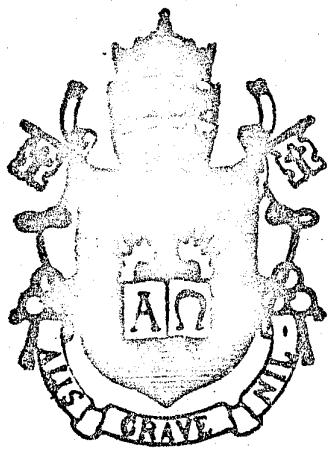

PUC

SÉRIE : Monografias em Ciência da Computação
Nº18/89

HIER - Highly Interactive Expert Resolver: Uma Ferramenta para Sistemas Especialistas

Marco Antonio Mortari Rezende

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP-22453
RIO DE JANEIRO - BRASIL

Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática

SÉRIE : Monografias em Ciência da Computação, 18/89

Editor: Paulo A. S. Veloso outubro, 1989

HIER - Highly Interactive Expert Resolver: Uma Ferramenta para Sistemas Especialistas

Marco Antonio Mortari Rezende

PUC/RIO

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.: (021) 529-9386 TELEX: 31078 FAX: (021) 274-4546
BITNET: userrtlc@lncc.bitnet

Resumo

A implementação de Sistemas Especialistas exige ambientes capazes de prover o alto nível de abstração requerido pela natureza das aplicações para as quais estão voltados. Com o advento das linguagens descritivas, fornecer tal abstração tornou-se tarefa menos árdua. Por outro lado, muitos recursos e facilidades desejados ainda não se encontram diretamente disponíveis. Este trabalho descreve uma ferramenta auxiliar - HIER - almejando suprir algumas dessas funções, tais como, consulta ao usuário, manutenção de restrições de integridade e explicações. Após uma breve introdução onde é descrito o contexto onde os sistemas especialistas se enquadram, juntamente com suas características e componentes principais, alguns sistemas já existentes são analisados. A seguir, detalha-se o protótipo do **Highly Interactive Expert Resolver**. No apêndice, apresenta-se a relação dos comandos disponíveis e regras sintáticas, exemplos de utilização e o código em Prolog do programa implementado.

Palavras-chave: sistemas especialistas, programação lógica, Prolog.

Abstract

The implementation of Expert Systems demands environments providing a high level of abstraction, as required by the nature of the applications involved. With the appearance of descriptive languages, such an abstraction has become less hard to offer. However, many desired resources and facilities are not promptly available yet. This work describes an auxiliary tool - HIER - to supply some of these functions, namely "query-the-user", maintenance of integrity constraints and explanations. Following a brief introduction describing the context of Expert Systems together with their characteristics and main components, some already existing systems are outlined. Next, the Highly Interactive Expert Resolver is detailed. In the appendix, a list of available commands and syntax rules, some examples of use and the Prolog code are presented.

Keywords: expert systems, logic programming, Prolog.

Sumário

1. Introdução

Cientes do sucesso alcançado pelos computadores nas aplicações empresariais e tecnológicas, especialistas de outras áreas da ciência ambicionaram introduzir tais máquinas na resolução de seus próprios problemas. Pensou-se em inaugurar uma certa automação na diagnose médica, aplicação de leis, prospecção mineral, &c. Tal ambição estimulou o desenvolvimento de sistemas capazes de lidar com um conhecimento altamente especializado e, ao mesmo tempo, incompleto e mesmo impreciso. Justamente devido a isso, os **Sistemas Especialistas** (SEs) apresentam características e comportamentos diferentes dos programas tradicionais para resolução de equações diferenciais ou gerenciamento de operações bancárias, por exemplo. Exemplos de SEs pioneiros incluem MYCIN (para diagnósticos médicos), DENDRAL (interpretações de espectogramas de massa) e PROSPECTOR (exploração mineral).

A primeira dificuldade relaciona-se a encontrar uma representação adequada para o conhecimento especializado. A próxima tarefa consiste em prover meios de aplicar tal conhecimento na solução de problemas simulando, de certa forma, o raciocínio humano.

Sumariando as principais características e recursos dos SEs tem-se:

- **simplicidade de uso.** Por lidar diretamente com pessoas não specializadas e dificilmente possuindo conhecimentos de computação, a interação com o usuário deve ser clara e simples e em uma linguagem (próxima a) natural;

- **capacidade de armazenar e manipular uma quantidade muito grande de conhecimento.** Dada a natureza extremamente complexa das aplicações com as quais os SEs estão envolvidos, estes devem ser capazes de lidar com bases de conhecimento de grandes dimensões, sendo muitas vezes necessário usar de heurística para obter uma solução rapidamente;

- capacidade de aprender. Decorre da incompletude do conhecimento armazenado. Durante a interação com o usuário, o sistema deve poder adquirir novos conhecimentos ou mesmo "mudar de opinião" a respeito de suas conclusões. Para tal, é necessária a

- representação do conhecimento de forma modular e expansível, a fim de permitir modificações simples, rápidas e sem maiores conseqüências;

- suporte a fatos e deduções imprecisas. Os especialistas não possuem, muitas vezes, certeza absoluta das conclusões às quais chegam ou porque os dados em que se baseiam não são precisos ou porque não estão totalmente certos se podem inferir determinado resultado mesmo a partir de informações precisas ou ainda por ambos os motivos. É comum, por conseguinte, o emprego de fatores de confiabilidade para dar uma medida da exatidão de um resultado alcançado;

- explicação do raciocínio. Muitas vezes, o usuário deseja ou necessita saber o porquê de uma conclusão ou de determinada pergunta, durante o processo de inferência, ter a ele sido feita. Deseja-se, portanto, que o sistema esteja abilitado a justificar, preferencialmente passo a passo, seu raciocíno.

São três os componentes básicos de um SE:

(i) base de conhecimento. Contém a descrição do conhecimento representado em alguma forma "de mais alto nível" como redes semânticas, regras de produção, frames, &c.

(ii) máquina de inferência. Realiza as deduções a partir da base de conhecimento. Há basicamente dois caminhos a seguir:

- começar pela conclusão e procurar pelas evidências que a justifiquem (backward chaining) ou

- tirar conclusões a partir de certas evidências (forward chaining);

(iii) componente de consulta e explicação. Tem como funções:

- obter, quando necessário e preciso, informações do usuário;
- validar as respostas recebidas;
- expor, quando indagado, os motivos que o levam a consultar o usuário;
- explicar as conclusões alcançadas.

Observa-se que a natureza das aplicações para as quais os SEs estão voltados requer um estilo de programação difícil de ser seguido utilizando-se as linguagens tradicionais como FORTRAN e Pascal. Tais linguagens, ditas prescritivas ou imperativas, constituem-se fundamentalmente em modelos da máquina de von Neumann e não provêm o nível de abstração almejado. É essencial, pois, ter disponível uma ferramenta com a qual o programador possa ser liberado da tarefa de organizar as estruturas de controle e voltar sua atenção para os dados e seus relacionamentos.

Das diversas classes de linguagens é a descritiva (ou declarativa) a que mais se adequa ao desenvolvimento de um SE. Agora, a tarefa do programador se reduz a especificar os relacionamentos entre os objetos de entrada e os de saída; o computador encarrega-se de realizar toda a computação a partir de tais declarações. Para aqueles familiarizados com a verificação formal de programas, pode-se dizer que, na programação descritiva, tudo se passa como se o programador fornecesse apenas as assertivas e o interpretador ou compilador, respectivamente, executasse ou gerasse todo o código correspondente.

As linguagens declarativas permitem implementar uma alta modularidade de maneira mais natural. Um programa, aqui, nada mais é do que uma coleção de definições e cada definição isola algum fragmento pequeno dos demais.

E, dentre as linguagens declarativas, aquelas baseadas em lógica mostram-se extremamente atraentes. Prolog, em particular, destaca-se devido a seu mecanismo de inferência bastante geral baseado em regras. Mais ainda, por ser uma linguagem tipicamente interpretada, provê uma série de facilidades para o desenvolvimento de aplicações interativas. No entanto, certos recursos extremamente desejáveis para o desenvolvimento e uso de SEs ainda não estão presentes nessa linguagem lógica, como facilidades de explicação e consulta.

Surge, naturalmente, a necessidade de construir um ambiente ainda mais poderoso que, além de herdar características e recursos do Prolog, proveja as facilidades acima apresentadas. É nesse contexto que foi projetado o HIER, suas características básicas apresentadas a seguir.

2. Sistemas Existentes

Esta seção resume a seção 2 de [Sen 89].

2.1 apes: Augmented Prolog for Expert Systems

apes estende o interpretador do micro-PROLOG na medida em que provê explicações do raciocínio e gerencia interações com o usuário. Logo, apes pode ser usado, em particular, para a construção de SEs baseados em lógica ou, de forma mais geral, para implementar qualquer programa lógico interativo.

Seus componentes principais são:

- **interpretador**. apes pode executar qualquer programa micro-Prolog desde que não inclua características não-lógicas. O interpretador mantém o rastreio do processo de inferência, que pode ser inspecionado por seu componente de explicação, além de automaticamente ativar o componente interativo quando necessário;

- **componente interativo**: implementa o modelo "consulte-o-usuário" (query-the-user) para programas lógicos interativos. Ele decide que questões colocar ao usuário, armazena respostas fornecidas após passarem por testes de validade e consistência e é capaz de explicar porque uma pergunta foi feita;

- **componente de explicação**: permite ao usuário explorar interativamente provas de respostas encontradas pelo interpretador e os motivos pelos quais uma resposta potencial não foi confirmada;

- **interface com o usuário (front-end)**. Permite editar, listar, executar e tranferir programas de e para arquivos externos.

Para prover explicações, apes mantém o rastreio de sua computação. Tal rastreio constitui a prova lógica do raciocínio e uma explicação

corresponde à apresentação "editada" dessa prova. A justificativa de sucesso de uma condição negada é feita mostrando que todos os caminhos possíveis para a solução do objetivo não negado falham.

Predicados não-definidos bem como aqueles rotulados como askable são encarados como "perguntáveis" pelo apes, ou seja, informações a respeito deles podem ser obtidas do usuário através do componente interativo. No segundo caso, o sistema tentará usar as cláusulas já definidas para resolver o objetivo em questão; se, no entanto, for necessário obter informações adicionais, o usuário será consultado.

Por outro lado, as declarações function e unique_answer limitam o número de respostas aceitáveis para determinado predicado. A primeira expressa que o último argumento de um predicado é univocamente determinado pelos demais. Unique_answer é mais flexível. Possui quatro argumentos, os últimos dois sendo listas de parâmetros do predicado e assumindo os papéis, respectivamente, de determinadores e determinados.

Questões são levantadas ao usuário em duas formas padrão (uma para consultas do tipo is e outra para as consultas which), a menos que gabaritos sejam criados para os predicados correspondentes através das declarações is_template e which_template. A segunda possui um parâmetro adicional que identifica as variáveis que ainda não estarão instanciadas no momento em que o usuário for consultado.

A apresentação de regras na tela pode ser melhorada através da definição do predicado read_as, devendo ser definido pelo menos um para cada predicado a ser apresentado. Tais gabaritos são também utilizados na apresentação de explicações e também quando são usados os formatos padrão de consulta ao usuário.

As respostas do usuário podem estar ainda sujeitas a restrições de validade adicionais definidas por predicados do tipo valid_answer. Tais declarações possuem quatro argumentos: o primeiro é o nome do predicado interativo cujos parâmetros estão sendo validados e, o

último, as condições que as respostas do usuário devem satisfazer para que sejam aceitas pelo apes.

A fim de gerar menus, o projetista deve definir restrições de validade tendo apenas uma condição da forma "X in_menu L", onde X representa uma variável livre e L uma lista de respostas aceitáveis (tal lista pode ser gerada dinamicamente). Essas respostas serão apresentadas durante a consulta.

É importante mencionar que as cláusulas definindo os gabaritos e as restrições de validade podem ter corpo, ou seja, podem ser especificadas condições e as regras somente se aplicarão caso tais condições sejam satisfeitas.

As condicões acima são executadas pelo interpretador micro-PROLOG e, portanto, não podem gerar interação mas podem apresentar características não-lógicas. Predicados embutidos e predicados declarados pelo usuário como non-int são tratados similarmente. Embora não sejam rastreiados (e, portanto, o sistema não é capaz de "explicá-los"), são executados mais rapidamente e utilizam menos memória para tal.

Em uma consulta ao usuário do tipo which, apes lista todas as respostas que pode obter sozinho e, então, solicita novas respostas. A menos que restrições do tipo unique_answer tenham sido impostas, o usuário pode fornecer zero ou mais respostas. Ao final, deve digitar end ou enough conforme seja, respectivamente, sua intenção: dizer que não existem, definitivamente, mais respostas para essa consulta ou indicar que não há mais respostas apenas para o propósito da avaliação corrente da consulta. Nesse caso, a mesma questão pode ser colocada novamente para a resolução de novos objetivos. Como no caso de consultas do tipo is, o usuário tem a opção de, antes de responder, entrar com valid para obter uma indicação do que é esperado; as restrições de validade eventualmente existentes são usadas para indicar as respostas possíveis ou aceitáveis pelo apes.

Outra opção de auxílio disponível é digitar **why** para requerer um rastreio editado da computação realizada até o momento.

O usuário especifica os objetivos a serem resolvidos através dos comandos de consulta providos pela interface: **is**, **which**, **confirm** e **find**. Os dois primeiros são executados diretamente pelo micro-Prolog e, os demais, pelo apes.

Após resolver uma consulta **confirm** ou **find**, apes pára permitindo ao usuário solicitar a explicação da resposta obtida. A explicação constitui-se em uma apresentação, passo a passo, de como as regras e os fatos do programa e as respostas fornecidas pelo usuário foram empregadas na conclusão da resposta. Durante esse processo, o usuário tem as seguintes opções:

- avançar na explicação através dos comandos **how** ou **why**, quando há somente uma condição a ser explicada, ou indicando o número de uma condição particular quando houver mais de uma a ser explorada;

- retroceder ao nível precedendo à explicação corrente através de **ok**;

- abandonar a explicação e solicitar outra solução para a consulta original através de **more**;

- abandonar a explicação e terminar a avaliação da consulta através de **stop**.

O componente interativo do apes permite, ainda, submeter consultas aninhadas no curso da interação para, por exemplo, fazer uma pergunta (**is** ou **which**), listar (**list**) um programa ou examinar o registro do diálogo.

O usuário pode, a qualquer instante, olhar a descrição da interação com o apes através de relações embutidas específicas. Pode

assim, ver o que lhe foi perguntado, o que ele disse e o que negou. Tem também a opção de listar, salvar ou apagar o diálogo.

2.2 ETC: extending VM/Prolog with an Expert Tool Capability

ETC, desenvolvido no centro científico da IBM, no Rio de Janeiro, oferece uma facilidade de consulta ao usuário baseada em regras, similarmente ao apes. ETC provê, em adição, uma forma alternativa para manusear chamadas a predicados indefinidos.

ETC provê o operador "%", que estende o interpretador do VM/Prolog no seguinte sentido. Se uma expressão da forma %p(X1, ..., Xn), onde p é uma fórmula atômica Prolog, ocorrer em um programa, será estabelecido um diálogo com o usuário sempre que o predicado p ou algum outro predicado por ele invocado não tenha sido previamente definido e a fórmula atômica não tenha sido perguntada e completamente respondida antes.

O diálogo se inicia, portanto, sempre que um objetivo involvendo o operador "%" é executado. O usuário é solicitado a fornecer zero ou mais respostas. O prompt terá um formato default a menos que o programador da aplicação tenha fornecido uma declaração is_template ou which_template. Tais declarações devem ser usadas para prover uma formulação em linguagem natural de padrões de consultas (fórmulas atômicas p(X1, ..., Xn), onde Xi é uma constante ou variável). Regras is_template devem ser usadas quando não existirem variáveis no padrão de consulta e, por conseguinte, a consulta é do tipo sim ou não. Quando variáveis estão presentes, devem ser usadas regras which_template.

Restrições de validade são definidas através de declarações do tipo valid_answer. Em tal declaração, o projetista deve introduzir uma expressão VM/Prolog que deve ser satisfeita com as variáveis do padrão de consulta substituídas pelos valores fornecidos pelo usuário. O projetista pode, ainda, especificar que as respostas devam pertencer a um conjunto (menu) dado por enumeração ou dinamicamente formado.

Em consultas ao usuário do tipo which, todas as respostas previamente fornecidas ou geradas de alguma forma (por exemplo, como o resultado de um trigger - ver abaixo), são apresentadas na tela. A resposta fornecida pelo usuário é verificada pela regra valid_answer correspondente, se existir. Se tal regra definir um menu, as opções são mostradas.

Os predicados sobre os quais o operador "%" estabeleceu um diálogo, são automaticamente inseridos em um log. Também é possível inserir qualquer outro predicado no log, bastando fornecer uma declaração do tipo catalog.

Quando uma questão é colocada ao usuário, é interessante ele poder, de alguma forma, "reutilizar" respostas fornecidas previamente para outras questões análogas. Isto é possível graças ao relacionamento is_like (comparável aos relacionamentos semânticos is_a e part_of, embora não seja uma relação de ordem nem necessariamente uma relação transitiva). O usuário pode, a qualquer instante, durante ou fora de um diálogo, declarar que um elemento, por exemplo, X, "is_like" outro elemento Y. Nenhum, um ou ambos podem ou não estar definidos quando a declaração é feita. Se, durante um diálogo, um dos elementos, X, por exemplo, já tenha ocorrido em uma consulta Q e uma outra consulta semelhante a essa mas envolvendo um elemento Y diferente, tal que o relacionamento is_like(X, Y) seja válido, é colocada ao usuário, ele pode aprender com as respostas de Q correspondentes a X fornecidas anteriormente.

No lugar de responder imediatamente a uma consulta, o usuário pode entrar com uma palavra chave ou comando a fim de:

- mostrar a regra de validade correspondente;
- mostrar um rastreio;
- mostrar os nomes dos predicados no log;
- cancelar a execução e, como conseqüencia, limpar o log;
- listar as cláusulas de algum predicado especificado;
- avaliar alguma expressão lógica;

- estabelecer um relacionamento is_like entre dois elementos especificados;
- apresentar instâncias do predicado associado para elementos declarados como "is_like" àqueles no predicado.

Os comandos para listar um predicado, avaliar alguma expressão lógica e atribuir um relacionamento is_like também podem ser invocados fora do diálogo.

A definição de "gabaritos" para apresentar cláusulas definindo um predicado em um formato de linguagem natural é feita através de declarações read_as.

Uma declaração trigger pode ser associada, pelo projetista, a um padrão de consulta específico. Em tal regra, ele introduz uma expressão VM/Prolog contendo comandos executáveis, usualmente para adicionar ou remover cláusulas da base. Se a resposta fornecida pelo usuário para uma consulta é aceita, a regra de trigger correspondente é executada.

O projetista pode, também, especificar declarações relacionadas a completude introduzindo cláusulas definindo os predicados unique_answer e complete_answer. O primeiro é empregado para declarar que apenas uma resposta válida é esperada do usuário para a consulta correspondente, de forma que o usuário não deve ser reconsultado posteriormente. O segundo permite ao projetista especificar uma expressão que será executada tão logo o usuário digite end após as respostas fornecidas à uma consulta do tipo which. Se a avaliação falha, o conjunto de resposta fornecido é considerado incompleto e o usuário continua a ser solicitado para respostas adicionais. Na ausência de regras complete_answer, o usuário não precisa fornecer valor algum, podendo digitar diretamente end. Esse também pode ser o caso de uma resposta negativa a uma consulta is.

Depois de a resposta ter sido aceita e da execução, se for o caso, do trigger associado, o diálogo para a consulta corrente é suspenso. Ele pode, contudo, ser continuado quando uma outra solução é

requisitada para o objetivo contendo o operador "%", exceto no caso de questões do tipo sim ou não ou se houver uma regra unique_answer associada ou ainda se o menu de respostas permissíveis tiver se esgotado.

Por fim, ETC provê facilidades de explicação e a possibilidade de limitar a ação do operador "%". O comando explain permite obter o rastreio da execução do objetivo fornecido como parâmetro. A fim de limitar o escopo do operador "%", a declaração hide deve ser usada. A fórmula atômica especificada como argumento é manuseada pelo interpretador VM/Prolog. Como efeito colateral, predicados chamados pelo predicado declarado como "escondido" não serão rastreiados. Predicados embutidos do VM/Prolog e várias "utilidades" (utilities) são tratadas de forma similar.

2.3 SYLLOG

SYLLOG é um sistema de bancos de dados especialista desenvolvido no centro de pesquisa da IBM T. J. Watson, em Yorktown Heights, Nova Iorque. Baseia-se em uma nova teoria de conhecimento declarativo, que torna possível expressar o conhecimento em uma linguagem próxima a natural, no lugar de tratar a aquisição de conhecimento de forma procedimental ou baseada em transições de estados. Na construção da base de conhecimento para o SYLLOG, o programador não especifica informações de controle para sua execução.

SYLLOG é constituído por uma série de componentes:

- um **gerenciador de telas**;
- um **arquivo de linguagem**, que permite mostrar mensagens em diversas línguas;
- um **carregador**, para preparar e verificar a base de conhecimento;
- um **componente de atualização**, para efetuar mudanças na base;
- uma **máquina de inferência**;
- uma **interface para o sistema gerenciador do banco de dados**;
- um **gerador de explicações**.

Para introduzir o conhecimento, o projetista escreve fatos e silogismos com o uso da linguagem SYLLOG de fatos e de regras, respectivamente. Supõe-se que ele faça uso de sentenças (menores do que a largura da tela) em um estilo semelhante ao inglês. Supõe-se que elementos exemplo ocorram em tais sentenças assumindo o papel de variáveis. Tais elementos consistem de uma palavra precedida por eg. Um exemplo de uma sentença SYLLOG seria "eg_verb eg_det eg_noun is a verb phrase".

Um grupo de fatos relacionados é escrito na forma de uma tabela, similar a um banco de dados relacional, exceto pelo fato de que uma tabela SYLLOG será encabeçada por uma sentença SYLLOG. Entre o cabeçalho e os fatos uma linha contendo exclusivamente símbolos de igual ("=====") deve ser inserida. Cada linha conterá os valores que as palavras exemplo assumem em cada fato.

Um silogismo corresponde a uma regra em inglês. É caracterizado por uma ou mais sentenças escritas acima de uma linha tracejada ("----"), as premissas, e por uma sentença abaixo da linha, a conclusão.

Toda a informação de controle necessária para garantir que o conhecimento seja executado corretamente é introduzida pelo próprio SYLLOG. Além disso, SYLLOG contém mecanismos internos que automaticamente otimizam o processo de resposta às consultas usando a base de conhecimento. Logo, a eficiência do sistema depende do que é especificado e das otimizações particulares realizadas pela versão corrente do SYLLOG.

Embora SYLLOG trabalhe com sentenças em linguagem semelhante ao inglês, não é necessário construir um dicionário ou gramática para a adição de novos elementos do inglês, sendo o significado das palavras extraído do próprio contexto. SYLLOG faz inferências logicamente corretas baseado no que lhe é dito. Por outro lado, ele não sabe muito sobre a linguagem em comparação com os outros programas de processamento de linguagem natural. Como conseqüência, advêm as vantagens de adicionar conhecimento ao sistema em qualquer linguagem e

de não ser necessário instruir o sistema sobre os detalhes daquela linguagem em que se está trabalhando. Por outro lado, surgem também desvantagens: restrição de se poder trabalhar apenas com sentenças declarativas simples e necessidade de uso das "palavras exemplo". Além disso, a mesma sentença deve ser usada consistentemente para significar a mesma coisa, a menos que silogismos tenham sido escritos para especificar que duas sentenças diferentes têm o mesmo significado.

Sobre a organização da base de dados do SYLLOG, acrescente-se que é possível especificar conhecimento rcursivo, raciocínio baseado em frames e, de certa forma, raciocínio orientado a objetos. Conhecimento aproximado ou baseado em julgamentos pode ser introduzido através de frases como "there is a fair expectation that ...", embora SYLLOG raciocine logicamente. Para introduzir conhecimento, é possível fazer uso de novas palavras ou frases de áreas específicas (por exemplo, "habeas corpus"). SYLLOG é extensível no sentido de que programas em Prolog ou em outras linguagens podem ser nomeados com sentenças em inglês e, com isso, tornarem-se conhecidos da base de conhecimento. Entretanto, o sistema não é capaz de prover explicações dentro de tais extensões.

Embora o conjunto de características do SYLLOG possa parecer complexo à primeira vista, a interface com o usuário é bastante simples. Foi projetada de forma a permitir que mesmo aqueles que não sejam programadores nem "engenheiros de conhecimento" possam construir suas próprias bases de conhecimento.

Depois de carregar uma base de conhecimento para a área de trabalho a partir de um arquivo CMS, SYLLOG apresentará ao usuário um menu de sentenças da base, agrupadas e ordenadas de modo apropriado. De fato, para a base de conhecimento carregada, SYLLOG mostra na tela as conclusões dos silogismos e os cabeçalhos das tabelas.

O uso do SYLLOG é basicamente feito através de teclas PF. Por exemplo, a tecla PF3 sempre permite retornar à tela anterior e, quando na primeira tela, ao sistema operacional VM. Para fazer uma pergunta,

o usuário deve selecionar uma sentença e pressionar a tecla PF apropriada. Uma nova tela contendo a resposta, na forma de uma tabela SYLLOG, é então mostrada.

Tendo obtido uma resposta, o usuário pode solicitar uma explicação do porquê de um fato específico ter sido concluído. Para tal, seleciona a linha correspondente na tabela e pressiona uma outra tecla PF. A explicação, apresentada em outra tela, mostra os silogismos que contribuiram para a solução escolhida, com seus elementos exemplo instanciados. É interessante citar que um rastreio da execução - lista com todos os passos que o sistema dá para alcançar a solução - é mais detalhado do que uma prova - caminho para a conclusão - que, por sua vez, pode ser mais detalhada do que uma explicação.

SYLLOG também pode ser indagado sobre uma conclusão negativa. Ele provê, nesse, caso, uma explicação mostrando o que seria necessário a fim de transformar a resposta negativa em positiva - pois não é tão simples produzir uma explicação de uma conclusão negativa para sistemas que usam a "hipótese do mundo fechado", como o SYLLOG.

Quando existe mais de uma maneira de se obter determinada resposta, a seguinte estratégia é adotada: se a conclusão desejada segue da base de conhecimento (a resposta à questão é "sim"), a explicação mais simples é mostrada antes; por outro lado, se a resposta é não, a explicação mais longa na qual nada é repetido deve ser mostrada primeiro.

A fim de dizer ao SYLLOG que fatos sobre determinada sentença devem ser obtidos, quando necessários, do usuário, o projetista deve criar uma tabela encabeçada por uma sentença precedida pela palavra "qtu:". Similarmente, prefixar o cabeçalho de uma tabelha SYLLOG com a palavra "sql:" informa que os fatos sobre a sentença correspondente devem ser obtidos de uma tabela SQL.

O sistema SYLLOG, que foi escrito em Prolog, contém sua própria máquina de inferência. Isso se justifica pelo fato do Prolog, embora

eficiente, não atribuir sempre o significado desejado às regras da base de conhecimento. Por exemplo, silogismos recursivos, embora tenham uma semântica declarativa simples, não podem ser facilmente interpretados declarativa e eficientemente ao mesmo tempo. SYLLOG trata corretamente muitos conjuntos de silogismos recursivos que não seriam manipulados de forma adequada pelo Prolog.

Conforme visto, o conhecimento é introduzido no SYLLOG em uma forma declarativa. Por outro lado, é necessário alguma garantia de que os procedimentos internos de raciocínio produzem resultados correspondendo ao significado declarativo atribuído. O padrão declarativo mais útil, que provê uma forma satisfatória de julgar uma máquina de inferência, parece ser aquele baseado na teoria de modelos da lógica. A noção de modelo pode ser usada para atribuir um significado declarativo às bases de conhecimento SYLLOG, em particular àquelas contendo negação. Idealmente, gostaria-se de ser capaz de provar que os procedimentos internos de raciocínio gerariam resultados de acordo com o modelo declarativo da base de conhecimento. Para máquinas de inferência simples, isso é possível de ser feito. Como as máquinas de inferência mais eficientes são também mais complicadas, elas são usualmente verificadas experimentalmente usando-se exemplos bem selecionados.

Para os propósitos do SYLLOG, um "modelo" de uma base de conhecimento é um conjunto de fatos sem variáveis satisfazendo às seguintes condições:

(i) cada fato da base de conhecimento está no modelo;

(ii) se as premissas de um silogismo da base de conhecimento podem ser instanciadas a fatos no modelo, então a instância correspondente à conclusão do silogismo também está no modelo.

Essas duas condições estão intimamente relacionadas ao procedimento de construção do modelo mínimo de Herbrand de um conjunto de cláusulas definidas. Em outras palavras, o modelo de uma base de conhecimento, como aqui proposto, é, de fato, o modelo mínimo de Herbrand do

conjunto de cláusulas associado aos fatos e aos silogismos da base de conhecimento.

3. O Protótipo HIER

3.1 Apresentação

O desenvolvimento do HIER objetivou alcançar maturidade no projeto de ferramentas para a construção de SEs. Embora não introduza conceitos inovadores, a implementação realizada de forma sistemática e uniforme originou um programa modular capaz de assimilar, de forma simples e natural, modificações e extensões futuras.

Desenvolvido sobre o Prolog, baseia-se em regras de produção expressas através de cláusulas de Horn. Em adição aos recursos do Prolog, provê as seguintes facilidades:

- consulta ao usuário de informações adicionais necessárias à inferência de determinado resultado;

- manutenção de restrições de integridade do tipo "resposta única" e "resposta válida";

- explicação passo a passo do raciocínio, tanto para perguntas do sistema (realizadas pelo componente de consulta) quanto para as conclusões obtidas;

- possibilidade de interação em uma linguagem mais próxima da natural através do uso de "gabaritos"

O diagrama abaixo apresenta a arquitetura do HIER:

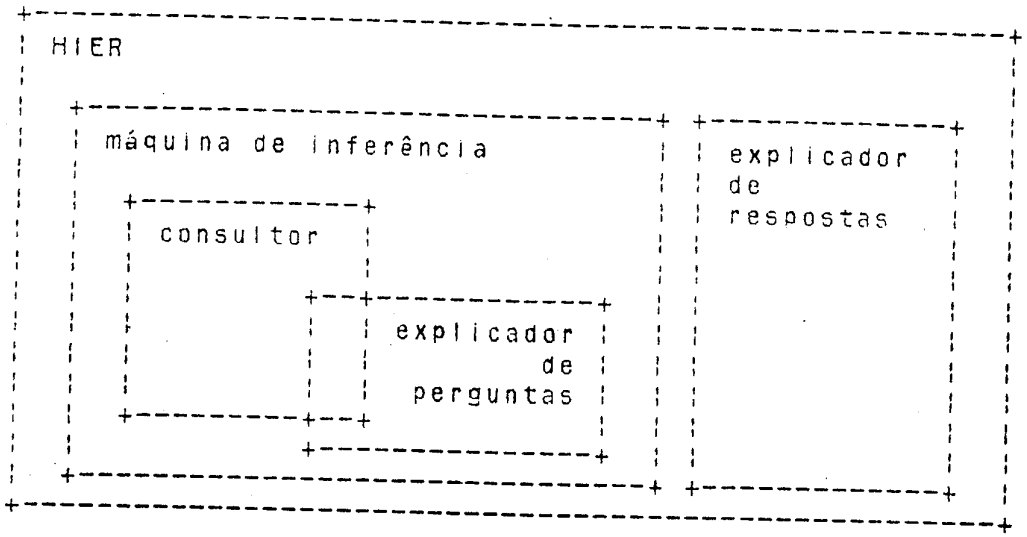

Figura 1

Embora não ilustrado na figura acima, o explicador de respostas comunica-se com o consultor através das cláusulas unitárias do tipo was_told (ver seções correspondentes).

A ativação da máquina de inferência e do explicador de respostas pode ser feita diretamente no Arity Prolog através dos comandos descritos abaixo. Além disso, O HIER conta com um menu (¡HIER) no qual é possível operar sobre um objetivo corrente. A ativação desse ambiente é feita pelo comando ¡HIER((objetivo)) (ou apenas ¡HIER, caso em que o objetivo é explicitamente solicitado). As diversas opções desse menu incluem (ver tabela.1): usar a máquina de inferência do HIER ou do próprio Arity Prolog para realizar uma consulta, ativar o explicador de respostas, trocar o objetivo corrente e, por fim, voltar ao Arity.

Cada um dos componentes é detalhado a seguir. Os comandos são especificados, sempre que se julgar conveniente, em BNF.

3.2 A máquina de inferência

O componente de inferência usa o mecanismo de encadeamento reverso (backward chaining), ou seja, procura pelas evidências que permitam obter determinada conclusão. Possui média granularidade: discerne conjunções, negações, predicados do sistema, fatos, regras e predicados perguntáveis (ao usuário). Engloba os componentes de consulta ao usuário e de explicação das perguntas colocadas pelo primeiro. A máquina de inferência é ativada por consultas do tipo is ou which, diretamente solicitadas no Arity Prolog (sintaxe descrita abaixo) ou através do menu ativado pelo comando iHIER. Nesse caso, o sistema encarrega-se de escolher a consulta adequada – which ou is, caso existirem ou não, respectivamente, variáveis não instanciadas.

```
<consulta> ::=
   is(<objetivo>). |
   which(<var_termo> : <objetivo>).

<objetivo> ::=
   <predicado Prolog>

<var_termo> ::=
   <variável Prolog> |
   <lista Prolog de variáveis Prolog>
```

Em is, <objetivo> não pode conter variáveis. Em which, todas as varáveis constantes em <objetivo> devem também constar em <var_termo> e vice-versa.

A uma resposta positiva a uma consulta do tipo is, HIER envia mensagem de sucesso e espera por uma solicitação de backtracking por parte do usuário, que deve digitar ;. Tal procedimento se repetirá até que todas as alternativas para a confimação do objetivo tenham se esgotado. Todavia, se qualquer outra tecla diferente de ; for pressionada, a consulta é concluída.

Em consultas do tipo which, HIER procura por todas as instâncias da variável (ou das variáveis) que fazem com que o objetivo seja satisfeito, mostrando tais respostas na tela. O backtracking é realizado automaticamente e, por isso, uma mesma resposta pode aparecer diversas vezes, o número de repetições correspondendo ao número de soluções por caminhos diferentes possíveis.

Após a conclusão de qualquer consulta o controle é retornado ao Arity Prolog ou ao iHIER, dependendo de como tenha sido feita a ativação ao HIER.

3.3 O consultor

O consultor é ativado quando todas as possibilidades de resolver um predicado rotulado como perguntável (askable) se esgotaram. Independente da consulta original, o consultor pode fazer perguntas do tipo is ou which, dependendo, respectivamente, de todas as variáveis do predicado estarem ou não instanciadas naquele momento.

```
<rótulo perguntável> ::=
   askable(<predicado Prolog>) :- <condição>. !
   askable(<predicado Prolog>.

<condição> ::=
   <corpo de cláusula Prolog>
```

Observação: a <condição> é verificada diretamente pelo Arity Prolog e não pela máquina de inferência do HIER.

No caso de uma pergunta is, o sistema solicita confirmação (yes ou ok) ou negação (no) de um fato. Como alternativa, o usuário pode entrar com continue ou, resumidamente, cont. O efeito, nesse caso, corresponde à negação temporária do fato; por conseguinte, a mesma pergunta poderá ser recolocada no futuro.

Se a pergunta for which, são requeridos valores para as variáveis não instanciadas. Se mais de uma variável não estiver instanciada, a resposta deve estar na forma de lista. Quando o usuário não possuir mais respostas a oferecer, deve digitar end ou ok (corresponde ao enough do apes). Em qualquer dos casos, há a alternativa de pedir explicação do porquê da pergunta ter sido feita: why ou explain. O controle é transferido para o explicador de perguntas, próximo componente a ser apresentado.

Quando o usuário confirma um fato em resposta a uma consulta is, o consultor o armazena na base de conhecimento juntamente com uma cláusula do tipo was_told(<fato>). Como conseqüência de uma negação (através de no - a base permanece inalterada para cont ou continue), apenas was_denied(<fato>) é inserido. Tais cláusulas poderão ser futuramente utilizadas para a manutenção e manipulação de um log do diálogo mantido com o usuário. Atualmente, é tratada apenas pelo explicador de respostas (mensagens diferentes são fornecidas para a justificativa de fatos introduzidos pelo usuário e para aqueles armazenados diretamente pelo projetista).

O consultor é também o responsável pela manutenção de certas restrições de integridade da base de connecimento. Mas, por estar aqui localizado, o mecanismo de validação apenas verifica as respostas fornecidas durante uma consulta; é tarefa do projetista da aplicação construir uma base íntegra a priori.

É ainda importante notar que as restrições abaixo enumeradas são automaticamente observadas antes de uma pergunta do tipo is ser colocada ao usuário. Portanto, qualquer resposta (yes, no ou continue) é imediatamente aceita.

São as seguintes as verificações feitas pelo sistema:

- resposta não repetida e não contraditória: são rejeitadas respostas já conhecidas ou que contradigam fatos armazenados na base;

- resposta válida: o projetista deve especificar, para cada predicado perguntável, pelo menos uma regra para validação de respostas (à exceção do caso analisado abaixo); somente respostas que atendam a uma dessas regras serão aceitas.

```
<cláusula para resposta válida> ::=
   valid_answer(<predicado Prolog>) :- <condição>. !
   valid_answer(<predicado Prolog>). !
   valid_answer(_).
```

A especificação da cláusula valid_answer(_) torna sem efeito qualquer outra cláusula para resposta válida armazenada na base de conhecimento: toda e qualquer resposta - para qualquer predicado! - será considerada válida no sentido aqui estudado;

- resposta única: tal restrição permite definir relacionamentos 1:m entre parâmetros de um predicado.

```
<cláusula para resposta única> ::=
   unique_answer(<predicado Prolog>, <determinadores>,
      <determinados>).

<determinadores> ::= <lista Prolog de variáveis Prolog>

<determinados> ::= <lista Prolog de variáveis Prolog>
```

A uma valoração de <determinadores> corresponde apenas uma valoração de <determinados> para o predicado sendo considerado. Isso não impede, no entanto, que vários determinadores correspondam aos mesmos determinados. Para o estabelecimento dessa restrição de integridade mais forte é necessário especificar uma cláusula para resposta única adicional com os mesmos determinadores e determinados só que, agora, com suas funções intercambiadas.

"Funções" podem ser especificadas como um caso particular. Os argumentos correspondem aos determinadores e, o resultado, ao determinado (nesse caso, uma lista com uma única variável).

Três outras classes de cláusulas auxiliam a manutenção de interações mais amigáveis com o usuário no sentido de permitir que os predicados e as perguntas feitas pelo consultor possam ser apresentadas em formato de linguagem natural. Tais cláusulas especificam "gabaritos" e são as seguintes:

```
<cláusula de gabarito> ::=

    read_as(<predicado Prolog>, <lista Prolog>) :-
        <condição>. !
    read_as(<predicado Prolog>, <lista Prolog>). !

    is_template(<predicado Prolog>, <lista Prolog>) :-
        <condição>.
    is_template(<predicado Prolog>, <lista Prolog>). !

    which_template(<predicado>, <var_termo>, <lista Prolog>)
        :- <condição>.
    which_template(<predicado>, <var_termo>, <lista Prolog>).
```

Em which_template, todas as variáveis não instanciadas do predicado (cujos valores serão perguntados) devem constar em <var_termo>.

A aplicação de uma cláusula de gabarito consiste em escrever os elementos da <lista Prolog> após a unificação do <predicado Prolog> e a <condição>, se houver, ter sido satisfeita (verificada diretamente pelo Arity Prolog).

Cláusulas read_as são empregadas pelo HIER para apresentação de predicados nas explicações e eventualmente em perguntas. Convém aqui lembrar que o objetivo de uma consulta (ver máquina de inferência) tem

de ser expresso em notação Prolog; read_as, is_template e which_template são usados pelo sistema apenas para apresentações na tela.

Cláusulas do tipo is_template e which_template são aplicadas na colocação de perguntas ao usuário. Caso isso não seja possível, tenta-se empregar algum gabarito especificado por read_as.

Não havendo uma cláusula de gabarito apropriada (contexto correto e <condição> satisfeita) para determinado predicado, formas default são empregadas.

3.4 O explicador de perguntas

Quando uma pergunta é feita ao usuário este, não entendendo o porquê dela ter sido levantada, tem a opção de digitar why ou explain e, com isso, ativar o explicador de perguntas. A regra que a máquina de inferência está tentando resolver é apresentada, usando, se possível, gabaritos especificados por read_as (tanto para a cabeça quanto para os predicados do corpo). A pergunta anterior é, então, repetida. Mas, se isso ainda não for suficiente, com um novo why é possível visualizar a regra um nível acima, ou seja, aquela que provocou a tentativa de resolução da regra anteriormente apresentada. O processo pode ser continuado até que se alcance o objetivo inicial. Uma tentativa de justificar tal objetivo gerará uma mensagem indicando a impossibilidade de fornecer mais explicações.

3.5 O explicador de respostas

A fim de obter a prova da confirmação ou negação de um objetivo pelo HIER, este dispõe de um componente específico independente de explicação. Faz-se a chamada ao explicador de respostas, no Arity Prolog, através do comando

```
explain(<objetivo>).
```

onde <objetivo> não deve conter variáveis.

O explicador, usando um mecanismo próprio independente da máquina de inferência, monta uma árvore de prova com todas as alternativas que justificam o objetivo ou que impedem sua confirmação. O algoritmo empregado para a construção dessa árvore é aquele descrito em [Wal 87].

O próximo passo consiste em caminhar passo a passo nessa árvore, começando pela raiz. A primeira regra é mostrada com os seus subobjetivos numerados a partir do 1. Para descer um nível, basta digitar o número correspondente ao subobjetivo. Para caminhar no sentido inverso, entra-se com up ou algum sinônimo (back ou return). Quando a folha da árvore tiver sido alcançada (ou seja, quando um fato ou sua negação for apresentado), o sistema volta automaticamente para o nível acima.

A qualquer instante existe a opção (end, commit ou ok) de abandonar o ramo em análise. O explicador espera por uma solicitação de backtracking, realizado se o usuário digitar ";". Se o backtracking for solicitado e não houver mais ramo a ser pesquisado ou se qualquer tecla diferente for pressionada, o controle é devolvido ao Arity ou ao iHIER, conforme o caso.

4. Conclusões

Fundamentalmente, os Sistemas Especialistas visam resolver problemas importantes que requeririam a presença de um especialista. Devido à natureza das aplicações envolvidas, o nível de abstração final está usualmente acima daquele provido diretamente pelas linguagens tradicionais ou prescritivas. Por outro lado, o mecanismo de inferência baseado em regras do Prolog, uma linguagem declarativa, mostra-se muito mais adequado à resolução de tais problemas. O sistema aqui apresentado, HIER, escrito em Prolog, procura criar um ambiente ainda mais evoluído que mais se aproxima do projetista e do usuário final. Para isso, provê facilidades que basicamente permitem obter informações adicionais do usuário e gerar explicações do raciocínio.

Funções particularmente úteis incluem a explicação de perguntas e a apresentação passo a passo de todas as justificativas, sendo também possível caminhar nos dois sentidos (da conclusão ou do objetivo) no caso do explicador de respostas. Adicionalmente, as cláusulas de gabarito permitem mostrar predicados em formato de linguagem natural.

O sistema desenvolvido é um protótipo. Isso significa que pode ser usado como um "laboratório" para a implementação de facilidades extras que se julgar convenientes (como o complete_answer, trigger e is_like do ETC).

Apêndices

A.1 Relação dos comandos e regras sintaticas

A tabela 1 apresenta, resumidamente, todos os comandos disponíveis no HIER.

local	comando	descrição
Arity	is(<objetivo>).	confirma ou não um objetivo
	which(<var_termo>: <objetivo>).	obtém as respostas que satisfazem um objetivo
	explain(<objetivo>).	obtém a explicação da confirmação ou não de um objetivo
	iHIER, iHIER(<objetivo>)	ativa o iHIER
ihier	1, isolve	resolve o objetivo corrente
	2, solve	resolve o objetivo corrente através do Arity
	3, explain, how, proof	obtém a explicação da confirmação ou não do objetivo corrente
	4, newgoal	troca o objetivo corrente
	5, end	volta ao Arity
consultor	yes, ok	confirma um fato
	no	nega um fato
	continue, cont	nega temporariamente um fato
	why, explain	ativa o explicador de perguntas que apresenta a regra que se está tentando resolver
explicador de perguntas	why, explain	obtém a regra que provocou a tentativa de resolução da regra apresentada
explicador de respostas	1, ..., n	desce um nível na árvore de prova, justificando o subobjetivo rotulado com o número escolhido
	up, back return	sobe um nível na árvore de prova
	commit, end, ok	abandona o ramo em análise da árvore de prova

Tabela 1

Regras sintáticas:

```
<consulta> ::=
   is(<objetivo>). |
   which(<var_termo> : <objetivo>).

<objetivo> ::=
   <predicado Prolog>

<var_termo> ::=
   <variável Prolog> |
   <lista Prolog de variáveis Prolog>


<rótulo perguntável> ::=
   askable(<predicado Prolog>) :- <condição>. |
   askable(<predicado Prolog>.

<condição> ::=
   <corpo de cláusula Prolog>


<cláusula para resposta válida> ::=
   valid_answer(<predicado Prolog>) :- <condição>. |
   valid_answer(<predicado Prolog>). |
   valid_answer(_).


<cláusula para resposta única> ::=
   unique_answer(<predicado Prolog>, <determinadores>,
      <determinados>).

<determinadores> ::= <lista Prolog de variáveis Prolog>

<determinados> ::= <lista Prolog de variáveis Prolog>


<cláusula de gabarito> ::=

   read_as(<predicado Prolog>, <lista Prolog>) :-
      <condição>. |
   read_as(<predicado Prolog>, <lista Prolog>). |

   is_template(<predicado Prolog>, <lista Prolog>) :-
      <condição>.
   is_template(<predicado Prolog>, <lista Prolog>). |

   which_template(<predicado>, <var_termo>, <lista Prolog>)
      :- <condição>.
   which_template(<predicado>, <var_termo>, <lista Prolog>).


<explicação> ::=
   explain(<objetivo>).
```

A.2 Exemplos

Exemplo 1.

```
alimento_saudavel(X) :-
   ingerivel(X),
   contem(X, Y),
   doce(Y),
   not prejudicial(X).

ingerivel(X) :-
   comivel(X).

ingerivel(X) :-
   bebivel(X).

ingerivel(X) :-
   engolivel(X).

prejudicial(X) :-
   radioativo(X).

prejudicial(X) :-
   toxico(X).

toxico(X) :-
   contem(X, Y),
   X \== Y,
   toxico(Y).

askable(contem(_, _)).
askable(doce(_)).
askable(comivel(_)).
askable(bebivel(_)).
askable(engolivel(_)).
askable(toxico(_)).

is_template(doce(X), ['E'' doce ', X, '? ']).

is_template(P, ['E'' ', F, ' ', X, '? ']) :-
   functor(P, F, _),
   member(F, [comivel, bebivel]),
   arg(1, P, X).

valid_answer(_).

read_as(contem(X, Y),[X, ' contem ', Y]).
read_as(doce(X), [X, ' e'' doce ']).

engolivel(sal).
engolivel(acucar).

comivel(chocolate).
comivel(carne).

bebivel(agua).
bebivel(cafe).
bebivel(leite_de_chernobyl).

contem(cafe, acucar).
contem(chocolate, acucar).
contem(leite_de_chernobyl, acucar).
contem(chiclete, aromatizante).
contem(chiclete, flavorizante).
contem(chiclete, corante).
contem(chiclete, conservante).

doce(acucar).
doce(chocolate).
not(doce(sal)).
not(doce(agua)).

radioativo(leite_de_chernobyl).

toxico(aromatizante).
toxico(flavorizante).
toxico(corante).
toxico(conservante).
not(toxico(sal)).
not(toxico(acucar)).
not(toxico(agua)).
not(toxico(chocolate)).
```

```
11.20 D api
Arity/Prolog Interpreter Version 4.0
Copyright (C) 1986 Arity Corporation
?- consult(hier), consult('hier.var'), consult('hier.exp').

yes
?- iHIER.
```

```
>> Please, enter the goal to be solved.
   ? alimento_saudavel(chocolate).

| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     alimento_saudavel(chocolate)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 1.
***Success.
   & ;

Backtracking...

>> Which chocolate contem Vl?
   Answer is açucar.
   ***It is already a fact chocolate contem acucar
   Answer is cacau.
   Answer is end.
E' doce cacau? continue.
E' bebivel chocolate? no.
>> Is engolivel(chocolate)? no.
***Fail

| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     alimento_saudavel(chocolate)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 1.
***Success.
   & ;

Backtracking...

E' doce cacau? no.
>> Which chocolate contem Vl?
   Answer is end.
***Fail

| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     alimento_saudavel(chocolate)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 3.
```

```
---------------------
  I have concluded alimento_saudavel(chocolate)
  because
  (1) I can show ingerivel(chocolate)
  (2) I know chocolate contem acucar
  (3) I know acucar e' doce
  (4) I can show not prejudicial(chocolate)
---------------------

Option?
1.

---------------------
  I have concluded ingerivel(chocolate)
  because
  (1) I know comivel(chocolate)
---------------------

Option?
1.

It is a fact comivel(chocolate)

---------------------
  I have concluded ingerivel(chocolate)
  because
  (1) I know comivel(chocolate)
---------------------

Option?
up.
---------------------
  I have concluded alimento_saudavel(chocolate)
  because
  (1) I can show ingerivel(chocolate)
  (2) I know chocolate contem acucar
  (3) I know acucar e' doce
  (4) I can show not prejudicial(chocolate)
---------------------

Option?
4.

---------------------
  I have concluded not prejudicial(chocolate)
  because
  (1) I cannot show prejudicial(chocolate)
---------------------

Option?
1.

---------------------
  I cannot conclude prejudicial(chocolate)
  because
  (1) I do not know radioativo(chocolate)
---------------------

Option?
end.
***End of explanation.
   & ;

Backtracking...


---------------------
  I have concluded alimento_saudavel(chocolate)
  because
  (1) I can show ingerivel(chocolate)
  (2) I know chocolate contem acucar
  (3) I know acucar e' doce
  (4) I can show not prejudicial(chocolate)
---------------------

Option?
4.

---------------------
  I have concluded not prejudicial(chocolate)
  because
  (1) I cannot show prejudicial(chocolate)
---------------------
```

```
Option?
1.
---------------------
  I cannot conclude prejudicial(chocolate)
  because
  (1) I know not toxico(chocolate)
---------------------

Option?
1.

It is a fact not toxico(chocolate)

---------------------
  I cannot conclude prejudicial(chocolate)
  because
  (1) I know not toxico(chocolate)
---------------------

Option?
commit.
***End of explanation.
   &

-----------------------------
| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     alimento_saudavel(chocolate)

>> Choose one:
  (1) Interactive resolution
  (2) Non-interactive resolution
  (3) Explanation
  (4) Change goal
  (5) End
>> Option? 5.

Thank you, I have learned a little bit more.
```

Exemplo 2.

```
askable(casado(_,_)).
unique_answer(casado(X,Y), [X], [Y]).
unique_answer(casado(X,Y), [Y], [X]).
```

```
?- iHIER(casado(marco,Y)).
+-----------------------------------------+
| Highly Interactive Expert Resolver      |
| MAMR, 1989                              |

I am trying to solve
     casado(marco,V1)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 2.
Answers:

+-----------------------------------------+
| Highly Interactive Expert Resolver      |
| MAMR, 1989                              |

I am trying to solve
     casado(marco,V1)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option?1.
>> Which casado(marco,V1)?
   Answer is adriana.
Answers:
   adriana
***End of which.

+-----------------------------------------+
| Highly Interactive Expert Resolver      |
| MAMR, 1989                              |

I am trying to solve
     casado(marco,V1)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 1.
Answers:
   adriana
***End of which.

+-----------------------------------------+
| Highly Interactive Expert Resolver      |
| MAMR, 1989                              |

I am trying to solve
     casado(marco,V1)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 4.
>> Please, enter the goal to be solved.
   ? casado(X,adriana).

+-----------------------------------------+
| Highly Interactive Expert Resolver      |
| MAMR, 1989                              |

I am trying to solve
     casado(V1,adriana)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 2.
Answers:
   marco
```

```
| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     casado(V1,adriana)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 1.
Answers:
   marco
***End of which.

| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     casado(V1,adriana)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 3.

I was told casado(marco,adriana)
***End of explanation.
   &

| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     casado(V1,adriana)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 4.
>> Please, enter the goal to be solved.
  ? casado(X,Y).

| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     casado(V1,V2)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 1.
Answers:
   [marco,adriana]

| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     casado(V1,V2)

>> Choose one:
   (1) Interactive resolution
   (2) Non-interactive resolution
   (3) Explanation
   (4) Change goal
   (5) End
>> Option? 1.
```

```
>> Which casado(V1,V2)?
   Answer is [marcio,marcia].
   Answer is [carlos,carla].
   Answer is [marco,marcia].
   ***Unique answer constraint.
      Fact casado(marco,marcia) refused.
   Answer is [rodolfo,adriana].
   ***Unique answer constraint.
      Fact casado(rodolfo,adriana) refused.
   Answer is [marcio,marcia].
   ***You have already told me casado(marcio,marcia)
   Answer is ok.
Answers:
   [marco,adriana]
   [marcio,marcia]
   [carlos,carla]
***End of which.

_____________________________________
| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     casado(V1,V2)

>> Choose one:
  (1) Interactive resolution
  (2) Non-interactive resolution
  (3) Explanation
  (4) Change goal
  (5) End
>> Option? 2.
Answers:
   [marco,adriana]
   [marcio,marcia]
   [carlos,carla]

_____________________________________
| Highly Interactive Expert Resolver
| MAMR, 1989

I am trying to solve
     casado(V1,V2)

>> Choose one:
  (1) Interactive resolution
  (2) Non-interactive resolution
  (3) Explanation
  (4) Change goal
  (5) End
>> Option? 5.
Thank you, I have learned a little bit more.
```

A.3 O Programa Prolog

```
/*-------------------------------------------------------------------------+
| Highly Interactive Expert Resolver                                       |
|                                                                          |
| Pontificia Universidade Catolica do Rio de Janeiro                       |
| Marco Antonio Mortari Rezende 8828613                                    |
+--------------------------------------------------------------------------+
| 21/09qui1989 |
+-------------*/

/*-------------------------------------------------------------------------+
| definicao das palavras chaves do rastreio interno                        |
+-------------------------------------------------------------------------*/
  :- op(160, xfy, and).
  :- op(170, xfy, because).
  :- op(150, fy, system).
  :- op(150, fy, fact).
  :- op(150, fy, told).
  :- op(150, fy, denied).
  :- op(160, fy, ?).
  :- op(150, fy, ??).

/*-------------------------------------------------------------------------+
| iHIER                                                                    |
+-------------------------------------------------------------------------*/
  iHIER :-
     cls,
     repeat,
     [! ask_goal(Goal),
     hier(Goal, HiOpt) !],
     HiOpt == end.

  iHIER(G) :-
     cls,
     [! hier(G, HO) !],
     HO == newgoal,
     repeat,
     [! ask_goal(Goal),
     hier(Goal, HiOpt) !],
     HiOpt == end.

  iHIER(_).

  ask_goal(Goal) :-
     nl, write(' Please, enter the goal to be solved.'),
     nl, write('  ? '),
     read(Goal).

  hier(Goal, HiOpt) :-
     repeat,
     [! display_header,
     ask_hier(Goal, HiOpt),
     respond_hier(HiOpt, Goal) !],
     (HiOpt == newgoal;
      HiOpt == end).

  ask_hier(Goal, HiOpt) :-
     display_hier(Goal),
     nl, write(' Option? '),
     rd_hier(HiOpt).

  respond_hier(isolve, Goal) :-
     insted(Goal), !,
     is(Goal).

  respond_hier(isolve, Goal) :-
     get_vars(Goal, Vars1),
     elem_or_list(Vars1, Vars), !,
     which(Vars: Goal).

  respond_hier(solve, Goal) :-
     insted(Goal),
     call(Goal),
     nl, write('***Success.'),
     nl, write('    '),
     again.

  respond_hier(solve, Goal) :-
     get_vars(Goal, Vars1),
     elem_or_list(Vars1, Vars), !,
     findall(Vars, Goal, Answers).
```

```
      nl, write('Answers:'),
      display_answers(Answers).

   respond_hier(solve, _) :-
      nl, write('***Fail').

   respond_hier(explain, Goal) :-
      explain(Goal).

   respond_hier(newgoal, _).

   respond_hier(end, _) :-
      nl, write('Thank you, I have learned a little bit more.').

/*-------------------------------------------------------------------------------+
| resolvedor interativo
|
| Observacao inicial: a solicitacao de uma resposta ou comando ao usuario
| sempre se faz atraves de tres tipos de predicados, de forma geral
| com as seguintes funcoes:
|    - "ask_": posicionar a questao;
|    - "rd_": obter entrada valida;
|    - "respond_": manipular a entrada validada
+-------------------------------------------------------------------------------*/
  is(Goal) :-
     hi_solve(Goal),
     nl, write('***Success.'),
     nl, write('    '),
     again.

  is(_) :-
     nl, write('***Fail').

  which(X: Goal) :-
     findall(X, hi_solve(Goal), Answers),
     Answers \== [],
     nl, write('Answers:'),
     display_answers(Answers),
     nl, write('***End of which.').

  which(_) :-
     nl, write('***Fail.').

/*--------------------
| hi_solve
*/
  hi_solve(Goal) :-
     hisolve(Goal, []).

  hisolve((P, Ps), Rules) :-
     !, hisolve(P, Rules),
     hisolve(Ps, Rules).

  hisolve(not(P), Rules) :-
     !, not(hisolve(P, Rules)).

  hisolve(P, Rules) :-
     sys(P), !,
     call(P).

  hisolve(P, Rules) :-
     fact(P).

  hisolve(P, Rules) :-
     fact(not(P)), !,
     fail.

  hisolve(P, Rules) :-
     clause(P, Ps),
     hisolve(Ps, [rule(P, Ps) | Rules]).

  hisolve(P, Rules) :-
     askable(P),
     not(unique_known(P)),
     isolve(P, Rules).

  isolve(P, Rules) :-
     insted(P), !,
     not(known(P)),
     valid_answer(P),
     ask_is(P, Rules).

  isolve(P, Rules) :-
     ask_which(P, Rules), !,
     recent_told(P),
     retract(recent_told(P)).

/*--------------------
```

```
! consulta ao usuario
*/
  ask_is(P, Rules) :-
     is_template(P, T),
     nl, display_list(T),
     rd_is(Answer), !,
     respond_is(Answer, P, Rules).

  ask_is(P, Rules) :-
     nl, write(' Is '),
     display_fact(P),
     write('? '),
     rd_is(Answer), !,
     respond_is(Answer, P, Rules).

  respond_is(yes, P, Rules) :-
     assert(P),
     assert(was_told(P)).

  respond_is(no, P, Rules) :-
     assert(was_denied(P)), !,
     fail.

  respond_is(continue, P, Rules) :-
     !,
     fail.

  respond_is(why, P, [Rule | Rules]) :-
     display_rule(Rule), !,
     ask_is(P, Rules).

  respond_is(why, P, []) :-
     nl,
     nl, write('***No more explanations possible.'),
     nl, !,
     ask_is(P, []).

  ask_which(P, Rules) :-
     which_question(P, Vars),
     repeat,
     (! continue_ask_which(P, Vars, Answer) !),
     respond_which(P, Answer, Rules).

  continue_ask_which(P, _, end) :-
     unique_known(P).

  continue_ask_which(P, Vars, Answer) :-
     rd_which(P, Vars, Answer).

  which_question(P, Vars) :-
     which_template(P, Vars, T),
     verify_vars(P, Vars),
     nl, display_list(T).

  which_question(P, Vars) :-
     get_vars(P, Varsl),
     elem_or_list(Varsl, Vars),
     nl, write(' Which '),
     display_fact(P),
     write('? ').

  respond_which(_, end, _).

  respond_which(P, why, [Rule | Rules]) :-
     display_rule(Rule), !,
     ask_which(P, Rules).

  respond_which(P, why, []) :-
     nl,
     nl, write('***No more explanations possible.'),
     nl, !,
     ask_which(P, []).

  respond_which(P, _, _) :-
     assert(P),
     assert(was_told(P)),
     assert(recent_told(P)),
     fail.

  /* descobre se ha' a restricao de resposta unica e, nesse caso, se essa
     resposta ja' e' conhecida (exclui o proprio P) */
  unique_known(P) :-
     generalize(P, PG),
     unique_answer(PG, DersG, DedsG),
     rename_vars((PG, DersG, DedsG), (Pl, Dersl, Dedsl)),
     Pl = P,
     insted_arg(Dersl),
     Dersl = DersG.
```

```
    PG,
    insted_arg(DedsG).

/*-------------------------------------------------------------------------+
| explicador                                                               |
+-------------------------------------------------------------------------*/
  explain(P) :-
     P, !,
     explain_1(P, yes).

  explain(P) :-
     explain_1(P, no).

  explain_1(P, YN) :-
     proof(P, E, YN),
     !! how(E, ' ') !!.
     nl, write('***End of explanation.'),
     nl, write('    '),
     again.

  explain_1(_, _).

/*--------------------
| explicacao do sucesso
*/
  proof((P, Ps), E and Es, yes) :-
     !, proof(P, E, yes),
     proof(Ps, Es, yes).

  proof(not(P), not(P), yes) :-
     sys(P), !,
     not(call(P)).

  proof(not(P), not(P) because E, yes) :-
     proof(P, E, no).

  proof(P, system P, yes) :-
     pos(P),
     sys(P), !,
     call(P).

  proof(P, told P, yes) :-
     pos(P),
     was_told(P).

  proof(P, fact P, yes) :-
     pos(P),
     fact(P),
     not(was_told(P)).

  proof(P, P because Es, yes) :-
     pos(P),
     clause(P, Ps),
     proof(Ps, Es, yes).

/*----------------------
| explicacao do insucesso
*/
  proof(not(P), ? not(P) because P, no) :-
     sys(P), !,
     call(P).

  proof(not(P), ? not(P) because E, no) :-
     proof(P, E, yes).

  proof((P, Ps), E and Es, no) :-
     !, decide(YN1, YN2, no),
     proof(P, E, YN1),
     proof(Ps, Es, YN2).

  proof(P, ?? P, no) :-
     sys(P), !,
     not(call(P)).

  proof(P, denied P, no) :-
     pos(P),
     was_denied(P).

  proof(P, fact not(P), no) :-
     pos(P),
     fact(not(P)).

  proof(P, ?? P, no) :-
     pos(P),
     not(clause(P, _)),
     not(was_denied(P)).

  proof(P, ? P because Es, no) :-
```

```
    pos(P),
    not(P),
    clause(P, Ps),
    proof(Ps, Es, no).

decide(yes, no, no).
decide(no, yes, no).
decide(no, no, no).

/*-------------------------------------------------------------------------------+
| explicador passo a passo
+-------------------------------------------------------------------------------*/
how(? P because Ps, Ok) :-
    repeat,
    [! nl,
    nl, write('--------------------'),
    nl, write('  I cannot conclude '),
    display_fact(P),
    nl, write('  because'),
    how_one(Ps, N),
    nl, write('--------------------'),
    nl,
    ask_how(Opt, N) !],
    respond_how(Ps, Opt, Ok).

how(P because Ps, Ok) :-
    repeat,
    [! nl,
    nl, write('--------------------'),
    nl, write('  I have concluded '),
    display_fact(P),
    nl, write('  because'),
    how_one(Ps, N),
    nl, write('--------------------'),
    nl,
    ask_how(Opt, N) !],
    respond_how(Ps, Opt, Ok).

how(denied P, Ok) :-
    nl,
    nl, write('I was denied '),
    continue_how(P, Ok).

how(?? P, Ok) :-
    nl,
    nl, write('Although it was not denied, it is not a fact '),
    continue_how(P, Ok).

how(told P, Ok) :-
    nl,
    nl, write('I was told '),
    continue_how(P, Ok).

how(fact P, Ok) :-
    nl,
    nl, write('It is a fact '),
    continue_how(P, Ok).

how(system P, Ok) :-
    nl,
    nl, write('System: '),
    continue_how(P, Ok).

continue_how(P, Ok) :-
    display_fact(P),
    /* ask_how(Opt, 0), */
    respond_how(_, return, Ok).

how_one(P and Ps, N) :-
    how_1(P and Ps, 1, N).

how_one(P, 1) :-
    how_1(P, 1, _).

how_1(P and Ps, I, N) :-
    how_1(P, I, N),
    inc(I, J),
    how_1(Ps, J, N),
    set_n(Ps, J, N).

how_1(? P because _, I, _) :-
    nl, write('  ('), write(I), write(') I cannot show '),
    display_fact(P).

how_1(P because _, I, _) :-
    nl, write('  ('), write(I), write(') I can show '),
    display_fact(P).
```

```
how_1(denied P, I, _) :-
    nl, write('  ('), write(I), write(') I know it is not true '),
    display_fact(P).

how_1(77 P, I, _) :-
    nl, write('  ('), write(I), write(') I do not know '),
    display_fact(P).

how_1(told P, I, _) :-
    nl, write('  ('), write(I), write(') I know '),
    display_fact(P).

how_1(fact P, I, _) :-
    nl, write('  ('), write(I), write(') I know '),
    display_fact(P).

how_1(system P, I, _) :-
    nl, write('  ('), write(I), write(') I know '),
    display_fact(P).

set_n(_ and _, _, _).
set_n(_, I, I).

respond_how(Ps, I, Ok) :-
    integer(I),
    get_li(Ps, I, Pi),
    how(Pi, Ok), !,
    Ok == commit.

respond_how(_, commit, commit).
respond_how(_, return, _).

get_li(P and Ps, I, Pi) :-
    I =\= 1,
    dec(I, J),
    get_li(Ps, J, Pi).

get_li(Pl and Ps, 1, Pl).
get_li(Pl, 1, Pl).

ask_how(Opt, N) :-
    nl, write('Option? '),
    nl,
    rd_how(Opt, N).

/*-------------------------------------------------------------------------------+
| entrada                                                                        |
+-------------------------------------------------------------------------------*/
/*---------------------
| obtem opcao valida do menu do HIER
*/
  rd_hier(HiOpt) :-
      repeat,
      read(Word),
      valid_hier(Word, HiOpt).

  valid_hier(Word, isolve) :-
      member(Word, [1, isolve]).

  valid_hier(Word, solve) :-
      member(Word, [2, solve]).

  valid_hier(Word, xplain) :-
      member(Word, [3, explain, how, proof]).

  valid_hier(Word, newgoal) :-
      member(Word, [4, newgoal]).

  valid_hier(Word, end) :-
      member(Word, [5, end, exit, quit]).

  valid_hier(_, _) :-
      nl, write('***Invalid option.'),
      nl,
      fail.

/*---------------------
| obtem resposta valida para "ask_is"
*/
  rd_is(Answer) :-
      repeat,
      read(Word),
      valid_is(Word, Answer).

  valid_is(Answer, yes) :-
      member(Answer, [yes, ok]).

  valid_is(Answer, no) :-
```

```
    member(Answer, [no]).

valid_is(Answer, continue) :-
    member(Answer, [continue, cont]).

valid_is(Answer, why) :-
    member(Answer, [why, explain]).

valid_is(_, _) :-
    nl, write('***Invalid answer.'),
    nl,
    fail.

/*--------------------
| obtem resposta valida para "ask_which"
*/
rd_which(P, Vars, Answer) :-
    repeat,
    nl, write('   Answer is '),
    read(Word),
    valid_which(P, Vars, Word, Answer).

valid_which(_, _, Word, end) :-
    member(Word, [end, ok]).

valid_which(_, _, Word, why) :-
    member(Word, [why, explain]).

valid_which(P, Vars, Vars, _) :-
    was_denied(P), !,
    nl, write('   ***You have already denied '),
    display_fact(P),
    fail.

valid_which(P, Vars, Vars, _) :-
    was_told(P), !,
    nl, write('   ***You have already told me '),
    display_fact(P),
    fail.

valid_which(P, Vars, Vars, _) :-
    fact(P), !,
    nl, write('   ***It is already a fact '),
    display_fact(P),
    fail.

valid_which(P, Vars, Vars, _) :-
    unique_known(P), !,
    nl, write('   ***Unique answer constraint.'),
    nl, write('      Fact '),
    display_fact(P),
    write(' refused.'),
    fail.

valid_which(P, Vars, Vars, nil) :-
    valid_answer(P).

valid_which(_, _, _, _) :-
    nl, write('   ***Invalid answer.'),
    fail.

/*--------------------
| obtem opcao valida para o explicador passo a passo
*/
rd_how(Opt, N) :-
    repeat,
    read(Word),
    valid_how(Word, N, Opt).

valid_how(Word, _, return) :-
    member(Word, [return, back, up]).

valid_how(Word, _, commit) :-
    member(Word, [commit, end, ok]).

valid_how(I, N, I) :-
    I >= 1,
    I =< N.

valid_how(_, _, _) :-
    nl, write('***Invalid option.'),
    nl,
    fail.

again :-
    get0(C),
    C =:= 59, !,
    nl.
```

```
    nl, write('Backtracking...'),
    nl,
    fail.

  again.
/*-------------------------------------------------------------------------+
| saida                                                                    |
+-------------------------------------------------------------------------*/
/*--------------------
| apresentacao do HHIER
*/
  /* mostra cabecalho */
  display_header :-
     nl,
     nl, write(' Highly Interactive Expert Resolver'),
     nl, write(' MAMR, 1989').

  /* mostra menu */
  display_hier(Goal) :-
     nl,
     nl, write('I am trying to solve'),
     nl, write('     '), display_fact(Goal),
     nl,
     nl, write(' Choose one:'),
     nl, write('   (1) Interactive resolution'),
     nl, write('   (2) Non-interactive resolution'),
     nl, write('   (3) Explanation'),
     nl, write('   (4) Change goal'),
     nl, write('   (5) End').

/*--------------------
|
*/
  display_answers([]).

  display_answers([H | T]) :-
     nl, write('    '), write(H),
     display_answers(T).

  display_rule(rule(P, Ps)) :-
     nl,
     nl, write('I want to solve the following rule:'),
     nl,
     nl, write('--------------------'),
     nl, write('  IF'),
     display_conjunction(Ps),
     nl, write('  THEN'),
     nl, write('     '), display_fact(P),
     nl, write('--------------------'),
     nl.

  display_conjunction((P, Ps)) :-
     !, nl, write('     '), display_fact(P),
     write(' AND'),
     display_conjunction(Ps).

  display_conjunction(P) :-
     nl, write('     '), display_fact(P).

  display_fact(P) :-
     name_var(P, Pl),
     read_as(Pl, Text), !,
     display_list(Text).

  display_fact(P) :-
     name_var(P, Pl),
     write(Pl).

  display_list([H | T]) :-
     write(H),
     display_list(T).

  display_list([]).
/*-------------------------------------------------------------------------+
| utilitarios                                                              |
+-------------------------------------------------------------------------*/
/*--------------------
| definicoes gerais
*/
  known(P) :-
     fact(P).

  known(P) :-
     fact(not(P)).

  known(P) :-
```

```
    was_denied(P).

  fact(P) :-
    clause(P, true).

  pos(not(_)) :-
    !, fail.

  pos(_).

/*--------------------
! verifica se um predicado (diferente de "true") e' predicado do sistema
*/
  sys(P) :-
    P \== true,
    functor(P, F, N),
    system(F/N).

/*--------------------
! verifica se todos os argumentos de um predicado estao instanciados
*/
  insted(P) :-
    P =.. [H | T],
    insted_arg(T).

  insted_arg([]).

  insted_arg([H | T]) :-
    nonvar(H),
    insted_arg(T).

/*--------------------
! verifica se todos os argumentos de um predicado que forem variaveis
! aparecem, sozinhos e na mesma ordem, em uma lista
*/
  verify_vars(P, Vars) :-
    get_vars(P, V),
    elem_or_list(V, Vl), !,
    Vl == Vars.

/*--------------------
! obtem uma lista com as variaveis nao instanciadas de um predicado
*/
  get_vars(P, Vars) :-
    P =.. [H | T],
    get_var_l(T, [], VarsI),
    invert_list(VarsI, Vars).

  get_var_l([], Vars, Vars).

  get_var_l([H | T], Vars, NewVars) :-
    var(H),
    get_var_l(T, [H | Vars], NewVars).

  get_var_l([H | T], Vars, NewVars) :-
    get_var_l(T, Vars, NewVars).

/*--------------------
! obtem o elemento de uma lista unitaria ou a propria lista, caso esta
! possua mais de um elemento
*/
  elem_or_list(List, List) :-
    List = [_ | [_]].

  elem_or_list([Elem], Elem).

/*--------------------
! generaliza um predicado (transforma variaveis em novas variaveis e
! constantes em variaveis)
*/
  generalize(P, PG) :-
    functor(P, F, Arity),
    functor(PG, F, Arity).

/*--------------------
! inverte uma lista
*/
  invert_list(L, LI) :-
    invert_list_l(L, [], LI).

  invert_list_l([], L, L).

  invert_list_l([H | T], L, LI) :-
    append([H], L, Ll),
    invert_list_l(T, Ll, LI).

/*--------------------
! concatena duas listas
*/
```

```
append([], L, L).
append([H | T], L, [H | Tl]) :-
    append(T, L, Tl).

/*-------- ------------
| descobre se um elemento e' membro de uma lista
*/
member(X, [X | _]).
member(X, [_ | Y]) :-
    member(X, Y).
```

```
/*-------------------------------------------------------------------------+
| Highly Interactive Expert Resolver                                       |
| + manipulacao de variaveis                                               |
|    (Antonio Luz Furtado, PUC/RIO)                                        |
| Pontificia Universidade Catolica do Rio de Janeiro                       |
| Marco Antonio Mortari Rezende 8828613                                    |
+-------------------------------------------------------------------------+
| 21/09qua1989 |
+--------------*/

rename_vars(X,Y) :-
  listvar(X,L1),
  copl(L1,L),
  cop2(X,L,Y), !.

copl([],[]).
copl([V|R],[[V,W]|S]) :-
  copl(R,S).

cop2(X,L,Y) :-
  case([atomic(X) -> Y = X,
        functor(X,F,N) -> (functor(Y,F,N), cop3(0,N,X,L,Y)),
        var(X) -> (cop4(X,Xl,L), Y = Xl)]).

cop3(N,N,X,L,Y) :- !.
cop3(I,N,X,L,Y) :-
  J is I + 1,
  arg(J,X,A),
  arg(J,Y,B),
  cop2(A,L,B),
  cop3(J,N,X,L,Y).

cop4(X,Xl,[[Y,Xl]|R]) :-
  X == Y, !.
cop4(X,Xl,[F|R]) :-
  cop4(X,Xl,R).

v_on(X,[Y|Z]) :-
  var(Y), !,
  (X == Y ;
   v_on(X,Z)).
v_on(X,[X|_]).
v_on(X,[Y|Z]) :-
  v_on(X,Z).

card([],0).
card([A|R],N) :-
  card(R,M),
  N is M + 1 .

no_dup([X|Xs],Ys) :-
  v_on(X,Xs),
  no_dup(Xs,Ys).
no_dup([X|Xs],[X|Ys]) :-
  not v_on(X,Xs),
  no_dup(Xs,Ys).
no_dup([],[]).

v_flat(A,B,[A|B]) :-
  var(A), !.
v_flat(A,B,B) :-
  atomic(A), !.
v_flat([A|R],B,C) :- !,
  v_flat(R,B,T),
  v_flat(A,T,C).
v_flat(A,B,C) :-
  A =.. [F|R], !,
  v_flat(R,B,C).

listvar(X,Y) :-
  v_flat(X,[],T1),
  invert_list(T1,T2),
  no_dup(T2,T3),
  invert_list(T3,Y).

name_var(F0, F) :-
  rename_vars(F0, F),
  listvar(F,L),
  name_var1(L,1).
```

```
name_var1([],I).
name_var1([X|R],I) :-
   J is I + 1,
   int_text(I,K),
   concat($VS,K,X),
   name_var1(R,J).
```

Bibliografia

[Cla 84] Clark, K. L., e McCabe, F. G. micro-PROLOG: Programming in Logic. Prentice-Hall International, Englewood Cliffs, N. J., EUA, 1984.

[Fur 86] Furtado, A. L., e Souza, C. R. VM/Prolog, etc - adding an expert tool capability to VM/Prolog. Em Proceedings of the ITL Conference on Expert Systems. 1986.

[Ham 84] Hammond, P., e Sergot, M. apes: Augmented Prolog for Expert Systems. apes 1.1 Reference Manual. micro-PROLOG version. Logic Based Systems Ltd., Richmond, Inglaterra, julho de 1984.

[Mar 86] Marcus, C. Prolog Programming. Addison-Wesley Publishing Company, Inc., Reading, Mass., EUA, 1986.

[Sen 89] Sena, G. J. Expert systems shells in Prolog. Em preparo.

[Ste 86] Sterling, L., e Shapiro, E. The Art of Prolog. MIT Press, Mass., 1986.

[Tzo 88] Tzoar, D., e Walker, A. The SYLLOG Expert Database System Version 0.71. Notes for Users. IBM T. J. Watson Research Center, Yorktown Heights, N. I., EUA, 1988.

[Wal 87] Walker, A. (Ed.). Knowledge Systems and Prolog: A Logical Approach to Expert Systems and Natural Language Processing. Addison-Wesley Publishing Company, Inc., Reading, Mass., EUA, 1987.