos da pragmática, contemplados no subconjunto da linguagem Prolog que utilizamos [1].

A ideia geral é que o Tutor-Prolog saiba solucionar os problemas que propõem ao aluno.

---

1 No decorrer deste Capítulo utilizaremos a nomenclatura e as definições apresentadas por Sterling e Shapiro [STE;SHA 87] p. 1-9. Assim, um programa em lógica é constituído por termos e comandos. Existem três comandos básicos: os factos, os objectivo e as regras. Existe apenas uma estrutura de dados: o termo lógico.
Um **facto** representa uma relação existente entre objectos (por exemplo: pai(jose,pedro).). Neste texto, a relação será chamada de **predicado**. Quando os objectos são conhecidos (jose e pedro) são chamados de **átomos**. Caso contrário, temos os factos universais pois utilizam **variáveis** (por exemplo: gostar(X,laranja). Um conjunto finito de factos constitui uma forma simples de um programa em lógica.
Um **objectivo** serve para se obter informações a partir de um programa em lógica. Assim, através de um objectivo podemos conhecer se uma determinada relação é verdadeira considerando-se os factos existentes (?-pai(jose,pedro).). A resposta a um objectivo determina a consequência lógica de um programa. Podemos ter, ainda, uma conjunção de vários objectivos separados pela vírgula (',') - objectivos conjuntivos.
As **regras** são os comandos mais importantes de um programa em lógica. Elas são usadas para definir novas relações em termos das já existentes. Uma regra tem a forma genérica A:-B1,B2,...,Bn. Com n>=0 e onde A representa a cabeça da regra e os Bi's, o seu corpo. Tanto A como os Bi's são objectivos.
As regras, os factos ou os objectivos são também chamados de Cláusulas de Horn ou, simplesmente, cláusulas.

Também é apresentado o recurso à Lógica para ultrapassar as dificuldades advindas do controle exigido pela linguagem Prolog. É destacada a adaptação do Tutor-Prolog à interacção e a possibilidade de simulação da correcção e da execução dos problemas.

Os aspectos da interpretação da linguagem são tratados através do uso de um meta-interpretador escrito em Prolog, que contém as regras da linguagem Prolog (sintaxe, semântica e pragmática) e adquire conhecimentos a respeito do programa que o aluno escreve (modelo da interacção). O objectivo principal é a emissão de um diagnóstico completo, sempre que sejam detectados erros ou confusões no programa do aluno que levem o sistema a intervir para efectuar correcções, orientações ou apenas apresentar alertas.

O diagnóstico é construído dentro dos diversos módulos e fornecido ao aluno através do material instrucional. Ou seja, o diagnóstico vai sendo formado de acordo com os resultados da análise feita pelo meta-interpretador, de acordo com o contexto instrucional em que o tutorial está se desenvolvendo e, ainda, de acordo com o modelo da interacção.

O meta-interpretador do Tutor-Prolog tem como principal finalidade o tratamento das interrupções provocadas por falhas ocorridas durante a escrita de programas. Quando uma falha é detectada, estão previstos dois tipos de acções: a falha é corrigida pelo Tutor-Prolog que utiliza para isso as regras que descrevem a linguagem e as informações adquiridas durante a interacção e, é solicitada a intervenção do aluno para auxiliar

na correcção pois não possui regras ou conhecimentos suficientes para realizar a correcção.

Eventualmente, os conhecimentos provenientes da interacção são nulos. Esta situação é difícil de ocorrer, pois de uma sessão para a outra ficam armazenados os conhecimentos relevantes para a intervenção seguinte. As actividades da própria sessão, naturalmente, são sempre as mais importantes (ver Capítulo 3).

De um modo geral, o **diagnóstico** é composto pelas mensagens que apontam o motivo da falha, pela regra que foi utilizada para a detecção da falha ou da inadequação do exemplo, pela indicação de como o erro pode ser corrigido dentro do contexto em que ocorreu, pela correcção automática ou pela geração de um novo exemplo feita pelo Tutor-Prolog e, finalmente, pela regra que era esperada. Portanto, todo o exemplo que o aluno escreve é inserido dentro de um contexto (lição apresentada ou sequência lógica de desenvolvimento de um programa). Logo, um exemplo pode estar sintacticamente correcto e ser considerado inadequado num determinado contexto de trabalho.

No exemplo 4.1 é apresentada a geração automática da frase exemplo (Português) e do exemplo Prolog (objectivo); a interacção com o aluno, que engloba a escrita de um exemplo inadequado; o diagnóstico; e o reforço tutorial, com a geração de novos exemplos adequados ao contexto da lição actual.

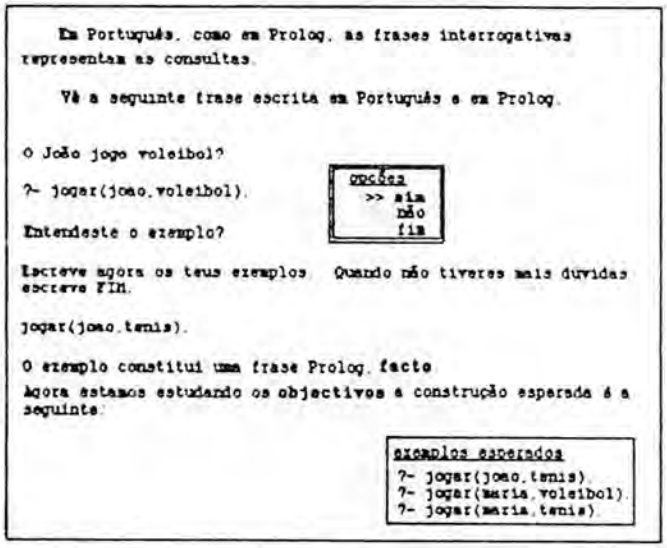


Figura 4.1 Exemplo correcto em contexto inadequado

Como já foi dito, as falhas, detectadas durante a meta-interpretação, podem ser corrigidas coloquialmente ou automaticamente. Este procedimento permite a edição do código do programa ao nível da meta-interpretação. Permite, ainda, a interrupção temporária do processo de meta-interpretação de um programa ou de um objectivo, para que novas informações sejam introduzidas, a fim de que uma determinada unificação se verifique.

4.1 Descrição do meta-interpretador

Segundo Brown [BRO 79], todo o interpretador divide-se em dois grandes módulos: o de análise do texto de entrada e o de execução dos procedimentos. O meta-interpretador Prolog não foge à esta regra geral.

A sintaxe da linguagem Prolog é extremamente simples, o que torna o anlalisador sintático fácil de ser implementado. Tanto os

comandos como os dados são delimitados pelo caracter ponto (".").
O texto Prolog pode estender-se por várias linhas e ser escrito em formato livre. Estes factores simplificam as rotinas de recuperação de erros ao contrário do que ocorre com a maioria das linguagens de programação que possuem palavras reservadas, margens, caracteres de continuação, etc.

O analisador léxico também não apresenta surpresas. Os "tokens" são delimitados por brancos ou pelo fim da linha ou, ainda, pelo aparecimento de um caracter que não possa fazer parte do "token" que está sendo identificado.

Como estamos no nível da meta-interpretação, no que se refere a codificação, optamos por trabalhar com textos quase sem modificações.

A execução de um programa Prolog se baseia em duas operações princiapais: a unificação e o retrocesso.

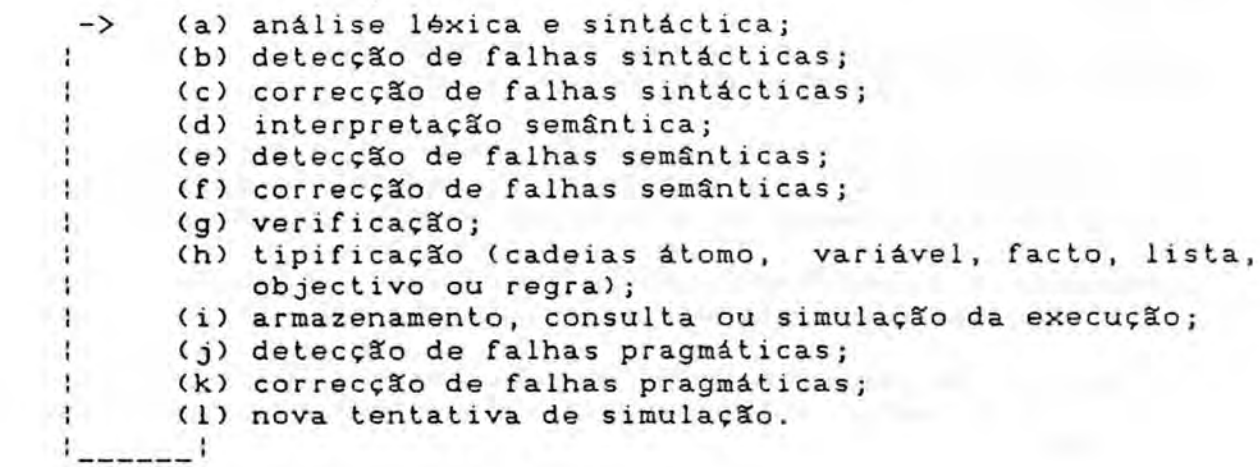
Segundo Yokota [YOK 83], a unificação consiste na localização da cláusula chamada, na busca dos argumentos da cláusula chamada e do objectivo que está sendo tentado, e no exame da unificação dos argumentos.

Quando é solicitada a satisfação de um objectivo, a meta-interpretação deve apenas procurar uma cláusula cuja cabeça possua o memsmo predicado e a mesma aridade do objectivo. Localizada a cláusula com as características requeridas é preciso buscar seus argumentos. As variáveis presentes na cláusula pos-

suem uma abrangência de tipo local. A estrutura utilizada para representar as variáveis no meta-interpretador possui a forma de uma lista. A lista é composta pela referência à variável e o código que lhe foi atribuído. Após a satisfação do objectivo a lista é eliminada. Nos casos em que o argumento é uma estrutura (cláusula composta), optou-se pela cópia da estrutura em uma área de trabalho [MEL 82].

No que se refere ao retrocesso, o meta-interpretador Prolog permite a sua execução normal (controlada pelo algorítimo do interpretador Prolog) ou a opção por um processo de optimização (retrocesso inteligente). No primeiro caso, quando ocorre a falha na satisfação de um objectivo, o controle da execução deve retroceder até a última opção assinalada, desfazendo todas as associações de variáveis executadas desde então. No segundo caso a ideia é de criar um mecanismo que evite a exploração de opções que não levem a satisfação do objectivo. Assim, quando um predicado falha, deve ser tentada primeiro uma alternativa que não o chame novamente. A manutenção do histórico da execução é feita através de uma tabela que indica quais as variáveis que foram instanciadas e o instante de seu instanciamento, de um conjunto de cláusulas contendo o predicado, o número de argumentos, organização dos argumentos na cláusula que foi instanciada, instante do instanciamento, valores utilizados no instanciamento, alternativas para cada instância e ordem de escrita das cláusulas, de uma pilha contendo os factos Prolog que indicam os predicados que foram satisfeitos e da hipótese de retrocesso.

O processo de análise léxica, sintáctica, semântica e pragmática pode passar pelos seguintes passos:



A exemplo de outros sistemas tutoriais (Tutor LISP [AND 82], Tutor BASIC [BAR 76] e Tutor LOGO [MIL 82]), o Tutor-Prolog também possui características de ambiente para o ensino de linguagens de programação.

As informações utilizadas no ambiente de resolução dos problemas estão organizados em forma de hipóteses, factos Prolog, regras DCG (para a meta-interpretação das regras da linguagem que pode ser feita na forma "left-right" ou "right-left"), contadores de erros, além das estruturas já referidas para o tratamento do retrocesso. Durante a meta-interpretação é realizada, ainda, a tipificação das cláusulas em átomo, variável, cadeia, lista, facto, objectivo ou regra. Essas medidas, para além de auxiliarem o aluno na aprendizagem, visam um diagnóstico mais apurado das eventuais falhas.

Em linhas gerais, o meta-interpretador pode ser descrito

através do esquema da figura 4.2 seguinte.

```
Entrada: um Programa P ou um objectivo G
Saída:  solução de P, instâncias de G ou falha
Algorítmo:

  iniciar o solucionador com uma cópia de G, G'
  dividir G' em cláusulas G'1, G'2, ..., G'i

  se a execução estiver seleccionada com a opção retrocesso inte-
    ligente, gerar a lista com todas as instâncias possíveis para
    os predicados/número de argumentos envolvidos em cada G'i

    enquanto o solucionador não estiver vazio, e existirem dados,
      instanciar G'i e guardar as instâncias obtidas para cada G'i

      se G'i  pertencer a um G' do tipo regra e G'i não puder ser
        instanciado,  então congelar G'i e prosseguir a  execução
        em G'i+1

      se o  solucionador estiver vazio e faltarem dados para  que
        as  variáveis congeladas possam ser instanciadas,  ou  se
        G'i  pertencer  a  um G' do tipo objectivo e se  G'i  não
        puder ser instanciada,

        então emitir o diagnóstico e esperar pela intervenção  do
          aluno

          se o aluno introduzir novos dados,  prosseguir a execu-
            ção a partir de G'i

            se não

              se retrocesso inteligente,  retroceder ao  instante
                que provocou a falha

              se não retrocesso normal

          se não existirem mais dados, falhar a execução

          se faltarem  cláusulas  para que o programa  possa  ser
            executado,  emitir o diagnóstico e aguardar a  inter-
            venção do aluno

            se o  aluno  introduzir novas cláusulas  reiniciar  o
              solucionador

            se não, falhar a execução

  Se o solucionador estiver vazio a execução obteve sucesso.
```

Figura 4.2 Algorítmo de meta-interpretação

4.2 Análise sintáctica

Para Boulay e Sothcott [BOU;SOT 80] a sintaxe constitui o aspecto central de um Tutor que pretende ensinar uma determinada linguagem de programação.

No Tutor-Prolog a detecção e a correcção automática ou orientada dos erros sintácticos visa ajudar o aluno a melhor fixar a sintaxe da linguagem. Com esse objectivo, e como estamos trabalhando ao nível da meta-interpretação, são considerados como válidos certas cláusulas cujas construções o interpretador Arity/Prolog não aceita.

As informações de naturesa sintáctica, que o Tutor-Prolog adquire durante a interacção, referem-se ao controle do número de parêntesis, colchetes e chaves, ao uso de operadores (aritméticos e lógicos) e às restrições envolvidas na sintaxe da lista.

O processo de detecção e de correcção dos erros cometidos pelo aluno na escrita das clásulas envolve a determinação da zona da cláusula onde ocorreu o conflito; a geração das hipóteses para a reescrita da zona de conflito com base nas regras da linguagem e nos conhecimentos gerados a partir das informações adquiridas durante a interacção; os testes das hipóteses com o auxílio do aluno (erros semânticos e pragmáticos); a substituição, na cláusula inicial, da zona de conflito pela nova composição; e, a reinterpretação da clásula.

No Tutor-Prolog, a forma geral para a análise de uma cláusula Prolog possui a seguinte estrutura:

```
-------------------------------------------------------------------
    termo(Termo_analisado,Lista_de_variáveis_inicial,
          Lista_de_variáveis_resultante,Termo_escrito,Prioridade).
-------------------------------------------------------------------
```

Na lista de variáveis estão armazenadas a representação sintáctica e a representação Prolog de uma variável (endereço). Estas informações auxiliam a apresentação visual dos exemplos que compõem as lições, das simulações e do procedimento "espiar". Auxiliam ainda na interpretação (declarativa e operacional) dos objectivos e das regras.

```
lista[A : B,...]
      ¦   ¦
      ¦   +-> endereço
      ¦
      +-> cadeia que representa a variável
```

A tipificação está ligada às prioridadades definidas no meta-interpretador e é feita durante a análise, isto é, o Tutor-Prolog gera hipóteses sobre o tipo que a cláusula pode pertencer, de acordo com as regras que vão sendo utilizadas. Veja-se, por exemplo, a análise de:

```
a(x,y):-b(x,y),b(y,x).
```

onde "a" e "b" são predicados e "x" e "y", átomos. A cláusula a(x,y) é considerada como do tipo facto até que o operador ":-"

seja encontrado. A partir desse instante passa a ser classificada como do tipo regra, sendo então analisada de acordo com as exigências sintácticas, semânticas e pragmáticas do actual tipo. Para além da utilidade no ensino das listas, dos átomos e das variáveis, a tipificação também é utilizada no sentido de "saber o que fazer" com a clásula analisada, ou seja, tratando-se de um facto ou de uma lista, deve ser memorizado; tratando-se de um objectivo, deve ser satisfeito e tratando-se de uma regra, deve ser executada. A execução de uma regra requer cuidados específicos, pois a sua cabeça só deve ser instanciada após o seu corpo estar completamente instanciado.

Apresentamos em seguida quatro exemplos de erros sintácticos detectados e corrigidos directamente pelo Tutor-Prolog, ou com o diagnóstico do Tutor-Prolog e a intervenção do aluno (figuras 4.3 e 4.4)

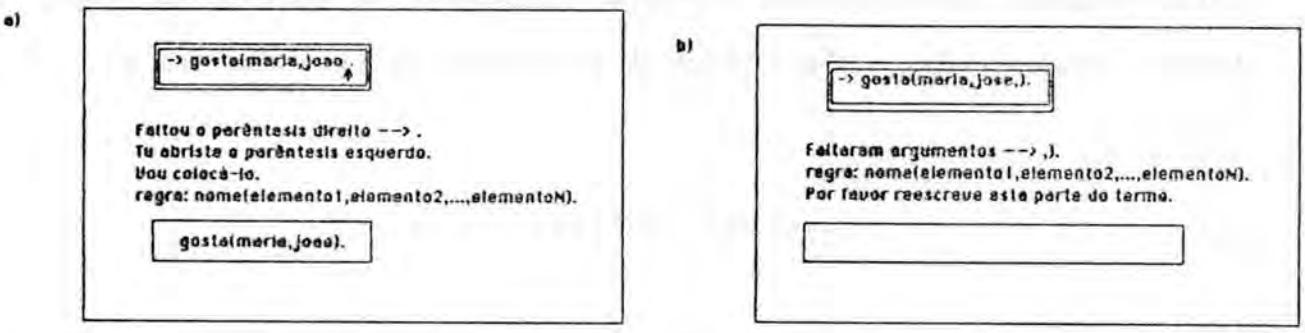


Figura 4.3 Detecção de falhas sintácticas

Durante a análise da cláusula são geradas até 5 listas contendo os "tokens" que a compõem (os "tokens" são divididos em: nome, número, variável, anónima, string e pontuação). Assim, ao iniciar a análise, existe uma única lista contendo todas as partes que compõem a clásula. Supondo o exemplo "a)" da figura 4.3.

```
L_token = [gosta,(,maria,joao,.]
```

A medida que as partes vão sendo analisadas, é gerada uma nova lista.

```
                    instante 1    _____2______
L_token_analisados = [gosta,(,maria,joao]
```

No momento em que é detectado um conflito, isto é, o recebimento de um "token" diferente do esperado pelas regras do meta-interpretador, é gerada uma outra lista contendo a partícula que gerou o conflito. Ou seja, o instante 2 da análise foi interrompido sem que uma operação prevista de mudança de estado fosse encontrada. Num instante podem ser análisados um ou mais "token".

```
L_token_conflito = [.]
```

Os elementos que ainda restarem da lista inicial, após a detecção do conflito, são colocados em outra lista, ou seja, a lista com os elementos que não foram analisados.

L_token_resto = [.]

Após a divisão tem início a geração das hipóteses para a correcção do conflito. As hipóteses são geradas a partir das regras do meta-interpretador e das informações adicionais que vão sendo adquiridas durante a análise de uma cláusula particular ou durante a interacção como um todo.

No caso do exemplo "a)" a regra é "predicado(argumento1,

que assinala a existência de um parentesis direito.

Com base nestas informações é gerada a hipótese de que antes da partícula ".", último elemento da lista inicial, deve ser acrescentada uma partícula contendo um parêntesis esquerdo.

L_token_acrescentar = [)]

Após a operação de correcção, caso a lista L_token_resto contenha o caracter "." a análise sintática termina com sucesso, caso contrário, a lista inicial é iniciada com a nova cláusula (corrigida). Neste caso, a cláusula corrigida é o resultado da união das listas que contém os "tokens" analisados, o "token" que deve ser acrescentado o os "tokens" que ainda estão por analisar.

A correcção do erro talvez tenha de ser feita pelo aluno, pois o Tutor não possui regras e/ou não gerou os conhecimentos

necessários para sair do impasse. Nesse caso, é feita a junção da parte da cláusula já analisada com a correcção proposta pelo Tutor ou pelo aluno. Caso o aluno opte por não corrigir a cláusula, o processo de análise é abandonado, ficando o Tutor a espera de una nova intervenção. A acção de abandono pode implicar o retrocesso a instantes anteriores, a fim de que informações recolhidas sejam eliminadas da história da interacção.

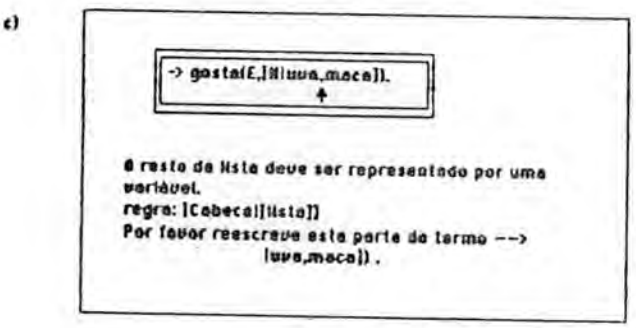


Figura 4.4 Correcção de erros sintácticos - listas

A lista Prolog pode ser assumida como um caso particular de uma expressão funcional. Isso permite que construções como [lista] sejam aceitas dentro do contexto instrucional destinado ao ensino da contrução lista. O controle desta situação é facilitado devido ao facto de o Tutor-Prolog manter informações a respeito do contexto instrucional activo e da tipificação das cláusulas. Assim como a lista, a escrita de X é reconhecida como válida dentro do contexto instrucional das variáveis.

4.3 Interpretação semântica

Trataremos, agora, dos aspectos da semântica declarativa e operacional do subconjunto da linguagem Prolog considerado pelo Tutor-Prolog.

Para a meta-interpretação semântica e pragmática, a simulação é uma componente indispensável, pois através dela é garantida a continuidade de uma operação. Ou seja, quando o meta-interpretador não tem condições de eliminar o motivo da falha, expõe a situação ao aluno e orienta-o para que forneça novas informações (factos ou regras), visando a criação de novas alternativas para a solução do problema. Após o processo de aquisição de novas informações ou regras, a simulação prossegue do ponto (instante) onde havia ocorrido a interrupção. Assim, durante a simulação é permitida a mudança do tópico do diálogo para ser feita a aquisição de novas informações, após o que é retomado o tópico inicial [COE 79].

a) Semântica declarativa

As informações sobre a semântica declarativa relacionam-se com os predicados declarados no programa e com o número de argumentos associados a cada símbolo. No instante zero da interacção, ou seja, quando o aluno inicia a escrita de seus programas, o

Tutor-Prolog não possui conhecimento contextual. A medida que o aluno vai escrevendo os seus programas, o modelo da interacção Prolog vai sendo ocupado com informações relativas ao contexto ("background") da interacção. O modelo da interacção gerado permite detectar contradições entre as informações declaradas e a forma como são consultadas.

Assim, se o aluno estiver escrevendo o programa "membro", pode acontecer a seguinte situação:

```
membro(X,[H|T]).
```

Neste instante o Tutor-Prolog armazena informações sobre o predicado "membro". São elas, o predicado "membro" e o número de argumentos associados a este predicado "2". Estas informações compõem o "dicionário de programas":

```
-----------------------------------------------------------------

                        [membro:2].

-----------------------------------------------------------------
```

Se, entretanto, durante a escrita da segunda cláusula do programa membro o aluno cometer um erro, o Tutor-Prolog vai procurar corrigí-lo de acordo com o conhecimento acumulado até o momento (regras sintácticas definidas ao nível do meta-interpretador e as informações que já possui a respeito do predicado "membro"), como se exemplifica em seguida na figura 4.5.

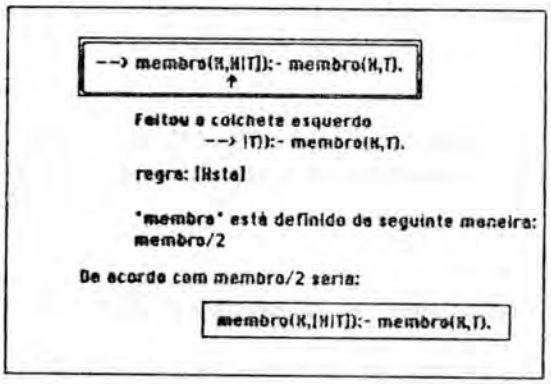


Figura 4.5 Utilização de conhecimentos dinâmicos para a correcção de erros semânticos

---

Durante o processo de análise e de correcção deste exemplo, foram geradas as seguintes listas:

```
L_token = [membro,(,X,H,|,T,],),:-,membro,(,X,T,),.]
L_token_analisados = [membro,(,X,H]
L_token_conflito = [|]
L_token_resto = [T,],),:-,membro,(,X,T,),.]
```

Quando, no instante 2 da análise, for encontrada a partícula "|", o Tutor gera duas hipóteses: a de que a cláusula em análise é uma lista e a de que faltou o colchete direito.

Adicionalmente, existe a informação de que durante a interacção o predicado "membro" já foi utilizado e que a ele estão associados dois argumentos, bem como a informação de que no instante 2 da análise, antes da detecção do conflito, foram encontrados 2 argumentos.

De posse destas informações, o Tutor gera a primeira hipótese de correcção da falha. Esta hipótese é considerada "forte", pois é suportada pelo modelo da interacção.

```
hipótese(tutor,p(correcção(membro(X,[H|T]:-membro(X,T).)),
      conhecimento(!),[membro:2],membro(X,H|T]):-membro(X,T).))
```

Após o aluno confirmar a hipótese do Tutor, a lista inicial recebe a cláusla corrigida e o processo de análise é reiniciado.

Após a escrita deste exemplo as informações, que permanecem no sistema, são as seguintes:

```
Lista de variáveis: [X:_0028,H:_0029,T:_0030]
Tipo(regra)
Dicionário de programas: [membro:2]
```

Caso o aluno discordasse da correcção, o Tutor geraria, ainda, a hipótese de que, neste caso ao predicado "membro" estaria associado apenas um argumento. Esta última hipótese, considerada como "fraca", seria baseada unicamente nas regras de análise da estrutura lista.

Caso existissem informações que indicassem a utilização de números diferentes de argumentos, associados a um mesmo predicado, o Tutor-Prolog passaria a simular todas as alternativas de correcção possíveis. A cada alternativa gerada, seria solicitada a intervenção do aluno para confirmá-la, ou não [STE;SHA 87].

Considerando o modelo existente sobre os programas do aluno e a escrita de uma nova cláusula, como é apresentado na figura 4.6, os passos seguidos pelo Tutor-Prolog na tentativa de correcção da falha ocorrida seriam:

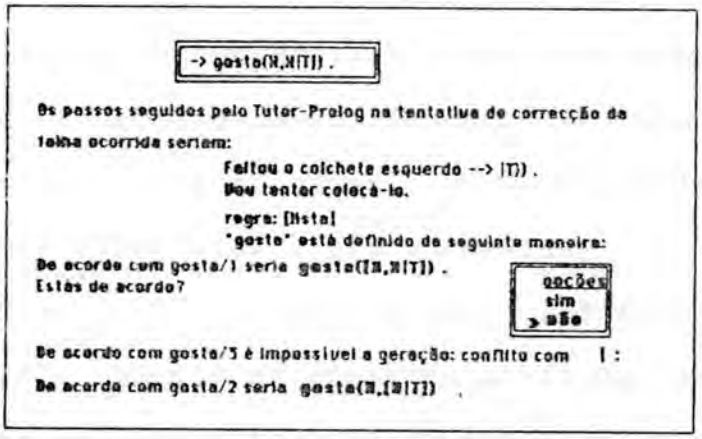


Figura 4.6 Tentativas de solução de um erro

As seguintes informações foram utilizadas para a realização do diagnóstico do exemplo da figura 4.6:

```
lista de variáveis [X:_0031,T_0032]
tipo(regra)
Dicionário de programas [gosta:1:3:2]
hipótese(tutor,p(correcção(gosta([X,X|T])),conhecimento(|)),
                                    [gosta:1:2:3],gosta(X,X|T])))
hipótese(tutor,correcção(gosta(X,[X|T])),conhecimento(|)),
                          [gosta:1:3:2],gosta(X,X|T])))
hipótese(tutor,p(corrcção('e impossível a geração:conflito com |'),
                 conhecimento(|)),[gosta:1:3:2],gosta(X,X|T])))
```

As duas primeiras hipótese são cinsideradas fortes pois,

estão de acordo com as regas da linguagem e com as informações obtidas da interacção. A última é considerada como fraca pois a tentaiva de correcção, feita pelo Tutor, entra em conflito com as regras da linguagem. Caso a cláusula que o aluno deseja introduzir deva conter três argumentos este deve reescrevê-la.

As informações do dicionário de programas são utilizadas ainda nas mensagens que notificam a escrita de cláusulas com o mesmo predicado, mas com números diferentes de argumentos. Nesses casos, a história da utilização do predicado é apresentada como diagnóstico e, juntamente com a informação referente ao tipo da cláusula em questão, leva o Tutor a, se a cláusula for do tipo facto apenas emitir a mensagem de alerta. Se, no entanto, a cláusula for do tipo objectivo duas hipóteses podem ser geradas: a hipótese de que faltam dados para que o objectivo possa ser satisfeito e a hipótese de que a cláusula possui argumentos a mais ou a menos do que deveria ter.

Em ambos os casos, o Tutor-Prolog oferece orientação a respeito do motivo da falha, que no caso é a impossibilidade de unificação do objectivo. Contudo, o Tutor-Prolog não abandona a execução do objectivo: suspende a interpretação, apresenta o diagnóstico da falha e, se possível, toma uma decisão com base nos conhecimentos que possui. Quando isso não e possível a decisão envolve a participação directa do aluno.

No caso de o aluno confirmar a hipótese de que o motivo da falha é a inexistência de informações que permitam satisfazer o

objectivo, é dada ao aluno a oportunidade de fornece-las. Como os passos (instantes) da execução são mantidos durante a entrada das informações, ela será retomada a partir da cláusula onde ocorreu a falha (as unificações feitas com sucesso são mantidas na memória até o final da execução do problema - ver figura 4.7). Caso o aluno não queira fornecer novas informações, a execução é abandonada.

b) Semântica operacional

As informações utilizadas na semântica operacional localizam-se na lista de variáveis criada durante a análise sintáctica, no tipo atribuído a cada cláusula, no dicionário de programas gerado durante a semântica declarativa e nas informações sobre o retrocesso geradas durante a semântica operacional.

As informações que apoiam a simulação, quando esta utiliza o retrocesso inteligente[2], estão organizadas num conjunto de cláusulas compostas de cinco argumentos. A cada variável instanciada corresponde uma nova cláusula.

---

2 A possibilidade da utilização do retrocesso inteligente durante a solução dos objectivos prende-se, apenas, com razões pedagógicas, isto é, os alunos demonstraram compreender melhor o percurso inteligente realizado na tentativa de solução dos objectivos.

```
-----------------------------------------------------------------
data(variável,instante_do_instanciamento,[alternativas_para_unifi
     caçãoj,valor_utilizado,cláusula)
                              .........
-----------------------------------------------------------------
```

As informações referem-se às variáveis utilizadas numa determinada consulta, ao instante de seu instanciamento, ao valor utilizado no instanciamento, às possibilidades de retrocesso e à cláusula que a variável em questão está associada. A sua utilização é feita juntamente com os factos referentes às unificações já realizadas e as informaçõeso referente ao instanciamento que provocou a falha.

Assim, visando a optimização do mecanismo de retrocesso do Prolog, ao falhar a execução de um objectivo durante a execução de um programa, o controle activa o mecanismo de retrocesso que busca a última alternativa marcada. Como nem sempre a reunificação da última alternativa marcada elimina o motivo da falha, foram gerados conhecimentos relativos à execução, de maneira que tal situação seja reconhecida e sejam seleccionadas apenas as alternativas que evitem a repetição da falha [PER;POR 79][3], a fim de evitar os processos improdutivos de reunificação. A seguir descreveremos, em linhas gerais, à forma como implementámos e utilizámos alguns conceitos básicos desta abordagem. As

---

3 São mantidas informações apenas para um ponto de retrocesso. Não consideramos, nesta implementação, o caso de unificações das variáveis livres.

principais diferenças dizem respeito a forma de programação que, no caso do Tutor-Prolog é feita, como vimos, através do armazenhamento de informações em forma de cláusulas. A análise das informações leva em consideração o conjunto das cláusulas. Este processo, apesar de realizar um número maior de acessos à memória, diminui o tempo global da meta-interpretação.

Seguindo esta ideia, dotamos o Tutor-Prolog de capacidade para realizar o retrocesso inteligente. Assim, a execução retrocede directamente ao ponto (instante) de unificação da variável que provocou a situação de falha. Como não tratamos o caso das variáveis livres só existe um ponto de retrocesso.

Caso existam outras cláusulas com possibilidades de unificação com o objectivo, esta existência é assinalada e o controle do retrocesso é desviado para elas. Caso não existam possibilidades alternativas, o último entre os pontos de retrocesso associados às variáveis instanciadas antes da última unificação é escolhido. Esta será, então, a unificação associada às variáveis recém-instanciadas.

Através da figura 4.7 é possível acompanhar a geração do histórico da execução, e da geração da hipótese para o retrocesso.

```
                                                    factos existentes:
                                                      a(5).   a(7).
                                                      b(5,8). b(7,8).
instantes:  1      2      3     4                     c(3).   c(8).
       ?- a(X), b(X,Y), c(Z), d(X).                  d(7).

Informações  geradas para orientar o retrocesso:

                          1o.
data(X,1,[7],5,a(X)).                    fact(a(5)).
data(X,2,[7],5,b(X,Y)).
data(Y,2,[8],8,b(X,Y)).                  fact(b(5,8)).
data(Z,3,[8],3,c(Z)).                    fact(c(3)).

falha na tentativa de unificação de d(5).

motivo_da_falha(a(5)).

                          2o.
data(X,1,[],7,a(X)).                     fact(a(7)).
data(X,2,[],7,b(X,Y)).
data(Y,2,[],8,b(X,Y)).                   fact(b(7,8)).
permanece o mesmo
data(X,4,[],7,d(X)).                     fact(d(7)).

unificações realizadas para a resposta: X=7 Y=8 Z=3

          1o.                                2o.

a(5).  b(5,8).  c(3).      a(7).  b(7,8).  d(7).
```

Figura 4.7 Passos no retrocesso inteligente

Quando ocorreu a falha, houve o retrocesso para o instante 1, onde havia sido instanciada a variável "X". O ponto de retrocesso é obtido através do conhecimento do motivo da falha "a(5)", que, por sua vez, é obtido através da consulta as cláusulas "data". Como a variável "X" também é utilizada no instante 2, o retrocesso teve que ser feito também em "b(X,Y)". Logo, a pesqui-

sa dos factos necessita ser indexada pela variável em questão e pelo instante do seu instanciamento.

A manutenção duma lista (3o. argumento do facto) com as possibilidades de reunificação deve-se à necessidade de conhecer, com antecedência, se existe ou não a possibilidade de uma nova unificação. A manutenção da cláusula, à qual a variável está associada, também está ligada à apresentação do diagnóstico, que reconstitui os instantes anteriores a ocorrência da falha, visando a melhor compreensão do aluno.

A utilização destes conhecimentos, para a orientação e a detecção de falhas relativas à semântica operacional, pode ser vista através dos dois exemplos nas figuras, 4.8 e 4.9. Na execução de um objectivo, estes conhecimentos permitem a realização do diagnóstico do motivo pelo qual o objectivo não pôde ser satisfeito.

Figura 4.8 Utilização do retrocesso inteligente

Observando a execução do programa anterior com a opção "espiar", figura 4.9, pode-se verificar como os conhecimentos gerados durante a interacção são utilizados para a apresentação visual da execução aos alunos. Neste caso, já utilizando o retrocesso orientado.

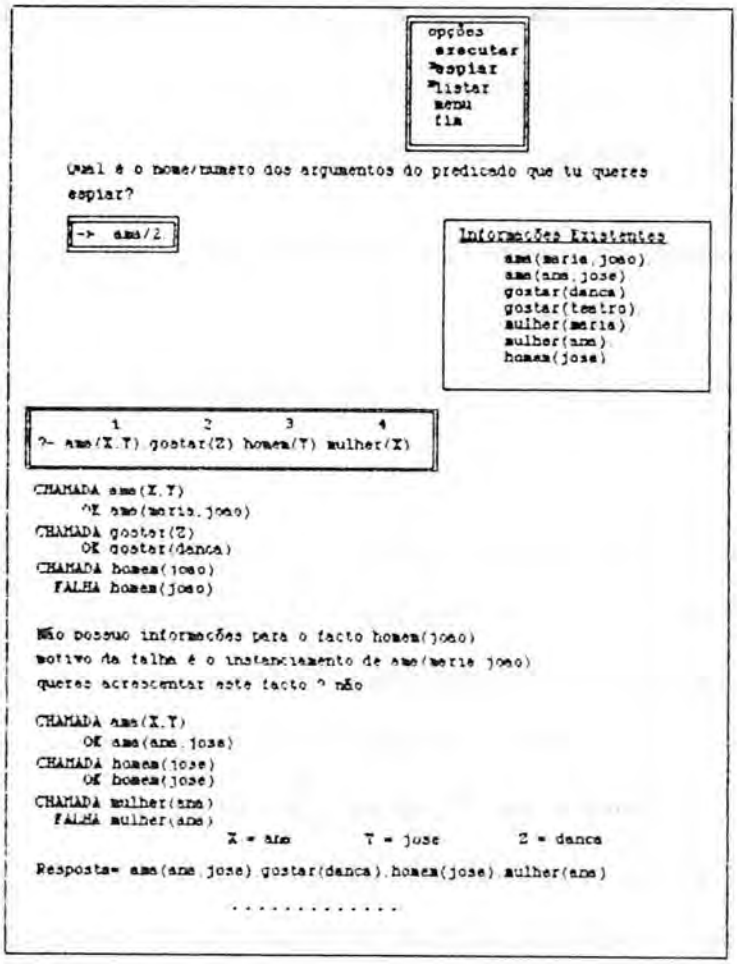


Figura 4.9 Opção espiar

4.3 Pragmática

A actuação do Tutor-Prolog no campo da pragmática resume-se aos aspectos que dizem respeito à metodologia para escrever programas Prolog e à forma de como alterá-los quando estão organizados incorrectamente, ou seja, a reorganização das regras, a alteração do plano de execução projectado pelo aluno e até a modificação de parâmetros definidos no problema (exemplos (4.10 e

4.12). Estes aspectos dizem respeito à semântica operacional e à pragmática. Durante a interacção o aluno, naturalmente, acompanha as operações executadas pelo Tutor [VIC 87].

No decorrer do trabalho experimental com os alunos notámos que os seus erros mais frequentes, na área da pragmática, ocorriam no controle, ou seja, na componente não lógica do Prolog [KOW 79a].

Visando suprir parte destas deficiências e, assim, aumentar a flexibilidade do Tutor-Prolog, foi desenvolvido um mecanismo para o "congelamento de cláusulas", inspirado na implementação do Prolog II [COL 82]. Isto possibilita que um programa seja executado segundo o aspecto da Programação Lógica, deixando de lado a ordem exigida pela linguagem Prolog. Posteriormente, é emitido o diagnóstico explicando a ordem exigida na linguagem Prolog, como é apresentado no exemplo 4.12.

A necessidade de ordenar a escrita das cláusulas dentro de um programa é um dos aspectos que os alunos não percebem de imediato. Consideremos então o seguinte exemplo:

```
-> soma(S):- S is X+Y, valor1(X), valor2(Y).
```

Esta organização representa a hipótese do aluno para a solução do problema soma, em que a ordem das cláusulas é inapropriada se tivermos em conta a linguagem Prolog. O meta-interpretador tem condições de resolvê-lo satisfatoriamente, utilizando o congelamento de cláusulas.

Ao tentar a unificação de S, o meta-interpretador verifica que não há possibilidades de unificação, pois as variáveis X e Y não foram ainda instanciadas. Gera então a hipótese de que a ordem das cláusulas dentro do programa está inadequada e passa a tentar verificá-la. Assim, a unificação da cláusula S is X+Y é congelada e o processo de execução segue na próxima cláusula. Quando "X" e "Y" forem unificadas, a execução retorna à cláusula (instante) que se encontra em estado de espera para efectuar o seu instanciamento.

A indicação do congelamento é mantida numa estrutura (congelados) que contém as variáveis que não estão instanciadas. Para cada variável são mantidas as seguintes informações: a cadeia que a representa, o seu endereço, o instante do congelamento e a cláusula a que pertence. Considerando o problema soma temos:

```
-----------------------------------------------------------------
     congelados([[S:_212,1,S is X+Y],[X:_213,1,'S is X+Y'],
                 [Y:_214,1,'S is X+Y']])
-----------------------------------------------------------------
```

Após o congelamento, a execução prossegue no instante 2, onde a cláusula "valor1" é instanciada e X é retirada da lista. No entanto, a cláusula S is X+Y não pôde, ainda, ser resolvida e a execução segue no instante 3, quando Y é instanciada, sendo retirada da lista. Agora, a cláusula congelada (instante 1) pode ser unificada e o problema soma, executado com sucesso, como mostra a figura 4.10.

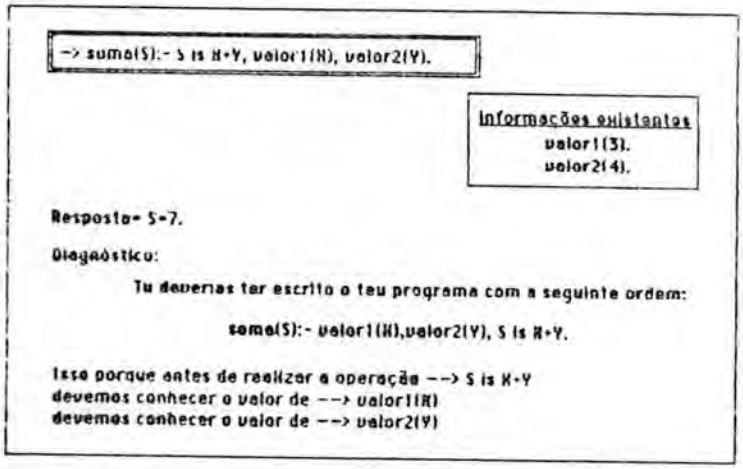


Figura 4.10 Congelamento de cláusulas

As acções do Tutor-Prolog, no que se refere à pragmática, podem ser observadas, ainda, no acompanhamento dos exercícios propostos pelo Tutor. Estes fazem parte do bloco instrucional das regras.

A forma geral da tela, durante a apresentação dos exercícios orientados, é a indicada na figura 4.11.

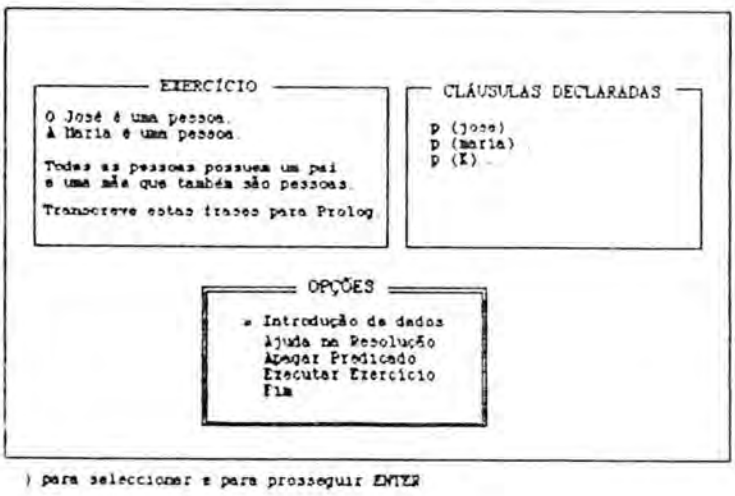


Figura 4.11 Menus para o módulo de exercícios

Temos uma janela contendo o enunciado do problema, uma janela para o aluno escrever a sua solução para o problema proposto, e uma terceira janela contendo as opções (comandos ao dispor do aluno) durante este tipo de interacção.

Durante a transcrição deste problema o Tutor-Prolog observará vários aspectos:

- se o aluno aprendeu a representação de factos. Caso contrário, o conteúdo instrucional relativo a este assunto deverá ser revisto e reforçado;
- se o aluno consegue representar a generalização "Todas as pessoas" e representá-la como cabeça da regra. Caso contrário, noções como a de consultas envolvendo variáveis e quantificadores deverão ser revistas, bem como a formação de regras; e
- se o aluno consegue dividir o problema proposto em cláusulas (condições). Caso contrário, a transcrição de frases do Português para o Prolog devera ser revista de forma mais detalhada.

O Tutor-Prolog não exige que a escrita do problema seja feita na ordem da sua apresentação, pois as informações pragmáticas se encarregam da ordenação das cláusulas. No entanto, o relacionamento entre as variáveis e/ou constantes e entre os predicados utilizados é exigido e verificado pelo Tutor-Prolog. A coerência é verificada através das informações semânticas que o Tutor tem armazenado no enquadramento e que adquire durante a escrita do programa. As informações são mantidos em estruturas como:

```
cabeça(PN,LNC,PS,AN,[AD],...,I).
corpo(PN,LNC,PS,AN,[AD],...,I).
dados(PN,LNC,PS,AN,[AD],...,I).
```

Nesta representação, "cabeça" tem a descrição para a cabeça de cada regra, "corpo" contém a descrição para cada condição do corpo de cada regra e "dados" contém a descrição para os dados necessários para testar o programa. No que respeita às variáveis, PN representa a ordem que a cláusula deve ocupar dentro de cada regra; LNC, a ordem que a cláusula deve ocupar dentro da regra; PS, o predicado que o aluno dá à cabeça da regra; AN, o número de argumentos que a cláusula cabeça deve ter; I, o instante do instanciamento de cada cláusula (é aqui também assumido o conceito de estruturas temporais apresentado na implementação do retrocesso inteligente e no congelamento); e AD, uma lista formada por tantas listas quantos forem os argumentos que devem ser declarados na cláusula cabeça. Cada sublista contém quatro elementos: o número de ordem do argumento dentro de cada cláusula, a localização de referências feitas ao argumento em outras cláusulas (se houver), a localização deste argumento na cláusula que o referencia e o nome utilizado pelo aluno para representar o argumento.

Um problema pode estar descrito em mais de um enquadramento. Cada descrição corresponde a uma forma diferente de solucioná-lo.

Os enquadramentos são rotulados pelo nome do programa. Enquanto o Tutor não identificar entre as soluções previstas, a que está representando o raciocínio do aluno, vai preenchendo vários enquadramentos simultaneamente.

Veja-se, abaixo, o exemplo prático da resolução de um problema auxiliada pelas informações, sobre a organização e a compo-

sição das partes do problema, organizadas no enquadramento.

Para o problema exemplo, apresentado na primeira janela da figura 4.12, temos o seguinte enquadramento:

```
programa1(cabeca(cabeca,PS:X,1,[1,3:4,1:1,A],_),
          corpo(1,PS:X,1,[1,3,2,B],_),
          corpo(2,PS:X,1,[1,4,2,B1],_),
          corpo(3,PS,2,[1,cabeca,1,A],[2,1,1,B],_),
          corpo(4,PS,2,[1,cabeca,1,A],[2,2,1,B1],_),
          dados(dados,PS:X,1,[1,_,_,C],_),
          dados(dados,PS:X,1,[1,_,_,D]),_).
```

Supondo a seguinte interacção descrita através da sequência de exemplos apresentados na figura 4.12 (1) e (2).

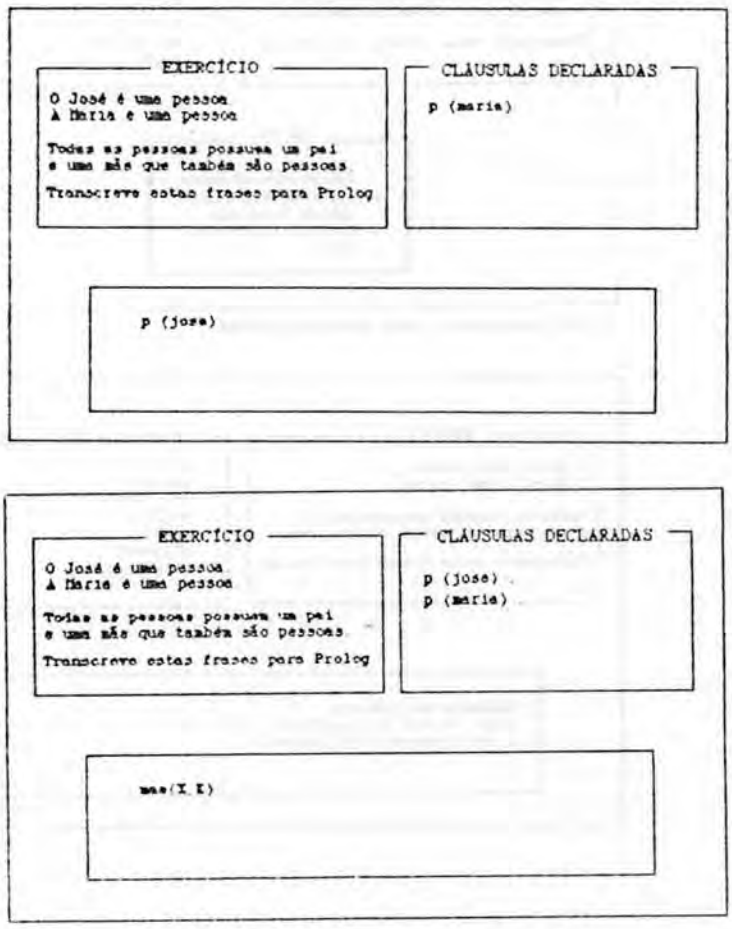


Figura 4.12 Auxílio na escrita de programas (1)

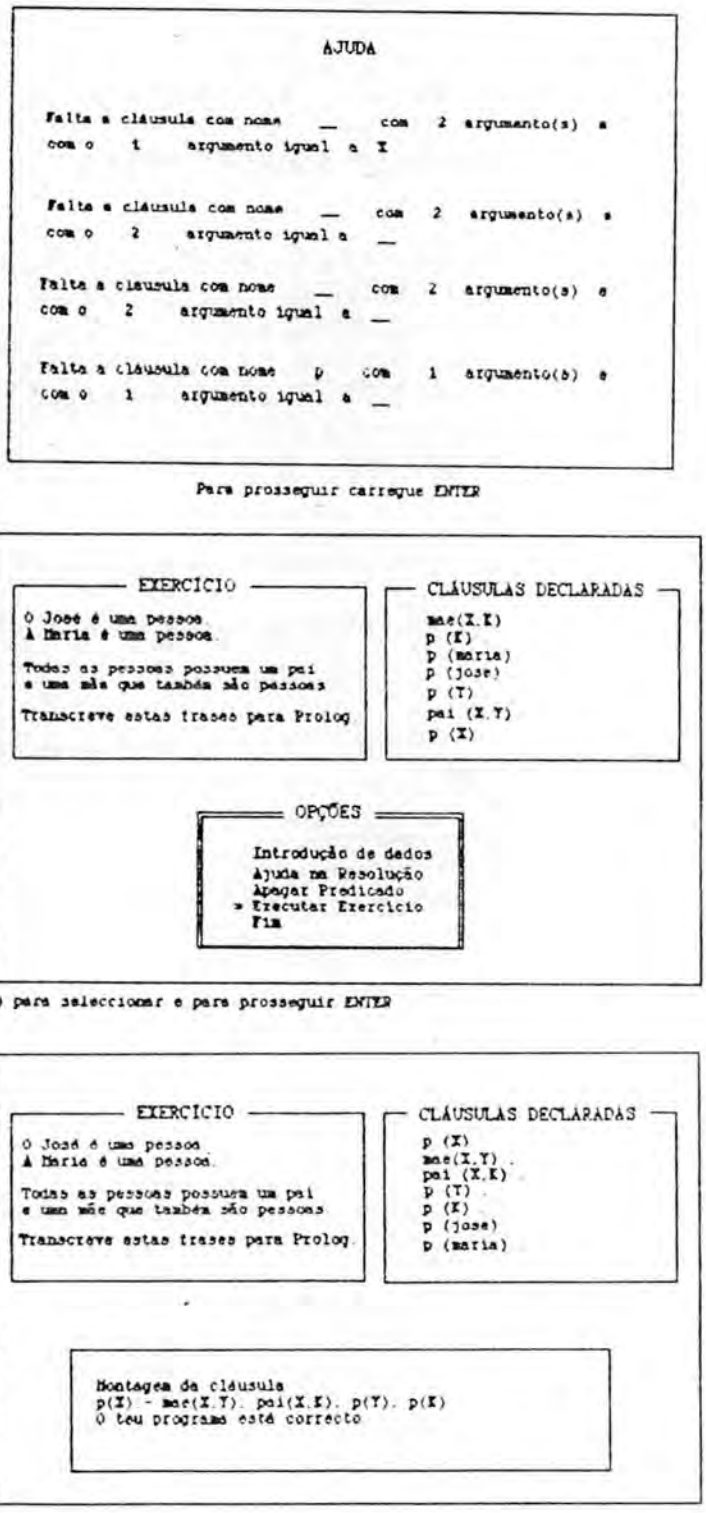


Figura 4.12 Auxílio na escrita de programas (2)

Note-se que o Tutor-Prolog, neste caso, aceita também a definição de um valor constante para o lugar das variáveis.

A organização das cláusulas (apresentadas na terceira janela) é feita a partir da cabeça, indicada por "cabeça", e à medida que as cláusulas vão sendo utilizadas, são retiradas da memória auxiliar "Cláusulas Declaradas". Por último, os dados, indicados por "dados", são organizados e colocados numa nova janela chamada "Informações Existentes". A partir desse instante, o tratamento que o Tutor-Prolog prevê para a regra é idêntico aos casos anteriormente apresentados.

O módulo de exercícios é composto por outros problemas retirados de Lee [LEE 86], como por exemplo:

1. Obtenha os blocos A, B e C.
2. Coloque o bloco C na mesa (representada graficamente).
3. Coloque o bloco B sobre o bloco C.
4. Coloque o boco A sobre o bloco B.

Cuja representação Prolog esparada, pelo Tutor é a seguinte:

```
pilha(A;B;C):-
  obter_bloco(A),
  obter_bloco(B),
  obter_bloco(C),
  colocar(C,mesa),
  colocar(B,C),
  colocar(A,B).
```

Ou a geração de estruturas mais cumplicadas como no problema a seguir:

Gerar uma estrutura para representar as seguintes carracterísticas de um veículo: automovel destinado ao transporte de

passageiros podendo transportar até quatro pessoas e movido a gasolina.

Cujo uma das representações Prolog esperada pelo Tutor é:

```
estrutura(automovel,
        [ [tipo,ligeiro],
          [utilidade,transporte_de_passageiros],
          [lugares,quatro],
          [combustivel,gasolina] ]
        ).
```

Todas as representações apresentadadas levam em conta os problemas futuros que compõem as actividades previstas na utilização do Tutor-Prolog no âmbito do ensino da matemática [ZAM 89].

Em relação à utilização de estruturas como os enquadramentos para a descrição dos programas exercícios, concluímos que elas são insuficientes e demasiado restritas para representar os conhecimentos dinâmicos adquiridos durante a tentativa de solução, e que revelaram pouco flexíveis para o manuseio necessário na operação "esquecer". Nesses casos, quando o aluno opta por ordenar o esquecimento da parte do código já escrito, torna-se muito difícil coordenar o processo recursivo necessário para adequar as demais informações aprendidas a partir da cláusula que deve ser esquecida. Por outro lado, a estrutura mostrou-se eficiente para as necessidades do comando "ajudar", que serve para o Tutor-Prolog orientar o aluno na escrita das cláusulas que ainda são necessárias para a programação completa do exercício. Cabe

ainda salientar que no Tutor-Prolog, quando uma variável é instanciada, a propagação realiza-se tanto para as cláusulas anteriores como para as posteriores, ao contrário do que ocorre nos sistemas de enquadramentos clássicos. A única excepção é feita para a definição da cláusula cabeça, que apenas é preenchida após as demais descrições (corpo e dados) estarem preenchidas. Depois de preenchido, um enquadramento passa a fazer parte do modelo do aluno, pois contém a sua solução para o problema proposto. No Capítulo 6, destinado ao Trabalho Futuro, retomamos este assunto.

Neste Capítulo foram apresentadas a geração do diagnóstico e a meta-interpretação das cláusulas escritas pelo aluno que envolve a detecção e a correcção automática ou orientada das falhas sintácticas, semânticas e pragmáticas ocorridas durante a escrita dos exemplos. Foram ainda apresentados exercícios pré-definidos no Tutor-Prolog, que são utilizados para a fixação dos conteúdos instrucionais e para a apresentação de formas de representação da informação.

5. INTERACÇÃO EM LINGUA NATURAL ESCRITA

Numa sociedade de sistemas comunicantes inteligentes são trocados conhecimentos durante os diálogos interactivos [SER;GAS;COE 88]. Este mesmo processo ocorre entre o Tutor-Prolog e o aluno, entre os vários módulos que compõem o Tutor-Prolog e, num nível simplificado, entre o Tutor-Prolog e outros programas aplicativos. A capacidade de um tutor inteligente para comunicar naturalmente com o aluno é um factor que deve ser considerado cuidadosamente, pois dela depende grandemente a flexibilidade do tutor.

Neste Capítulo apresentaremos as nossas reflexões e experiências em relação ao recurso à língua natural, como forma de comunicação complementar dos menus, no ambiente do Tutor-Prolog. Não pretendemos construir um modelo sofisticado de compreensão do Português, mas somente explorar as vias e as carracterísticas mais pertinentes à aplicação em causa, no sentido de simultaneamente esclarecer direcções futuras de trabalho e de ajudar os alunos a usarem o Tutor-Prolog.

Durante as primeiras experiências práticas, que efectuámos na escola secundária dos Olivais, observámos que os alunos tinham dúvidas quanto ao significado das palavras utilizadas no texto das lições. Assim, num segundo protótipo apresentámos um pequeno

interpretador de frases interrogativas, a fim de possibilitar a consulta a uma base de dados contendo as definições dos termos informáticos em geral e da linguagem de programação Prolog em particular. Por exemplo, "interpretador", "variáveis", etc.

A observação das interacções dos alunos com esse pequeno interpretador provou que as construções sintácticas (estilo) utilizadas variavam, de aluno para aluno, e que o dicionário definido não cobria todas as palavras utilizadas.

Com o desenvolvimento da capacidade de aprendizagem de palavras e de variantes sintácticas, os alunos passaram a utilizar a língua natural também para consultar a base de dados sobre Prolog e ensinar ao Tutor-Prolog as palavras pertencentes ao contexto de suas aplicações (no caso, o estuário do rio Tejo). Esta nova aplicação implicou o desenvolvimento de um pequeno módulo que aceitasse textos e os organizasse adequadamente em bases de dados individuais - uma por aluno.

Estas observações levaram-nos a concluir que a comunicação com o aluno deve ser feita através de:

- menus para a orientação geral do aluno no âmbito da apresentação de cada módulo instrucional;
- língua natural escrita, para questoẽs particulares dirigidas ao Tutor ou a uma possível base de dados gerada pelo aluno, a respeito de um assunto do seu interesse.

A fim de tornar a interacção mais agradável e objectiva foram utilizados recursos tais como as cores, para ressaltar os

tópicos; os ícones, para os casos mais específicos, como a representação de direcções ou locais (setas), lixo, etc.; e os gráficos, para resumir, agrupar e, assim, representar a solução encontrada para um ou para vários problemas apresentados pelo aluno.

Defendemos a ideia de que os sistemas com facilidades de adaptação a novos domínios e com capacidades de interacção coloquial em língua natural escrita, são mais atractivos do que os sistemas vocacionados para tratar apenas de determinados domínios.

5.1 Modelo da interface

Um dos desafios, no desenvolvimento de sistemas transportáveis para a interpretação da língua natural, é o fornecimento da informação relevante para que o sistema possa, com êxito, vencer a barreira entre a imagem que o utilizador tem sobre o domínio do discurso e a forma como a informação está estruturada, neste domínio, para processamento computacional. Para isto, o sistema deverá ter um modelo que inclua a informação acerca dos objectos do domínio: as suas propriedades, as relações entre eles e as frases e palavras usadas para os identificar. Deve, ainda, saber como relacionar a informação desse modelo com o conteúdo da base de dados. Assim, estes sistemas deverão possuir características que lhes permitam, de um modo simples, adquirir conhecimentos

pertinentes na construção de um modelo apropriado a cada domínio.

O modelo de aprendizagem testado neste protótipo é bastante dependente do aluno e deve contribuir para orientá-lo no que se refere à inclusão de novos sintagmas ou de variantes estruturais. Verificámos que menus, activados pelo interpretador gramatical, que contenham exemplos com possíveis situações para a acomodação das novas palavras às regras, facilitam e orientam mais adequadamente o aluno. Os menus são seleccionados de acordo com o modelo (hipótese) que o tutor vai gerando para a análise de cada frase. As hipóteses são geradas a partir do conhecimento disponível em cada interacção.

No Tutor-Prolog considera-se apenas a análise de frases escritas em Português (o processo é transportável para outra língua natural qualquer), possibilitando a correcção automática de erros ortográficos e a aprendizagem externa de novas palavras e de novas regras sintácticas [LOP;VIC 88].

A apresentação dos menus é activada automaticamente para iniciar um diálogo ou para prosseguí-lo. O mesmo ocorre se for detectado um "conflito", isto é, um contexto desconhecido pelo sistema.

O menu principal serve para orientar o aluno sobre as opções de trabalho, de acordo com a configuração apresentada na figura 5.1.

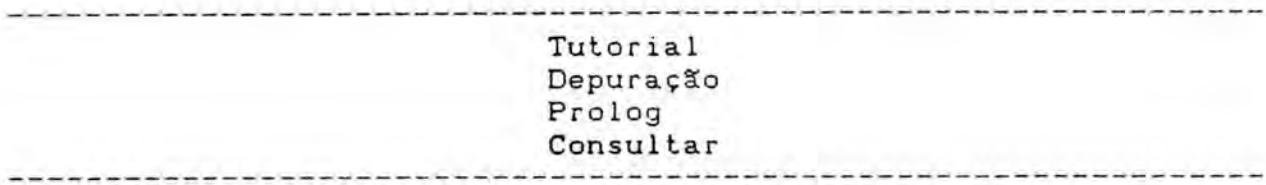


Figura 5.1 Níveis de trabalho no tutor

Já apresentámos, no Capítulo 2, a comunicação através de menus. Agora, descreveremos as experiências na interpretação do subconjunto da língua natural escrita (Português). Serão apenas tratados os aspectos da aprendizagem com base na geração de hipóteses e com o auxílio de exemplos. O nosso objectivo restringe-se à aprendizagem ao nível da sintaxe, pois o nosso maior interesse relaciona-se com a investigação de formas de aprendizagem e não com a Linguística Computacional. Assim, a forma de aprendizagem aqui apresentada é diferente da apresentada no Capítulo 3 (adaptação), onde o Tutor-Prolog gera hipóteses e tenta verificá-las a partir das informações, que obtém de forma indirecta, a respeito do conhecimento do aluno. Neste caso, a aprendizagem depende das informações fornecidas directamente pelo aluno (hipóteses do aluno) e do conhecimento existente no Tutor naquele determinado instante (hipóteses do Tutor), ou seja, aprendizagem por instrução [MIC 86].

Para além da forma de interacção auxiliada por menus, foi testada outra, em que o aluno dá crédito irrestrito às hipóteses do Tutor. Isto é, o aluno, em particular, não auxilia o tutor no processo de aprendizagem. Numa experiência mais abrangente seria

natural que a gramática inicial fosse alargada, de forma a conter maior poder de interpretação, o que possibilitaria um menor recurso ao processo de aprendizagem.

No entanto, sabemos que é difícil definir um conjunto de regras que contemplem todas as possibilidades de construções de frases válidas ou um dicionário que contenha todas as palavras possíveis de serem escritas pelo utilizador. Em casos concretos de consulta a bases de dados, quando se restringe o universo de aplicação, torna-se mais fácil conseguir tal aspiração. No caso em estudo este desejo é impossível, porque, embora trabalhemos com um subconjunto da língua natural escrita e tenhamos um objectivo restrito e bastante explorado do ponto de vista da linguística computacional - a consulta a bases de dados - o universo de aplicação a que o tutor se destina é aberto. As bases que o aluno construirá podem, em princípio, versar sobre qualquer assunto e será sobre o assunto escolhido que o aluno procurará informações.

Como nesta situação o tutor do sistema é o aluno, as informações adquiridas poderão ser inadequadas do ponto de vista gramatical. Neste aspecto concordamos com as posições defendidas por Turk [TUR 88], que diz ser o uso social das formas aprendidas o que realmente interessa na aprendizagem linguística, e com Suchmam [SUC 87], que diz ser a linguagem utilizada pelos alunos dependente da cultura local. Se determinada nova estrutura for incorrecta, do ponto de vista sintáctico, mas for correntemente usada por um

certo aluno, sem provocar ambiguidades, ela deve ser adquirida pelo Tutor. Num tutor, o importante é a comunicação, isto é, a manutenção de um diálogo mais ou menos cooperante de modo a não perder a ligação com o aluno. O Tutor-Prolog deseja manter a motivação do aluno para a aprendizagem da linguagem Prolog, o que não seria possível se fosse obrigado a intervir constantemente devido a falhas na escrita das consultas. Se, no entanto, as capacidades linguísticas do tutor se destinassem ao ensino da língua, a estratégia certamente seria outra.

Para atingir este objectivo, tornou-se necessário que o nosso pequeno interpretador da língua natural escrita, composto por um conjunto de 24 regras gramaticais iniciais (ver anexo 4), possuisse capacidade para aprender palavras novas e diferentes estruturas sintácticas, para além de corrigir erros ortográficos. A análise da frase é feita recorrendo ao formalismo DCG de descrição de gramáticas de língua natural [PER;WAR 80].

Apresentaremos, agora, uma pequena gramática livre do contexto que é utilizada para a compreensão dos demais exemplos que tratam da evolução gramatical, de acordo com o exposto em [LOP;VIC 88].

```
s --> np,vp                          (R1)
np --> det, np_nucleo                (R2)
np --> pronome                       (R3)
np_nucleo --> nom_prop               (R4)
np_nucleo --> nom, adj               (R5)
vp --> vp_nucleo, pp                 (R6)
vp_nucleo --> aux, aux_comp          (R7)
vp_nucleo --> vi                     (R8)
vp_nucleo --> vt, np                 (R9)
aux_comp --> adj                     (R10)
aux_comp --> np                      (R11)
pp --> prep, np                      (R12)
pp --> []                            (R13)
adj --> []                           (R14)
det --> []                           (R15)
```

Figura 5.2 Núcleo gramatical inicial

Actualmente, este pequeno núcleo gramatical evolui de acordo com o modelo utilizado no Tutor-Prolog, apresentado a seguir.

Neste modelo existe uma pequena gramática G que evolui para uma grámatica $g(G,U_i)$, de acordo com os conhecimentos a que o utilizador U_i (neste caso o aluno) recorre ao longo de um intervalo de tempo, por exemplo, uma sessão.

Na figura 5.3, ---U---> representa a situação onde a gramática G, através duma interacção com o aluno U_i, evolui para uma nova gramática g(G,U). U representa cada aluno particular u1, u2,...,ui.

```
                G ---u1 ---> g(G,u1)
                G ---u2 ---> g(G,u2)
                G ---u3---> g(G,u3)
                G
```

Figura 5.3 Evolução da gramática no modelo do Tutor-Prolog

A gramática do Tutor-Prolog está organizada ao redor de um conjunto de categorias abertas e fechadas e das que permitem o reconhecimento pelo vazio ([]). As categorias abertas permitem a aprendizagem de novas palavras (substantivos, adjectivos e verbos), à semelhança do que acontece nos sistemas TEAM [GRO;APP; MAR;PER 87] e TELI [BAL;STU 84,86], entre outros. No caso de o aluno desconhecer a categoria a que a palavra pertence, esta é guardada como suspensa. O Tutor-Prolog procurará, então, durante as interacções seguintes, adquirir informações que lhe permitam classificá-la. As categorias fechadas não permitem a aprendizagem de novas palavras, como por exemplo, determinantes, pronomes, preposições e verbos auxiliares, que sempre são utilizadas, independente da aplicação. As categorias que permitem o reconhecimento pelo vazio ([]) como, por exemplo, os determinantes, as preposições, os adjectivos e o sintagma nominal constituem a última hipótese de análise da palavra gerada pelo Tutor. Com referência ao tratamento do vazio a abordagem feita no Tutor é original, pois nos trabalhos citados este assunto não é mencionado.

Do ponto de vista da implementação, para aprender uma nova palavra o Tutor-Prolog necessita apenas de informações que indiquem os argumentos que a regra ou o facto a ser gerado deve conter. Ou seja, para aprender um novo vocábulo é necessário apenas que estejam definidos os argumentos que devem ser associados à categoria, não sendo necessário existir vocábulos definidos

à partida. Por exemplo, para que um substantivo seja anexado ao dicionário, basta definir a estrutura da figura 5.4.

```
dic_*(...,Par_i,Par_i+1,...,Par_n) -->
        ( notifica_vazio(*) )

dic_*(...,Par_i,Par_i+1,...,Par_n) -->
                [#].

dic_*(...,Par_i,Par_i+1,...,Par_n) -->
        tenta_corrigir(dic_*).
```

Onde * representa o nome de uma das categorias conhecidas pelo sistema, n ∈ (0,1,2). Neste caso, * corresponde à categoria substativo.

Se n=2 então:

- Par_1 identifica o género (masculino ou feminino) do vocábulo #;

- Par_2 identifica o número (singular ou plural) do vocábulo #.

Se n=1 então:

- Par_2 identifica o número do vocábulo #.

Se n=0 o género e o número não fazem parte da definição do vocábulo #.

Figura 5.4 Estrutur do dicionário

O mesmo processo deve ser implementado para o tratamento das conjugações das formas verbais.

Para que a aprendizagem seja possível, o processo de análise sintáctica é acompanhado pela geração de hipóteses sobre a utilização de determinada regra gramatical. Assim, quando um sintagma não consegue ser analisado com uma determinada regra sintáctica, inicia-se a geração das hipóteses relativas ao motivo

da falha, juntamente com um processo interactivo que visa a aquisição de novas informações.

5.2 Geração de hipóteses

Se a frase escrita estiver de acordo com os conhecimentos existentes no Tutor-Prolog, a sua interpretação é automática. A interpretação, nesse caso é realizada em apenas um passo, isto é, de forma imediata. No entanto, podem ocorrer situações não previstas (falhas) nas regras da gramática do Tutor-Prolog. Estes casos podem vir a ser solucionados pelos meta-conhecimentos, que através da geração de hipóteses e da interacção com o aluno procuram ultrapassar tais situações e, se possível, actualizar as regras gramaticais de forma a que passem a contemplá-las. O Tutor-Prolog gera todas as hipóteses possíveis na tentativa de solucionar um determinado problema mas, testá-as uma a uma, guardando as outras para o caso de ser necessário o retrocesso na busca de nova solução. Este processo é identico ao adoptado no meta-interpretador Prolog (correcção automática de erros e retrocesso inteligente).

Assim, as hipóteses relacionam-se com o aparecimento de três situações:

- correcção de erros ortográficos;
- aprendizagem de palavras novas pertencentes à categorias conhecidas pelo Tutor;

- aprendizagem de construções sintácticas novas; e
- aprendizagem de categorias gramaticais desconhecidas pelo Tutor.

5.2.1 Correcção de erros ortográficos

O facto de uma dada palavra, escrita pelo utilizador, não poder ser categorizada sintacticamente, não é razão para se concluir que a palavra seja desconhecida. Pode acontecer que a palavra apenas esteja mal escrita [BER 87].

Na figura 5.5, apresentada em seguida e retirada de uma observação prática, um aluno consulta a sua base de dados sobre flores.

```
-> Quais são as fllores de inverno?
                ^
   Palavra corrigida: flores

                                    -> Assuntos
                                       flores
                                       prolog
                                       areias
```

Figura 5.5 Correcção de erros ortográficos

A primeira hipótese gerada é a da palavra ter sido escrita de forma incorrecta. Esta hipótese pode ser confirmada ou abandonada facilmente, dependendo apenas de uma consulta ao dicionário. A correcção, portanto, é feita apenas com base nas informações contidas no dicionário morfo-sintáctico. Se no referido dicionário aparecessem definidas duas palavras com o

mesmo número de letras e divergindo apenas numa letra, o algorítmo não possuiria regras para decidir com precisão. Solicitaria, por conseguinte, a intervenção do utilizador para esclarecer a ambiguidade, ou seja, a existência de duas hipóteses para a correcção da mesma palavra. Outra restrição diz respeito ao número de letras que compõem a palavra. Quanto maior o número de letras, maior será o grau de certeza na correcção. O limite mínimo considerado é a existência de duas palavras com três letras iguais nas "mesmas" posições. É importante salientar que a palavra pode ser analisada de três maneiras: a partir da direita, a partir da esquerda e a partir do centro. No caso do exemplo da figura 5.5, a primeira abordagem detecta a existência de duas letras iguais em posições iguais ("f,l") e um conflito ("l"). Na segunda tentativa são analisadas as letras "o,r,e,s" que estão de acordo com a palavra escrita no dicionário. No final resta apenas uma letra "l" que provoca a diferença e, neste caso, a estratégia do Tutor é eliminá-la.

Assim, o algorítmo de correcção realiza uma pesquisa indexada sequencial no dicionário. A procura é indexada pela categoria que corresponde à hipótese da regra de análise que gerou o conflito e é feita de forma sequencial, dentro das palavras que compõem esta categoria, até achar a mais parecida (maior número de letras iguais nas mesmas posições) com a palavra em causa. O algorítmo permite detectar erros no início, no meio e no fim de uma palavra (inclusive erros consecutivos). Apesar da pesquisa ser indexada, o tempo de resposta do sistema tende a

aumentar com a inclusão de novas palavras (evolução do dicionário).

A comparação è feita a partir de duas listas que contêm as listas das letras que formam as palavras (a escrita pelo aluno e a seleccionada no dicionário). A palavra escrita pelo aluno e a palavra mais parecida encontrada no dicionário, dentro da categoria correspondente à hipótese que o Tutor-Prolog está tentando verificar.

As palavras são comparadas letra a letra e, quando ocorre alguma diferença, a adequação é feita a partir da palavra encontrada no dicionário. O aluno pode sempre discordar das hipóteses apresentadas pelo Tutor-Prolog.

A concordância (género e número) poderá ser feita após a consulta ao dicionário, com base nas informações já obtidas na análise, segundo o método utilizado por Lopes [LOP 87].

Apesar do Tutor-Prolog utilizar a correcção automática de erros ortográficos, defendemos a ideia de que mais importante do que um sistema saber corrigir um erro é a sua capacidade de conviver com ele. Ou seja, é fundamental que um sistema entenda as frases do aluno e execute os seus desejos, apesar dos erros que as suas mensagens possam conter.

5.2.2 Aprendizagem de palavras novas

Se não houver no dicionário nenhuma palavra semelhante à escrita pelo aluno, é plausível a hipótese de que a palavra seja desconhecida. Neste caso, o Tutor-Prolog tenta adquirir conhecimento interrogando o aluno. No entanto, em vez de perguntar qual é a categoria sintáctica da palavra desconhecida, utiliza as regras da gramática que dispõe, emitindo uma ou mais hipóteses que tenta verificar logo a seguir. Por exemplo, quando um aluno pergunta:

"Quais são os peixes coloridos?"

O interpretador possui regras para analisar a parte da frase "Quais são" ((R3) e (R7)). A estas, segue-se a regra (R2) que espera um sintagma nominal.

Supondo, no entanto, que o Tutor desconheça a palavra coloridos, pela aplicação da regra (R5) gerará a hipótese de coloridos ser um adjectivo ainda desconhecido. Ao aluno vai ser perguntado se a palavra desconhecida pertence à categoria correspondente à hipótese formulada (aprendizagem através de exemplos). Para confirmar esta hipótese, o aluno pode solicitar ajuda ao tutor, que apresentará, neste caso, a lista dos adjectivos conhecidos. Caso o aluno confirme, a hipótese do tutor passa a ser considerada forte. A nova palavra é aprendida (anexada no dicionário) e a análise prossegue normalmente, isto é, segue explorando a mesma regra gramatical.

Para a geração das hipóteses, é mantido o controle sobre a regra que está sendo utilizada (início e fim). Isto é importante porque durante a tentativa de análise de uma frase várias regras podem vir a ser utilizadas e a cada uma delas corresponder um conjunto de hipóteses. No entanto, no final da análise são utilizadas para a aprendizagem do sistema apenas as hipóteses que foram verificadas. As demais devem ser eliminadas.

Quando uma hipótese do Tutor é verificada, isto é, confirmada pelo aluno, a indicação de conflito é retirada e a palavra ou construção sintáctica que a gerou é aprendida. Caso o aluno não confirme uma das hipóteses do Tutor e não apresente uma hipótese alternativa, a palavra é armazenada como suspensa. Este último caso ocorre quando, ambos, aluno e Tutor, não possuem conhecimentos suficientes para confirmar ou negar as hipóteses geradas como, por exemplo, a interacção apresentada na figura 5.6.

As hipóteses podem conter informações para a aprendizagem de novas palavras (1) e para a geração de novas regras ou transformação das já existentes (2). As informações contidas nestas hipóteses são organizadas da seguinte forma:

```
(1)   hipotese(tutor,p(Palavra,Categoria))        ou
      hipotese(aluno,p(Palavra,Categoria))

(2)   hipotese(tutor,p(Numero_da_regra_actual,
                       Nova_composição) )
```

Assim, para que uma nova palavra seja aprendida de forma completa é necessário, para além de conhecê-la, saber a categoria

gramatical a qual pertence. Caso o aluno e o Tutor desconheçam a informação relativa à categoria, a palavra é armazenada de forma incompleta. Para que uma regra seja transformada é necessário conhecer qual a regra que deve ser reescrita, isto é, o seu número e a sua nova composição.

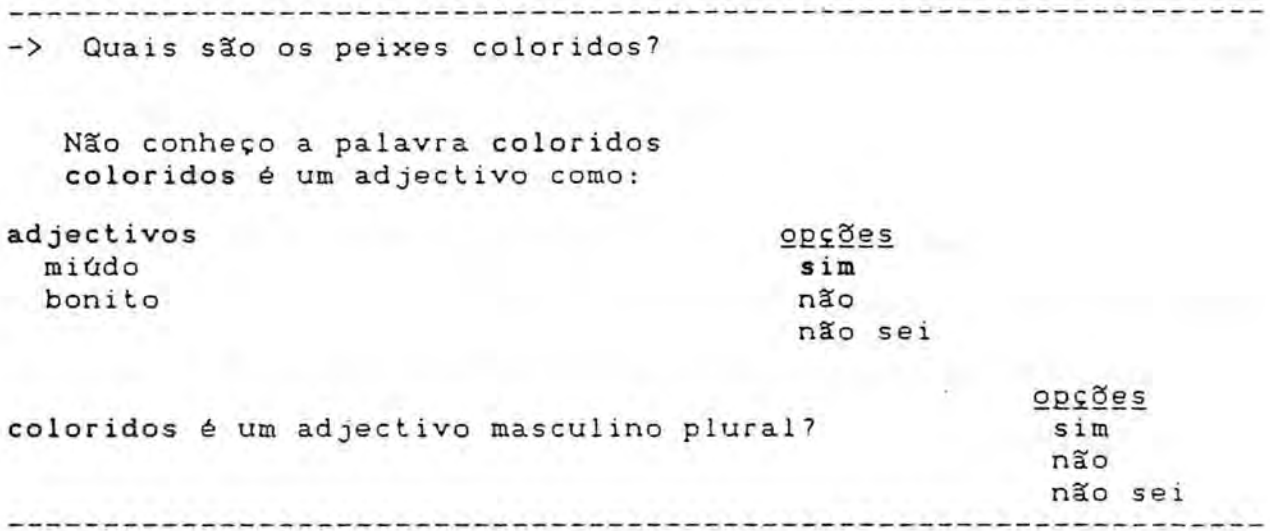


Figura 5.6 Aprendizagem de palavras

Caso o aluno negue a hipótese, o Tutor passa a orientá-lo, para que informe a categoria gramatical a qual a palavra pertence, através de uma janela contendo as categorias conhecidas pelo sistema. Depois do aluno prestar esta informação, o Tutor verifica no dicionário se a palavra já se encontra definida na categoria indicada. Uma situação destas pode significar que a regra, que está a ser utilizada, tenha de ser abandonada ou reescrita, inserindo-se no caso de aprendizagem de novas construções sintácticas. Caso a palavra não exista, é automaticamente definida.

Após estas operações é gerada a hipótese de que a formação da frase em análise é uma variação da regra que está sendo explorada. A análise da frase segue com a tentativa de reconhecimento dos demais componentes.

A aprendizagem de novas palavras pode ser feita directamente pelo tutor, isto é, sem a intervenção do aluno, bastando que haja uma base de dados (devidamente adaptada) contendo um dicionário da língua portuguesa. Para que isso seja viável em microcomputadores, pode ser utilizado um sistema distribuído em rede. Se, durante a interpretação, fosse encontrada uma palavra desconhecida, o aluno receberia, apenas, uma mensagem como, por exemplo: "Tu utilizaste uma palavra que desconheço. Vou procurá-la no dicionário".

5.2.3 Aprendizagem de variações das construções sintácticas conhecidas

Consideremos o exemplo da figura 5.7, em que, inicialmente, o tutor desconhece as palavras coloridas e peixes.

```
-> Quais são as coloridas especies de peixes?

   Não conheço a palavra coloridas
   coloridas é um substantivo como:

substantivos conhecidos                    opções
  margaridas                               sim
  palmas                                   não
  cravos                                   não sei
```

Figura 5.7 Hipótese do Tutor

Neste ponto do diálogo é apresentada uma janela, com as categorias sintácticas conhecidas pelo tutor, e é pedido ao aluno que escolha uma das categorias em que classifica a palavra, como ocorre na figura 5.8. A solicitação da categoria só é feita porque o Tutor-Prolog não possui mais hipóteses que permitam a sua classificação. Em seguida, e para confirmação, há uma nova apresentação da palavra.

```
-------------------------------------------------------------------
                                          categorias
                                          substantivos
                                    ->    adjectivos
 coloridas é um adjectivo como:           verbos
                                            ......

adjectivos                        opções
  grande                          sim
  miúdas                        nao
  bonitas                         nao sei

coloridas é um adjectivo feminino plural?

                                                      opções
Não conheço a palavra: especies                       sim
especies é um substantivo como:                       não
                                                      não sei

substantivos conhecidos
  margaridas
  palmas

especies é um substantivo comum plural?
-------------------------------------------------------------------
```

Figura 5.8 Hipótese do aluno

A partir do momento em que falhou a hipótese de coloridos ser um substantivo, e de o aluno ter ensinado ao tutor de que se trata de um adjectivo, tornando fraca a primeira hipótese do Turtor-Prolog, conclui-se que deve haver uma outra regra, desconhecida do Tutor, para verificar a estrutura sintáctica veiculada pelo aluno.

Esta hipótese do aluno gera uma nova descrição para "np_nucleo":

np_nucleo --> nom, adj (R5)

como (R5) é a única regra que contém a categoria "nom" a direita, dá origem à variação (R5'):

np_nucleo --> adj, nom (R5')

As hipóteses estão armazenadas internamente sob a forma de uma pilha. A passagem de uma regra para outra é marcada pela hipótese que representa o fim de cada regra "hipotese(tutor,fim_regra)". Assim, teremos na pilha, apresentada na figura 5.9, as seguintes hipóteses visando a reescrita da regra para a análise da frase anterior.

```
hipotese(tutor,fim_regra).
hipotese(tutor,adj).
hipotese(tutor,nom).
hipotese(tutor,det).
```

Figura 5.9 Armazenamento das hipóteses

A história das regras percorridas, para que a frase seja completamente analisada, é apresentada na figura 5.10.

```
(3)   reg_corrente(np_nucleo).
(2)   reg_corrente(np).
(1)   reg_corrente(np).
```

Figura 5.10 Histórico das regras percorridas

Como foi dito, a análise de uma frase pode passar por duas etapas. A primeira, analisa a frase caso esta esteja de acordo com as regras definidas e todas as palavras que a compõem façam parte do dicionário. A segunda, corresponde à meta-interpretação, que é realizada quando alguma incompatibilidade é encontrada, ou seja, a primeira tentativa é abandonada e uma nova tentativa de análise é iniciada, com a adição de controle às regras gramaticais (processo idêntico ao do meta-interpretador Prolog). Assim, na pilha da história da análise, (1) corresponde à primeira tentativa ("análise livre"), (2) corresponde à segunda tentativa, onde foi detectado um conflito ("análise controlada"), e (3) representa a parte da regra que deve ser reescrita para possibilitar a análise completa da frase. O aluno pode acompanhar estas etapas graficamente como é apresentado no exemplo da figura 5.11. A frase do exemplo (4) não pode ser directamente analisada pelas regras do núcleo gramatical inicial apresentado na figura 5.2. Mais especificamente, devido à estrutura actual da regra (R5).

```
-----------------------------------------------------------------
>> O colorido carro é rápido.
Mapa das regras percorridas

->  np(*) -> ¦  Primeira tentativa de análise com as regras
             ¦  (R1), (R2) e (R5)
             ¦
             ¦
    ¦<-------   Retrocesso para a alteração da regra (R5)
    ¦
    ¦
    np(2) >> vp(3) >> vp_nucleo(1) >> aux_comp(1)   Segunda tenta-
tiva de análise onde ocorre a alteração da regra  (R5).
-----------------------------------------------------------------
```

Figura 5.11 Gráfico da análise sintáctica

Após a apresentação gráfica, o aluno pode olhar as regras resultantes, de acordo com a sequência apresentada no gráfico, através do comando "espiar", que possibilita a listagem das regras do meta-interpretador.

A representação gráfica apresentada pelo aluno é feita de uma forma diferente. Ou seja, ao invés de serem usadas as convenções "np, vp, vp_nucleo, aux_comp, etc." são utilizadas letras "a, b, c, ..." de modo a transmitir a noção da sequência dos acontecimentos ocorridos durante a análise da frase. A legenda contendo o signivifaco de cada letra é apesentada numa janela.

Supondo que o sistema não conheça a categoria advérbio e que o aluno escreva a seguinte frase:

```
-> As rosas são flores muito bonitas.

   não conheço a palavra: muito

 muito é um adjectivo como:

adjectivos                          opções
  grande                            sim
  miúdas                            nao
  bonitas                           nao sei

    Qual é a sua categoria sintáctica?

                          Categorias
                        substantivo comum
                        substantivo próprio
                        adjectivo
                        verbo
                        outra
   >> advérbio.
```

Figura 5.12 Aprendizagem de uma categoria gramatical

A informação (hipótese do aluno) apresentada no exemplo 5.12 leva o Tutor-Prolog a gerar a hipótese de que novas regras devem ser criadas e outras devem ser reescritas:

```
adj1 --> adverb, adj          (R16)
adverb --> []                 (R17)
adverb --> [muito]            (R18)
```

As regras (R5'), (R10) e (R15), também devem ser substituídas pelas novas regras seguintes:

```
aux_comp --> adj1             (R10')
np_nucleo --> nom1, adj1      (R5'')
noun1 --> adj1, nom           (R15')
```

Nos casos em que o aluno desconhece a categoria à qual a palavra pertence, o Tutor pesquisa de forma ascendente o dicionário, com o objectivo de verificar a existência da palavra desconhecida em outra categoria gramatical. Desta pesquisa são omitidas todas as categorias pertencentes às regras já percorridas e à regra corrente. Caso a palavra não seja encontrada, ou o aluno discorde da categoria encontrada (hipótese do Tutor) mas desconheça a categoria adequada, a palavra é armazenada sob o símbolo predicativo "desconheço".

Uma mesma palavra pode estar armazenada no dicionário em diferentes categorias. Por exemplo, na frase "Os peixes nadam **bem**." a palavra **bem** é armazenada como advérbio e na frase "O bem vence o mal.", como substantivo comum.

Relembrando o procedimento para reescrever as regras da gramática apresentado no Capítulo 3 temos:

- determinação da zona de conflito;
- geração de hipóteses de reescrita da zona de conflito;
- teste das hipóteses com o auxilio do aluno;
- nova designação da zona de conflito;
- substituição, na regra inicial, da zona de conflito pela sua nova designação;
- criação das novas regras, tendo como cabeça a designação da zona de conflito e como corpo a zona de conflito ou as hipóteses geradas durante a análise da frase.

Até este momento vimos como se dá a detecção da zona de conflito. Abordaremos em seguida o processo de reescrita propriamente dito.

No momento da reescrita o sistema conhece a regra e a cláusula onde houve o conflito. Para que este seja eliminado, resta-lhe apenas reescrever a regra. Vejamos um exemplo prático, de acordo com a gramática descrita no Anexo 4, ao invés da forma simplificada apresentada na figura 5.2, que contém informações de controle, da qual foi seleccionada a regra da figura 5.13.

```
-----------------------------------------------------------------
    sint_nom(1,Gen,Num) -->
        { regra_corrente(sint_nom) }, % marca o início das
        prep,                         % regras para a análise
        det(Gen,Num),                 % do sintagma nominal
        adj(Gen,Num),
        nome_com(Gen,Num),            % marca o fim da primeira
        { fim_regra(sint_nom,1) }.    % regra para o sint_nom
-----------------------------------------------------------------
```

Figura 5.13 Regra gramatical com controle

A regra anterior é utilizada para analisar a expressão:

"contra a janela enorme"

Durante a análise, a frase provoca a geração das seguintes hipóteses, a partir da regra da figura 5.13:

```
hipotese(aluno,fim_regra).      ^
hipotese(aluno,adj).            :
hipotese(aluno,nome_com).       :
hipotese(aluno,det).            :
hipotese(aluno,prep).           :

regra(sint_nom,1).
```

Com estas informações, o sistema começa por criar uma lista com a sequência de invocações indicadas por "hipotese", que é comparada com a lista que contém as invocações efectivamente realizadas. Partindo da regra da figura 5.13 temos:

```
L_hip == [prep, det, adj, nome_com] % invocações
                                      indicadas
L_cat == [prep, det, nome_com, adj, fim_regra] % invocações
                                                 realizadas
```

Estas duas listas originam outras quatro, que contém respectivamente: as partes comuns entre as invocações realizadas e as indicadas pelas hipóteses; as hipóteses fracas, composta pelos elementos das invocações realizadas que não pertencem aos elementos comuns; as hipóteses fortes, com os elementos das invocações indicadas que pertencem aos elementos comuns; e a lista restante com os elementos de que estão depois do sinal de fim de regra.

```
L_comum= [prep,det]
L_fraca= [nome_com,adj]
L_forte= [adj,nome_com]
L_resto= []
```

A lista "L_resto" é utilizada nos casos em que a cláusula foi anteriormente reescrita. Nestas situações, após "fim_regra" existem invocações a novas regras. Estas invocações necessitam ser separadas para que o nome da nova cláusula possa ser correctamente determinado e para que a invocação da nova regra se dê antes das já existentes.

A nova cláusula construída deve ter a mesma estrutura que todas as outras do seu tipo, tendo-se especial cuidado com a manutenção da coerência dos parâmetros, entre os quais se incluem os utilizados pelas regras DCG e os destinados a fins específicos, como, por exemplo, género e número. As informações sobre os parâmetros envolvidos por uma cláusula que vai ser reescrita são mantidas em tabelas auxiliares.

Para além das tabelas, é ainda necessária a geração de variáveis não instanciadas (processo idêntico ao do meta-interpretador Prolog) para serem utilizadas como parâmetros das regras DCG e como parâmetros adicionais específicos. Há a necessidade, também, de informações que indiquem (indexação) a posição que cada parâmetro de fim específico deve utilizar dentro da cláusula.

Assim, se não existirem invocações após a marca de fim de

regra (L_resto = []), e sendo "X" o nome predicativo da cabeça da regra onde se deu o conflito e "Id" o identificador da cláusula onde ele ocorreu, o nome da nova regra será o resultado da concatenação de "X" e "Id". Caso já tenham sido anexadas invocações após a marca de fim de regra (L_resto \= []), e "Id" o identificador da cláusula onde se deu o conflito, o nome da nova regra será o resultado da concatenação de "X", "_" e "Id".

Abordaremos em seguida a substituição da regra inicial pela sua nova designação.

Supondo que se pretenda construir uma cláusula com nome "**sint_nom**" e identificador "1", e a existência da lista L = [nome_com, adj, sint_nom1], seriam então construídas mais três listas. A de endereço, L1 = [_P1, _P2, _P3, P4], onde, genericamente, "_Pi" é um endereço do tipo "_3579" que é utilizado para gerar os parâmetros das cláusulas que serão reescritas, como por exemplo, na cláusula "sint_nom1", P2 e P4 (figura 5.15).

```
sint_nom(P1,P2):-
        nome_com(P1,P3),
        adj(P3,P4),
        sint_nom1(P4,P2). % geração de nova regra
```

Figura 5.15 Acréscimo de uma cláusula na regra original

A de variáveis, L2 = [_Var1, _Var2, _Var3], onde, genericamente, "_Vari" possui a mesma forma que _Pi e também representa parâmetros, que neste caso são específicos, podendo representar, por exemplo, o número de sequência da nova regra e a concordância

de género e número. Estes parâmetros são anexados à cláusula, como apresentado na figura 5.15.

```
sint_nom(1,Var1,Var2,P1,P2):-
     nome_com(Var1,Var2,P1,P3),
     adj(Var1,Var2,P3,P4),
     sint_nom1(Var3,Var1,Var2,P4,P2). % novos parâmetros
```

Figura 5.15 Acréscimo de parâmetros numa cláusula

Para obter-se a coerência entre os parâmetros, isto é, para que sejam usados os mesmos endereços, sempre que se queira referenciar os mesmos argumentos é necessário a utilização da terceira lista L3, que, neste caso, tem a seguinte configuração: L3 = [id:1, genero:2,numero:3], onde L3 indexa L2.

Este método garante que o interpretador Prolog seja o único responsável pela gestão do espaço disponível em memória, assegurando que nunca seja atribuido o mesmo endereço a duas variáveis diferentes não instanciadas.

Após a geração destas informações, resta apenas construir a regra em causa. Este processo é executado pelos procedimentos "const_cabeça", seguido de "const_corpo" (ver Anexo 4), que utilizam as informações das listas L1, L2 e L3 para construir a cabeça e o corpo da regra. Depois de construída, a regra é anexada ao núcleo gramatical inicial.

A escolha do lugar onde será armazenada a nova regra é importante, visto que o processamento na linguagem Prolog é feito

na ordem descendente e da direita para a esquerda. Assim, durante a anexação, uma das três situações seguintes pode ocorrer:

(a) a nova regra ser anexada antes das existentes;

(b) a nova regra ser anexada no final do conjunto de regras existentes para aquele caso ou

(c) a nova regra ser anexada entre as regras que compõem o conjunto de regras para aquele caso.

A anexação dos casos (a) e (b) é trivial. Já para o caso (c), os seguintes passos devem ser observados: as regras existentes vão sendo retiradas e guardadas numa estrutura, até que o número indicado em cada regra retirada seja menor ou igual ao da regra a anexar[1]. Quando isso ocorrer, a nova regra é anexada e as que foram retiradas são, a seguir, recolocadas.

Na figura 5.16 é explicada a utilização das pilhas, as quais contêm a regra corrente, a indicação da regra antecessora ("invocada por"), as regras invocadas e as hipóteses geradas durante a análise.

---

1 O mesmo processo é realizado nos casos em que uma nova cláusula deve ser inserida no corpo de uma regra já existente.

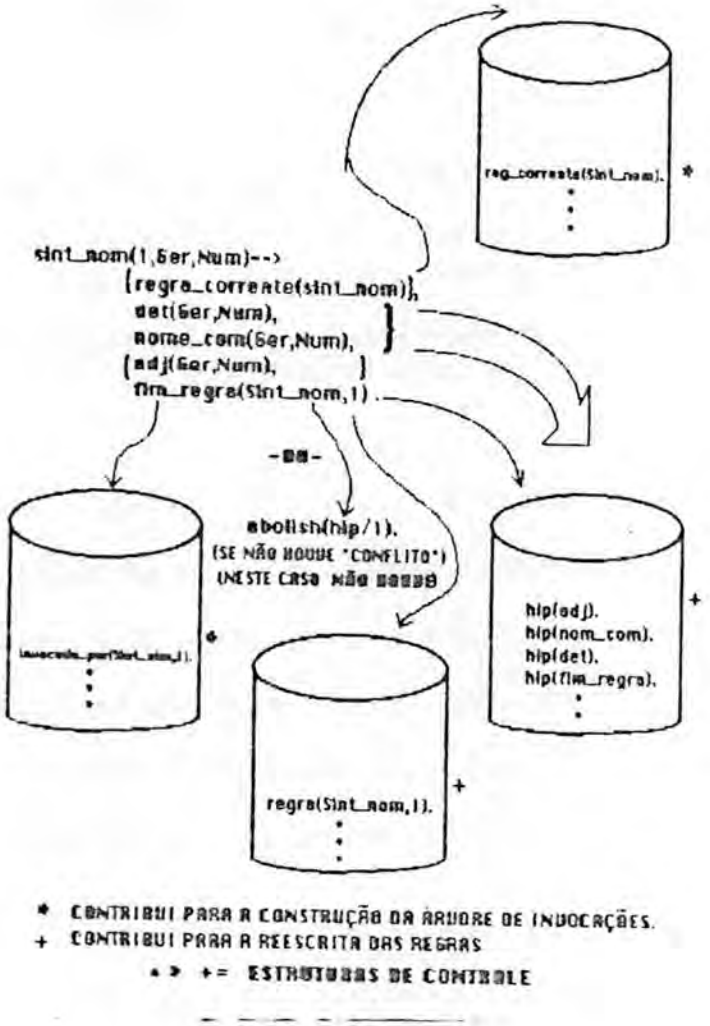


Figura 5.16 Visão geral da utilização das pilhas

O processo de controle para a reescrita das regras também é dificultado, particularmente, pela existência de regras do tipo:

```
* --> [].    (vazio)
```

Desta forma, quando uma palavra desconhecida é encontrada, como vimos, a primeira hipótese que o Tutor-Prolog gera é a de

que a palavra está mal escrita. Caso esta hipótese não suceda, deve ser verificado se a suposta categoria da palavra (hipótese que corresponde à regra corrente) permite a análise pelo vazio. Em caso positivo, esta opção de análise deve ser notificada através da adição dum facto à história da análise: "notif_vazio(*)".

Se, por exemplo, a análise dum sintagma iniciar em (R1), encontrar o primeiro vazio em (R2) e terminar em (R5), a análise da frase prosseguirá normalmente em (R6), se todos os elementos, entre (R2) e (R5), forem satifeitos pelo vazio. Caso contrário, todos os factos "notif_vazio(*)" devem ser retirados da base de dados e um novo processo de análise, que leve em conta esta situação, deve ser iniciado.

Estas informações a respeito da regra utilizada para a análise do terminal, juntamente com as informações obtidas através do diálogo com o aluno, possibilitam a transposição eficaz deste problema. Assim, se no reconhecimento condicionado (segunda tentativa) e no processo de aquisição de informações externas (durante o diálogo com o utilizador), o utilizador disser que a classe gramatical do vocábulo desconhecido não é igual a da hipótese apresentada pelo Tutor-Prolog, este terá que pesquisar todos os seus conhecimentos de modo a detectar a(s) última(s) regra(s) que foram satisfeitas pelo vazio, imediatamente antes de ser invocada a regra corrente. Caso não existirem regras nesta situação, a análise gramatical prossegue.

No caso de haver regra(s) em tais condições, todas as cláusulas referentes a essa(s) regra(s) são testadas para detectar se a categoria do novo vocábulo pode ser abrangida por uma dessas cláusulas. Caso se obtenha uma resposta afirmativa, o sistema é obrigado a retroceder e a recomeçar a análise gramatical. Logo, é necessário manter informações que descrevam quais as regras que foram satisfeitas (pelo vazio ou não) a cada instante e qual a regra que está sendo utilizada no momento, isto é, que se deseja satisfazer. Este processo é apresentado graficamente na figura 5.17.

Além de permitir o controle do tratamento do vazio, estas informações servem para a construção da "árvore de invocações" no final da análise de cada frase. Esta árvore indica, de modo gráfico, os pontos onde se deu o retrocesso e qual a sua causa, permitindo distinguir graficamente o processo de reconhecimento normal, do controlado. Nos exemplos seguintes esta representação gráfica será apresentada.

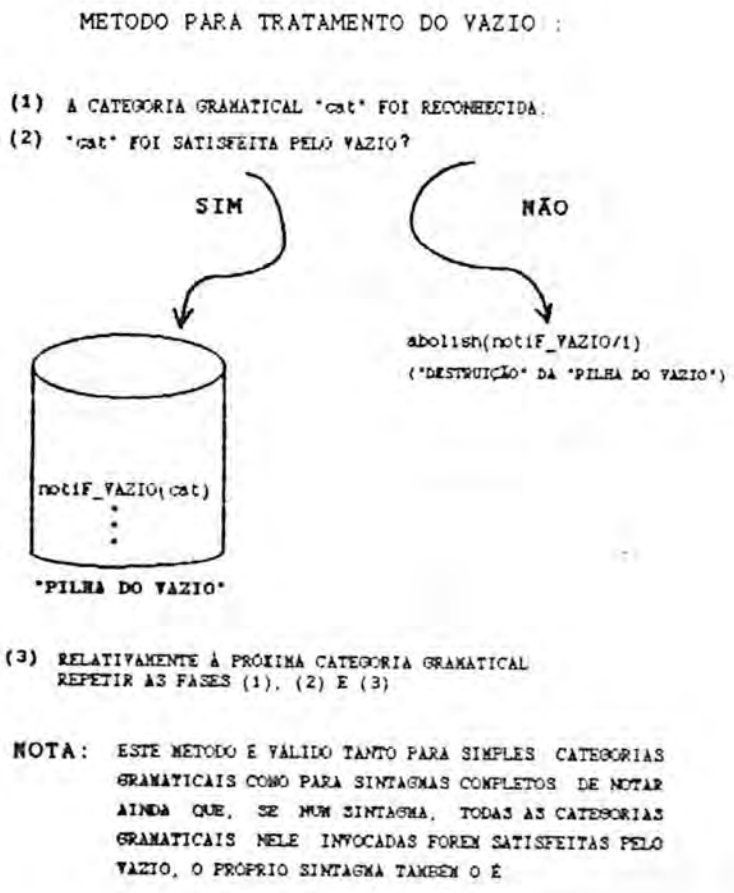


Figura 5.17 Tratamento do vazio

Assim, através das hipóteses geradas a partir das regras que o Tutor-Prolog possui e dos conhecimentos do aluno, o núcleo gramatical inicial vai evoluindo para melhor adaptar-se ao utilizador. As novas regras geradas passam a fazer parte do modelo da interface de cada aluno.

O processo de evolução da interface, através da sua adaptação ao modelo do aluno, é representado na figura 5.18.

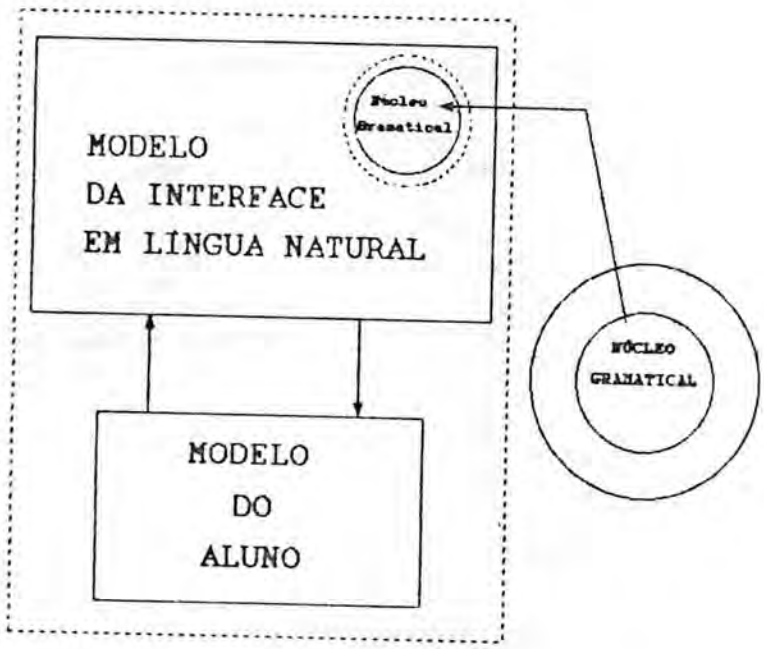


Figura 5.18 Modelo para a evolução gramatical (adaptado de [COE;LOP;VIC 88])

Para melhor compreensão da evolução, que ocorre nas regras do núcleo gramatical, passamos a apresentar uma sequência de exemplos práticos onde os menus são omitidos pois o objectivo é a apresentação da gráfica do processo de análise e a gramática resultante após cada processo de aprendizagem.

A figura 5.19 apresenta as regras da gramática que serão, de alguma forma, alteradas pelos exemplos apresentados.

```
sint_nom(1,Gen,Num) -->
     ( regra_corrente(sint_nom)),
     prep,
     det(Gen,Num),
     nome_prop(Gen,Num),
     ( fim_regra(sint_nom,1) ).

sint_nom(2,Gen,Num) -->
     ( regra_corrente(sint_nom) ),
     nome_com(Gen,Num),
     adj(Gen,Num),
     ( fim_regra(sint_nom,2) ).

sint_compls(1,Gen,Num):-
     ( regra_corrente(sint_compls) ),
     adj(Gen,Num),
     ( fim_regra(sint_compls,1) ).
```

Figura 5.19 Gramática utilizada no protótipo

A apresentação da sequência de exemplos a seguir tem por finalidade ilustrar a evolução, que ocorre na gramática apresentada na figura 5.19, devido ao processo de interpretação das frases que compõem cada exemplo.

```
>> O carro colorido é rápido

   Mapa da análise

   -> sint_nom(*) -> ¦   1a tentativa de análise da frase
                     ¦
  <------------------¦   retrocesso
  ¦
  ¦
   sint_nom(2) >> sint_verb(3) >> sint_verb1(1) >> sint_compls(1)

                                    2a. tentativa, onde ocorre um
                                    processo de reescrita da regra
                                    sint_nom(2,Gen,Num).
```

A regra "sint_nom(2,Gen,Num)" foi alterada devido ao acréscimo de uma cláusula que prevê a possibilidade da existência de um determinante antes de um substantivo comum.

```
sint_nom(2,Gen,Num) -->
     ( regra_corrente(sint_nom) ),
     det(Gen,Num),                          % acrescenta
     nome_com(Gen,Num),                     % uma nova
     adj(Gen,Num),                          % cláusula
     ( fim_regra(sint_nom,2) ).
```

Figura 5.20 Reescrita por acréscimo de cláusula

```
>> O carro rápido é colorido

   Mapa da análise

   sint_nom(2) >> sint_verb(3) >> sint_verb1(1) >> sint_compls(1)
```

Figura 5.21 Análise sintáctica em um só passo

Como a frase da figura 5.21 foi escrita após a do exemplo da figura 5.20 e tem a mesma estrutura da anterior, a sua análise é imediata, não sendo necessária a aprendizagem de palavras e de novas estruturas gramaticais.

```
>> O colorido carro é rápido.

   Mapa da análise

 -> sint_nom(*) -> ¦
                   ¦
   ¦<--------------¦
   ¦
   sint_nom(2) >> sint_verb(3) >> sint_verb1(1) >> sint_compls(1)
```

Neste caso, a evolução acontece através da reescrita da regra "sint_nom(2,Gen,Num)", que é dividida nas duas variantes "sint_nom2".

```
-----------------------------------------------------------------
     sint_nom(2,Gen,Num) -->
          { regra_corrente(sint_nom) },
          det(Gen,Num),
          { fim_regra(sint_nom,2) }.
          sint_nom2(X,Gen,Num).          % transformação
                                         % de uma regra através
     sint_nom2(2,Gen,Num):-              % do desmenbramento de
          { regra_corrente(sint_nom2) }, % de suas cláusulas
          adj(Gen,Num),
          nome_com(Gen,Num),
          { fim_regra(sint_nom2,3).

     sint_nom2(1,Gen,Num):-
          nome_com(Gen,Num),
          adj(Gen,Num),
          { fim_regra(sint_nom2,1) }.
-----------------------------------------------------------------
```

Figura 5.22 Reescrita por transformação da regra

Na figura 5.22 é apresentado o desmembramento da regra "sint_nom", para possibilitar a análise de construções como a do exemplo da figura 5.21, onde aparece a sequência: determinante, substantivo comum, adjectivo,... Ou a construção do exemplo da figura 5.22: determinante, adjectivo, substantivo comum, ...

```
-----------------------------------------------------------------
>> E rápido o carro colorido

   Mapa da análise

-> sint_nom(*) -> sint_nom3(*) -> ¦
                                  ¦
¦<--------------------------------¦
¦
sint_nom(2)>>sint_nom3(2)>>sint_verb(3)>>sint_verb1(1)>>sint_compls(*)
                                                        ¦
                                                        ¦
                                                    ¦<-----
                                              sint_compls(1)
```

Neste exemplo ocorre um processo de acréscimo de uma nova cláusula (não terminal) à regra "sint_compls" e a geração automática das regras "sint_compls1".

```
sint_compls(1,Gen,Num):-
     ( regra_corrente(sint_compls) ),
     adj(Gen,Num),
     ( fim_regra(sint_compls,1)
     sint_compls1(X,Gen,Num).    % chamada da nova regra

sint_compls1(2,Gen,Num):-        % parte já existente
     ( regra_corrente(sint_compls1) ),
     det(Gen,Num),
     nome_com(Gen,Num),
     adj(Gen,Num),
     ( fim_regra(sint_compls1,2)).
                                          % nova regra gerada
sint_compls1(1,Gen,Num):-                 % para o tratamento
     invocacao_vazio(sint_compls1,1). % do vazio
```

Figura 5.23 Reescrita e acréscimo de nova regra

A ordem em que a evolução ocorre é importante para a estratégia usada na solução do problema. Se, por exemplo, a frase doa figura 5.23 fosse a primeira frase escrita na sessão e em seguida fosse escrita a frase da figura 5.22, os conhecimentos adquiridos durante a análise da frase da figura 5.23 seriam utilizados também para a análise da frase da figura 5.22.

Neste caso, os passos seguidos para a análise da frase da figura 5.23 continuariam os mesmos. O exemplo da figura 5.22, no entanto, provocaria a geração do gráfico abaixo e, consequentemente, a alteração de outras regras.

```
sint_nom(0) >> sint_verb(*)
---------------------------
!                          !
sint_nom(0) >>  sint_verb(1) >> sint_compls(1) >> sint_compls1(1)
```

Uma nova regra, destinada à análise de um sintagma nominal, seria acrescentada no final do conjunto das regras já existentes para esse fim, de acordo com o exposto na figura 5.24.

```
sint_nom(0,Gen,Num):-
        invocacao_vazio(sint_nom,0).
```

Figura 5.24 Acréscimo de uma nova regra

Se, no entanto, a frase da figura 5.22 fosse a primeira da interacção, as regras existentes seriam apenas as da figura 5.19 e o processo de evolução apresentado no exemplo da figura 5.22 não seria alterado.

5.3 Alternativa de utilização

Como já foi sublinhado na introdução deste Capítulo, foram construídos dois processos de aprendizagem. No primeiro, o aluno colabora com o tutor na tentativa de solucionar um conflito. No segundo processo, que passaremos a apresentar em seguida, são assumidas as hipóteses geradas pelo Tutor-Prolog.

Se determinados alunos não estiverem dispostos a colaborar quando ocorrerem conflitos, o Tutor-Prolog pode assumir como verdadeiras as suas hipóteses fortes. Neste caso, a interacção prosseguirá sem a componente interactiva e as hipóteses do tutor serão assumidas como verdadeiras.

No final da sessão o aluno tem acesso a um resumo com todas as transformações ocorridas nas regras e nos conhecimentos do Tutor-Prolog como é apresentado na figura 5.25. O aluno pode,

então, alterar as situações que considerar inadequadas. Se isto ocorrer, a situação de diálogos auxiliados por menus é restabelecida.

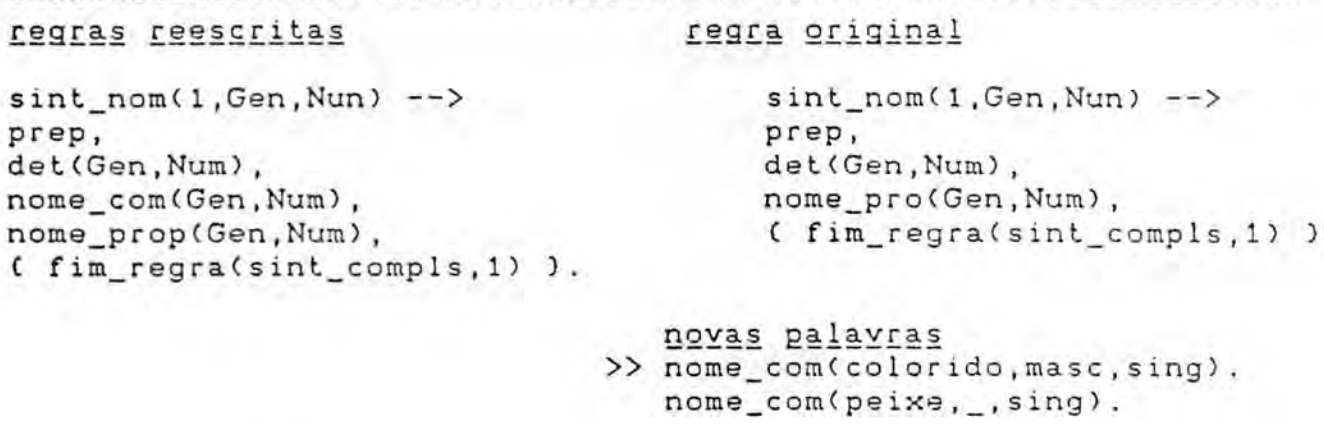


Figura 5.25 Prioridade às hipóteses do Tutor_prolog

Supondo que, de acordo com o exemplo da figura 5.8, a palavra coloridas tenha sido aprendida de forma inadequada e, como aparece no exemplo da figura 5.25, o aluno desejar desfazer o equívoco, o processo interactivo é retomado.

A substituição de uma categoria gramatical ou de um ou mais argumentos de uma cláusula no dicionário é um processo simples. Entretanto, a reescrita inadequada de uma regra, que pode ter sido ocasionada devido ao tutor ter optado por uma hipótese incorrecta, é um processo um pouco mais complexo. Para isso, o sistema deve manter o histórico dos passos efectuados durante a reescrita, a fim de poder restabelecer (através do retrocesso) a

regra original ou parte dela, e, então, iniciar um novo processo de reescrita auxiliado pelos conhecimentos do aluno.

Consideramos que as investigações, realizadas no domínio do Tutor-Prolog, relativas à geração do modelo da interface, são mais uma alternativa a ser explorada, visando a obtenção de interfaces em língua natural mais eficientes e agradáveis.

Estudos complementares na área da sintaxe e o seu alargamento para a área da semântica [ROS;LOP 88], [GRO;APP; MAR;PER 87], [BAL 86] e da pragmática [BAL 86], contribuirão certamente para que o assunto desenvolvido neste capítulo possa ser melhor explorado.

6. TRABALHO FUTURO

O estado actual do Tutor-Prolog pode ser aperfeiçoado através do alargamento de sua área de abrangência tanto para a obtenção de um ambiente de resolução de problemas através da programação em Lógica, como para o ensino da linguagem Prolog.

A evolução visando o ensino completo da linguagem Prolog é aqui apresentada dentro do que designamos por "ensino e formação profissional". São vários os tópicos que podem ser desenvolvidos neste sentido. Estes tópicos serão agora comentados, bem como as alterações necessárias para a transformação do Tutor-Prolog num produto comercial. Assim, primeiramente são apresentadas as alterações que consideramos mais importantes para aperfeiçoar o Tutor-Prolog, tendo em conta apenas os seus objectivos actuais. Em segundo lugar são focadas as alterações que visam o alargamento dos seus objectivos.

6.1 Aprendizagem

Como foi apresentado nos capítulos anteriores, ficam armazenadas as alterações feitas às regras do Tutor-Prolog, resultantes do processo de aprendizagem, no modelo de cada aluno. Elas representam o processo de adaptação do Tutor-Prolog ao aluno.

As alterações estão classificadas em dois grupos: as que foram perfeitamente reconhecidas e organizadas dentro dos conhecimentos já existentes e as que, por falta de conhecimentos do Tutor ou do aluno, ficaram classificadas em categorias fictícias (desconhecidas). Assim, todo o conhecimento adquirido é utilizado para a adaptação das regras do Tutor às necessidades do aluno, ao nível da cópia activa para cada utilizador.

Ao nível da interface em língua natural, o Tutor-Prolog apresenta uma **abordagem inovadora** em termos de IT, pois difere dos modelos tradicionais usados em sistemas como o NEOMYCIN [CLA 84], o SCHOLAR [CAR 70] e o WHY [STE;COL;GOL 77]. No seguimento das experimentações descritas no Capítulo 5, que chamaremos por conveniência de **modelo A**, propomos novas formas de evolução das gramáticas, às quais chamaremos de modelos B, C, D e E, conforme [COE;LOP;VIC 88].

Tendo em conta o modelo A, que já foi apresentado detalhadamente, serão agora tratados os novos modelos, isto é, as formas alternativas de se realizar a evolução do núcleo gramatical inicial e central. As formas de evolução que apresentámos, através do caso da língua natural, podem ser adaptadas para os demais componentes do sistema. A organização em modelos possibilita uma melhor apresentação didáctica das ideias que possuímos acerca da evolução dos programas computacionais.

Assim, trataremos agora dos modelos B, C, D e E. Nestas propostas, não são levados em conta os aspectos referentes à

viabilidade computacional para a sua utilização prática.

Modelo B

Como no modelo A, o ponto de partida é a gramática G, que evolui para uma gramática G1, idêntica a g(G,U), como consequência da interacção com o aluno U. No entanto, neste modelo, a nova gramática (G1) é usada como gramática de entrada para cada nova interacção. Este processo é apresentado sucintamente na figura 6.1.

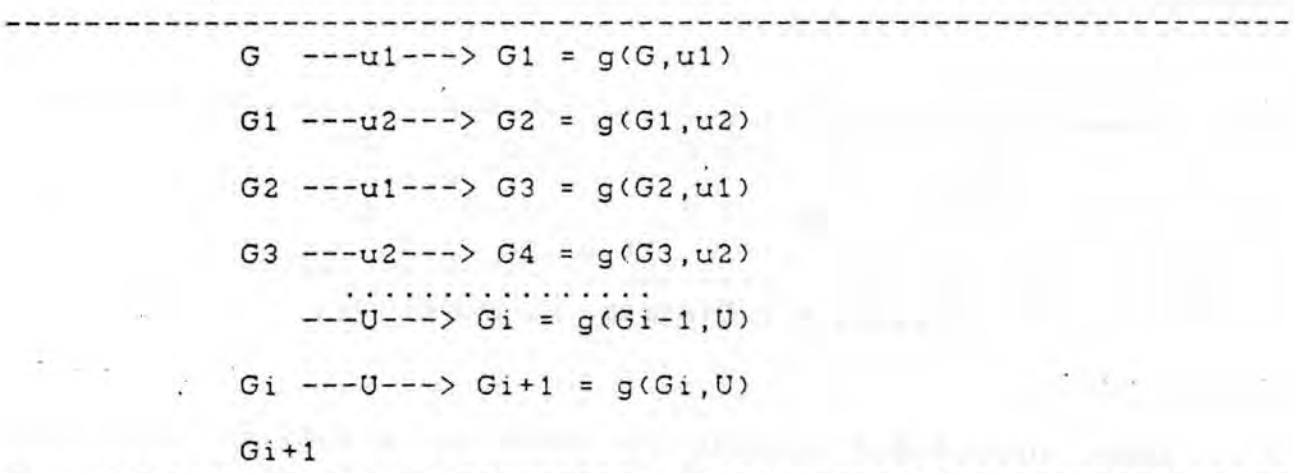


Figura 6.1 Diagrama do Modelo B

Neste modelo, o núcleo inicial é sempre transformado e usado como entrada para a nova interacção. Cada utilizador pode, portanto, alterar a gramática.

Para que o modelo A, adoptado no Tutor-Prolog, possa evoluir para a forma B, basta apenas armazenar a gramática após cada interacção, transformando-a, assim, no núcleo gramatical do

próximo utilizador. Como o Tutor-Prolog corre em microcomputadores com MS/DOS, que admite apenas um utilizador, não existe o problema da simultaneidade de utilizadores.

Modelo C

Neste modelo, o núcleo gramatical alterado é sempre armazenado, servindo como entrada para a próxima sessão do mesmo utilizador. Assim, após algumas sessões existem tantas gramáticas quantos forem os utilizadores do sistema, mais a gramática inicial. Ou seja: n+1 gramáticas de acordo com a figura 6.2.

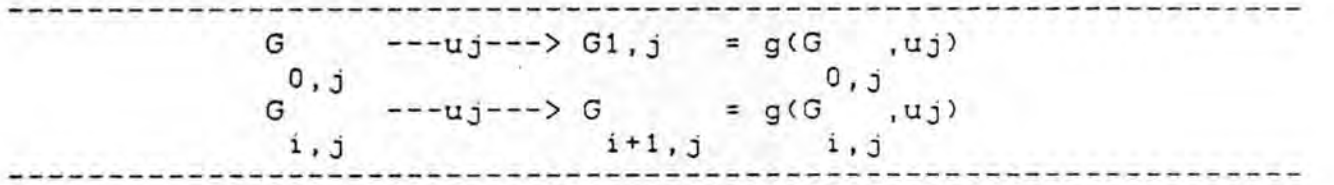


Figura 6.2 Diagrama do Modelo C

Cada utilizador possui uma cópia da gramática totalmente adaptada para si próprio, sendo mantido intacto o núcleo gramatical inicial. Naturalmente, o sistema pode ter vários utilizadores simultaneamente.

Tanto para o modelo A, como para o modelo C, é necessário apenas que após cada sessão de trabalho, o aluno mantenha uma cópia actualizada, em disquete, da sua gramática.

Modelo D

No modelo D existe um núcleo inicial de gramática que é alterado, de sessão para sessão, através de um processo de aprendizagem por experiência. Este processo é activado após o encerramento de uma sessão de trabalho. A aceitação ou rejeição das transformações depende da adopção de regras heurísticas. Por exemplo, seleccionar as transformações mais usadas, seleccionar as transformações adoptadas por vários utilizadores, dar maior crédito às alterações feitas por utilizadores que demonstraram ter bons conhecimentos de Português, etc. É de salientar que, para a adopção da última alternativa, torna-se necessário o modelo do utilizador.

Durante uma sessão interactiva várias alterações podem ser feitas (substituição de regras e inclusão de novas regras) de forma "on-line" de acordo com o exposto no modelo A. O conjunto de acções realizadas descreve as transformações que devem ser feitas no código da gramática t(g(G,U))=G como é apresentado na figura 6.3.

A função tog representa o conjunto de transformações que devem ser feitas em G. Ou seja, o conjunto de acções que aplicadas a g(G,U) reconstituem a gramática inicial G para cada utilizador U.

No final de várias sessões com os utilizadores, a interface pode "olhar" para cada memória privada com o objectivo de gerar o

novo código do núcleo gramatical, que conterá as alterações comuns que não entraram em conflito.

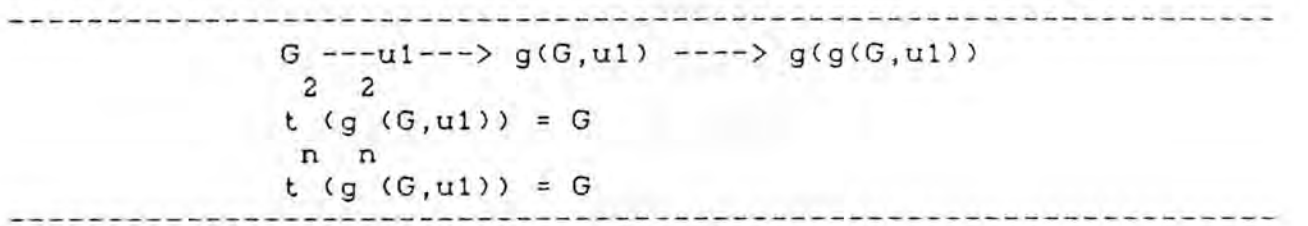


Figura 6.3 Diagrama do modelo D

A função tg(G,u1) representa as transformações que devem ser feitas na gramática de trabalho do utilizador u1 no fim da primeira sessão de trabalho. A função $tg^{n}(G,u1)$ representa as transformações, que devem ser feitas no código da gramática G, para restaurar o estado da mesma no final de n interacções com o utilizador u1.

$T=(tg^{ni}(G,Ui)$; onde ni representa o número de interacções do utilizador u_i)

Após a actualização do núcleo gramatical inicial, são mantidas nas gramáticas individuais apenas as regras que não foram incorporadas no núcleo gramatical principal.

Assim, com a passagem do modelo A para o modelo D, a evolução do sistema será contínua, isto é, dar-se-á tanto no núcleo gramatical, como nos sistemas que o utilizam. Na figura 6.4 é apresentada a evolução do núcleo gramatical e do Tutor-

Prolog. Neste caso, apenas o Tutor-Prolog utiliza o núcleo gramatical. Em [COE;LOP;VIC 88] é suposto que o núcleo gramatical sirva a vários programas aplicativos, a sistemas periciais, a tutores, e a sistemas de consulta a base de dados. No entanto, tanto o núcleo gramatical, como os programas que o utilizam, evoluem no tempo.

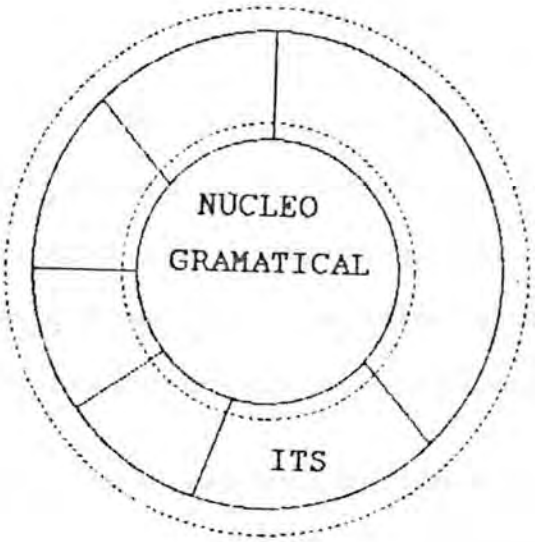


Figura 6.4 Evolução integrada do sistema

Modelo E

Neste modelo o núcleo gramatical alterado é sempre armazenado, servindo como entrada para a próxima sessão.

A diferença fundamental entre o modelo D e o modelo E consiste na eliminação das gramáticas individuais, no modelo E, após

a alteração do núcleo inicial principal. Assim, só as alterações seleccionadas de acordo com um determinado critério é que são utilizadas, pois passam a incorporar o núcleo inicial que, desta forma, evolui.

Os modelos apresentados podem ser cracterizados de modo sucinto através dos seguintes aspectos: o núcleo inicial, as classes de utilizadores (de acordo com a exploração realizada), as capacidades de aprendizagem e o processo de evolução.

- No Modelo A, o núcleo inicial não é alterado. Existe uma cópia do núcleo para cada aluno e as alterações realizadas somente são válidas durante a sessão. Não existem critérios para a alteração do núcleo gramatical e não existe memória das regras gramaticais alteradas, nem dos diálogos realizados.

- No Modelo B, o núcleo inicial é alterado sem critérios, existindo apenas uma cópia do núcleo, facto que impossibilita a sua utilização por mais de um aluno ao mesmo tempo. As alterações são de carácter permanente, pois são armazenadas a cada sessão. O sistema possui memória das alterações gramaticais realizadas, mas estas não são associadas a cada utilizador.

- No Modelo C, o núcleo inicial não é alterado, existindo uma cópia do núcleo para cada aluno. As alterações feitas nas cópias individuais têm carácter permanente e são realizadas sem critérios. A evolução é permanente para cada utilizador e o modelo prevê a existência de memória para as alterações realiza-

das.

- No Modelo D, o núcleo inicial é alterado segundo certos critérios, existindo uma cópia do núcleo para cada utilizador. As alterações são de carácter permanente, de acordo com os critérios que geram as novas regras. A memória das regras alteradas é mantida após cada sessão. Durante as sessões cada aluno altera o seu código individual e, ao terminar todas as sessões, é feita a actualização do núcleo central de acordo com certos critérios mantidos no modelo do utilizador (por exemplo, nível de conhecimento da língua Portuguesa, histórico da interacção, etc.). A partir da nova versão do núcleo central é feita a compatibilização das gramáticas individuais.

- No Modelo E, o núcleo inicial é alterado de acordo com critérios, existindo também uma cópia da gramática para cada utilizador. A aprendizagem é de carácter permanente, havendo memória das alterações realizadas. Após o fim das sessões individuais, o núcleo inicial é alterado de acordo com os critérios utilizados para a selecção das alterações. Estas são feitas individualmente e devem passar a fazer parte do núcleo inicial. Após a alteração do núcleo inicial, as gramáticas individuais são destruídas.

No caso da interpretação da língua natural escrita é importante que exista uma capacidade para a geração automática de exemplos, com vista a auxiliar o aluno durante os processos

iniciados pelo tutor para a aquisição externa de conhecimentos. Ou seja, quando o aluno ou o Tutor possuir dúvidas sobre a categoria gramatical de determinada palavra, o Tutor-Prolog passaria a gerar exemplos visando auxiliar a decisão do aluno. Os exemplos poderão ser gerados a partir da regra que provocou o conflito e/ou que deu origem às hipóteses de classificação. As frases exemplos seriam geradas de forma ascendente (semelhante ao processo da geração de termos Prolog para o ensino da linguagem), a partir do dicionário. Esta capacidade é fundamental para a aquisição de conhecimento semântico e pragmático.

Ainda, e em relação às frases escritas em língua natural, partindo-se das actuais capacidades do tutor, é importante que o processo inverso também seja desenvolvido, no que se refere a geração automática de exemplos Prolog a partir de frases em Português. Ou seja, o aluno deve escrever frases em Português, e o Tutor deve analisá-las e transformá-las em termos Prolog. Estas alterações são obtidas facilmente, pois, como podemos ver no exemplo da figura 6.5, possuem um grau de complexidade inferior às apresentadas anteriormente.

-> O João ama a Maria.

ama(joao,maria).

Figura 6.5 Geração de exemplos

O método de aprendizagem através de modelos e de exemplos pode ser utilizado, ainda, para alargar as capacidades do módulo que auxilia a escrita de programas (Cápitulo 4). Para isso, o tutor terá que ter possibilidades para gerar um novo modelo para o problema proposto, a partir da solução apresentada pelo aluno. Assim, é evidente que o tutor necessita poder verificar se a nova estrutura corresponde a uma solução correcta para o problema. Trabalhos neste sentido tem sido realizados por [GAR;VER 84], [JOH;ELL;SOL 85].

No entanto, nenhum destes trabalhos propõe a geração de novos modelos (regras, equadramentos, redes semânticas, ...) para descrever a solução do aluno. Utilizam apenas modelos contendo os erros frequentemente cometidos pelos novos programadores, de modo que a solução do aluno é enquadrada num dos modelos e, a partir dele, feito o seu redirecionamento para o modelo considerado correcto pelo Tutor. O que propomos é diferente, pois o Tutor-Prolog construiria um novo enquadramento contendo a solução proposta pelo aluno e tentaria prová-la, para concluir se está ou não correcta. O novo modelo seria gerado a partir do momento em que as regras existentes passassem a ser inadequadas à solução do aluno. Este ponto seria mantido para futuras alterações, que viessem incorporar o novo modelo, ou, então, para reorientar o raciocínio do aluno. A reorientação poderá ser feita a partir do comando "ajudar" (Capítulo 4)). Na figura 6.6 são apresentadas duas soluções possíveis para um mesmo problema.

Cálculo da média entre três números.

Primeira solução possível gerada pelo aluno A:

```
media(N1,N2,N3,M):- M is (N1+N2+N3)/3.
```

Segunda solução possível gerada pelo aluno B:

```
media([L],M):- soma(L,S,_), M is S/3.

soma([],S,_).
soma([N|R],S,Saux):- Saux is S+N,
                     soma(R,Saux,_).
```

Figura 6.6 Formas diferentes de solucionar um mesmo problema

Outras soluções seriam ainda possíveis, como por exemplo a obtenção de modo dinâmico do número de elementos que compõem a lista L.

O que pretendemos é que o Tutor-Prolog tenha capacidade para gerar e incorporar os modelos correspondentes a todas as soluções correctas que os alunos apresentarem.

O programa AM de [LEN 82] realiza aprendizagem de conceitos matemáticos (teoria dos conjuntos). Tanto os conceitos iniciais como os adquiridos estão representados em enquadramentos.

Ainda, na linha de Garijo e Verdejo [GAR;VER 84], é importante alargar as capacidades do tutor para a realização da análise de textos que contenham a descrição lógica dos programas propostos pelo utilizador. Com esta capacidade o Tutor-Prolog

poderá evoluir no sentido da programação automática. É nossa intenção que a análise do texto escrito pelo aluno seja feita de maneira interactiva. Ou seja, com o tutor tendo capacidades para ajudar o aluno a definir logicamente o seu problema. É natural que o domínio de intervenção do tutor seja bem definido. Nesta linha de acção - análise da correcção de programas e interpretação de textos (definições lógicas de problemas) - existe já bastante trabalho realizado. Por exemplo, [MAT 82], [SLE;HEN 82], [MIL 82], [BUR 82], [SEL 82], [KIM 82], [O'SH 82] e [THO 87] entre outros. Assim, o Tutor-Prolog poderia ser alargado de forma a possuir a capacidade de gerar a solução de um programa em Prolog, a partir de um texto em Português contendo a definição desse problema.

6.2 Arquitectura de um Tutor Inteligente

No Cápitulo 2 descrevemos com pormenor a arquitectura actual do Tutor-Prolog, e observámos que a sua organização evoluiu durante a fase de desenvolvimento dos três protótipos. Face à arquitectura tradicional, foram melhorados alguns módulos e adicionados outros que julgámos imprescindíveis. Mas, durante todo o nosso trabalho tivemos ocasião para reflectir sobre as limitações e para ponderar sobre as melhores pistas a encarar no trabalho futuro.

De facto, o triângulo típico Modelo do Domínio - Modelo do

Aluno - Modelo da Comunicação de um TI deve dar lugar a um quadrilátero, onde um novo modelo, o da Aprendizagem, tem um lugar especial, como se apresenta na figura 6.7.

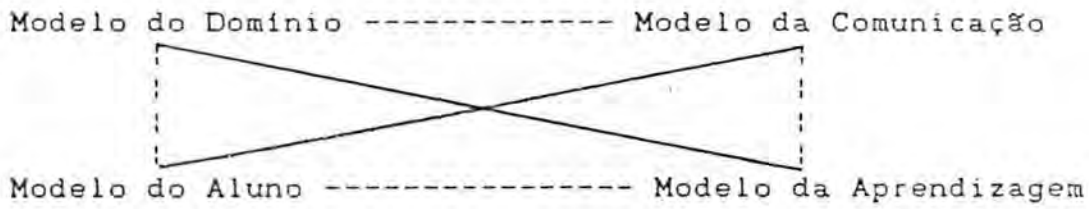


Figura 6.7 Arquitectura de um Tutor Inteligente

Na secção anterior apresentámos alguns argumentos a favor desta alteração. Nos modelos apresentados, para a evolução do modelo da comunicação, são utilizadas várias estratégias de aprendizagem. No entanto, a sua inclusão deve ser mais ampla, e se tal ocorrer modificará de facto a estrutura habitual de um TI.

O modelo da aprendizagem tem por objectivo propiciar uma espécie de negociação entre o conhecimento do domínio (também chamado de material instrucional, modelo ideal, modelo do perito, solucionador de problemas, etc) e o conhecimento do aluno (modelo do aluno), pois ambos desempenham um papel activo, tendo perguntas a serem respondidas e informações a serem trocadas. Logo, ambos ensinam e aprendem. Este intercâmbio faz-se através do modelo da comunicação (modelo da interface). Ou seja, no modelo do domíno encontra-se o conhecimento que deve ser comunicado e no modelo do aluno encontra-se o conhecimento que deve ser aprendido ou que deve ser corrigido pelo tutor. Logo, a troca de

conhecimentos entre o modelo da aprendizagem e o do domínio envolve também o modelo do aluno. Naturalmente, esta ligação só acontece em sistemas que permitem a aprendizagem no nível do modelo do domínio (caixa branca).

O módulo de comunicação tem como objectivo clarificar a apresentação do material instrucional, facilitar o uso do sistema e torná-lo atractivo. Para isso, são utilizadas capacidades conversacionais como menus, processamento de língua natural ou pseudo-natural, e capacidades de representação gráfica. Assim, a comunicação faz a ponte entre a representação interna do tutor inteligente e a representação externa para o aluno. A organização do diálogo, realizado entre o módulo de comunicação e o aluno, embora dependa da estrutura do domínio (caixa preta ou branca), deve possuir capacidade de aprender com a interacção.

Por sua vez, a organização do módulo de aprendizagem depende de como é conduzido o processo de diálogo no tutor. Do ponto de vista tradicional, as arquitecturas dos TI's permitem formas determinadas de diálo, como por exemplo: o método em que o tutor conduz o diálogo de ensino (SCHOLAR [COL 75]); as perguntas e respostas (SOPHIE-I [BRO;BOR 75]); o método socrático (WHY [STE; COL 77]) e a simulação (STEAMER [STE 83]). Na nossa proposta de uma nova arquitectura, defendemos que se deva introduzir uma maior flexibilidade nas interacções, através do recurso às técnicas de aprendizagem.

Segundo Michalsky [MIC 86], as estratégias de aprendizagem

são as seguintes: implantação directa do conhecimento, instrução, dedução, analogia e indução. As duas primeiras podem ser classificadas de formas explícitas de aquisição de conhecimentos e as três últimas, como implícitas.

No modelo da aprendizagem temos a forma de como essa deve ser conduzida, isto é, de como e quando o modelo do utilizador, da comunicação e do domínio devem ser alterados. Os modelos podem ser gerados de forma individual ou por classes de utilizadores (por exemplo, o mesmo modelo para todos os utilizadores que consultam a mesma base de dados), a sua validade pode ser permanente ou temporária (apenas uma sessão).

Os modelos da comunicação e da aprendizagem apresentam características comuns. Por exemplo, tanto um quanto o outro, podem conter conhecimentos que intersectam o modelo do aluno, podem ser uma parte do modelo do aluno, ou podem ser diferentes do modelo do aluno. No entanto, em nossa opinião os modelos são complementares, pois representam sempre as diferenças existentes entre o conhecimento do sistema e do utilizador. A principal diferença entre o modelo do aluno e os demais modelos é que as informações de todos os modelos exepto as do aluno são do tipo permanente, isto é, podem ser alteradas mas nunca removidas.

Assim, em resumo defendemos que o desempenho de um tutor inteligente deve estar organizado ao redor da aprendizagem, isto é, dependerá do sucesso com que as novas informações, provenientes do conhecimento que o aluno possui sobre o assunto que o

tutor está ensinando, são representadas no modelo do aluno e, se for o caso, anexadas ao modelo da comunicação ou ao modelo do domínio.

6.3 Técnicas de programação

Durante as observações práticas ficou patente a necessidade de desenvolver mecanismos que fossem capazes de ordenar as cláusulas que descrevem o conhecimento do aluno sobre determinado problema e o modo de resolvê-lo. Em consonância com os objectivos actuais do Tutor-Prolog optamos pela solução mais livre que foi apresentada no Capítulo 4. No entanto, no futuro são necessários mecanismos mais gerais, como alargar o tutorial para abranger o ensino profissional da linguagem Prolog. Neste sentido, as capacidades de depuração semelhantes às desenvolvidas por Shapiro [SHA 82] e por [PER 84] ou, ainda, os métodos utilizados no programa depurador do projecto PITS - "Prolog Intelligent Tutoring System" - [LOO;ROS 87], tornam-se necessárias.

Neste item abordaremos também o ensino de técnicas de programação [BRN 88], [STE;SHA 87], pois o tema permite a apresentação das linhas gerais de evolução que pretendemos para o Tutor-Prolog e, de certa forma, envolve os itens anteriores. Como já visto no item 6.2, o ensino de exercícios, que abarca o uso de tecnicas de programação, é um dos tópicos do Tutor-Prolog que necessita de ser alargado em trabalho futuro. Assim, somos da

opinião de que um tutor automático necessita de representar internamente várias técnicas de programação adaptadas para a linguagem que pretende ensinar. A título de exemplo, a utilização do corte pode ser ensinada da seguinte maneira: "se o procedimento teste executar satisfatoriamente, então executar caso1; senão, executar caso2". Ou seja, como está apresentado na figura 6.8.

```
cond :- teste, !, caso1.
cond :- caso2.
```

Figura 6.8 Utilização do corte

O mesmo pode ocorrer com a introdução do comando "fail", isto é, através de programas que introduzem a noção de repetição, como o da figura 6.9.

```
impressao_de_dados:-
        fact(X),
        write(X), nl,
        fail.

impressao_de_dados.

fact(1).
fact(30).
```

Figura 6.9 Introdução do comando fail

Além do uso do corte e da falha, a recursão também é um conceito importante na introdução de técnicas de programação. No caso particular da recursão, as observações indicam, que a sua introdução deve ser feita a partir da repetição, que em geral, é facilmente compreendida pelos alunos. Os programas que requerem a

entrada de dados a partir do operador levam o aluno a generalizar, através da repetição, o processo de leitura dos dados.

No Anexo 2 são apresentadas as soluções dos alunos para os problemas que envolviam o uso de acumuladores. Os mesmos problemas podem vir a ser utilizados para o alargamento do ensino da recursão e da manipulação de estruturas, como as listas. Na experiência realizada, os alunos limitaram-se a manter a interacção através da repetição. Pensamos que a solução recursiva é menos eficiente em termos de controle, mas pode ser mais eficiente em termos da estrutura construída. Por outro lado, é evidente que a análise de programas que contenham a recursão, ou mesmo a repetição, exigirá maiores capacidades do tutor. Neste sentido, podem ser utilizadas as tradicionais bibliotecas de erros ou, memo, uma exigência prévia que obrigue o aluno a informar qual a técnica utilizada na construção do programa (repetição, recursão, de procura), tais como as propostas por Brna para depuração interactiva de programas Prolog [BRN 88].

6.4 Ensino da linguagem Prolog para profissionais

Os sistemas educativos e os sistemas para a formação profissional possuem certas características comuns. As aplicações da IA na formação profissional auxiliam o estudante na aprendizagem de um processo ou procedimento específico. Neste aspecto, as regras fundamentais e os objectivos básicos são os mesmos de uma apli-

cação com a finalidade educacional, desenvolvida com as metodologias da Inteligência Artificial [VIC 87a]. Assim, os resultados desta investigação podem ser utilizados, por exemplo, para o desenvolvimento de tutores para a formação profissional. Isso envolve, evidentemente, novas estratégias de ensino, maior flexibilização do tutor e a utilização de uma linguagem adequada aos utilizadores interessados neste tipo de ensino. Na actual arquitectura do Tutor-Prolog, o terceiro nível de complexidade de ensino pode ser utilizado para este fim. Para viabilizar este objectivo, são necessárias várias alterações, das quais passaremos a descrever apenas as principais.

6.4.1 Controle do ensino

Como já foi apresentado no Capítulo 3, as estratégias tutoriais podem ser representadas através de um gráfico com três dimensões, as quais correspondem ao nível de complexidade assumido para ensinar um conteúdo, a um determinado bloco instrucional, que contém lições de acordo com a organização didáctica assumida para o ensino da linguagem, e a uma determinada posição (lição) dentro da árvore que descreve cada bloco instrucional (tempo).

O nível de complexidade que deve ser utilizado na apresentação do material instrucional é obtido através das informações contidas no modelo do aluno. A estratégia de ensino do terceiro nível é menos tutorial e, consequentemente, o tutor transfere o

controle da interacção prioritariamente para o aluno. Assim, o Tutor-Prolog apresenta a definição formal do conteúdo que está sendo abordado, cabendo ao aluno a exploração desta definição. A sua principal característica é a aposta na **simulação**, que é uma componente fundamental dos sistemas CBI. Quando o aluno comete erros, o tutor intervém com o **diagnóstico** utilizado para a **detecção do erro** e com as possíveis **soluções** para a eliminação da falha. Pensamos que a estratégia para a intervenção deve continuar a ser a mesma até aqui adoptada, ou seja, uma intervenção imediata apenas nas falhas que envolvem a sintática e a semântica declarativa. O diagnóstico é composto pela mensagem de erro, pela localização do erro, pela(s) regra(s) utilizada(s) na sua detecção e pela orientação visando a correcção da falha. Quando o tutor gera mais de que uma hipótese de solução, para um erro ocorrido na escrita do termo Prolog, apresenta-as ao aluno e solicita a sua intervenção para que a solução adequada seja obtida. O processo de depuração deste nível deve envolver mais a participação do aluno e, os métodos como os de Shapiro [SHA 87] e Pereira [PER 84] são recomendados.

Toda a correcção ou ajuda oferecida pelo Tutor ao aluno apoia-se nos conhecimentos adquiridos durante a interacção [VIC 88], isto é, durante a escrita de exemplos, na solicitação de exemplos, na simulação ou na solicitação de explicações por parte do aluno. Com estas informações, o tutor gera conhecimentos (dinâmicos) que passam a fazer parte do modelo do aluno. A partir deste modelo são feitos o diagnóstico, a orientação e a planifi-

cação da apresentação do material instrucional, de acordo com as linhas gerais previstas num dos três níveis de complexidade de ensino existentes.

Actualmente, o terceiro nível de ensino no Tutor-Prolog encontra-se delineado do ponto de vista da idealização e da concepção, ou seja, destina-se à formação profissional e portanto deve apresentar características, uma linguagem e um conteúdo instrucional, condizentes com a população a que se destina.

6.4.2 Taxonomia assumida para o ensino da linguagem

Segundo a nossa experiência no ensino da linguagem Prolog a alunos universitários e ainda nos cursos de formação profissional [VIC 86], a organização do ensino da linguagem Prolog deveria, para fins didácticos, ser distribuida como se indica na figura 6.10.

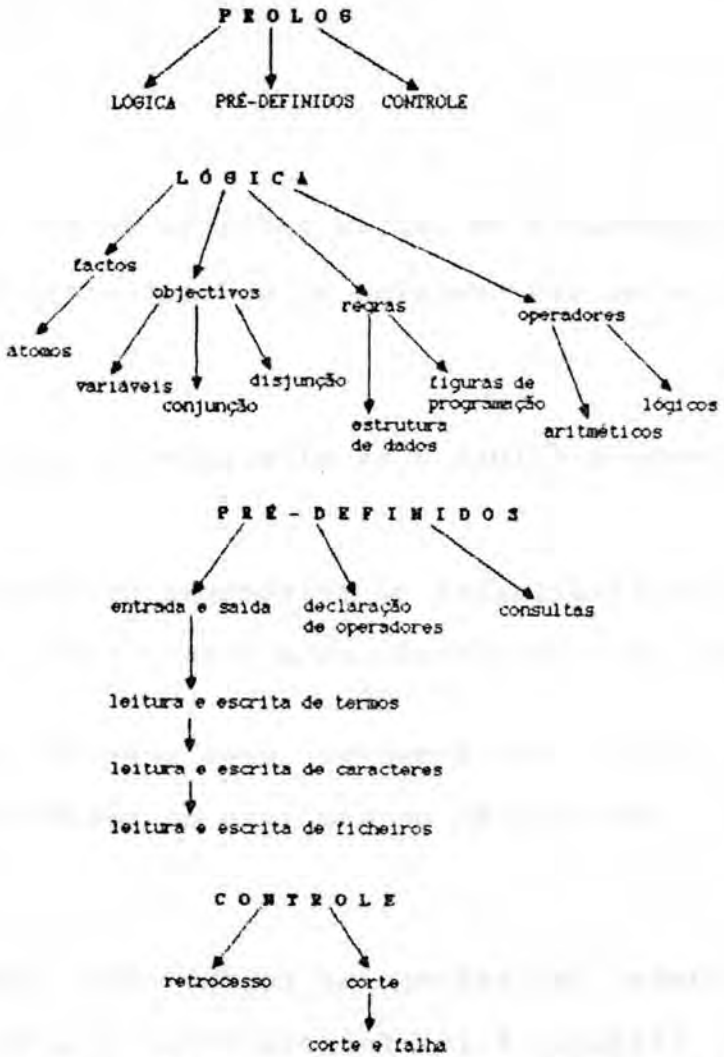


Figura 6.10 Taxonomia para o ensino profissional

Defendemos a introdução do ensino dos comandos de entrada e de saída e o uso do corte ("cut") e da falha ("fail"), assuntos que até aqui ainda não tinham sido abordados. Nestes casos, a capacidade de simulação (execução do mesmo programa, por exemplo, com ou sem corte) é um aspecto importante para o ensino neste nível de complexidade, e será necessário reforçar os exemplos referentes à representação do conhecimento, pois estes alunos, normalmente, já conhecem os conceitos básicos da programação. Há também a necessidade de criar um tutorial para ensinar a utilizar os diversos "pacotes" que compõem o ambiente computacional disponível, pois a motivação que leva estes utilizadores a desejarem

conhecer a programação em lógica apoia-se em aplicações concretas, necessárias no seu trabalho ou no seu currículo escolar.

6.4.3 A arquitectura actual e as alterações propostas

Algumas das facilidades já existentes no Tutor-Prolog foram planeadas tendo em consideração esta proposta de alargamento.

O uso do corte, por exemplo, está previsto dentro do meta-interpretador (com exepção para o caso do retrocesso inteligente).

A facilidade de espiar as regras que compõem o meta-interpretador (figura 6.11) foi concebida visando o ensino de nível 3.

```
                                          remover
                                          listar
                                        >>espiar
                                          executar
-> variável.

     variavel(V) -->
          maiuscula(H),
          resto_palavra(T)
          ;
          sublinha(A),
          resto_palavra([H|T]).
```

Figura 6.11 Definição de variável ao nível do meta-interpretador

A definição das regras que meta-interpretam a linguagem, tal como está apresentado na figura 6.12, pode assim ser observada pelo aluno, que passa a aprender a linguagem a partir da obser-

vação das regras que compõem o programa que a interpreta [ABE;SUS 85].

Outro aspecto importante é a escolha da "linguagem" (mais, ou menos técnica) a ser utilizada para o ensino (alunos com bons conhecimentos de matemática, lógica e de outras linguagens de programação). Ela deve ser mais formal, logo, mais precisa [COL 85], [COH 85]. Uma lição destinada ao ensino do conceito de átomos poderá vir a ter o conteúdo instrucional apresentado na figura 6.12.

Um átomo tem a forma **p(t1, ..., tm)**, onde **p** é um predicado com aridade **m** e cada **ti(i=1, ..., m)** é um termo.

Um termo é uma variável, uma constante ou tem a forma **f(t1,..., tk)** onde **f** é uma função com aridade **K** e cada **ti(i=1,...,K)** é um termo.

Represente logicamente a frase:

O João ama a vida.

Forneça o termo:

Figura 6.12 Exemplo de uma lição de nível 3

A intervenção (análise, mensagens e diagnóstico) também deve assumir uma linguagem mais formal.

6.5 Transferência da experiência do Tutor-Prolog para outros sistemas

A realização de estudos visando a transferência do modelo de

concepção do Tutor-Prolog para tutores que visem o ensino de outras linguagens, incluindo as tradicionais, também tem interesse. Como os princípios que fundamentam o Tutor-Prolog são os princípios gerais da Pedagogia, da Didáctica, da Psicologia e da Informática, os problemas de adaptação das ideias apresentadas ficam reduziados às características particulares de cada linguagem de programação e do ambiente computacional disponível. Ou seja, as outras linguagens também podem ser ensinadas em vários graus de complexidade, consoante os objectivos que o curso pretenda alcançar. O curso pode ter o seu conteúdo instrucional representado de acordo com as estratégias de ensino apresentadas no Capítulo 3. Quanto aos modelos cognitivos, amplamente utilizados no ensino automatizado das linguagens da IA, a sua utilização em IT's que visem o ensino de linguagens tradicionais parece ser perfeitamente possível. De facto, os modelos cognitivos são também utilizados em tutores destinados ao ensino da Matemática INTEGRATE, WEST, BUGGY e o QUADRATIC e da Geografia SCHOLAR. Ou seja, a sua utilização está ligada ao desenvolvimento de tutores inteligentes e não ao conteúdo que um determinado tutor pretende ensinar. Por fim, a meta-interpretação e a depuração inteligente de programas já é amplamente utilizado em sistemas como o PROUST, destinado ao ensino da linguagem Pascal, o BIP destinado ao ensino da linguagem BASIC e o SPADE destinado ao ensino da linguagem LOGO. Entretanto, a depuração de um programa Prolog difere bastante de programas em Lisp ou em Pascal. Num programa Prolog não são encontradas as palavras-chaves do Pascal, nem as funções

do Lisp. Um programa Prolog tem uma leitura declarativa e procedimental. Um programa Prolog pode ser não determinístico e, nesse caso, o depurador deve ter em conta todas as soluções possíveis. Ainda, um programa Prolog pode ter o controle conduzido por comandos como a falha e o corte.

Por outro lado, a componente de ensino também é uma peça importante para a informação e a formação interna dos utentes de um grande edifício de serviços. Assim, um tutor que conheça a organização da instituição e as suas regras pode, por exemplo, fornecer as informações e a ajuda necessária aos seus funcionários para resolverem dificuldades burocráticas, e em particular iniciar os novos funcionários à cultura dessa instituição. O conhecimento sobre a organização é o módulo que corresponde ao perito do sistema. O conhecimento que o módulo perito deve conter pode ser sintetizado da seguinte maneira: tipo da organização, características do modelo burocrático adoptado, departamentalização, organograma, funcionários e aspectos de conhecimento geral da organização (objectivos específicos, pessoal chave, formulários, segurança social, vantagens, etc). Assim, seu conteúdo instrucional será composto pela estrutura da organização, pelas leis a serem respeitadas, pelas operações de consulta às diversas componentes estruturais instaladas, etc.

Com este conhecimento o tutor poderá, por exemplo, auxiliar os funcionários na organização do seu horário de trabalho, ou na escolha do período para o gozo das suas férias. Poderá informar,

também, a respeito das regalias sociais (serviços de saúde, alimentação, desportos, ...) que a organização dispõe. Ainda, o tutor poderá estar capacitado a prestar auxílio aos visitantes (uma espécie de relações públicas) da organização.

Logo, o sistema tutor de uma organização tem como finalidade auxiliar as pessoas relacionadas com a organização e, nesse contexto, qualquer utilizador pode solicitar auxílio ao perito em causa. É natural que os objectivos dos vários utilizadores sejam diferentes. No entanto, mesmo dentro do âmbito de uma organização, é possível estabelecer classes de utilizadores. As classes podem ser divididas em visitante, funcionários e peritos. Os critérios utilizados para essa divisão são o objectivo e o grau de conhecimento sobre a organização em causa. Neste último item temos que observar se o utilizador possue conhecimentos ou se os quer adquir.

Os visitantes constituem um tipo de utilizadores esporádico e que, normalmente, pretendem informações gerais sobre a organização. Nestes casos, a actuação do tutor é bastante simplificada e não é necessário gerar ou guardar um modelo do visitante.

No caso dos funcionários, os objectivos podem variar bastante. Pensamos que neste caso o sistema poderá ter um modelo base no qual serão introduzidas as **adaptações** necessárias para cada utilizador que sair do processo comum projectado para a classe.

Para a classe dos peritos na organização o sistema terá que

possuir capacidades de aprendizagem que permitam reter o conhecimento que lhe é transmitido.

Estabelecidas as classes gerais de utilizadores, ao iniciar uma sessão, o sistema deverá caracterizar o objectivo da consulta para poder determinar a classe a que pertence um utilizador particular.

O modelo da cada utilizador poderá seguir o adptado no Tutor-Prolog, isto é, partir da existência de modelos padrões iniciais, que no caso do tutor organizacional representam, por exemplo, o modelo de como se comporta um funcionário novo e de como se comporta um funcionário antigo, ao invés dos modelos do Tutor-Prolog, que represenatm o aluno com desempenho escolar fraco, médio e bom. A partir desses modelos iniciais são feitas as adptações para cada utilizador particular. Ainda, partindo da ideia de que toda a organização tem uma cultura própria e de que esta cultura é expressa através de uma linguagem particular, o conhecimento do perito pode vir a ser representado, como no Tutor-Prolog, através de regras de produção, que descrevam formalmente esta linguagem.

Como no tutor organizacional está prevista a interacção com o perito, que irá transmitir os conhecimentos sobre uma organização em particular, toda a experiência realizada no domínio da aprendizagem, no Tutor-Prolog, servirá para o desenvolvimento do módulo de aquisição de conhecimentos. Assim, o tutor organizacional adquire conhecimentos, através da interacção com um perito,

sobre uma dada organização e ensina, aos demais utilizadores, os conhecimentos adquiridos.

A inclusão de um tutor inteligente dentro da estrutura de uma organização informatizada ("Edifício Inteligente") pode ser observada na figura 6.13.

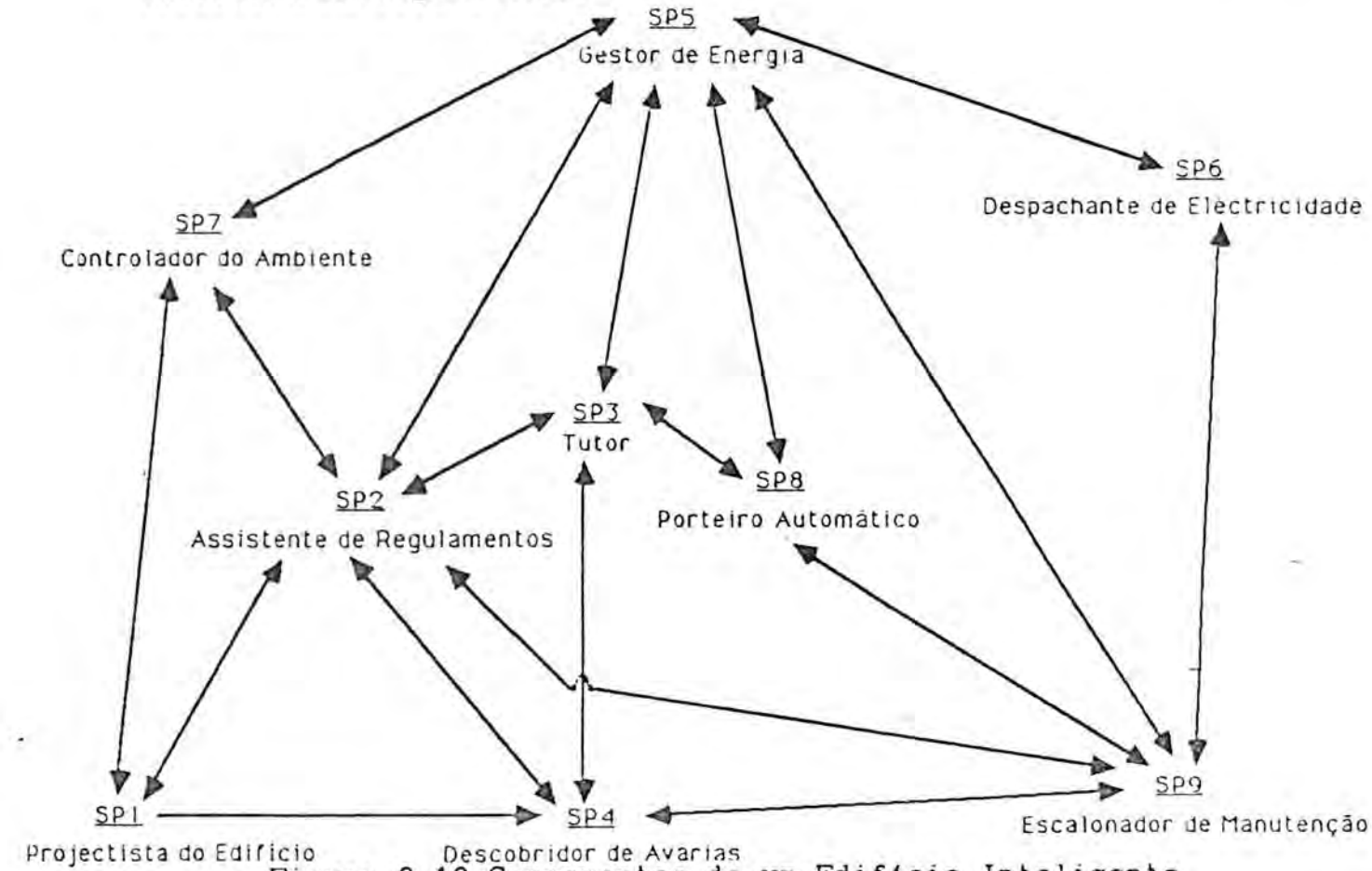


Figura 6.13 Componentes de um Edifício Inteligente

Por último, a transformação do Tutor-Prolog num produto comercial exige a obtenção de um código compilado, o que requer ajustes, na forma da utilização dos comandos "retract" e "assert", às normas exigidas pelo compilador Arity/Prolog". A compilação agilizaria o tempo de resposta do sistema como um todo. Do ponto de vista da ferramenta utilizada, a adopção do

programa de gestão da tela do Arity/Prolog e o uso do "ratinho" também contribuiriam para a diminuição e a simplificação do código dos programas que compõem o Tutor-Prolog e, consequentemente, para a diminuição do tempo de resposta, que é um dos aspectos mais importantes para um sistema comercial.

As alterações aqui propostas contribuirão para alargar a amplitude do Tutor-Prolog, no sentido de também vir a ensinar a Programação em Lógica e, particularmente, a Linguagem de Programação Prolog, a alunos de diferentes faixas etárias e com diferentes conhecimentos sobre lógica e sobre a própria linguagem.

7. CONCLUSÕES

Os seres humanos recorrem geralmente aos seus conhecimentos anteriores quando aprendem novas matérias. Do mesmo modo, os TI's que não são capazes de explorar o modelo do conhecimento dos seus utilizadores e de aprender com eles comportam-se de forma inadequada durante o processo de ensino e de acompanhamento dos seus alunos. Ora, no projecto do sistema Tutor-Prolog, nomeadamente nas fases da sua idealização, conceptualização e desenvolvimento, estes aspectos foram determinantes para organizarmos a sua arquitectura, pois o recurso ao modelo do aluno ajuda a prever e a impedir erros ou a detectá-los, e também a corrigí-los de forma mais fácil.

Este trabalho começou a ser idealizado em 1982, na Universidade Federal do Rio Grande do Sul, quando nos emvolvemos na UFRGS num projecto de Ensino Assistido por Computador (SISCAI) para linguagens de programação convencionais (FORTRAN e BASIC). O conteúdo instrucional do SISCAI foi elaborado pelos professores das respectivas disciplinas, através de um conjunto de programas desenvolvidos especialmente para esta finalidade (organização das telas, ligação entre as lições, ligações entre as respostas e os retornos, ...). Neste sistema a actuação do aluno resumia-se à escolha de alternativas que continham as soluções para os problemas apresentados.

A participação neste projecto propiciou-nos o estudo dos aspectos principais do ensino assistido, tais como a geração do material, o planeanto e a apresentação deste material (o aspecto do professor e o aspecto do aluno). Durante este estudo, fomos conduzidos naturalmente a ponderar o recurso às técnicas e instrumentos da IA.

A partir de 1983 iniciámos o desenvolvimento em Prolog de um sistema para a criação de material instrucional ("courseware"). Este sistema já utilizou facilidades da IA, tais como a interpretação da língua natural [VIC 85]. Complementarmente, participámos (1984-1985) em experiências visando o ensino da linguagem Prolog a pessoas de faixas etárias e formações distintas, que não conheciam linguagens de programação ou que estavam ingressando em algum curso universitário [VICC 85;86].

Destas experiências foi possível constatar:

- a falta de professores na área de programação em lógica;
- a inexistência de cursos adequados às necessidades dos vários grupo de alunos (em empresas, em universidades, e em escolas);
- as dificuldades apresentadas pelos alunos que já conheciam outras linguagens de programação;
- as dificuldades apresentadas pelos alunos que não conheciam linguagens de programação; e,
- a inexistência de um ambiente computacional adequado à introdução do Prolog no ensino secundário.

Este período de incubação, por assim dizer, do Tutor-Prolog

levou-nos a concluir que um ambiente ideal para o ensino devia levar o aluno a criar modelos que reflitam a estrutura e a composição do assunto que estiver sendo estudado.

No que se refere à conceptualização e ao desenvolvimento do Tutor-Prolog, a nossa investigação envolveu a geração de três protótipos em ambiente de microcomputador com MS-DOS.

Os testes com o primeiro protótipo demonstraram a necessidade de evoluir para uma segunda versão, devido a deficiências encontradas na gestão da comunicação, feita através da língua natural escrita, e no controle do tutorial, que era pouco sensível à interacção. Assim, o tempo de resposta era demasiado longo, pois os alunos necessitavam escrever durante toda a interacção e nem sempre as construções sintácticas utilizadas eram reconhecidas pelo Tutor. Durante as primeiras experiências, os alunos manifestaram também o interesse em gerar as suas próprias bases de dados e em consultá-las através do uso de um subconjunto da língua natural escrita.

O segundo protótipo caracterizou-se pela comunicação auxiliada por menus e pela introdução da possibilidade de consultar o sistema através da utilização da língua natural escrita. Apenas nestes casos (consulta) é que a interacção passou a ser feita através da língua natural escrita, que pode ser auxiliada por menus. Isto possibilitou a flexibilização do controle das sessões instrucionais e o alargamento do protótipo para viabilizar a geração de bases de dados pelo aluno.

A aprendizagem também foi abordada no que respeita a formação do modelo do ensino (aperfeiçoamento do uso das regras estratégicas) e no modelo da interação. Foram, ainda, tratadas a a depuração de programas, a simulação e o diagnóstico orientado.

Este segundo protótipo apresentou deficiências no que se refere ao ensino das figuras básicas de programação (decisão, repetição e recursão) que envolviam o manuseio de estruturas de representação da informação (cadeias, listas, árvores e enquadramentos).

O terceiro protótipo passou a cobrir os nossos objectivos no que se refere ao ensino de um subconjunto da linguagem Prolog. Neste foram investigados aspectos relativos ao acompanhamento da escrita de programas, envolvendo a detecção e a orientação na correcção dos erros e o auxílio na obtenção de uma solução para o problema proposto. Foi investigado o uso de enquadramentos para a representar os programas propostos pelo Tutor-Prolog ao aluno.

Todo o trabalho foi baseado na observação prática, sendo as várias versões do protótipo dirigidas ao ensino secundário (dos 14 aos 16 anos), e testadas na Escola No. 1 dos Olivais em Lisboa. A observação foi dividida em três fases distintas:

- na primeira fase os alunos foram observados utilizando o Tutor-Prolog, isto é, a parte tutorial, a parte de desenvolvimento de programas (depuração) e a consulta a base de dados;
- na segunda fase os alunos foram observados apenas na utilização dos módulos de depuração e de consulta. Isto é,

o módulo tutorial foi omitido. Os alunos recebiam os conhecimentos a respeito da linguagem a partir da escrita de seus programas (dados e regras). Nesta etapa tornou-se necessária a intervenção de um agente externo (professor);

- na terceira e última etapa os alunos observados desenvolveram as mesmas actividades (problemas a solucionar) realizadas nas etapas anteriores, mas utilizando o programa "Folha de Cálculo".

As conclusões desta experiência podem ser apreciadas detalhadamente em [ZAM 88a] e ainda de forma resumida no Anexo 3. Limitamo-nos aqui a apresentar algumas conclusões gerais que foram apresentadas pelos alunos e por nós no final de cada observação.

Quanto a utilização do Tutor-Prolog:

- Relativamente ao conteúdo instrucional, foi possível observar que os alunos do 9o. ano realizaram uma exploração mais efectiva dos dados armazenados. Os alunos do 7o. ano restringiam-se mais aos problemas propostos pelo Tutor e não efectuavam perguntas de carácter exploratório. Neste aspecto o uso do Tutor-Prolog apresentou melhores resultados do que o uso do programa Folha de Cálculo".

- No desenvolvimento e no uso de regras, os alunos realizaram as consultas em tabelas e os cálculos directos (busca dos dados na BD e realização de cálculos) com grande facilidade, mas apresentaram certa dificuldade na geração de problemas que envolviam procedimentos intermediários (divisão do problema em subproblemas) ou procedimentos recursivos.

- Os alunos apresentaram interesse especial pelos resultados gráficos, demonstrando ser importante que um ambiente de programação proporcione também estas facilidades aos seus utilizadores. Neste áspecto particular o uso do programa Folha de Cálculo apresentou melhores resultados do que a ligação prevista entre o Tutor-Prolog e o programa "Open Access".

- O Tutor-Prolog apresentou melhores resultados no auxílio à busca da solução de problemas, isto é, nas operações de depuração. Aqui, o aspecto principal foi a manutenção do diálogo (interacção,) entre o aluno e o tutor, em situações de erro.

Quanto ao desenvolvimento do Tutor-Prolog:

Os principais aspectos teóricos envolvidos na idealização, na concepção e no desenvolvimento dos TI's (ensino e adaptação) foram investigados e aprofundados no decorrer do trabalho prático efectuado.

Durante as duas primeiras observações procuramos definir a taxonomia para o ensino da Programação em Lógica que melhor se adaptasse a estes utilizadores. Foi assim que surgiu a actual organização do modelo de ensino do Tutor-Prolog. Estas observações permitiram também a inclusão das facilidades de edição que actualmente compõem o Tutor-Prolog e a utilização de várias formas de comunicação homem-máquina (menus, símbolos, língua natural e gráficos).

No item relativo ao ensino das regras, a ordenação das cláusulas foi o tema que mais teve de ser trabalhado.

Foi possível verificar ainda, que a nossa meta de ensinar um subconjunto da linguagem Prolog a partir de exemplos expressos através da língua natural escrita (Português) foi bem sucedida, pois no final do tutorial os grupos observados já conseguiam especificar e programar os seus próprios problemas (parte lógica e parte dos dados).

Os resultados observados foram melhores quando os alunos usaram o Tutor-Prolog completo, isto é, com um ambiente e ensino assistido, depuração, consulta e auxílio do Tutor-Prolog nas operações de manuseio de ficheiros. As experiências que envolveram o uso directo do módulo de depuração ou do próprio interpretador Prolog não conduziram a bons resultados, devido ao desconhecimento, por parte dos alunos, dos comandos do sistema operativo, a não familiarização com o uso de editores de texto e a dificuldade na compreensão das mensagens em inglês.

A utilização de uma gramática, escrita em Prolog, para a análise, a interpretação e a depuração das cláusulas escritas foi uma experiência bem sucedida apesar das dificuldades introduzidas pelo processo de depuração. Para ultrapassar essas dificuldades, foi adoptada uma estratégia em que a análise modificava-se no decorrer de uma mesma cláusula ("left-right" ou "right-left). A ideia da utilização de regras gramaticais para a interpretação da linguagem Prolog, ao nível de um TI, é defendida também por Ross

e Lewis [ROS;LEW 88].

Com relação à comunicação homem-máquina, verificámos que uma interface deve possibilitar a utilização de um leque variado de formas de comunicação. Observámos que a utilização de uma única forma (ou menus, ou língua natural, ou códigos, ...) torna o processo monótono e pouco flexível com o aluno quase sempre apresentando dúvidas não previstas. O último protótipo observado demonstrou que a possibilidade de recorrer a diferentes maneiras para comunicar desejos e conhecimentos entre o Tutor e o aluno é o processo que melhor resultados apresenta. É de ressaltar, ainda, o ganho em vivacidade que a interacção obtém com a utilização de cores, gráficos e a formatação dinâmica da tela do terminal.

No que se refere à interface é de salientar o avanço que o Tutor-Prolog proporciona em termos da utilização da aprendizagem auxiliada por menus ao nível do subconjunto da língua natural escrita, se levarmos em conta os sistemas tradicionais como o NEOMYCIN [CLA 84], SCOLAR [CAR 70] e WHY [STE;COL 77], que apenas aceitam construções sintácticas e semânticas correctas e cujos sistemas não possuem flexibilidade para a aprendizagem no que se refere a comunicação em língua natural.

Da mesma forma, o Tutor-Prolog é pioneiro na área do ensino da linguagem de programação Prolog, pois actualmente existe apenas o Active Prolog Tutor (APT) que, no entanto, não apresenta as principais características presentes em ICAI (modelos cognitivos,

estratégias de ensino e capacidades para a aprendizagem). Mesmo as experiências já realizadas (Imperial College) ou em curso (Weizmann Institut), apesar da inegável importância para o estudo do ensino da linguagem Prolog, não podem ser consideradas como sendo TIs.

Assim, concluímos que um tutor inteligente necessita de **incentivar** a exploração dos conteúdos instrucionais; possuir vários planos de ensino e um modelo para guiar a apresentação do conteúdo instrucional; ser **sensível** às necessidades do utilizador adequando-se às necessidades individuais; **dominar**, o máximo possível, o assunto que ensina; possuir **conhecimento** para tentar resolver situações não previstas nas regras existentes e aprender com tais situações; possuir características de **ensino assistido** (carácter tutorial); possuir mecanismos para a **depuração inteligente** e a orientação na detecção e eliminação de falhas; permitir a **simulação** automática e conduzida de problemas; possuir **memória retroactiva** que descreva o raciocínio (passos) utilizado pelo aluno e pelo tutor durante a exploração de determinado conteúdo instrucional. Todas essas características observadas, praticamente, orientaram a concepção, o projecto e o desenvolvimento do Tutor-Prolog prossibilitando alguns avanços quando comparado com as experiências similares anteriormente citadas.

Assim, este trabalho, pioneiro em Portugal e no Brasil, esclarece a diversidade de formas de utilização dos computadores nas escolas no âmbito do Projecto Minerva e contribui para o

desenvolvimento de áreas como:

- Tutores Inteligentes, particularmente no que se refere ao ensino através de exemplos e a aprendizagem através do uso de modelos. A planificação dinâmica do material instrucional dentro de diferentes níveis de complexidade. A geração de modelos para representar os conhecimentos do aluno. O uso de estratégias de ensino.

- Linguagem de programação Prolog, no que se refere à meta-interpretação em particular para a análise das listas, ao retrocesso inteligente, à consulta de bases de dados e à organização das cláusulas nos programas. E, principalmente no seu ensino e divulgação.

- Ao ensino da programação e a aspectos da interface homem-máquina.

- Inteligência Artificial, pois trata de aspectos como a meta-interpretação da linguagem Prolog, da modificação de programas através da reescrita automática de regras visando a sua evolução no sentido de melhor servir aos seus utilizadores e da modificação do plano de execução do sistema de acordo com o desenvolvimento da interação.

Resumindo, as principais contribuições técnicas e científicas que este trabalho de investigação apresenta são as seguintes: a adequação e a evolução dos programas através da aquisição de conhecimentos e da transformação das regras que os compõem; a utilização de várias formas de comunicação homem-máquina (menus, língua natural, gráficos e palavras-chaves) dentro de um mesmo ambiente; a organização do ensino dentro de três vectores principais, a saber, os blocos instrucionais (material de ensino), as estratégias de ensino e o tempo em que as acções ocorrem; e o desenvolvimento de sistemas que interagem com base em hipóteses.

Portanto, este trabalho aborda aspectos de interesse para os investigadores das áreas de Engenharia Informática e de Inteli-

gência Artificial, em particular no que se refere ao desenvolvimento dos Tutores Inteligentes, e que se constitui numa alternativa para a introdução do uso dos computadores nas escolas secundárias.

BIBLIOGRAFIA

[ABE;SUS 85]
ABELSON, H.;SUSSMAN, J., Structure and interpretation of computer programs, London, MIT, 1985.

[ALL;PER 80]
ALLEN, J. ; PERRAULT, C., Analyzing intention in utterances, Artificial Intelligence, 15, p. 143-178, 1980.

[AND 85]
ANDERSON, J., Production systems, learning, and tutoring in D. Kahr ; P. Langley ; R. Neches (eds.),Self-modifyng production systems:Models of learning and development, Cambridg, MIT,1985.

[AND;TEI 85a]
ANDERSON, J.;TEISER, B., The LISP tutor, Byte, no.10, 1985a, p 159-175.

[BAL;STU 84]
BALLARD, B.;STUMBERGER, D., Sematic Acquisition in TELI:a transportable, user-customized natural language processor, Colling-84, Stanford, 1984, p 20-29.

[BAL 86]
BALLARD, B., User specification of syntactic case frames in TELI, a transportable, user-customized natural language processor, Colling-86, Bonn, 1986, p 454-460

[BAE;BEA;ATK 76]
BARR, A.;BEARD, M.;ATKINSON, R.C., The computer as a tutorial laboratory: the Stanford BIP Project, International of Man-Machine Studies, 8, 1976, p 567- 596.

[BAR;FEI 82]
BARR, A.;FEIGENBAUM, E., The handbook of artificial intelligence, vol. 2, Los Altos, William Kaufmann, 1982.

[BAU;SAT 87]
BAULAY ; SATHCOTT, Computer teaching programming: an introdutory survey of the field, in Artificial Intelligence and Education, (ed) Lawler;Yazdani, Ablex Publishing, USA, 1987.

[BEG;HOG 87]
BEGG, I. M.; HOGG, I., Authoring systems for ICAI, Artificial Intelligence & Instruction applications and methods, Addison-Wesley, USA, 1987, p 323-346.

[BER 87]
BERGHEL, H., A logic framework for the correction of errors in elctronic documents. Information Processing and Management no. 23 (5), 1987, p 477-494.

[BLO 87]
BLOC, L., Computational Models of Learning, Springer-Verlag, Berlin, 1987 (ed.).

[BOU;SOT 87]
BOULAY, B.; SOTHCOTT, C., Computers teaching programming: an introductory survey of the field, Artificial intelligence and education (ed.) Lawler;Yazdani, Ablex, USA, 1987.

[BRA 86]
BRAZDIL, P., Transfer of knowledge between systems: some teaching and learning strategies, Universidade do Porto, maio 1986.

[BRA 87]
BRAZDIL, P., Aprendizagem através da resolução de problemas, Revista de Informática da API, vol. 6 no. 3, maio 1987.

[BRA;EIS 88]
BRAYSHAW, M;EISENSTADT, M., Adding data and procedure abstraction to the transparent Prolog machine, the Open University, 1988.

[BRN;BRA;BUN;DOD;ELS;FUN 88]
BRNA, P.;BRAYSHAW, M.;BUNDY, A.;DODD, T.;ELSOM-COOK, M.;FUNG, P., An overvieu of Prolog debugging tools, Research Paper 398, Department of Artificial Intelligence, Edinburgh, 1988.

[BRN;BUN;DOD;EIS;LOO;PAI;SMI;SOM 88]
BRNA, P.;BUNDY, A.; DODD, T.;EISENSTADT, M.;LOOI, C.;PAIN, H.; SMITH, B. ; SOMEREN, M., Prolog programming techniques, Department of Artificial Intelligence, Edinburgh, 1988.

[BRO;BUR 78]
BROWN ; BURTON, Diagnostic models for procedural bugs in basic mathematical skills, Cognitive Science, 1978, p. 155-192.

[BRO;BUR 79]
BROWN ; BURTON, An investigation of computer coaching for informal learning activities, Intrnational Journal of Man-Machine Studies, 11, 1979, p.5-24.

[BRO 79]
BROWN, Writing interactive compilers and interpreters, Chichester, John Wiley, 1979.

BROWN, J.;BURTON, R., Pedagogical, natural language and knowledge engineering techniques in SOPHIE I,II and III, in SLEGMAN, D., BROWS J., Intelligent tutoring systems, Academic Press, 1981.

[BUN 83]
BUNDY, A., The computer modelling of mathematical reasoning, Academic Press, 1983.

[BUR 82]
BURTTON, R., Diagnosing bugs in a simple procedural skill, in Intelligent Tutoring systems, (eds.) by Sleeman;Brown, Great Britain, Academic Press, 1982, p 157-182.

[BUR;BRO]
BURTTON, R.;BROWN, Diagnostic models for procedural bugs in basic mathematical skills, Cognitive Science, no. 2, p 155-192, 1976.

[BUR 86]
BURSTEIN, M., Concept formation by incremental analogical reasoning and debugging, Machine Learning: an artificial intelligence approach, Vol. II, Morgan Kaufman, 1986.

[CAR 70]
CARBONELL, J., AI in CAI: An Artificial Intelligence approach to computer assisted instruction, IEE Transactions on Man-Machine Systens, MMS - 11, 4, 1970.

[CAR;LAN 87]
CARBONEL, J.; LANGLEY, P., Machine Learning, in Encyclopedia of Artificial Intelligence, vol 1, (eds.) by SHAPIRO, S., 1987.

[CAR;HAY 83]
CARBONELL, J.;HAYES,P., Robust parsing using multiple construction-specifc strategies, in BLOC,L. (ed.),Natural Language Parsing Sysrems, Springer-Verlag, Berlin, 1983.

[CAR;COL 83]
CARBONEL, J.;COLLINS,A., Natural semantics in artificial intelligence, Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford CA: Stanford Research Institute, 1983.

[CAR;GOL 77]
CARR;GOLDSTEIN, Overlays: A theory of modeling for computer aided instruction, Artificial Intelligence Laboratory, Memo 406, Mssachusetts Institute of Technology, 1977.

[CER 77]
CERL (Computer-based Education Research Laboratory), Demonstration of the PLATO IV computer based education system, final report, University of Illinois, Urbana-champaing, Illinois, 1977.

[CHA 86]
CHAN, Implementation of microcomputers in elementary schools: Asurvey and evaluation, Department of Computer Science, University of British Columbia, may 1986.

[CLA 82]
CLANCEY, W. J., Tutoring rules guiding a case method dialogue, in Intelligent Tutoring systens, (eds.) by D. Sleeman and J. S. Brown, Great Britain, Academic Press, 1982, p 201-222.

[CLA 82]
CLANCEY, W., Exploration of teaching and problem-solving strategies, 1979-1982, Depto. of Computer Science, Stanford University, may 1982.

[CLA 86]
CLANCEY, W., Qualitative student models, Anual Reviews of computer Science, vol. 1, Anual Reviews, Palo Alto, California, 1986, p. 381-450.

[CLA 87]
CLANCEY, W., Knowledge_based tutoring, IJCAI 87, tutorial no.4, 1987.

[CLA 87]
CLANCEY, W., Methodology for building an intelligent totoring system, Artificial Intelligence & Instruction applications and methods, Addison-Wesley, USA, 1987, p 193-227.

[CLO 81]
CLOCKSIN, W.;MELLISH, C., Programming in Prolog, Springer-Verlag, Berlin, 1981.

[COE 79]
COELHO, H., A program conversing in portuguese providing a library service, LNEC, 1979.

[COE;COT;PER 80]
COELHO, H.; COTTA, J.; PEREIRA,L.M., How to solve it with prolog, Lisboa, LNEC, 1980.

[COE;COT 88]
COELHO, H.; COTTA, J. How to learn, teach and use Prolog by example, Springer-Verlag, 1988.

[COE;LOP;VIC 88]
COELHO,H.;LOPES, G.;VICCARI, R. M., Models for grammar evolution, (working paper), 1988.

[COH 85]
COHEN, J., Describing Prolog by its interpretation and compilation, Communications of the ACM, vol 28, no 12, december 1985 p 1311-1324.

[COL 85]
COLMERAUER, A., Prolog in 10 figures, Communication of ACM, vol. 28, n. 12, december, 1985.

[COS 88]
COSTA, E., Aprendizagem: conceitos, estratégias e problemas, I Escola Portuguesa de Inteligencia Artificial, Mira, 1988.

[COS 87]
COSTA, E., Ensino assistido por computador e inteligência artificial, Revista de Informática da API, vol. 6 no. 3, maio 1987.

[COS 86a]
COSTA, E. O papel do conhecimento em programas de ensino assistido por computador, Depto de Engenharia Electrotécnica, Universidade de Coimbra, 1986a.

[COS 86b]
COSTA, E.; et. al. A resolution based method for discovering students' misconceptions, Actas do EPIA-86, Lisboa, 1986b.

[DAL 84]
DAHL, V., ABRAMSON, H. On gapping grammars, technical report 84-2, Department of Computer Science, The University of British Columbia, 1984.

[DEJ;MOO 86]
DEJONG, G.; MOONEY, R., Explanation Based Learning: an alternative view, Machine Learning, vol. 1, no. 1, 1986.

[DRE 86]
DRESCHER, G., A mechanism for early piagetian learning, AAAI87, Seattle, Washington, 1986, p 290-294.

[DUC;COS;KOD 86]
DUCHENOY, S.;COSTA, E;KODRATOFF,Y., A method for discovering students misconceptions in intelligent tutoring systems, ECAI-86,1986.

[ENN 81a]
ENNALS, J., Logic as a computer language for children, Imperial College, Department of Computing and Control, London, 1981a.

[ENN 81b]
ENNALS, J., Prolog an introduction for teachers, Imperial College Department of computing and control, London, 1981b.

[ENN 81c]
ENNALS, J., Prolog can link diverse subjects with logic and fun, Practical Computing, march 1981c, p 91-92.

[ENN 87]
ENNALS, J., The fifth generation and training strategies, Artificial Intelligence Review, no. 1, 1987, p 131-134.

[ELC 83]
ELCOCK, E., The pragmatics on Prolog: some comments, Logic Programming Workshop, 1983.

[FAR;AND;REI 84]
FARRELL;ANDERSON;REISER, An interactive computer based tutor for LISP, AAAI84, University of Texas at Uastin, USA, 1984, p. 106-109.

[FIN;KAS 86]
FININ, T.;KASS, R., Rules for the implicit acquisition of knowledge about the user, AAAI86, Seattle, Washington, 1986, p. 295-306.

[FOR 87]
FORD, L., Teaching strategies and tactics in intelligent computer aided instruction, Artificial Intelligence Review, vol. 1, No. 3, 1987.

[FOR;RAD 86]
FORSYTH, R.;RADA,R., Machine Learning applications in expert systems and information retrieved, Ellis Horwood, England, 1986.

[GAG 62]
GAGNE, R., The acquisition of knowledge, Psichological Rewiew, vol. 69, 1962, p. 355-365.

[GAR;VER 84a]
GARIJO, F.;VERDEJO, M., Diseño de un sistema experto para la classification de problemas y construccion de programas, Facultad de Informática, Universidad del País Vasco, 1984a.

[GAR;VER 84a]
GARIJO, F. J.;VERDEJO, M., CAPRA: an intelligent system for teaching programming concepts, Facultad de Informática, Universidad del pais Vasco, january 1984b.

[GAR;VER 84c]
GARIJO, F. J.;VERDEJO, M., Knowledge representation for teaching programming in an ICAI environment, CAIA-84, Facultad de Informática, Universidad del pais Vasco, sept 1984c.

[GEN 80]
GENESERETH, M., The role of plans in intelligent teaching systems, Department of computer science, Stanford University, november 1980.

[GEN;NIL 87]
GENESERETH,M.;NILSON,N., Logical Foundations of Artificial Intelligence, Morgan Kaufmann Publishers, Los Altos, 1987.

[GER 81]
GERSHMAN, A., Finding out what the user wants - steps toward an automated yellow pages assistant, 7ht International Conference on Artificial Inteligence, p. 423-425, 1981.

[GOL 82]
GOLDSTEIN, I., The genetic graph: a representation for the evolution of procedural knowledge, in Intelligent Tutoring systems, (eds.) by D. Sleeman and J. S. Brown, Great Britain, Academic Press, 1982, p 51-75.

[GOO 85]
GOODMAN, B., Communication and miscommunication, Techinical Report 5681, Bolt. Beranek and Newman, 1985.

[GOO 86]
GOODMAN, B., Miscommunication and plan recognition, International Workshop on User Modeling, Maria Laach, West Germany, 1986.

[GRE 88]
GREENFIELD, P., An investigation into the applicability of definite clause grammars for use in intelligent tutoring systems, Department of Computer Science, University of Birmingham, 1988.

[GRO;APP;MAR;PER 87]
GROSZ, B.;APPELT, D.;MARTIN, P.; PEREIRA, F., TEAM: An Experiment in the Design of Transportable Natural Language Interfaces, Artificial Intelligence, 1987, 32. p 173-243.

[HAB;PER 83]
HABERMANN, N.; PERRY, D., ADA for experienced Programmers, Addison Wesley, 1983.

[HAR 87]
HARMON, P., Intelligent job aids: how AI will change training in the next five years, Artificial Intelligence & Instruction applications and methods, Addison-Wesley, USA,1987, p 165-190.

[HEW 77]
HEWITT, C., Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, 8, 1977.

[HIN 88]
HINTIKKA, J., Knowledge and belief, Ithaca:Cornell University, in Knowledge, Belief and User Modelling, Self, J., Centre for Research on computer and learning, University of Lancaster, 1988.

[JAC 85]
JACKSON, P., Implementing a game-theoretic interpretation of logic, Department of Artificial Intelligence, University of Edinburgh, 1985.

[JOH;LEW;SOL 85]
JOHNSON, W.; LEWIS; SOLOWAY, E., An automatic debugger for Pascal programs, BYTE, abril 1985, p. 179-190.

[KAU;GRU 86]
KAUFFMANN, H.;GRUMBACH, A., MULTILOG: MULTIple worlds in LOGic programming, Proc. of 7th European Conference on Artificial Intelligence-86 ECAI-86, Brighton, july 1986, p 290-305.

[KAW;MIZ;KAK;TOY 86]
KAWAI ; MIZOGUCHI ; KAKUSHO ; TOYODA, A framework for ICAI systems based on inductive inference and logic programming, Third International Conference of Logic Programming (ed) SHAPIRO, E., London, 1986.

[KEA 88]
KEANE, M., Where is the beef: the absence of pragmatic theories of analogy, eccai88, Munique, p. 327332, 1988.

[KEA 87]
KEARSLEY, G., Artificial Intelligence & Instruction appications and methods, Addison-Wesley, USA, 1987.

[KEA 83]
KEARSLEY, G., Computer-based training : A guide to selection and implementation, Reading, MA:Addison-Wesley, 1983.

[KEA;HUN;SEI 83]
KEARSLEY, G.;HUNTER, B.;SEIDEL,R., Two decades of computer based instruction projects:What have we learned?, T.H.E. Journal, 1983, 10(3), p 90-94; 10(4), p 90-96.

[KIM 82]
KIMBALL, R., A self-improving tutor for symbolic integration, in Intelligent Tutoring systems, (eds.) by D. Sleeman and J. S. Brown, Great Britain, Academic Press, 1982, p 283 305.

[KOB 85]
KOBSA, A., Three steps in constructing mutual belief models from user assertions, 6th European Conference on Artificial Intelligence, p. 423-427, 1985.

[KOB 86]
KOBSA, A.,Using situation description and russellian attitudes for representing beliefs and wants, 9th International Conference on Artificial intelligence, p. 513-515, 1986.

[KOD;TEC 87]
KODRATOFF, Y.; TECUCI, G., DICIPLE: a knowledge-intensive aproach to learning apprentice systems for planning and design, Laboratoie de Recherche en Informatique, Universite de Paris-Sud, 1987.

[KON 84]
KONOLIGE, K., Belief and incompleteness, SRI, 1984.

[KOW 79a]
KOWALSKI, R., Algorithm = Logic + Control, C.A.C.M. Vol 22, No.7, 1979a.

[KOW 79b]
KOWALSKI, R., Logic for problem sulving, North Holland, New York, 1979b.

[KOW 82]
KOWALSKI, R., Logic as a Computer Language for Children, ECAI-82, 1982.

[KUL;KUL;COH 80]
KULIK, J.;KULIK, C.;COHEN, P., Effectiveness of computer-based college teaching: a meta-analysis of findings, Review of Educational Research, 50, 1980, p. 525-544.

[LAN;SIM;BRA;ZYT 87]
LANGLEY, P.;SIMON, H.;BRADSHAW, G.,ZYTKOV, P., Scientifc discovery:computation an explorations of the creative processes, MIT Press, 1987.

[LAN;ZYT;SIM;BRA 86]
LANGLEY, P.;ZYTKOV, J.;SIMON, H.;BRADSHAW, G., The search for regularity: four aspects of scientifc discovery, in Machine Learning: an artificial intelligence approach, vol. ii, Morgam Kaufman, 1986.

[LAN 82]
LENAT, D., The nature of heuristics, Artificial Inteligence, vol. 19,2, p.189-249, 1982.

[LAI;ROS;NEW 85]
LAIRD, J.; ROSENBLOOM, P.; NEWELL, A., **Chunking in soar: the anatomy of a genaral learning mechanism**, Carnegie-Mellon University, 1985.

[LAN;BRE;FAR 83]
LANTZ, B.;BREGAR, W.;FARLEY, A., An intelligent CAI system for teaching equation solving, **Journal of Computer-Based Instruction**, no. 10, pp 35-42, 1983.

[LEE;COE;COT 85]
LEE,R.;COELHO,H.;COTTA,J., Temporal Inferencing on Administrative Databases, **Information Systems**, Vol. 10, no. 2, 1985.

[LON;CLA 82]
LONDON, B.;CLANCEY, W., **Plan recognition strategies in student modeling:prediction and description**, Depto of Computer Sicence, Stanford University, may 1982.

[LOO;ROS 87]
LOOI, C.;ROSS, P., **Automatic program debugging for a Prolog intelligent tutoring system**, Department of Artificial Intelligence, University of Edimburg, 1987.

[LOW 86]
LOW, R., **Retrocesso Inteligente em Prolog**, Universidade Federal do Rio Grande do Sul, Brasil, set 1986.

[LOP 86]
LOPES, G., **Conceptualização de um interlocutor automático**, Instituto Superior Técnico, Universidade Técnica de Lisboa, 1986. (Tese de Doutoramento)

[LOP;VIC 84]
LOPES, G.;VICCARI, R., An Intelligent monitor interaction in portuguese language, **ECAI-84**, Pisa, 1984.

[LOP;VIC 88]
LOPES, G.;VICCARI, R., Natural Language Syntax Learning and Guided Sepelling Correction: the case of an intelligent tutor for Prolog, (working paper) 1988.

[MAT 82]
MATZ, M., Towards a process model for high school algebra errors, in **Intelligent Tutoring systems**, (eds.) by D. Sleeman and J. S. Brown, Great Britain, Academic Press, 1982, p 25-49.

[MAZ;ROK;SZU 86]
MAZUD, M.;ROKOTOZAFY, R.; SZUMACHOWSKI-DESPLAND,A., Méta-génération de code par réécriture de termes guidée par modèles dárbres, **INRIA**, décembre 1986.

[MEL 82]
MELLISH, C. An alternative to structure sharing in the implementation of Prolog interpreter, in: CLARK, K & TARNLUND, S.,Logic Programming, New York, Academic Press, p. 99-106, 1982.

[MIC 86]
MICHALSKY, R., Learning strategies and automated knowledge acquisition: an overview, in Computational models of learning (Bloc ed.), Springer-Verlag, p.1-19, 1986.

[MIC;CAR;MIT 86]
MICHALSKY, R.; CARBONEL, J.; MITCHELL, T., Machine Learning: an artificial intelligence approach, vol.2, Morgan Kaufman, 1986.

[MIL 82]
MILLER M., A structured planning and debugging environment for elementary programming, in Intelligent Tutoring systems, (eds.) by D. Sleeman and J. S. Brown, Great Britain, Academic Press, 1982, p. 119-133.

[MIT;KEL;KEA 86]
MITCHELL, T.;KELLER, R.;KEDARCABELLI, S., Explanation based generalization: a unifying view, Machine Learning, vol. 1, no. 1, p. 4780, 1986.

[MON;POR 88]
MONTEIRO, L., PORTO, A., Modules for logic programming based on context extension, UNL, 1988.

[MOO 81]
MOORE,R.C., Automatic Deduction for Commonsense Reasoning:an Overview, SRI, 1981.

[MOR;ROL 85]
MORIK, K.;ROLLINGER, C., The real-estate agent-modeling user by uncertain reasoning, AI Magazine, 6, p. 44-52, 1985.

[MUL 84]
MULLAN, A., Children and computers in the classroom, London, Castle House Publications, 1984.

[MUR 85]
MURRAY, William R., Heuristics and formal methods in automatic program debugging, IJCAI-85, Los Angeles, 1985.

[NAI 86]
NAISH, L., Negation and control in Prolog, Lectures Notes in Computer Science, Springer_Verlag, 1986.

[NAW 87]
NAWROCKI, L. H., Artificial intelligence applications to maintenance trainingm, Artificial Intelligence & Instruction applications and methods, Addison-Wesley, USA, 1987, p 135-163.

[NEW;SIM 72]
NEWELL, A. ; SIMON, H., Human problem solving, Englewood cliffs, Prentice Hall, 1972.

[NEW;SIM 63]
NEWELL, A.;SIMON, H., GPS:A program that simulates human trhout, in FEIGENBAUM, E.;FELDMAN, D., (eds), Computers and thouht, New York, MacGraw-hill, 1963.

[ORL;STR 81]
ORL, J. ; STRING, J., Computer-based instruction for military training, Defense Management Journal, 18 (2), 1981, p. 46-54.

[O'SH 82]
O'SHEA,T., A self-improving quadratic tutor, Intelligent Tutoring systems, (eds.) by D. Sleeman and J.S. Brown, Great Britain, Academic Press, 1982, p 309-334.

[OHL 87]
OHLSSON, S., Transfer of training in procedural learning a matter of conjectures and refutations?, in BOLC, L., (eds), Computational models of learning, 1987, p 55-88.

[PAI;BUN 85]
PAIN, H.;BUNDY, A., What stories should we tell novice Prolog programmers?, Department of Artificial Inteligence University of Edinburgh, Edinburgh, 1985.

[PAR;PER;SEI 87]
PARK, O.; PEREZ, R.; SEIDEL, R., Intelligent CAI: Old Wine in New Bottles, or a New Vintage?, Artificial Intelligence & Instruction applications and methods, Addison-Wesley, USA, 1987, p 11-45.

[PAP 80]
PAPERT, S., Mind-storms children, computers, and powerful ideas, Basic Books, New York, 1980.

[PAP 72a]
PAPERT, S., Teaching children thinking, Programmed Learning and Educational technology, vol. 9, n. 2, 1972, p 117-175.

[PAP 72b]
PAPERT, S., Teaching children to be mathematicians versus teaching about mathematics, International Journal of Mathematics, Education, Science, and Technology, vol. 3, 1972b, p 249-262.

[PER;POR 79]
PEREIRA, L.M.;PORTO, A.,Intelligent backtracking and sidetracking in horn clause programs - the theory, Universidade Nova de Lisboa, october 1979.

[PER;POR 81]
PEREIRA, L. M.;PORTO, A., Selective backtracking, Universidade Nova de Lisboa, july 1981.

[PER 82]
PEREIRA, L. M., Logic control with logic, Universidade Nova de Lisboa, february 1982.

[PER 84]
PEREIRA, L.M., Rational debugging of logic programs, Universidade Nova de Lisboa, february 1984.

[PER;BRU 81]
PEREIRA, L.M.;BRUYNOOGHE,M., Revision of top-down logical reasonning through intelligent backtracking, Centro de Informática da Universidade Nova de Lisboa, march 1981.

[PER;CAL 88]
PEREIRA, L.M.;CALEJO, M., A framework for Prolog debugging, Joint Logic Programming Conference, Symposium of Logic Programming, 1988.

[PER;WAR 80]
PEREIRA, F.;WAREN, Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmentd transition networks, Artificial Intelligence, 13, 1980, p.231-278.

[POR 84]
PORTO, A., Controle sequencial de programas em lógica, UNL, 1984.

[POL 57]
POLYA, G., How to solve it, Anchor, New York, 1957.

[PRO 85]
PROJECTO MINERVA, Informatique/intelligence artificielle, Colloque franco-portugais, Coimbra, december 1985.

[PRE 88]
PRESSLER, J., An Intelligent natural deduction proof tutor, Computerised Logic Teaching Bulletin, vol. 1, no. 1, march 1988.

[RAP 79]
RAPOSO, E.P., Introdução à gramática generativa sintaxe do português, Moraes, Lisboa, 1979.

[RES]
RESNICK, L., Holding an instructional conversation:comments on chapter 10 by Collins, University of Pittsburgh.

[REI 78]
REISER, B.; et al., Programming viewed as an engineering activity, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, january 1978.

[REI 85]
REISER, B.;et al., Dynamic student modelling in an intelligent tutor for LISP programming, IJCAI-85 Los Angeles, 1985.

[RES 73a]
RESNICK, L., Hierarchies in children's learning: A symposium, Instructional Science, vol. 2, 1973a, p. 311-362.

[RES 73b]
RESNICK, L.; et al., Task analysis in curriculum design: A hierarchically sequenced introductory mathematics curriculum, Journal of Aplied Behavior Analysis, vol 6, 1973b, p 679-710.

[RIC 79]
RICH, E., User modeling via stereotypes, cognitive Science, 3, p. 329-354, 1979.

[ROS;LEW]
ROSS, P.; LEWIS, J., Plan recognition for intelligent tutoring systems, Departament of Artificial Intelligence, University of Edinburh, 1988.

[ROS;LOP 88]
ROSADO, P.;LOPES, G., Interfaces de língua natural com capacidade para aprenderem novos vocábulos, seu significado e para se adaptarem a novos utilizadores: uma experiência, UNL, 1988 (comunicação pessoal).

[SAI 87]
SAINT-DIZIER, P., Contextual discontinuous grammars theoretical aspects and implementation, INRIA, février 1987.

[SCH;MAL;SHA 86]
SCHERZ, Z.;MALER, O.;SHAPIRO, E., The use of logic programming in education, EURIT 86, 1986, p 531-537.

[SER;COE;GAS 87]
SERNADAS,C.;COELHO,H.;GASPAR,G., Communicating Knowledge Systems: Big Talk Among Small Systems, Applied Artificial Intelligence, Vol.1, nos.3 and 4, 1987.

[SEL 74]
SELF, J., Student models in computer-aided instruction, International Journal of Man-Machine Studies, 6,1974, p 261-276.

[SEL 88]
SELF,J., Knowledge, belief and user modelling,AI III:methodology, systems and applications, O'SHEA, T.;SGUREV, V., (eds), North-Holland, 1988.

[SEL 87]
SELF, J., The application of machine learning to student modelling, Artificial Intelligence and education, LAWLER;YAZDANI (ed.), vol 1, 1987, p. 267-280.

[SHN 80]
SHNEIDERMAN, B., Software psuchology, winlhrop publishers, Cambridge, 1980.

[SLE 82]
SLEEMAN, D., Assessing aspects of competence in basic algebra, in Intelligent Tutoring Systems, Academic Press, London, 1982, p 185-197.

[SLE;HEN 82]
SLEEMAN, D.;HENDLEY, R., A structured planing and environment for elementary programming, Intelligent Tutoring Systems, (eds.) by D. Sleeman and J. S. Brown, Academic Press, London, 1982, p 157-182.

[SLE;BRO 82]
SLEEMAN, D.;BROWN, J., Intelligent Tutoring Systems (eds.), Academic Press, London, 1982.

[SLE 86]
SLEEMAN, D., Some principles of intelligente tutoring, Instructional Science, 14, 1986, p293-326.

[SHA 82]
SHAPIRO, E., Algorithmic program Debugging, The MIT Press, London, 1982.

[SIL 88]
SILVA, P., Estudo de um sistem adaptável as características linguísticas dos seus utilizadores, Faculdade de Ciências de Lisboa, Lisboa, 1988 (Estágio de conclusão de curso).

[SIM 81]
SIMON, J., Education et Informatisation de la société:repport au President de la République, synthése de P. LEMOINE, Bulletin de Liaison, INRIA, n. 72, 1981.

[SKI 68]
SKINNER, B., **The tecnology of teaching**,Appleton-Century Crofts, New York, 1968.

[STA 87]
STABLER, E., Restricting logic Grammars with government-binding theory. **Computational Linguistics**, no. 13 (1-2) p 1-10, 1987.

[STE;SHA 87]
STERLING, L.;SHAPIRO, E., **The Art of Prolog**, London, MIT, 1987.

[STE;COL 77]
STEVENS, A.;COLLINS, A.,The goal structure of a socractic tutor, Proceeding of 1977 annual conference, **Associaction for computing machinery**, Seattle, octuber, p. 256-263, 1977.

[STE;COL;GOL 82]
STEVENS, A.;COLLINS,A.;GOLDIN,E., Misconceptions in students' understanding, in **Intelligent Tutoring systems** (eds.) by D. Sleeman and J. S. Brown, Academic Press, Great Britain, p. 13-23, 1982.

[SUC 87]
SUCHMAN, L., **Plans and situated actions: the problem of human-machine communication**, Cambridge University Press, Cambridge, England, 1987.

[THO 87]
THOMPSON, P., Mathematical microworlds and intelligent computer-assisted instruction, in **Artificial Inteligence & Instruction Applications and Methods**, by Kearsley Greg P.,Addison-Wesley, 1987, p 83-107.

[TEI 81]
TEITELBAUM, T.;REPS, T., The cornell program synthesiser: a syntax directed programming environment, **Communications of the ACM**, 24, 9, 563-573, 1981.

[TUR 88]
TURK, C., Evolutionary methodology in natural language acquisition by machine learning, **Cybernetics and systens'88**, (eds.) Robert Trappl, Kluwer Academic, Viena, 1988.

[VIC 84a]
VICCARI, R. M., Interpretação de Língua Natural Escrita: uma experiência, **SUCESU-84**, Rio de Janeiro, 1984a.

[VIC 85b]
VICCARI, R. M., Desenvolvimento de um sistema inteligente para o Ensino Assistido por Computador, **I Seminário Brasileiro de Software e Hardware**, UFRJ, Rio de Janeiro, 1984b.

[VIC 85]
VICCARI, R. M., Sistema para instrução assistida por computador em Inteligência Artificial, UFRGS, Porto Alegre, 1985. (Tese de Mestrado)

[VIC 85a]
VICCARI, R. M., Comunicação em linguagem natural com base de dados SQL, II Simpósio Brasileiro de Inteligência Artificial, São José dos Campos, S. Paulo, 1985a.

[VIC 85b]
VICCARI, R. M., Controle de diálogos na interacção em linguagem natural, II Simpósio Brasileiro de Inteligência Artificial, São José dos Campos, S. Paulo, 1985b.

[VIC 87a]
VICCARI, R.M., Execução de cláusulas Prolog, Actas do 3o EPIA, Braga, Portugal, 1987a e Informática e Universidade, no. 2, 1987.

[VIC 87b]
VICCARI, R.M., Um tutor para o ensino da programação em lógica: reflexões oportunas sobre a. sua aplicação aos edifícios inteligentes, CIIC-87, LNEC, Lisboa, 1987b.

[VIC 88]
VICCARI, R. M., Interação coloquial com correção automática de erros, IBERAMIA 88, 1988.

[VIC;COST;COE 87]
VICCARI,R.M.;COSTA,E.;COELHO,H., A Prolog Tutor for Logic Programming, PEG - Prolog Education Group - Proceedings of the Annual Conference, Exeter, England, 1987.

[VIC;COE;COS 88]
VICCARI, R.M.;COELHO, H.;COSTA, E., Behaving as a tutor by mental "image" guidance", Proceedings of the Machine Learning Workshop, Sesimbra, Portugal, 1988.

[VIC;COE;COS 88]
VICCARI, R. M;COELHO, H.;COSTA, E., Pragmatic Attachment Devices for Conversations With Tutors, Applied Artificial Intelligence An International Journal, vol. 2, 1988 (a ser publicado).

[VOS 84]
VOSE, G., Macintosh pascal, Byte, 9, 6, 1984.

[ZAM;VIC 88]
ZAMBUJO, C. VICCARI, R., The question today is not whether computers should be used in school but to decide how to introduce them, ICME6, Budapeste, 1988.

[ZAM 89]
ZAMBUJO, C. Matemática como um valor humano e científico na época dos computadores, Dissertação de Mestrado, Faculdade de Ciências, 1989, (a ser publicada).

[ZAM 87a]
ZAMBUJO, C., A Prolog database on tagus estuary problems implications on the mathematics and logic concepts, PME-XI, Montreal, 1987a.

[ZAM 87b]
ZAMBUJO, C., Erreur, nereste pas calhée, CIEAEM 39, Sherbrook, 1987b.

[ZAM 87c]
ZAMBUJO, C., O ensino da matemática na época dos computadores, revista O professor, no. 100, Lisboa, 1987c.

[ZAM 86]
ZAMBUJO, C., Mathematices and active life, CIEAEM38, Southampton, 1986.

[WAL 87]
WALLACH, B., Development strategies for ICAI on small computers, Artificial Intelligence & Instruction applications and methods, Addison-Wesley, USA, 1987, p 305-322.

[WEN 87]
WENGER, E., Artificial inteligence and tutoring systems, Morgan Kaufmann, Los Altos, 1987.

[WIN 82]
WINSTON, P., Learning new principles form precedents and exercices, Artificial Intelligence, vol. 19,3, 1982.

[WIN 80]
WINSTON, P., Learning and reassoning by analogy, ACM, vol 23, no. 12, December 1980, p. 609703.

[YOK 83]
YOKOTA, M; et alii, The design and implementation of a personal sequential inference machine: PSI, Tokyo, Institute for New Generation Computer Technology, 1983.

ANEXO 1

EXPLORAÇÃO DO TUTOR-PROLOG

A - Apresentação do ambiente de ensino do Tutor-Prolog

Com a finalidade de observar ambientes para o ensino de uma linguagem de programação e para a programação em si, foram incluídos no ambiente tutorial algumas facilidades comuns aos programas editores e interpretadores: inserir, remover, listar, resumir, ajudar e retroceder. Além destes comandos existem outros três criados especialmente para esta aplicação: dicionário, esquecer e espiar.

(a) Inserir

O aluno pode introduzir construções válidas na linguagem Prolog durante a exploração do ambiente tutorial e do ambiente de depuração, como é apresentado na figura 1.1. Pode também escrever textos (pequenas definições) a respeito do assunto com o qual se encontra trabalhando. Estas definições ficam armazenadas na uma base de dados do aluno.

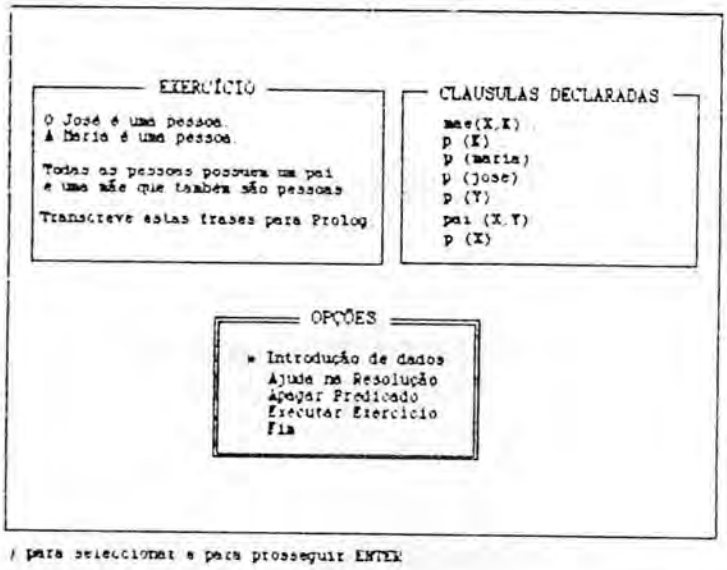


Figura 1.1 Introdução de dados

(b) Remover

O aluno pode ter necessidade de remover factos armazenados inadequadamente ou com o objectivo de fazer novas simulações que não envolvam parte das informações existentes, como é apresentado na figura 1.2.

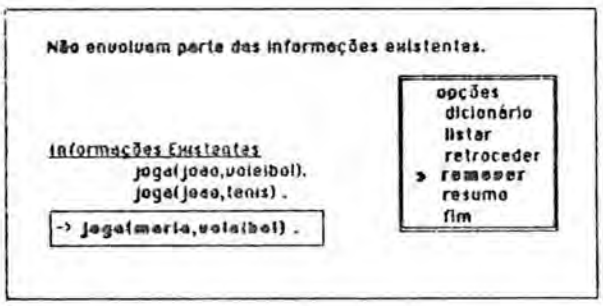


Figura 1.2 Remoção de dados

(c) Listar

Para tornar o ambiente de ensino mais flexível, encontram-se disponíveis opções para listar o conteúdo da base de dados e para listar os programas escritos pelo aluno, como é apresentado na

figura 1.3.

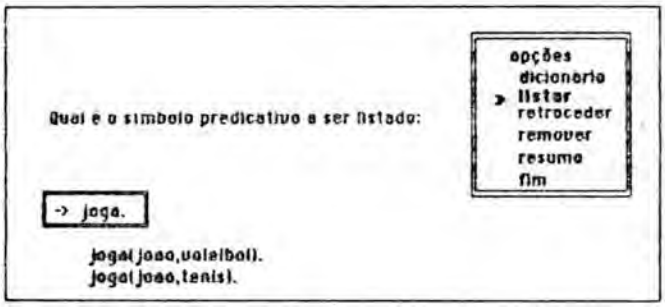


Figura 1.3 Listagem dos dados

(d) Resumir

Destina-se à obtenção do resumo do conteúdo apresentado em cada módulo instrucional Prolog. Por exemplo, o resumo do conteúdo instrucional do módulo dos objectivos, como é apresentado na figura 1.4.

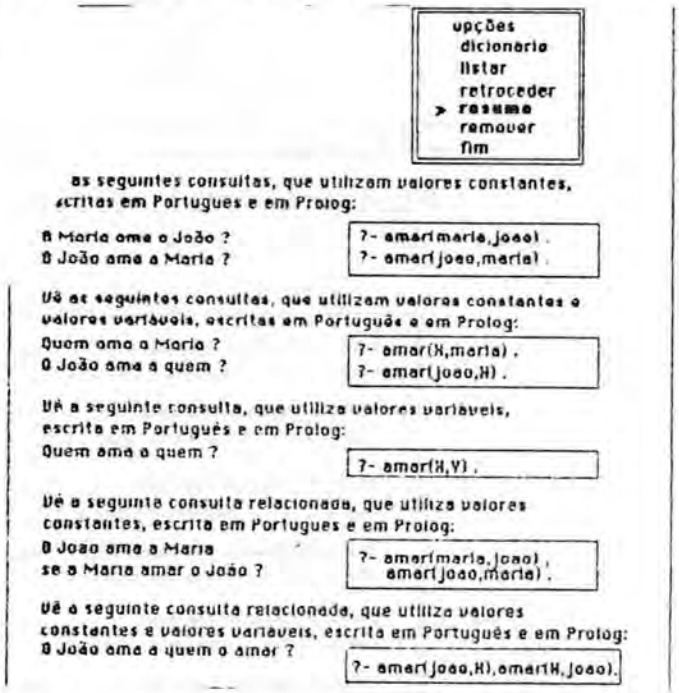


Figura 1.4 Resumo do conteúdo de um bloco instrucional

(e) Ajudar

Durante a escrita dos exercícios propostos pelo Tutor-Prolog

e durante a interacção em língua natural, o aluno pode utilizar este comando para obter informações, no primeiro caso, sobre quais as cláusulas que ainda deverão ser escritas e, no segundo, sobre os conhecimemtos linguísticos que o Tutor-Prolog possui, como é apresentado na figura 1.5 (1) e 1.6 (2).

Figura 1.4 Solicitação de ajuda (1)

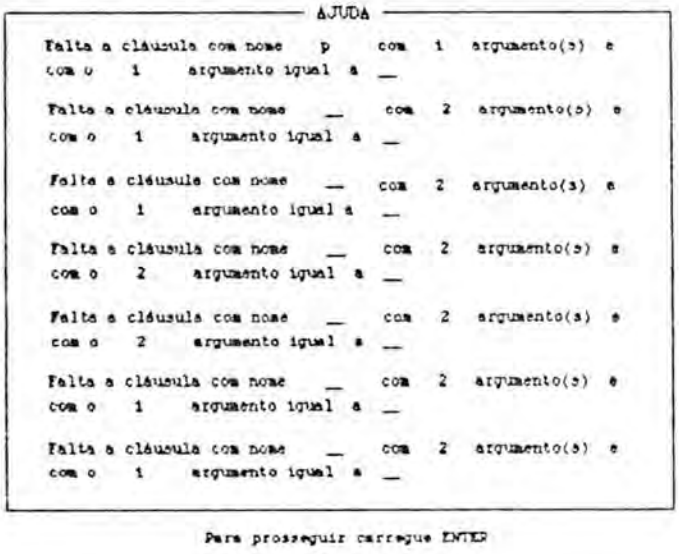


Figura 1.5 Solicitação de ajuda (2)

(b) Retroceder

E uma opção para voltar para trás a lições já passadas, o que possibilita ao aluno o acesso a conteúdos já apresentados quando achar necessário, como é apresentado na figura 1.6.

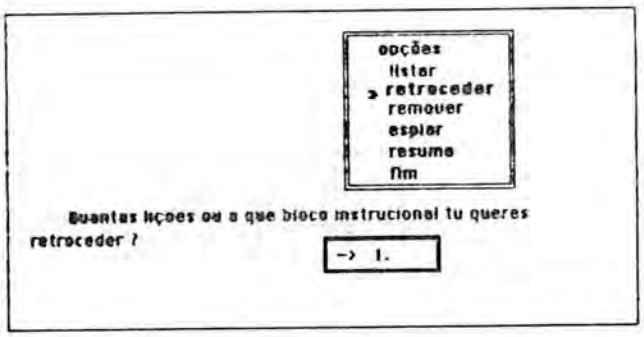


Figura 1.6 Retroceder lições

(d) Dicionário

O aluno pode ter acesso às informações relativas aos símbolos predicativos e ao número de argumentos associados com cada símbolo utilizado na escrita de seus programas, como é apresentado na figura 1.7.

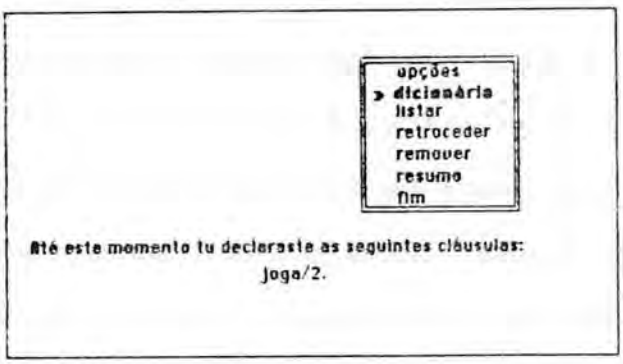


Figura 1.7 Apresentação do dicionário de programas

(h) Esquecer

O comando esquecer possui três formas de aplicação: esquecer todo o conhecimento armazenado no enquadramento e todos os factos e cláusulas geradas; esquecer somente determinadas cláusulas; e esquecer um determinado símbolo predicativo, como é apresentado na figura 1.8.

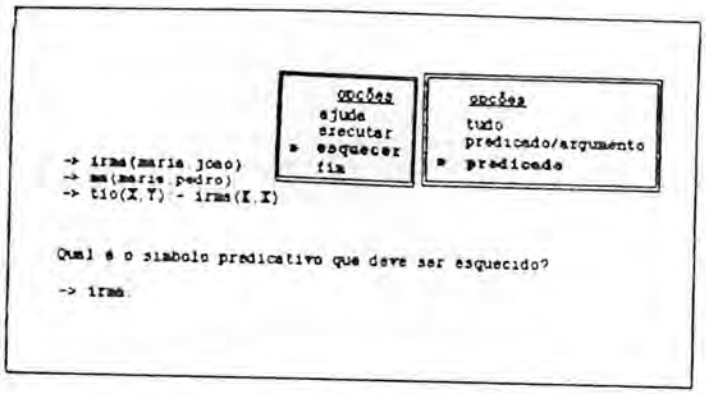


Figura 1.8 A opção esquecer

(f) Espiar

Durante a simulação da execução dos programas o aluno pode ver o modo como o meta-interpretador interpreta os seus comandos (acesso ao "trace") ou aceder às estruturas internas que definem o conhecimento existente no meta-interpretador. Logo, o meta-interpretador não é uma "caixa preta", pois o aluno tem acesso às estruturas internas do conhecimento, como é apresentado na figura 1.9.

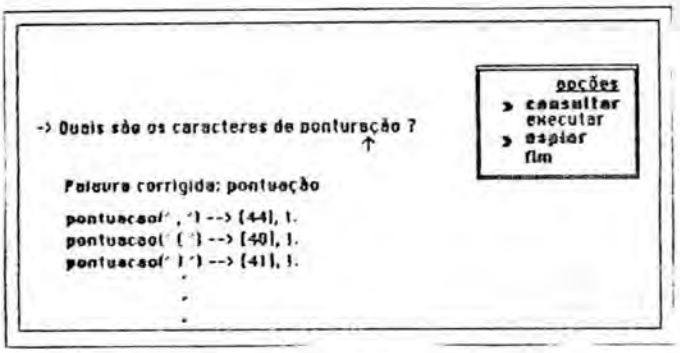


Figura 1.9 A opção espiar

Todas estas facilidades podem ser accionadas pelo aluno através do menu principal ou dos menus específicos de cada ambiente de trabalho. O desenvolvimento destes comandos foi feito paralelamente a investigação prática.

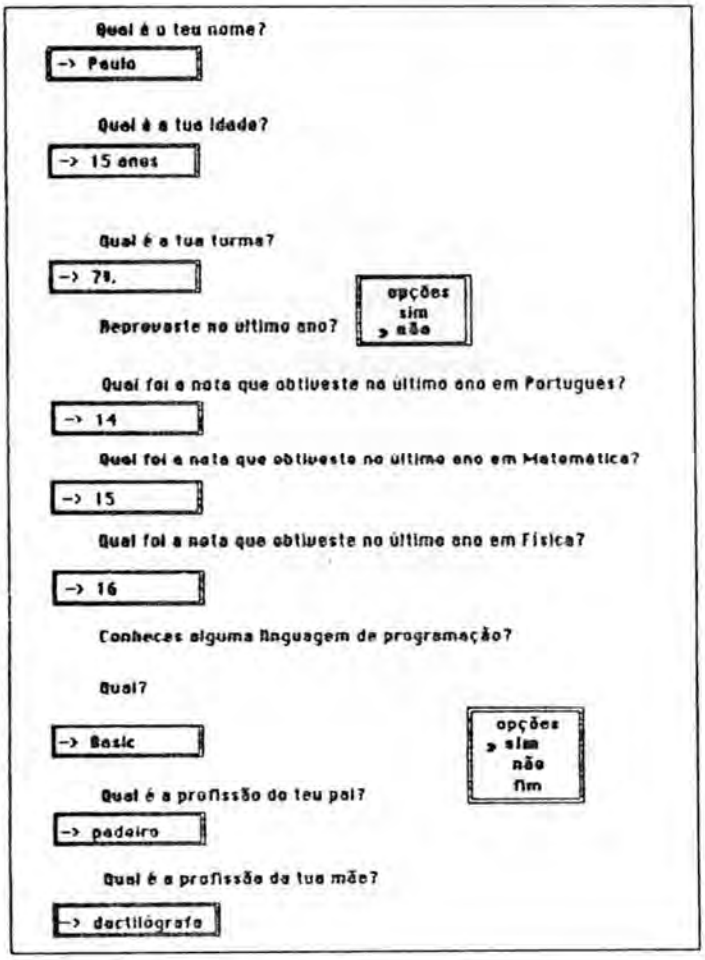
B - Exemplo de uma utilização do Tutor-Prolog

Nesta interacção simulamos um aluno com aproveitamento escolar médio (nossas investigações práticas foram feitas com esta classe de alunos) e que já conhece uma linguagem de programação. Estes dados são levados em conta pelo Tutor-Prolog ao planear a primeira interacção, o que o leva a iniciar o conteúdo instrucional pelo bloco dos factos em vez de optar pelo bloco dos átomos, que é o primeiro bloco da taxonomia padrão do Tutor-Prolog [2].

---

2 A apresentação dos exemplos fica um pouco prejudicada devido à impossibilidade da representação das marcações e da sobreposição de janelas feitas dinamicamente na tela do terminal. As opções do aluno são assinaladas com o sinal >>. As opções que se encontram apenas em negrito representam as opiniões do Tutor-Prolog. Na interacção tutorial as opiniões do Tutor-Prolog são marcadas com a cor verde.

A parte inicial da interacção destina-se a obtenção dos dados pessoais do aluno.



Estas informações estáticas influenciam directamente no carregamento dos módulos tutoriais e na determinação do plano instrucional a ser utilizado pelo Tutor-Prolog.

- Se a idade do aluno for inferior a 12 anos, o módulo tutorial deverá avançar até ao módulo dos objectivos. Não deverá prosseguir a apresentação com o módulo das listas e das regras. A observação empírica e a discussão do assunto com professores de Matemática demonstraram que antes dos 12 anos, ou seja até ao 6o ano, o aluno ainda não adquiriu os conceitos lógicos básicos exigidos nestes itens;

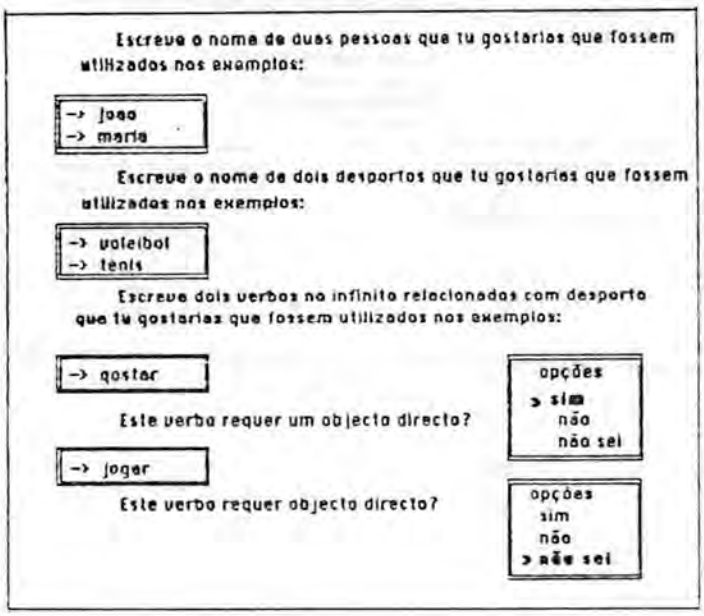
- Se o aluno responde favoravelmente quanto ao conhecimento de outra linguagem de programação, o Tutor-Prolog passa a pressupor que este domina os conceitos de átomos;

- De acordo com a nota que o aluno possuir em Português e com o contexto onde ocorrem os erros, o Tutor-Prolog optará por reforçar e variar as mensagens relativas aos erros de natureza sintáctica ou não. O Tutor-Prolog possui uma hipótese de que o aluno que apresenta dificuldades na visualização da sintaxe da sua língua natural, apresentará os mesmos problemas em relação a outra linguagem qualquer;

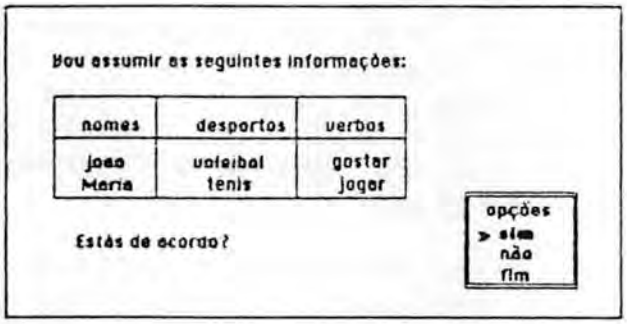
- As informações referentes às notas obtidas em Matemática e Física servem para analisar, a posteriori, a sua relação com o desempenho do aluno no aprendizado da linguagem de programação Prolog. Ou seja, não é assumida nenhuma hipótese a priori, e

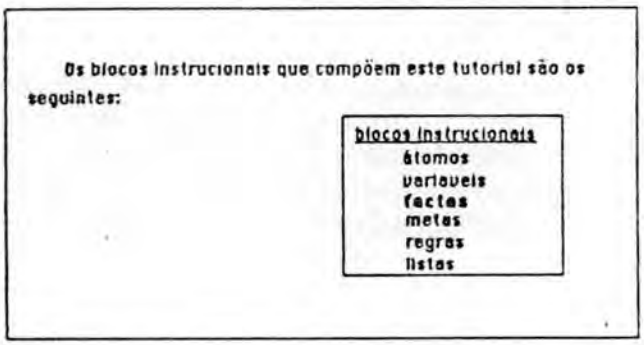
- As informações referentes à profissão do pai e da mãe do aluno relacionam-se com a hipótese de que os filhos de pais graduados encontram no ambiente familiar maior incentivo e maiores condições materiais para o estudo. Estas hipóteses também serão analisadas a posteriori.

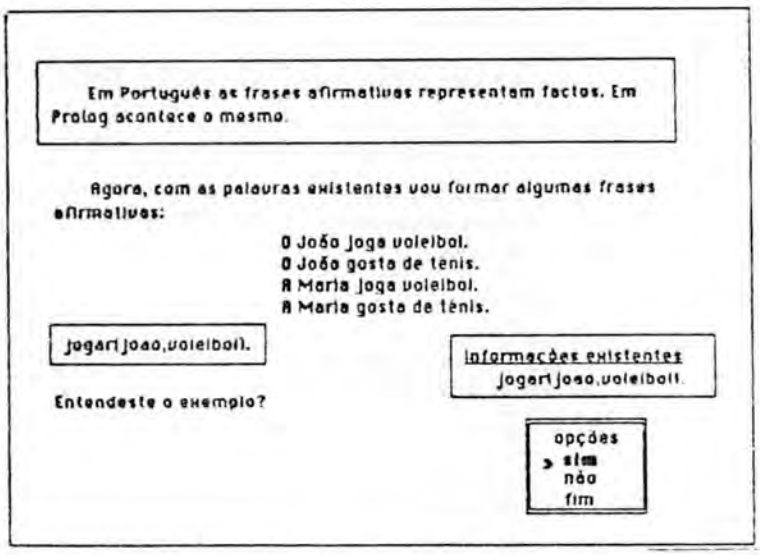
Concluída a identificação do aluno, tem início um novo diálogo com a finalidade de obter elementos que viabilizem a geração automática de exemplos (frases em Português e em Prolog). Começando então, o processo de ensino da linguagem Prolog.

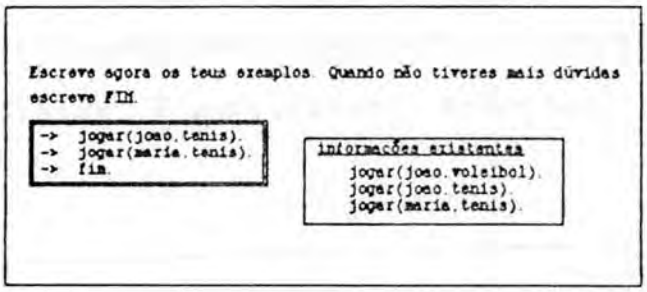


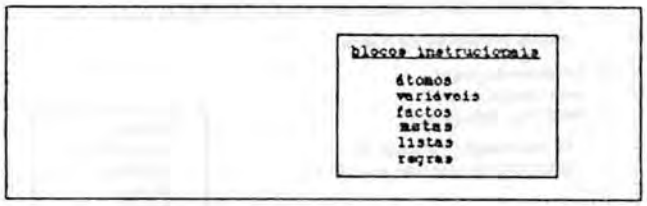
Concluída a fase de identificação do aluno e de obtenção do contexto para a geração dos exemplos, tem início o processo de ensino.

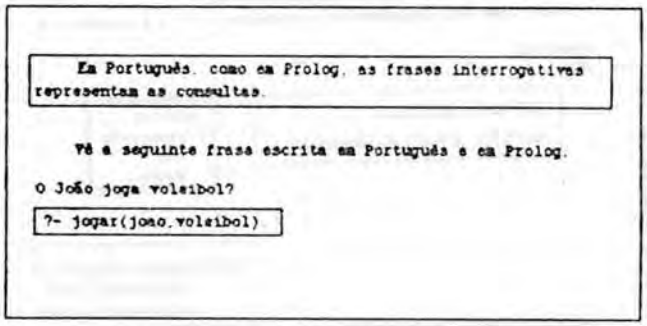


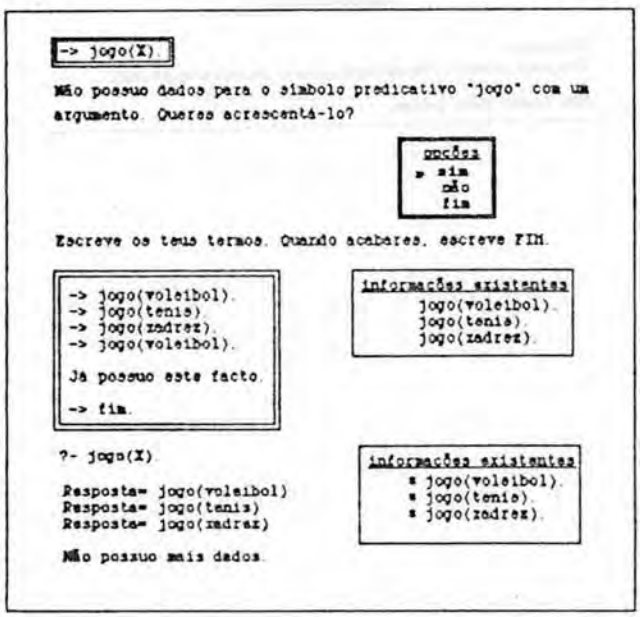




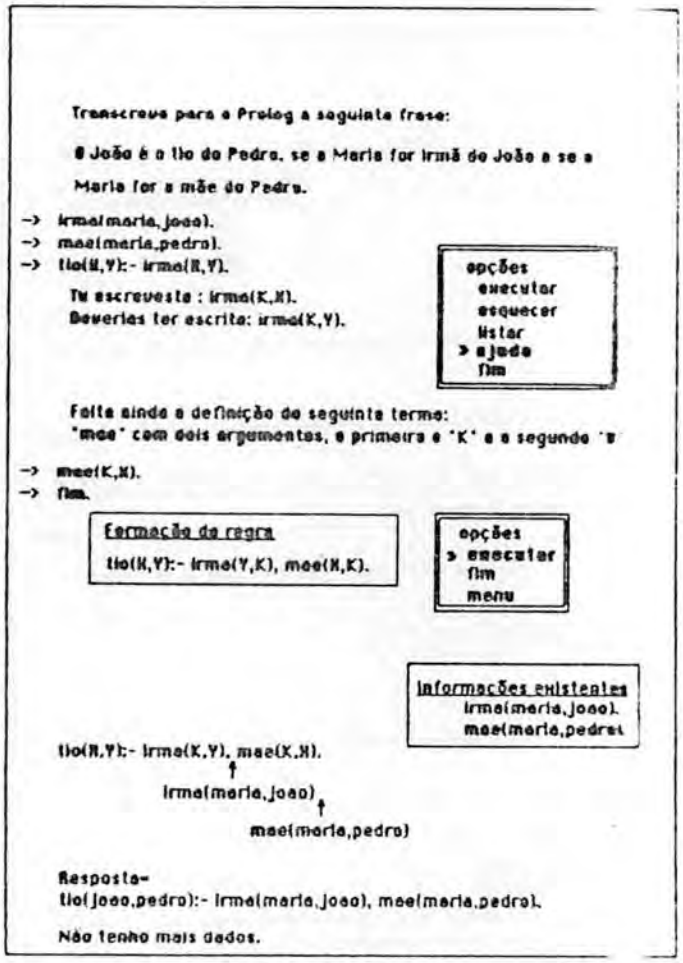

Escreve agora os teus exemplos. Quando não tiveres mais dúvidas escreve FIM.
-> jogar(joao,tenis).
-> jogar(maria,tenis).
-> fim.
informações existentes
jogar(joao,voleibol).
jogar(joao,tenis).
jogar(maria,tenis).


blocos instrucionais
átomos
variáveis
factos
metas
listas
regras


Em Português, como em Prolog, as frases interrogativas representam as consultas.
Vê a seguinte frase escrita em Português e em Prolog.
O João joga voleibol?
?- jogar(joao,voleibol).


-> jogo(X).
Não possuo dados para o simbolo predicativo "jogo" com um argumento. Queres acrescentá-lo?
opções
» sim
não
fim
Escreve os teus termos. Quando acabares, escreve FIM.
-> jogo(voleibol).
-> jogo(tenis).
-> jogo(xadrez).
-> jogo(voleibol).
Já possuo este facto.
-> fim.
informações existentes
jogo(voleibol).
jogo(tenis).
jogo(xadrez).
?- jogo(X).
Resposta= jogo(voleibol)
Resposta= jogo(tenis)
Resposta= jogo(xadrez)
Não possuo mais dados.
informações existentes
» jogo(voleibol).
» jogo(tenis).
» jogo(xadrez).

A última etapa do tutorial é destinada aos exercícios para a fixação dos conteúdos instrucionais apresentados.



C - Avaliação da investigação prática

Este anexo está refere-se apenas à observação do uso do Tutor-Prolog nas fases de seu desenvolvimento. Assim, são relatadas as observações que dizem respeito à construção de um tutor inteligente.

As observações relativas ao uso do Tutor-Prolog como instrumento pedagógico encontram-se relatadas com pormenor em Zambujo [ZAM 89], onde podem ser encontradas informações sobre a metodologia utilizada durante a investigação (1986 a 1988), a característica do contexto de estudo (escola, professores, alunos, equipamentos e temática), a organização pedagógica (duração, fases, significância e avaliação do estudo), e uma descrição das actividades realizadas no âmbito do estudo da matemática. Ainda, no referido trabalho, são feitas comparações entre o rendimento dos alunos ao longo das três observações realizadas, ou seja, a realização das mesmas actividades no ambiente do Tutor-Prolog, no ambiente do programa Folha de Cálculo e no ambiente típico da sala de aula.

As actividades realizadas no ano lectivo de 1986 - 1987 relacionaram-se com o uso do Tutor-Prolog:

- material instrucional e sondagem das necessidades para um ambiente de programação orientada (depuração, simulação, diagnóstico, orientação);

- aplicações matemáticas no âmbito das actividades pedagógicas que focaram o estudo do estuário do rio Tejo.

No ano lectivo de 1987 - 1988 foram realizadas actividades interdisciplinares visando:

- o desenvolvimento e o teste de aspectos relacionados com a aquisição de conhecimentos e a aprendizagem por parte do Tutor-Prolog;
- o ensino de procedimentos recursivos e estruturas de dados mais potentes no que se refere ao aluno.

Dentro deste ambiente foram observados aspectos do ensino e da aprendizagem da Linguagem de Programação Prolog para alunos do 2o. grau. As aplicações envolveram a construção e a exploração de Bases de Dados (factos, estruturas de dados, consultas, regras).

O material instrucional que compõe o Tutor-Prolog está dirigido para a criação e a exploração de Bases de Dados. Este material é composto pelo conteúdo intrucional e pelas facilidades voltadas para a orientação na escrita de programas (depuração).

Os oito estudantes seleccionadas para a primeira investigação, dois meninos e duas meninas, em cada grupo de observação, pertenciam as 7o e 9o anos do ensino secundário. Cada grupo pode ser considerado de nível médio, quer em aproveitamento escolar geral, quer nos resultados obtidos no pré-teste aplicado com vistas à selecção. Considerou-se ainda, o nível social semelhante entre os dois grupos.

O material utilizado na investigação constou de 1

microcomputador Olivetti M24 com memória de 256 Kb. Esta configuração de memória prejudicou bastante o desenvolvimento das actividades, pois todo o trabalho teve que ser dividido em módulos separados e desvinculados do ponto de vista operacional. Com isso, o aluno perdeu um pouco a visão do todo interligado e foi obrigado, muitas vezes, a repetir a representação dos dados pois a manipulação de arquivos que, nestes casos, não podia ficar sob o controle do Tutor-Prolog, causava certa confusão e gerava, muitas vezes, perdas de informações. No que se refere ao "software" foram utilizados, além da linguagem de programação Arity Prolog, o tutor, editores de texto e pacotes gráficos. O Sistema Operativo utilizado foi o MS/DOS.

(a) Actividades

Numa primeira fase, com duração aproximada de três sessões de duas horas cada, os alunos realizaram a parte tutorial relativa ao material instrucional [VIC 86]. O material instrucional apresentado era composto por simulações, exercícios e diagnósticos com vistas à introdução da sintaxe, da semântica e da pragmática da linguagem de programação Prolog orientada para a geração e exploração de pequenas Bases de Dados. O tutorial desenvolve-se com a apresentação de problemas escritos em Português e a sua transcrição para o Prolog. Os problemas transcritos eram simulados e acompanhados pelo Tutor-Prolog. Os aspectos desenvolvidos foram a geração de factos, de objectivos e de regras. As estruturas de dados observadas nesta fase, foram as

cadeias as listas, as regras tipo "Se Então" e o cálculo de predicados.

Nesta primeira fase os alunos de ambas as turmas não apresentaram dificuldades quanto a compreensão geral do material instrucional e à representação lógica dos problemas propostos. As observações mais significativas foram em relação ao desconhecimento de alguns sintagmas (termos mais técnicos como: interpretador, lógica, Prolog, ...). Para resolver este problema foi desenvolvido o módulo de consulta a sintagmas referentes ao conteúdo instrucional. Este módulo é acedido em Lingua Natural (LN) e pôde ser testado ainda no ano lectivo 86 - 87. Sua utilização foi bem recebida pelos alunos que propuseram que sua actuação fosse alargada para a definição de termos relativos a aplicação por eles desenvolvida (ver 2a fase). Esta solicitação foi desenvolvida e observada conjuntamente com os professores das disciplinas de Português, que juntaram-se à investigação no ano lectivo 87 - 88.

Ainda, relativamente ao conteúdo instrucional, foi possivel observar que os alunos do 9o ano realizaram uma exploração mais efectiva dos dados armazenados. Os alunos do 7o. ano restringiam-se mais aos problemas propostos pelo Tutor-Prolog e não efectuavam perguntas de carácter exploratório.

Numa segunda fase, com duração aproximada de 12 sessões de 1h30 min. cada, os alunos desenvolveram três actividades do ponto de vista do uso do Tutor-Prolog e três actividades do ponto de

vista pedagógico. Estas últimas relacionadas com o estuário do Tejo.

A primeira actividade envolveu a criação e a consulta directa de uma Base de Dados sobre diversos aspectos do estuário do Tejo (fauna, flora, solo, ...).

Visando a passagem do aluno do módulo tutorial para o módulo de auxílio na escrita de programas foi-lhe apresentada a seguinte lista de dados:

a) Sapais (S/VCC+DV)

crustáceos	12	espécies	(48,0%)
moluscos	5	"	(20,0%)
oligoquetas	4	"	(16,0%)
poliquetas	1	"	(4,0%)
vários	3	"	(12,0%)

b) Vasas Negras Compactas (VNC)

crustáceos	5	espécies	(33,3%)
poliquetas	4	"	(26,7%)
oligoquetas	3	"	(20,0%)
moluscos	2	"	(13,3%)
vários	1	"	(6,7%)

c) Ostreiras (O)

poliquetas	14	espécies	(31,8%)
crustáceos	11	"	(25,0%)
moluscos	8	"	(18,2%)
espongiários	3	"	(6,8%)
oligoquetas	2	"	(4,6%)
hidrários	2	"	(4,6%)
peixes	3	"	(6,8%)
vários	1	"	(2,2%)

d) Areias Vasosas (AV)

poliquetas	16	espécies	(32,6%)
crustáceos	14	"	(23,3%)
moluscos	5	"	(11,6%)
oligoquetas	3	"	(7,0%)
hidrários	2	"	(4,7%)
antozoários	1	"	(2,3%)
peixes	1	"	(2,3%)
vários	1	"	(2,3%)

e) Vasas Arenosas (VA)

crustáceos	10	espécies	(37,1%)
poliquetas	6	"	(22,2%)
moluscos	6	"	(22,2%)
oligoquetas	2	"	(7,4%)
antozoários	1	"	(3,7%)
peixes	1	"	(3,7%)
vários	1	"	(3,7%)

e) Areias (A)

crustáceos	15	espécies	(75,0%)
poliquetas	3	"	(15,0%)
antozoários	1	"	(5,0%)
oligoquetas	1	"	(5,0%)

Os conteúdos instrucionais envolvidos nesta etapa são o aprendizado da sintaxe e da semântica da linguagem de programação Prolog, no que se refere a átomos, variáveis, factos e listas, a operações de entrada e saída de dados (geração e consulta de BD) e a estruturas de daddos como as cadeias e as listas.

Base de Dados e consultas geradas pelas amostras durante 3 sessões.

1a.

```
info(o(moluscos,8,18.2)).      info(o(hidrarios,2,4.6)).
info(o(crustaceos,11,25)).     info(o(peixes,3,6.8)).
info(o(espongiarios,3,6.8)).   info(o(varios,1,2.2)).
info(o(poliquetas,14,31.8)).   info(o(poliquetas,7,3.8)).
info(o(oligoquetas,2,4.6)).
```

Consultas efectuadas

```
?-o(X,Y,Z).        ?-o(hidrarios,A,B).
?-o(X,Y,4.6).
```

2a.

```
info(vnc(crustaceos,5,33.3)).   info(av(oligoquetas,3,7.0)).
info(o(poliquetas,14,31.8)).    info(av(hidrarios,2,4.7)).
info(s(moluscos,5,20)).         info(av(antozoarios,1,2.3)).
info(va(crustaceos,10,37.1)).   info(s(poliquetas,1,4)).
info(a(crustaceos,15,75.0)).    info(s(varios,3,12)).
info(ss(sapais,25)).            info(vnc(varios,1,6.7)).
 ..........................................
```

Consultas efectuadas

```
?-tejo(tipos_de_solo,S).    ?-s(X,Y,Z).
?-ss(sapais,S).
```

3a.

```
info(va(varios,1,3.7)).          info(vnc(poliquetas,4,26.7)).
info(a(oligoquetas,1,5.0)).      info(av(poliquetas,16,32.6)).
info(s(oligoquetas,4,16)).       info(ss(sapais,25)).
info(vnc(varios,1,6.7)).         info(av(moluscos,5,11.6)).
 ..................................................
```

Consultas efectuadas

```
?-sapais(crustaceos,X).      ?-s(crustaceos,X,Y).
?-va(peixes,X,3.7).          ?-o(X,Y,Z).
?-av(peixes,1,2.3).          ?-tejo(moluscos,A).
?-tejo(X,Y).
```

1a interacção

```
info(vnc(oligoquetas,3,20)).        info(s(crustaceos,12,48)).
info(s(varios,3,12)).               info(s(especies,25,100)).
info(s(oligoquetas,4,16)).
info(poliquetas(anelideos,segmentacao_bem_visiveis,laterais)).
info(poliquetas(lobos,dorsal,vertical)).
info(s(moluscos,5,20)).             info(vnc(poliquetas,4,26,7)).
info(s(poliquetas,1,4)).
```

Consultas realizadas

```
?-s(X,Y,Z).                          ?-s(crustaceos,X,A).
?-s(oligoquetas,X,Y).                ?-poliquetas(P).
?-poliquetas(X).                     ?-poliquetas(X,Y,Z).
?-poliquetas(X,Y,Z).                 ?-s(oligoquetas,4,X).
?-s(X,Y,Z).                          ?-s(especies,25,100).
?-s(especies,A,B).                   ?-vnc(moluscos,3,X).
?-vnc(moluscos,3,X,Z).               ?-s(moluscos,5,Y).
```

2a. interacção

```
info(o(moluscos,8,18.2)).           info(o(hidrarios,2,4.6)).
info(o(crustaceos,11,25)).          info(o(peixes,3,6.8)).
info(o(espongiarios,3,6.8)).        info(o(varios,1,2.2)).
info(o(poliquetas,14,31.8)).        info(o(poliquetas,7,3.8)).
info(va(moluscos,6,22.2)).          info(vnc(poliquetas,4,26.7)).
info(va(oligoquetas,2,7.4)).        info(o(crustaceos,11,25)).
```

Consultas realizadas

```
?-o(poliquetas,X,Y).                 ?-svncoavvaa(oligoquetas,P,Q).
?-s(oligoquetas,A,B).                ?-vnc(oligoquetas,X,Y).
?-o(oligoquetas,X,Y).                ?-av(oligoquetas,A,B).
?-va(oligoquetas,C,D).               ?-a(oligoquetas,K,D).
?-svncavvaa(oligoquetas,X,Y).        ?-svncoavvaa(oligoquetas,X,Y).
```

3a. Interacção

```
info(crustaceos(animais_aquaticos,respiracao_branquial,apendices)).
info(crustaceos(quintina,corpo_metamerizado)).
info(hidrarios(animais_aquaticos,forma_de_polipo,
```

```
    cavidade_gastrovascular)).
info(tejo(percentagem,vnc,8.6)).
info(antozoarios(forma_de_polipo,cavidade_gastrovascular,dividida
_por_septos)).
info(tejo(percentagem,ostreiras,25.3)).
info(espongiarios(animais_pluricelulares,vida_fixa,formas,variadas)).
info(tejo(percentagem,av,24.7)).
info(tejo(percentagem,areiras,10.3)).
info(tejo(percentagem,va,15.5)).
tejo(percentagem,_1535,_1539).
```

As Bases de Dados foram criadas no ambiente tutorial que realizava a correcção automática ou coloquial dos erros de natureza sintáctica-semântica [VIC 87], a geração dos ficheiros de dados, o salvamento das sessões de trabalho, a reposição de ambientes anteriores e a simulação comentada das consultas realizadas.

Nesta fase foram utilizados os editores de texto para a organização das questões levantadas pelos alunos, visando a exploração da Base de Dados. As questões eram de natureza qualitativa e quantitativa. As primeiras foram resolvidas com o auxílio dos professores da disciplina de Biologia e com o uso de bibliografia técnica. As segundas foram transcritas para o Prolog e dirigiam-se à Base de Dados. Nesta primeira actividade foram realizadas apenas consultas directas e consultas relacionadas.

Questões propostas pelos alunos visando a exploração da BD:

- Qual a percentagem de moluscos existente no estuário do Tejo?
- Quantas espécies de crustáceos existem no estuário do Tejo?
- Qual a percentagem de oligoquetas existentes no estuário do Tejo?

- Quantas espécies de peixes existem no estuário do Tejo?
- Que espécies de Poliquetas predominam no estuário do Tejo?
- Nas áreas vasosas existe uma espécie de peixe. Qual é a espécie?
- Também nas vasas arenosas existem peixes. Qual é a espécie?
- Quais são as espécies de crustáceos que predominam em todas as unidades de população?

Questões propostas pelos alunos visando a anexação e a consulta de definições:

- O que são sapais?
- O que são areias vasosas?
- O que são vasas arenosas?
- O que são oligoquetas?
- O que são hidrários?
- O que são ostreiras?
- O que são vasas negras compactas?
- O que são areias?
- O que são poliquetas?
- O que são antozoarios?

A ideia de utilizar o módulo de L.N. para a definição de sintagmas relativos a aplicação do aluno surgiu nesta fase. Os alunos desejavam colocar no Tutor-Prolog (no ambiente relativo ao módulo do aluno) as definições dos termos da Biologia (sapais, oligoquetas, areias, ...).

A segunda actividade envolveu a geração de novos dados a partir dos armazenados na BD durante a primeira. Nesta etapa foi possível observar aspectos relacionados com o desenvolvimento de um ambiente de programação inteligente. O objectivo das obsernvações e posterior desenvolvimento de facilidades para a depuração e orientação durnate a escrita de programas, foi a exploração de pequenas BD criadas pelo utilizador.

As regras geradas pelos alunos relacionaram-se com o cálculo da média, da mediana, do índice de constância, do índice de fidelidade, da frequência, de percentagens, pesquisas em tabelas (noções de intervalos), etc.

No desenvolvimento e no uso de regras, os alunos representaram com grande facilidade as consultas em tabelas, os cálculos directos (busca dos dados na BD e realização de cálculos) mas apresentaram certa dificuldade na geração de problemas que envolviam procedimentos intermediários ou procedimentos recursivos. Note-se que a recursão não foi abordada directamente pelo Tutor-Prolog na fase da apresentação do material intrucional. Também ficou claro que os alunos aprenderam com facilidade a representação da informação em forma de listas, mas apresentaram dificuldades na geração e compreensão de problemas que envolvessem a sua manipulação.

Os alunos que foram introduzidos directamente na escrita de programas tiveram maiores dificuldades, que os demais, nas actividades que envolviam o uso de estruturas de dados e a escrita de regas para a manipulação destas estruturas. Nos outros aspectos do uso da linguagem de programação Prolog o aproveitamento foi, apenas, ligeiramente inferior.

No decorrer desta segunda fase das observações ficou patente, também, a necessidade de desenvolver e apresentar aos alunos estruturas mais potentes de representação das informações, como as redes semânticas, as árvores e os quadros.

Não houve diferenças significativas entre os exemplos desnvolvidos pelo 7o ou pelo 9o anos. Verificaram-se diferenças quanto a organização dos intervalos dos programas de classificações. Os alunos do 9o ano geraram intervalos mais precisos. Também, quanto ao programa para o cálculo da mádia os alunos do 7o ano geraram uma estrutura não recursiva e chamavam a cláusula media(X,Y,Z) a cada nova solicitação.

Exemplos de Prolemas desenvolvidos:

Exemplo (1) calculo da percentagem

```
percentagem(X,N,Y):- info(X,X1),
                     Y is (X1*100)/N.

%dados
info(lagares,1936980).
```

Exemplo (2) maior e menor valor da percentagem
% areia

```
maior_valor(areia,Estacao,E,Y,X):- retract(info(areia,Estacao1,Areia,_
                                   (Areia > Y,
                                    maior_valor(areia,Estacao1,E,Areia,X)
                                    ;
                                    maior_valor(areia,Estacao,E,Y,X).

maior_valor(areia,Estacao,Estacao,Y,Y).

%dados
info(areia,c01,2,96,2).
info(areia,c02,6,89,5).
   ..............
```

Nota: Foram desenvolvidos programs semelhantes para a obtenssão do menor valor. A pesquisa envolveu outros tipos de solos além das areias.

Exemplo (3) indice de constância

```
constancia_esp(A,X):- retract(info(A,Ncol,Nest)),
                      X1 is Nest*10,
                      X2 is Ncol/X1,
                      X is X2 * 100 .
contancia_esp(A,X).

% dados
info(vnc,10,5).
info(vac,15,6).
   ........
```

Exemplo (4) classificacao dos biótipos

```
classificacao(areia,N,X):- percentagem(areia,N,Z),
                           compara(Z,X).

compara(Z,X):- Z >= 0,  Z < 16, X = vnc.
compara(Z,X):- Z >= 15, Z < 51, X = va.
compara(Z,X):- Z >= 50, Z < 96, X = av.
compara(Z,X):- Z >= 95, X = a.
```

Exemplo (5) cálculo da média

```
media(X,Y,Z):- soma(X,K),
               Y is K/Z.

soma(X,Y):- info(o(crustaceos,X,D)),
            acumula_valores(X,Y).
soma(X,Y).
acumula_valores(X,Y):- retract(soma_aux(D)),
                       Y1 is Y+D,
                       assert(soma_aux(Y1)),
                       !, fail.3

% dados
```

Base de Dados criada na actividade 1

---

3 O "!" e o "fail" que são elementos de controle foram por nós introduzidos devido a estrutura que o programa adquiriu durante o seu desenvolvimento.

Exemplo (6) Indice de Fidelidade F de uma especie A num povoamento P

```
ind_fid(A,F,Lista,C1):-
                        constancia_esp(A1,C2),
                        (A1==A,C1=C2;true),
                        append(Lista,[C2],Lista1),
                        ind_fid(A,F,Lista1,C1).

ind_fid(A,F,Lista,C1):-constancia_tot(A,Lista,C1,F).

constancia_tot(A,Lista,C1,F):-soma(Lista,T1,T1),
                              F is (C1/T1)*100.

soma([],C,T1):- T1=C.
soma([H|T],C,T1):- (var(C),T2 is H ; T2 is H+C),
                   soma(T,T2,T1).

append([],R,R).
append([H|T],R,[H|L]):-append(T,R,L).
```

Nesta segunda actividade ficou constatada também a necessidade de conhecimentos pragmáticos no Tutor-Prolog. Os alunos não entenderam a componente de controle do Prolog e tiveram dificuldades usá-la Logo, todos os programas desenvolvidos pelos alunos não utilizaram estes mecanismos. Daí o terceiro protótipo posuir um módulo pragmático que realiza a organização automática das cláusulas dentro de um programa. Este módulo, como os demais, tem uma componente de ensino que, aquando do diagnóstico de uma falha realiza a sua correcção ou, orienta o aluno para que realize a correcção apresentando sempre os motivos pelos quais esta correcção é necessária [VIC 88].

Nesta etapa realizamos, também, tentativas de confrontar o aluno directamente com o interpretador Prolog sem a intermediação do Tutor-Prolog. Os alunos demonstraram preferir o desenvolvimen-

to no Tutor-Prolog. Não se adaptaram às mensagens do interpretador, à manipulação de arquivos de programas (via editor de textos), à interrupção da execução sempre que ocorriam falhas, à necessidade de edição e reinterpretação dos programas, salvamentos, etc.

A terceira actividade relacionou-se com a ligação do Tutor-Prolog com "software" gráfico. Os alunos utilizaram as BD geradas no ambiente tutorial para gerar gráficos relacionados com as actividades matemáticas.

Os alunos apresentaram interesse especial pelos resultados gráficos demonstrando ser importante que um ambiente de programação proporcione também estas facilidades aos seus utilizadores.

D - Observações Gerais

A estratégia usada para a apresentação da linguagem de programação Prolog, que consistiu em comparar a linguagem natural escrita com a linguagem lógica, apresentou resultados satisfactórios.

As mensagens direccionadas especificamente a cada caso, (alerta a falhas, estímulos,...), as alternativas múltiplas na condução do ensino e a correcção automática de erros sintácticos, foram recebidos com surpresa pelos alunos que já haviam

trabalhado com outas linguagens de programação ou com editores de textos. Os menus orientados com grafismos, assinalando, por exemplo, os avanços realizados no conteúdo em estudo ou monitorando a execução das regras, desempenharam uma função de estímulo e de curiosidade no aluno que, assim, pôde acompanhar de forma mais concreta o seu desempenho e planear as suas sessões de trabalho.

No geral, o software até então desenvolvido e/ou observado, mostrou-se claro, objectivo e de fácil utilização, mas pouco flexível no que se refere ao controle que o aluno deve realizar para aceder aos vários módulos.

Quanto ao recurso à criação e consulta de pequenas Bases de Dados para, a partir deles, introduzir os conceitos envolvidos pela linguagem Prolog, foi bem aceito por ambos os grupos. Ressalte-se, porém, que os alunos da 9a. série demonstraram, logo a partida, grande interesse em modificá-los e em criar novas versões de acordo com seus objectivos.

Os alunos da 9a. série (15 anos) resolveram todos os problemas com grande desenvoltura excepto o manuseio de listas. Deste último caso concluiu-se pela necessidade de explicações mais detalhadas quanto a sua utilização e manuseio.

Ficou demonstrado também que a utilização do Tutor-Prolog em grupos (2 alunos em cada máquina) não apresenta um bom resultado, pois cria um clima de concorrência entre os grupos (2 máqui-

nas), gerando atitudes como o salto de blocos instrucionais e a diminuta exploração do sistema. A preocupação passa a ser o grupo ao lado: não ser ultrapassado e, de preferência, acabar o conteúdo antes dos outros colegas.

Já nas sessões realizadas individualmente, o aluno apresentou-se mais calmo, mais concentrado e realizou uma exploração mais criativa e pormenorizada dos caminhos oferecidos pelo Tutor-Prolog.

Foi realizada também uma observação experimental com alunos de escolaridade e idade inferior àquelas que o Tutor-Prolog se destina. Com esta observação geramos hipóteses que são manipuladas pelo controle do Tutor-Prolog (imagem do aluno) [VIC 87]. A actuação dos alunos com idade inferior a 12 anos de idade comprovou a necessidade de planificar as lições levando-se em conta a idade e a série escolar. Estes alunos apresentaram bom desempenho no manuseio de factos, constantes, variáveis e operações de consulta simples, mas apresentaram dificuldades no manuseio de consultas que relacionavam condições para a sua resolução satisfactória. Nenhum aluno observado neste grupo entendeu o conceito e a utilização de listas. Esta observação serviu apenas para confirmar as hipóteses sobre a planificação do material instrucional e sobre a adopção de níveis de complexidade para o ensino da linguagem de acordo com a idade e com o grau de escolaridade.

E - Avaliação Lógica

Esta avaliação foi realizada pela Prof. Cristina Zambujo, da escola no. 1 dos Olivais, e foi por nós resumida neste relatório.

(a) Descrição do modo de avaliação

A avaliação foi essencialmente feita através da observação directa. Nosso interesse é levantar questões sobre novos ambientes proporcionados pelo uso do computador. Contudo, utilizou-se um teste de lógica, um teste sobre conceitos matemáticos e um teste sobre interpretação de fenómenos da vida real. A realização destes testes visava a eventual contribuição de linhas gerais para a investigação que seria realizada. Assim, trinta alunos voluntários (7o, 8o e 9o anos de escolaridade) foram submetidos a um pré-teste avaliando conceitos referentes à lógica de predicados, o qual também contribuiu para a escolha dos oito alunos envolvidos na experiência (alunos que obtiveram resultados médios relativamente ao total de alunos que fizeram o teste).

Esse mesmo teste de lógica foi realizado pelos alunos envolvidos na experiência, no final desta fase de estudos. O objectivo foi o de detectar se haviam ocorrido evoluções nestes conceitos.

Outros dois testes foram realizados no término desta fase da

experiência com a finalidade de observar a existência ou não de diferenças de comportamento entre os oito alunos que participaram da experiência e outros alunos que serviram de controle. Oito alunos do grupo de controle eram oriundos da mesma turma dos do grupo experimental. Cada grupo possuia características semelhantes, relativamente ao meio social e ao aproveitamento escolar, sobretudo nas disciplinas de Matemática e Português. Logo, foi feita uma correspondência biunívoca características dos alunos do grupo experimental e os alunos do grupo de controle.

Os testes continham problemas envolvendo conceitos de estatística (médias de tendência central), confecção e interpretação de gráficos de barras e diagramas de sector, percentagens, proporções, variáveis e interpretação de fenómenos da vida real.

Os resultados e as observações relativas aos testes sobre as noções de matemáticas estão descritos em [ZAN 87].

(b) Teste de lógica

O objectivo principal deste teste foi o de detectar noções de lógica de 1a. ordem possuidas pelos alunos antes e depois da utilização da linguagem Prolog.

Os conteúdos abordados foram designações, designados, proposições, termos compostos, símbolos predicativos, símbolos funcionais, quantificadores e traduções de linguagens. O teste foi composto por 11 questões com diversas alíneas e cuja duração

foi de 1h.

Aspectos gerais abordados no teste:

- Tradução para a linguagem matemática e vice-versa.
- Expressões designatórias, proposicionais, expressões designatórias equivalentes, etc.
- Indicações de número designados por expressões numéricas.
- Concretizações de condições em determinados universos.
- Tradução para linguagem corrente de proposições identificadas por p e q verdadeiras e r falsas (p V ~ r, p=>q, ~p A ~q, etc) e respectivo valor lógico.
- Negação de proposições escritas em linguagem corrente.
- Tradução para a linguagem simbólica da lógica, utilizando os símbolos V, A, ~ , =>, sendo dadas em linguagem corrente.
- Tradução da linguagem simbólica (utilizada em Prolog) para a linguagem comum e a interpretação de, por exemplo, gosta(maria,laranjas), gosta(joao,X), etc.
- Traduções semelhantes às anteriores só que da linguagem corrente para a lógica de 1a ordem das cláusulas de Horn.
- Identificação de esquemas simples e confecção de árvores.
- Identificação de valores lógicos de frases existentes em esquemas utilizando símbolos da lógica.

As questões eram independentes umas das outras e continham um número diferentes de alíneas, sendo-lhes atribuído 1 ponto para cada alínea certa e 0 ponto para cada alínea errada. Logo, as questões possuiam pontuações diferentes.

Para o nosso objectivo o resultado que interessa são as diferenças mais marcantes entre os dois grupos (inicial e final)

relativamente a cada variável (variável dependente).

Elaboraram-se, então, dois gráficos para cada ano de escolaridade, onde se pode observar as diferenças entre os resultados iniciais e finais em termos de média de pontuação para cada questão.

Observando o gráfico do 7o. ano (gráfico (1)), verificou-se, no teste final, uma melhoria geral com maior significância nas questões 5, 6, 7 e sobretudo na 9.

Nas questões 3 e 4 (indicação de números designados por expressões numéricas e concretização de condições em universos finitos) não foi possível a atribuição de pontuação.

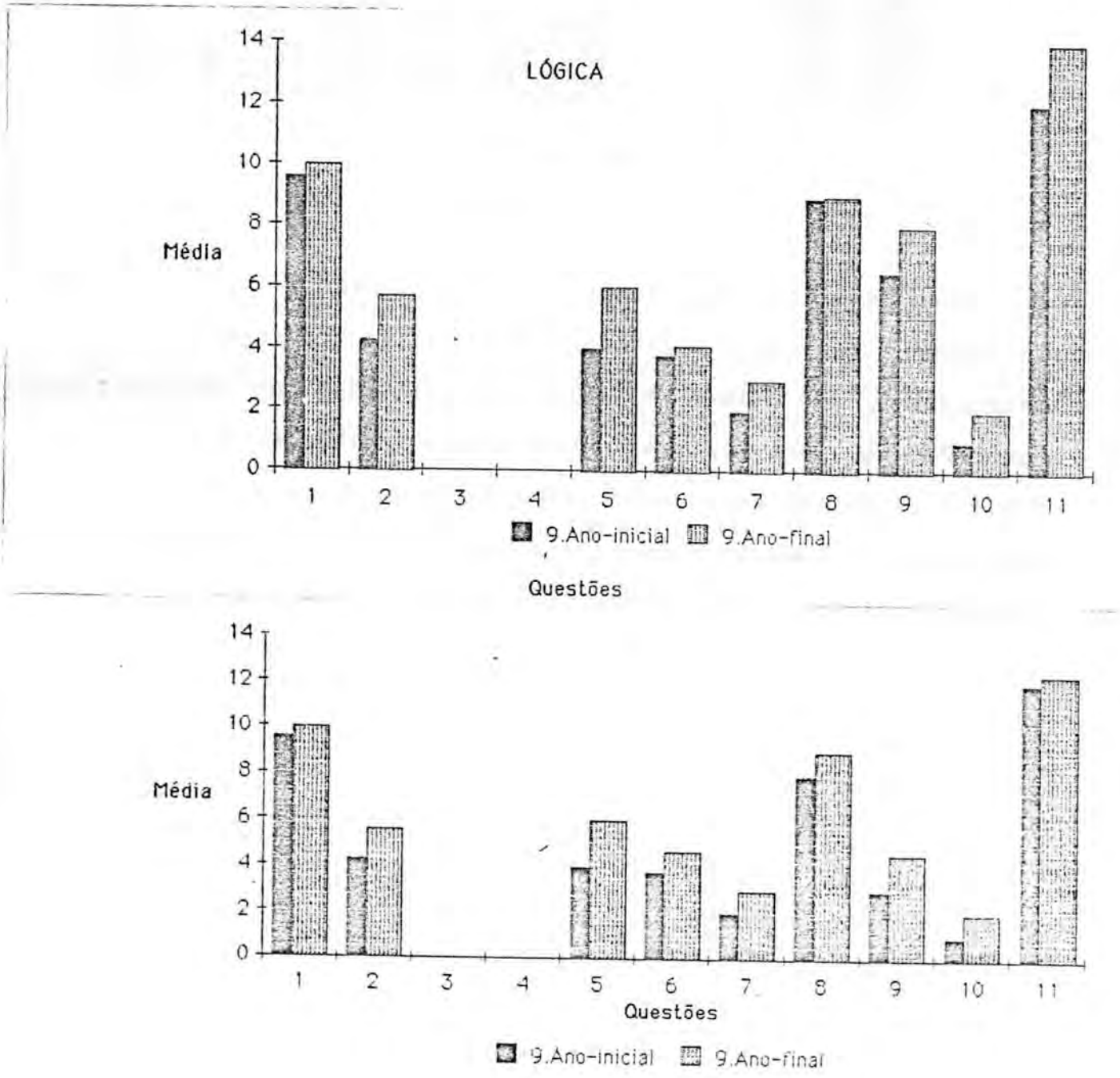
No gráfico (2) do 9o. ano verifica-se também uma melhoria geral no teste final, com mais significância nas questões 5, 7, 9 e 11. Quanto as questões 3 e 4, ocorreu o mesmo que no 7o. ano.

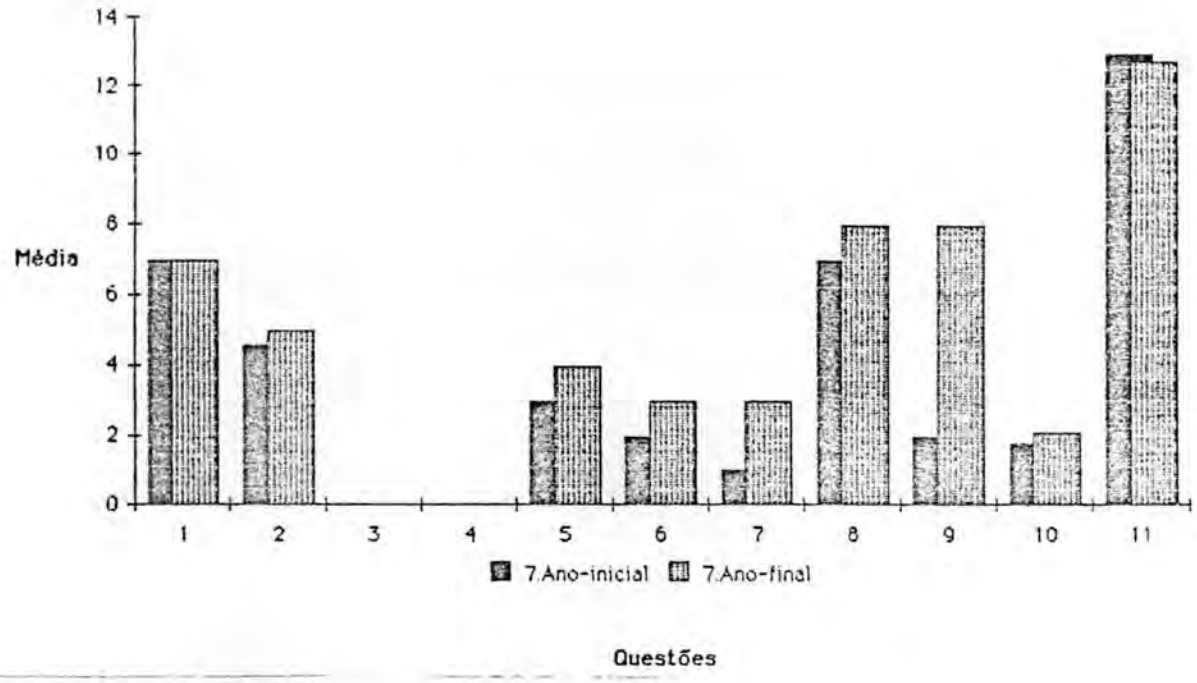
Observando-se ainda, os resultados do teste inicial do 7o. e do 9o. ano, constata-se, de uma maneira geral, uma média superior de respostas certas no 9o ano, especialmente na questão 9.

Quanto aos resultados do teste final do 7o e 9o anos, observa-se um nível ligeiramente superior no 9o ano. Logo nas observações iniciais (conteúdo instrucional) ficou claro que os alunos do 9o. ano apresentavam maior desenvoltura na exploração do Tutor-Prolog.

Para a observação do desempenho obtido sem a utilização do

módulo tutorial, foi selecionado um terceiro grupo de quatro alunos, dois meninos e duas meninas pertencentes ao 9o. ano. O citério de selecção destes alunos foi o mesmo utilizado para os grupos anteriores e o resultado da observação está descrito no gráfico (3)





Para a última observação foi seleccionado um terceiro grupo de alunos, com características idênticas as dos anteriores, que desenvolveu as mesmas actividades realizadas no ambiente do Tutor-Prolog, mas utilizando o programa "Folha de Cálculo" [ZAN 89]. O objectivo desta observação foi o de comparar o desempenho dos alunos no desenvolvimento das mesmas tarefas em programas com características e facilidades diferentes.

ANEXO 2

TRABALHOS RELACIONADOS

Com o surgimento das linguagens de programação de alto nível, foram realizadas várias experiências sobre o ensino de linguagens de programação. As metodologias de ensino têm-se situado, em geral, no uso de material instrucional convencional (manuais, livros) e, recentemente, no uso de sistemas automatizados. Neste anexo procuramos apresentar os trabalhos mais significativos para a nossa investigação, ou seja, para a concepção das linhas gerais do Tutor-Prolog. Foram, assim, estudadas as metodologias utilizadas no ensino das linguagens de programação LOGO, LISP e PROLOG, para diferentes faixas etárias, tenham estas utilizado métodos tradicionais ou recorrido ao auxílio de computadores.

A - Ensino da linguagem Prolog

No caso do ensino da linguagem de programação Prolog, as experiências até então realizadas firmam-se, basicamente, no uso de material instrucional convencional. Este trabalho, no entanto, procura testar a metodologia do ensino automatizado da Programação em Lógica, particularmente da linguagem de

programação Prolog, através da idealização, concepção e desenvolvimento do Tutor-Prolog. Assim, consideramos de grande interesse para o trabalho que desenvolvemos, entre outras, as seguintes experiências:

(a) Experiências com o ensino do Prolog

A experiência do Imperial College intitulada "Logic as a computer language for children" foi fundamental para nós, pois o objectivo do Tutor-Prolog é ensinar Prolog. O projecto foi dirigido por Robert Kowalski [KOW 82], e conduzido por Richard Ennals [ENN 81a,81b,81c] do Imperial College de Londres. Envolveu crianças com idade entre 10 e 11 anos, que frequentavam a escola pública "Park House Middle School" em Wimbledon.

O material empregado compreendeu manuais para o professor, manuais para os alunos e recursos de máquina. No que se refere à máquina, no primeiro trabalho (1978) foi utilizado o computador do Imperial College ligado via telefone com a escola.

Com o material pedagógico descrito, foram feitos vários tipos de actividades educacionais, tais como: exposição de novos pontos ou de exemplos difíceis de serem desenvolvidos no quadro-preto, trabalhos individuais que envolviam a impressão de resultados, respostas a questionários com posterior correcção do professor (individual ou colectiva), consulta a bases de dados,

escrita de programas, uso de programas previamente preparados e trabalhos em grupos envolvendo a impressão de exercícios ou o desenvolvimento de programas. O material de acompanhamento foi utilizado de acordo com as exigências e a situação de cada turma.

Os assuntos utilizados nas experiências pertenciam à História e à Geografia, como é exemplificado na figura 2.1.

```
-----------------------------------------------------------------
Inglês: What is the population of Italy?
Prolog: which(x:Italy population x)

Inglês: What happened before the Treaty of Rome?
Prolog: which(x:x before(Treaty of Rome))

Inglês: What are the principles of the Socialists?
Prolog: which(x:Socialists principle x)

Inglês: What are the names of men who are heads of their
        households?

Prolog: which(x:x sex Male and
                x relation Head)

               (William Leck)
               (Richard Truman)
               (Benjamin Poyner)
        No (more) answers
-----------------------------------------------------------------
```

Figura 2.1 A Experiência do Imperial College

Esta experiência é importante por ter sido a primeira a investigar o ensino da linguagem de programação Prolog fora do âmbito do ensino universitário.

No Instituto Weizmann [SCH;MAL;SHA 87], em Israel, estuda-se a introdução da linguagem Prolog no currículo dos cursos de ensino primário. Pretende-se que o Prolog seja a primeira

linguagem a ser ensinada tanto para os professores, como para os alunos. Como justificação para esta escolha, o grupo de investigadores apresenta os seguintes argumentos:

- a linguagem Prolog proporciona uma oportunidade para o ensino de vários aspectos da Lógica e simultaneamente introduz o uso do computador;
- é orientado para o processamento simbólico e também executa processamentos numéricos, e
- é uma linguagem de alto nível fácil de ser ensinada.

O curso desenvolvido naquele Instituto é administrado por um programa que controla as etapas do ensino. Durante o curso os alunos devem realizar operações de inserção, de remoção e de consulta utilizando elementos colocados dentro de uma "caixa" desenhada no terminal de vídeo. Toda a interacção é orientada por menus.

Na primeira etapa a caixa encontra-se vazia e os alunos devem realizar operações com objectos. Por exemplo:

Inserir

pera
maça

Na segunda etapa, os alunos devem colocar dentro da caixa frases que contenham ideias verdadeiras. Isto é, frases definidas positivas como:

a_vida_é_bonita.

Na terceira etapa, os alunos devem fazer perguntas a

respeito do conteúdo da caixa. As perguntas podem conter a conjunção (,) e a disjunção (;).

```
?- a_árvore_é_verde , a_árvore_é_alta.
?- hoje_chove ; hoje_faz_sol.
```

Nesta etapa do curso é introduzida, também, a ideia da construção de regras.

```
está_nublado :- está_chovendo.
```

Na quarta e última etapa, o aluno é levado a introduzir na caixa, que é dividida ao meio no sentido vertical, as frases em língua natural num lado, e no outro, a sua representação em Prolog. Após esta etapa, são repetidas as etapas dois e três, mas utilizando a sintaxe normal da linguagem Prolog.

O processo foi testado com 10 alunos e seus professores, e no final do curso todos formularam projectos para serem aplicados nas disciplinas de Biologia, Química e História.

(b) O tutor APT

O APT - Active Prolog Tutor - é comercializado pela empresa Solution Systems e tem como objectivo o ensino da linguagem Prolog. No ambiente tutorial são permitidas apenas algumas

actividades de carácter educacional, não sendo as aplicações comerciais suportadas pelo ambiente. O método de ensino é baseado em textos e no uso do interpretador Prolog. A sintaxe ensinada é compatível com várias implementações da linguagem e corre em ambiente IBM PC, XT, AT, ou compatíveis, numa configuração PCDOS 2.0 com um mínimo de 512 K.

A estratégia de ensino é através de exemplos. A apresentação do material é feita com o uso de duas janelas permanentes. A primeira contém o enunciado do problema e a segunda, o código. Sempre que for necessário, é criada uma terceira janela contendo explicações de ajuda.

A sua principal característica é o recurso à três tipos de mecanismos de acompanhamento do rasto da execução dos programas ("trace"):

- o acompanhamento da execução normal da linguagem e da ordenação das cláusulas;
- o acompanhamento da passagem de parâmetros entre as cláusulas durante a execução de programas recursivos e
- o acompanhamento da execução das metas (retrocesso).

O APT não gera o modelo do aluno, o que faz com que o processo de ensino seja sempre o mesmo (pré-definido), isto é, ele independe da evolução da interacção e dos conhecimentos de cada aluno.

B - Ensino da linguagem Lisp

(a) Os tutores inteligentes para o ensino da linguagem LISP

Ao contrário do que acontece com o ensino da linguagem Prolog, para o ensino da linguagem LISP, existe já bastante trabalho realizado no desenvolvimento de tutores inteligentes.

Passamos, agora, a descrever os aspectos mais importantes dos tutores Lisp desenvolvidos por Anderson e pela sua equipa.

No que se refere ao uso de estratégias de ensino, os sistemas tutoriais desenvolvidos pelas equipas de Anderson [AND 85a] e de Reiser [REI 85], apresentam contribuições importantes para o estado da arte dos TI's. Os seus tutores foram desenvolvidos para o ensino de disciplinas como a Matemática (Geometria) e a programação (Lisp). Tal como no nosso caso, o desenvolvimento destes tutores foi baseado em investigações empíricas. As observações envolveram apenas alunos universitários.

A estratégia dos tutores de Anderson consiste em acompanhar a interacção do aluno passo a passo e em intervir quando este comete um erro.

No tutor Lisp [REI 85], o estudante interage com o tutor para escrever o seu programa Lisp. A interacção é feita em duas etapas: o tutor apresenta o problema e o estudante responde

escrevendo a solução (a codificação do problema).

A análise da solução proposta é feita mediante uma base de conhecimentos, onde estão representadas as várias formas em que o problema pode ser codificado. O processo de análise é auxiliado por uma base de dados, que contém os vários erros que podem ser cometidos por programadores iniciados (catálogo de erros).

Estes tutores geram, ainda, uma sequência particular de mensagens tutoriais de acordo com uma sequência particular de aplicações de regras. Assim, o número de lições diferentes que podem ser geradas é bastante diversificado. Por exemplo, se o aluno usar uma combinação de regras **x**, **y**, **z** receberá a lição A. Se usar a combinação **x**, **y**, **w** receberá a lição B. Isto é, o aluno pode receber como retorno uma simples mensagem ou um texto com explicações mais detalhadas. A estratégia do tutor depende das acções do aluno.

Este tutor possui dois tipos de modelo, o do aluno ideal (modelo do Tutor) e o particular de cada aluno (real).

No modelo ideal estão os conhecimentos necessários para se resolverem problemas. A organização destes conhecimentos é feita através do uso de regras de produção. As regras descrevem os possíveis erros, que podem ocorrer durante a escrita de uma função Lisp e no uso da sintaxe da linguagem Lisp geral, representam também o uso de estratégias erradas e detectam confusões semânticas e erros na manipulação de objectos, como as

variáveis.

O modelo particular do estudante visa o fornecimento imediato e adequado da retroacção, quando ocorrem erros, e o aconselhamento na busca da solução do problema. Assim, o tutor usa o modelo ideal e os catálogos (de erros, de planos, ...) para responder às dúvidas do aluno ocorridas durante a busca da solução de um problema. Com isto, gera um conjunto de regras de produção que descrevem o modelo de cada aluno. Este modelo representa um conjunto básico de conhecimentos que o tutor julga ser do domínio do aluno, e um conjunto de possíveis ideias erradas, que o tutor julga que o aluno possui sobre o problema. Esta versão do Tutor Lisp, que é de 1985, contém lições sobre as funções básicas da linguagem Lisp e utiliza, para isso, 375 regras de produção que compõem o modelo ideal e 475 situações de erros associadas a estas regras.

Vejamos, por exemplo, uma parte do diálogo entre o aluno e tutor Lisp referente à definição da função "rightp", apresentada em seguida na figura 2.2.

Define rightp. It takes 3 arguments, side1, side2, and side3, which represent the lengths of the three sides of a given triangle, where side3 is the longest side. rightp tests whether the three sides from a right triangle, the sum of the squares of side1 and side2 is equal to the square of side3. However, your function should allow for a measurement error of 2 percent. In other words, your function should return t if the sum of the square of the first two sides is within 2 percent of the square of the third side.
For example (rightp 3 4 5.01) = t, because 3 squared + 4 squared = 25, which is within 2 percent of 5.01 squared.

1. (defun rightp (side1 side2 side3) (cond))

   Yes, you could use cond here, but there is a much easier way to test if the sum of the square of the first two sides in within 2 percent of the square of the third. You can use a predicate, since we only have one test to do.

2. (defun rightp (side1 side2 side3)

   (lessp (difference) <NUM2>))

   You want code the amount of difference between the sum of the squares of the first two sides and the square of the third side, and you do not care which of those two numbers is larger. So we want to take the absolute value of their difference.

3. (defun rightp (side1 side2 side3)
   (lessp (abs (difference <NUM3> <NUM4>))
   <NUM2>))

   Which do you want to do first?

PRESS:	IF YOU WANT TO:
1	write code to get the sum of the squares of the first two sides.
2	write code to get the square of the third side.

   Menu choice: 1

4. (defun rightp (side1 side2 side3)
   (lessp (abs(difference (plus) <NUM4>))
   <NUM2>))

   No You should create a helping function for this code, rather than coding it directly. Our function, rightp, is now 4 levels of embedding deep, and the relation we are about to code-the sum of squares of two numbers-would be a useful helping function. Thus, let's code the sum square of the first two sides using a helping function.

5. (defun rightp(side1 side2 side3)
   (lessp (abs (difference (sumsq <ARGUMENTS>) <NUM4>))
   <NUM2>))

   What do you want the helping function sumsq to do?

PRESS:	IF YOU WANT TO:
1	Take 2 arguments, the square of the side and the square of the second side. Compute their sum.
2	Take 2 arguments, the first side and the second side. Compute the sum of the squares of these numbers.
3	Have the tutor choose.

```
Menu choice: 1

No. We wouldnt need a helping function just to compute the
sum of two numbers. You should create your helping function
so that just a sum.

It would better to define sumsq so that it takes two
argumenst, id this case, [1] the first side, and [2] the
second side, and computes the sum of the squares of these
numbers.

Menu choice: 2

6. (defun rghtp (side1 side2 side3)
     (lessp (abs(difference (sumsq side1 side2) (times)))
     <NUM2>))

You should use a helping function that have already coded,
square. Since this function computes the square of a number,
you can use it to code the square of the side.

7.   (defun rightp (side1 side2 side3))) (plus)))
      (lessp (abs (difference (sumsq side1 side2) (square
       side3))) (plus)))
     I don't understand that, Rob. How are you going to code 2
     percent of the square of the third side?

[EXPLAIN]

     Ok. I'll explain what I wanted. You could either code this
     directly or create a helping function here to code 2 percent
     of the square of the third side. Let's create a helping
     function just to make things easier. We'll call this helping
     function percentage. We will define percentage when we are
     finished coding rightp.
     ..........................................................
```

Figura 2.2 Interacção no tutor Lisp

(b) O tutor LISP da Advanced Computer Tutoring, Inc

O tutor LISP é comercializado pela Advanced Computer Tutoring, Inc.. Este tutor acompanha o progresso do aluno, podendo fornecer uma retroacção ou orientação detalhada quando

necessário. Prevê dois tipos de cursos, sendo um introdutório e voltado para utilizadores da área humanística e outro, para engenheiros. Pode ser usado também para formação profissional, por exemplo, em indústrias. Este tutor é fruto das investigações realizadas na Universidade Norte Americana de Carnegie-Mellon e contempla áreas como a aprendizagem e a depuração de programas, com orientação no sentido de obter a melhor solução possível para o problema apresentado. O aluno pode ainda solicitar explicações adicionais em qualquer ponto do desenvolvimento do seu programa.

Os principais módulos que compõem o tutor Lisp são o modelo ideal do aluno, o catálogo de erros, o conhecimento tutorial e a interface baseada em menus.

O programa corre em máquinas como VAX 780, VAX 730 e Xerox 1100, e ocupa de 3 a 4 Mb para servir a dois alunos.

(c) Ensino da programação através da geração automática de programas

Com relação à geração automática de programas tem especial interesse o método proposto por Garijo e Verdejo [GAR;VER 84] no trabalho intitulado "Diseño de um sistema experto para la clasificacion de problemas y construccion de programas".

A inclusão deste tópico deve-se à importância do assunto, no que se refere à reescrita das regras realizada pelo Tutor-Prolog. A auto-programação através da reescrita tem o objectivo de

aprendizagem e evolução para melhor auxiliar o aluno.

O sistema em questão controla e guia o processo de ensino mediante as informações fornecidas pelo tradutor, que dialoga com o aluno em língua natural escrita, pelo o perito, que explica ao aluno a solução gerada para o problema proposto, e pela base de dados do aluno, composta por informações sobre o desenvolvimento da sessão.

O **tradutor** interpreta um subconjuto da língua natural escrita (Castelhano) através da qual o aluno propõe problemas e utiliza o perito para explicar ao aluno a solução gerada e para controlar a aprendizagem. A aprendizagem ao nível da língua natural não está prevista. O perito constrói um problema e o resolve, explicando passo a passo o raciocínio seguido na obtenção da solução e mostrando o conhecimento utilizado para a tomada de decisão.

O **perito** possui uma base de conhecimentos onde estão armazenados conhecimentos sobre objectos (tipos de dados, funções, predicados e enunciados); a relação conceptual entre os elementos e sua estrutura hierárquica (os algoritmos abstractos, os tipos de problemas e sua classificação, e as linguagens de programação em que os algoritmos podem ser expressos); o conhecimento a respeito da forma de raciocínio sobre os objectos (regras: pares ordenados ou de reescrita). A base de conhecimento é modificada de forma declarativa de acordo com o desenvolvimento das sessões.

Existe, ainda, no sistema, uma base de dados do aluno onde são armazenadas informações relativas à evolução das sessões e, naturalmente, do aluno.

As capacidades de raciocínio e de dedução, presentes no sistema, permitem:

- construir, automaticamente, a partir do enunciado do problema expresso em língua natural escrita, um enunciado formal e um programa;
- explicar sistematicamente o processo de construção e
- analisar a correcção de um programa, a partir do seu enunciado, e raciocinar sobre os erros cometidos pelo aluno na solução apresentada.

Os problemas passíveis de serem solucionados pelo sistema estão classificados em níveis de complexidade. Assim, por exemplo, o nível 1 engloba problemas definidos sobre tipos numéricos (inteiros, booleanos, ...) que tenham por objectivo o cálculo de funções e expressões.

A estruturação da especificação é feita através da definição dos dados e dos objectos em descrições separadas e em ordem descendente (regras de produção).

A compreensão de um enunciado dá-se da seguinte forma: toda a informação implícita ou explícita necessária para a classificação do problema é retirada do seu enunciado. A compreensão dá-se quando é possível caracterizar o problema. As informações recolhidas possibilitam a obtenção do algoritmo. Isto é, a compreensão do enunciado produz como resultado uma

representação que permite enquadrar o problema num nível hierárquico, raciocinar sobre os elementos que o compõem e associar um esquema de tratamento sobre o qual possa ser baseada a construção de um algoritmo e posteriormente o programa.

A representação interna da percepção corresponde a um grafo semântico onde os nós não terminais são enquadramentos conceptuais, os arcos estão etiquetados pelas descrições do enquadramentos e os nós terminais contém valores concretos. A estrutura de um problema baseia-se portanto no enunciado, no algoritmo e no programa. O enunciado, feito em língua natural escrita, não deve conter ambiguidades. O tradutor, que interpreta o enunciado, passa ao perito a sua definição formal. Para obter uma solução, o perito deve conhecer os objectos e as suas propriedades, recorrer a uma metodologia de resolução e raciocinar sobre as suas decisões.

Na compreensão do enunciado é verificada a coerência semântica do texto e a existência da possibilidade de realizar os objectivos nele contidos. Estas operações correspondem ao primeiro estádio de tratamento.

O segundo estádio corresponde à construção do algoritmo através do uso de abstracções sobre os objectos caracterizados no enunciado e da classificação do problema numa arquitectura descendente.

Para finalizar este item sobre a construção de TI's é

importante lembrar os trabalhos de Clancey e sua equipa no desenvolvimento das várias versões dos sistemas MYCIN, NEOMYCIN e GUIDON. Estes trabalhos nas áreas da medicina e da engenharia, no período de 1979 a 1982 [CLA 82] e as ilações feitas a partir do desenvolvimento destes sistemas [KEA 87], infuenciaram indirectamente o nosso trabalho.

Assim, as áreas tratadas no decorrer da investigação daquele grupo podem ser resumidas nos seguintes tópicos gerais:

- representação modular do conhecimento organizado em regras de produção;

- transcrição para o inglês do conhecimento representado nas regras;

- desenvolvimento de mecanismos de acompanhamento dos passos envolvidos por um raciocínio ("history trace");

- desenvolvimento de uma gramática que serve para a interpretação sintáctica e para raciocinar a cerca do próprio sistema;

- desenvolvimento de um subsistema para a apresentação do vocabulário que pode ser usado nas questões possíveis de serem formuladas a respeito do raciocínio utilizado pelo sistema MYCIN na realização de um diagnóstico ("Por que é que você pergunta X?", "Como é que você usa X para concluir a respeito de Y"?);

- exploração da estrutura e das estratégias do meta-conhecimento necessárias para os sistemas MYCIN e GUIDON (estudos epistemológicos);

- construção do modelo do aluno a partir do meta-conhecimento do sistema (imagem);

- determinação das meta-estratégias necessárias para um diagnóstico;

- caracterização da estrutura que mais se adequa ao meta-conhecimento, para uma grande base de conhecimentos (NEOMYCIN); e

- caracterização do grau de independência necessário entre os programas e o conhecimento para a realização dos diagnósticos (investigação das falhas computacionais ocorridas no diagnóstico).

É natural que toda esta experiência sirva de base para qualquer trabalho na área de TI's, pois aborda tópicos necessários ao seu desenvolvimento. No entanto, deter-nos-emos apenas em trabalhos que infuenciaram directamente a concepção, a idealização e a construção do Tutor-Prolog.

C - Ensino da linguagem Logo

Embora as experiências que recorrem à linguagem LOGO no ensino não usem técnicas de IA, ao contrário do que ocorre com os TI's, a sua inclusão nesta breve síntese justifica-se pela importância que tem na introdução do uso dos computadores como ferramentas de ensino.

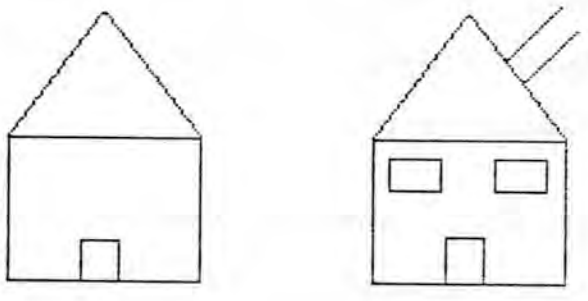
Os primeiros trabalhos realizados por Papert [PAP 72;80]

foram influenciados pelas ideias de Piaget e sugerem que é mais importante ajudar as crianças a desenvolver e a aperfeiçoar suas próprias teorias do que ensinar-lhes as teorias que consideramos correctas [PAP 80]. Dentro desta linha, a metodologia básica da linguagem LOGO é a de utilizar o computador como um simulador que ajuda a explorar conceitos.

Assim, e tendo em vista a importância das experiências realizadas com LOGO para os objectivos específicos do desenvolvimento do Tutor-Prolog, torna-se importante citar os estudos de Mullan [MUL 84], relatados no seu livro "Children and computers in the classroom". O trabalho compreende vários aspectos, entre os quais: a apresentação de uma possível metodologia para a introdução do computador e do ensino da linguagem de programação Logo no currículo das escolas do ensino básico. De acordo com esta metodologia, a linguagem Logo deve ser introduzida gradualmente, tentando despertar a curiosidade e a motivação do aluno para que venha a desenvolver os seus próprios programas. As primeiras lições baseiam-se no uso da "tartaruga" para o desenvolvimento de motivos geométricos. Com base neste assunto (Geometria), são introduzidos os demais comandos e conceitos da linguagem. A metodologia utiliza também jogos pedagógicos geralmente bastante atraentes para a faixa etária em questão (ensino básico).

Ainda com relação à utilização da linguagem Logo, participámos (como observador) durante o ano de 1985 das

experiências realizadas no Laboratório de Estudos Cognitivos (LEC) da UFRGS. Os trabalhos, então realizados neste laboratório, possuiam um carácter interdisciplinar e destinavam-se a observar a forma como se dá o processo de aprendizagem em crianças em idade escolar (ensino básico). Nas disciplinas de Português e de Matemática (Geometria), o processo consistia na pré-programação de programas núcleos que, num primeiro passo, deveriam ser explorados pelos alunos e, num passo posterior, ser alterados visando a inclusão das ideias do aluno. No que se refere à geometria, eram apenas utilizadas operações que envolvessem a manipulação da "tartaruga". Já em relação ao Português, as actividades envolviam a manipulação da estrutura lista para a concatenação de palavras visando a geração de frases. Na figura 2.3 apresenta-se o exemplo de um núcleo de problema e da alteração realizada pelo aluno.



núcleo inicial

alterações feitas pelo aluno

Figura 2.3 A experiência do LEC

Na experiência da UFRGS foram observadas crianças com

desenvolvimento mental normal e crianças com problemas neurológicos (mentais ou de coordenação motora)

Esta observação serviu-nos principalmente como base para a organização hierárquica dos problemas que compõem o Tutor-Prolog.

D - Diagnóstico de erros em programas

Devido à importância da área da depuração de programas para os sistemas ICAI e principalmente, devido à importância no ensino da programação, incluimos aqui, uma pequena apresentação do trabalho desenvolvido por Shapiro na área da depuração de programas Prolog e do sistema PROUST que é voltado para a depuração de programas Pascal.

No campo da depuração de programas em Prolog, o trabalho desenvolvido por Shapiro [SHA 82] contempla aspectos semelhantes aos abordados no Tutor-Prolog, ou seja: a meta-interpretação e a depuração de programas Prolog.

No seu trabalho, Shapiro desenvolveu programas meta-interpretadores para demonstrar o funcionamento da maquinaria interna da linguagem e para orientar o utilizador na depuração de programas Prolog, em particular, os recursivos. A principal característica destes meta-interpretadores é a execução da depuração de forma interactiva. Detectada uma falha, o utilizador é orientado no sentido de realizar a sua remoção. Com este

objectivo, o meta-interpretador interpreta passo a passo o programa do aluno, permitindo-lhe o acompanhamento e o controle do processo de meta-interpretação. Veja, por exemplo, o processo interactivo apresentado na figura 2.4.

```
isort(Xs,Ys) <--
     Buggy insertion sort.

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]) <--
      X > Y, insert(X,Ys,Zs).
insert(X,[],[X]).
```

Usando-se o diagnóstico ascendente desenvolvido por Shapiro, o processo de depuração interactiva dar-se-á da seguinte forma:

```
false_solution(isort([3,2,1],X),C)?

Is the goal isort([],[]) true?
true.
Is the goal insert(1,[],[1]) true?
true
Is the goal insort([1],[1]) true?
true.
Is the goal insert(2,[1],[2,1]) true?
false.

X = [3,2,1],
C = insert(2,[1],[2,1]) <-- 2 >= 1.  o contra exemplo
```

Figura 2.4 O método utilizado por Shapiro

O sistema PROUST, desenvolvido por Johnson e Soloway em 1983 e 1985 [JOH;LEW;SOL 85], detecta erros em pequenos programas escritos em Pascal, por programadores iniciados.

PROUST explora a ideia da identificação de erros descritos em bibliotecas que possuem catalogados os erros que frequentemente ocorrem em programas escritos por programadores inexperientes. As bases de conhecimentos são organizadas através de redes semânticas. No sistema existe uma biblioteca de planos de programas, outra sobre conhecimentos de programação e, naturalmente, a que contém os erros de programação que ocorrem com frequência nos programas escritos por novos programadores.

Durante a análise de um programa já concluído pelo programador, PROUST gera hipóteses sobre a intenção do programador e sobre o que ele deseja realizar com o programa. Estas hipóteses referem-se a pontos específicos de um programa Pascal, como os pontos de decisão, a recursão, a entrada de dados, a falta de iniciação de variáveis, as variáveis declaradas mas nunca utilizadas, os conceitos incorrectos a respeito da programação e as falhas na integração entre as rotinas de um mesmo programa. No sistema PROUST não existem funções de ensino (lições). PROUST é um sistema que obtém sucesso no diagnóstico de erros em programas escritos por alunos.

D - Interpretação da língua natural escrita

O nosso objectivo, ao incluir este assunto no Tutor-Prolog, relaciona-se com o surgimento da necessidade prática de

possibilitar aos alunos uma forma alternativa aos menus de comunicação. Assim, buscando a flexibilização do sistema, optou-se pela inclusão de um pequeno módulo de Processamento de Língua Natural (P.L.N.), pois é a forma natural que os alunos recorrem quando não encontram opcões ao nível dos menus, que satisfaçam as suas necessidades. Por outro lado, interessa-nos observar como os sistemas de P.L.N. evoluem e se adaptam aos seus utilizadores (modelo da interface).

Dentro destes dois objectivos gerais o protótipo desenvolvido utiliza as ideias dos sistemas T.E.A.M. (Transportable English database Access Medium) [GRO;APP;MAR;PER 87] e do sistema T.E.L.I. (Transportable English-Language Interface) [BAL;STU 84] e [BAL 86]. Ou seja, a construção de sistemas de processamento de língua natural que permitam aos seus utilizadores a definição, a visualização e a actualização da informação associada a palavras e a estruturas fraseológicas conhecidas.

Esta ideia vem de encontro às necessidades observadas durante os testes com o primeiro protótipo de P.L.N. do Tutor-Prolog, que tinha um carácter estático tradicional, isto é:

a) não dava a conhecer aos seus utilizadores o estado do seu conhecimento (que palavras e que tipos de frases era capaz de processar) e

b) não possuia capacidades de aprendizagem de novos

vocábulos ou de novas estruturas fraseológicas.

A experiência demonstrou que é impossível ao projectista, ou mesmo aos utilizadores, prever todas as palavras e todas as construções que são utilizadas durante a consulta a uma base de dados.

Ainda, devido a esta finalidade - consulta a bases de dados - é necessario que todas as palavras que compõem uma consulta sejam analisáveis, para que o sistema possa responder correctamente. Logo, é importante que o P.L.N. tenha mecanismos que possibilitem a aprendizagem e a consulta ao utilizador nos casos de desconhecimento de palavras ou de ambiguidades surgidas durante a análise da frase.

O Tutor-Prolog, no que se refere à aprendizagem linguística, possui dois tipos de interacções:

- o aluno interage com o tutor, através de menus, sempre que ocorrer uma situação de conflito; ou
- o aluno autoriza o tutor a assumir as suas hipóteses como verdadeiras e no final da interacção, caso o tutor tenha assumido algumas hipóteses incorrectas, corrige-as.

(a) O sistema T.E.A.M.

Em relação à interpretação da língua natural escrita, o

sistema T.E.A.M. (Transportable English database Access Medium) [GRO;APP;MAR;PER 87] é o que apresenta maiores semelhanças com a nossa investigação, em particular no processo de aquisição de conhecimentos.

O T.E.A.M. foi concebido para interactuar com dois tipos de utilizadores : o perito em base de dados e o utilizador final. E através da comunicação com o perito que se realiza a aquisição dos conhecimentos necessários para que o sistema possa adaptar-se a novas bases de dados ou para expandir as suas potencialidades de resposta às interrogações acerca da informação contida numa base de dados já existente (por exemplo, aprendendo novos verbos, adjectivos, nomes ou sinónimos de palavras já conhecidas). Assim, podemos identificar dois modos de funcionamento: o de aquisição de conhecimento e o de interrogação à base de dados e geração de respostas.

No Tutor-Prolog temos apenas um utilizador: o aluno que fornece informações para o módulo de aquisição de conhecimento e que deseja consultar a base de conhecimentos do Tutor-Prolog (conhecimentos sobre Prolog) e criar a sua base de dados (informações referentes a sua aplicação). Assim, no nosso caso, não existe o perito. O Tutor-Prolog possui, ainda, mecanismos que permitem o armazenamento de informações numa base de dados do aluno.

Vejamos os dois modos de funcionamento do T.E.A.M.:

- O módulo de interrogação e resposta possui duas componentes fundamentais: o sistema "dialogic", responsável pela representação em lógica formal do sentido das expressões da Língua Natural, ou seja, a interpretação semântica da frase, e uma segunda componente, designada por "tradutor do esquema", que faz a tradução destas formas lógicas para directivas de interrogação à base de dados.

- A aquisição permite alterar o conhecimento durante as sessões com o perito.

Uma vez que no desenvolvimento do nosso sistema não nos deteremos em aspectos de semântica e pragmática da LN, o ponto com mais características comuns é o da aquisição de conhecimentos sintácticos, pelo que analisaremos as componentes afectadas por este processo e o modo como ele é realizado.

A linha de investigação que seguimos consiste na criação de um modelo que identifique as características de construção gramatical das frases usadas por cada aluno. Repare-se que os utilizadores do Tutor-Prolog, que constituem um conjunto heterogéneo, interagem com o sistema com o intuito de extrair informações já conhecidas pelo sistema ou que ele próprio anteriormente criou. A aquisição de conhecimentos tem, para nós, o objectivo de adaptar cada vez mais o interpretador da língua natural escrita às palavras e às estruturas proposicionais utilizadas pelo aluno. Assim, nosso objectivo é o da aprendizagem ao nível do dicionário morfo-sintáctico e para a reescrita das regras sintácticas.

O "léxico" do sistema T.E.A.M. é uma base de conhecimento

que contém a informação essencial à analise morfológica, sintáctica e semântica do contexto onde cada palavra pode estar inserida. As palavras do "léxico" estão logicamente divididas em duas classes: a classe fechada (totalmente definida) e a classe aberta (onde pode haver aprendizagem). As pertencentes à classe fechada (pronomes, conjunções e determinantes) são de número finito e geralmente não muito elevado. Estas palavras ocorrem com bastante frequência em interrogações a qualquer base de dados, tendo funções gramaticais especificas e uma interpretação semântica definida e independente do dominio em causa. As palavras da classe aberta (nomes, verbos e adjectivos,...) estão relacionadas com o domínio da base de dados em causa.

A cada elemento do "léxico" são associadas informações sintácticas e semânticas. A informação sintáctica consiste numa categoria principal (por exemplo, nome, verbo, adjectivo), numa subcategoria e informação morfológica (por exemplo, graus comparativos e plurais irregulares). A informação semântica depende da categoria sintáctica.

O trabalho por nós desenvolvido, como foi referido, segue uma linha análoga à adoptada no T.E.A.M. relativamente à informação sintáctica contida no léxico.

Vejamos, ainda, alguns aspectos que consideramos importantes no sistema T.E.A.M., mas que são contemplados apenas particularmente na nossa implementação.

Uma vez que o "léxico" não se encontra todo definido à partida, é necessário que o T.E.A.M. tenha mecanismos que lhe permitam suportar diversas formas de ambiguidade lexical. A mais simples destas revela-se quando uma mesma palavra pode ter duas interpretações do ponto de vista sintáctico.

As duas outras componentes afectadas pela aquisição são o "esquema conceptual" e o "esquema da base de dados".

O "esquema conceptual" tem informação acerca dos objectos, propriedades e relações no domínio da base de dados. O "esquema da base de dados" relaciona todos os predicados no "esquema conceptual" com a sua representação na base de dados específica.

A acção destas componentes está bastante interrelacionada, pois ambas contribuem para a execução do "tradutor do esquema". O esquema conceptual, em particular, contribui para a acção da componente pragmática do T.E.A.M., que tenta resolver problemas relacionados com os "predicados imprecisos" (por exemplo, as palavras inglesas "in", "with", "has" e "of"), substituindo-os por predicados mais específicos escolhidos por rotinas especialistas (semânticas ou pragmáticas).

O "tradutor do esquema" pode ser visto como um sistema que reescreve simplificadamente a forma lógica inicial, obtendo face à base de dados as mesmas respostas.

Tentando realizar estes objectivos, T.E.A.M. é um sistema orientado por "menus" (ver figura 2.5). O sistema de aquisição

consiste em um "menu" de comandos genéricos, três "menus" associados respectivamente a relações, campos e elementos léxicos, e uma janela para resposta a questões.

No caso de ser necessária informação linguística sofisticada, como na aquisição de novos verbos, o T.E.A.M. extrairá do perito humano a informação que precisa fazendo-lhe perguntas acerca de frases simples, permitindo que este use nas respostas a sua intuição acerca da língua que conhece.

Esta flexibilidade levantará problemas, por exemplo, na aquisição de novos verbos, dado ser necessário obter informação explícita relacionada com a teoria da linguística, informação essa que irá afectar o "esquema conceptual" e o "esquema da base de dados".

Assim, o T.E.A.M. deve descobrir quantos argumentos cada verbo deve ter e se estes são ou não opcionais, os sintagmas preposicionais que podem ser usados com o verbo, etc.

O facto dos verbos serem usados na resposta a questões sobre uma base de dados relacional, simplifica o processo de aquisição. Por exemplo, nenhum verbo exigirá um complemento.

Como nosso protótipo limitou-se à aquisição sintáctica, os problemas semânticos e pragmáticos aqui apresentados não serão abordados.

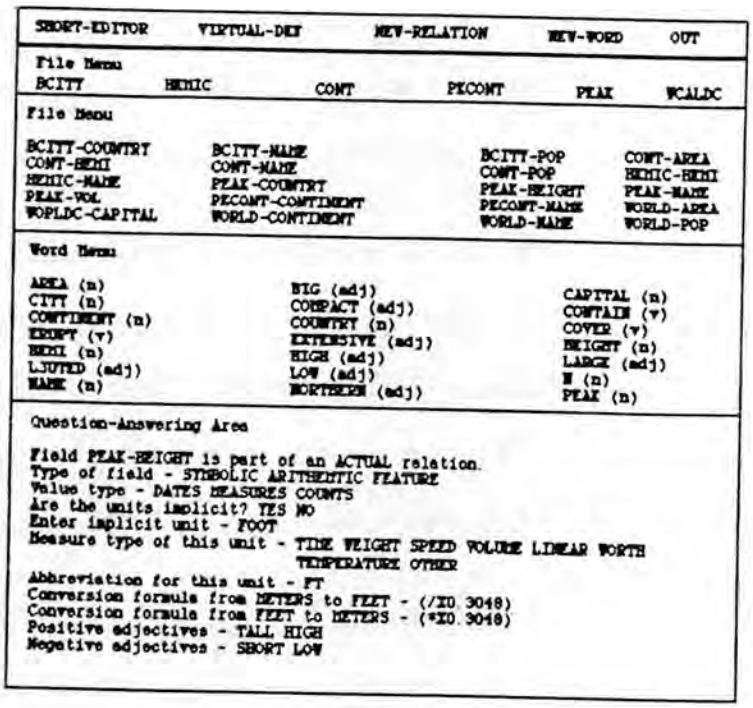


Figura 2.5 A interacção no sistema T.E.A.M.

(b) **O sistema T.E.L.I.**

O objectivo de obter um sistema de processamento de língua natural (P.L.N.) que permitisse aos seus utilizadores a definição, visualização e actualização da informação associada a palavras e estruturas fraseológicas conhecidas, levou ao desenvolvimento do sistema T.E.L.I. ("Transportable English-Language Interface") [BAL;STU 84].

Ao considerar como objectivo no desenvolvimento do T.E.L.I. a capacidade de aprendizagem de novo vocabulário e sua semântica, tenta-se minorar uma certa incomodidade verificada com os sistemas de P.L.N., que:

(1) não dão a conhecer aos utilizadores o seu estado de conhecimento (que palavras e que tipos de frases são capazes de processar) e

(2) não vão ao encontro das necessidades dos seus utilizadores no que se refere ao vocabulário, à sintaxe e à semântica. A experiência demonstrou ser impossível aos utilizadores ou aos projectistas dos sistemas, prever antecipadamente todas as palavras, significados associados e frases que ocorrerão durante uma interrogação a uma base de dados.

Portanto, o sistema T.E.L.I. foi construido com a intenção de ser de tal modo "transportável" que a especificação do conhecimento adquirido seja feita pelos utilizadores finais em qualquer momento, durante o processamento da frase em questão.

O T.E.L.I. possibilita o P.L.N. independente do domínio mantendo uma informação detalhada acerca dos tipos de frases que, de acordo com cada caso (ou domínio), serão reconhecidos pelo sistema.

O conhecimento adquirido pelo sistema estará também, em determinadas situações, ao dispor do utilizador, quer para consulta, quer para actualização. As situações onde isso se poderá verificar são as seguintes:

- quando o sistema é pela primeira vez confrontado com uma nova base de dados;
- quando o conhecimento for directamente solicitado pelo utilizador. Enquanto decorre a interacção, o utilizador pode perguntar quais as palavras e frases associadas com objectos de um domínio específico;
- quando novas palavras são acrescentadas ao vocabulário do sistema;
- após falhar o reconhecimento de uma frase devido à existência de uma palavra desconhecida; e,

- quando a informação semântica deve ser tida em conta. Os utilizadores, especificando as relações sintácticas das frases desejadas, podem modificar as definições correntes das estruturas verbais, preposicionais, etc. Para isso, deverão especificar as relações sintácticas referentes às frases em questão.

Vejamos agora, de modo breve, como é feita no T.E.L.I. a aquisição do conhecimento e o modo como este se encontra representado.

O T.E.L.I. encontra-se preparado para processar cinco tipos de frases: "Adjective phrase", "Noun-Modifier phrase", "Prepositional phrase", "Verb phrase" e "Functional Noun phrase". Para ilustrar o processo de aquisição, utilizaremos um tipo de frase que envolva situações representativas do processo geral, no caso, as "adjective phrases".

Antes de ser fornecida informação ao sistema (sobre o tipo de frases a reconhecer), este deve conhecer a classe de cada um dos seus componentes. Cada componente poderá ser de classe "fechada" ou "aberta", apenas sendo permitida a aprendizagem de novos vocábulos pertencentes à "classe aberta". Para a completa especificação do tipo de frases a ser interpretada, o projectista da interacção deverá, para cada componente, indicar:

- "O nome interno" a ser usado durante o processamento;
- "O tipo de enquadramento" da componente;
- "O nome externo" que poderá ser usado no diálogo com o utilizador.

Considerando o exemplo das "adjective phrases", o seu modelo poderá ter a seguinte especificação :

```
(adjinfo
 (cabeca        entidade        "Sujeito"    )
 (adj           adjectivo       "Adjectivo"  )
 (prep          prep            "Preposição")
 (obj           entidade        "Objecto"    ),
```

onde :

- "adjinfo" é um símbolo a ser usado internamente para referenciar a especificação das "adjective phrases";
- "cabeca", "adj", "prep" e "obj" são nomes internos arbitrários;
- "entidade", "adjectivo" e "prep" denotam partes do discurso. No entanto, "entidade" denota um subconjunto de nomes que constitui o "tipo de objectos" do domínio em causa e
- finalmente, os nomes externos "Sujeito", etc, podem ser qualquer sequência de caracteres cujo texto faça sentido do ponto de vista do utilizador.

Em seguida, o projectista da interacção deverá especificar um número arbitrário de metavariáveis com as quais se irá fazer a unificação de acordo com a especificação anterioremente feita. Por exemplo, na frase:

"a room can be adjacent to a corridor"

Se o reconhecimento desta frase fosse feito como uma "adjective phrase", poder-se-ia ter a seguinte especificação:

```
(adjinfo(a Cabeca can be Adj Prep a Obj))
```

No caso de ser vantajosa a especificação de elementos opcionais, esta poderá ser feita através da inclusão desses elementos entre parêntesis, como por exemplo:

(subj verb (obj) (part) (prep obj))

Esta expecificação é, naturalmente, equivalente às oito especificações abaixo:

```
(subj verb)
(subj verb obj)
(subj verb obj part)
(subj verb obj prep obj)
(subj verb part)
(subj verb part prep obj)
(subj verb prep obj)
(subj verb obj part prep obj)
```

Colocando-nos agora numa situação de "comum" utilizador, analisaremos do seu ponto de vista o sistema T.E.L.I.. Existem 2 formas do utilizador indicar qual a informação que mais lhe convém visualizar:

- por "menus" e
- através da especificação em pseudo-linguagem (com vocábulos em inglês).

Para ilustrar uma interacção orientada por "menus", consideremos o exemplo de um utilizador que deseja saber quais os objectos do domínio que poderão estar "em" ("in") determinado "condado" ("county"). Eis na figura 2.6 a sequência de diálogo estabelecido.

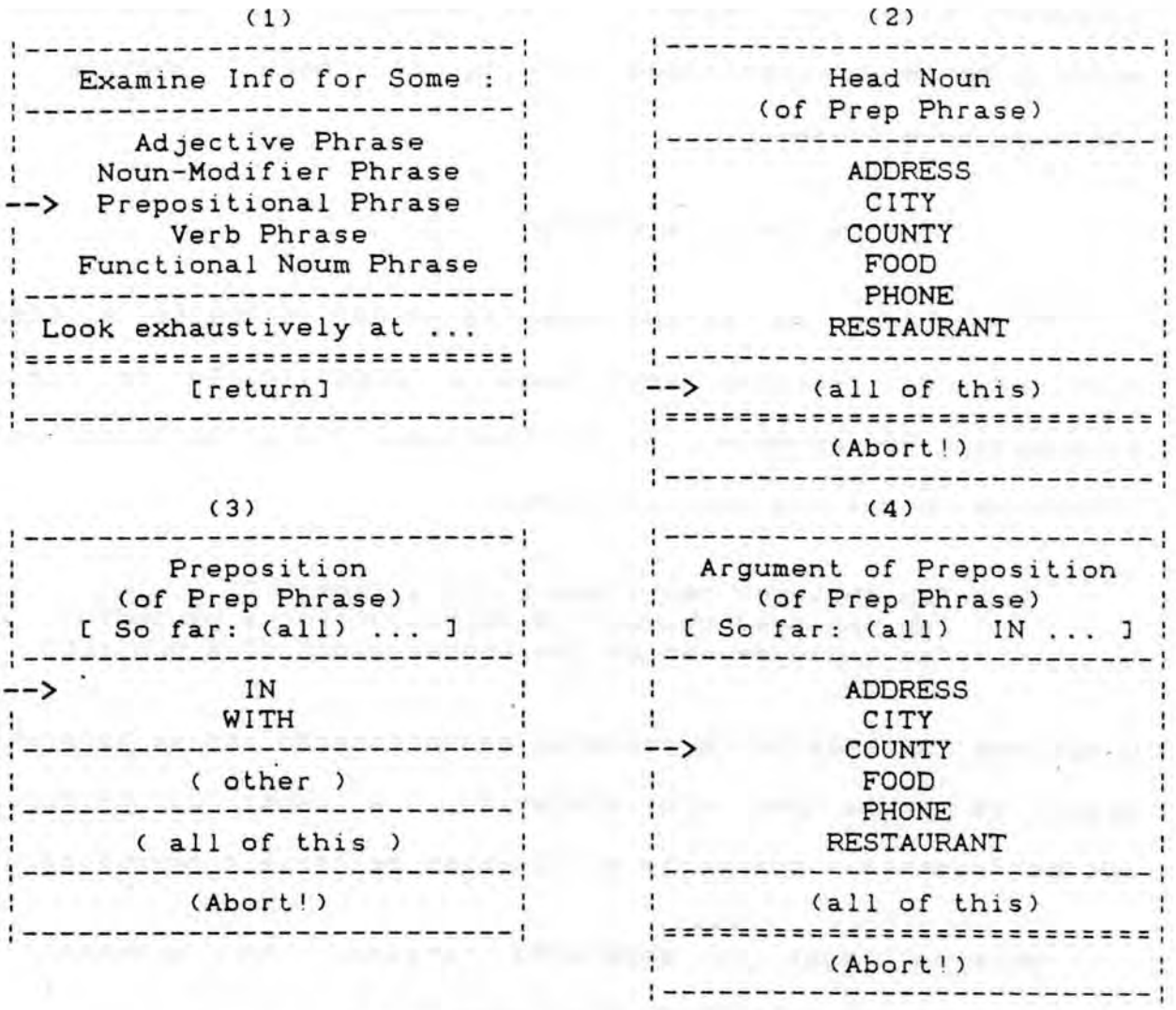


Figura 2.6 Interacção no sistema T.E.L.I.

Como pode ser observado, o sistema possibilita ao utilizador não só a selecção de um tipo específico de objecto, mas também permite que o conjunto de "tipos de objecto" sejam considerados como um todo (note-se a escolha feita no "menu" (2)).

Relativamente à especificação em pseudo-linguagem, o utilizador deverá escrever uma frase (em Inglês) indicando através do metacarácter "?" as componentes para as quais deseja

conhecer todos os valores. Por exemplo, a mesma questão anteriormente especificada através de "menus" poderia ser indicada pela frase:

"a ? can be in a county".

O processo de aprendizagem de novos vocábulos é também possível de realizar recorrendo à especifcação em pseudo-linguagem. Por exemplo, se considerarmos que as palavras entre chaves são novas nas especificações:

"an employee can (report) to a manager"
"an employee can be (responsible) for a project"
"an employee can be the (supervisor) of a project"

O sistema, através de um processo de unificação com os diferentes tipos de frase por ele conhecido, é capaz de determinar automaticamente a que parte do discurso pertence o novo vocábulo.

Note-se aqui a separação lógica, do processo de aprendizagem, em duas fases distintas:

- a primeira, onde é feita a especificação completa de uma ou mais estruturas fraseológicas conhecidas e
- na segunda, cada vocábulo desconhecido é aprendido automaticamente por confrontação da frase a interpretar com as estruturas fraseológicas conhecidas.

Pode-se ainda acrescentar que, ao contrário do diálogo orientado por "menus", a especificação através da pseudo-linguagem permite a ocorrência de certas ambiguidades. As experiências efectuadas revelaram que cerca de 80% dos casos não apresentaram

qualquer ambiguidade; 15% eram ambíguos e 5% impossíveis de qualquer tipo de processamento.

Analisaremos agora como são processadas as especificações feitas em pseudo-linguagem. Após receber uma especificação deste tipo, o sistema deve:

- determinar qual o tipo de frase que foi escrita;
- detectar quaisquer novos vocábulos na frase e
- ter em conta os metacaracteres existentes.

Por exemplo, supondo que o utilizador deseje saber quais os objectos que poderão estar associados com um empregado ("employee") e que "associado" ("associated") é um vocábulo desconhecido pelo sistema. Neste caso o sistema não conhece nada que possa estar "associado com" ("associated with") um empregado ("employee"), mas dá ao utilizador a hipótese de adicionar este conhecimento. Se o utilizador escrever:

"an employee can be associated with a ?"

Esta especificação toma numa primeira fase a forma:

"an employee can be ?? with a ?"

Onde "??" marca a posição do vocábulo desconhecido.

No passo seguinte é feita a substituição de cada palavra pela componente do discurso a ela associada.

a (noun entity) can be ?? (prep) a ?

O processo de unificação que ocorrerá obedece aos seguintes

princípios:

```
"?"       - unifica com qualquer componente do discurso;
"??"      - unifica com qualquer componente do discurso de classe
            aberta;
"x"       - unifica com qualquer x;
(x y ...) - unifica com qualquer x y ...
```

Em particular, a resultante deste processo relativamente ao exemplo em análise sera a "entity" can be "prep" a "entity".

Neste ponto a estrutura intermédia que continha "??" é reexaminada e comparada com a frase inicialmente escrita pelo utilizador. Este é então solicitado a confirmar a hipótese de que o vocábulo "associated" é um novo adjectivo. Por invocação das rotinas de actualização do léxico o novo vocábulo será então aprendido pelo sistema.

```
Phrase Type :          Adjective Phrase
Head Noum :            employee
Adjective :            associated
Preposition :          with
Object :               (all)
```

Finalmente, o sistema mostra, através de um "menu", todos os objectos com os quais um empregado pode estar associado. Os formatos escolhidos para mostrar ao utilizador a informação relevante reflectem os seguintes desejos: permitir que a informação seja simultaneamente visualizada e actualizada e minimizar o número de "menus de objectivos específicos".

Em particular, o sistema construirá, sempre que possível, um "menu" onde se poderá analisar separadamente cada componente do diálogo.

ANEXO 3

CODIGO DOS PROGRAMAS QUE COMPOEM O TUTOR-PROLOG

1. MODULO DE CONTROLE DO SISTEMA

```
ola:-
        cls,
        write('As tuas intervencoes devem'),
        write(' terminar sempre com um ponto final.'), nl,
        cor(35), write('Bem-vindo ao sistema.'),
        cor(37), write(' Tu ja o conheces?'),
        nl, tget(R,C),
        respostas,
        tmove(R,C),
        leitor(S), !,
        case2(S,A),
        controle1(A).

controle1(nao):-
        cls,
        carrega_tutor,
        [lingua,dic],dial, !,
        cls, controle.

controle1(sim):-
        menu(tutorial,Y,Z), !.

controle1(menu):-
        menu(tutorial,Y,Z).

menu(X,Y,Z):-
        cls, tmove(3,0),
        write('Tu queres trabalhar em qual dos modulos?'),
        tmove(5,40),
        ( X==tutorial,
         write('Tutorial.'),
         tmove(5,40),
         wa(9,112), cor(36),
         true ;
         write('Tutorial.')),
         tmove(6,40),
         write('Auxilio na escrita de programas.'),
         cor(33),
```

```
            tmove(7,40),
            write('Prolog.'),
            cor(36),
            tmove(8,40),
            write('Voltar atras.'),
            cor(37),
            tmove(9,40),
            write('Perguntar.'),
            tmove(10,40),
            cor(31),
            write('Remover.'),
            tmove(11,40),
            cor(37),
            write('Resumo.'),
            tmove(12,40),
            cor(35),
           ( X==fim,
            write('fim.'),
            tmove(12,40),
            wa(4,112),true ;
            write('fim.')),
            tmove(13,0),
            cor(37),
            write('Responde de acordo com as'),
            write(' possibilidades existentes? '),
           leitor(A), !,
            case_a(A,X,Y,Z).

case_a([tutorial|_],X,_,_):-
          cartut,!,tutor.

case_a([auxilio|_],X,_,_):-
          carrega_auxilio, !, cls,
          programacao.

case_a([prolog|_],X,_,_):- !.

case_a([voltar|_],X,Y,Z):-
          nl, write('Quantas licoes tu queres voltar a tras?'),
          leitor(S), !,
          volta(X,Y,Z,S).

case_a([perguntar|_],X,Y,Z):-
          [lingua],
          inicia, !,
          limpa_lingua,
          ( var(X), cls,
           controle ;
           monta_exemplos(X,Y,Z) ).
```

```
case_a([remover|_],X,Y,Z):-
          nl, write('Escreve o termo que dejejas remover:'),
          nl, nl,
          tget(R,C),
          tscroll(0,(0,0),(13,79)),
          tmove(R,C),
          info_ex,
          read(S), !,
          ( info(S),
           retract(info(S)),
           retract(S) ; nl, nl,
           write('Nao possuo esta informacao.'), nl ),
           tscroll(0,(0,0),(13,79)),
           tmove(R,C),
           info_ex,
           monta_exemplos(X,Y,Z).

case_a([fim|_],X,Y,Z):-
           !, adeus(X,Y,Z).

case_a([resumo|_],X,Y,Z):-
           cls,
           lista_possibi(Z),
           tmove(22,0),
           write('De que modulo?'), nl,
           leitor(S),
           ( in(tutor) ;
            cartut,[bd]),
            case_res(S).

case_a([_|_],X,Y,Z):-
          write('Responde de acordo com as possibilidades'),
          write(' exixtentes?'),
          leitor(A), !,
          case_a(A,X,Y,Z).

case_res([facto|_]):-
          cls,
          ( passou(facto) ;
           carrega1),
           monta_exemplos(facto,2,nao).

case_res([lista|_]):-
          cls, tmove(6,0),
          ( passou(lista) ;
           carrega4 ),
           monta_exemplos(lista,5,_).

case_res([meta|_]):-
          cls,
          ( passou(meta) ; [metas2] ),
           monta_exemplos(meta,11,nao).
```

```
case_res([regra|_]):-
        cls,
        ( passou(regra) ; [regras] ),
         monta_exemplos(regra,0,sim).

case_res(_):-
        nl,
        write('Nao possuo o resumo referente a este modulo.'),
        nl.

volta(X,Y,Z,[H|T]):-
        ( number(H),
         ( H>1,Y1 is ( Y - H ),
          cls, tmove(24,0),
          monta_exemplos(X,Y1,Z) ;
          monta_exemplos(X,Y,Z) ) ).

volta(X,Y,Z,A):-
          nl,
          write('Responde com um numero.'),
          case_a([voltar|_],X,Y,Z), !.

tutor:-
          cls,
          open(Handle,'arq.ari',r),
          repeat,
             le(Handle,X),
          close(Handle), !,
          continuacao,
          nl, write('Qual e o conteudo que tu queres estudar?'),
          open(H,'arq2.ari',r),
          repeat,
             le2(H,X1),
          close(H), !,
         ( nome(O,O1,_,_,_,_,_), true ;
          [bd]),
          inicia_cont,
          menu2.

menu2:-
        ( ponto(P,Z,K,D),
         retract(ponto(P,Z,K,D))
         indice_conhec(P,Z,K,D)
        ; true ),
         lista_possibi(Z),
         nl,
         leitor(S), !,
         tscroll(0,(0,45),(10,79)),
         casecont(S,A),
        ( A==Z,
         cont_cont(A,K,D) ;
         cont_cont(A,0,YN) ).
```

```
continuacao:-
        continua(X,Y,Z),
        assertz(ponto(P,X,Y,Z)).

continuacao:-
        assertz(ponto(P,facto,0,sim)).

inicia_cont:-
        assert(cont_adeq(0)),
        assert(cont_inad(0)),
        assert(cont_sim(0)),
        assert(cont_nao(0)).

conteudo:-
        msg0(inicio,16),
        traco,
        segue.

imprime_ex(P,N,YN):-
        tmove(24,0),
        nl, write('Entendeste o exemplo? '),
        tget(R,C),
        respostas, !,
        tmove(R,C),
        nl, leitor(Y), !, nl,
        tscroll(0,(24,0),(24,0)),
        tmove(24,0),
        segue_curso(P,N,Y,YN).

segue_curso(P,N,[sim|_],YN):-
        N1 is N+1,
        retract(cont_sim(Y1)),
        assert(cont_sim(N1)),
        monta_exemplos(P,N1,sim).

segue_curso(P,N,[nao|_],YN):-
        retrac(cont_nao(Y1)),
        Y2 is Y1 + 1,
        assert(cont_nao(Y2)),
        monta_exemplos(P,N,nao).

segue_curso(P,N,[menu|_],YN):-
        !, cls,
        menu(P,N,YN).

perg_exec(B):-
        !, traco,
        write('Queres tentar executar o exemplo?'), nl,
        leitor(A), !,
        case2(A,B).
```

```
indice_conhec(P,X,Y,Z):-
          retract(cont_sim(S)),
          assert(cont_sim(0)),
          retract(cont_nao(N)),
          assert(cont_nao(0)),
          retract(cont_adeq(A)),
          assert(cont_adeq(0)),
          retract(cont_inad(I)),
          assert(cont_inad(0)),
          K is A+I,
          J is I*100,
          Ind_ina is J/K,
          ( N > 0,
           ( Ind_ina > 0.5,
            assert(imagem_nivel(0)) )
           ; assertz(ponto(_,P,Y,nao)) ).


% ligação com programas externos
% exemplo par o caso do BASIC

:inicio
api
c:\gwbasic %1
pause
goto inicio
```

1.1 Controle do módulo tutorial

```
%controla o carregamento dos modulos para o ensino da linguagem Prolog

le2(H,Z):-
          read(H,X),
          ( X==end_of_file, !
          ; sel_info(X), fail ).

sel_info(info(Z)):-
          assertz(info(Z)),
          assertz(Z), !.

sel_info.

le(Handle,Y):-
          read(Handle,X),
          ( X==end_of_file, !
          ; assertz(X), fail ).

cont_cont(exercicios,_,_):-
          menu2.

cont_cont(menu,_,_):-
          menu(inicio,_,_).

cont_cont(lista,P,Q):-
          cls, !, cor(31),
          ( abolish(monta_exemplos/3) ; true ),
          ( passou(X),
           retract(passou(X))
           ; inicia, true ),
          assert(passou(lista)),
          mens_geral(1),
          carrega4, !,
          mens_geral(2),
          tmove(24,0),
          monta_exemplos(lista,P,Q).

cont_cont(facto,P,Q):-
          cls, !, cor(31),
          mens_geral(1),
          carrega1, !,
          inicia, !,
          mens_geral(2),
          tmove(24,0),
          assertz(passou(facto)),
          monta_exemplos(facto,P,Q).
```

```
cont_cont(meta,P,Q):-
          ( passou(facto),
          retract(passou(facto) ),
          assert(passou(meta))
          ; cls, cor(31),
          mens_geral(1),
          carrega1, !,
          inicia, ! ),
          mens_geral(2),
          tmove(24,0),
          monta_exemplos(meta,P,Q).

cont_cont(regra,P,Q):-
          cls,
          ( abolish(monta_exemplos/3) ; true ),
          ( passou(X),
           retract(passou(X)) ;inicia ),
          [regras],
          assert(passou(regra)),
          monta_exemplos(regra,0,sim).

cont_cont(A,P,Q):-
          ( passou(facto) ; cls, !,
           cor(31),
           mens_geral(1),
           carrega1, !,
           inicia, ! ),
          mens_geral(2),
          tmove(24,0),
          monta_exemplos(A,P,Q).

casecont([B|_],A):-
          member(B,
          [constante,variavel,facto,meta,lista,regra,exercicios,menu]),
          !, A=B.

casecont([B|_],A):-
          write('Responda novamente.'), nl,
          leitor(S), !,
          casecont(S,A).

lista_possibi(Z):-
          tget(R,C),
          tscroll(0,(0,45),(10,79)),
          tmove(2,50),
          cor(35),
          write($CONTEUDOS$),
          cor(37),
          impr_possib(2,Z,
          [facto,constante,variavel,meta,lista,regra,exercicios,menu]
          ),
          tmove(R,C), !.
```

```
impr_possib(N,Z,[]).

impr_possib(N,Z,[H|T]):-
          N1 is N+1,
          tmove(N1,50),
          ( Z==H,
           write(H), write('.'),
           tmove(N1,50),
           wa(10,112) ;
           write(H), write('.') ),
          impr_possib(N1,Z,T).

controle:-
          ( limpa_dial, ! ; true ),
           cor(31), !,
          mens_geral(2),
          cor(37),
          entra_info(1),
          padrao, !,
          limpa_info,
          cls, nl,
          write('No modulo tutorial tu vais aprender os '), nl,
          write('seguintes conteudos:'),
          tmove(24,0),
          lista_possibi(facto),
          segue, !,
          cor(31),
          mens_geral(1),
          cartut,
          carrega1,
          cls, traco,
          msg0(licao,0),
          traco, segue, cls, traco,
          conteudo,
          inicia, !,
          assertz(passou(facto)),
          monta_exemplos(facto,0,YN).

entra_info(1):-
          write('Escreve o nome de uma pessoa:'),
          leitor(S), !, nl,
          N=1,
          seleciona(N,S).

entra_info(2):-
          write('Escreve o nome de outra pessoa:'),
          leitor(S), !, nl,
          N=2,
          seleciona(N,S).
```

```
entra_info(3):-
          ( contexto(X),
           retract(contexto(X)) ; true ),
          write('Escreve o nome de um desporto:'),
          leitor(S),
          !, nl,
          N=3,
          seleciona(N,S).

entra_info(4):-
          write('Escreve o nome de outro desporto:'),
          leitor(S),
          !, nl,
          N=4,
          seleciona(N,S).

entra_info(5):-
          write('Escreve um verbo, no infinitivo,'), nl,
          write(' relacionado com desporto:'),
          leitor(S),
          !, nl,
          N=5,
          seleciona(N,S).

entra_info(6):-
          write('Escreve outro verbo, no infinitivo,'), nl,
          write(' relacionado com desporto:'),
          leitor(S),
          !, nl,
          N=6,
          seleciona(N,S).

entra_info(7).

seleciona(N,[nao|_]):-
          cor(31),
          write('Preferes que eu use as palavras ja existentes?'),
          cor(37), nl,
          tget(R1,C),
          respostas, !,
          tmove(R1,C),
          leitor(A), !,
          case1(N,A).

seleciona(N,[H|L]):-
          N<3,
          name(H,S,_,_,_,_,_),
          incrementa(N).
```

```
seleciona(N,H):-
          N<3,
          ( contexto(X),
           retract(contexto(X)) ; true ),
          assertz(contexto(nome)),
          prepara(H), nl,
          incrementa(N).

seleciona(N,[H|L]):-
          N>4, N<7,
          verbo(H,_,_,_,_),
          incrementa(N).

seleciona(N,H):-
          N>4, N<7,
          prepara(H), nl,
          p_H1(H,H1),
          last(L,H1),
          delete(L,H1,P),
          name(H2,H1),
          name(P1,P),
          write('Este verbo requer um objecto directo?'),
          nl, read(X), !, nl,
          ( X==sim,
           assertz(verbo(H2,P1,[],H2,com)) ;
           write('A concordancia desejada e ? '),
           write(H2),
           write(' de'),
           concorda(H2,P1) ),
          incrementa(N).

seleciona(N,[H|L]):-
          N>2, N<5,
          assertz(objeto2(H)),
          incrementa(N).

concorda(H2,P1):-
          nl,
          read(X1), nl,
          enquadra(X1,H2,P1).

enquadra(sim,H2,P1):-
          assertz(verbo(H2,P1,de,H2,de)).

enquadra(nao,H2,P1):-
          write(H2),
          write(' no'), nl,
          read(X1), nl,
          ( X1==sim,
           assertz(verbo(H2,P1,no,H2,com)) ;
          assertz(verbo(H2,P1,na,H2,com)) ).
```

```
p_H1([X|_],H1):-
          name(X,H1), !.

ultimo([R1|[]],R1).

ultimo([H|T2],R1):-
          ultimo(T2,R1).

incrementa(N):-
          X is N+1,
          entra_info(X).

padrao:-
          write('Vou assumir as seguintes informacoes:'),
          nl, traco, cor(33),
          write(' NOMES DE PESSOAS'),
          cor(35), write(' VERBOS  '), cor(36),
          write('ESPORTES'),
          cor(37), traco,
          busca_padrao(0).

busca_padrao(2):-
          nl, write('Tu  estas de acordo ? '),
          leitor(S), !,
          case2(S,Z),
          ( Z==sim ; entra_info(1) ).

busca_padrao(N):-
          retract(name(X,S,U,P1,P,[X|D],D)),
          retract(verbo(Y,O,K,H2,M)),
          retract(objeto2(Z)),
          write('      '),
          write(X),
          write('         '),
          write(Y),
          write('     '),
          write(Z),
          nl, J is N+1,
          assertz(name(X,S,U,P1,P,[X|D],D)),
          assertz(verbo(Y,O,K,H2,M)),
          assertz(objeto2(Z)),
          busca_padrao(J).

case1(N,[sim|_]):-
          [bd], !.

case1(N,[menu|_]):-
          !, menu(facto,0,sim).

case1(N,[nao|_]):-
          entra_info(N).
```

```
case1(N,[_|_]):-
          write('Responde de acordo com as'),
          write(' possibilidades existentes:'), nl,
          leitor(S), !,
          case1(N,S).

mens_geral(1):-
          write('Aguarde o carregamento do sistema.'),
          cor(37), nl.

mens_geral(2):-
          cls,
          write('Sempre termina tuas intervencoes'),
          write(' com um ponto final.'),
          nl.
```

```
% Controle da interaccao Prolog (tutor e auxilio)

programacao:-
          op(1200,fx,'?-'),
          create(H,'arq.ari'),
          traco,
          write('Podes iniciar a escrita do programa.'),
          nl,
          write('Para sair deste modulo escreve fim.'),
          traco,
          ( abolish('L'/1) ; true),
          assert('L'(13)),
          apaga(13),
          tmove(24,0),
          repeat,
              veio(fim), !,
              salva_info.

veio(F):-
         nl,
         lexico(E,R,S),
         ( E=[token(fim,nome)|_],
           F=fim ;
           E=[token(menu,nome)|_],
           menu,!,
           F=continua ;
           assert(exemplo(nada)),
           analisa(T,Lista,Lisai,E,[]), !,
           retract(exemplo(V)),
           testa_execucao(V,T,R,S), !,
           F=continua ).

testa_execucao(facto,T,_,E):-
          ( info(T),
           write('Ja possuo este facto.'), nl
          ; assertz(info(T)),
           assertz(T) ),
           info_ex.

testa_execucao(meta,T,R1,E):-
          retract(meta1(P1)),
          assertz(meta2(P1)),
          info_ex,
          ( exec_ok(X),
           retract(exec_ok(X))
          ; true ), nl,
          execrec(P1,_,2,R1), !.
```

```
testa_execucao(regra,T,R,E):-
        assertz(regra(T)),
        assertz(T),
        T=..S,
        pega_meta(S,[S1],S2,S3),
        ( S3 == is,
         execute(T) ;
        ( exec_ok(X),
         retract(exec_ok(X))
        ; true ),
         calcula_tam(E,Tam), nl,
         execrec(S1,S2,Tam,R)).

testa_execucao(lista,T,_,_):-
        assertz(lista(T)).

testa_execucao(constante,T,_,_):-
        assertz(constante(T)).

testa_execucao(variavel,T,_,_):-
        assertz(variavel(T)).

testa_execucao(_,_,_,_).

pega_meta([S3,S2|S1],S1,S2,S3).

calcula_tam(E,Tam):-
        conta(E,0,Tam).

conta([45|T],N,Tam):-
        Tam=N,!.

conta([H|T],N,Tam):-
        N1 is N+1,
        conta(T,N1,Tam).

salva_info:-
        open(H,'arq.ari',a),
        guarda_factos(H),
        guarda_metas(H),
        guarda_variaveis(H),
        guarda_constantes(H),
        guarda_listas(H),
        guarda_regras(H), !,
        close(H).

exemplos(Y):-
        cor(32),
        traco,
        write('Agora escreve os teus exemplos. '),
        write('O teu exemplo deve terminar sempre por um ponto '),
        write('final. Quando nao tiveres mais duvidas escreve '),
```

```
        cor(31),
        write('fim.'),
        traco, cor(37),
        ( abolish('L'/1) ; true ),
        assert('L'(13)),
        apaga(13),
        inic,
        repeat,
          veio(Y,fim),
          inic,
          cls, !.

veio(Y,F):-
       lexico(E,R,L),
       ( E==[], !,
        F=continua
       ; ( E=[token(fim,nome)|_],
          F=fim ;
        assert(exemplo(nada)),
        analisa(T,Lista,Lisai,E,[]), !,
        retract(exemplo(V)),
        case_exmp(Y,V,T,R,L),!,
      F=continua) ).

case_exmp(Y,V,T,R,E):-
       nl, er_g(E1),
       ( E1==0,
        ( V==Y,
         retract(cont_adeq(X)),
         X1 is X + 1,
         assert(cont_adeq(X1)),
         testa_execucao(V,T,R,E),! ) ).

case_exmp(Y,V,T,R,E):-
        nl, er_g(E1),
        ( E1==0,
         retract(cont_inad(X1)),
         X2 is X1 + 1,
         assert(cont_inad(X2)),
         traco,
         cor(32),
         write('O exemplo nao e '),
         cor(31), write(Y), cor(32), nl,
         msg0(Y,_), nl, write('Tente novamente.'),
         cor(37), nl ).

case_exmp(Y,V,T,R,E).
```

1.2 Dialogo inicial para a obtenção da priemira imagem do aluno

```
dial:-
          create(Handle,'arq.ari'),
          msg(1,Handle),
          close(Handle), !,
          create(H,'arq2.ari'), !.

msg(1,Handle):-
          write('Qual e o teu nome? '),
          leitor(X), !,
          linha_d(Handle,1,X).

linha_d(Handle,1,[]):-
          !, msg(1,Handle).

linha_d(Handle,1,H):-
          assertz(nome(H)),
          write(Handle,
          nome(H)),
          write(Handle,'.'),
          nl(Handle),
          msg(2,Handle).

msg(2,Handle):-
          nl, write('Que idade tens? '),
          leitor(X), !,
          linha_d(Handle,2,X).

linha_d(Handle,2,[]):-
          !, msg(2,Handle).

linha_d(Handle,2,H):-
          assertz(idade(H)),
          write(Handle,idade(H)),
          write(Handle,'.'),
          nl(Handle),
          pega_numero(H,N,idade,2), !,
          write(Handle,idade_n(N)),
          write(Handle,'.'),
          nl(Handle),
          assertz(idade_n(N)),
          msg(3,Handle).

msg(3,Handle):-
          nl, write('Qual e o teu ano? '),
          leitor(X), !,
          linha_d(Handle,3,X).
```

```
linha_d(Handle,3,[]):-
          !, msg(3,Handle).

linha_d(Handle,3,H):-
          assertz(turma(H)),
          write(Handle,turma(H)),
          write(Handle,'.'),
          nl(Handle),
          msg(4,Handle).

msg(4,Handle):-
          nl, write('Reprovaste no ano anterior? '),
          leitor(X), !,
          linha_d(Handle,4,X).

linha_d(Handle,4,[]):-
          !, msg(4,Handle).

linha_d(Handle,4,H):-
          write(Handle,letra(H)),
          write(Handle,'.'),
          nl(Handle),
          msg(5,Handle).

msg(5,Handle):-
          nl, write('Qual foi a nota que obtiveste'),
          write(' em Portugues no ultimo'),
          write(' periodo? '),
          leitor(X), !,
          linha_d(Handle,5,X).

linha_d(Handle,5,[]):-
          !, msg(5,Handle).

linha_d(Handle,5,H):-
          assertz(portugues(H)),
          write(Handle,portugues(H)),
          write(Handle,'.'),
          nl(Handle),
          pega_numero(H,N,nota,5),
          write(Handle,portugues_n(N)),
          write(Handle,'.'),
          nl(Handle),
          assertz(portugues_n(N)),
          msg(6,Handle).

msg(6,Handle):-
          nl, write('Qual foi a nota que obtiveste'),
          write(' em Matematica no ultimo'),
          write(' periodo? '),
          leitor(X), !,
          linha_d(Handle,6,X).
```

```
linha_d(Handle,6,[]):-
          !, msg(6,Handle).

linha_d(Handle,6,H):-
          assertz(matematica(H)),
          write(Handle,matematica(H)),
          write(Handle,'.'),
          nl(Handle),
          msg(7,Handle).

msg(7,Handle):-
          nl,
          write('Qual foi a nota que obtiveste')
          write(' em Fisica no ultimo'),
          write(' periodo? '),
          leitor(X), !,
          linha_d(Handle,7,X).

linha_d(Handle,7,[]):-
          !, msg(7,Handle).

linha_d(Handle,7,H):-
          assertz(fisica(H)),
          write(Handle,fisica(H)),
          write(Handle,'.'),
          nl(Handle),
          msg(8,Handle).

msg(8,Handle):-
          nl,
          write('Qual e a profissao do teu pai? '),
          leitor(X), !,
          linha_d(Handle,8,X).

linha_d(Handle,8,[]):-
          !, msg(8,Handle).

linha_d(Handle,8,H):-
          assertz(profpai(H)),
          write(Handle,profpai(H)),
          write(Handle,'.'),
          nl(Handle),
          msg(9,Handle).

msg(9,Handle):-
          nl, write('Qual e a profissao da tua mae? '),
          leitor(X), !,
          linha_d(Handle,9,X).

linha_d(Handle,9,[]):-
          !, msg(9,Handle).
```

```
linha_d(Handle,9,H):-
          assertz(profmae(H)),
          write(Handle,profmae(H)),
          write(Handle,'.'),
          nl(Handle),
          msg(10,Handle).

msg(10,Handle):-
          nl, write('Conheces alguma linguagem de programacao?'),
          leitor(X), !,
          ( X=[sim|_],
            msg(11,Handle) ; true ).

msg(11,Handle):-
          nl, write('Qual e o nome da linguagem que conheces? '),
          leitor(X), !,
          reconhece_d(Handle,11,X).

reconhece_d(Handle,11,[]):-
          !, msg(11,Handle).

reconhece_d(Handle,N,[H|T]):-
          case(Handle,N,H).

case(Handle,N,basic):-
         write(Handle,linguagem(basic)),
         write(Handle,'.'),
         nl(Handle),
         assertz(linguagem(basic)).

case(Handle,N,assembler):-
         write(Handle,linguagem(assembler)),
         write(Handle,'.'),
         nl(Handle),
         assertz(linguagem(assembler)).

case(Handle,N,maquina,X):-
         write(Handle,linguagem(assembler)),
         write(Handle,'.'),
         nl(Handle),
         assertz(linguagem(maquina)).

case(Handle,N,pascal):-
         write(Handle,linguagem(pascal)),
         write(Handle,'.'),
         nl(Handle),
         assertz(linguagem(pascal)).
```

```
case(Handle,N,fortran):-
        write(Handle,linguagem(fortran)),
        write(Handle,'.'),
        nl(Handle),
        assertz(linguagem(fortran)).

case(Handle,N,_,N).

pega_numero([],N,Y,X):-
        nl, nl, write('Escreve a tua '),
        write(Y),
        write(' com numeros.'), nl,
        msg(X,Handle).

pega_numero([C|R],N,Y,X):-
        ( number(C),
          N=C, ! ;
          pega_numero(R,N,Y,X) ).
```

2. META-INTERPRETADOR PROLOG

```
execrec(X,Y,R,espiar):-
            ( conta_spy(Z),
              retract(conta_spy(Z)) ; true),
              assertz(conta_spy(5)),
              tscroll(0,(5,20),(20,40) ),
              execute(X,Y,espiar),
              R1 is R+1,
              tmove(R1,0),
              divide(X,X),
              resposta(Y,X).

execrec(X,Y,R,espiar):-
              exec_ok(1).

execrec(X,Y,R,espiar):-
              nl,
              write('Nao possuo informacoes para'),
              write('resolver o teu problema.'),
              nl.

% prototipo para o trace

execute(true,Y,espiar):- !.

execute((P,Q),Y,espiar):- !,
             execute(P,Y,espiar),
             execute(Q,Y,espiar).

execute(P,Y,espiar):-
             functor(P,X,A), !,
             ( system(X/A),
              cont_spy,
              write('CHAMA  '),
              cont_spy,
              P ;
              cont_spy,
              write('CHAMA  '),
              write(P),
             ( clause(P,Q),
              cont_spy,
              write('Ok     '),
              write(P) ;
              cont_spy,write('FALHA  '),
              write(P), fail ),
              execute(Q,Y,espiar) ).
```

```
cont_spy:-   retract(conta_spy(N)),
             N1 is N+1,
             ( N1==20,
              tmove(6,20),
              assertz(conta_spy(5)) ;
              assertz(conta_spy(N1)),
              tmove(N1,20)), !.


%executor de metas e regras


execrec(X,Y,Tam,R):-
              execute(X,Y,Tam,R,_),
              divide(X,X),
              resposta(Y,X).
execrec(X,Y,Tam,R):-
              exec_ok(1).

execrec(X,Y,Tam,R):-
              nl, analise_semant(X,Y,Func,A), nl,
              write('Adiciona factos para '),
              write(Func),
              write(' com '),
              write(A),
              write(' argumentos.'), nl.

execrec(X,Y,Tam,R1):-
              nl,
              write('Nao possuo informacoes para resolver'),
              write('o teu problema.'), nl.

substitui(T1,Lista,Lisai,T):-
              member(T1,Lista,T).

substitui(T1,Lista,Lisai,T):-
              atom_string(T1,X),
              string_term(X,T),
              append(Lista,[T1,T],Lisai),
              ( lisai(L),
               retract(lisai(L)) ; true ),
               assertz(lisai(Lisai)), !.

execute(true,Y,Tam,R,N):- !.

execute((P,Q),Y,Tam,R,N):- !,
              execute(P,Y,Tam,R,Tam1),
              execute(Q,Y,Tam1,R,T).
```

```
execute(P,Y,Tam,R,Tam1):-
                functor(P,X,A), !,
                ( system(X/A),
                 P, nl ; name(X,L),
                 length(L,N),
                 C is Tam+N+1+A,
                 tmove(R,C),
                 write('^'),
                 tam_args(P,A,0,PP),
                 clause(P,Q),
                 Tam1 is Tam+N+A+PP,
                 execute(Q,Y,Tam1,_,Tam1)).

execute(T):-    call(T),
                write(T), nl, !.

analisa(T,Lista,Lisai,L,[]):-
                inic,
                termo(T,Lista,Lisai,L,L,[]),
                er_g(N),
                N=<0.

analisa(T,Lista,Lisai,L,[]):-
                retract(nova_lista(R)),
                desliga,
                inic,
                termo(T,Lista,Lisai,R,R,[]).

analisa(T,Lista,Lisai,L,_):-
                limpa_janela1,inic,
                write('Nao entendo o termo.'),
                traco,
                limpa_janela2.


%analisador sintatico semantico


lexico(X,R,Lista1) :-
                lex1(X,R,Lista1).

lex1(X,R,Lista1):-
                read_string(80,X1),
                tget(R,C),
                testa_vaz(X1), !,
                list_text(Lista1,X1),
                pula_brancos(Lista1,L1),
                reconhece(L1,X), !.

lex1(X,R,Lista1):-
                lexico(X,R1,Z).
```

```
testa_vaz($$):-
              !, fail.

testa_vaz(X1).

reconhece([],[]):- !.

reconhece(L,[H|T]):-
              gettoken(H,L,R),
              pula_brancos(R,R1),
              reconhece(R1,T).

gettoken(token(N,nome))-->
              nome(N), !.

gettoken(token(N,pontuacao))-->
              pontuacao(N), !.

gettoken(token(N,variavel))-->
              variavel(N), !.

gettoken(token(N,anonima)) -->
              sublinha(N).

gettoken(token(N,numero)) -->
              numero(N).

gettoken(token(N,string)) -->
              string(N), !.

pula_brancos -->
              [X],{X=<32},
              pula_brancos.

pula_brancos(L,L).

nome(N)-->    palavra(N) ;
              "[]",
              { N=[] } ;
              simbolo(N) ;
              solo(N) ;
              entre_apostrofos(N) ;
              "{}", { N={} }.

entre_apostrofos(N)-->
              "'",
              caract(T),
              { name(N,[39|T]) }.

caract([H|T]) -->
              "'", { H=39,T=[] }, !.
```

```
caract([H|T]) -->
              [H], caract(T).

palavra(N) -->
              minuscula(H),
              resto_palavra(T),
              ( name(N,[H|T]) ).
resto_palavra([H|T]) -->
              outro_caract(H),
              resto_palavra(T).

resto_palavra([]) -->
              [].

outro_caract(N) -->
              letra(N), !.

outro_caract(N) -->
              digito(N), !.

outro_caract(N) -->
              sublinha(N).

simbolo(N) -->
              car_simb(H),
              resto_simb(T),
              ( name(N,[H|T]) ).

resto_simb([H|T]) -->
              car_simb(H),
              resto_simb(T).

resto_simb([]) -->
              [].

car_simb(X) -->
              [X],
   (pertence(X,[35,36,38,42,43,45,47,58,60,61,62,63,64,92,94,96,126]))

solo(N) -->
              [X],
              ( pertence(X,[33,37,59]), name(N,[X]) ).

numero(N) -->
              int(I), ".",
              int(F),
              ( name(N1,I),
              name(N2,F),
              length(F,L),
              ( L>0,N is  N1+N2/10^L) ).
numero(N) -->
              int(I), ( name(N,I) ).
```

```
int([H|T]) -->
             digito(H), int(T).

int([]) -->  [].

variavel(V) -->
             maiuscula(H),
             resto_palavra(T),
             { name(V,[H|T]) } ;
             sublinha(A),
             resto_palavra([H|T]),
             { name(V,[A,H|T]) }.

string(N) -->
             [34],
             resto_string(T),
             { name(N,[34|T]) }.

resto_string([34]) -->
             [34].

resto_string([H|T]) -->
             [H],
             resto_string(T).

minuscula(L) -->
             [L], {L > 96, L < 123}.

maiuscula(L) -->
             [L], {L > 64, L < 91}.

letra(L) -->
             maiuscula(L), !.

letra(L) -->
             minuscula(L).

digito(L) -->
             [L], { L>47,L<58 }.

sublinha(95) -->
             [95].

pontuacao(',') -->
             [44], !.

pontuacao('(') -->
             [40], !.

pontuacao(')') -->
             [41], !.
```

```
pontuacao('.') -->
            [46], !.

pontuacao('[') -->
            [91], !.

pontuacao(']') -->
            [93], !.

pontuacao('!') -->
            [33], !.

pontuacao('{') -->
            [123], !.

pontuacao('|') -->
            [124], !.

pontuacao('}') -->
            [125].

pertence(H,[H|_]):-  !.

pertence(I,[H|T]):-
            I@>H,
            pertence(I,T).

termo(T,Lista,Lisai,L) -->
            subtermo(T,Lista,Lisai,L,1200), !.

subtermo(T,Lista,Lisai,L,1200) -->
            ( er_g(X), X>0 ), !.

subtermo(T,Lista,Lisai,L,1200) -->
            termo(T,Lista,Lisai,L,1200), !.

subtermo(T,Lista,Lisai,L,999) -->
            termo(T,Lista,Lisai,L,0), !.

termo(T,Lista,Lisai,L,1200)-->
            op(O,P,fx),
            full_stop, !.

termo(T,Lista,Lisai,L,1200)-->
            op(O,P,fy),
            full_stop, !.
```

```
termo(T,Lista,Lisai,L,1200) -->
            op(O,P,fx),
            { retract(operador(K)),
              assert(operador(1)) },
              subtermo(S,Lista,Lisai,L,1200),
            { assert(meta1(S)),
              append([O],[S],R),
              T=..R,
              retract(exemplo(Z)),
              assert(exemplo(meta)) }, !.

termo(T,Lista,Lisai,L,1200) -->
              subtermo(S,Lista,Lisai,L,999),
              full_stop(Lista,Lisai),
              { T=S }, !.

termo(T,Lista,Lisai,L,1200) -->
            subtermo(S,Lista,Lisai,L,999),
            { op(O,P,xfx) ;
              op(O,P,yfx) ;
              op(O,P,xfy) ;
            { (var(Lisai),
              Lisai=Lista ;
            true ) },
            virgula(O,Lista,Lisai,L)),
            { retract(operador(K)),
              assert(operador(1)) },
              subtermo(S1,Lisai,Nova,L,1200),
            { append([O],[S,S1],R),
              T=..R,
              retract(exemplo(Z)),
              assert(exemplo(regra)) }, !.

termo([],Lista,Lisai,L,0,[],[]).

termo(T,Lista,Lisai,L,0) -->
              functor(F),
              { (func(X),
                retract(func(X)) ; true ),
                assertz(func(F)) },
                abre_par(L),
                argumentos(Lista,Lisai,L,Larg,0),
                fecha_par(L),
              { er_g(E),
                ( E>0 ;
                  append([F],
                  Larg,R),
                  T=..R,
                  functor(T,N,A),
                  ve_dic(T,N,A),
                  retract(exemplo(Z)),
                  assert(exemplo(facto)) ) }.
```

```
termo(T,Lista,Lisai,L,0) -->
              constante(T) ;
              variavel(T1,Lista,Lisai),
              ( substitui(T1,Lista,Lisai,T) ) ;
               anonima(T1,Lista,Lisai),
               ( substitui(T1,Lista,Lisai,T) ) ;
              lista(T,Lista,Lisai,L) ;
              string(T) ;
              abre_par(L),
              subtermo(T,Lista,Lisai,L,1200),
              fecha_par(L) ;
              abre_chave,
              subtermo(T,Lista,Lisai,L,1200),
              fecha_chav(L).

op(Simb,P,Ass) -->
              [token(Simb,nome)],
              ( nonvar(Simb),
               current_op(P,Ass,Simb) ).
functor(X) -->
              [token(X,nome)].

argumentos(Lisai,X1,L,_,_,[],[]).

argumentos(Lisai,X1,L,Arg,N) -->
              subtermo(H,Lisai,Z,L,999),
              ( (var(Z),Z=Lisai;true) ),
               resto_arg(Z,X1,L,T,N),
              ( (T=[],Arg=[H];Arg=[H|T]) ).

argumentos(Lisai,X1,L,Arg,N,C,X):-
              er_g(P),
              ( P=0,retract(er_g(P)),
               assert(er_g(1)),
               erro(argumentos,C,L) ; true ).

resto_arg(Z,X1,L,T,N)-->
              ( retract(certo(M)),
               M1 is M+1,
               assert(certo(M1)),
               N1 is N+1,
               retract(cont_arg(J)),
               assert(cont_arg(N1)) ),
               virgula(X,Z,X1,L),
               argumentos(Z,X1,L,T,N1).

resto_arg(Z,X1,L,[],N) -->
              [],
              ( X1=Z, N1 is N+1,
               retract(cont_arg(J)),
               assert(cont_arg(N1)) ).
```

```
lista(T,Lisai,X1,L) -->
                lista_vazia(T) ;
                abre_colch,
                ( retract(certo(Z)),
                  Z1 is Z+1,
                  assert(certo(Z1)) ),
                  exprlista(T,Lisai,X1,L),
                  fecha_colch(L).

exprlista([H|T],Lisai,X1,L) -->
                  subtermo(H,Lisai,X2,L,999),
                  barra_vertical,
                  resto_lista(T,X2,X1,L) ;
                  subtermo(H,Lisai,X2,L,999),
                  ( retract(cont_arg1(J)),
                    J1 is J+1,
                    assert(cont_arg1(J1)) ),
                  virgula(X,Lisai,X2,L),
                  exprlista(T,X2,X1,L).

exprlista([H],Lisai,X1,L) -->
                  subtermo(H,Lisai,X1,L,999),
                  ( retract(cont_arg1(J)),
                    J1 is J+1,
                    assert(cont_arg1(J1)) ).

resto_lista(T,X2,X1,L) -->
                  variavel(T1,X2,X1),
                  ( substitui(T1,X2,X1,T),
                    retract(cont_arg1(J)),
                    J1 is J+1,
                    assertz(cont_arg1(J1)) ) ;
                    anonima(T1,X2,X1),
                  ( substitui(T1,X2,X1,T),
                    retract(cont_arg1(J)),
                    J1 is J+1,
                    assert(cont_arg1(J1)) ).

resto_lista(T,X2,X1,L,A,X):-
                    er_g(N1),
                    ( N1=0,
                      retract(er_g(N1)),
                      assert(er_g(1)),
                      erro(resto_lista,A,L) ; true ).

constante(X) -->
                    [token(X,nome)] ;
                    [token(X,numero)].

variavel(X,Lista,Lisai) -->
                    [token(X,variavel)].
```

```
anonima(X,Lista,Lisai) -->
                [token(_,anonima)],
                ( X='_' ).

string(X) -->   [token(X,string)].

full_stop(Lista,Lisai) -->
                [token('.',pontuacao)].

full_stop([],Lista,Lisai,A,X):-
                er_g(O),
                ( O=0,
                  erro(full_stop,A),
                  reconhece([46],E),
                  full_stop(E,S,Lista,Lisai)
                ; true ).

abre_par(L) --> [token('(',pontuacao)],
                ( retract(par(X)),
                 C is X+1,
                 assert(par(C)) ).

abre_par(L) --> ( par(X),X=0),
                 [token(')',pontuacao)],
                 ( write('faltou o parentese esquerdo e nao'),
                   write('sei onde tu queres '),
                   write('coloca-lo.'), nl,
                   write('fornece novamente o termo:'),
                   traco,
                   limpa_janela2,
                   desliga,
                   lexico(E,R,Lista1),
                   assert(nova_lista(E)),
                   retract(er_g(NP)),
                   assert(er_g(1)),
                   retract(cont_arg(P)),
                   assert(cont_arg(0)) ).

abre_chave -->  [token('(',pontuacao)].

abre_colch -->  [token('[',pontuacao)],
                ( retract(abre_colchete(X)),
                 X1 is X+1,
                 assert(abre_colchete(X1)) ).

barra_vertical -->
                [token('|',pontuacao)].

lista_vazia([]) -->
                [token([],nome)].
```

```
fecha_par(L) -->
                [token(')',pontuacao)].

fecha_par(L) -->
                ( er_g(N1), N1>0 ).

fecha_par(L,A,X):-
                operador(K),
                ( K=0,
                 erro_par1(L,A) ).
fecha_par(L,A,X):-
                operador(K),
                ( K>0,
                 cont_arg(N),
                ( N=0,
                 erro_par1(L,A)) ).

fecha_par(L)-->
               ( operador(K),
                ( K>0,
                 cont_arg(N),
                 ( N>0, nl,
                  limpa_janela1,
                  write('faltou o parentese direito e nao sei'),
                  write('onde tu queres '),
                  write('coloca-lo. Declarastes '),
                  par(X), write(X), nl,
                  write('fornece novamente o termo:'),
                  traco, limpa_janela2, desliga,
                  lexico(E,R,Lista1),
                  assert(nova_lista(E)),
                  retract(er_g(NP)),
                  assert(er_g(1)),
                  retract(cont_arg(P)),
                  assert(cont_arg(0))
               ) ) ).

erro_par1(L,A):-
                retract(er_g(N1)),
                assert(er_g(1)),
                erro(fecha_par,A).

fecha_colch(L) -->
              [token(']',pontuacao)].

fecha_colch(L,A,X):-
              er_g(N1),
              ( N1=0,
              retract(er_g(N1)),
              assert(er_g(1)),
              erro(fecha_colch,A),
              reconhece([93],E),
```

```
                reverse(L,L1),
                reverse(A,A1),
                subtract(L1,A1,L11),
                reverse(L11,L1),
                ( dic(Dic),
                 retract(func(F)),
                 member2(F,Dic,Arity,Z),
                 ( Z\=[],
                  length(Z,NN),
                  ( NN=1,
                   altera(Z,E,A,L1,R)
                  ; write('No dicionario '),
                   write(F),
                   write(' esta definido com os seguintes'),
                   write(' argumentos.'), nl,
                   imp_dic(F,Z),
                   altera(Z,E,A,L1,R) )
                 ; append(L1,E,L2),
                 append(L2,A,R),
                 assert(nova_lista(R)) )
                ; append(L1,E,L2),
                 append(L2,A,R), nl, nl,
                 reconstitui(R), nl,
                 write(R), nl,
                 write('Estas de acordo?'),
                 read(S),
                ( S==sim, !,
                 assert(nova_lista(R) )
                ; erro_lista(fecha_colch,A,E,L1) )
               ) ; true ).

fecha_chav(L) -->
               [token(')',pontuacao)].

fecha_chav(L,A,X):-
               er_g(N1),
               ( N1=0,
                retract(er_g(N1)),
                assert(er_g(1)),
                erro(fecha_chave,A),
                reconhece([125],E),
                reverse(L,L1),
                reverse(A,A1),
                subtract(L1,A1,L11),
                reverse(L11,L1),
                append(L1,E,L2),
                append(L2,A,R),
                assert(nova_lista(R))
               ;true ).

virgula(O,L1,L2,L)-->
               ( er_g(K),K>0 ).
```

```
virgula(']',L1,L2,L,A,X):-
              er_g(N1),
              ( N1=0,certo(J),
               ( J=0,
                erro_lista(abre_colch,A,L,T) ) ).

virgula(']',L1,L2,L,A,X):-
             ( er_g(N1),
              N1=0,
              ( certo(J), J=0,
               ( cont_arg(N), N>0,
                ( dic(Dic),
                 retract(func(F)),
                 member2(F,Dic,Arity,Z),
                 retract(er_g(N1)),
                 assert(er_g(1)),
                ( Z\=[],
                 length(Z,NN),
                 ( NN=1,
                  erro_lista(abre_colch,A,L,T)
                 ; write('No dicionario '),
                  write(F),
                  write(' esta definido com os seguintes'),
                  write(' argumentos.'), nl,
                  imp_dic(F,Z),
                  pega_contexto(Z,A,L,T)
                ) ;
                 erro _lista(confusao,esquerdo,A,L))
             ) ) ) ).

virgula(']',L1,L2,L,A,X):-
              er_g(N1),
              ( N1=0, certo(J),
               ( J=0,
                retract(er_g(N1)),
                assert(er_g(1)),
                ( cont_arg(N), N>0,
                 ( dic(Dic),
                  imp_dic(Dic),
                  erro_lista(confusao,esquerdo,A,L)
                 ;
                  erro_lista(abre_colch,A,L,T) )
               ) ) ).

virgula(O,L1,L2,L)-->
              [token(Simb,pontuacao)],
              { Simb==',',O=',' }.
```

```
virgula(O,L1,L2,L)-->
            [token(Simb,pontuacao)],
            ( Simb=='|',
              O='|',
              retract(certo(N)),
              assert(certo(0)) ),
              virgula(O,L1,L2,L).

virgula(O,L1,L2,L)-->
            [token(Simb,pontuacao)],
            ( ( abre_colchete(X),
                X=0, Simb==']', O=']',
                retract(certo(N)),
                assert(certo(0)) ) ),
            virgula(O,L1,L2,L).

erro_lista(confusao,ED,A,L):-
            limpa_janela1,
            write('faltou o colchete '),
            write(ED),
            write(' e nao sei onde tu queres coloca-lo.'),
            nl, write('forneca novamente o termo:'),
            traco, limpa_janela2,
            desliga,
            lexico(E,R,Lista1),
            assert(nova_lista(E)),
            retract(er_g(N)),
            assert(er_g(1)).

erro_lista(X,A,L,T):-
            ( cont_arg(N2),
            N2>0, msg(X,M),
            limpa_janela1,
            write(M), traco,
            limpa_janela2,
            N1 is N2-1, N is N2+N1+1,
            reverse(L,L1),
            reverse(A,A1),
            subtract(L1,A,L11),
            reverse(L11,L1),
            acha_lug_esq(L1,A,L,N,T)).

% rotina para a correcção automática dos erros sintácticos

erro(X,L):-
            msg(X,M),extrai(L,L1),
            desliga,
            limpa_janela1,
            imprime([M,' --> '|L1]),
            traco,
            limpa_janela2.
```

```
%Rotina  para  a  intervenção  do  aluno  aquando  da  correcção
%orientada dos erros

erro(X,A,L):-
            !, msg(X,M),
            extrai(A,L1),
            desliga,
            limpa_janela1,
            imprime([M,' --> '|L1]),
            traco, limpa_janela2,
            reverse(L,L1),
            reverse(A,A1),
            subtract(L1,A,R1),
            reverse(R1,R),
            pede,
            lexico(E,Lin,Lista1),
            append(R,E,R2),
            assert(nova_lista(R2)).

acha_lug_esq(Le,Ld,L,0,T):-
            reconhece([91],E),
            append(Le,E,Le1), !,
            append(Le1,Ld,R),
            reconstitui(R), nl,
            write('Estas de acordo?'),
            read(S),
            ( S==sim,
             assert(nova_lista(R)),!
            ; !,
            pega_contexto(T,Le,L,_) ).

acha_lug_esq(Le,Ld,L,N,T):-
           last(P,Le),
           subtract(Le,[P],Le1),
           N1 is N-1,
           append([P],Ld,Ld1),
           acha_lug_esq(Le1,Ld1,L,N1,T).

extrai([],[]):- !.

extrai([token(X,_)|R],[X|T]):-
           extrai(R,T).

msg(fecha_par,'faltou parentese direito vou coloca-lo'):- !.

msg(fecha_colch,'faltou colchete direito vou coloca-lo') :- !.

msg(fecha_chave,'faltou chave direita vou coloca-lo') :- !.

msg(resto_lista,'corpo da lista deve ser representado por variavel'):-!

msg(argumentos,'faltaram argumentos'):-!.
```

```
msg(abre_colch,'faltou o colchete esquerdo vou coloca-lo'):-!.

msg(full_stop,'faltou o ponto final do termo vou coloca-lo').

pede:-   imprime('conserta o erro e fornece novamente o'),
         write(' resto do termo: '), nl.

imprime(L):-
         nl, retract('L'(N)),
         apaga(N),
         ( L=[_|_],
          monta_linha(L) ;
          write(L) ),
         ( N>=23,
           assert('L'(13)) ;
           N1 is N+1,
           assert('L'(N1)) ).

desliga:-
         'L'(X), X=:=13 ;
         assert('L'(13)).

liga:-   retract('L'(X)),
         apaga(X),
         ( X>=23,
          assert('L'(13)) ;
          X1 is X+1,
          assert('L'(X1)) ).

monta_linha([]):-
         nl, !.

monta_linha([H|T]):-
        write(H),
        monta_linha(T).

apaga(N):-
        N=:=13 ; true.

limpa_janela1:-
        tscroll(0,(0,0),(3,79)),
        tmove(1,0).

limpa_janela2:-
        tscroll(0,(24,0),(24,79)),
        tmove(24,0).

ve_dic(T,N,A):-
       dic(Dic),
       member1(N,A,Dic).
```

```
ve_dic(T,N,A):-
        dic(Dic),
        not( member(N,Dic)),
        retract(dic(Dic)),
        append(Dic,[N,A],Dic1 ),
        assertz(dic(Dic1)).

ve_dic(T,N,A):-
        ( dic(Dic),
         retract(dic(Dic)),
         tget(R,C),
         tscroll(0,(12,0),(18,79)),
         tmove(12,0),
         write('No dicionario '),
         write(N),
         write(' esta definido com os seguintes argumentos.'),
         tmove(13,0),
         imp_dic(Dic),
         append(Dic,[N,A],Dic1),
         assertz(dic(Dic1)),
         tmove(R,C) ).

ve_dic(T,N,A):-
         append(Dic,[N,A],Dic1),
         assertz(dic(Dic1)).

reconstitui([]).

reconstitui([H|T]):-
         divide_1(H),
         reconstitui(T).

divide_1(token(X,Y)):-
         write(X).

altera(Z,E,A,L1,R):-
         pega_alternativa(Z,E,A,L1,R,T).

pega_alternativa([],E,A,L1,R,T):-
         erro_lista(fecha_colch,A,E,L1).

pega_alternativa([H|T],E,A,L1,R,T):-
         cont_arg1(NE),
         N is NE-1,
         acha_lugar(N,H,L1,A,L2,L3),
         append(L2,E,L4),append(L4,L3,R),
         nl, reconstitui(R), nl,
         write('Estas de acordo?'),
         read(S), nl,
         ( S==sim,
          assert(nova_lista(R)), !
         ; pega_alternativa(T,E,A,L1,R1,_)).
```

```
acha_lugar(N,Arity1,L1,A,L2,L3):-
        ( Arity1 == N,
         ( var(L2),
          L2=L1
         ;true ),
         ( var(L3),
           L3=A
         ; true ) ).

acha_lugar(N,Arity1,L1,A,L2,L3):-
        last(La,L1),
        subtract(L1,[La],Lb),
        last(Laa,Lb),
        subtract(Lb,[Laa],L2),
        append([Laa,La],A,L3), !,
        N2 is N-1,
        acha_lugar(N2,Arity1,L2,L3,L2,L3).

pega_contexto([],A,L,T):-
        !, cerro_lista(confusao,direito,A,L).

pega_contexto([H|T],A,L,T):-
        cont_arg(N2),
        ( N2>=H,
         erro_lista(fecha_colch,A,L,T)
        ; erro_lista(confusao,direito,A,L) ).

imp_dic(F,[]):-
        nl, !.

imp_dic(F,[H|T]):-
        nl, write(F), write('/'), write(H),
        imp_dic(F,T).
```

```
% controle do retrocesso inteligente

execrec(X):-
        execute(X,0),!.
execrec(X):-
        ( fail_data(F),
          retract(fail_data(F) ); true ),
          execute(X,0),
          respostas(X).

execrec(X).

execute(true,Z):- !.

execute((P,Q),I):-
         Inst is I+1,
         execute(P,Inst),
        ( fail_data(X),
         !,fail ;
         execute(Q,Inst) ),!.

execute(P,Inst):-
       ( fact(P),
        clause(P,Q) ;
        functor(P,X,A),!,
       ( system(A/X),
        P, nl ;
        execute_inteligent(P,Inst,A,P),!,
        execute1(P)) ).

execute1(P):-
       ( clause(P,Q),
        nl, write(P),
        assert_fact(P) ;
       ( fail_data(1),
        !,fail ;
        functor(P,X,A),
        arg(A,P,Value),
        find_table(Alternative,A,P,Inst,Return,V),
        instantiat_fail,
        retract_fact(Return),
        !,fail) ).

instantiat_fail:-
        assert(fail_data(1)), !.

retract_fact(Return):-
        retract(fact(Return)),
        assertz(mot_fail(Return)),
        nl, write('O motivo da falha e: '),
        write(Return), nl,
        ve_outros(Return), !.
```

```
assert_fact(P2):-
        assertz(fact(P2)), !.


% procura alternativas para as variaveis instanciadas


find_table(T,0,P,Inst,X,V).

find_table(T,A,P,Inst,X,V):-
        arg(A,P,Value),
        data(V,Inst,T,H,X),
        A1 is A-1,
        ( T=[], !, fail ; true ),
        find_table(Alternative,A1,P,Inst,X,V).


% guarda as possibilidades de retrocesso


execute_inteligent(X,Inst,0,P1).

execute_inteligent(X,Inst,N,P1):-
       arg(N,X,Value),
       ( var(Value),
       ( exec_ok(1),
        retract(data(Value,Inst,List,_,X)),
        true ;
       ( mot_fail(M),retract(data(Value,Inst,List,_,X)),
        true ;
        findall(Value,X,List) ) ),
       instant(X,Inst,N,Value,List) ;
       assertz(data(Value,Inst,[],Value,X))),
       A is N-1,
       execute_inteligent(X,Inst,A,X).

instant(P1,Inst,N,Value,[]):-
       !, fail.

instant(P1,Inst,N,Value,[C|R]):-
       assertz(data(Value,Inst,R,C,P1)),
       arg(N,P1,V),
       V=C, !.

ve_outros(R):-
       functor(R,X,A),
       pesq_arg(A,R).

pesq_arg(0,R).
```

```
pesq_arg(A,R):-
      arg(A,R,X),
      tira_dados(X),
      A1 is A-1,
      pesq_arg(A1,R).

tira_dados(X):-
     data(X,_,[],X,Y),
     retract(fact(Y)),
     retract(data(X,_,[],X,Y)),
     fail.

tira_dados(X).

limpa_dados:-
    retract(data(_,_,_,_,_)),
    fail.

limpa_dados.

limpa_fact:-
    retract(fact(X)), fail.

limpa_fact.

respostas((X)):-
    nl, write('resposta='),
    write(X), nl,
    exam_fact,
    ( mot_fail(M),
     retract(mot_fail(M)) ; true ),
     assert(exec_ok(1)), !, fail.

exam_fact:-
    fact(Y),
    data(_,_,L,_,Y),
    ( L\=[],
     retract(fact(Y)) ; true ), fail.
    exam_fact.
```

2.1 Gerador aotomático de exemplos Prolog

```
% gera os termos Prolog
% T saida do termo
% X indica constante
% X1   "   No.   "
% Y variavel
% Y1 indica No. variaveis
% Z tipo de termo desejado
% As chamadas de (),[]{} sao ficticias

:-assertz(minuscula(97)).
:-assertz(maiuscula(64)).
:-assertz(num(47)).

termo1(T,Lisai,Lisent,X,X1,Y,Y1,facto):-
                    functor(T1),(X1>1,
                    assertz(flag(1));true),
                    argumentos(T5,Lisai,Lisent,T3,X,X1,Y,Y1),
                    retract(list_arg(Ta)),
                    append([T1],Ta,T6),T=..T6,!.

termo1(T,Lisai,Lisent,X,X1,Y,Y1,objectivo):-
                    op(0,1150,fx),
                    termo1(T1,Lisai,Lisent,X,X1,Y,Y1,facto),
                    append([0],[T1],R),T=..R.

termo1(T,Lisai,Lisent,X,X1,Y,Y1,regra):-
                    functor(F),
                    argumentos(T5,Lisai,Lisent,T3,X,X1,Y,Y1),
                    retract(list_arg(Ta)),
                    append([F],Ta,R1),!,Ra=..R1,
                    argumentos(T1,Lisai,Lisent,T2,_,_,Y,Y1),
                    retract(list_arg(Tb)),
                    argumentos(T6,Lisai,Lisent,T4,X,X1,_,_),
                    retract(list_arg(Tc)),
                    append([F],Tc,R2),!,
                    append(R2,Tb,R3),!,
                    Rb=..R3,
                    op(0,1200,xfx),
                    append([0],[Ra,Rb],R4),!,T=..R4.

termo1(T,Lisai,Lisent,X,X1,Y,Y1,atomo,0):-
                    ( X==constante1,X1\=0,
                    constante1(T,X,X1)).

termo1(T,Lisai,Lisent,X,X1,Y,Y1,variavel1,0):-
                   ( Y==variavel1,Y1\=0,
                    variavel1(T,Lisai,Lisent) ) ;
                   ( Y==anonima,Y1\=0,
                    anonima(T) ).
```

```
termo1(T,Lisai,Lisent,X,X1,Y,Y1,lista,0):-
                    lista(T,Lisai,Lisent,X,X1,Y,Y1),
                    retract(lista(P)),
                    T=P.

termo1(T,Lisai,Lisent,X,X1,Y,Y1,string,0):-
                    string(T,X,X1).

termo1(T,Lisai,Lisent,X,X1,Y,Y1,funcao):-
                    functor(F),
                    termo1(T2,Lisai,Lisent,X,X1,Y,Y1,facto),
                    append([F],T2,R),!,T=..R.

termo1(T,Lisai,Lisent,X,X1,Y,Y1,Z):- abre_chave(T1),
                    termo1(T2,Lisai,Lisent,X,X1,Y,Y1,Z),
                    fecha_chave(T3), append(T1,[T2,T3],T).

op(Simb,P,Ass):-    current_op(P,Ass,Simb).


% manutenção do contexto

functor(T):-        retract(verbo(T,O,K,X,Z)),assertz(verbo(T,O,K,X,Z))

functor(T):-        constante1(T,X,X1).

argumentos(T,Lisai,Lisent,Z,X,X1,Y,Y1):-
                    termo1(T1,Lisai,Lisent,X,X1,Y,Y1,_,0),
                    append(Z,[T1],T),!,
                    ( X1=0,Y2 is Y1 - 1,
                     resto_arg(T,Lisent,XX,T,X,X1,Y,Y2) ;
                     X2 is X1 - 1,
                     resto_arg(T,Lisent,XX,T,X,X2,Y,Y1) ).

resto_arg(T,Lisai,Lisent,Z,X,0,Y,0):-
                    ( flag(F),retract(flag(F)),reverse(T,T1),
                     assertz(list_arg(T1)), ! ;
                     assertz(list_arg(T)),! ).

resto_arg(T,Lisai,Lisent,Z,X,X1,Y,Y1):-
                    argumentos(P,Lisai,Lisent,Z,X,X1,Y,Y1).

lista(T,Lisai,Lisent,X,X1,Y,Y1):-
                    ( var(X1),
                     var(Y1),
                    lista_vazia(T),assertz(lista([])) ),!.

lista(T,Lisai,Lisent,X,X1,Y,Y1):-
                    exprlista(T2,Lisai,Lisent,X,X1,Y,Y1).

exprlista(P,Lisai,Lisent,X,0,Y,0):-
                    assert(lista(P)),!.
```

```
exprlista(P,Lisai,Lisent,X,X1,Y,Y1):-
                    ( nonvar(X1),
                    ( var(Y1);Y1<2),
                     termo1(T1,Lisai,Lisent,X,X1,Y,Y1,Z,0),
                     X2 is X1 - 1,
                    ( member_var(T1,Lisent,T11),
                      list_text(Tt,T11),
                      append(P,[T11],H), !
                     ; append(P,[T1],H), ! ),
                     ( nonvar(Y),X2==0,
                       barra_vertical(Ta),
                       resto_lista(T3,Lisent,XX,variavel1),
                       member_var(T3,XX,T4),
                       Y2 is Y1-1,
                       list_text(T2,Ta),
                       list_text(Ta4,T4),
                       append(T2,Ta4,Pa), !,
                       last(H1,H),
                      ( list_text(H2,H1),
                       append(H2,Pa,P1a), ! ;
                       int_text(H1,H2),
                       list_text(H3,H2),
                       append(H3,Pa,P1a), ! ),
                       list_text(P1a,Pa1),
                       atom_string(Pb,Pa1),
                       length(H,N),(N=1,
                       append(M,[Pb],P1), ! ;
                       subtract(H,[H1],H3),
                       append(H3,[Pb],P1))
                       ; P1=H, Y2=Y1 ),
                       exprlista(P1,Lisai,Lisent,X,X2,Y,Y2) ), !.

exprlista(P,Lisai,Lisent,X,X1,Y,Y1):-
                    var(X1),
                    ( Y1\=0,
                     termo1(T1,Lisai,Lisent,X,X1,Y,Y1,Z,0),
                     Y2 is Y1 - 1,
                    ( member_var(T1,Lisent,T11),
                     list_text(Tt,T11),
                     append(P,[T11],H), !
                     ; append(P,[T1],H), ! ),
                    ( Y2=1,
                     barra_vertical(Ta),
                     list_text(T2,Ta),
                     resto_lista(T3,Lisent,XX,Z),
                     subtract(XX,Lisent,Xa),
                     member_var(T3,Xa,Pa1),
                     list_text(Pa2,Pa1),
                     append(T2,Pa2,Pa), !,
                     transf_H(H,H5),
                     list_text(H1,H5),
                     append(H1,Pa,Pp), !,
```

```
                    list_text(Pp,Pa3),
                    atom_string(Pa4,Pa3),
                    append(M,[Pa4],P1), !,
                    exprlista(P1,Lisai,Lisent,_,_,_,0) ;
                    exprlista(H,Lisai,Lisent,_,_,Y,Y2))), !.

lista_vazia([]).

resto_lista(T,Lisai,Lisent,variavel1):-
                    variavel1(T,Lisai,Lisent).

resto_lista(T,Lisai,Lisent,anonima):-
                    anonima(T,Lisai,Lisent).

constante1(T,X,X1):-
                    letra_minuscula(T,X1).

constante1(T,X,X1):-numero(T).

constante1(T,X,X1):-
                    letra_minuscula(T1,X1),
                    numero(T2), append(T1,T2,T).

variavel1(T,Lisai,Lisent):-
                    letra_maiuscula(T,Lisai,Lisent).

string(T,constante1,X):-
                    gera_string(T2,T3,X),concat(T3,T).

gera_string(T2,T3,0):-T3=T2.

gera_string(T2,T3,X):-
                    letra_minuscula(T1,X),
                    append(T2,[T1],Z),!,
                    X1 is X-1,
                    gera_string(Z,T3,X1).

string(T,variavel1,X):-
                    gera_stringv(T2,T3,X),
                    concat(T3,T).

gera_stringv(T2,T3,0):-T3=12.

gera_stringv(T2,T3,X):-
                    letra_maiuscula(T1,Lisai,Lisent),
                    append(T2,[T1],Z),!,
                    X1 is X-1,
                    gera_string(Z,T3,X1).
```

```
% matenção do contexto

letra_minuscula(T,2):-
                    retract(objeto2(T)),
                    assertz(objeto2(T)).

letra_minuscula(T,1):-
                    retract(name(T,S,P,Q,R,O,I)),
                    assertz(name(T,S,P,Q,R,O,I)).

letra_minuscula(T,_):-
                    retract(minuscula(N)),
                    N1 is N+1,
                    ( N1 < 123,
                     name(T,[N]),
                     assert(minuscula(N1))
                    ; assert(minuscula(97)) ).

letra_maiuscula(T,Lista,XX):-
                    retract(maiuscula(N)),
                    N1 is N+1,
                    ( N1 < 91,
                      name(N2,[N1]),
                      substitui(N2,Lista,XX,T),
                      assert(maiuscula(N1))
                    ; assert(maiuscula(64)) ).

numero(T):-
                    retract(num(N)), N1 is N+1,
                    ( N1 < 58,
                      name(T,[N1]),
                      assert(num(N1))
                    ; assert(num(47)) ).

anonima(T,Lisai,Lisent):-
                     name(T,[95]).

virgula(',').
cut('!').
abre_chave('(').
fecha_chave(')').
barra_vertical('|').

substitui(T1,Lista,Lisai,T):-
                    member(T1,Lista,T).

substitui(T1,Lista,Lisai,T):-
                    atom_string(T1,X),string_term(X,T),
                    append(Lista,[T1,T],Lisai),!.

transf_H([L|T],L).
```

2.1 Exemplo de exercícios orientados

```
ola:-
        cls,window('INFORMACOES EXISTENTES',3,3,40,2,70,12),
        c(4,45).

% formatação da tela

window(Cab,Cor1,Cor2,X1,Y1,X2,Y2):-
      W is X2 - X1,W1 is W//2 ,
      name(Cab,C),
      length(C,N),
      A1 is N//2,A is W1-A1,
      A2 is X1 + A ,
      reth(X1,Y1,W,Cor1),
      canto(X2,Y1,191,Cor1),
      Y is Y1 + 1 ,
      retv(X2,Y,Y2,Cor1),
      canto(X1,Y1,218,Cor1),
      tmove(Y,X1),
      retv(X1,Y,Y2,Cor1),
      canto(X1,Y2,192,Cor1),
      X is X1 + 1 ,
      reth(X,Y2,W,Cor1),
      canto(X2,Y2,217,Cor1),
      tmove(Y1,A2),
      writec(Cab,Cor2), !.

 reth(X1,Y1,W,Cor) :-
      tmove(Y1,X1),
      wca(W,196,Cor).

canto(X,Y,Cod,Cor) :-
      tmove(Y,X),
      wca(1,Cod,Cor).

retv(X,Y2,Y2,_).

retv(X,Y,Y2,Cor) :-
      tmove(Y,X),
      wca(1,179,Cor),
      Y1 is Y + 1,
      retv(X,Y1,Y2,Cor).

cls(X1,Y1,X2,Y2) :-
      tscroll(0,(Y1,X1),(Y2,X2)),
      tmove(0,0).
```

```
writec(T,Cor) :- name(T,L),
                 length(L,N),
                 wa(N,Cor),
                 write(T).

c(N,Z) :-
         ( N<10 , tmove(N,Z) , write('ola') ,
          N1 is N + 1 , c(N1,Z)
         ; tmove(11,43),
          write('PRESS ANY KEY TO CONTINUE'),
          get0_noecho(R) ,
          cls(41,3,69,11),
          N1 is 4 ,
          c(N1,Z) ).


% variações previstas para o programa 1


programa(1,[0,Ps1,1,1,A,_,
            1,Ps2,2,1,A,_,
            1,Ps2,2,2,B,_,
            2,Ps3,2,1,A,_,
            2,Ps3,2,2,B1,_,
            3,Ps1,1,1,B,_,
            4,Ps1,1,1,B1,_,
            data,Ps1,1,1,maria,_,
            data,Ps1,1,1,jose,_] ).

programa(1,[0,Ps1,1,1,A,_,
            1,Ps2,3,1,A,_,
            1,Ps2,3,2,B,_,
            1,Ps2,3,3,C,_,
            2,Ps1,1,1,B,_,
            3,Ps1,1,1,C,_,
            data,Ps1,1,1,maria,_,
            data,Ps1,1,1,jose,_] ).


%variações previstas para o progra 2


programa(2,[0,Ps1,2,1,A,_,
            0,Ps1,2,2,B,_,
            1,Ps2,2,1,C,_,
            1,Ps2,2,2,A,_,
            2,Ps3,2,1,C,_,
            2,Ps3,2,2,B,_,
            data,Ps2,2,1,maria,_,
            data,Ps2,2,2,joao,_,
            data,Ps3,2,1,maria,_,
            data,Ps3,2,2,pedro,_] ).
```

```
programa(2,[0,Ps1,2,1,A,_,
            0,Ps1,2,2,B,_,
            1,Ps2,3,1,C,_,
            1,Ps2,3,2,A,_,
            1,Ps2,3,3,B,_,
            data,Ps3,3,1,maria,_,
            data,Ps3,3,2,joao,_,
            data,Ps3,3,3,pedro,_] ).


% Controle do modulo de exercicios
%   N representa o numero do exercicio

le(N,L,T,I):-
        ( var(T),
          tmove(17,15),
          lexico(Ltoken),
          tmove(25,80),
          !,
          ve_fim(N,Ltoken,L,I)
          ;
          ( T == [token(fim,nome),token(.,pontuacao)],
            ve_fim(N,T,L,I)
            ;
            prep_analise(T,Ltoken),
            true
          )
        ),
        assert(exemplo(nada)),
        analisa(Term,Lista,Lisai,Ltoken,[]),
        !,
        retract(exemplo(_)),
        ( nonvar(Lisai),
          retract(lisai(Lvar))
          ;
          true
        ),
        substitui_L(Term,[Term1],Cab),
        ( nonvar(Term1),
          encontra_var(Lvar,_,Lvar1),
          processa(Term1,L,Lvar1,Lvar2,I,I1,T1),
          processa(Cab,L,Lisai,Lvar3,I1,I2,T1)
          ;
          ( nonvar(Term),
            processa(Term,L,Lvar,Lvar1,I,I2,T1)
            ;
            I2 = I
          )
        ),
        !,
        janela_opcoes(N,L,T1,I2).
```

```
ve_fim(N,Ltoken,L,I):-
        ( Ltoken == [token(fim,nome),
          token(.,pontuacao)],
          tmove(0,0),
          analisa_programa(N,L,J),
          ( nonvar(J),
            retract(clausula1(Claus)),
            prep_analise(Claus,Lt),
            assertz(exemplo(nada)),
            analisa(Term,V,LL,Lt,[]),
            !,
            retract(exemplo(_)),
            testa_execucao(regra,Term,R,Ltoken)

% ligação com o meta-interpretador

            ;
            janela_prossegue,
            janela_principal(N,L,A,I)
          )
          ;
          ( Ltoken == [token(menu,nome),token(.,pontuacao)],
            janela_prossegue,
            janela_principal(N,L,A,I)
            ;
            true
          )
        ).

prep_analise(Term,Ltoken):-
        string_term(St,Term),
        list_text(L1,St),
        delete1(44,L1,L2),
        delete1(44,L2,L3),
        delete1(91,L3,L4),
        delete1(93,L4,L5),
        append(L5,[46],L6),
        reconhece(L6,Ltoken).

%   Analise da regra

substitui_L(Term1,Term2,Cab):-
        Term1=..L,
        current_op(1200,xfx,Op),
        member_op(Op,L,Term2,Cab), !.

substitui_L(_,_,_).

member_op(Op,[Op,Cab|Cor],Cor,Cab):-
            !.

member_op(_,_,_,_).
```

```
%   Preenchimento do enquadramento

processa((P,Q),L,Lvar,Lvarf,I,I2,Term1):-
        !,
        processa(P,L,Lvar,Lvarf,I,I1,Term1),
        processa(Q,L,Lvarf,Lvar2,I1,I2,Term1).

processa(Term,L,Lvar,Lvarf,I,I2,Term1):-
        I2 is I+1,
        functor(Term,Nome,Arg),
        AA is Arg,
        ( nonvar(Lvar),
          repoem_var(Term,Termf,1,Arg,Lvar,Lvarf),
          decom_args(Nome,Termf,AA,Arg,L,Term1,I2)
          ;
          decom_args(Nome,Term,AA,Arg,L,Term1,I2)
        ).

repoem_var(Term,Termf,Posi,0,Lvar,Lvarx):-
        Termf = Term,
        Lvarx = Lvar,
        !.

repoem_var(Term,Termf,Posi,Arg,Lvar,Lvarx):-
        encontra_var(Lvar,V,Lvarf),
        argrep(Term,Posi,V,Term1),
        Posi1 is Posi + 1,
        Arg1 is Arg - 1,
        !,
        repoem_var(Term1,Termf,Posi1,Arg1,Lvarf,Lvarx).

encontra_var([V,_|Lvarf],V,Lvarf).

decom_args(Nome,Term,AA,0,L,Term1,I).

decom_args(Nome,Term,AA,Arg,L,Term1,I):-
        arg(Arg,Term,X),
        Arg1 is Arg-1,
        poem_lista(Nome,AA,Arg,L,X,Y,Term,Term1,I),
        ( nonvar(Y),
          ( var(Term1),
            ( clausula(Y,Term,_),
              true
              ;
              assert(clausula(Y,Term,I)),
              janela_clausulas
            )
            ;
            ( clausula(Y,Term1,_),
              true
              ;
              assert(clausula(Y,Term1,I)),
```

```
                janela_clausulas
              )
            )
            ;
            true
          ),
          !,
          decom_args(Nome,Term,AA,Arg1,L,Term1,I).

poem_lista(Nome,AA,Arg,L,X,Y,Term,Term1,I):-
          (  member_data(Nome,AA,Arg,X,Y,I,L)
             ;
             ( member(Nome,AA,Arg,X,Y,I,L)
               ;
               ( abolish(lista/1)
                 ;
                 true
               ),
               divide_lista(L),
               analisa_erro(Term,Nome,AA,Arg,X,Term1)
             )
          ).

member_data(Nome,AA,Arg,X,Y,I,[data,Nome,AA,Arg,X,E|T]):-
          Y=data,
          ( var(E),
            E=I
            ;
            true
          ).

member_data(Nome,AA,Arg,X,Y,I,[_,_,_,_,_,_|T]):-
          member_data(Nome,AA,Arg,X,Y,I,T).

member(Nome,AA,Arg,X,Y,I,[0,Nome1,_,_,_,E|T]):-
          retract(numero_info(X1)),
          assert(numero_info(X1)),
          I < X1,
          !,
          member(Nome,AA,Arg,X,Y,I,T).

member(Nome,AA,Arg,X,Y,I,[Y,Nome,AA,Arg,X,E|T]):-
          var(E),
          E=I      ;
          true.

member(Nome,AA,Arg,X,Y,I,[_,_,_,_,_,_|T]):-
          member(Nome,AA,Arg,X,Y,I,T).
```

```
%Controle dos termos escritos de acordo com o esperado

analisa_erro(Term,Nome,AA,Arg,X,Term1):-
        append([],[Nome,AA,Arg,X],P1),!,
        ( lex(P,P1)
          ;
          janela_dados,
          tmove(18,15),
          write_cor('Este termo nao faz parte')
          write(' da definicao deste problema.',31),
          janela_prossegue
        ),
        !,
        divide2(P1,P,Term),
        divide1(P,P1,Term,Term1),
        janela_prossegue.


divide1([],P1,Term,Term1):-
        Term1=Term.

divide1([H|T],P1,Term,Term1):-
        ( disjoint(H,P1),
          !,
          tmove(18,15),
          write_cor('Devias ter escrito ',31),
          write_variavel(H,31,18,35),
          Term=..L,
          corrige_term(Term,H,P1,L,Term1),
          tmove(19,15),
          write_cor('O correcto e: ',31),
          write_variavel(Term1,31,19,32),
          divide1(T,P,Term1,Term1)
          ;
          divide1(T,P1,Term,Term1)
        ).


divide2([],P,Term).

divide2([H|T],P,Term):-
        ( disjoint(H,P),
          janela_dados,
          tmove(17,15),
          write_cor('Escreveste ',31),
          write_variavel(H,31,17,27)
          ;
          true
        ),
        !,
        divide2(T,P,Term).
```

```
corrige_term(Term,H,[M,N,O,P],[M1,N1,O1,P1],Term1):-
        M1\=M,
        troca_H(M,[M1,N1,O1,P1],L1),
        Term1=..L1,
        !.

corrige_term(Term,H,[M,N,O,P],_,Term1):-
        argrep(Term,O,H,Term1).

troca_H(H,[J|T],L1):-
        append([H],T,L1),
        !.

lex(P,P1):-
        lista(P),
        search_similar_word(P1,P),
        lista(P).

%       Comando AJUDA

ajuda_programa(N,L,J):-
        janela_ajuda,
        analisa_programa(N,L,J).

analisa_programa(N,L,J):-
        abolish(lista/1),
        divide_lista(L),
        !,
        vazios(N,L,J).

divide_lista([]):-
        !.

divide_lista([A,B,C,D,E,_|T]):-
        append([],[B,C,D,E],U),
        !,
        assertz(lista(U)),
        divide_lista(T).

vazios(N,L,J):-
        lista(L1),
        member(Z,L1),
        var(Z), nl,
        examina_L1(L1).

vazios(N,L,J):-
        imprime_clausula(N,0,J).

vazios(_,_,_):-
        !.
```

```
imprime_clausula(N,K,J):-
        clausula(K,X,I),
        !,
        case_clausula(N,K,X,I,J),
        K1 is K+1,
        imprime_clausula(N,K1,J).

imprime_clausula(_,_,_):-
        !.

examina_L1([B,C,D,E]):-
        nl,
        write_ecran('Falta a clausula com nome',23),
        ( var(B),
          write_ecran('_ _',23)
          ;
          write_ecran(B,23)
        ),
        write_ecran(' com ',23),
        write_ecran(C,23),
        write_ecran(' argumento(s) e ',23),
        tget(Lin,Col),
        Lin1 is Lin + 1,
        tmove(Lin1,0),
        write_ecran('com o ',23),
        write_ecran(D,23),
        write_ecran(' argumento igual a',23),
        ( var(E),
          write_ecran('_ _',23)
          ;
          write_ecran(E,23)
        ),
        !, fail.


%       Pragmatica da regra


case_clausula(N,A,X,I,1):-
        A==0,
        !,
        janela_dados,
        tmove(17,15),
        write_cor('Montagem da clausula  ',31),
        write_variavel(X,31,18,15),
        tmove(18,19),
        write_cor(':',31),
        write_cor('-',31),
        d_numero_claus,
        current_op(1200,xfx,O),
        append([X],[O],R),
        assertz(clausula1(R)).
```

```
case_clausula(N,_,X,_,_):-
        tget(Li,Co),
        Co1 is Co + 1,
        write_variavel(X,31,Li,Co1),
        retract(clausula1(R)),
        append(R,[X],R1),
        assert(clausula1(R1)),
        d_numero_claus,
        ( numero_claus(0),
          write_cor('.',31),
          tmove(19,15),
          write_cor('O teu programa esta correcto.',31)
          ;
          write_cor(',',31)
        ).

d_numero_claus:-
        retract(numero_claus(X)),
        X1 is X-1,
        assert(numero_claus(X1)).


%       Comando ESQUECER


limpa_clausulas(Nome,N_arg,Lista,Lista3):-
        clausula(_,Claus,I),
        functor(Claus,Nome1,N_arg1),
        ( N_arg == N_arg1,
          Nome == Nome1,
          retract(clausula(_,Claus,I)),
          functor_sub(Lista,Nome1,I,Lista1),
          argumentos_sub(Lista1,Claus,N_arg,I,Lista2)
        ),
        limpa_clausulas(Nome,N_arg,Lista2,Lista3).

limpa_clausulas(_,_,Lista,Lista2):-
        Lista2 = Lista.

divide_Lex([token(Nome,nome),token(/,_),token(N_arg,numero),
            token(.,pontuacao)],Nome,N_arg,_,_,_).
divide_Lex(_,_,_,N,L,L2):-
        !,
        resposta_esquecer(N,predicado,L,L2).

instante_N(Nome,N_arg):-
        clausula(_,A,_),
        functor(A,N1,Y1) ,
        ( Nome == N1,
          N_arg == Y1
        ).
```

```
instante_N(Nome,N_arg):-
        janela_dados,
        tmove(18,15),
        write_cor('Nao encontro o predicado ',31),
        write_cor(Nome,31),
        write_cor(' com  ',31),
        write_cor(N_arg,31),
        write_cor(' argumento(s).',31),
        janela_prossegue.

functor_sub(Lista,Nome,I,Lista1):-
        localiza_func(Lista,Nome,W,I,Lista1),
        !.

argumentos_sub(Lista1,Claus,0,I,Lista2):-
        Lista2 = Lista1.

argumentos_sub(Lista1,Claus,N_arg,I,Lista2):-
        arg(N_arg,Claus,Arg),
        localiza_arg(Lista1,Arg,W,I,Listai),
        N_arg1 is N_arg - 1,
        !,
        argumentos_sub(Listai,Claus,N_arg1,I,Lista2).

localiza_func([],Nome,_,_,[]):-
        ( esquecidos(X),
          retract(esquecidos(X)),
          append(X,[Nome],Y),
          !
          ;
          Y=[Nome]
        ),
        assert(esquecidos(Y)),
        !.

localiza_func([Ord,Nome,N_args,N_arg,Arg,I|L],Nome,W,I,
        [Ord,W,N_args,N_arg,Arg,I1|Ln]):-    !,
        localiza_func(L,Nome,W,I,Ln).

localiza_func([Ord,Nome,N_args,N_arg,Arg,E|L],Nome,W,I,
                        [Ord,W,N_args,N_arg,Arg,E|Ln]):-    !,
        localiza_func(L,Nome,W,I,Ln).

localiza_func([Ord,Nome1,N_args,N_arg,Arg,E|L],Nome,W,I,
                      [Ord,Nome1,N_args,N_arg,Arg,E|Ln]):-
        !,
        localiza_func(L,Nome,W,I,Ln).
```

```
localiza_arg([],Arg,_,_,[]):-
        ( esquecidos(X),
          retract(esquecidos(X)),
          append(X,[Arg],Y),
          !
          ;
          Y=[Arg]
        ),
        assert(esquecidos(Y)),
        !.

localiza_arg([data,Nome,N_args,N_arg,Arg,I|L],Arg,W,I,
                    [data,Nome,N_args,N_arg,Arg,I|Ln]):-
        !,
        localiza_arg(L,Arg,W,I,Ln).

localiza_arg([Ord,Nome,N_args,N_arg,Arg,I|L],
        Arg,W,I,[Ord,Nome,N_args,N_arg,W,I|Ln]):-
        !,
        localiza_arg(L,Arg,W,I,Ln).

localiza_arg([Ord,Nome,N_args,N_arg,Arg1,E|L],Arg,W,I,
                    [Ord,Nome,N_args,N_arg,Arg1,E|Ln]):-
        !,
        localiza_arg(L,Arg,W,I,Ln).

compara_fun(X,K):-
        ( member1(X,K),
          janela_dados,
          tmove(17,15),
          write_cor('Podera haver conflicto com ',31),
          tmove(18,15),
          write_cor('o simbolo predicativo ',31),
          write_variavel(X,31,18,37)
          ;
          true  ),
        !.

compara_args(_,0,_):-
        !.

compara_args(P,N,K):-
        arg(N,P,X),
        ( member1(X,K),
          janela_dados,
          tmove(18,15),
          write_cor('Podera haver conflicto com o argumento ',31),
          write_variavel(X,31,18,60)
          ;
          true  ),
        N1 is N-1,
        !, compara_args(P,N1,K).
```

```
%  Menus do modulo de exercicios

tutor(N):-
        programa(N,L),
        janela_boneco,
        janela_principal(N,L,T,0).

janela_boneco :-
        tmove(23,0),
        wa(79,16),
        cls(0,0,79,23),
        window('',4,4,janela_dupla,31,40,6,65,14),
        tmove(8,45),
        write_cor('TUTOR INTELIGENTE',31),
        tmove(12,43),
        write_cor('EXERCICIOS ORIENTADOS',31),
        tmove(6,17),
        wa(5,112),
        cls(17,4,22,6),
        tmove(6,19),
        wca(1,92,112),
        tmove(6,20),
        wca(1,47,112),
        tmove(5,18),
        wca(1,48,112),
        tmove(5,21),
        wca(1,48,112),
        tmove(8,19),
        wa(1,128),
        cls(19,7,20,8),
        tmove(9,12),
        wa(15,128),
        cls(12,8,27,9),
        tmove(13,16),
        wa(7,128),
        cls(16,10,23,13),
        tmove(13,12),
        wa(2,128),
        cls(12,10,14,13),
        tmove(13,25),
        wa(2,128),
        cls(25,10,27,13),
        tmove(13,11),
        wa(3,64),
        cls(11,13,14,13),
        tmove(13,25),
        wa(3,64),
        cls(25,13,28,13),
        tmove(19,16),
        wa(2,208),
        cls(16,14,18,19),
        tmove(19,21),
```

```
        wa(2,208),
        cls(21,14,23,19),
        tmove(19,14),
        wa(4,64),
        cls(14,19,18,19),
        tmove(19,21),
        wa(4,64),
        cls(21,19,25,19),
        janela_prossegue.

janela_prossegue  :-
        tmove(24,0),
        wa(80,15),
        write_cor('                                To proceed  ENTER',15),
        tmove(25,80),
        get0_noecho(_),
        cls(0,24,79,24).

janela_principal(N,L,T,I):-
        tmove(23,0),
        wa(1,112),
        cls(0,0,79,23),
        janela_exercicio(N),
        janela_clausulas,
        janela_opcoes(N,L,T,I).

janela_exercicio(N):-
        window(' EXERCICE ',94,29,janela_simples,31,1,1,43,11),
        exercicio(N).

exercicio(1):-
        tmove(3,2),
        write_cor('John is a person .',30),
        tmove(4,2),
        write_cor('Mary is a person .',30),
        tmove(6,2),
        write_cor('Every body have a father and a mother ',30),
        tmove(7,2),
        write_cor('that are also persons .',30),
        tmove(9,2),
        write_cor('Write in Prolog this Information .',30),
        assert(numero_claus(5)),
        assert(numero_info(7)).

exercicio(2):-
        tmove(3,2),
        write_cor('O Joao e o tio do Pedro ,',30),
        tmove(4,2),
        write_cor('se a Maria for irma do Joao e ',30),
        tmove(5,2),
        write_cor('se a Maria for a mae do Pedro .',30),
        tmove(7,2),
```

```
        write_cor('Transcreva estas frases para Prolog.',30),
        assert(numero_claus(3)),
        assert(numero_info(5)).

janela_clausulas:-
        cls(50,1,78,11),
        window('CLAUSES DECLARATION',95,81,janela_dupla,31,50,1,78,11)
        !,
        coloca_clausulas(A,Y,3,55).

coloca_clausulas(A,Y,Lin,Col):-
        retract(clausula(X1,X,X2)),
        assertz(clausula(X1,X,X2)),
        !,
        imprime(X,Y,Lin,Col).

coloca_clausulas(A,Y,Lin,Col).

imprime(X,Y,Lin,Col):-
        ( var(Y) ,
          !,
          ( Lin < 10 ,
            tmove(Lin,Col),
            write(X),
            write('.'),
            tmove(Lin,Col),
            wa(13,31),
            Lin1 is Lin+1 ,
            Y = X ,
            coloca_clausulas(X,Y,Lin1,Col)
            ;
            tmove(13,50),
            write_cor('     < ENTER >  to proceed ',31),
            tmove(25,80),
            get0_noecho(R) ,
            cls(54,3,77,10),
            tmove(13,50),
            wa(29,51),
            cls(50,13,78,13),
            Lin1 is 3 ,
            imprime(X,Y,Lin1,Col)
          )
        ).

imprime(X,Y,Lin,Col):-
        ( X == Y,
          !
          ;
          ( Lin < 10 ,
            tmove(Lin,Col),
            write(X) ,
            write('.'),
```

```
            tmove(Lin,Col),
            wa(13,31),
            Lin1 is Lin+1 ,
            coloca_clausulas(X,Y,Lin1,Col)
            ;
            tmove(13,50),
            write_cor('     < ENTER > to proceed ',31),
            tmove(25,80),
            get0_noecho(_) ,
            cls(54,3,77,10),
            tmove(13,50),
            wa(29,51),
            cls(50,13,78,13),
            Lin1 is 3 ,
            imprime(X,Y,Lin1,Col)
          )
        ).

janela_opcoes(N,L,T,I):-
        cls(0,14,79,23),
        janela_aviso,
        window(' OPTION ',161,26,janela_simples,26,25,14,50,21),
        menu(6,0,29,20,26,26,['End','Exercice Execution',
        'Clear Predicate','Resolution Help','Data Introdution'],A),
        resposta_opcoes(N,A,L,L2,I).

janela_aviso :-
        cls(0,24,0,24),
        tmove(24,15),
        write_cor('Use ( ',14),
        tmove(24,21),
        put(24),
        tmove(24,21),
        wa(1,10),
        tmove(24,22),
        put(25),
        tmove(24,22),
        wa(1,10),
        tmove(24,23),
        write_cor('  ) to select and to proceed ',14),
        write_cor(' ENTER ',10),
        tmove(25,80).

resposta_opcoes(N,dados,L,_,I):-
        !,
        janela_dados,
        le(N,L,_,I).

resposta_opcoes(N,ajuda,L,_,I):-
        ajuda_programa(N,L,_),
        janela_prossegue,
        janela_principal(N,L,A,I).
```

```
resposta_opcoes(N,esquecer,L,L2,I):-
        janela_esquecer(N,L,L2,I),
        !,
        janela_principal(N,L2,A,I).

resposta_opcoes(N,executar,L,_,I):-
        !,
        le(N,L,[token(fim,nome),token(.,pontuacao)],I).

resposta_opcoes(N,fim,L,_,I).

janela_ajuda :-
        tmove(24,0),
        wa(1,128),
        cls(0,24,79,24),
        window(' HELP ',28,28,janela_simples,31,0,0,79,23).

janela_dados :-
        tmove(24,0),
        wa(79,128),
        cls(15,24,79,24),
        cls(2,14,77,21),
        window('',31,31,janela_dupla,31,10,15,70,21).

janela_esquecer(N,L,L2,I):-
        cls(0,14,79,21),
        window(' REMOTION ',161,26,janela_simples,26,30,15,50,21),
        menu(5,0,34,19,26,26,['Predicate','All'],R),
        !,
        resposta_esquecer(N,R,L,L2).

resposta_esquecer(N,tudo,_,L1):-
        retract(programa(N,L1)),
        abolish(clausula/3).

resposta_esquecer(N,predicado,L,L2):-
        janela_dados,
        tmove(17,15),
        write_cor('Write predicate(name)/arguments(number) .',31),
        tmove(19,15),
        lexico(E),
        divide_Lex(E,Nome,N_arg,N,L,L2),
        instante_N(Nome,N_arg),
        limpa_clausulas(Nome,N_arg,L,L2).

janela_analizador:-
        tmove(24,0),
        wa(1,128),
        cls(0,24,79,24),
        window('  ANALYSYS  ',31,31,janela_dupla,31,1,14,78,22).
```

```
% Predicados de uso geral

apaga(_,[],[]).

apaga(A,[A|T],T).

apaga(A,[B|T],[B|T1]):-
        apaga(A,T,T1).

concat([],L,L).

concat([A|T],L,[A|T1]):-
        concat(T,L,T1).

uniao([],L,L):-
        !.

uniao([A|T],L,[A|T1]):-
        not(pertence(A,L)),
        !,
        uniao(T,L,T1).

uniao([_|T],L,L1):-
        uniao(T,L,L1).

pertence(A,[A|_]).

pertence(A,[_|T]):-
        pertence(A,T).

diferenca_conj([],_,[]).

diferenca_conj([A|T],L,L1):-
        pertence(A,L),
        !,
        diferenca_conj(T,L,L1).

diferenca_conj([A|T],L,[A|T1]):-
        !,
        diferenca_conj(T,L,T1).

ultimo([A],A).

ultimo([_|T],A):-
        !,
        ultimo(T,A).
```

2.2 Formação das sentencas exemplos da Língua Natural

```
sentenca:-
          cor(32),
          write('Agora, com as informacoes existentes,'),
          write(' vou fazer algumas'),
          write(' frases afirmativas.'),
          cor(37), nl, nl,
          sentence_out(afirmativa), !.

sentenca_int:-
          cor(32),
          write('Em Portugues as perguntas sao'),
          write(' representadas por '),
          write('frases interrogativas como:'),
          cor(37), nl,
          testa_sexoh,
          testa_sexom,
          sentence_out(interrogativa), !.

sentence_out(afirmativa):-
          name(X,S,_,_,_,_,_),
          verbo(Y,O,K,_,_),
          objeto2(Z),
          append([S],[X,O,K,Z,'.'],F),
          imprime_frase(F), !.

sentence_out(afirmativa).

sentence_out(interrogativa):-
          name(X,S,_,_,_,_,_),
          verbo(Y,O,K,_,_),
          objeto2(Z),
          append([S],[X,O,K,Z,'?'],F),
          imprime_frase(F), !.

sentence_out(interrogativa).

sentence_out(interrogativa,X,N):-
          name(X,S,_,_,_,_,_),
          verbo(N,O,K,_,P),
          ( K==[],
           append([o],
           [que,O,S,X,'?'],F)
          ; append([K],
           [que,O,S,X,'?'],F)),
           imprime_frase(F), !.

sentence_out(interrogativa,_,_).
```

```
sentence_out(interrogativa_var):-
        name(X,S,_,_,_,_,_),
        verbo(Y,O,K,_,_),
        ( K==[],
         append([o],
         [que,O,S,X,'?'],F)
        ; append([K],
         [que,O,S,X,'?'],F) ),
         imprime_frase(F), !.

sentence_out(interrogativa_var).

sentence_out(interrogativa,X1,X,Y):-
        name(X1,S,_,_,_,_,_),
        verbo(X,O,K,_,P),
        append([S],[X1,O,K,Y,'?'],F),
        imprime_frase(F), !.

sentence_out(interrogativa,_,_,_).

sentence_out(interrogativa2):-
        verbo(Y,O,K,_,P),casek(K,K1),
        append([quem],[O,K1,'?'],F),
        imprime_frase(F).

sentence_out(interrogativa2).

sentence_out(conj,X,Y,Z):-
        name(X,S,_,_,_,_,_),
        verbo(Y,O,K,_,P),
        append([S],[X,O,P,quem,Y,K,Z,'?'],F),
        imprime_frase(F), !.

sentence_out(conj,_,_,_).

sentence_out(conj):-
        name(X,S,_,_,_,_,_),
        verbo(Y,O,K,_,P),
        objeto2(Z),
        append([S],[X,O,P,[],quem,O,K,Z,'?'],F),
        imprime_frase(F), !.

sentence_out(conj).

sentence_out(conj,X1,Y,X,Z):-
        name(X1,S,_,_,_,_,_),
        name(X,S1,_,_,_,_,_),
        verbo(Y,O,K,_,P),
        append([S],[X1,O,P,S1,X,se,S1,X,Y,K,Z,'?'],F), !.
```

```
sentence_out(conj2):-
        retract(name(X1,S,U,P1,P,[X1|D],D)),
        assertz(name(X1,S,U,P1,P,[X1|D],D)),
        verbo(Y,O,K,_,P),
        retract(name(X,S1,U,P1,P,[X|D],D)),
        assertz(name(X,S1,U,P1,P,[X|D],D)),
        objeto2(Z),
        append([S],[X1,O,P,S1,X,se,S1,X,Y,K,Z,'?'],F),
        ( imprime_frase(F)
        ; !, true ).

sentence_out(conj2).

casek(de,K1):-
        K1='do que', !.

casek(o,K1):-
        K1='quem', !.

casek(no,K1):-
        K1='a onde', !.

casek([],K1):-
        K1='o que',!.

meta_sent_int(T):-
        verbo(X,_,_,_,_),
        name(Y,S,_,_,_,_,_),
        objeto2(Z),
        append([X],[Y,Z],R),
        T=..R.

meta_sent_int_var(T):-
        verbo(X,_,_,_,_),
        name(Y,S,_,_,_,_,_),
        letra_m(N),
        capitals(N,W),
        append([X],[Y,W],R),
        T=..R.

monta_sent_int_pro:-
        meta_sent_int(T),
        imprime_pro(T).

monta_sent_int_pro.

monta_sent_var_pro:-
        meta_sent_int_var(T),
        imprime_pro(T).

monta_sent_var_pro.
```

```
monta_sent_var2_pro:-
         meta_sent_int_var2(T),
         imprime_pro(T).

monta_sent_var2_pro.

imprime_fact(T):-
        write(T),
        write('.'),
        nl, !, fail.

imprime_pro(T):-
        write(' ?- '),
        write(T),
        write('.'),
        nl, !, fail.

testa_sexoh:-
        homem(X),
        arruma_h(X).

testa_sexoh.

arruma_h(X):-
        retract(name(X,_,U,P1,P,[X|D],D)),
        assert(name(X,o,U,P1,P,[X|D],D)),
        !,fail.

testa_sexom:-
        mulher(X),
        arruma_m(X).

testa_sexom.

arruma_m(X):-
        retract(name(X,_,U,P1,P,[X|D],D)),
        assert(name(X,a,U,P1,P,[X|D],D)),
        !, fail.

imprime_frase([[]|_]):-
        !, fail.

imprime_frase([N|T]):-
        capitals(N,N1),
        write(N1),
        write(' '),
        imp_cont(T),
        !, fail.

imp_cont([N|[]]):-
        write(N), nl, !.
```

```
imp_cont([[]|T]):-
        imp_cont(T).

imp_cont([N|Ns]):-
        ( name(N,_,_,_,_,_,_),
          capitals(N,N1),write(N1)
        ; write(N) ),
        write(' '),
        imp_cont(Ns).
```

```
% conteudo para ensino de factos atomos e variaveis
% para os níveis 1 e 2

monta_exemplos(constante,0,YN):-
          traco, msg0(constante,12), traco,
          monta_exemplos(constante,1,sim), !.

monta_exemplos(constante,1,sim):-
          termo1(X,Lisai,Lisent,constante,1,_,_,atomo,0),
          write('um valor fixo como a palavra: '),
          cor(35), write(X), cor(37), nl,
          imprime_ex(constante,1,sim), !.

monta_exemplos(constante,2,sim):-
          name(X,_,_,_,_,_,_),
          write('um valor fixo como o nome: '),
          cor(33), write(X), cor(37), nl,
          imprime_ex(constante,2,sim), !.

monta_exemplos(constante,3,sim):-
          write('letras minusculas como a letra : '),
          cor(31),
          sting(T,constante,1),
          write(T), cor(37), nl,
          imprime_ex(constante,3,sim), !.

monta_exemplos(constante,4,sim):-
          numero(N),
          write('um valor fixo como o numero: '),
          cor(36), write(N), cor(37), nl,
          imprime_ex(constante,4,sim), !.

monta_exemplos(constante,5,sim):-
          exemplos(constante), !.

monta_exemplos(constante,1,nao):-
          traco, msg0(constante,12), traco,
          imprime_ex(constante,1,nao), !.

monta_exemplos(constante,N,nao):-
          monta_exemplos(constante,5,sim).

monta_exemplos(variavel,0,YN):-
          traco, msg0(variavel,11), traco,
          write('letras maiusculas como: '),
          cor(35),
          string(T,variavel1,1),
          write(T),
```

```
          cor(37), nl,
          imprime_ex(variavel,1,sim), !.


monta_exemplos(variavel,2,sim):-
          numero(N),
          write('um conjunto de letra(s) e digito(s) '), nl,
          string(T,variavel1,1),
          write(' sempre iniciando com letra maiuscula como: '),
          cor(36),
          write(T),
          write(N), cor(37), nl,
          imprime_ex(variavel,2,sim), !.

monta_exemplos(variavel,3,sim):-
          string(X,variavel1,1),
          string(T,constante1,2),
          write('um conjuto de letras, sempre iniciando '),
          write('por letra maiuscula, como: '),
          cor(33), write(X),
          write(T), cor(37), nl,
          imprime_ex(variavel,3,sim), !.

monta_exemplos(variavel,4,sim):-
          exemplos(variavel).

monta_exemplos(variavel,1,nao):-
          monta_exemplos(variavel,2,sim), !.

monta_exemplos(variavel,2,nao):-
          traco, msg0(variavel,11), traco,
          cor(31), write('X'), cor(37),
          write(' ou '), cor(31),
          numero(N),
          write('X'), write(N), cor(37), nl,
          imprime_ex(variavel,2,sim), !.

monta_exemplos(variavel,N,nao):-
          monta_exemplos(variavel,4,sim).

monta_exemplos(variavel,3,nao):-
          traco, msg1(variavel,15), cor(35),
          string(T,variavel,1),
          string(T1,constante1,2),
          write(T), write(T1), cor(37),
          string(Y,variavel1,3),
          write(' e '),
          cor(35), write(Y), cor(37),
          traco,
          imprime_ex(variavel,3,nao), !.
```

```
monta_exemplos(facto,0,sim):-
          sentenca,
          traco, msg0(facto,9), traco,
          termo1(T,Lisai,Lisent,constante1,2,_,_,facto),
          tget(Lin,Col),
          tmove(Lin,20),
          write(T), write('.'),
          cor(37),
          tscroll(0,(0,0),(6,79)),
          tmove(23,0),
          testa_execucao(facto,T,_,_),
          imprime_ex(facto,0,sim), !.

monta_exemplos(facto,1,sim):-
          exemplos(facto),
          !, cls, tmove(24,0),
          ( ponto(P,Z,Z1,Z2),
          retract(ponto(P,Z,Z1,Z2)) ; true ),
          assertz(ponto(facto,meta,0,sim)),
          menu2.

monta_exemplos(facto,0,nao):-
          monta_exemplos(constante,1,sim), !,
          monta_exemplos(facto,0,sim), !.

monta_exemplos(facto,N,nao):-
          sentenca,
          facto_res,
          monta_exemplos(facto,1,sim), !.


% conteudo para o ensino dos objectivos para os niveis 1 e 2


monta_exemplos(meta,0,sim):-
          meta_sent_int1(T),
          functor(T,X,N),
          arg(1,T,X1),
          arg(2,T,Y),
          sentence_out(interrogativa,X1,X,Y),
          traco,
          msg0(licao,3),
          traco, cor(36), tget(Lin,Col),
          tranf_imp(T,N,T1),
          imp_termo(T1),
          cor(37),
          perg_exec(B),
          ( B==sim,
           tscroll(0,(23,0),(24,79)),
           assert(meta1(T)),
           Lin1 is Lin-2,
           testa_execucao(meta,_,Lin1,_) ;
```

```
            ( B==menu,menu(meta,0,sim) ; nl, true ) ),
           imprime_ex(meta,0,sim), !.

monta_exemplos(meta,1,sim):-
           info_ex,
           exemplos(meta), !,
           monta_exemplos(meta,2,sim), !.

monta_exemplos(meta,2,sim):-
           meta_sent_int_var1(T),
           functor(T,N,A),
           arg(1,T,X1),
           sentence_out(interrogativa,X1,N),
           traco, msg0(licao,4), traco,
           msg0(licao,5),
           traco, cor(32), tget(Lin,Col),
           tranf_imp(T,A,T1),
           imp_termo(T1),
           cor(37),
           perg_exec(B),
           Lin1 is Lin+1,
           ( B==sim,
            assert(meta1(T)),
            tscroll(0,(5,0),(13,79)),
            tscroll(0,(Lin1,0),(24,79)),
            testa_execucao(meta,_,Lin,_) ;
            ( B==menu,
              menu(meta,2,sim) ;
              nl, true ) ),
           imprime_ex(meta,2,sim), !.

monta_exemplos(meta,3,sim):-
           exemplos(meta),
           !, monta_exemplos(meta,4,sim), !.

monta_exemplos(meta,4,sim):-
           traco,
           write('Vejamos agora a pergunta com 2 variaveis:'),
           nl,
           sentence_out(interrogativa2), !,
           traco,
           meta_sent_int_var2(T),
           cor(31),
           tranf_imp(T,2,T1),
           imp_termo(T1),cor(37),
           tget(Lin,Col),
           perg_exec(B),
           ( B==sim,
            Lin1 is Lin+1,
            tscroll(0,(Lin1,0),(24,79)),
            assert(meta1(T)),
            testa_execucao(meta,_,Lin,_) ;
```

```
          ( B==menu,
            menu(meta,4,sim) ;
            nl, true ) ),
          imprime_ex(meta,4,sim), !.

monta_exemplos(meta,5,sim):-
          exemplos(meta), !, cls,
          ( idade_n(X),
           X>12,
           traco,
           msg0(licao,6),
           traco,
           monta_exemplos(meta,6,sim)
          ; adeus(fim,_,_) ), !.

monta_exemplos(meta,6,sim):-
          termo1(T,Lisai,Lisent,constante1,1,variavel1,1,facto),
          objeto2(Z1),
          functor(T,N,X),
          arg(1,T,X1),
          sentence_out(conj,X1,N,Z1),
          traco,cor(36),
          write('?-'),
          write(T),
          write(','),
          arg(2,T,W),
          append([N],[W,Z1],R),
          T1=..R,
          write(T1),
          write('.'),
          tget(Lin,Col),
          cor(37),
          perg_exec(B),
          ( B==sim,
           append([','],[T,T1],R3),
           R4=..R3,
           assert(meta1(R4)),
           tscroll(0,(0,0),(5,79)),
           Lin1 is Lin+1,
           tscroll(0,(Lin1,0),(24,79)),
           tmove(Lin,0),
           testa_execucao(meta,_,Lin,_) ;
           ( B==menu,
            menu(meta,6,sim) ;
            nl, true ) ),
          !, imprime_ex(meta,6,sim).

monta_exemplos(meta,7,sim):-
          cor(33),
          verbo(X,_,_,_,_),
          retract(name(Y,S,U,P1,P,[Y|D],D)),
          assertz(name(Y,S,U,P1,P,[Y|D],D)),
```

```
            retract(name(X2,S1,U,P1,P,[X2|D],D)),
            assertz(name(X2,S1,U,P1,P,[X2|D],D)),
            objeto2(Z),
            traco,
            sentence_out(conj,Y,X,X2,Z), nl,
            append([X],[Y,X2],R),
            T=..R,
            write('?-'),
            write(T),
            write(','),
            append([X],[X2,Z],R1),
            T1=..R1,
            write(T1),
            write('.'),
            tget(Lin,Col),
            cor(37),
            perg_exec(B),
            ( B==sim,
             append([','],[T,T1],R3),
             R4=..R3,
             assert(meta1(R4)),
             Lin1 is Lin-2,
             tscroll(0,(Lin1,0),(24,79)),
             tmove(Lin1,0),
             testa_execucao(meta,_,Lin1,_) ;
            ( B==menu,
             menu(meta,7,sim) ; nl, true ) ),
             imprime_ex(meta,7,sim), !.

monta_exemplos(meta,8,sim):-
            info_ex,
            exemplos(meta),
            !, cls, tmove(24,0),
            ( ponto(P,Z,Z1,Z2),
             retract(ponto(P,Z,Z1,Z2)) ; true ),
             assertz(ponto(meta,regra,0,sim)),
             menu2.

monta_exemplos(meta,0,nao):-
            traco,
            meta_sent_int1(T),
            cor(35), tget(Lin,Col),
            write(T), write('.'), cor(37), nl,
            msg1(constante,14),
            !, nl, tget(Lin,Col),
            cor(33), write(T),
            write('.'), cor(37),
            tmove(Lin,Col), wa(25,112), nl,
            write('e respondera'),
            cor(32), write(' nao '),
            cor(37),
            write('caso este facto nao exista.'),
```

```
            traco,
            sentenca_int,
            monta_exemplos(meta,1,sim), !.

monta_exemplos(meta,2,nao):-
            traco,
            write('A variavel deve ser colocada na '),
            write('posicao onde queremos obter as'),
            nl, write('respostas possiveis.'),
            termo1(T,Lisai,Lisent,constante1,1,variavel1,1,facto),
            tranf_imp(T,1,T1),
            imp_termo(T1),
            get(L,C),
            assert(meta1(T)),
            testa_execucao(meta,_,L,_), nl,
            write('Veja oque a simples troca '),
            write('de posicao da variavel provoca.'), traco,
            cor(35),
            functor(T,N1,X3),
            arg(1,T,X4),
            arg(2,T,X5),
            append([N1],[X4,X3],R),
            T1=..R,
            nl, tranf_imp(T1,1,TT),
            imp_termo(TT),
            assert(meta1(T1)),
            tget(L1,C1),
            testa_execucao(meta,_,L1,_), nl,
            cor(37),
            monta_exemplos(variavel,0,YN), !,
            monta_exemplos(meta,3,sim), !.

monta_exemplos(meta,6,nao):-
            monta_exemplos(meta,6,nao), !.

monta_exemplos(meta,7,nao):-
            write('Ve as seguintes sentencas escritas em '), nl,
            write('Portugues:'), traco, cor(35),
            write('A Maria joga com Joao.'), nl,
            write('A Maria joga com  Paulo.'), nl,
            write('A Maria joga tenis.'), nl,
            write('O Paulo joga tenis.'), nl,
            write('O Joao joga tenis.'),
            cor(37), traco,
            write('Ve as informacoes que'),
            write(' ja estao escritas em Prolog.'),
            info_ex, nl,
            write('Escreve, em Prolog, as que estao faltando.'),
            exemplos(facto),
            !, traco,
            write('Ve a seguinte frase em Portugues:'), nl, nl,
            termo1(T,Lisai,Lisent,constante1,2,_,_,facto),
```

```
functor(T,X,A),
arg(1,T,Z),
arg(2,T,X2),
sentence_out(conj,Z,X,X2,Z), nl,
write('Nesta frase temos uma condicao '),
cor(31),
write('SE'), cor(37),
write('esta condicao tambem deve ser representada'),
nl,
write('em Prolog'), nl, nl,
write('Ve a mesma frase escrita em Prolog:'), nl, nl,
write('?-'),
write(T),
cor(31),
write(','),
cor(37),
append([X],[X2,Z],R1),
T1=..R1,write(T1),
write('.'),
tget(Lin,Col),
perg_exec(B),
( B==sim,
 append([','],[T,T1],R3),
 R4=..R3,
 assert(meta1(R4)),
 Lin1 is Lin+1,
 tscroll(0,(Lin1,0),(24,79)),
 tmove(Lin,0),
 testa_execucao(meta,_,Lin,_)
;
 ( B==menu,
  menu(meta,7,nao) ; nl, true ) ),
traco,
write('Ve agora com o uso de variaveis:'),
nl, cor(35),
termo1(T,Lisai,Lisent,constante1,1,variavel1,1,facto),
objeto2(W),
functor(T,X,A),
arg(1,T,A1),
sentence_out(conj,A1,X,W),
nl, cor(37),
nl, write('?-'),
write(T),
write(','),
arg(2,T,Z1),
append([X],[W,Z1],R),
T1=..R,
write(T1), write('.'),
tget(Lin2,Col2),
Lin3 is Lin2+1,
tscroll(0,(Lin3,0),(24,79)),
append([','],[T,T1],R3),
```

```
            R4=..R3,
            perg_exec(B),
            ( B==sim,
             assert(meta1(R4)),
             tmove(Lin2,0),
             testa_execucao(meta,_,Lin2,_) ;
            ( B==menu,
             menu(meta,7,nao) ; nl, true ) ),
             imprime_ex(meta,7,nao), !.

monta_exemplos(meta,N,nao):-
            cls, tmove(24,0),
            write('Resumindo a comparacao das sentencas'),
            write(' interrogativas em Portugues'),
            write(' e em PROLOG.'), nl,
            cor(33),
            sentence_out(interrogativa),
            cor(31), traco,
            monta_sent_int_pro,
            cor(37), traco, tget(R,C),
            respostas,tmove(R,C),
            segue, cls,
            cor(36),
            sentence_out(interrogativa_var),
            cor(35), traco, cor(33),
            monta_sent_var_pro, traco,
            cor(37), tget(R1,C1),
            respostas,
            tmove(R1,C1),
            segue, cls, cor(35),
            sentence_out(interrogativa2),
            traco, cor(32),
            monta_sent_var2_pro,
            cor(37),
            tget(R2,C2),
            respostas,
            tmove(R2,C2), segue,
            !, tmove(24,0),
            monta_exemplos(meta,5,sim), !.

meta_sent_int1(T):-
            termo1(T,Lisai,Lisent,constante1,2,_,_,facto).

meta_sent_int_var1(T):-
            termo1(T,Lisai,Lisent,constante1,1,variavel1,1,facto).

meta_sent_int_var2(T):-
            termo1(T,Lisai,Lisent,_,_,variavel1,2,facto).

facto_res:-
            termo1(T,Lisai,Lisent,constante1,2,_,_,facto),
            imprime_fact(T).
```

```
facto_res.

monta_sent_int_pro:-
          meta_sent_int1(T),
          imprime_pro(T).

monta_sent_int_pro.

monta_sent_var_pro:-
          meta_sent_int_var1(T),
          imprime_pro(T).

monta_sent_var_pro.

monta_sent_var2_pro:-
          meta_sent_int_var2(T),
          imprime_pro(T).

monta_sent_var2_pro.

imprime_fact(T):-
          write(T),
          write('.'),
          nl, !, fail.

imprime_pro(T):-
          write('?-'),
          write(T),
          write('.'),
          nl, !, fail.

imp_termo(T):-
          write('?-'),
          write(T),
          write('.').

tranf_imp(T,0,Z):-
          Z=T, !.

tranf_imp(T,X,Z):-
          arg(X,T,Value),
          ( var(Value),
           letra_m(N),
           !, argrep(T,X,N,T1)
          ;
           T1=T ),
          X1 is X-1,
          tranf_imp(T1,X1,Z).
```

```
letra_m(N2):-
          retract(maiuscula(N)),
          N1 is N+1,
          ( N1 < 91,
          name(N2,[N1]),
          assert(maiuscula(N1) ) ;
          assert(maiuscula(64)) ).


%         lições para as listas nos níveis 1 e 2


monta_sent_lista:-
          name(X,S,_,_,_,_,_),
          verbo(Y,_,_,_,_),
          retract(objeto2(Z1)),
          assertz(objeto2(Z1)),
          retract(objeto2(Z2)),
          assertz(objeto2(Z2)),
          imprime_sent_lista(S,X,Y,Z1,Z2),
          nl, cor(32),
          write('Podemos desmembrar '),
          write('esta frase em 2 factos como:'),
          cor(31), nl,
          monta_fatos(X,Y,Z1,Z2), nl,
          cor(32),
          write('Ou     entao,     representar    parte    destas
          informacoes'),
          write(' em forma de lista.'),
          nl, cor(35),
          imprime_lista(X,Y,Z1,Z2),
          cor(37), !.

imprime_lista(X,Y,Z1,Z2):-
          append([],[Z1,Z2],R1),
          write(Y), write('('),
          write(X),
          write(','),
          write(R1),
          write(')'),
          write('.'), nl.

imprime_sent_lista(S,X,Y,Z1,Z2):-
          capitals(X,X1),
          ( var(S), true ;
           capitals(S,S4),
           write(S4),write(' ') ),
          write(X1), write(' '),
          verbo(Y,O,_,_,_),
          write(O), write(' '),
          write(Z1),
          write(' e '),
```

```
            write(Z2),
            write('.'), nl.

monta_fatos(X,Y,Z1,Z2):-
            append([Y],[X,Z1],R1),
            T1=..R1,
            write(T1), write('.'), nl,
            append([Y],[X,Z2],R2),
            T2=..R2,
            write(T2), write('.'), nl.
monta_exemplos(lista,0,sim):-
            traco,msg0(licao,7), traco,
            monta_sent_lista,
            traco,
            msg0(lista,10), traco,
            imprime_ex(lista,0,sim), !.

monta_exemplos(lista,0,nao):-
            monta_exemplos(lista,1,nao), !.

monta_exemplos(lista,1,sim):-
            cls, tmove(24,0),
            monta_sent_lista, traco,
            msg0(lista,10), traco,
            write('Escreve as informacoes existentes em forma de'),
            write(' lista.'), nl,
            info_ex,
            monta_exemplos(lista,6,sim), !.

monta_exemplos(lista,1,nao):-
            write('Vejamos com maior detalhe:'), nl,
            termo1(R,Lisai,Lisent,constante1,1,_,_,lista,0),
            write('uma lista com um elemento como: '),
            cor(31), write(R),
            cor(37), nl,
            imprime_ex(lista,1,nao), !.

monta_exemplos(lista,2,sim):-
            termo1(R,Lisai,Lisent,constante1,2,_,_,lista,0),
            write('uma lista com mais de um elemento como: '),
            cor(32), write(R), cor(37),
            nl, imprime_ex(lista,2,sim), !.

monta_exemplos(lista,2,nao):-
            traco, msg0(lista,10), nl,
            write('Neste caso temos uma lista com dois '), nl,
            write('valores constantes.'),
            traco, nl,
            imprime_ex(lista,2,nao), !.
```

```
monta_exemplos(lista,3,sim):-
          write('uma lista com cabeca constante e, '),
          write('naturalmente, o corpo variavel: '),
          cor(33),
          termo1(T,Lisai,Lisent,constante1,1,variavel1,1,lista,0),
          write(T), cor(37), nl,
          imprime_ex(lista,3,sim), !.

monta_exemplos(lista,3,nao):-
          traco,msg0(licao,17), nl,
          write('O corpo da lista deve ser representado '),
          write('sempre por uma variavel.'), nl,
          imprime_ex(lista,3,nao), !.

monta_exemplos(lista,4,sim):-
          write('uma lista com cabeca e corpo'),
          write(' variavel como: '), cor(35),
          termo1(R,Lisai,Lisent,_,_,variavel1,2,lista,0),
          write(R), cor(37), nl,
          imprime_ex(lista,4,sim), !.

monta_exemplos(lista,4,nao):-
          traco, msg1(lista,13), traco,
          imprime_ex(lista,4,sim), !.

monta_exemplos(lista,5,_):-
          nl, write('resumindo os exemplos:'), nl,
          termo1(T0,Lisai,Lisent,_,_,_,_,lista,0),
          imp_listas(T0),
          write('  vazia'),
          termo1(T1,Lisai,Lisent,constante1,3,_,_,lista,0),
          imp_listas(T1),
          write('  com atomos'),
          termo1(T2,Lisai,Lisent,constante1,1,variavel1,1,lista,0),
          imp_listas(T2),
          write('  com atomo(s), separador e variavel'),
          termo1(T3,Lisai,Lisent,_,_,variavel1,2,lista,0),
          imp_listas(T3),
          write('   com variavel, separador e variavel'),
          nl, imprime_ex(lista,5,sim), !.

monta_exemplos(lista,6,sim):-
          exemplos(lista),
          !, cls,
          exercicios,!,menu(fim,_,_).

monta_exemplos(lista,N,nao):-
          monta_exemplos(lista,6,sim), !.

imp_listas(T):-
          write(T),
          nl, !.
```

```
% Mensagens de reforço instrucional


msg0(licao,0):-
          linha(0,X),
          imprime_f(0,X), !.

linha(0,[*,na,linguagem,de,programacao,+,prolog,temos,regras]).
linha(0,[a,que,devemos,obedecer,para,escrever,programas,',',tal]).
linha(0,[como,em,qualquer,outra,linguagem,.,*,se]).
linha(0,[o,nosso,programa,estiver,bem,escrito]).
linha(0,[sera,entendido,' ',e,' ',nossos,desejos,serao]).
linha(0,[executados,.]).
linha(0,['fim']).

msg0(licao,3):-
          linha(3,X),
          imprime_f(3,X), !.

linha(3,[*,agora,que,ja,sabes,como,se,armazenam,factos]).
linha(3,['(',informacoes,')',na,memoria,do,'  ',computador,vamos]).
linha(3,[aprender,como,estas,informacoes,podem]).
linha(3,[ser,obtidas,.,*,na,linguagem,+,prolog,',']).
linha(3,[como,em,+,portugues,uma,consulta,e,representada]).
linha(3,[por,uma,pergunta,do,tipo,:]).
linha(3,['fim']).

msg0(licao,4):-
         linha(4,X),
         imprime_f(4,X), !.

linha(4,[*,em,+,portugues,'(',de,que,')',representa,uma,forma]).
linha(4,[de,pergunta,nao,restritiva,isto,e,',',tanto,pode]).
linha(4,[receber,uma,so,resposta,comovarias,.,*,assim,',']).
linha(4,[por,exemplo,:,'(',de,que,e,o,que,')',representam]).
linha(4,[algo,variavel,pois,podemos,receber,como]).
linha(4,[resposta,varios,objectos,.]).
linha(4,[*,em,+,prolog,existem,tambem,formas,de,representar]).
linha(4,[perguntas,nao,restritivas,usando]).
linha(4,[variaveis,.,as,variaveis,',',ao,contrario,das]).
linha(4,[constantes,',',sao,representadas,por,palavras]).
linha(4,[iniciadas,por,letra,maiuscula,ou]).
linha(4,[apenas,uma,letra,maiuscula,.]).
linha(4,['fim']).

msg0(licao,5):-
          linha(5,X),
          imprime_f(5,X), !.

linha(5,[*,vejamos,como,perguntas,que,admitem,varias]).
linha(5,[respostas,sao,escritas,em,+,prolog,.]).
linha(5,['fim']).
```

```
msg0(licao,6):-
        linha(6,X),
        imprime_f(6,X), !.

linha(6,[*,vejamos,ainda,outra,forma,de,se,escrever]).
linha(6,[perguntas,em,+,prolog,:,as,perguntas,relacionadas,.]).
linha(6,[*,vejamos,frases,escritas,em,+,portugues,e,as]).
linha(6,[frases,correspondentes,escritas,em,+,prolog,.]).
linha(6,['fim']).

msg0(licao,7):-
        linha(7,X),
        imprime_f(7,X), !.

linha(7,[*,ate,o,momento,aprendemos,a,representar,factos]).
linha(7,[em,+,formulas,+,atomicas,.,no,entanto,existe,uma]).
linha(7,[segunda,maneira,de,representar,informacoes,:]).
linha(7,[a,+,lista,.,*,vejamos,a,seguinte,frase,escrita]).
linha(7,[em,+,portugues,:]).
linha(7,['fim']).

msg0(meta,8):-
        linha(8,X),
        imprime_f(8,X), !.

linha(8,[*,em,+,prolog,as,frases,interrogativas,',']).
linha(8,[isto,e,',',as,que,esperam,uma,resposta,do]).
linha(8,[interpretador,sao,chamadas,de,+,metas,.]).
linha(8,['fim']).

msg0(facto,9):-
        linha(9,X),
        imprime_f(9,X), !.

linha(9,[*,em,+,prolog,as,frases,afirmativas,constituem]).
linha(9,[os,+,factos,.]).
linha(9,['fim']).

msg0(lista,10):-
        linha(10,X),
        imprime_f(10,X), !.

linha(10,[*,o,conteudo,duma,lista,e,delimitado,pelos]).
linha(10,[simbolos,:,'[',']',.]).
linha(10,['fim']).

msg0(variavel,11):-
        linha(11,X),
        imprime_f(11,X), !.

linha(11,[*,uma,variavel,pode,ser,qualquer,conjunto,de]).
linha(11,[caracteres,iniciado,por,letra,maiuscula,.]).
```

```
linha(11,['fim']).

msg0(constante,12):-
        linha(12,X),
        imprime_f(12,X), !.

linha(12,[*,um,valor,constante,e,qualquer,conjunto,de]).
linha(12,[caracteres,em,letras,minusculas]).
linha(12,[ou,um,valor,numerico,.]).
linha(12,['fim']).

msg1(lista,13):-
        linha(13,X),
        imprime_f(13,X), !.

linha(13,[*,o,termo,'[',']',representa,a,lista,vazia,.,*,o]).
linha(13,[termo,'[',+,x,'|',+,y,']',representa,a,lista,onde,a]).
linha(13,[cabeca,e,+,x,e,o,corpo,da,lista,e,+,y,.,*,o]).
linha(13,[casamento,do,termo,'[',+,a,',',+,b,'|',+,x,']']).
linha(13,[com,a,lista,'[',1,',',2,',',3,',',4,']',resulta,em,:]).
linha(13,[*,a,variavel,'A',recebe,o,valor,1]).
linha(13,[*,a,variavel,'B',recebe,o,valor,2,.]).
linha(13,[*,a,variavel,'X',recebe,a,lista,'[',3,',',4,']',.]).
linha(13,[*,o,casamento,do,termo,'[','X','|','Y',']']).
linha(13,[com,a,lista,'[',a,']',resulta,em,:]).
linha(13,[*,x,'=',a,.]).
linha(13,[*,y,'=','[',']',.]).
linha(13,['fim']).

msg1(constante,14):-
        linha(14,X),
        imprime_f(14,X), !.

linha(14,[*,uma,pergunta,com,valores,constantes,como,a,anterior]).
linha(14,[pode,somemnte,ser,confirmada,ou]).
linha(14,[negada,.,isto,'e',',o,interpretador,apresenta,a,resposta]).
linha(14,[se,possuir,na,memo,ria,o,facto,:]).
linha(14,['fim']).

msg1(variavel,15):-
        linha(15,X),
        imprime_f(15,X), !.

linha(15,[*,uma,variavel,como,'X',significa,que,'X']).
linha(15,[pode,receber,varios,valores,.,isto,e,',',X ']).
linha(15,[pode,receber,os,valores,que,satisfazem,a]).
linha(15,[condicao,estabelecida,.,*,por,exemplo,:]).
linha(15,[humano,'(','X',')',.,*,neste,caso,'X']).
linha(15,[pode,receber,os,seguintes,valores,:]).
linha(15,['fim']).
```

```
msg0(inicio,16):-
        linha(16,X),
        imprime_f(16,X), !.

linha(16,[*,em,+,prolog,os,nomes,'(',proprios,ou,comuns,')']).
linha(16,[constituem,os,+,o,+,b,+,j,+,e,+,c,+,t,+,o,+,s,da]).
linha(16,[linguagem,e,os,verbos,constituem]).
linha(16,[as,+,r,+,e,+,l,+,a,+,c,+,o,+,e,+,s,que,ligam,os]).
linha(16,[objetos,.,*,frasespodem,ser,escritas,na,linguagem]).
linha(16,[+,prolog,.,*,claro,que,a,sintaxe,da,linguagem]).
linha(16,[+,prolog,e,diferentedo,+,portugues,.]).
linha(16,[*,vejamos,como,se,representam,algumas,frases]).
linha(16,[em,+,prolog,.]).
linha(16,['fim']).

msg0(licao,17):-
        linha(17,X),
        imprime_f(17,X), !.

linha(17,[*,a,lista,+,prolog,divide-se,em,duas,partes,:]).
linha(17,[a,cabeca,e,o,corpo,.]).
linha(17,[*,em,+,prolog,',',isto,e,representado,da]).
linha(17,[seguinte,maneira,:]).
linha(17,['[',+,cabeca,'|',+,corpo,']',.]).
linha(17,['fim']).

msg0(exercicio,18):-
        linha(18,X),
        imprime_f(18,X), !.

linha(18,[*,um,elemento,faz,parte,de,uma,lista,se,ele,for]).
linha(18,[a,cabeca,da,lista,ou,se,ele,estiver,presente]).
linha(18,[no,corpo,da,lista,.]).
linha(18,['fim']).

msg0(regra,19):-
        linha(19,X),
        imprime_f(19,X), !.

linha(19,[*,as,regras,sao,usadas,para,representar,definicoes,.]).
linha(19,[*,uma,regra,divide-se,em,:]).
linha(19,[cabeca,separador,e,corpo,.,*,por,exemplo,',']).
linha(19,[*,cabeca,'=',' ']).
linha(19,[relacao(elemento1,elemento2,elementoN),'      ']).
linha(19,[*,separador,'=','":-"','                      ']).
linha(19,['                    ']).
linha(19,[*,corpo,'=','    ',relacao(elemento1,elemento2,
      elementoN),.]).
linha(19,['fim']).
```

```
msg0(prolog,20):-
        linha(20,X),
        imprime_f(20,X), !.

linha(20,[*,prolog,linguagem,de,programacao,em,logica,.]).
linha(20,['fim']).

msg0(sintaxe,21):-
        linha(21,X),
        imprime_f(21,X), !.

linha(21,[*,parte,da,ciencia,da,linguagem,que,estuda,a,maneira]).
linha(21,[de,ligar,as,'palavras,',a,fim,'de,',por,essa,'forma,',se]).
linha(21,[exprimir,com,maior,precisao,o,pensamento,logico,.]).
linha(21,['fim']).

msg0(X,X):-
        linha(X,Y),
        imprime_f(X,Y),!.
```

5. ANALISE DO SUB-CONJUNTO DA LINGUA NATURAL

```
:-
        op(500,fx,:),
        op(400,xfy,&),
        op(400,xfy,->),
        op(400,xfy,<-),
        create_world(def),
        create_world(leitor),
        create_world(ecran),
        create_world(hist),
        code_world(_,leitor),
        [-le],
        code_world(_,ecran),
        [-ecran],
        code_world(_,def),
        [-rees1],[-verb],[-ttt],[-trace],
        [-predcom],[-gram],[-categ],[-lexico],[-correc],
        inicia.

/*
   As regras de tipo (1.1) que formam o dicionario do sistema.
*/


dic_det(masc,sing) -->
        [o].

dic_det(masc,plu) -->
        [os].

dic_det(fem,sing) -->
        [a].

dic_det(fem,plu) -->
        [as].

dic_det(masc,sing) -->
        [um].

dic_det(masc,plu) -->
        [uns].

dic_det(fem,sing) -->
        [uma].

dic_det(fem,plu) -->
        [umas].
```

```
dic_det(_,_) -->
        tenta_corrigir(dic_det).

dic_det(_,_) -->
        { notifica_vazio(det) },
        [].


dic_prep -->
        [por].

dic_prep -->
        tenta_corrigir(dic_prep).

dic_pron_int -->
        [que].

dic_pron_int -->
        tenta_corrigir(dic_pron_int).

dic_nome_com(masc,sing) -->
        [carro].

dic_nome_com(_,_) -->
        tenta_corrigir(dic_nome_com).

dic_nome_prop(masc,sing) -->
        [paulo].

dic_nome_prop(_,_) -->
        tenta_corrigir(dic_nome_prop).

dic_adj(masc,sing) -->
        [colorido].

dic_adj(fem,sing) -->
        [colorida].

dic_adj(masc,plu) -->
        [coloridos].

dic_adj(fem,plu) -->
        [coloridas].

dic_adj(masc,sing) -->
        [alto].

dic_adj(fem,sing) -->
        [alta].

dic_adj(_,sing) -->
        [enorme].
```

```
dic_adj(masc,sing) -->
        [pequeno].

dic_adj(fem,sing) -->
        [pequena].

dic_adj(fem,plu) -->
        [pequenas].

dic_adj(_,_) -->
        tenta_corrigir(dic_adj).

dic_verb_aux(sing) -->
        [e].

dic_verb_aux(plu) -->
        [sao].

dic_verb_aux(_) -->
        tenta_corrigir(dic_verb_aux).

dic_verb_trans(sing) -->
        [explique].

dic_verb_trans(_) -->
        tenta_corrigir(dic_verb_trans).

dic_desconhecido -->
        [@@@@@@].

dic_desconhecido -->
        tenta_corrigir(dic_desconhecido).


/*
        TABELAS COM INFORMACAO DE CONTROLO
*/


list_vazio([det]).

classe_fechada([det , pron_int , prep , verb_aux]).

classe_aberta([nome_com , nome_prop ,
               adj , verb_trans , desconhecido]).
```

```
/*
   As regras (1.2) correspondentes as diferentes categorias
   gramaticais conhecidas pelo sistema.
*/

det(Gen,Num) -->
        dic_det(Gen,Num),
        !, constroi_hip_cat_vazio(det), !.

prep -->
        dic_prep,
        ( !, constroi_hip(prep), ! ).

prep -->
        casos_particulares([],True_fail,prep),
        !,
        ( True_fail ).

prep -->
        (
        adquire_conhecimento(prep,Nova_cat),
        estuda_nova_cat(prep,Nova_cat,True_fail),
        ( constroi_hip(Nova_cat), True_fail, ! )
        ;
        [_W],
        prep
        ).


pron_int -->
        dic_pron_int,
        ( !, constroi_hip(pron_int), ! ).

pron_int -->
        casos_particulares([],True_fail,pron_int),
        !,
        ( True_fail ).

pron_int -->
        (
        adquire_conhecimento(pron_int,Nova_cat),
        estuda_nova_cat(pron_int,Nova_cat,True_fail),
        ( constroi_hip(Nova_cat), True_fail, ! )
        ;
        [_W],
        pron_int
        ).

nome_com(Gen,Num) -->
        dic_nome_com(Gen,Num),
        ( !, constroi_hip(nome_com), ! ).
```

```
nome_com(Gen,Num) -->
        casos_particulares([Gen,Num],True_fail,nome_com),
        !,
        { True_fail }.

nome_com(Gen,Num) -->
        (
        adquire_conhecimento(nome_com,Nova_cat),
        estuda_nova_cat(nome_com,Nova_cat,True_fail),
        { constroi_hip(Nova_cat), True_fail, ! }
        ;
        [_W],
        nome_com(Gen,Num) ).


nome_prop(Gen,Num) -->
        dic_nome_prop(Gen,Num),
        { !, constroi_hip(nome_prop), ! }.

nome_prop(Gen,Num) -->
        casos_particulares([Gen,Num],True_fail,nome_prop),
        !,
        { True_fail }.

nome_prop(Gen,Num) -->
        (
        adquire_conhecimento(nome_prop,Nova_cat),
        estuda_nova_cat(nome_prop,Nova_cat,True_fail),
        { constroi_hip(Nova_cat), True_fail, ! }
        ;
        [_W],
        nome_prop(Gen,Num) ).


adj(Gen,Num) -->
        dic_adj(Gen,Num),
        { !, constroi_hip(adj), ! }.

adj(Gen,Num) -->
        casos_particulares([Gen,Num],True_fail,adj),
        !,
        { True_fail }.

adj(Gen,Num) -->
        (
        adquire_conhecimento(adj,Nova_cat),
        estuda_nova_cat(adj,Nova_cat,True_fail),
        { constroi_hip(Nova_cat), True_fail, ! }
        ;
        [_W],
        adj(Gen,Num)
        ).
```

```
verb_aux(Num) -->
        dic_verb_aux(Num).


verb_trans(Num) -->
        dic_verb_trans(Num).


desconhecido -->
        dic_desconhecido,
        { !, constroi_hip(desconhecido), ! }.

desconhecido -->
        casos_particulares([],True_fail,desconhecido),
        !,
        { True_fail }.

desconhecido -->
        (
        adquire_conhecimento(desconhecido,Nova_cat),
        estuda_nova_cat(desconhecido,Nova_cat,True_fail),
        { constroi_hip(Nova_cat), True_fail, ! }
        ;
        [_W],
        desconhecido).

/*
   Regras de tipo (1.3) para enquadramento de sintagmas completos
*/

frase -->
        { repeticao },
        sint_nom(Id,Gen,Num),
        sint_verb(Num),
        (
        possivel_reescrita,
        repeticao
        ),
        sint_compls(Id1,Gen1,Num1),
        analisa_resto_frase,
        { possivel_reescrita }.


sint_nom(2,Gen,Num) -->
        { regra_corrente(sint_nom) },
        nome_com(Gen,Num),
        adj(Gen,Num),
        { fim_regra(sint_nom,2) }.
```

```
sint_nom(1,Gen,Num) -->
        nome_prop(Gen,Num),
        prep,
        det(Gen,Num),
        { fim_regra(sint_nom,1) }.


sint_compls(1,Gen,Num) -->
        { regra_corrente(sint_compls) },
        adj(Gen,Num),
        { fim_regra(sint_compls,1) }.


sint_verb(Num) -->
        { regra_corrente(sint_verb) },
        verb_aux(Num),
        { asserta(invocado_por(sint_verb,3)) },
        sint_verb1(Num).

sint_verb(Num) -->
        verb_trans(Num),
        { asserta(invocado_por(sint_verb,2)) }.

sint_verb(_) -->
        !,
        aprender_verbo_trans,
        { asserta(invocado_por(sint_verb,1)) }.


sint_verb1(Num) -->
        { regra_corrente(sint_verb1) },
        verb_trans(Num),
        { asserta(invocado_por(sint_verb1,3)) }.

sint_verb1(Num) -->
        verb_aux(Num),
        { asserta(invocado_por(sint_verb1,2)) }.

sint_verb1(_) -->
        adquire_conhecimento(verb_trans,Nova_cat),
        continua_para_compls(Nova_cat),
        { asserta(invocado_por(sint_verb1,1)) }.

/*
   Analise gramatical da frase.

   (Geracao das hipoteses relevantes -> aquelas referentes a
   regras onde se verificou uma situacao de conflito.)
*/
```

```
inicia:-
        repeat,
          analisa_frase,
          not (engano_na_escrita),
          const_arvore_invocacoes,
        nl,
        !.

analisa_frase:-
        cls,
        limpa_historial,
        code_world(_,ecran),
        entrada_sis(L_ent),
        code_world(_,def),
        cls,
        limpa_tudo,
        frase(L_ent,L_sai),
        !.


limpa_historial:-
        code_world(_,hist),
        current_predicate(Pred/Arg),
        functor(Pred,Arg,Cab),
        clause(Cab,Corpo),
        (
        retract(reg_antiga(Cab,Corpo))
        ;
        retract(reg_nova(Cab,Corpo))
        ),
        fail.

limpa_historial.

limpa_tudo:-
        limpa_bd,
        abolish(recomecar/0),
        abolish(reg_corrente/1),
        abolish(reg_reescrita/1),
        abolish(invocado_por/2),
        abolish(reconhecimento_livre/0),
        abolish(frase_escrita/1),
        abolish(engano_na_escrita/0).

limpa_bd:-
        abolish(notif_vazio/2),
        abolish(notif_vazio/1),
        abolish(nao_notif_vazio/0),
        abolish(hip/1),
        abolish(regra/2),
        abolish(nao_existe_concordancia/0),
        abolish(conflito/0),
```

```
        abolish(lista_inv/1),
        abolish(nao_tenta_corrigir/1),
        abolish(ja_houve_correccao/1),
        limpa_exclusao,
        !.

limpa_exclusao:-
        abolish(analise_em_exclusao/1),
        abolish(ja_houve_exclusao/0).

/*
 As construcoes gramaticais necessarias para o "dialogo" proposto
 encontram-se no ficheiro "gram.ari".

 O lexico encontra-se no ficheiro "lexico.ari"

 Predicados de controle :
*/


repeticao:-
        (
        reconhecimento_livre
        ;
        asserta(reconhecimento_livre)
        ),
        !,
        repete.

repete.

repete:-
        (
        retract(reconhecimento_livre)
        ;
        asserta(reconhecimento_livre)
        ),
        (
        recomecar,
        asserta(reg_corrente(repete_vazio))
        ;
        asserta(reg_corrente(repete))
        ),
        limpa_bd,
        abolish(recomecar/0),
        nl, write($Segue-se nova tentativa $), nl,
        !,
        repete.
```

```
notifica_vazio(Cat):-
        (
           (
           reconhecimento_livre
           ;
           nao_notif_vazio
           )
        ;
        asserta(notif_vazio(Cat))
        ),
        !.


nao_notifica_vazio:-
        (
        nao_notif_vazio
        ;
        asserta(nao_notif_vazio)
        ),
        !.


regra_corrente(Regra):-
        asserta(reg_corrente(Regra)),
        !.


fim_regra(_,_):-
        recomecar,
        limpa_bd,
        !, fail.

fim_regra(Regra,Id):-
        asserta(invocado_por(Regra,Id)),
        (
        reconhecimento_livre
        ;
           (
           todas_inv_vazio,
           asserta(notif_vazio(Regra,Id))
           ;
           abolish(notif_vazio/2)
           ),
        act_hipoteses(Regra,Id),
        limpa_exclusao
        ), !.


invocacao_vazio(_,_):-
        recomecar,
        limpa_bd,
        !, fail.
```

```
invocacao_vazio(Regra,Id):-
        asserta(invocado_por(Regra,Id)),
        (
        reconhecimento_livre
        ;
        asserta(notif_vazio(Regra,Id)),
        limpa_exclusao
        ), !.


todas_inv_vazio:-
        findall(Cat_v,retract(notif_vazio(Cat_v)),L_v),
        findall(Hip,hip_geradas(Hip),L_h),
        !,
        L_v == L_h.


act_hipoteses(Regra,Id):-
        not(conflito),
        tira_hip.

act_hipoteses(Regra,Id):-
        asserta(regra(Regra,Id)),
        asserta(hip(fim_regra)),
        retract(conflito).


hip_geradas(Hip):-
        hip(H),
        acaba_hip(H,Cut_true,Fail_true),
        Cut_true,
        Fail_true,
        Hip = H.


acaba_hip(fim_regra,!,fail):-
        !.

acaba_hip(_,true,true):-
        !.

tira_hip:-
        hip(Cat),
        (
        Cat == fim_regra
        ;
        tira_hipotese(Cat),
        fail
        ).

tira_hip.
```

```
constroi_hip(_):-          /*A existencia deste facto (recomecar)*/
        recomecar,         /*na b.d. ira implicar o retrocesso   */
        !, fail.           /*e o inicio de um novo reconhecimento*/

constroi_hip(engano_na_escrita):-
        !.

constroi_hip(Cat):-
        (
        reconhecimento_livre
        ;
        analise_em_exclusao(Inv), !,
           (
           Inv \= Cat, !, fail
           ;
           abolish(analise_em_exclusao/1),
           asserta(ja_houve_exclusao),
           asserta(hip(Cat)),
           abolish(notif_vazio/1)
           )
        ;
        asserta(hip(Cat)),
        abolish(notif_vazio/1)
        ),
        !.

constroi_hip_cat_vazio(_,_,_):-
        recomecar,         /*A existencia deste facto (recomecar)*/
        !, fail.           /*na b.d. ira implicar o retrocesso   */
                           /*e o inicio de um novo reconhecimento*/
constroi_hip_cat_vazio(engano_na_escrita):-
        !.

constroi_hip_cat_vazio(_,[],[]):-
        !.

constroi_hip_cat_vazio(Cat,L,L):-
        (
        reconhecimento_livre
        ;
        analise_em_exclusao(Inv), !,
           (
           Inv \= Cat, !, fail
           ;
           abolish(analise_em_exclusao/1),
           asserta(ja_houve_exclusao),
           asserta(hip(Cat)),
           act_cat_vazio(Cat)
           )
        ;
        ( ha_hip_verbo
        ;
```

```
        asserta(hip(Cat)),
        act_cat_vazio(Cat) )
        ), !.


act_cat_vazio(Cat):-
        ult_notif(X),
        X == Cat,
        !.

act_cat_vazio(_):-
        abolish(notif_vazio/1),
        !.


ult_notif(X):-
        notif_vazio(X),
        !.


casos_particulares(_,fail,_,_,_):-
        recomecar,
        !.

casos_particulares(_,true,_,[],[]):-
        !.

casos_particulares(_,_,_,_,_):-
        ja_houve_exclusao,
        !, fail.

casos_particulares(_,fail,Cat,_,_):-
        (
        reconhecimento_livre
        ;
        analise_em_exclusao(_)
        ),
        !.

casos_particulares(L_par,true,Cat,[Pal|Resto],[]):-
        existe_parinst(L_par),
        nao_notifica_vazio,
        const_dic_termo(Cat,Pal,Resto,Termo),
        Termo,
        abolish(nao_notif_vazio/0),
        assertz(nao_existe_concordancia),
        termina_analise(L_par,Termo,Pal,1).

casos_particulares(_,true,_,L,L):-
        abolish(nao_notif_vazio/0),
        ha_hip_verbo,
        !.
```

```
casos_particulares(_,Meta_car,Cat,L,L_res):-
        todas_inv_vazio,
        reg_corrente(Regra),
        !,
        (
        existe_outro_predicado(L,L_res),
        abolish(lista_inv/1),
        write($SITUACAO ANOMALA -> o predicado anterior
                                              devia ter $), nl,
        write($                        feito o
                                reconhecimento desta palavra$),
        nl,
        Meta_car = true
        ;
        existe_outra_clausula(Regra,L,L_res),
        abolish(lista_inv/1),
        Meta_car = fail
        ;
        adquire_conhecimento(Cat,Nova_cat,L,L1),
        estuda_nova_cat(Cat,Nova_cat,Aux_meta_car,L1,L2),
           (
           Aux_meta_car == true,
           constroi_hip(Nova_cat),
           Meta_car = Aux_meta_car,
           L_res = L2
           ;
           pertence_cat_lista_inv(Nova_cat,Meta_car),
           abolish(lista_inv/1),
           abolish(conflito/0),
              (
              Meta_car == true,
              L_res = [],
              actualiza_para_predicado
              ;
              L_res = L,
              actualiza_para_clausula(Nova_cat)
              )
           ;
           constroi_hip(Nova_cat),
           abolish(lista_inv/1),
           Meta_car = true,
           [_|L_res] = L
           )    ), !.

pertence_cat_lista_inv(Nova_cat,Meta_car):-
        lista_inv(L_1_cat),
        pertence([Meta_car] -> L_cat,L_1_cat),
        pertence(Nova_cat,L_cat),
        !.
```

```
existe_outro_predicado([Pal|Resto],[]):-
        notif_vazio(Reg_notificada,_),
        existe_rec(true,Reg_notificada,Pal,Resto,Cat_rec),
        actualiza_para_predicado.

actualiza_para_predicado:-
        asserta(recomecar),
        limpa_bd,
        nl, write($Vou entrar em backtracking por causa de
                    vazio$), nl.

existe_outra_clausula(Regra,[Pal|T],[Pal|T]):-
        existe_rec(fail,Regra,Pal,T,Cat_rec),
        actualiza_para_clausula(Cat_rec).

actualiza_para_clausula(Cat_rec):-
        tira_hip,
        asserta(analise_em_exclusao(Cat_rec)).

existe_rec(Meta_car,Regra,Pal,Resto,Cat_rec):-
        current_predicate(Regra/Nargs),
        functor(Cabeca,Regra,Nargs),
        list_vazio(L_v),
        findall(L_cat,constroi_1_1_cat(
                        Meta_car,Cabeca,L_v,L_cat),L_1_cat),
        assertz(lista_inv(L_1_cat)),
        !,
        tenta_reconhecer(L_1_cat,Pal,Resto,Cat_rec).

constroi_1_1_cat(Meta_car,Cabeca,L_v,L_cat):-
        clause(Cabeca,Corpo),
        constroi_1_cat(Corpo,L_v,Lista,Meta_car),
        L_cat = [Meta_car] -> Lista.

constroi_1_cat((Inv,Resto),L_v,L,Meta_car):-
        functor(Inv,Nome,_),
        Nome == regra_corrente,
        !,
        constroi_1_cat(Resto,L_v,L,Meta_car).

constroi_1_cat((Inv,Resto),L_v,[Nome_inv|T],Meta_car):-
        functor(Inv,Nome_inv,_),
        pertence(Nome_inv,L_v),
        !,
        constroi_1_cat(Resto,L_v,T,Meta_car).

constroi_1_cat((Inv,_),L_v,[Nome_inv],Meta_car):-
        functor(Inv,Nome_inv,_),
        (
        classe_fechada(L_f),
        pertence(Nome_inv,L_f)
        ;
```

```
        classe_aberta(L_a),
        pertence(Nome_inv,L_a)
        ),
        !.

constroi_1_cat((Inv),L_v,[Nome_inv],Meta_car):-
        Meta_car == fail,
        functor(Inv,Nome_inv,_),
        Nome_inv == invocacao_vazio,
        !.

constroi_1_cat((Inv),L_v,[],_):-
        !.

tenta_reconhecer(L_1_cat,Pal,Resto,Cat_rec):-
        pertence(A -> L_inv,L_1_cat),
        pertence(Cat,L_inv),
        reconhece(Cat,Pal,Resto),
        abolish(notif_vazio/1),
        (
        Cat == invocacao_vazio,
        Cat_rec = sem_sentido
        ;
        [Cat_rec|T] = L_inv),
        !.

reconhece(invocacao_vazio,_,_):-
        !.

reconhece(Cat,Pal,Resto):-
        const_dic_termo(Cat,Pal,Resto,Termo),
        Termo, !,
        (
        retract(notif_vazio(_)),
        !, fail
        ;
        true ), !.

ha_hip_verbo:-
        hip(Cat),
        verbo_prim_ou_seg(Cat),
        !,
        pertence(Cat,[verb_aux,verb_trans]).

verbo_prim_ou_seg(Cat):-
        pertence(Cat,[verb_aux,verb_trans]).

verbo_prim_ou_seg(fim_regra):-
        !, fail.

verbo_prim_ou_seg(_).
```

```
existe_parinst([A|T]):-
        nonvar(A),
        !.

existe_parinst([_|T]):-
        !,
        existe_parinst(T).

estuda_nova_cat(_,engano_na_escrita,true,_,[]):-
        abolish(regra/2),
        abolish(conflito/0),
        !.

estuda_nova_cat(Cat_esperada,Cat_esperada,true,[_|T],T):-
        !.

estuda_nova_cat(Cat_esperada,Cat,true,L,L):-
        pertence(Cat,[verb_aux,verb_trans]),
        zona_conflito, !.

estuda_nova_cat(_,_,fail,[_|T],T):-
        zona_conflito, !.


zona_conflito:-
        conflito, !.
zona_conflito:-
        assertz(conflito), !.

adquire_conhecimento(Cat,Nova_cat,[Pal|T],[Pal|T]):-
        aprende_sozinho(Cat,Pal,T,Nova_cat),
        !.

adquire_conhecimento(Cat,Nova_cat,[Pal|T],[Pal|T]):-
        abolish(nao_notif_vazio/0),
        code_world(_,ecran),
        inter_accao(Pal,Cat,Nova_cat,T),
        code_world(_,def),
        cls,
        (
        Nova_cat \= engano_na_escrita,
        const_dic_termo(Nova_cat,Pal,Nao_inst,Novo_termo),
        verifica_existencia(Novo_termo),
        !
        ;
        asserta(Nova_cat) ), !.


aprende_sozinho(Cat,Pal,Resto,Nova_cat):-
        (
        classe_fechada(L)
        ;
```

```
        classe_aberta(L)
        ),
        apaga(Cat,L,Lista),
        pertence(Nova_cat,Lista),
        const_dic_termo(Nova_cat,Pal,Resto,Termo),
        nao_notifica_vazio,
        Termo,
        abolish(nao_notif_vazio/0).


verifica_existencia(Novo_termo):-
        clause(Novo_termo,(true)).

verifica_existencia(Novo_termo):-
        asserta(Novo_termo).

const_dic_termo(Nome,Pal,Resto,Cat):-
        junta_dic(Nome,Dic_nome),
        current_predicate(Dic_nome/Args),
        functor(Cat,Dic_nome,Args),
        Aux is Args - 1,
        arg(Aux,Cat,[Pal|Resto]),
        arg(Args,Cat,Resto),
        !.

junta_dic(Nome,Dic_nome):-
        name(dic_,L1),
        name(Nome,L2),
        concat(L1,L2,L3),
        name(Dic_nome,L3).

termina_analise(L_par,Termo,Pal,N):-
        nl,
        write($A analise feita levou a concluir nao haver
                concordancia.$),nl,
        write($Esperava-se que a palavra $),write(Pal),
        write($ se encontrasse no $),write(Num),nl,
        write(L_par),nl,write(Termo),nl,
        write($REESCREVA A SUA FRASE$),nl,nl.


possivel_reescrita:-
        nao_existe_concordancia,
        !.

possivel_reescrita:-
        retract(regra(Regra,Id)),
        organiza_informacao(Regra,Id),
        fail.

possivel_reescrita:-
        !.
```

```
organiza_informacao(Regra,Id):-
        nl,nl,write(Regra),nl,write($ $),write(Id),nl,
        current_predicate(Regra/Args),
        functor(Cabeca,Regra,Args),
        arg(1,Cabeca,Id),
        clause(Cabeca,Corpo),
        coloca_historial(reg_antiga(Cabeca,Corpo)),
        findall(Nome,lista_cat(Nome,Corpo),L_cat),
        write(L_cat),nl,
        tira_fim_regra,
        lista_hip([],L_hip),
        trata_caso_verbo(L_hip,L_hip_res),
        write(L_hip),nl,
        reescreve(Regra,Id,L_cat,L_hip_res),
        !.


tira_fim_regra:-
        ve_hipotese(Cat),
        Cat == fim_regra,
        retract(hip(Cat)).

tira_fim_regra.

lista_hip(L,L_hip):-
        tira_hipotese(Cat),
        Cat \= fim_regra,
        !,
        lista_hip([Cat|L],L_hip).

lista_hip(L,L):-
        !.

ve_hipotese(Cat):-
        hip(Cat),
        !.

tira_hipotese(Cat):-
        retract(hip(Cat)),
        !.


trata_caso_verbo(L_hip,L_hip_res):-
        (
        pertence(verb_aux,L_hip),
        apaga(verb_aux,L_hip,L_hip_res)
        ;
        pertence(verb_trans,L_hip),
        apaga(verb_trans,L_hip,L_hip_res)
        ),
        !.
```

```
trata_caso_verbo(L_hip,L_hip).

lista_cat(Nome,Corpo):-
        pert1(Elem,Corpo),
        functor(Elem,Nome,_).


pert1(A,(A,_)).

pert1(A,(_,T)):-
        !,
        pert1(A,T).

pert1(A,(A)).


coloca_historial(Facto):-
        code_world(_,hist),
        assertz(Facto),
        code_world(_,def),
        !.

/*
 Predicados de controle, referentes a' "zona verbal".
*/

aprender_verbo_trans([],[]):-
        (
        nao_existe_concordancia
        ;
        nl, nl,
        write($FRASE SEM COMPONENTE VERBAL !$),
        nl, nl
        ),
        !.

aprender_verbo_trans(_,_):-
        reconhecimento_livre,
        !, fail.

aprender_verbo_trans(L,L_res):-
        adquire_conhecimento(verb_trans,Nova_cat,L,L1),
        (
        Nova_cat == verb_trans,
        [_|L_res] = L
        ;
        existe_outro_predicado(L,L_res),
        !, fail
        ;
        [_|L_aux] = L,
        gera_conflito_anterior(Nova_cat,L_aux,L_res)
        ), !.
```

```
gera_conflito_anterior(Nova_cat,L,L_res):-
        invocado_por(Regra,Id),
        !,
        (
        regra(X_r,X_i),
        Regra == X_r,
        Id == X_i,
        tira_fim_regra
        ;
        asserta(regra(Regra,Id)),
        current_predicate(Regra/Nargs),
        functor(Cabeca,Regra,Nargs),
        arg(1,Cabeca,Id),
        clause(Cabeca,Corpo),
        findall(Nome,lista_cat(Nome,Corpo),L_cat),
        diferenca_conj(L_cat,
                       [regra_corrente,fim_regra,
                                  invocacao_vazio],L_catres),
        gera_hipoteses(L_catres)
        ),
        asserta(hip(Nova_cat)),
        categorias_esperadas(L,L_res),
        !.


gera_hipoteses(L_catres):-
        pertence(Cat,L_catres),
        asserta(hip(Cat)),
        fail.

gera_hipoteses(_).


categorias_esperadas([],[]):-
        nl, nl,
        write($FRASE SEM COMPONENTE VERBAL !$),
        nl, nl.

categorias_esperadas(L,L_res):-
        concat(L1,L_res,L),
        ultimo(L1,Pal),
        adquire_conhecimento(_,Nova_cat,[Pal],_),
        (
        pertence(Nova_cat,[verb_aux,verb_trans])
        ;
        asserta(hip(Nova_cat)),
           (   L_res == []
           ;
           fail )
        ),
        asserta(hip(fim_regra)),
        !.
```

```
continua_para_compls(verb_trans,[_|T],T):-
        !.

continua_para_compls(_,L,L):-
        !.

analisa_resto_frase([],[]):-
        !.

analisa_resto_frase(L,L_res):-
        aprender_verbo_trans(L,L_res),
        !.

/*      Suporte computacional para a reescrita de regras.

        Eventual reescrita de regras.
*/

reescreve(Regra,Id,L_cat,L_hip):-
        diferenca_conj(L_cat,[regra_corrente,
                                        invocacao_vazio],L_cat1),
        cat_comum(L_cat1,L_hip,L_comum,L_fraca,L_forte,Resto),
        prep_rees(Regra,Id,L_cat,L_comum,Resto,L_forte,L_fraca).


cat_comum([],[],[],[],[],[]):-
        nl,nl,write($ACONTECEU UM ERRO (1) $),nl.

cat_comum([],L,[],[],L,[]):-
        !.

cat_comum(L,[],[],L_res,[],Resto):-
        segue_fim_regra(L,L_res,Resto).

cat_comum([A|T],[A|T1],[A|T2],L,L1,Rs):-
        !,
        cat_comum(T,T1,T2,L,L1,Rs).

cat_comum(L,L1,[],L_res,L1,Resto):-
        segue_fim_regra(L,L_res,Resto).


segue_fim_regra([fim_regra|T],[],T).

segue_fim_regra([A|T],[A|T1],L):-
        segue_fim_regra(T,T1,L).
```

```
prep_rees(Regra,Id,L_cat,[],Resto,L_forte,_):-
        (
        Resto == []
        ;
        [Inv|_] = Resto,
        rees_resto(Inv)
        ),
        (
        L_forte \= [],
        rees_a_prim_regra(Regra,Id_prim_clause),
        Id1 is Id_prim_clause + 1
        ;
        current_predicate(Regra/Nargs),
        functor(Termo,Regra,Nargs),
        descobre_id(Termo,Id_ult_clause),
        Id1 is Id_ult_clause - 1
        ),
        !,
        rees_lforte_lfraca(Regra,Id1,lista_forte,L_forte,Resto).

prep_rees(Regra,Id,L_cat,L_comum,Resto,L_forte,L_fraca):-
        rees_lcomum(Regra,Id,L_cat,Resto,L_comum,Nova_cat),
        (
        L_forte \= [],
        rees_lforte_lfraca(Nova_cat,2,lista_forte,L_forte,[]), !,
        rees_lforte_lfraca(Nova_cat,1,lista_fraca,L_fraca,[])
        ;
        rees_lforte_lfraca(Nova_cat,2,lista_forte,L_fraca,[]), !,
        rees_lforte_lfraca(Nova_cat,1,lista_forte,L_forte,[])
        ),
        !.


rees_lcomum(Regra,Id,L_cat,Resto,L_comum,Nova_cat):-
        act_lcomum(Regra,Id,Resto,L_comum,L_comum1,Nova_cat),
        (
        pertence(regra_corrente,L_cat),
        L_result = [regra_corrente|L_comum1]
        ;
        L_result = L_comum1
        ),
        cat_sem_par_gram(L_sempar),
        define_args(L_result,L_sempar,0,Ns,[],L_argsem),
        Nargs is Ns + 1,
        lista_nao_inst(L_parninst,Nargs),
        junta_os_id([Nova_cat|Resto],L_argsem,L_argsem1),
        listasemant_nao_inst(L_argsem1,Regra,L_sem,L_semninst),
        const_clausula(lista_comum,Regra,Id,L_result,
                                L_parninst,L_sem,L_semninst).
```

```
rees_1forte_1fraca(Nova_cat,Id,Tipo_lista,[],_):-
        lista_nao_inst(L_parninst,2),
        listasemant_nao_inst([],Nova_cat,L_sem,L_semninst),
        const_clausula(Tipo_lista,Nova_cat,Id,[invocacao_vazio],
                                L_parninst,L_sem,L_semninst).

rees_1forte_1fraca(Nova_cat,Id,Tipo_lista,Lista,Resto):-
        act_1forte_1fraca(Tipo_lista,Lista,Resto,L_result),
        cat_sem_par_gram(L_sempar),
        define_args(L_result,L_sempar,0,Ns,[],L_argsem),
        Nargs is Ns + 1,
        lista_nao_inst(L_parninst,Nargs),
        junta_os_id(Resto,L_argsem,L_argsem1),
        listasemant_nao_inst(L_argsem1,Nova_cat,
                                L_sem,L_semninst),
        const_clausula(Tipo_lista,Nova_cat,Id,L_result,
                                L_parninst,L_sem,L_semninst).


rees_a_prim_regra(Regra,Id_p_clause):-
        current_predicate(Regra/Nargs),
        functor(Cabeca,Regra,Nargs),
        clause(Cabeca,Corpo),
        arg(1,Cabeca,Id_p_clause),
        tira_prim_inv(Corpo,Corpo_act),
        retract((Cabeca:-Corpo)),
        asserta((Cabeca:-Corpo_act)).


tira_prim_inv((Prim_inv,Resto),Resto):-
        functor(Prim_inv,Nome,_),
        Nome == regra_corrente.

tira_prim_inv(_,_):-
        nl, nl,
        write($(ERRO) ->> A primeira invocacao$),
        write($ da regra NAO e' "regra_corrente"$),
        nl, nl.


listasemant_nao_inst(L_argsem,Regra,L_sem,L_semninst):-
        tab_regra(L_reg -> L),
        pertence(X,L_reg),
        (
        X = Regra
        ;
        string_search(X,Regra,_)
        ),
        uniao(L,L_argsem,L1),
        length(L1,Len),
        (
        Len \= 0,
```

```
        lista_nao_inst(L_semninst,Len),
        coloca_indice(1,L1,L_sem)
        ;
        L_semninst = [],
        L_sem = []
        ),
        !.

listasemant_nao_inst(_,Regra,_,_):-
        nl, write($FALTA A ENTRADA NA TABELA SEMANTICA,
                                                RESPEITANTE $),
        write($A' REGRA :$),nl,
        write($    --->    $),write(Regra),nl.

dummy([_|T],T).

lista_nao_inst(Lista,Dim):-
        functor(Estruct,args,Dim),
        Estruct =.. L_aux,
        dummy(L_aux,Lista).


act_1comum(Regra,Id,[],L_comum,L_comum1,Nova_cat):-
        name(Regra,Lista),
        name(Id,Ext),
        concat(Lista,Ext,Aux),
        name(Nova_cat,Aux),
        concat(L_comum,[fim_regra,Nova_cat],L_comum1),
        asserta(reg_reescrita(Regra -> Nova_cat)).

act_1comum(Regra,Id,[A|T],L_comum,L_comum1,Nova_cat):-
        name(A,Lista),
        name(a_,[_,Under]),
        name(Id,Int),
        concat([Under],Int,Ext),
        concat(Lista,Ext,Aux),
        name(Nova_cat,Aux),
        concat(L_comum,[fim_regra,Nova_cat,A|T],L_comum1),
        rees_resto(A),
        asserta(reg_reescrita(Regra -> Nova_cat)).


act_1forte_1fraca(Tipo_lista,Lista,Resto,L_result):-
        concat(Lista,[fim_regra|Resto],Lista_r),
        (
        Tipo_lista == lista_forte,
        L_result = [regra_corrente|Lista_r]
        ;
        L_result = Lista_r

        ).
```

```
rees_resto(Inv):-
        current_predicate(Inv/Nargs),
        functor(Termo,Inv,Nargs),
        descobre_id(Termo,Id_ult_clause),
        coloca_inv_vazio(Termo,Id_ult_clause,Nargs).


descobre_id(Termo,Id):-
        findall(Ids,cada_id(Termo,Ids),L_id),
        ultimo(L_id,Id).


cada_id(Termo,Ids):-
        clause(Termo,_),
        arg(1,Termo,Ids).


coloca_inv_vazio(Termo,Id,Nargs):-
        arg(1,Termo,Id),
        clause(Termo,Corpo),
        (
        (_,_) = Corpo,              /* No corpo existem pelo */
        Id1 is Id - 1,              /* menos duas invocacoes */
        functor(Termo,Nome,_),
        functor(Termo_aux,Nome,Nargs),
        arg(1,Termo_aux,Id1),
        arg(Nargs,Termo_aux,Var_ninst),
        Nargs1 is Nargs - 1,
        arg(Nargs1,Termo_aux,Var_ninst),
        assertz((Termo_aux:-invocacao_vazio(Nome,Id1)))
        ;
        (Inv) = Corpo,
        functor(Inv,Nome,_),
        Nome == invocacao_vazio
        ;
        nl, nl,
        write($ERRO ->> existe uma regra
                                apenas com uma invocacao$),
        write($ <> de "invocacao_vazio" $),
        nl, nl
        ).


define_args([],_,N,N,L,L).

define_args([A|T],L_sempar,Cont,Nargs,L_aux,L_argsem):-
        (
        pertence(A,L_sempar),
        Cont1 is Cont,
        L_aux1 = L_aux
        ;
        Cont1 is Cont + 1,
```

```
            (
            tab(L_cat -> L_args),
            pertence(A,L_cat),
            uniao(L_aux,L_args,L_aux1)
            ;
            L_aux1 = L_aux
            )
        ),
        !,
        define_args(T,L_sempar,Cont1,Nargs,L_aux1,L_argsem).

junta_os_id(Resto,L_sem_aux,L_sem):-
        length(Resto,Len_resto),
        Len_resto1 is Len_resto + 1,
        constroi_lista_id(Len_resto1,Lista),
        concat(L_sem_aux,Lista,L_sem),
        !.
constroi_lista_id(0,[]):-
        !.

constroi_lista_id(Len_resto,[id|T]):-
        Len1 is Len_resto - 1,
        constroi_lista_id(Len1,T).

const_clausula(Tipo_lista,N_cab,Id,Inv_corpo,
                              L_parninst,L_sem,L_semninst):-
        pertence(id : Ind,L_sem),     /* A componente "Ind" da */
        !,                            /* "L_semninst" e' inst. */
        comp(Ind,1,L_semninst,[Id]), /* com o valor de "Id"   */
        const_cabeca(N_cab,Id,Inv_corpo,L_parninst,
                        L_sem,L_sem_res,L_semninst,Cabeca),
        const_corpo(1,3,N_cab,Id,Inv_corpo,
                         L_parninst,L_sem_res,L_semninst,Corpo),
        coloca_bd(Tipo_lista,N_cab,Id,Inv_corpo,Cabeca,Corpo).


const_cabeca(Nome_cab,Id,Inv_corpo,L_parninst,
                        L_sem,L_sem_res,L_semninst,Cabeca):-
        (
        Inv_corpo == [invocacao_vazio],
        encontra(1,1,L_parninst,L_res)
        ;
        encontra(1,2,L_parninst,L_res)
        ),
        encontra_semant(Nome_cab,L_sem,L_sem_res,
                                  L_semninst,L_res1),
        concat(L_res1,L_res,L_result),
        Cabeca =.. [Nome_cab|L_result].
```

```
const_corpo(_,_,N_cab,Id,[Inv],
                        L_parninst,L_sem,L_semninst,(Estruct)):-
        (
        Inv == fim_regra
        ;
        Inv == invocacao_vazio
        ),
        Estruct =.. [Inv,N_cab,Id],
        !.

const_corpo(I,_,N_cab,Id,[Inv],
                        L_parninst,L_sem,L_semninst,(Estruct)):-
        encontra(I,2,L_parninst,L_res),
        encontra_semant(Inv,L_sem,L_sem_res,L_semninst,L_res1),
        concat(L_res1,L_res,L_result),
        Estruct =.. [Inv|L_result],
        !.

const_corpo(I,J,N_cab,Id,[fim_regra|T],
                L_parninst,L_sem,L_semninst,(Estruct,Resto)):-
        Estruct =.. [fim_regra,N_cab,Id],
        !,
        const_corpo(I,J,N_cab,Id,T,
                        L_parninst,L_sem,L_semninst,Resto).

const_corpo(I,J,N_cab,Id,[Inv,Inv1|T],
                L_parninst,L_sem,L_semninst,(Estruct,Resto)):-
        (
        Inv1 == fim_regra,
        T == [],
        encontra(I,2,L_parninst,L_res),
        encontra_semant(Inv,L_sem,L_sem_res,L_semninst,L_res1),
        concat(L_res1,L_res,L_result),
        Estruct =.. [Inv|L_result],
        I1 is I, J1 is J
        ;
        Inv == regra_corrente,
        Estruct =.. [Inv,N_cab],
        L_sem_res = L_sem,
        I1 is I, J1 is J
        ;
        encontra(I,J,L_parninst,L_res),
        encontra_semant(Inv,L_sem,L_sem_res,L_semninst,L_res1),
        concat(L_res1,L_res,L_result),
        Estruct =.. [Inv|L_result],
        I1 is J, J1 is J + 1
        ),
        !,
        const_corpo(I1,J1,N_cab,Id,[Inv1|T],
                        L_parninst,L_sem_res,L_semninst,Resto).
```

```
coloca_bd(lista_comum,N_cab,Id,_,Cabeca,Corpo):-
        tira_pred_anteriores(N_cab,Id,[],Lista_pred),
        asserta((Cabeca:-Corpo)),
        poe_pred_anteriores(Lista_pred),
        coloca_historial(reg_nova(Cabeca,Corpo)).

coloca_bd(_,_,_,Inv_corpo,Cabeca,Corpo):-
        (
        [regra_corrente|_] = Inv_corpo,
        asserta((Cabeca:-Corpo))
        ;
        assertz((Cabeca:-Corpo))
        ),
        coloca_historial(reg_nova(Cabeca,Corpo)).

tira_pred_anteriores(N_cab,Id,T,T1):-
        current_predicate(N_cab/Nargs),
        functor(Cabeca,N_cab,Nargs),
        continua_a_tirar(Id,Cabeca,Corpo),
        !,
        tira_pred_anteriores(N_cab,Id,[Cabeca&Corpo|T],T1).
tira_pred_anteriores(_,_,T,T):-
        !.

continua_a_tirar(Id,Cabeca,Corpo):-
        clause(Cabeca,Corpo),
        retract((Cabeca:-Corpo)),
        arg(1,Cabeca,Id1),
        !,
        Id1 > Id.

poe_pred_anteriores(Lista_pred):-
        pertence(Cabeca&Corpo,Lista_pred),
        asserta((Cabeca:-Corpo)),
        fail.

poe_pred_anteriores(_).

encontra(I,J,L_args,L_res):-
        comp(I,1,L_args,A1),
        comp(J,1,L_args,A2),
        concat(A1,A2,L_res),
        !.

encontra_semant(Nome_regra,L_sem,L_sem_res,L_semninst,L_res):-
        (
        tab(Lista -> L)
        ;
        tab_regra(Lista -> L)
        ),
        pertence(X,Lista),
        string_search(X,Nome_regra,_),
```

```
        const_semant_ninst(L,L_sem,L_sem_res,
                                L_semninst,[],L_res),
        !.

const_semant_ninst([],L_sem_res,L_sem_res,_,L,L).

const_semant_ninst([A|T],L_sem,L_sem_res,L_semninst,L,L_res):-
        pertence(A:Ind,L_sem),
        (
        A == id,
        apaga(A:Ind,L_sem,L_sem1)
        ;
        L_sem1 = L_sem
        ),
        comp(Ind,1,L_semninst,L1),
        concat(L,L1,L2),
        const_semant_ninst(T,L_sem1,L_sem_res,
                                L_semninst,L2,L_res).

comp(_,_,[],[]).

comp(I,I,[A|_],[A]).

comp(I,Cont,[A|T],L):-
        Cont1 is Cont + 1,
        comp(I,Cont1,T,L).

coloca_indice(_,[],[]).

coloca_indice(Ind,[A|T],[A:Ind|T1]):-
        Ind1 is Ind + 1,
        coloca_indice(Ind1,T,T1).


/*
 TABELAS SEMANTICAS
*/

cat_grams([det , nome_com , nome_prop , adj ,
           pron_int , prep , desconhecido]).

cat_sem_par_gram([regra_corrente , fim_regra , invocacao_vazio]).

tab([det , nome_com , nome_prop , adj] -> [genero , numero]).

tab([pron_int, prep, desconhecido] -> []).

tab([verb_aux , verb_trans] -> [numero]).


tab_regra([sint_nom , sint_compls] -> [id,genero,numero]).
```

```
/*   Suporte computacional para a interacção coloquial com o
     utilizador.

     Interaccao coloquial com o utilizador.

    (Gestao do ecran durante a interaccao com o utilizador.
     Este ficheiro e' carregado para o "mundo" : ecran)

  Gestao do ecran durante para a entrada no sistema.
*/

entrada_sis(L_ent):-
        tmove(23,0),
        wa(1,31),
        tscroll(0,(0,0),(23,79)),
        janela($$,amarelo,esq_simples,vermelho,0,0,23,79),
        tmove(2,2),
        write_attr($Frase a analisar :$,normal),
        tmove(4,2),
        write_attr($>> $,normal),
        code_world(_,leitor),
        leitor(L_ent),
        code_world(_,ecran),
        asserta(frase_escrita(L_ent)),
        !.

/*
  Gestao do ecran durante a aprendizagem de uma categoria
  gramatical.
*/

inter_accao(Pal,Cat,Nova_cat,Resto_frase):-
        frase_escrita(Frase),
        !,
        repeat,
          tmove(23,0),
          wa(1,31),
          tscroll(0,(0,0),(23,79)),
          tmove(0,3),
          write_attr($Frase a analisar :$,normal),
          tmove(1,3),
          write_attr($>> $,normal),
          esc_lista(Frase,Resto_frase),
          atribui_cat(Cat,Cat_atribuida),
          prim_jan(Pal,Cat_atribuida,Opcao),
          prox_janela(Pal,Cat_atribuida,Opcao,Nova_cat).


prim_jan(Pal,Cat_atribuida,Opcao):-
        jan_inf(Pal,piscar,esq_dupla),
        jan_hip(Pal,Cat_atribuida,piscar_rev_vid,esq_dupla),
        jan_quest,
```

```
        uso_do_teclado(1),
        jan_resp(Opcao,esq_simples),
        !.

prox_janela(Pal,desconhecido,sim,desconhecido):-
        !.

prox_janela(Pal,Cat,sim,Cat):-
        !,
        janela($ AQUISICAO DE CONHECIMENTO $,amarelo,
                esq_simples,vermelho,0,0,23,79),
        janela_dados(sim,Cat,_,L_dados,_).

prox_janela(Pal,Cat,nao,Nova_cat):-
        repeat,
          janela_aquisicao(Opcao,Pal,Cut_true,
                                        True_fail,Coor_aux),
          Cut_true,
          True_fail,
        janela_dados(nao,Opcao,Nova_cat,L_dados,Coor_aux),
        !.

prox_janela(Pal,Cat,nao_sei,Nova_cat):-
        !,
        janela_ajuda(Cat,Nova_cat).

prox_janela(Pal,Cat,engano,engano_na_escrita):-
        !.

jan_inf(Pal,Attr_pal,Tipo_esq):-
        janela($ INFORMACAO : $,amarelo,
                        Tipo_esq,vermelho,3,7,8,37),
        tmove(5,9),
        write_attr($Nao conheco a palavra :$,normal),
        tmove(6,9),
        write_attr($ -> $,normal),
        write_attr(Pal,Attr_pal),
        !.

jan_hip(Pal,Cat_atribuida,Attr_cat,Tipo_esq):-
        janela($ HIPOTESE : $,amarelo,
                        Tipo_esq,vermelho,3,43,8,73),
        (
        converte(Str,Cat_atribuida),
        !
        ;
        cls, write($  ERRO -> Nao existe entrada em "converte"$),
        nl,
        keyb(_,_)
        ),
        tmove(5,45),
        write_attr($A palavra : $,normal),
```

```
        write_attr(Pal,rev_vid),
        tmove(6,45),
        write_attr($e' um  $,normal),
        write_attr(Str,Attr_cat),
        write_attr($.$,normal),
        !.

uso_do_teclado(1):-
        tmove(23,30),
        write_attr($COMO USAR O TECLADO :$,verde),
        tmove(24,1),
        write($<$), put(25), tmove(24,2), wa(1,10), tmove(24,3),
        write($> == Escolher opcao de resposta$),
        write($          $),
        write($<$),
        write_attr($ENTER$,verde_preto),
        write($> == Aceitar opcao corrente$),
        !.

jan_quest:-
        janela($ QUESTAO $,rev_vid,
                           esq_simples,vermelho,10,3,14,41),
        tmove(12,5),
        write_attr($A minha "HIPOTESE" esta' correcta ?$,normal),
        !.


jan_resp(Opcao,Tipo_esq):-
        janela($ RESPOSTA $,amarelo,
                           Tipo_esq,vermelho,10,47,22,69),
        poe_menu([@,$Sim$,@,$Nao$,@,$Nao sei$,@,$Enganei-me$,#,
                  $(Desejo voltar a$,#,$escrever a frase)$],
                  10,51,L_coord),
        faz_a_escolha(L_coord,Coord),
        indexa(Coord,L_coord,1,Index),
        opcoes(1,Index,Opcao),
        !.

janela_aquisicao(Opcao,Pal,Cut_true,True_fail,Coor_aux):-
        janela($ AQUISICAO DE CONHECIMENTO $,amarelo,
                esq_simples,vermelho,0,0,23,79),
        tmove(2,2),
        write_attr($Vamos tentar identificar a palavra : $,
                                                  normal),
        write_attr(Pal,rev_vid),
        tmove(4,2),
        write_attr($Qual a sua categoria gramatical ?$,normal),
        janela($ Categorias $,amarelo,
                           esq_simples,vermelho,6,10,20,33),
        Lista = [@,$Nome proprio$,@,$Nome comum$,@,$Adjectivo$,@,
                  $Verbo transitivo$,@,$Preciso ajuda$,#,
                  $(Nao tenho a$,#,$certeza)$],
```

```
        poe_menu(Lista,6,14,L_coord),
        janela($$,sem_sentido,esq_dupla,vermelho,21,18,23,61),
        poe_menu([$Quero rectificar a resposta anterior$],21,22,
                                                    Coor_aux),
        concat(L_coord,Coor_aux,L_coordres),
        faz_a_escolha(L_coordres,Coordres),
        indexa(Coordres,L_coordres,1,Index),
        opcoes(2,Index,Opcao),
        !,
        (
        Opcao == rectifica,
        Cut_true = !,
        True_fail = fail
        ;
        Cut_true = true,
        True_fail = true
        ),
        !.


janela_dados(_,Opcao,Nova_cat,_,_):-
        Opcao == ajuda,
        !,
        janela_ajuda(_,Nova_cat).

janela_dados(Sim_nao,Opcao,Opcao,L_dados,Coor_aux):-
        repeat,
          jan_entrada_dados(Sim_nao,Opcao,L_dados,
                            Cut_true,True_fail,Coor_aux),
          Cut_true,
          True_fail.

jan_entrada_dados(Sim_nao,verb_trans,L_dados,
                  Cut_true,True_fail,Coor_aux):-
        (
        Sim_nao == sim,
        L1 = 4, C1 = 23,
        L2 = 9, C2 = 56
        ;
        L1 = 6,  C1 = 43,
        L2 = 11, C2 = 76
        ),
        repeat,
          janela($$,sem_sentido,esq_simples,
                                  vermelho,L1,C1,L2,C2),
          Lr is L1 + 2, Cr is C1 + 2, tmove(Lr,Cr),
          write_attr($Escreva o infinitivo do verbo$,normal),
          Lr1 is Lr + 1, tmove(Lr1,Cr),
          write_attr($ -> $,normal),
          leitor_lim([Infinitivo],25),
          poe_jan_rectifica(Sim_nao,Coor_aux),
          jan_confirm(Opcao3,Coor_aux),
```

```
        ( Opcao3 == correcto,
        Cut_true = true, True_fail = true
        ;
        Opcao3 == rectifica,
        Cut_true = !, True_fail = fail
        ;
        fail ).

jan_entrada_dados(Sim_nao,Opcao,L_dados,
                  Cut_true,True_fail,Coor_aux):-
        ( Sim_nao == sim,
        L1  = 2,   C1  = 50,
        L2  = 13,  C2  = 76,
        Lr1 = 3,   Cr1 = 14,
        Lr2 = 5,   Cr2 = 54,
        Lr3 = 10,  Cr3 = 54,
        Lm1 = 3,   Cm1 = 63,
        Lm2 = 8,   Cm2 = 63
        ;
        L1  = 9,   C1  = 50,
        L2  = 20,  C2  = 76,
        Lr1 = 7,   Cr1 = 46,
        Lr2 = 12,  Cr2 = 54,
        Lr3 = 17,  Cr3 = 54,
        Lm1 = 10,  Cm1 = 63,
        Lm2 = 15,  Cm2 = 63 ),
        repeat,
          tmove(Lr1,Cr1),
          write_attr($Indique o seu genero e numero :$,normal),
          janela($$,sem_sentido,esq_simples,
                                  vermelho,L1,C1,L2,C2),
          tmove(Lr2,Cr2), write_attr($Genero$,normal),
          tmove(Lr3,Cr3), write_attr($Numero$,normal),
          ( Opcao \== adj,
          Lista = [$MASCULINO$,@,$FEMININO$]
          ;
          Opcao == adj,
          Lista = [$MASCULINO$,$FEMININO$,$INDEFINIDO$] ),
          poe_menu(Lista,Lm1,Cm1,L_c1),
          poe_menu([$SINGULAR$,$PLURAL$,$INDEFINIDO$],
                                          Lm2,Cm2,L_c2),
          faz_a_escolha(L_c1,Op1),
          faz_a_escolha(L_c2,Op2),
          poe_jan_rectifica(Sim_nao,Coor_aux),
          jan_confirm(Opcao3,Coor_aux),
        ( Opcao3 == correcto,
        Cut_true = true, True_fail = true
        ;
        Opcao3 == rectifica,
        Cut_true = !, True_fail = fail
        ;
        fail ).
```

```
janela_ajuda(Cat,Opcao):-
        janela(0 AJUDA 0,amarelo,esq_simples,vermelho,0,0,23,79),
        repeat,
          jan_ajuda(Cat,L_coord),
          escolha_ajuda(L_coord,Cut_true,True_fail,Opcao),
          Cut_true,
          True_fail,
          esc_cat_mostra_ex(Cat,Opcao).


esc_cat_mostra_ex(Cat,desconhecido):-
        !.

esc_cat_mostra_ex(Cat,mostrar_def):-
        mostra_os_exemplos(Cat,Cut_true,True_fail),
        Cut_true,
        True_fail.

esc_cat_mostra_ex(Cat,Opcao):-
        repeat,
          escolhe_cat(Cat_esc,Cut_true,True_fail),
          Cut_true,
          True_fail,
          mostra_os_exemplos(Cat_esc,Cut_true1,True_fail1),
          Cut_true1,
          True_fail1.

jan_ajuda(Cat,L_coord):-
        janela(0 Para o ajudar posso : 0,amarelo,
                          esq_simples,vermelho,2,1,11,54),
        tmove(4,5),
        ( var(Cat),
        write_attr(0Mostrar palavras de
                          categoria a selecionar0,normal),
        Dim = 42,
        Lista = [@,@,@,0Desistir de tentar
                               identificar a palavra0,#,
                      0(Assumo que ela e'
                        de categoria "desconhecida")0], !
        ;
        write_attr(0Mostrar palavras da categoria : 0,normal),
        converte(Str,Cat),
        write_attr(Str,normal),
        string_length(Str,Len),
        Dim is 32 + Len,
        Lista = [@,0Mostrar palavras de categoria a selecionar0,
                 @,0Desistir de tentar identificar a palavra0,#,
                 0(Assumo que ela e'
                   de categoria "desconhecida")0],
        ! ),
        poe_menu(Lista,4,5,L_c),
        concat([(4,4,Dim)],L_c,L_coord), !.
```

```
escolha_ajuda(L_c,Cut_true,True_fail,Opcao):-
        poe_jan_rectifica(sim,Coor_aux),
        concat(L_c,Coor_aux,L_coord),
        faz_a_escolha(L_coord,Op),
        indexa(Op,L_coord,1,Index),
        length(L_coord,Len),
        (
        Len == 3,
        opcoes(aj3,Index,Opcao)
        ;
        opcoes(aj4,Index,Opcao) ),
        !,
        (  (
           Opcao == mostrar_indef
           ;
           Opcao == mostrar_def
           ),
        Cut_true = true, True_fail = true
        ;
        Opcao == desconhecido, Cut_true = true, True_fail = true
        ;
        Cut_true = !, True_fail = fail
        ), !.

escolhe_cat(Cat,Cut_true,True_fail):-
        var(Cat),
        janela($ CATEGORIAS : $,amarelo,
                    esq_simples,vermelho,2,55,12,78),
        Lista = [$Nome proprio$,@,$Nome comum$,@,$Adjectivo$,@,
                 $Verbo transitivo$],
        poe_menu(Lista,3,59,L_coord),
        poe_jan_rectifica(sim,Coor_aux),
        concat(L_coord,Coor_aux,L_coordres),
        faz_a_escolha(L_coordres,Coordres),
        indexa(Coordres,L_coordres,1,Index),
        opcoes(categ,Index,Cat),
        !,
        ( Cat == rectifica,
        Cut_true = !, True_fail = fail
        ;
        Cut_true = true, True_fail = true
        ), !.

escolhe_cat(_,!,true):-
        !.

mostra_os_exemplos(Cat,Cut_true,True_fail):-
        janela($ EXEMPLOS : $,amarelo,
                          esq_dupla,vermelho,14,27,23,52),
        tmove(15,30),
        mostra_exemp(Cat,Cut_true,True_fail),
        !.
```

```
mostra_exemp(Cat,Cut_true,True_fail):-
        junta_dic(Cat,Dic_cat),
        (
code_world(_,def),
        current_predicate(Dic_cat/Nargs),
code_world(_,ecran),
        functor(Cabeca,Dic_cat,Nargs),
        N is Nargs - 1,
        findall(Pal,exemp_pal(Cabeca,Pal,N),L_pal)
        ;
        L_pal = [] ),
        !,
        escreve_janela(L_pal,22,30,Cut_true,True_fail),
        !.

exemp_pal(Cabeca,Pal,N):-
        code_world(_,def),
        clause(Cabeca,true),

code_world(_,ecran),
        arg(N,Cabeca,[Pal|_]).

/*
 O predicado "junta_dic" encontra-se aqui repetido por razoes  de
 eficiencia
*/

junta_dic(Nome,Dic_nome):-
        name(dic_,L1),
        name(Nome,L2),
        concat(L1,L2,L3),
        name(Dic_nome,L3).

escreve_janela([],Lininf,Col,Cut_true,True_fail):-
        tmove(22,21),
        write_attr($NAO CONHECO
                    MAIS PALAVRAS DA CATEGORIA$,rev_vid),
        tscroll(0,(14,55),(20,78)),
        janela($$,sem_sentido,esq_dupla,vermelho,14,56,18,78),
        Lista = [$Outra categoria$,@,$Outra ajuda$],
        poe_menu(Lista,14,60,L_coord),
        faz_a_escolha(L_coord,Coord),
        indexa(Coord,L_coord,1,Index),
        opcoes(janvaz,Index,Opcao),
        (
        Opcao == outracat,
        Cut_true = true, True_fail = fail
        ;
        Cut_true = !, True_fail = fail
        ),
        !.
```

```
escreve_janela([Pal|T],Lininf,Col,Cut_true,True_fail):-
        tget(L,C),
        L @< Lininf,
        write_attr(Pal,normal),
        L1 is L + 1,
        tmove(L1,C),
        !,
        escreve_janela(T,Lininf,Col,Cut_true,True_fail).

escreve_janela([Pal|T],Lininf,Col,Cut_true,True_fail):-
        janela($$,sem_sentido,esq_dupla,vermelho,14,56,20,78),
        Lista = [$Mais exemplos$,@,$Outra categoria$,
                 @,$Outra ajuda$],
        poe_menu(Lista,14,60,L_coord),
        faz_a_escolha(L_coord,Coord),
        indexa(Coord,L_coord,1,Index),
        opcoes(jancheia,Index,Opcao),
        (
        Opcao == outracat,
        Cut_true = true, True_fail = fail, !
        ;
        Opcao == outraaju,
        Cut_true = !, True_fail = fail, !
        ;
        !,
        tscroll(1,(15,28),(22,51)),
        tmove(21,Col),
        write_attr(Pal,normal),
        tmove(Lininf,Col),
        escreve_janela(T,Lininf,Col,Cut_true,True_fail)
        ).


poe_jan_rectifica(sim,Coor_aux):-
        janela($$,sem_sentido,esq_dupla,vermelho,21,18,23,61),
        poe_menu([$Quero rectificar a resposta anterior$],
                                              21,22,Coor_aux),
        !.

poe_jan_rectifica(nao,Coor_aux):-
        !.

jan_confirm(Opcao3,Coor_aux):-
        janela($ CONFIRMACAO $,vermelho,esq_dupla,vermelho,
                14,18,20,61),
        Lista = [@,$Os dados fornecidos estao correctos$,@,
                 $Ha' incorreccoes nos dados fornecidos$],
        poe_menu(Lista,14,22,L_c),
        concat(L_c,Coor_aux,L_coord),
        faz_a_escolha(L_coord,Op3),
        indexa(Op3,L_coord,1,Index),
        opcoes(op3,Index,Opcao3), !.
```

```
poe_menu([],_,_,[]):-
        !.


poe_menu([X|T],L,C,L_coord):-
        L1 is L + 1,
        tmove(L1,C),
        (
        X == @,
        Resto = T
        ;
        X == #,
        [X1|Resto] = T,
        write_attr(X1,normal)
        ),
        !,
        poe_menu(Resto,L1,C,L_coord).

poe_menu([X|T],L,C,[(Lin,C1,Dim)|T1]):-
        Lin is L + 1,
        C1 is C - 1,
        string_length(X,Dim),
        tmove(Lin,C),
        write_attr(X,normal),
        !,
        poe_menu(T,Lin,C,T1).


faz_a_escolha(L_coord,(L,C,Dim)):-
        [(A,B,_)|_] = L_coord,
        tmove(A,B),
        repeat,
          pertence((L,C,Dim),L_coord),
          tira_poe_rev_vid(tira,Dim),
          tmove(L,C),
          tira_poe_rev_vid(poe,Dim),
          analisa_input,
          !.


analisa_input:-
        repeat,
          keyb(_,Key),
          (
          Key == 28,
          Cut_true = !,
          True_fail = true
          ;
          Key == 80,
          Cut_true = !,
          True_fail = fail
          ;
```

```
            tget(L,C),
            put(7),
            tmove(L,C),
            Cut_true = true,
            True_fail = fail
            ),
          Cut_true,
          True_fail,
          abolish(dim_anterior/1).


tira_poe_rev_vid(Tira_poe,Dim):-
          tget(L,C),
          C_ant is C - 1,
          tmove(L,C_ant),
          (
          Tira_poe = tira,
          put(00),
          atributo(normal,Cod),
             (
             retract(dim_anterior(Dant)),
             Dim1 = Dant
             ;
             Dim1 = Dim
             )
          ;
          wca(1,175,26),
          atributo(rev_vid,Cod),
          asserta(dim_anterior(Dim)),
          Dim1 = Dim
          ),
          !,
          C1 is C + 1,
          tmove(L,C1),
          wa(Dim1,Cod),
          tmove(L,C),
          wa(1,17),
          !.


janela(Cab,Cor_cab,Tipo_esq,Cor_esquadria,L1,C1,L2,C2):-
          tmove(L2,C1),
          wa(1,31),
          tscroll(0,(L1,C1),(L2,C2)),
          W is C2 - C1,
          W1 is W // 2,
          atom_string(Atom,Cab),
          name(Atom,Aux),
          length(Aux,N),
          A1 is N // 2,
          A is W1 - A1,
          A2 is C1 + A,
```

```
        cor(Cor_esquadria,Cor_esq),
        esq_horiz(Tipo_esq,Esq_horiz),
        esq_vert(Tipo_esq,Esq_vert),
        canto_sup_esq(Tipo_esq,Canto_sup_esq),
        canto_sup_dir(Tipo_esq,Canto_sup_dir),
        canto_inf_esq(Tipo_esq,Canto_inf_esq),
        canto_inf_dir(Tipo_esq,Canto_inf_dir),
        !,
        reth(Esq_horiz,L1,C1,W,Cor_esq),
        canto(Canto_sup_dir,L1,C2,Cor_esq),
        L is L1 + 1,
        retv(Esq_vert,L,C2,L2,Cor_esq),
        canto(Canto_sup_esq,L1,C1,Cor_esq),
        retv(Esq_vert,L,C1,L2,Cor_esq),
        canto(Canto_inf_esq,L2,C1,Cor_esq),
        C is C1 + 1,
        reth(Esq_horiz,L2,C,W,Cor_esq),
        canto(Canto_inf_dir,L2,C2,Cor_esq),
        (
        Cab \= $$,
        tmove(L1,A2),
        write_attr(Cab,Cor_cab)
        ;
        true
        ),
        !.


reth(Esq_horiz,L,C,W,Cor_esq):-
        tmove(L,C),
        wca(W,Esq_horiz,Cor_esq).


retv(_,L,_,L,_):-
        !.

retv(Esq_vert,L,C2,L2,Cor_esq):-
        tmove(L,C2),
        wca(1,Esq_vert,Cor_esq),
        Aux is L + 1,
        !,
        retv(Esq_vert,Aux,C2,L2,Cor_esq).


canto(Tipo_canto,L,C,Cor_esq):-
        tmove(L,C),
        wca(1,Tipo_canto,Cor_esq).
```

```
write_attr(Pal,Attr):-
        (
        atributo(Attr,Codigo)
        ;
        cor(Attr,Codigo)
        ),
        (
        name(Pal,L)
        ;
        atom_string(Atom,Pal),
        name(Atom,L)
        ),
        length(L,N),
        write(Pal),
        tget(Lin,Col),
        Col1 is Col - N,
        tmove(Lin,Col1),
        wa(N,Codigo),
        tmove(Lin,Col),
        !.

write_attr(Pal,Attr):-
        cls,
        write($Nao conheco o codigo para o atributo pretendido$),
        nl.


indexa(X,[X|_],I,I):-
        !.

indexa(X,[_|T],N,I):-
        N1 is N + 1,
        !,
        indexa(X,T,N1,I).


leitor_lim(L_pal,Lim):-
        tget(L,C),
        repeat,
          tmove(L,C),
          read_string(Lim,Str),
          code_world(_,leitor),
          constroi_listap(Str,L_pal),
          code_world(_,ecran),
        !.

opcoes(1,1,sim).
opcoes(1,2,nao).
opcoes(1,3,nao_sei).
opcoes(1,4,engano).
opcoes(2,1,nome_prop).
opcoes(2,2,nome_com).
```

```
opcoes(2,3,adj).
opcoes(2,4,verb_trans).
opcoes(2,5,ajuda).
opcoes(2,6,rectifica).
opcoes(op3,1,correcto).
opcoes(op3,2,incorrecto).
opcoes(op3,3,Op):-
        opcoes(2,6,Op).
opcoes(aj3,1,mostrar_indef).
opcoes(aj3,2,desconhecido).
opcoes(aj3,3,Op):-
        opcoes(2,6,Op).
opcoes(aj4,1,mostrar_def).
opcoes(aj4,2,mostrar_indef).
opcoes(aj4,3,desconhecido).
opcoes(aj4,4,Op):-
        opcoes(2,6,Op).
opcoes(categ,5,Cat):-
        opcoes(2,6,Cat).
opcoes(categ,Index,Cat):-
        opcoes(2,Index,Cat).
opcoes(janvaz,1,outracat).
opcoes(janvaz,2,outraaju).
opcoes(jancheia,1,mais_ex).
opcoes(jancheia,2,outracat).
opcoes(jancheia,3,outraaju).


atributo(normal,31).
atributo(invisivel,17).
atributo(rev_vid,26).
atributo(piscar,159).
atributo(piscar_rev_vid,249).

cor(vermelho,29).
cor(amarelo,105).
cor(verde,26).
cor(verde_preto,10).

esq_horiz(esq_dupla,205).
esq_horiz(esq_simples,196).
esq_vert(esq_dupla,186).
esq_vert(esq_simples,179).

canto_sup_esq(esq_dupla,201).
canto_sup_esq(esq_simples,218).

canto_sup_dir(esq_dupla,187).
canto_sup_dir(esq_simples,191).

canto_inf_esq(esq_dupla,200).
canto_inf_esq(esq_simples,192).
```

```
canto_inf_dir(esq_dupla,188).
canto_inf_dir(esq_simples,217).

converte($nome proprio$,nome_prop).
converte($nome comum$,nome_com).
converte($adjectivo$,adj).
converte($verbo transitivo$,verb_trans).
converte($"desconhecido"$,desconhecido).

atribui_cat(Cat,Cat):-
        classe_aberta(Lista),
        pertence(Cat,Lista),
        !.

atribui_cat(_,desconhecido):-
        !.

/*
  Os  predicados "pertence",  "concat" e "classe_aberta" estao de
  novo aqui definidos por razoes de eficiencia.
*/

pertence(X,[X|_]).

pertence(X,[_|T]):-
        pertence(X,T).

concat([],L,L).
concat([A|T],L,[A|T1]):-
        concat(T,L,T1).

classe_aberta([nome_com , nome_prop ,
                   adj , verb_trans , desconhecido]).

/*
  Gestao  de  ecran  durante a interacco com  o  utilizador  para
  correccao de um possivel erro ortografico.
*/

inter_accao_correc(Pal_incorrec,Pal_correc,Resto_frase):-
        tmove(23,0),
        wa(1,31),
        tscroll(0,(0,0),(23,79)),
        frase_escrita(Frase),
        jan_correc_erros(Frase,Pal_incorrec,Pal_correc,
                                        Opcao,Resto_frase),
        cls,
        !,
        Opcao == sim,
        !.
```

```
jan_correc_erros(Frase,Pal_incorrec,Pal_correc,Opcao,Resto_frase):-
        janela($ DETECCAO E CORRECCAO DE ERROS ORTOGRAFICOS $,
                        amarelo,esq_simples,vermelho,0,0,23,79),
        uso_do_teclado(1),
        tmove(2,2),
        write_attr($Frase a analisar :$,normal),
        tmove(3,2),
        write_attr($>> $,normal),
        esc_lista(Frase,Resto_frase),
        janela($ HIPOTESE : $,amarelo,
                        esq_dupla,vermelho,5,2,9,30),
        tmove(6,4),
        write_attr($Houve um erro ortografico$,normal),
        tmove(7,4),
        write_attr($na escrita da palavra :$,normal),
        tmove(8,4),
        write_attr($-> $,normal),
        write_attr(Pal_incorrec,piscar),
        janela($ PROPOSTA DE CORRECCAO : $,amarelo,esq_dupla,
                vermelho,11,25,16,60),
        tmove(13,27),
        write_attr($Palavra incorrecta : $,normal),
        write_attr(Pal_incorrec,normal),
        tmove(14,27),
        write_attr($Palavra corrigida  : $,normal),
        write_attr(Pal_correc,rev_vid),
        janela($ QUESTAO $,rev_vid,
                        esq_simples,vermelho,18,2,23,53),
        tmove(20,4),
        write_attr($Posso admitir a "HIPOTESE" como
                certa e corrigir$,normal),
        tmove(21,4),
        write_attr($o erro de acordo com a
                        "PROPOSTA DE CORRECCAO ?$,normal),
        janela($ RESPOSTA :$,amarelo,
                        esq_dupla,vermelho,17,64,23,77),
        poe_menu([@,$Sim$,@,$Nao$],17,69,L_coord),
        faz_a_escolha(L_coord,Coord),
        indexa(Coord,L_coord,1,Index),
        (
        Index == 1,
        Opcao = sim
        ;
        Opcao = nao
        ),
        !.

esc_lista(Frase,Resto_frase):-
        concat(L_aux,Resto_frase,Frase),
        conta_ate_ao_ultimo(0,L_aux,Cont),
        imprime_a_lista(Frase,Cont,0),
        !.
```

```
conta_ate_ao_ultimo(Cont,[],Cont):-
        !.

conta_ate_ao_ultimo(Cont,[Pal],Aux):-
        Cont1 is Cont + 1,
        conta_ate_ao_ultimo(Cont1,[],Aux).

conta_ate_ao_ultimo(Cont,[Pal|T],Aux):-
        Cont1 is Cont + 1,
        conta_ate_ao_ultimo(Cont1,T,Aux).

imprime_a_lista([],_,_):-
        !.

imprime_a_lista([Pal|T],Cont,Cont1):-
        Cont2 is Cont1 + 1,
        Cont == Cont2,
        write_attr(Pal,piscar),
        write_attr($ $,normal),
        imprime_a_lista(T,Cont,Cont2).

imprime_a_lista([Pal|T],Cont,Cont1):-
        write_attr(Pal,normal),
        write_attr($ $,normal),
        Cont2 is Cont1 + 1,
        imprime_a_lista(T,Cont,Cont2).

/*    Suporte computacional para detecção e correcção de erros
      ortográficos.

      Deteccao e correccao de erros ortograficos.
*/

tenta_corrigir(_,[],[]):-
        !.

tenta_corrigir(_,_,_):-
        reconhecimento_livre,
        !, fail.

tenta_corrigir(Categ,[Pal|T],T):-
        name(Categ,[_,_,_,_|Lcat]),
        name(Cat,Lcat),
        analise_em_exclusao(Cat),
        !.

tenta_corrigir(Categ,[Pal|_],_):-
        nao_tenta_corrigir(Lista),
        pertence(Pal,Lista),
        !, fail.
```

```
tenta_corrigir(Categ,[Pal|_],_):-
        ja_houve_correccao(Lista),
        pertence(Pal,Lista),
        !.

tenta_corrigir(Categ,[Pal|T],T):-
        procura_palavras(Categ),
        name(Pal,Lista),
        !,
        lex(Pal_correc,Lista),
        !,
        aceita_correccao(Pal,Pal_correc,T),
        !.


procura_palavras(Categ):-
        current_predicate(Categ/Args),
        functor(Termo,Categ,Args),
        clause(Termo,Corpo),
        Aux is Args - 1,
        arg(Aux,Termo,[Pal_categ|_]),
        nonvar(Pal_categ),
        memoriza(Pal_categ).

procura_palavras(_).

lex(X,P1):-
        lista(P),
        search_similar_word(P1,P),
        abolish(lista/1),
        nl,
        name(X,P),
        !.

lex(_,_):-
        abolish(lista/1),
        !, fail.


aceita_correccao(Pal_incorrec,Pal_correc,Resto_frase):-
        code_world(_,ecran),
        inter_accao_correc(Pal_incorrec,Pal_correc,Resto_frase),
        code_world(_,def),
        (
        retract(ja_houve_correccao(Lista)),
        concat([Pal_incorrec],Lista,L1),
        asserta(ja_houve_correccao(L1))
        ;
        asserta(ja_houve_correccao([Pal_incorrec]))
        ),
        !.
```

```
aceita_correccao(Pal_incorrec,_,_):-
        (
        retract(nao_tenta_corrigir(Lista)),
        concat([Pal_incorrec],Lista,L1),
        asserta(nao_tenta_corrigir(L1))
        ;
        asserta(nao_tenta_corrigir([Pal_incorrec]))
        ),
        !, fail.

memoriza(H):-
        name(H,L),
        assertz(lista(L)),
        !, fail.

search_similar_word([A|L1],[A|P2]):-
        lista([A|X]),
        sort([A|X],L4),
        sort([A|L1],L3),
        sublista(L3,L4,C,[A|L1],[A|X]),
        sublista(L4,L3,C1,[A|L1],[A|X]),
        !,
        equal(X,P2).

search_similar_word([A|L1],[A1|L1]):-
        lista([A1|L1]).

search_similar_word(P,[A|P]):-
        lista([A|P]).

search_similar_word([A|L1],L1):-
        lista(L1).

sublista([],[],C,[A|L1],[A|P2]):-
        var(C), !
        ;
        c_e_c([A|L1],[A|P2],K,KK),
        (
        nonvar(KK)
        ;
        fail
        ),
        KK >= 2,
        !.

sublista(T,[],C,[A|L1],[A|P2]):-
        var(C), !
        ;
        c_e_c([A|L1],[A|P2],K,KK),
        (
        nonvar(KK)
        ;
```

```
        fail
        ),
        KK >= 2,
        !.

sublista([],U,C,[A|L1],[A|P2]):-
        var(C), !
        ;
        c_e_c([A|L1],[A|P2],K,KK),
        (
        nonvar(KK)
        ;
        fail
        ),
        KK >= 2, !.

sublista([H|T],U,C,[A|L1],[A|P2]):-
        member_a(H,U),
        !,
        sublista(T,U,C,[A|L1],[A|P2]).

sublista([H|T],U,C,[A|L1],[A|P2]):-
        (
        var(C),
        C=1
        ;
        true
        ),
        (
        C1 is C+1, !,
        C1 == 2, !,
        sublista(T,U,C1,[A|L1],[A|P2])
        ;
        fail
        ).

c_e_c([],[],K,KK):-
        K >= 2,
        equal(K,KK),
        !.

c_e_c(L1,[],K,KK):-
        K >= 2,
        equal(K,KK),
        !.

c_e_c([],P2,K,KK):-
        K >= 2,
        equal(K,KK),
        !.
```

```
c_e_c([A|L1],[B|P2],K,KK):-
        (
        var(K),
        K=0, !
        ;
        true
        ),
        A == B,
        K1 is K + 1,
        c_e_c(L1,P2,K1,KK),
        !.

c_e_c([A|L1],[B|P2],K,KK):-
        K >= 2,
        equal(K,KK),
        !.

c_e_c([A|L1],[B|P2],K,KK):-
        c_e_c(L1,P2,K,KK),
        !.

c_e_c([A|L1],[B|P2],K,KK):-
        c_e_c(L1,[B|P2],K,KK),
        !.

c_e_c([A|L1],[B|P2],K,KK):-
        c_e_c([A|L1],P2,K,KK),
        !.

concat([],L,L).

concat([A|T],L,[A|T1]):-
        concat(T,L,T1).

equal(X,X).

member_a(H,[H|_]):-!.

member_a(I,[_|T]):-
        member_a(I,T).

/*    Suporte computacional para a construção da árvore de
      invocações.

      Construcao da arvore de invocacoes.
*/

const_arvore_invocacoes:-
        repeat,
          cls,
          tira_todas_reg_corrente([],L_reg_corrente),
          tira_todas_invocado_por([],L_invocado_por),
```

```
        constroi_arvore_invs(L_reg_corrente,
                             L_invocado_por,L_nomes),
        renomear(L_nomes,a,[],L_nom_invoc,L_inv),
        renomear(L_nom_invoc,[],[sint_nom,sint_compls],L_muda),
        [Inv -> C | T] = L_inv,
        dimensao(C,Dim_c),
        const_coord(T,(1,1),(20,70),(1,1),
              1,Dim_c,[(Inv -> C,(1,1))],L_coord_invert),
        inverte_lista(L_coord_invert,[],L_coord),
        [(Inv -> C,(1,1))|T1] = L_coord,
        tmove(1,0),
        put(26), write(Inv), write($($), write(C), write($)$),
        desenha_arvore(T1,(1,1),(20,70),(1,1),
                                   1,Dim_c,L_muda,26),
        inverte_lista(L_nom_invoc,[],Lp_nom_invoc),
        nl, nl, write(Lp_nom_invoc).

tira_todas_reg_corrente(L,L1):-
        retract(reg_corrente(Regra)),
        !,
        tira_todas_reg_corrente([Regra|L],L1).

tira_todas_reg_corrente(L,L):-
        !.

tira_todas_invocado_por(L,L1):-
        retract(invocado_por(Regra,Id)),
        !,
        tira_todas_invocado_por([(Regra,Id)|L],L1).

tira_todas_invocado_por(L,L):-
        !.

constroi_arvore_invs([],[],[]):-
        !.

constroi_arvore_invs([R,repete,R1|T],[(R1,N)|T1],
                                   [R->'*' , R1<-N|T2]):-
        constroi_arvore_invs(T,T1,T2).

constroi_arvore_invs([R,repete_vazio,R1|T],
                        [(R1,N)|T1],[R->[] , R1<-N|T2]):-
        constroi_arvore_invs(T,T1,T2).

constroi_arvore_invs([R|T],[(R,N)|T1],[R->N|T2]):-
        constroi_arvore_invs(T,T1,T2).

constroi_arvore_invs([R|T],[(R1,N)|T1],[R->'*' , R1<-N|T2]):-
        R \== R1,
        constroi_arvore_invs(T,T1,T2).
```

```
constroi_arvore_invs(_,_,[]):-
        write($houve bronca !$),
        read(_),
        !.

/*
 Clausula "renomear"
*/

renomear([],_,L,L,[]).

renomear([A|T],Prox_inv,L,L1,[B|T1]):-
        (
        Nome -> C = A
        ;
        Nome <- C = A
        ),
        (
        pertence(Nome -> Inv,L),
        L_aux = L,
        Prox_inv1 = Prox_inv
        ;
        Inv = Prox_inv,
        concat([Nome -> Inv],L,L_aux),
        define_prox_inv(Inv,Prox_inv1)
        ),
        (
        pertence(Nome,L_muda_aux),
        concat(Lm,[C],Lm_aux)
        ;
        Lm_aux = Lm ),
        ( N1 -> C1 = A,
        B = Inv -> C
        ;
        N1 <- C1 = A,
        B = Inv <- C ),
        !,
        renomear(T,Prox_inv1,L_aux,L1,T1).


renomear(_,L,[],L):-
        !.

renomear(L_nomes,L,[Nome|T],L1):-
        (
        pertence(Nome -> Inv,L_nomes),
        concat(L,[Inv],L_aux)
        ;
        L_aux = L ),
        !,
        renomear(L_nomes,L_aux,T,L1).
```

```
define_prox_inv(Inv,Prox_inv):-
        name(Inv,L),
        inverte_lista(L,[],L1),
        nova_combinacao(L1,L_nova_comb),
        inverte_lista(L_nova_comb,[],L_res),
        name(Prox_inv,L_res),
        !.

nova_combinacao([Z],[A,A]):-
        name(z,[Z]),
        name(a,[A]), !.

nova_combinacao([S],[S1]):-
        S1 is S + 1,
        !.

nova_combinacao([Z|T],[A|T1]):-
        name(z,[Z]),
        name(a,[A]),
        !,
        nova_combinacao(T,T1).

nova_combinacao([S|T],[S1|T]):-
        S1 is S + 1,
        !.

const_coord([],_,_,_,_,_,L,L):-
        !.

const_coord([Prim|T],(Li,Ci),(Lf,Cf),
                        (Lant,Cant),D_inv,D_id,Lista,L1):-
        (
        Inv -> Id = Prim,
        dimensao(Inv,D_inv1),
        dimensao(Id,D_id1),
        Aux is D_inv1 + D_id + Cant + 2,
                (
                Aux < Cf,
                Cant1 is Aux + 1,
                Lant1 = Lant
                ;
                Cant1 is Ci + 2;
                Lant1 is Lant + 1
                )
        ;
        Inv <- Id = Prim,
        dimensao(Inv,D_inv1),
        dimensao(Id,D_id1),
        pertence((Inv -> _,(Laux,Cant1)),Lista),
                (
                Cant < Cant1,
                Lant1 is Lant + 3
```

```
                ;
                Lant1 is Lant + 2 )
        ),
        concat([(Prim,(Lant1,Cant1))],Lista,L_res),
        !,
        const_coord(T,(Li,Ci),(Lf,Cf),(Lant1,Cant1),
                                    D_inv1,D_id1,L_res,L1).

inverte_lista([],L,L).

inverte_lista([A|T],L,L1):-
        inverte_lista(T,[A|L],L1).

dimensao(Nome,Dim):-
        name(Nome,Lista),
        length(Lista,Dim).

desenha_arvore([],_,_,_,_,_,_,_):-
        !.

desenha_arvore([Prim|T],(Li,Ci),(Lf,Cf),(Lant,Cant),
                        D_inv,D_id,L_muda,Tipo_seta):-
        (
        (Inv -> Id,(L,C)) = Prim,
        C1 is D_inv + D_id + Cant + 2,
        Seta = Tipo_seta,
        escreve_seta_frente((Lant,C1),(L,C),Ci,Cf,Seta),
        escreve_inv(Inv,Id)
        ;
        (Inv <- Id,(L,C)) = Prim,
                (
                pertence(Inv,L_muda),
                Seta is 26 - Tipo_seta + 16
                ;
                Seta = Tipo_seta
                ),
        C1 is D_inv + D_id + Cant + 2,
        Lant1 is Lant + 1,
        escreve_seta_baixo(Lant,Lant1,C1),
        C2 is C1 - 1,
        escreve_seta_tras((Lant1,C2),(L,C),Ci,Cf),
        escreve_inv(Inv,Id)
        ),
        dimensao(Inv,D_inv1),
        dimensao(Id,D_id1),
        !,
        desenha_arvore(T,(Li,Ci),(Lf,Cf),(L,C),
                        D_inv1,D_id1,L_muda,Seta).

escreve_inv(Inv,Id):-
        write(Inv), write($($),
        write(Id), write($)$).
```

```
escreve_seta_frente((L,C),(L,C),_,_,_):-
        !.

escreve_seta_frente((Lant,C1),(L,C),Ci,Cf,Seta):-
        tmove(Lant,C1),
        put(Seta),
        (
        C1 == Cf,
        C1_aux is Ci + 1,
        Lant_aux is Lant + 1
        ;
        C1_aux is C1 + 1,
        Lant_aux = Lant
        ),
        escreve_seta_frente((Lant_aux,C1_aux),(L,C),
                                              Ci,Cf,Seta).

escreve_seta_baixo(Lant,Lant1,C1):-
        tmove(Lant,C1),
        put(25),
        tmove(Lant1,C1),
        put(25).

escreve_seta_tras((Lant,C),(L,C),_,_):-
        tmove(Lant,C),
        put(25),
        tmove(L,C),
        !.

escreve_seta_tras((Lant,C2),(L,C),Ci,Cf):-
        tmove(Lant,C2),
        put(45),
        (
        C2 == Ci,
        C2_aux = Cf,
        Lant_aux is Lant + 1
        ;
        C2_aux is C2 - 1,
        Lant_aux = Lant
        ),
        !,
        escreve_seta_tras((Lant_aux,C2_aux),(L,C),Ci,Cf).
```

```
/*
     Suporte computacional para a execução de objectivos de
     âmbito geral.
     Objectivos de ambito geral.
     Leitor de expressoes.
*/

leitor(L_pal):-
        tget(L,C),
        repeat,
          tmove(L,C),
          le_frase(L_pal),
        !.

le_frase(L_pal):-
        read_line(0,Str),
        constroi_listap(Str,L_pal).

constroi_listap(Str,L_pal):-
        list_text(L_car,Str),
        tira_redundante(L_car,L_car1),
        asserta(l_a(L_car1)),
        findall(Pal,lcar_lpal(Pal),L_pal),
        !,
        L_pal \== [''].

lcar_lpal(Pal):-
        continua,
          retract(l_a(L)),
          encontra_pal(L,L_pal,L_pal1),
          name(Pal,L_pal),
          tira_redundante(L_pal1,Resto),
          junta_bd(Resto).

continua.

continua:-
        l_a(_),
        continua.

encontra_pal([],[],[]):-
        !.

encontra_pal([C|T],[Min|T1],L):-
        not(caracter_redundante(C)),
        maiuscula_minuscula(C,Min),
        !,
        encontra_pal(T,T1,L).

encontra_pal([_|T],[],T):-
        !.
```

```
tira_redundante([],[]):-
        !.

tira_redundante([A|T],L):-
        caracter_redundante(A),
        !,
        tira_redundante(T,L).

tira_redundante(L,L):-
        !.

junta_bd([]):-
        !.

junta_bd(Resto):-
        asserta(l_a(Resto)),
        !.

caracter_redundante(32).

maiuscula_minuscula(C,Min):-
        C > 64, C < 91,
        Min is C + 32,
        !.

maiuscula_minuscula(C,C):-
        !.

apaga(_,[],[]).

apaga(A,[A|T],T).

apaga(A,[B|T],[B|T1]):-
        apaga(A,T,T1).

concat([],L,L).

concat([A|T],L,[A|T1]):-
        concat(T,L,T1).

uniao([],L,L):-
        !.

uniao([A|T],L,[A|T1]):-
        not(pertence(A,L)),
        !,
        uniao(T,L,T1).

uniao([_|T],L,L1):-
        uniao(T,L,L1).

pertence(A,[A|_]).
```

```
pertence(A,[_|T]):-
        pertence(A,T).

diferenca_conj([],_,[]).

diferenca_conj([A|T],L,L1):-
        pertence(A,L),
        !,
        diferenca_conj(T,L,L1).

diferenca_conj([A|T],L,[A|T1]):-
        !,
        diferenca_conj(T,L,T1).

ultimo([A],A).

ultimo([_|T],A):-
        !,
        ultimo(T,A).
```