UM COMPILADOR DE REGRAS PARA
GERAÇÃO DE UM SISTEMA ESPECIALISTA
COM ENCADEAMENTO REGRESSIVO

Leliane Nunes de Barros

Orientador:

Prof.Dr. Antonio Eduardo Costa Pereira

Dissertação apresentada ao Instituto
de Ciências Matemáticas de São Carlos,
Carlos, da Universidade de São Paulo,
para obtenção do título de mestre
em Ciências de Computação.

São Carlos

Dezembro de 1986

AGRADECIMENTOS

Agradeço a atenção e orientação dispensada pelo Prof. Dr. Antonio Eduardo Costa Pereira; a Prof. Dra. Dóris Ferraz de Aragon pela complementação teórica dada ao trabalho e as professoras Maria Carolina Monard e Maria das Graças Volpi Nunes pelas críticas que contribuíram para melhorar esta dissertação.

Este trabalho foi patrocinado pelas
Instituições: CNPq , CAPES , FINEP,
FAPESP (processo no. 85/1240 - 3).

INDICE

RESUMO	1
INTRODUÇÃO	2
CAPITULO I	
Conceitos Gerais	4
Alguns Sistemas Especialistas	5
Características de um Sistema Especialista	7
Tipos de Engenhos de Inferência	10
Encadeamento Progressivo	10
Encadeamento Regressivo	14
Raciocínio Inexato	17
CAPITULO II	
A Base de Conhecimento	21
CAPITULO III	
Como Implementar um Engenho de Inferência de Encadeamento Progressivo	29
Como Implementar um Engenho de Inferência de Encadeamento Regressivo	33
CAPITULO IV	
Unificação do Engenho de Inferência com a Base de Conhecimento	40

CAPITULO V

O Compilador de Conhecimento	52
------------------------------------	----

CAPITULO VI

Esboçando uma Teoria de Diagnose.....	63
---------------------------------------	----

APENDICE

O Programa Compilador de Regras	74
---------------------------------------	----

Uma Base de Conhecimento sobre Vinhos	80
---	----

Como Usar o Compilador de Regras	86
--	----

REFERENCIAS BIBLIOGRAFICAS	87
----------------------------------	----

RESUMO

Neste trabalho introduzimos uma maneira original de contruir um Sistema Especialista: transformar uma Base de Conhecimento em um programa PROLOG. Enquanto que todos os Sistemas Especialista encontrados na bibliografia são controlados por um "shell", obtemos, com essa transformação, um Sistema Especialista cujo controle é feito pelo próprio PROLOG. Apresentamos também, um protótipo de um compilador para automatizar a referida transformação.

ABSTRACT

This work introduces a way of transforming directly a Knowledge Base into a PROLOG program. With the method presented here we obtain an Expert System, the control of which is carried out by PROLOG itself, without requiring the use of a shell. We also present a model of a compiler which automates the mentioned method of transformation.

INTRODUÇÃO

Um Sistema Especialista (SE) é um programa que se comporta como um especialista humano em algum domínio específico do conhecimento. Especialista é qualquer pessoa que possua vasta experiência em resolver problemas em uma área limitada da ciência ou tecnologia, sendo que esta experiência lhe permite tirar conclusões certas até mesmo nos casos em que não existem métodos precisos para se procurar a solução de um problema. Da mesma forma que o especialista humano, um SE também deve ser capaz de tratar com domínios cujas informações e métodos são incompletos e inexatos. Exemplos de domínios frequentemente tratados por SE incluem: diagnose médica, projetos eletrônicos e análises científicas. No capítulo I, expomos as origens dos Sistemas Especialistas, suas principais características, apresentando alguns dos mais conhecidos e as principais técnicas usadas para implementá-las.

Uma característica de SE que é sempre destacada é a sua modularidade, uma vez que ele é normalmente formado por uma Base de Dados, uma Base de Conhecimento e um Engenho de Inferência. O Engenho de Inferência, comumente chamado de "shell" (esqueleto de um SE), é o responsável pela automatização do raciocínio do sistema e tem sido, desde o início, a preocupação maior daqueles que querem construir um SE. No entanto, sabe-se hoje que a maior dificuldade e complexidade está em se construir a Base de Conhecimento, isto é, como extrair o conhecimento do especialista humano e como organizá-lo para ser usado pelo sistema. No capítulo II

mostramos um protótipo de uma Base de Conhecimento sobre "vinhos", e discutimos sua organização.

No capítulo III apresentamos dois tipos de "shell" : um engenho de encadeamento progressivo e outro de encadeamento regressivo, que são tradicionalmente usados já há duas décadas.

Um "shell", por ser um módulo distinto da Base de Conhecimento, é na verdade um interpretador de regras. Com isso, o SE torna-se lento, do mesmo modo que as linguagens de programação interpretadas são mais lentas do que as compiladas. No capítulo IV está a maior contribuição desta dissertação: mostramos uma maneira original de transformar uma Base de Conhecimento em um programa PROLOG [Clark-84] e desta maneira conseguimos um sistema que resolve problemas sem fazer uso de um "shell". Assim, conseguimos um apreciável ganho em termos de tempo de processamento.

Finalmente no capítulo V, apresentamos um protótipo do compilador que transforma regras de conhecimento em um programa PROLOG equivalente. Este compilador permite que o engenheiro de conhecimento use em nosso sistema com a mesma metodologia que se utiliza em sistemas baseados em "shell" e com a qual está acostumado. Em poucas palavras, ele pode entrar com suas regras em nosso sistema da mesma forma que o faria em sistemas baseados em "shell".

No capítulo VI, esboçamos uma teoria sobre diagnose e justificamos de maneira informal que a metodologia que criamos está correta.

CAPITULO I

CONCEITOS GERAIS

Sistemas Especialistas tiveram sua origem nos Provedores Automáticos de Teoremas. Para provar teoremas, os matemáticos recorreram a lógica simbólica e suas regras de inferência. Porém, o raciocínio lógico não somente se aplica para assuntos da matemática, mas para resolver problemas de qualquer área do conhecimento humano. Assim, os métodos que foram descobertos para provar teoremas foram sendo aplicados em outras áreas de Inteligência Artificial, assim como análise e síntese de programas e resolvedores de problemas.

O primeiro resolvidor de problemas significativo foi GPS [Ernst-69], um programa criado por Newell, Shaw e Simon em 1957. GPS podia resolver jogos e quebra-cabeças e ainda provar teoremas em Cálculo de Predicados. GPS foi criado para ser geral, para resolver problemas independente do domínio em questão. No entanto, havia problemas que GPS não podia resolver e por isso seus criadores resolveram adicionar outros métodos ao GPS. Surgiu então a seguinte questão: os métodos de GPS, assim como todos os métodos em Inteligência Artificial, são tentativas de imitar o raciocínio humano, e isso tem sido uma tarefa bastante difícil.

Dessa forma, GPS pecou por querer ser geral. No entanto, ele ainda hoje é considerado um programa inteligente, a ponto de recentemente Simon, um de seus criadores, ter ganhado o prêmio Nobel da Economia, por seus métodos terem sido aplicados com sucesso, nesta área.

1.1 Alguns Sistemas Especialistas

Em 1964, foi criado DENDRAL [Buchanan-78], um sistema usado pela Química para gerar todas as configurações possíveis dos átomos de um determinado composto. Criado na Universidade de Stanford por Joshua Lederberg e Feigenbaum, DENDRAL, ao contrário de GPS, é um resolvidor de problemas de um domínio específico, ou o que chamamos hoje de "Sistema Especialista". Seu grande sucesso e eficiência provaram o poder de seus métodos e o tornaram modelo para tantos outros. Também Lederberg recebeu um prêmio Nobel: o de Química.

Outro sistema que obteve sucesso foi MYCIN [Shortliffe-76] : um Sistema Especialista sobre infecções bacteriológicas do sangue. Podemos dizer, neste caso, que as tarefas do médico consistem em:

1. verificar através dos sintomas do paciente se ele sofre de infecção bacteriológica;
2. qual é o organismo responsável;
3. quais drogas são apropriadas;
4. quais podem ser aplicadas.

MYCIN coleta dados do paciente e resultados de laboratório e chega a uma solução. Ele mostra sua conclusão juntamente com o grau de confiança que nela tem. Para operar com suas medidas de incerteza, MYCIN se utiliza de alguns métodos heurísticos. O sistema pode ainda responder perguntas sobre a sua

linha de raciocínio, fornecendo ainda referências de artigos e outras publicações. Ele possui cerca de 500 regras do tipo:

" Se a infecção é bacteremia primária e

se o lugar do cultivo é um dos lugares estéreis e

se o lugar de penetração do organismo é o trato-intestinal

então podemos supor (0.7) que a identidade do organismo é

bacteróide."

Outro sistema bastante conhecido é o R1, também conhecido por XCON [MacDermott-80] da Digital Equipment Corp (DEC), criado em 1980 e usado para configurar computadores da linha VAX. A DEC, ao produzir o VAX, quis dar liberdade ao freqüês para escolher os equipamentos de sua instalação particular. Muitos detalhes precisavam ser tratados, o que requeria um grande conhecimento da linha de equipamentos oferecidos e de suas restrições. Para substituir uma grande equipe de especialistas eletrônicos é que foi criado o R1.

Ainda outro Sistema Especialista que obteve êxito foi PROSPECTOR [Duda-79], criado em 1978 em Stanford por Duda e Hart. PROSPECTOR auxilia geólogos em suas pesquisas para encontrar depósitos de minerais sólidos.

1.2 Características de um Sistema Especialista

Todos esses Sistemas Especialistas têm sido comercializados e usados por profissionais de suas áreas. Seus desempenhos competem com os de profissionais humanos. Isso vem salientar uma

importante característica de SE: o que mais se espera de um Sistema Especialista é que ele tenha o mesmo desempenho de um profissional da área, seja qual for a dificuldade da tarefa.

Dentre as principais características dos SE nós podemos citar as seguintes:

1. Tradicionalmente, a arquitetura de SE é baseada nos sistemas de produção e consiste de:

- (a) uma coleção de fatos (a Base de Dados);
- (b) um conjunto de regras de produção (a Base de Conhecimento);
- (c) um sistema de controle (o Engenho de Inferência ou "shell").

2. Para simular um especialista humano, o SE deve contar com as informações dadas pela pessoa que o consulta. Assim, o sistema terá que lhe fazer perguntas, durante seu raciocínio, para que o usuário lhe forneça os fatos que o ajudem a resolver o problema;

3. Um SE, assim como um especialista humano, deve saber operar com dados imprecisos e inexatos. Assim, ele coleta fatores de certeza que expressam o grau de confiança que o usuário tem nas suas informações, para serem processados juntamente com os

fatores de certeza dados pelo especialista às regras do conhecimento;

4. O sistema deve ser capaz de explicar ao usuário como chegou às suas conclusões e porque o sistema precisa de uma particular informação. Um médico, por exemplo, deseja saber como um determinado diagnóstico foi tirado a partir dos dados forneci-

dos; um geólogo pode querer saber porque precisa fornecer uma determinada informação que lhe exija testes caros e trabalhosos. Além dessas explicações, outra seria de grande importância: porque não foi tirada uma particular conclusão ao invés da conclusão fornecida. Este tipo de explicação pode algumas vezes ser mais elucidativo do que as outras citadas acima e assim fazer com que o usuário confie definitivamente no resultado;

5. A Base de Conhecimento de um SE deve crescer com o tempo, do mesmo modo como um especialista humano aumenta seus conhecimentos constantemente. Portanto, uma característica importante é que Sistemas Especialistas possuam facilidades para modificar e expandir seus fatos e regras da Base de Conhecimento.

Nós poderíamos ainda incluir como característica de Sistemas Especialistas o fato de operarem sobre uma grande Base de Conhecimento. Assim como em tantas outras áreas da Ciência de Computação, manipular conhecimento tem mostrado ser uma tarefa delicada e que requer técnicas apropriadas. Em vista dessas dificuldades, existe hoje uma área dedicada a este assunto: a Engenharia de Conhecimento. Entre as técnicas por ela empregada, a mais popular é representar o conhecimento através de regras do tipo "SE-ENTÃO", desde que qualquer que seja o conhecimento

passível de um algoritmo [Tärnlund-77], pode ser expresso através dessas regras.

Regras se-então têm a forma:

SE A e B e C e D

ENTÃO G e H e I e J.

onde A, B, C e D são as condições ou antecedentes e G, H, I e J são as conclusões ou consequentes.

Um exemplo de regras desse tipo é:

SE chove e

venta

ENTÃO

molha e

faz frio.

É comum utilizar-se de regras SE-ENTÃO para representar o conhecimento que o sistema precisa para agir como especialista. Eis como poderiam ser representados conhecimentos sobre a escolha de vinhos, usando regras:

r1: SE o prato principal é ave e

não tem peru

ENTÃO a melhor cor é tinto.

r2: SE tem molho e

o molho é cremoso

ENTÃO o melhor corpo é pesado.

1.3 Tipos de Engenheiros de Inferência

Para resolver problemas através das regras do conhecimento, o computador precisa de um esquema para saber como e quando utilizar-se delas. Tal esquema é denominado método de inferência. Os métodos de inferência mais populares são o encadeamento progressivo ("forward chainer") e, o encadeamento regressivo ("backward chainer"). A diferença entre os dois métodos está na direção em que se procura a solução de um problema.

Qualquer que seja o problema a ser resolvido pelo computador, devemos fornecer-lhe os dados ou informações a respeito das condições da atual consulta. Assim, para chegarmos a solução de um problema podemos fazer um encadeamento progressivo, a partir dos dados, utilizando as regras do conhecimento; ou então, levantamos a hipótese da validade de uma determinada solução e, utilizando as regras do conhecimento, fazemos um encadeamento regressivo a partir da hipótese, constatando a validade dos fatos, o que significa provar que a hipótese levantada é a solução do problema.

1.3.1 Encadeamento Progressivo

Um grande número de SE requerem o uso do método de encadeamento progressivo. O mais famoso desses sistemas é o XCON. Ele configura computadores a partir dos dados do freguês e, "trabalhando para frente", fornece uma configuração baseada sobre

aqueles dados. Neste exemplo, o encadeamento progressivo é o único a ser aplicado, uma vez que levantar hipóteses a respeito de configurações possíveis, levaria o computador a uma explosão combinatória.

Vejamos como resolver um problema parecido com o que XCON resolve, usando o método de encadeamento progressivo. Para isso, inicialmente o sistema deve ter disponível um conjunto de fatos tais como:

f1: PROLOG requer editor.

f2: PROLOG é ferramenta de Inteligência Artificial.

f3: o trabalho vai ser realizado por um Sistema Especialista.

f4: Sistema Especialista é da área de Inteligência Artificial.

f5: PROLOG requer computador.

f6: computador tem video.

f7: computador tem impressora.

E um conjunto de regras:

r1: SE o trabalho vai ser realizado por um X e

X é da área de Y

ENTAO

vai trabalhar com Y.

r2: SE vai trabalhar com Y e

Z é ferramenta de Y

ENTAO :

vai usar Z e deve adquirir Z.

r3: SE vai usar Z e

Z requer Y

ENTAO :

deve adquirir Y.

r4: SE deve adquirir Y e

Y tem Z

ENTAO :

deve adquirir Z.

onde X, Y e Z são variáveis.

As regras são examinadas uma após a outra. Quando as condições de uma regra estão no conjunto de fatos, as conclusões da regra são consideradas verdadeiras e adicionadas ao conjunto de fatos. No nosso exemplo, as condições da regra r1 são todas verdadeiras, pois correspondem aos fatos f3 e f4 do conjunto dado. Assim, a conclusão de r1, passa a ser um novo fato:

f8: vai trabalhar com Inteligência Artificial.

Da mesma forma usando a regra r2, cujas condições correspondem aos fatos f8 e f2, suas conclusões passam a ser os fatos:

f9: vai usar PROLOG.

f10: deve adquirir PROLOG.

Com a regra r3 e os fatos f9 e f1, concluimos o fato:

f11: deve adquirir editor.

Como o sistema tenta fazer todos os casamentos possíveis dos fatos com as conclusões da regra, no caso de r2, também casam os fatos f9 e f5 com suas condições, o que nos permite concluir ainda outro fato através da mesma regra:

f12: deve adquirir computador.

Com a regra r4 e fazendo o casamento de suas condições, ora com f12 e f6, ora com f12 e f7, concluimos:

f13: deve adquirir video.

f14: deve adquirir impressora.

Assim, em adição ao conjunto inicial de fatos, o sistema termina por inferir, através do encadeamento progressivo sobre as regras do conhecimento, os seguintes novos fatos:

f8: vai trabalhar com Inteligência Artificial.

f9: vai usar PROLOG.

f10: deve adquirir PROLOG.

f11: deve adquirir editor.

f12: deve adquirir computador.

f13: deve adquirir video.

f14: deve adquirir impressora.

Com esse exemplo notamos que no encadeamento progressivo, o sistema parte dos fatos e caminha para as conclusões.

1.3.2 Encadeamento Regressivo

Já no caso de um domínio no qual se aplica o encadeamento regressivo, o que temos inicialmente é um conjunto de hipóteses. Como exemplo, vejamos como o sistema faria para diagnosticar se um paciente está com catapora ou sarampo. Para isso, duas hipóteses são levantadas:

h1: o paciente tem catapora.

h2: o paciente tem sarampo.

São necessárias ainda, algumas informações sobre o estado do paciente. Esses fatos são fornecidos inicialmente ao sistema, ou como é mais frequente neste tipo de encadeamento, o sistema pede as informações ao usuário sempre que delas necessitar. No nosso exemplo, as informações são:

f1: o paciente tem febre.

f2: o paciente tem corisa.

f3: o paciente tem manchas avermelhadas no corpo.

São fornecidas as seguintes regras sobre as duas doenças:

r1: SE o paciente possui vesículas e
o paciente tem febre e
o paciente tem dores articulares
ENTÃO o paciente tem catapora.

r2: SE o paciente possui exantemas e
o paciente tem febre e
o paciente tem inflamação nas vias
inflamatórias
ENTÃO o paciente tem sarampo.

r3: SE o paciente tem bolhas de água no corpo
ENTÃO o paciente possui vesículas.

r4: SE o paciente tem manchas avermelhadas no corpo
ENTÃO o paciente possui exantemas.

r5: SE o paciente tem coriza
ENTÃO o paciente tem inflamações nas vias
respiratórias.

No encadeamento regressivo, o sistema examina as regras uma a uma e quando as hipóteses casam com as conclusões de uma regra, são estabelecidas novas hipóteses que são as condições daquela regra. Assim, o sistema prossegue recursivamente até que as hipóteses estabelecidas correspondam aos fatos e, com isso, a hipótese inicial fica provada. Desta forma, selecionando a regra r2 cuja conclusão corresponde à hipótese h2, estabelecemos mais três hipóteses:

h3: o paciente possui exantemas.

h4: o paciente tem febre.

h5: o paciente tem inflamação nas vias respiratórias.

Como h4 corresponde ao fato f1, h4 está provada. O sistema deve agora provar h3 e h5 para validar a hipótese inicial de que o paciente tem sarampo.

Usando a regra r4 cuja conclusão corresponde a h3, estabelecemos mais uma hipótese:

h6: o paciente tem manchas avermelhadas no corpo.

que é imediatamente provada por corresponder ao fato f3. Portanto, fica provada a hipótese h3.

Resta ainda, provar h5. Com a regra r5, estabelecemos a hipótese:

h7: o paciente tem coriza.

que também é imediatamente provada por corresponder ao fato f2, o que prova a hipótese h5.

Portanto, com as hipóteses h3, h4 e h5 provadas, a hipótese inicial h2 torna-se válida.

Procedendo de maneira análoga e independente, podemos facilmente verificar que a hipótese h1 não pode ser validada. Portanto, o sistema conclui que o paciente tem sarampo.

Com essa ilustração, podemos observar claramente a direção em que o sistema trabalha usando o encadeamento regressivo: ele parte dos objetivos e caminha em direção aos fatos.

iniciais. Se há mais regras para considerar, ele examina a primeira; se a regra não for detonável, ele recorre a seguinte; se a regra for detonável, os fatos novos que foram inferidos são adicionados à Base de Dados, e o engenho recomeça com todas as regras e a nova Base de Dados; se a regra for detonável, mas os fatos inferidos já pertencem à Base de Dados, o engenho recorre a seguinte regra (evitando que o sistema entre em "looping")."

O sistema deve considerar ainda que para resolver um problema em que uma estrutura é modificada até se chegar a uma configuração desejada (por exemplo, no jogo da velha onde a configuração desejada é uma em que haja uma trinca alinhada de O ou X), uma das regras deve inferir um fato (PARE) que avise ao sistema que a solução foi encontrada e que o sistema deve parar de operar; se o engenho esgotar todas as regras e não conseguir inferir a solução desejada, então deve ser enviada uma mensagem de falha.

Usando uma pseudo-linguagem próxima ao PROLOG e um esquema de explanação do tipo "top-down" (partindo de definições mais abrangentes para as mais simples), vejamos a seguir como um engenho progressivo pode ser implementado. As palavras em negrito são chamadas de rotinas, descritas mais adiante.

forward 1

O sistema conseguiu resolver o problema e os fatos inferidos são listados,

PROLOG é uma linguagem de programação capaz de usar regras através do uso de encadeamento regressivo. E tanto pela teoria quanto pela experiência, sabe-se que é possível resolver qualquer problema computável usando PROLOG. Isto quer dizer que, se o problema for bem conhecido e existir por trás dele uma teoria capaz de produzir algoritmos, podemos expressá-lo por meio de regras "SE-ENTÃO" e resolvê-lo com segurança usando PROLOG, ou outro sistema baseado em encadeamento regressivo ou até mesmo uma linguagem procedural.

Uma faceta interessante da programação lógica é como podemos implementar um sistema de encadeamento progressivo usando PROLOG, uma linguagem baseada em encadeamento regressivo. O artigo de [Merritt-86], mostra uma maneira simples de implementar em PROLOG um sistema do tipo XCON da DEC que trabalha através de encadeamento progressivo.

1.4 Raciocínio Inexato

No caso de problemas onde não existe teoria capaz de gerar algoritmo, algumas pessoas tiveram a idéia de trabalhar com regras e dados aos quais está associado um fator de certeza. Estas regras ou dados nem sempre são dignas de confiança absoluta. Há vários formalismos para tratar esses fatores de certeza, mas o que usaremos aqui é um dos mais simples e populares e segue o modelo de incerteza usado por MYCIN [Shortliffe, 1976]. Para explicá-lo, admitamos que se deseje construir um SE em vinhos. Este sistema deverá ser capaz de

recomendar um vinho da mesma forma que um oinologista o faria e, para isto, poderá usar regras como as seguintes:

r1: SE

a cor recomendada é tinto e

o corpo recomendado é medio e

a doçura recomendada é doce

ENTAO

o vinho é Gamay com 0.95.

r2: SE

a cor recomendada é tinto e

o corpo recomendado é médio e

a doçura recomendada é médio

ENTAO

o vinho é Gamay com 0.9.

r3: SE

a cor recomendada é branco e

o corpo recomendado é médio e

a doçura recomendada é doce

ENTAO

o vinho é Chablis com 0.85.

Para facilitar as explicações a seguir, consideremos a definição: em uma regra, a conclusão e as condições são compostas por um atributo e seu valor. Por exemplo, o atributo vinho assume valores Gamay ou Chablis, enquanto o atributo cor recomendada assume valores tinto ou branco.

O valor entre 0 e 1 em cada regra, chamado fator de certeza indica a confiança que se deposita na conclusão, quando se tem certeza absoluta das evidências representadas pelas condições. Por exemplo em r1, o fator de certeza é de 0.95 do vinho ser Gamay se a cor recomendada é tinto, o corpo recomendado é médio e a doçura recomendada é doce. Este fator de certeza nós chamaremos de fator de certeza do especialista. Porém, a cada condição de uma regra também corresponde um fator de certeza, seja ela uma informação dada pelo usuário cujo fator de certeza também deve ser fornecido por ele, seja ela uma hipótese a ser verificada por uma outra regra com seu respectivo fator de certeza. Desta maneira, o sistema deve calcular o fator de certeza da hipótese acumulado pelas condições da regra. Vamos traduzir a regra r1 para uma cláusula PROLOG de forma que o sistema possa reservar variáveis que recebam os fatores de certeza citados:

```
vinho(Gamay,0.95,Cert_Acum) :-
```

```
    cor_recomendada(tinto,Cert_Espec1,Cert_Acum1) ,  
    corpo_recomendado(medio,Cert_Espec2,Cert_Acum2) ,  
    doçura_recomendada(doce,Cert_Espec3,Cert_Acum3) ,  
    minimo(Cert_Acum4,[Cert_Acum1,Cert_Acum2,Cert_Acum3]),  
    Cert_Acum is 0.95 * Cert_Acum4.
```

Cada condição passa a ser verificada por outra regra, onde será calculado o fator de certeza acumulado (Cert_Acum). Para calcular o fator de certeza acumulado em uma regra, procedemos da seguinte maneira: escolhemos o mínimo entre os fatores de

certeza acumulados das condições da regra e formamos o produto dele com o fator de certeza do especialista, resultando no fator de certeza acumulado pela regra. Caso o fator de certeza acumulado pela regra seja menor do que 0.2, o sistema desiste de provar a hipótese por aquela regra. Caso uma conclusão resultar numa pergunta ao usuário, o fator de certeza acumulado será aquele fornecido pelo usuário. Quando uma das hipóteses for uma cláusula negada, o fator de certeza acumulado será o complemento de um, do fator de certeza da cláusula.

O sistema pode inferir mais do que uma mesma hipótese com fatores de certeza diferentes. Por exemplo, as regras r1 e r2 podem permitir inferir duas hipóteses iguais: "o vinho é Gamay com 0.5" ou "o vinho é Gamay com 0.7". Para combinar as duas hipóteses o sistema faz o seguinte cálculo:

$$CF = 0.5 + 0.7 - 0.5 \times 0.7$$

ou seja, a soma menos o produto dos fatores a serem combinados, o que resulta em um número também entre 0 e 1. Intuitivamente, podemos interpretar este cálculo, admitindo domínios cujos eventos são independentes. Observando que a mesma hipótese pode ser verificada pela regra r1 ou pela regra r2, o que significa que as duas contribuem para aumentar o grau de certeza na hipótese, daí a soma, mas que uma regra "independe" da outra, daí a subtração do produto.

No caso de existirem mais do que duas hipóteses iguais, o sistema combina o resultado da primeira combinação com o fator de certeza da próxima hipótese a ser combinada e assim prossegue recursivamente.

CAITULO II

A BASE DE CONHECIMENTO

Em princípio, qualquer formalismo consistente no qual possamos expressar o conhecimento sobre algum domínio pode ser usado em um Sistema Especialista. No entanto, o uso de regras "SE-ENTÃO", também chamadas de regras de produção por Post em 1936, é o formalismo mais comumente usado para representar conhecimento. Além disso, uma regra de produção pode ser uma cláusula lógica [Kowalski-79] e receber tratamentos de inferência lógica como o princípio de resolução [Robinson-65].

Alguns domínios são facilmente representados por regras de produção. Tais domínios são aqueles de conhecimento empírico, ou seja, onde em suas regras, a associação de uma premissa e sua conclusão está baseada em observações de associações passadas. Isto é o que chamamos de representação superficial do conhecimento [Michaelson-85]. Como exemplo podemos citar o diagnóstico de doenças infecciosas. Em muitos casos, um médico não explica o porque de seu diagnóstico ou terapia, mas conta suas experiências passadas e se apoia em dados estatísticos.

Uma representação profunda do conhecimento [Michaelson-85], contendo causa e efeito, pode ser através de regras de produção, mas, no entanto, há formalismos mais poderosos para isso (estruturas de "frame" ou redes semânticas). Este é o caso do SE DENDRAL [Buchanan-78]. Sistemas Especialistas que possuam este tipo de representação são, sem dúvida alguma, mais poderosos

e confiáveis.

Há, contudo, domínios onde a teoria é pouco sólida, como em áreas da medicina. A este tipo de domínio, um SE baseado em regras se aplica muito bem.

O uso de PROLOG [Clark-84] em SE baseado em regras, resulta numa enorme vantagem: um conjunto de regras de conhecimento passa a ser um programa PROLOG e pode ser executado através dos mecanismos de inferência de PROLOG. Isto é o que veremos no capítulo IV.

Contudo, poucas pessoas usariam um sistema que se comunicasse em uma linguagem de computação. O sistema deve ser capaz de conversar em um razoável subconjunto da linguagem natural. Isto facilita a compreensão do sistema tanto do usuário como do especialista responsável por transferir o conhecimento.

Para conciliar essa necessidade com as necessidades do sistema em questão deve ser construído um "compilador de regras" responsável por traduzir as regras escritas em um subconjunto da linguagem natural para a forma que o sistema deseja. Por exemplo, no caso do "shell" de encadeamento progressivo - capítulo III - o sistema interpreta as regras escritas em forma de uma lista, contendo o nome da regra, os antecedentes e os consequentes; no caso do "shell" de encadeamento regressivo - capítulo III - o sistema interpreta as regras escritas como cláusulas PROLOG; e no caso do programa que propomos no capítulo IV, o sistema não interpreta regras, mas as têm incorporadas. No primeiro caso, já propomos em trabalho anterior, um "editor de regras" [Barros-86], que traduz regras para listas. No último caso propomos um protótipo do "compilador de regras" no capítulo V.

O assunto que escolhemos para construir o protótipo de um SE baseado em regras refere-se a escolha do melhor vinho em uma refeição, cuja Base de Conhecimento mostramos no Apêndice. Como já vimos, a cada regra deve ser atribuído um fator de certeza - entre 0 e 1 - que corresponde ao grau de confiança que o especialista deposita na regra. Por exemplo, sejam as seguintes regras do tipo SE-ENTÃO:

SE a cor recomendada é branco e
o corpo recomendado é leve e
a doçura recomendada é seco

ENTÃO :

o vinho recomendado é Chablis,
com fator de certeza 1.

SE a melhor cor é tinto e
a cor preferida é tinto

ENTÃO :

a cor recomendada é tinto,
com fator de certeza 1.

SE o prato principal não é peixe e
tem molho e
o molho é de tomate

ENTÃO :

a melhor cor é tinto,
com fator de certeza 0.9.

que na forma adotada pelo nosso sistema ficam:

e1: vinho(Chablis) com 1 se

cor_recomendada(branco) e

corpo_recomendado(leve) e

doçura_recomendada(seco).

e2: cor_recomendada(tinto) com 1 se

melhor_cor(tinto) e

pergunte(cor_preferida(tinto)) .

e3: melhor_cor(tinto) com 0.9 se

pergunte(não prato_principal(peixe)) e

pergunte(tem_molho) e

pergunte(molho(tomate)).

Enumeramos a seguir os principais aspectos da linguagem que adotamos para descrever a Base de Conhecimento:

1. Na frente de cada regra é colocada a referência da explicação correspondente a regra, que por sua vez também deve ser declarada junto à Base de Conhecimento. Por exemplo, as explicações devem ser declaradas na forma:

explicação (e1, "Para encontrar um vinho para a
sua refeição, preciso determinar
as seguintes características :
cor, corpo e docura recomendadas").

explicação (e2, "Para determinar qual é a cor recomendada do vinho, preciso determinar a melhor cor e a sua cor de vinho preferida").

explicação (e3, "Para determinar a melhor cor do vinho, preciso saber qual será o prato principal, se tem molho e de que tipo ele será").

2. Em cada conclusão ou condição devem ser explicitados o atributo e seu valor na forma de uma estrutura do tipo:

⟨ATRIBUTO⟩ (⟨VALOR⟩)

que no nosso exemplo, temos:

vinho(tinto)
cor_recomendada(tinto)
doçura_recomendada(seco)

ou, no caso de ser o atributo que não possui valor, como no caso de tem_molho, que assume valores de verdadeiro ou falso, a estrutura passa a ter a forma simples:

⟨ATRIBUTO⟩

que no exemplo, temos:

tem_molho

3. No caso de uma condição corresponder a uma pergunta feita ao usuário, isso deve ser expresso na forma:

pergunte(<ATRIBUTO> (<VALOR>))

ou

pergunte(<ATRIBUTO>)

que no exemplo, temos:

pergunte(cor_preferida(tinto))

pergunte(tem_molho)

4. Quando uma condição for uma negação, deve ser colocado o não diante do atributo:

não <ATRIBUTO> (<VALOR>)

ou

não <ATRIBUTO>

e se a condição a ser negada for uma pergunta, a estrutura fica:

pergunte (não <ATRIBUTO> (<VALOR>))

ou

pergunte (não <ATRIBUTO>)

que no exemplo, temos:

pergunte(não prato_principal(peixe))

Junto com a Base de Conhecimento devem ser declaradas os tipos de perguntas a serem feitas ao usuário. No protótipo aqui apresentado, usamos três tipos de perguntas:

a. Perguntas do tipo `sn` - cuja resposta do usuário deve ser `sim` ou `não`;

b. Perguntas do tipo `escolha_incerta` - cuja resposta do usuário deve ser uma das apresentadas na lista de opções, junto com o fator de certeza correspondente a sua escolha;

c. Perguntas do tipo `escolha_certa` - cuja resposta também deve ser uma das apresentadas na lista de opções, mas que não requer fator de certeza do usuário.

As declarações dos tipos de perguntas e suas listas de opções devem estar presentes na Base de Conhecimento na seguinte forma:

```
perg( <ATRIBUTO> , <LISTA_DE_OPÇÕES> , <TIPO> )
```

Vejamos alguns exemplos de perguntas do protótipo que construímos sobre vinhos:

```
perg(cor_preferida,[tinto,branco,rosel],escolha_certa).
```

```
perg(prato_principal,[carne,ave,peixe],escolha_incerta).
```

```
perg(tem_molho,[sim,não],sn).
```

```
perg(molho,[muito_temperado,doce,cremoso,tomate],  
      escolha_incerta).
```

Finalmente, ainda com a Base de Conhecimento deve ser informado ao sistema quais são os atributos que serão objetos de consulta. Esses atributos devem ser declarados na forma:

```
gols( <LISTA_DE_ATRIBUTOS> ).
```

No nosso exemplo, temos:

```
gols([vinho]).
```


Tudo o que foi dito sobre a linguagem usada para descrever a base de conhecimento pode ser resumido na forma de Backus-Naur:

```
<regra> ::= <nome_da_explicação>: <frase_simples>  
                                     com <certeza> se <cauda>.  
  
<regra> ::= <frase_simples>.  
  
<cauda> ::= <frase_simples>  
  
<cauda> ::= <frase_simples> e <cauda>  
  
<frase_simples> ::= <estrutura>  
  
<frase_simples> ::= nao <estrutura>  
  
<frase_simples> ::= pergunte ( <estrutura> )  
  
<frase_simples> ::= pergunte ( nao <estrutura> )  
  
<estrutura> ::= <atributo>  
  
<estrutura> ::= <atributo> ( <argumentos> )  
  
<argumentos> ::= <valor>  
  
<argumentos> ::= <valor> , <argumentos>  
  
<valor> ::= <constante atômica do PROLOG>  
  
<valor> ::= <lista de constantes atômicas do PROLOG>  
  
<atributo> ::= <átomo do PROLOG>  
  
<certeza> ::= <real entre 0 e 1>  
  
<nome_da_explicação> ::= <valor>
```

CAPITULO III

3.a COMO IMPLEMENTAR UM ENGENHO DE INFERENCIA DE ENCADEAMENTO PROGRESSIVO

O encadeamento progressivo é um engenho de inferência "modus ponens": avança dos dados para as conclusões. Este modo de procedimento responde à questão: "O que nós podemos concluir dos dados?"

A priori este tipo de engenho é utilizado em sistemas que não interagem com o usuário durante seu raciocínio. Ele faz uma suposição fundamental: que toda a informação que ele necessitará estará disponível no início.

Em um sistema de encadeamento progressivo, dizemos que uma regra do tipo SE-ENTÃO é "detonável" quando todas as suas condições são fatos já pertencentes a Base de Dados, e assim ela pode ser usada para inferir fatos novos (suas respectivas conclusões).

A operação de um engenho progressivo pode ser explicada facilmente:

"Dado um conjunto de regras e fatos, o engenho tenta tirar todas as possíveis conclusões (fatos novos). Ele começa examinando as regras que lhe são dadas. Se ele não encontrar mais nenhuma regra para ser considerada, ele sai com as conclusões encontradas - a lista de fatos - que também inclui os fatos

forward 2

O sistema não conseguiu atingir a condição de parada mas os fatos inferidos são listados,

SE

não há mais regras para extrair mais fatos.

forward 3

Encontre-fatos-novos da primeira regra da lista de regras e use-os,

SE

nenhuma das situações anteriores acontecerem.

forward 4

Chame recursivamente forward para o resto das regras,

SE

a primeira regra não puder encontrar fatos novos.

encontre-fatos-novos

Uma lista de fatos novos é retirada de uma regra,

SE

a regra for detonável e pode-usar os fatos de sua conclusão.

detonável

Uma regra é detonável,

SE

classifique-antec puder classificar os anteceden-
tes, ou como uma lista vazia, ou uma função executável (e então
executá-la), ou como um fato já pertencente à Base de Dados.

pode-usar

Pode usar alguns fatos pertencentes à lista dos conse-
quentes da regra,

SE

eles não pertencerem à Base de Dados.

use 1

Chame forward para o resto das regras sem alterar a
lista de fatos,

SE

a lista dos fatos novos da regra analisada for
vazia.

use 2

Chame forward para todas as regras novamente, mas com a
lista acrescida dos fatos novos retirados da regra excluída,

SE

a lista dos fatos novos não for vazia.

Este algoritmo foi retirado do livro de Peter Sell [Sell-85]. No entanto para ser possível sua implementação foi importante introduzir dois aspectos relevantes:

(a) o sistema deve saber que a uma determinada mensagem (por exemplo: PARE), enviada pelas regras à lista de fatos, a solução foi encontrada;

(b) o sistema deve reconhecer, entre os fatos da lista de antecedentes ou de consequentes, uma função executável.

3.b . COMO IMPLEMENTAR UM ENGENHO DE INFERENCIA DE ENCADEAMENTO REGRESSIVO

O encadeamento regressivo é um engenho de inferência "modus tollens": retrocede das hipóteses ou objetivos para os dados (desde que nós saibamos aonde queremos chegar, é claro). Este modo de proceder responde a questão: "Podemos provar as hipóteses a partir dos dados fornecidos?" .

Quando fazemos encadeamento regressivo a partir das hipóteses, são estabelecidas sub-hipóteses, as quais são tratadas do mesmo modo como as hipóteses principais.

As hipóteses, assim como as regras, são específicas do domínio, e precisam ser fornecidas ao sistema. O encadeamento regressivo não requer que todos os dados ou fatos estejam disponíveis desde o início; se ele não encontra o que precisa, ele pergunta ao usuário. Desta forma, este sistema é tipicamente

interativo.

A operação de um sistema de encadeamento regressivo é explicada a seguir:

"O sistema recebe um conjunto de hipóteses. Sua missão é retornar uma lista contendo somente as hipóteses válidas. Qualquer hipótese (ou sub-hipótese) que passa a ser válida é adicionada na Base de Dados (depois de ter certeza que ela já não está lá). Para verificar se um objetivo é válido, o engenho tenta usar os métodos abaixo, na seguinte ordem:

1. se a hipótese (ou sub-hipótese) já foi provada, ela é válida;

2. se sua validade já foi perguntada ao usuário e a resposta foi negativa, uma vez que ela não estava na lista dos provados, ela não é válida;

3. se há alguma regra relevante que poderia ser usada para provar sua validade, ela é disparada;

4. caso não aconteça nenhum dos casos anteriores, o engenho finalmente pergunta a validade da hipótese, diretamente ao usuário.

Analisemos mais de perto o passo 3 da especificação acima. Nesse passo, começa-se procurando uma regra relevante para a hipótese, ou seja, uma regra em que ela apareça entre os consequentes. Para verificar a validade de cada um dos antecedentes desta regra, o engenho faz uma recursão, tratando os antecedentes (sub-hipóteses), do mesmo modo como ele trata as hipóteses principais."

As definições em PROLOG que seguem expressam a especificação acima, com a vantagem de implementar o raciocínio nebuloso. O modelo de incerteza usado aqui é do tipo usado por MYCIN e já foi discutido no capítulo I.

Como exemplo, usaremos a Base de Conhecimento sobre vinhos discutida no capítulo II e listada no Apêndice. Assim uma de suas regras é do tipo:

```
SE  a cor recomendada é branca e
    o corpo recomendado é leve e
    a doçura recomendada é seco
ENTÃO o vinho indicado é Chablis com
      fator de certeza 1.
```

que em PROLOG , e na forma em que o engenho implementado espera, fica:

```
vinho (cf,A,1,chablis) :-
    cor_recomendada (cf,A1,E1,branco) ,
    corpo_recomendado (cf,A2,E2,leve) ,
    doçura_recomendada (cf,A3,E3,seco) ,
    A minimo (A1,A2,A3).
```

onde A,A1,...,E1,E2,... são variáveis PROLOG. Assim uma das hipóteses do problema seria :

```
vinho (cf,A,E,chablis)
```

onde vinho é um predicado ou atributo; chablis é o valor do predicado; A o fator de certeza (cf) acumulado pelo sistema. E é o fator de certeza dado pelo especialista.

No nosso exemplo, a lista de hipóteses principais que

queremos validar, será:

```
vinho (cf,A1,E1,gamay)
vinho (cf,A2,E2,chablis)
vinho (cf,A3,E3,savignon_blanc)
vinho (cf,A4,E4,charbonay)
vinho (cf,A5,E5,soave)
vinho (cf,A6,E6,riesling)
```

·
·
·

O programa prova deve retornar a lista das hipóteses que conseguiram ser provadas, com seus respectivos fatores de certeza ordenados, como por exemplo:

```
vinho (cf,0.72,1,gamay)
vinho (cf,0.670671,1,riesling)
vinho (cf,0.342,1,soave)
```

Como a cada hipótese principal corresponde uma regra, o sistema usa o predicado pré-definido de PROLOG, `findall`, para tentar provar todas as hipóteses através das regras, sem que a lista de hipóteses lhe seja dada explicitamente. Assim, o engenho inicia sua execução, através de prova com a seguinte consulta feita a PROLOG:

```
prova ( vinho (cf,A,E,Valor) ).
```

Para provar cada hipótese principal, o engenho, através de PROLOG, seleciona uma regra relevante, uma que tenha como consequente (ou cabeça, para PROLOG) o predicado `vinho`. A seguir o engenho passa a tentar provar cada uma das premissas da regra selecionada recursivamente, através de prova, estabelecendo novas hipóteses. As novas hipóteses, podem agora ser dos seguintes tipos:

1. primitivas: são predicados pré-definidos em PROLOG e portanto, correspondem a hipóteses a serem provadas por PROLOG e não por prova;

2. de decisão: são aquelas que possuem coeficientes de certeza, como as hipóteses principais, e portanto devem ser provadas por prova;

3. perguntáveis: são aquelas, cujo valor do predicado e fator de certeza devem ser perguntados diretamente ao usuário ou, no caso de sistemas complexos, devem receber informações de outros sistemas eletrônicos;

4. definidas: são hipóteses que não possuem fatores de certeza .

Nas hipóteses de decisão, prova procura provar todos os predicados de mesmo nome mas com valores e cf acumulados diferentes. As hipóteses provadas, cujo produto entre cf acumulado e cf do especialista for menor que 0.2 são abandonadas; hipóteses que possuem o mesmo valor para um dado predicado são substituídas por uma única hipótese, cujo cf acumulado é a soma menos o produto dos fatores a serem combinados.

Quando o sistema faz uma pergunta ao usuário, o engenheiro prevê que este lhe pergunte porque aquela informação é necessária. Para este caso, o "backward" constrói uma lista das cabeças de regras que foram usadas durante seu raciocínio. Assim, para consecutivos "porques" do usuário, o sistema vai lhe mostrando cada elemento desta lista até não possuir mais explicações.

O programa prova

```

sep ((P , Resto), P, Resto) :- !.

sep (P, P , verdade).

prova (verdade, Expl).

prova (Conjunção, Expl) :-
    sep (Conjunção, P, Resto) ,
    provado (P) , ! ,
    prova (Resto, Expl).

prova (Conjunção, Expl) :-
    sep (Conjunção, P, Resto) ,
    primitiva (P) , ! ,
    P ,
    prova (Resto, Expl).

prova (Conjunção, Expl) :-
    sep (Conjunção, P, Resto) ,
    P =.. [Fn, cf, AIR] ,
    clause (P, Corpo) , ! ,
    findall ([cf, A1, E1|R1], (P1 =.. [Fn, cf, A2, E1|R1] ,
        clause (P1, Corpo1) ,
        prova (Corpo1, [P|Expl]) ,
        A1 is A2 * E1) , Lista)

    merge (Lista, Lista1) ,
    max ([cf, AIR], Lista1) ,
    assert (provado (P)) ,
    prova (Resto, Expl).

prova (Conjunção, Expl) :-
    sep (Conjunção, P, Resto) ,
    clause (P, Corpo) ,
    prova (Corpo, [P|Expl]) ,
    assert (provado (P)) ,
    prova (Resto, Expl).

prova (Conjunção, Expl) :-
    sep (conjunção, P, Resto) ,
    perguntavel (P) ,
    foidito (P, Expl) ,
    assert (provado (P)) ,
    prova (Resto, Expl).

```

```
prova (Conjunção, Expl) :-  
    sep (Conjunção, not P, Resto) , !  
    not (prova (P, Expl)) ,  
    assert(provado(not P)) ,  
    prova(Resto, Expl).
```

CAPITULO IV

UNIFICAÇÃO DO ENGENHO DE INFERENCIA COM A BASE DE CONHECIMENTO

O "shell" que acabamos de descrever no CAPITULO III, como já dissemos, é um interpretador das regras do conhecimento (capítulo II). Por esse motivo, este é um processo lento.

Vamos apresentar uma maneira diferente de construir um Sistema Especialista com encadeamento regressivo. Ele se constituirá de um único programa, cujo conhecimento e engenho de inferência se misturam. Assim ao invés de se construir um programa, o "shell", que faça inferências sobre a base de regras, faremos um programa que permite que o próprio PROLOG faça as inferências.

Construir um programa único para SE é mais fácil quando se trabalha com PROLOG, fazendo uso de suas principais características:

1. Uma regra de produção representando um elemento do conhecimento é naturalmente traduzida para uma cláusula PROLOG;
2. PROLOG possui seu próprio engenho de inferência de encadeamento regressivo.

Além disso, é importante saber que já existem compiladores PROLOG (ARITY, ALS, QUINTUS e TURBO PROLOG) que geram códigos tão otimizados quanto "C", para programas que trabalham

com conhecimento.

A primeira vista pode parecer difícil representar conhecimento que faça seu próprio tratamento nebuloso. No entanto, vejamos a seguir como isso pode ser simples.

Vamos inicialmente examinar domínios cujas proposições satisfaçam o seguinte princípio:

P é provável se
entre as proposições possíveis
P é a mais provável.

Este princípio significa que para concluirmos que P é provável temos de encontrar todas as possíveis proposições pertencentes a mesma classe que P e, através dos fatores de certeza, mostrar ser P a mais provável delas.

Vejamos, como exemplo, as seguintes regras de conhecimento sobre vinhos:

e1: cor_recomendada(tinto) com 0.8 se
melhor_cor(tinto).

e2: cor_recomendada(branco) com 0.8 se
melhor_cor(branco).

e3: cor_recomendada(tinto) com 1 se
melhor_cor(tinto) e
pergunte(cor_preferida(tinto)).

e4: cor_recomendada(branco) com 1 se
melhor_cor(branco) e
pergunte(cor_preferida(branco)).

Proposições do tipo pergunte indicam perguntas ao usuário do SE: a `cor_preferida` deve ser perguntada diretamente ao usuário, enquanto a `melhor_cor` deve ser determinada através de um conjunto similar de regras, onde se levam em conta as características da refeição (`prato_principal`, `sabor`, `molho`, ...).

Utilizando o modelo de verdade descrito no princípio acima, as regras de `cor_recomendada` serão traduzidas para o seguinte programa PROLOG (PROLOG Padrão):

```
cor_recomendada_provavel(CFX,Valor) :-  
    findall(V,cor_recomendada_possivel(V),L) ,  
    max(cor-recomendada,c(CFX,Valor),L).
```

```
cor_recomendada_possivel(c(CFX,tinto)) :-  
    melhor_cor_provavel(CFY,tinto) ,  
    CFX = 0.8 * CFY.
```

```
cor_recomendada_possivel(c(CFX,branco)) :-  
    melhor_cor_provavel(CFY,branco) ,  
    CFX = 0.8 * CFY.
```

```
cor_recomendada_possivel(c(CFX,tinto)) :-  
    melhor_cor_provavel(CFX,tinto) ,  
    cor_preferida(tinto).
```

```
cor_recomendada_possivel(c(CFX,branco)) :-  
    melhor_cor_provavel(CFX,branco) ,  
    cor_preferida(branco).
```

A primeira cláusula, cor_recomendada_provável, chamaremos de cláusula-cabeça das regras do conhecimento que possuem o mesmo atributo como conclusão (no nosso exemplo o atributo é cor_recomendada); o conjunto de regras cor_recomendada se transformaram no conjunto de cláusulas cor_recomendada_possível. As explicações (e1,e2,e3,e4) serão esquecidas por hora para simplificar a discussão, mas serão inseridas no programa mais adiante.

O atributo cor_recomendada_possível passa a ter com argumento uma estrutura especial do tipo c(CFX,Valor), que é usada para amarrar o fator de certeza ao correspondente valor de cor_recomendada. Dizemos que uma estrutura especial é uma hipótese a respeito de cor_recomendada, por exemplo.

Através da cláusula-cabeça, usando findall, PROLOG constrói uma lista L de todos os termos V de cor_recomendada_possível, que serão da forma c(CFX,Valor). Assim L será uma lista de hipóteses a respeito do atributo provável, que no caso é cor_recomendada.

Para obter cada termo V, PROLOG prova cada cláusula de cor_recomendada_possível para retornar seus respectivos fatores de certeza. Por exemplo, em:

```
cor_recomendada_possivel(c(CFX,tinto)) :-  
    melhor_cor_provavel(CFY,tinto) ,  
    CFX = 0.8 * CFY.
```

PROLOG, prova melhor_cor_provável(CFY,tinto) retornando o valor de CFY para depois multiplicá-lo pelo fator de certeza 0.8 dado pelo especialista àquela regra e assim, finalmente, retornar com CFX.

Nas cláusulas cujo fator de certeza dado na regra original for igual a 1, o produto acima foi omitido (como no caso das duas últimas cláusulas).

Ao retornar de findall a lista L das hipóteses possíveis, max procura em L o termo c(CFX,Valor) de maior fator de certeza (CFX). Para isso max chama merge para processar pares de termos de mesmo Valor, através dos cálculos descritos no final do capítulo I. Depois de retirar termos de valores repetidos, max ordena os termos em ordem decrescente em relação aos seus fatores de certeza para finalmente retornar o de máximo fator de certeza.

No conjunto de cláusulas que calculam o termo de máximo fator de certeza, foi permitido (por não se usar "!") que PROLOG faça seu "backtracking" para utilizar os outros termos em caso de falha.

Note que cor_preferida, por ser uma pergunta do tipo escolha_certa (CAP.II), depois de traduzida não possui fator de certeza. Qualquer atributo que não seja uma pergunta, como melhor_cor, deve receber no programa, uma variável referente ao seu fator de certeza.

Assim, se quisermos provar que a cor recomendada é tinto, temos que provar em PROLOG:

```
cor_recomendada_provavel(CFX,tinto)
```

Através de max, PROLOG procura a hipótese sobre cor recomendada que contém valor tinto e retorna CFX que indica o quão provável é a cor recomendada ser tinto. Caso max não consiga

encontrar uma hipótese com valor tinto, a chamada acima falha, ou seja, `cor_recomendada_provavel(tinto)` não pode ser provada.

Vejamos agora o conjunto de regras sobre `melhor_cor` onde surgem outros problemas de tradução.

```
e5: melhor_cor(tinto) com 0.9 se
    pergunte(prato_principal(carne)) e
    pergunte(nao tem_vitela).
```

```
e6: melhor_cor(branco) com 1 se
    pergunte(prato_principal(peixe)).
```

```
e7: melho_cor(tinto) com 0.9 se
    pergunte(nao prato_principal(peixe)) e
    pergunte(tem_molho) e
    pergunte(molho(tomate)).
```

que em PROLOG ficam:

```
melhor_cor_provavel(CFX,Valor) :-
    findall(V,melhor_cor_possivel(V),L) ,
    max(melhor_cor,c(CFX,Valor),L).
```

```
melhor_cor_possivel(c(CFX,tinto)) :-
    prato_principal(CFY,carne) ,
    not(tem_vitela) ,
    CFX = 0.9 * CFY.
```

```
melhor_cor_possivel(c(CFX,branco)) :-
    prato_principal(CFX,peixe).
```

```
melhor_cor_possivel(c(CFX,tinto)) :-
```

```
    prato_principal(CFY,peixe) ,
```

```
    CFZ = 1 - CFY ,
```

```
    tem_molho ,
```

```
    molho(CFW,tomate) ,
```

```
    minimo(CF, [CFZ,CFW]) ,
```

```
    CFX = 0.9 * CF.
```

Portanto, a cada conjunto de regras que possuam conclusões de mesmo atributo, é construída uma cláusula cabeça - atributo e valor provável - e um corpo dos atributos e valores possíveis.

Note que `melhor_cor` não depende de mais nenhum atributo "provável", mas somente de perguntas feitas ao usuário. Essas perguntas, como já mostrado no capítulo II sobre a base de conhecimento, são classificadas pelo próprio especialista como perguntas que requerem respostas do tipo: sim ou não (sn), escolha_certa ou escolha_incerta.

A tradução das regras diferem de acordo com seu tipo. Por exemplo: `tem_vitela` e `tem_molho` são perguntas do tipo sn e, assim como as do tipo `escolha_certa`, não possuem variáveis para seus fatores de certeza depois da tradução.

No conjunto de regras dado acima, nós temos:

```
pergunte(nao tem_vitela)
```

que foi traduzido por:

```
not(tem_vitela)
```

Uma vez que `tem_vitela` é uma pergunta do tipo `sn`, e não possui fator de certeza, a sua negação pode ser verificada através do `not` de `PROLOG`, que é a negação por falha, ou seja, se a resposta for `sim`, `tem_vitela` tem sucesso e `not tem_vitela` falha; se a resposta for `não`, `tem_vitela` falha e `not tem_vitela` tem sucesso. Isso é melhor entendido quando olhamos o programa que é construído para cada pergunta declarada, de acordo com o seu tipo. Vejamos por exemplo o programa `tem_vitela`:

```
tem_vitela :-  
    pergunta(tem_vitela, _, Certeza) ,  
    EQ(Certeza,0) , 1 , fail.  
tem_vitela.
```

O programa chama `pergunta` que efetivamente fará a pergunta ao usuário ou verificará se já foi respondida. O programa associa certeza "1" a resposta `sim` e certeza "0" a resposta `não`. A primeira cláusula de `tem_vitela`, verifica se a certeza é igual a "0" e então não tenta outra cláusula devido ao corte (!), e assim faz com que `tem_vitela` falhe ("fail"), uma vez que neste caso a resposta foi negativa. Caso a certeza não seja igual a "0", e portanto será "1" (resposta `sim`), `PROLOG` executa a segunda cláusula de `tem_vitela` que sempre tem sucesso por não possuir condições.

Este exemplo de programa é construído para todas as perguntas do tipo `sn`.

Vejamos agora o caso das perguntas do tipo `escolha_certa`, como por exemplo, `cor_preferida` :

```

cor_preferida(Valor) :-
    pergunta(cor_preferida, Valor1, _) ,
    Valor = Valor1 and !.
cor_preferida(_) :- fail.

```

Neste caso, `cor_preferida` não possui fator de certeza como argumento (o "_" em `pergunta` significa que não importa o valor daquele argumento correspondente ao fator de certeza), mas ao contrário do caso anterior, possui um argumento referente ao valor dado àquele atributo, uma vez sua resposta exige uma escolha do usuário sobre uma lista de opções dadas. Assim o que `cor_preferida` pretende provar é se o valor (`Valor`) de seu atributo é o que foi respondido (`Valor1`) pelo usuário através de `pergunta`. Se isso acontecer então `cor_preferida` tem sucesso (o "!" é para evitar o backtracking de PROLOG). Caso `Valor` seja diferente de `Valor1` então PROLOG tenta a segunda cláusula de `cor_preferida` que falha incondicionalmente.

Só falta vermos como é o programa para perguntas do tipo `escolha_incerta`, como por exemplo, `prato_principal`:

```

prato_principal(CFX,Valor) :-
    pergunta(prato_principal, Valor1, CFX) ,
    Valor = Valor1 and !.
prato_principal(CFX, Valor) :- fail.

```

Neste caso, temos que incluir `CFX` como argumento de `prato_principal`, além de seu valor. O resto do programa é perfeitamente análogo ao de `escolha_certa` (`cor_preferida`).

Voltando as regras dadas pelo especialista sobre `melhor_cor`, notamos na última regra a seguinte condição:

```
pergunte(nao prato_principal(peixe))
```

cuja tradução fica:

```
prato_principal(CFY,peixe) ,  
CFZ = 1 - CFY
```

assim, não foi colocado "not" na frente de `prato_principal`, mas foi calculado seu novo fator de certeza `CFZ` que é o complemento de 1 do fator de certeza dado pelo usuário, o que resulta no fator de certeza da negação.

Se quisermos agora introduzir as explicações no programa `cor_recomendada`, temos:

```
cor_recomendada(CFX, Valor, Expl) :-  
    findall(V,cor_recomendada_possivel(V),L) ,  
    max(cor_recomendada, c(CFX,Valor), L).
```

```
cor_recomendada(c(CFX,tinto), Expl) :-  
    melhor_cor_provavel(CFY, tanto, [e1|Expl]) ,  
    CFX = 0.8 * CFY.
```

```
cor_recomendada(c(CFX,branco), Expl) :-  
    melhor_cor_provavel(CFY, branco, [e2|Expl]) ,  
    CFX = 0.8 * CFY.
```

...

Assim, todos os atributos, inclusive as perguntas,

carregam como argumento uma lista de explicações Expl. Em cada regra cujo atributo da cabeça se transformou em "atributo_possivel", a lista de explicações é acrescida da explicação daquela regra. Na realidade esta lista de explicações só será usada no momento em que o SE faz uma pergunta ao usuário e este devolver a pergunta "porque" ao sistema. Desta forma o SE mostrará uma explicação por vez a cada "porque" consecutivo do usuário, até chegar ao fim da lista, quando responde que não há mais explicações para ser dada.

A lista de explicações é inicializada na chamada do SE, como mostraremos a seguir.

Quando o especialista constrói seu arquivo com as regras do conhecimento, deve acrescentar uma sentença, onde estão declarados os objetivos do SE, ou seja, os objetos de consulta sobre os quais o SE pode responder, como por exemplo:

```
gols([vinho, cor_recomendada, corpo_recomendado]).
```

Assim, devemos construir as cláusulas chamadas para cada "gol" declarado. No caso do "gol" vinho, temos a seguinte chamada:

```
vinho :-  
    write("Os vinhos recomendados são:") ,  
    inic_dados ,  
    vinho_provavel(Certeza, Valor, [e0]) ,  
    write(Valor, "com certeza", Certeza), nl,  
    fail.  
vinho :-  
    get0(_).
```

O programa `inic_dados` é responsável por limpar a base de dados do SE referente a última consulta. O programa `vinho_provavel` é que efetivamente calcula o valor do vinho recomendado, bem como seu fator de certeza, e inicializa a lista de explicações com `e0`, que no caso corresponderá à explicação referente àquele "gol". Após escrever na tela o valor e a certeza do vinho recomendado, o programa `vinho` falha ("fail") e através do "backtracking" de PROLOG, tenta encontrar outro par de soluções (valor e fator de certeza). Isso é feito através de `max`, que pega o próximo máximo da lista dos vinhos possíveis. Com isso, o SE fornece uma lista de vinhos recomendados em ordem decrescente de seus fatores de certeza.

Quando todos os vinhos prováveis são encontrados, a segunda cláusula do programa `vinho` espera o usuário apertar uma tecla qualquer para terminar sua consulta.

CAPITULO V

O COMPILADOR DE CONHECIMENTO

Para simplificar a tarefa de se compilar a mão a Base de Conhecimento para o programa descrito no capítulo anterior, mostramos um protótipo de um compilador automático de conhecimento. O programa foi implementado em PROLOG Padrão, cujas facilidades de declaração de operadores (prioridade e associatividade), são muito próprias para a construção de compiladores.

A Base de Conhecimento deve ser organizada na forma descrita no capítulo II, com a ressalva de que todas as regras que possuam o mesmo atributo como conclusão, devem estar arquivadas em sequência (isso pode ser feito por um pré-compilador). Para ler o arquivo onde está armazenada a Base de Conhecimento, PROLOG lê um termo por vez (termo é tudo que termina por um ponto). Chamaremos aqui um termo por sentença. As sentenças, como já classificamos no capítulo II, podem ser:

- (1) explicações;
- (2) declaração dos atributos que serão objetos de consulta ("goals");
- (3) declaração de perguntas;
- (4) regras.

A seguir damos uma explicação das principais cláusulas do programa Compilador de Regras, cuja listagem completa se encontra no Apêndice:

Programa armazene_regras

O programa responsável por identificar cada tipo de sentença e prosseguir no seu tratamento é `armazene_regras`, cujo argumento é uma sentença. Cada cláusula desse programa, depois de compilar uma sentença, lê a próxima sentença e chama recursivamente `armazene_regras` para compilá-la.

`armazene_regras (1)` :

```
armazene_regras(explicação(E,Expl)) :-  
    !, guarde(explicação(E,Expl)),  
    read((Sentença) ,  
    armazene_regras(Sentença).
```

No caso da sentença ser uma explicação, ela já está na forma desejada - um fato PROLOG - e, portanto `armazene_regras` guarda a sentença sem modificá-la.

`armazene_regras (2)` :

```
armazene_regras(gols(Lista_de_gols)) :-  
    constrói_chamadas(Lista_de_gols) ,  
    read(Sentença) ,  
    armazene_regras(Sentença).
```

Quando a sentença for a declaração da lista de "gols", `armazene_regras` constrói as cláusulas chamadas - aquelas que inicializam o SE - para cada "gol" através de `constrói_chamadas`.

armazene_regras (3)

```
armazene_regras(perc(Atributo,Opções,Tipo_de_perc)) :-  
    !, guarde(perc(Atributo,Opções,Tipo_de_perc)) ,  
    trad_perc(perc(Atributo,Opções,Tipo_de_perc)) ,  
    read(Sentença) ,  
    armazeneregras(Sentença).
```

Como no caso da sentença do tipo explicação, a declarações do tipo perc, também devem ser guardadas como fatos de PROLOG. Além disso, é chamado o programa trad_perc que vai construir um programa para cada pergunta declarada (ver capítulo IV).

armazene_regras (4)

```
armazene_regras(E:Regra) :-  
    cabeça(Regra,C) ,  
    functor(C,Fn,Ar) ,  
    elimine(Fn,3) ,  
    cabeçalho(C,Cabeçalho) ,  
    guarde(Cabeçalho) ,  
    guarde_regra(Fn,Ar,E:Regra,Regra1) ,  
    armazeneregras(Regra1).
```

Finalmente, no caso da sentença ser uma regra, armazeneregras separa a cabeça da regra, ou seja, separa a conclusão da regra que, em seguida, é separada em functor

(atributo) e sua aridade. Com isso, ~~elimine~~ apaga todas as regras já compiladas daquele atributo para o caso do compilador ser usado como uma reconsulta. A chamada ao programa cabeçalho, é responsável por construir a cláusula-cabeça das regras que possuem o mesmo atributo como conclusão (a cláusula provável). Depois de guardar o cabeçalho, o programa `guarde_regra` é chamado e será responsável por compilar todas as regras com conclusões de mesmo atributo (o corpo de cláusulas possíveis); `guarde_regra` também lê sentenças do arquivo e quando a sentença lida, ou é uma regra, mas não possui conclusão de mesmo atributo que a regra que entrou em `armazene_regras`, ou ainda, a sentença não é uma regra, `guarde_regra` retorna e passa aquela sentença como argumento para a nova chamada de `armazene_regras`.

Programa `guarde_regra`

O programa `guarde_regra` é que fará a compilação das regras; ele possui quatro argumentos: o functor (1) e a aridade (2) da regra que `armazene_regras` identificou para chamar `guarde_regra` e que com isso pode conferir se a regra que ele ler pertence ao mesmo bloco dos possíveis; a regra a ser compilada (3) e a regra (ou sentença) que faz com que `guarde_regra` pare (4), ou seja, a regra que não tem o mesmo atributo que as anteriores. Vejamos o programa `guarde_regra`:

```

guarde_regra(Fn,Ar,E:Regra,Regra1) :-
    cabeça(Regra,C) ,
    functor(C,Fn,Ar) , ! ,
    trad(E,Regra,Regra_Prolog) ,
    guarde(Regra_Prolog) ,
    read(Prox_Regra) ,
    guarde_regra(Fn,Ar,Prox_Regra,Regra1).

guarde_regra(Fn,Ar,Regra,Regra).

```

Inicialmente, é separada a cabeça da regra do terceiro argumento de `guarde_regra`, para em seguida ser separada em functor e sua aridade. Se o functor e a aridade não forem os mesmos que os da regra anterior (os dois primeiros argumentos de entrada é que indicam isso), então a primeira cláusula falha e é executada a outra cláusula de `guarde_regra` que retorna, copiando a regra do terceiro argumento na do quarto argumento, ou seja, no quarto argumento retorna a regra diferente das anteriores. No caso do functor e a aridade serem os mesmos da regra anterior, `guarde_regra` chama `trad` que fará a tradução da regra para a correspondente regra PROLOG, lê a Próxima Regra e chama recursivamente `guarde_regra`.

Programa trad

Este programa fará a tradução de uma regra para PROLOG. Ele possui três argumentos: a explicação da regra (1); a regra a ser traduzida (2) e a regra PROLOG já traduzida (3). O programa possui uma única cláusula:

```
trad(E,Cab com CF1 se Cauda,(CabProlog :- Cauda Prolog)) :-
```

```
    Cab =.. [Fn,Valor] ,
```

```
    CabProlog =.. [Fn,possivel,c(CF,Valor), Expl] ,
```

```
    tradcauda(E,Expl,CF,CF1,[],Cauda,CaudaProlog).
```

A cabeça Cab da regra, ou seja, a sua conclusão, é separada em functor e valor para ser construída a cabeça PROLOG da regra traduzida, ou seja, sua conclusão. Por exemplo de a cabeça da regra a ser traduzida for:

```
cor_recomendada(tinto)
```

a cabeça PROLOG fica:

```
cor_recomendada(possivel,c(CF,tinto),Expl)
```

A cauda da regra é traduzida por tradcauda.

Programa: tradcauda

Para traduzir a cauda de uma regra, tradcauda possui sete argumentos :

- (1) a explicação E da regra;
- (2) a lista de explicações anteriores;
- (3) a variável para o fator de certeza acumulado;
- (4) o fator de certeza do especialista na regra;
- (5) uma lista para guardar os fatores de certeza de todas as premissas da cauda;
- (6) a cauda da regra - uma conjunção de condições ou predicados;
- (7) a tradução da regra - a regra PROLOG.

Vejamos o programa:

```
tradcauda(E,Expl,CF,CR,CFs,P1 e P2,(Prlg1 , Prlg2)) :-  
    ! , trad1(E,P1,Prlg1,Expl,CFs,NovoCFs) ,  
    tradcauda(E,Expl,CF,CR,NovoCFs,Prlg2).  
  
tradcauda(E,Expl,CF,CR,CFs,P,Prlg) :-  
    trad1(E,P,Prlg1,Expl,CFs,NovoCFs) ,  
    addmin(Prlg1,CF,CR,NovoCFs,Prlg).
```

A primeira cláusula de tradcauda cuida do caso em que a cauda (argumento 6) possui mais do que um predicado - P1 e P2 - onde P2 pode ser outra conjunção de predicados. Assim, tradcauda chama trad1 para traduzir um predicado (P1) e chama recursivamente tradcauda para traduzir o resto da conjunção (P2).

A segunda cláusula de tradcauda, cuida do caso em que a cauda só possui um predicado (os outros já foram traduzidos). Nesse caso, à tradução do último predicado deve ser adicionado o predicado PROLOG que calcula o mínimo dos fatores de certeza das condições da regra. Para isso é que tradcauda reservou a lista de variáveis correspondentes aos fatores de certezas (CFs).

Programa trad1 :

Este programa traduz cada predicado da cauda de uma regra. Os argumentos de trad1 são:

- (1) a explicação da regra na qual uma das condições é o predicado a ser traduzido ;
- (2) o predicado ;

- (3) a tradução do predicado - predicado PROLOG;
- (4) a lista de explicações;
- (5) a lista de fatores de certeza das outras condições;
- (6) a lista de fatores de certeza acrescida do fator de certeza do predicado, caso ele o possua.

Analizemos cada uma das cláusulas de trad1:

trad1 <1>

```
trad1(E,pergunte(Pergunta),PredProlog,Expl,CFs,CFs) :-
    perg(Pergunta,_,sn) , ! ,
    PredProlog =.. [Pergunta,[E|Expl]].
```

Quando o predicado a ser traduzido é uma pergunta e, ao chamar a declaração perg a Pergunta é do tipo sn, o predicado PROLOG é formado pelo atributo pergunta cujo argumento é a lista de explicações com a explicação da regra (E) na cabeça. Como a pergunta do tipo sn não possui fator de certeza, a lista dos CFs não foi alterada.

trad1 <2>

```
trad1(E,pergunte(não Pergunta),not(PredProlog),Expl,CFs,CFs) :-
    perg(Pergunta,_,sn) , ! ,
    PredProlog =.. [Pergunta,[E|Expl]].
```

Este é o mesmo tipo de pergunta que é tratada em trad1

(1), com a diferença de prever a negação da pergunta. Nesse caso, é colocado o not na frente do predicado PROLOG (a negação por falha de PROLOG).

trad1 (3) :

```
trad1(E,pergunte(Pergunta),PredProlog,Expl,CFs,CFs) :-
```

```
    Pergunta =.. [Predicado,Valor],
```

```
    perg(Predicado,_,escolha_certa), ! ,
```

```
    PredProlog =.. [Predicado,Valor,[E|Expl]].
```

Neste caso a Pergunta é identificada como sendo do tipo escolha_certa. Portanto, o predicado PROLOG é formado agora com dois argumentos : o valor do atributo da pergunta e a lista de explicações acrescida da explicação E da regra. Como a pergunta do tipo escolha_certa também não possui fator de certeza, a lista CFs não é alterada.

trad1 (4) :

```
trad1(E,pergunte(não(Pergunta),not(PredProlog),Expl,CFs,CFs) :-
```

```
    Pergunta =.. [Predicado,Valor],
```

```
    perg(Predicado,_,escolha_certa), ! ,
```

```
    PredProlog =.. [Predicado,Valor,[E|Expl]].
```

Este é o caso da pergunta ser do tipo escolha_certa mas negada e trad1 trata a negação como no caso anterior.

trad1 (5) :

```
trad1(E,pergunte(Pergunta),PredProlog,Expl,CFs,[CF|CFs]) :-  
    Pergunta =.. [Predicado,Valor] ,  
    perg(Predicado,_,escolha_incerta), ! ,  
    PredProlog =.. [Predicado,CF,Valor,[E|Expl]].
```

No caso da pergunta ser do tipo `escolha_incerta`, o predicado `PROLOG` é formado por três argumentos: o fator de certeza, o valor do atributo correspondente à pergunta e a lista de explicações acrescida da explicação da regra. Como neste caso o predicado possui fator de certeza, esse é acrescentado à lista dos CFs.

trad1 (6) :

```
trad1(E,pergunte(não Pergunta),(PredProlog,CFY=1-CFX),Expl,CFs,  
                                           [CF|CFs]) :-  
    Pergunta =.. [Predicado,Valor] ,  
    perg(Predicado,_,escolha_incerta), ! ,  
    PredProlog =.. [Predicado,CFX,Valor,[E|Expl]].
```

O tratamento da negação de pergunta do tipo `escolha_incerta` segue o que foi discutido no capítulo 1 sobre raciocínio inexato: ao invés de se usar a negação de `PROLOG`, calcula-se um novo fator de certeza para o predicado, que é o complemento de um do fator antigo.

trad1 <7>

trad1(E,Pred,PredProlog,Expl,CFs,[CF|CFs]) :-

Pred =.. [Fn,Valor] ,

PredProlog =.. [Fn,CF,Valor,[E|Expl]].

Quando o predicado não for pergunta, então ele é um predicado que deve ser verificado por outra regra e portanto, também possui 3 argumentos: fator de certeza, valor e lista de explicações.

CAPITULO VI

ESBOÇANDO UMA TEORIA DE DIAGNOSE

Introdução

Como sabemos, a Inteligência Artificial se propõe a compreender os métodos usados na execução de tarefas que são tipicamente realizados pelo homem e, a partir disso, construir uma teoria que possa ser computável.

Diagnosticar é considerada uma tarefa tipicamente humana. Vamos portanto, como cientistas da área de Inteligência Artificial que somos, fazer uma tentativa de construir uma Teoria da Diagnose.

Inicialmente, introduzimos alguns princípios que ajudam a caracterizar o que chamamos de diagnose feita por um especialista:

Princípio 1 : Devem ser considerados objetos da diagnose e seus atributos;

Princípio 2 : A finalidade do diagnóstico é determinar valores para os atributos;

Princípio 3 : Para fazer um diagnóstico, o especialista em geral parte de hipóteses com respeito aos valores dos atributos;

Princípio 4 : Durante a análise para chegar ao diagnóstico,

O Sistema Especialista que apresentamos antes, é um sistema de diagnose. Observando os princípios descritos acima, olhemos para o nosso sistema como um sistema de diagnose:

São dados ao sistema:

(a) Um objeto de interesse (p.ex.: vinho);

(b) Alguns atributos desse objeto (p.ex.: cor, corpo e doçura);

(c) Um conjunto de valores possíveis para os atributos (p.ex.: cor - branco, tinto, rose; corpo - leve, médio, pesado; doçura - doce, médio, seco).

O Sistema Especialista deseja determinar valores prováveis para os atributos, diante de um determinado domínio. Do conhecimento do domínio e das informações fornecidas, é gerada uma lista de hipóteses possíveis para os valores dos atributos, com um fator de certeza associado a cada uma delas. A Teoria da Diagnose que queremos esboçar é a respeito de como processar esta lista de hipóteses a fim de determinar a mais provável.

Vamos inicialmente descrever uma linguagem para tratar essa teoria que é uma extensão de uma linguagem de primeira ordem:

1. Seja f um predicado, X um termo e N um numeral, então $f(X,N)$ é uma estrutura especial.

2. Seja g um predicado, possível uma constante da linguagem, X um termo e N um numeral, então, $g(\text{possível}, X, N)$ é uma estrutura especial.

3. Se g é uma expressão da linguagem de primeira ordem, então g é uma expressão da linguagem.

A Teoria da Diagnose é uma extensão da Teoria de Conjuntos incluindo números racionais.

Axiomas Próprios

Ax1: Seja f um predicado, Val um argumento e $C \in [0,1]$ então $f(Val,C)$ é uma hipótese com respeito a f .

Ax2: H_f é um conjunto de hipóteses com respeito a f sss

(1) H_f é finito

(2) $\forall n, n \in H_f$ sss n é uma hipótese com respeito a f .

(3) Se $f(Val1,C1)$ e $f(Val2,C2)$ são hipóteses então
se $Val1 = Val2$ então $C1 = C2$.

Definição 1: Se $f(X,Y)$ é uma hipótese com respeito a f , diz-se que (X,Y) é o par associado a $f(X,Y)$.

Definição 2: Se $f(X,Y)$ é uma hipótese com respeito a f , $proj_1(X,Y)$ diz-se o valor da hipótese e $proj_2(X,Y)$ diz-se a certeza da hipótese.

Convenção: H_f e G_f são conjuntos de hipóteses com respeito a f .
 Q, R, S, P são hipóteses.

Definição 3:

$$\text{Val}_f H = \{x \mid \exists P, P \in H_f \wedge P = f(\text{Val}, C) \wedge x = \text{proj}_1^2(\text{Val}, C)\}$$

$$\text{Cert}_f H = \{y \mid \exists P, P \in H_f \wedge P = f(\text{Val}, C) \wedge y = \text{proj}_2^2(\text{Val}, C)\}$$

Definição 4: Sejam A e B conjuntos.

$$A \sim B \text{ sss existe } h \mid h : A \rightarrow B \text{ e } h \text{ é bijetora.}$$

Teorema 1: $H_f \sim \text{Val}_f H_f$.

Teorema 2: $\exists V, V \sim H_f$.

Teorema 3: $|H_f| = |\text{Val}_f H_f|$

Definição 5: Seja H_f um conjunto de hipóteses com respeito a f
 $\text{possivel}(P, f) \text{ sss } P \in H_f$

Teorema 4: $\text{possivel}(P, f) \text{ sss } P \text{ é uma hipótese com respeito a } f$.

Teorema 5: Seja H_f um conjunto de hipóteses com respeito a f .

Se $B \subseteq H_f$ então B é um conjunto de hipóteses com respeito a f .

Teorema 6: Seja H_f um conjunto de hipóteses com respeito a f .

Se $X \in \text{Pot } H_f \Rightarrow X \text{ é um conjunto de hipóteses.}$

Teorema 7: $C1 \in [0, 1] \wedge C2 \in [0, 1] \Rightarrow$

$$C1 + C2 - C1 * C2 \in [0, 1]$$

Vamos usar PROLOG para expressar os resultados dessa teoria porém, vamos mencionar os principais teoremas da teoria correspondentes aos programas PROLOG.

Como vamos construir programas em PROLOG para calcular os diagnósticos, vamos necessitar da noção de listas. Para tanto, usaremos a noção de n-upla associada a um conjunto.

Definição 6: $J = \{1, 2, \dots, n\}$

Definição 7: v é n-upla em A sss $v: J \rightarrow A$

Definição 8: Seja A um conjunto e $|A|^n = n$.

v é uma n-upla associada a A sss v é uma n-upla em A e $v(J) = A$.

Exemplo:

Seja $A = \{a, b, c\}$ então

(a, b, c) é uma n-upla associada a A

(b, a, c) é uma n-upla associada a A

Podemos assim, estabelecer uma teoria de n-uplas :

Definição 9: Seja v uma n-upla associada a A e $x \in A$.

Se $x = v_1$ diz-se que x é a cabeça de v .

Se $x = v_i$, com $i \neq 1$, diz-se que x está na cauda de v .

Comentário: Conjuntos e n-uplas são representados, em PROLOG, por listas.

COMENTARIO 3: Uma vez que um programa em PROLOG é constituído por cláusulas lógicas, eles podem ser vistos por nossa teoria, ora como axiomas, ora como teoremas e lemas a serem provados.

Definição 10: Seja $\text{possivel}(P,f)$ e $\text{possivel}(Q,f)$. P é mais provável que Q se a certeza de $P >$ certeza de Q .

Em PROLOG:

```
mais_provavel(P,Q) :-  
    possivel(P) ,  
    possivel(Q) ,  
    certeza(P) > certeza(Q).
```

Definição 11 : P é diagnóstico de f se P é possível com respeito a f e não há Q tal que Q é mais provável que P .

Em PROLOG:

```
diagnóstico(P,f) :-  
    possivel(P,f) ,  
    não(R mais_provavel P).
```

As definições dadas acima sobre diagnóstico são simples e suficientes para completar nossa Teoria de Diagnose. No entanto, para implementarmos o programa em PROLOG, usamos os seguintes programas:

- (a) diagnóstico ;
- (b) merge ;
- (c) merge1

LEMA1: Ao programa merge1 corresponde uma única função merge1:

Seja H um conjunto de hipóteses com respeito a f e $P \in H$.

merge1: $P \times \text{Pot}(H_f) \Rightarrow \text{Pot}(H_f)$ tal que

$$\text{merge1}(f(V,C), ()) = (f(V,C))$$

$$\text{merge1}(f(V,C), (f(V_1,C_1), f(V_2,C_2), \dots, f(V_n,C_n))) =$$

$$(f(V, C+C_1-C \times C_1), f(V_2,C_2), \dots, f(V_n,C_n)) \text{ se } V=V_1$$

$$\text{merge1}(f(V,C), (f(V_1,C_1), f(V_2,C_2), \dots, f(V_n,C_n))) =$$

$$(f(V_2,C_2), \text{merge1}(f(V,C), (f(V_2,C_2), \dots, f(V_n,C_n)))) \text{ se}$$

$$\text{não } V=V_1$$

A prova do lema 1 é feita por indução sobre o comprimento da n-upla associada ao conjunto H_f .

Em PROLOG:

```
merge1(f(V,C),[],[f(V,C)]).
merge1(f(V,C1), [f(V,C2)|Resto] , [f(V,C)|Resto]) :-
    C is C1 + C2 - C1 * C2.
merge1(f(V1,C1),[f(V2,C2)|Resto],[f(V2,C2)|NResto]):-
    not(V1=V2),
    merge1(f(V1,C1),Resto,NResto).
```

Justificação do programa:

O programa merge1 foi criado para eliminar, da lista de hipóteses colocada em seu segundo argumento (domínio), hipóteses que possuam o mesmo valor que a hipótese selecionada em seu primeiro argumento, retornando o conjunto resultante em seu terceiro argumento (contra domínio).

Vamos inicialmente mostrar que dado um conjunto de hipóteses, apenas uma das sentenças de `merge1` se aplicará a ele. De fato, o conjunto estará forçosamente enquadrado em um dos três casos abaixo:

(1) É vazio. Neste caso só a primeira sentença se aplica.

(2) A cabeça da lista (segundo argumento) que representa a n -upla associada ao conjunto de hipóteses de entrada, tem o mesmo valor V que o elemento do primeiro argumento de `merge1`. Neste caso só a segunda sentença se aplica.

(3) A cabeça da lista (segundo argumento) que representa o conjunto de hipóteses de entrada tem valor diferente do primeiro argumento. Neste caso só a terceira sentença se aplica.

Justifiquemos agora, que para cada caso citado acima, `merge1` retorna em seu terceiro argumento, uma lista que também representa um conjunto de hipóteses:

(1) Se o conjunto for vazio, ele é representado por uma lista vazia e a primeira sentença de `merge1` estabelece uma relação entre ele e o conjunto representado por $[f(V,C)]$;

(2) No caso do conjunto se enquadrar na segunda sentença, o primeiro de seus elementos é substituído. A substituição não altera o valor V de $f(V, _)$ e, portanto, se existia um mapeamento entre $[f(V,C2)|Resto]$ e um conjunto $ConjV$ de valores, tal mapeamento continuará existindo entre $[f(V,C)|R]$ e $ConjV$ e, pelo teorema 7, $C \in [0,1]$;

(3) Seja, agora, um conjunto B do terceiro caso, representado por uma lista $[f(V2,C2)|Resto]$. Se na chamada recursiva

de `merge1` na terceira sentença é gerado um conjunto de hipóteses `NResto`, então o conjunto gerado pela terceira sentença, $[f(V2,C2) | NResto]$ também é um conjunto de hipóteses. Por indução podemos chegar ao conjunto de entrada vazio, cuja situação já foi justificada. A introdução de $f(V2,C2)$ não alterou esse fato. Isso pode ser verificado, observando que $f(V2,C2)$ já pertencia ao conjunto de hipóteses de entrada (cabeça da lista associada a ele), ou seja, $[f(V2,C2) | Resto]$ e portanto, `Resto` não contém $f(V2,C2)$ e `NResto`, por ser o resultado de `merge1` em `Resto`, também não contém $f(V2,C2)$.

LEMA2: Associado ao programa abaixo, há uma função

$$\text{merge} : \{x | x \text{ é uma } n\text{-upla em } H\} \Rightarrow \text{Pot } H \text{ tal que}$$

$$\text{merge}(\emptyset) = \emptyset$$

$$\text{merge}((x_1, x_2, \dots, x_n)) = \text{merge1}(x_1, \text{merge}((x_2, \dots, x_n)))$$

Em PROLOG:

```
merge([], []).
```

```
merge([X|Y], L) :-
```

```
    merge(Y, L1) ,
```

```
    merge1(X, L1, L).
```

Justificação do programa:

Sentença 1: a lista de entrada é vazia. Neste caso, a lista gerada também é vazia e portanto representa um conjunto de hipóteses;

Sentença 2: a lista de entrada não representa um conjunto de hipóteses e para justificar que a lista gerada L é uma representação de um conjunto de hipóteses, façamos por indução:

(1) Admitamos que a chamada recursiva de merge para a lista de entrada menos um elemento, Y , gere uma lista L_1 que represente um conjunto de hipóteses. Como merge1 gera uma lista de hipóteses L a partir de L_1 , então merge de $[X|Y]$ também gera uma lista de hipóteses que é a própria lista L .

(2) Repetimos este processo recursivamente, até chegarmos na lista vazia que já foi justificada pela sentença 1.

O programa abaixo define um conjunto de hipóteses e faz o diagnóstico dele.

Teorema 8: Seja H_f um conjunto de hipóteses com respeito a f . Se v é uma n -upla de hipóteses com respeito a f tal que

$$\text{merge}(v) = H_f \text{ e}$$

$$\max \text{Cert } H_f = C \text{ então}$$

$$f(V, C) \text{ é o diagnóstico de } f.$$

Em PROLOG:

```

diagnóstico(f(V,C)) :-
    findall(f(V1,C1), f(possível, V1,C1), L) ,
    merge(L,L1),
    max(f(V,C),L1).

```

onde $f(\text{possível}, V1, C1)$ produz um par de valor certeza $V1, C1$.

Justificação: Qualquer que seja L , $\text{merge}(L, L1)$ a transforma em um conjunto de hipóteses pelo lema 2. $\text{max}(f(V, C), L1)$ faz com que $f(V, C)$ seja o elemento de $L1$ com maior C . Isto significa que $f(V, C)$ é possível em $L1$. E como $f(V, C)$ é o elemento de maior C em $L1$, não existe outro mais provável que ele.

Com essa teoria podemos afirmar que, ao transformar as regras do conhecimento em programas equivalentes ao Teorema 8, obtemos um sistema que faz diagnose, com os mesmos resultados que um Sistema Especialista faz diagnose com o auxílio de um "shell".

· APENDICE ·

O PROGRAMA COMPILADOR DE REGRAS

```
:- op(700,xfy,e).
:- op(650,xfx,com).
:- op(800,xfx,se).
:- op(670,fx,nao).
:- op(820,xfx,:).

trad1(E,pergunte(nao Pergunta),not(PredProlog), Expl,CFs,CFs) :-
    perg(Pergunta,_,sn),!,
    PredProlog =.. [Pergunta,[EIExpl]].

trad1(E,pergunte(Pergunta),PredProlog, Expl,CFs,CFs) :-
    perg(Pergunta,_,sn),!,
    PredProlog =.. [Pergunta,[EIExpl]].

trad1(E,pergunte(nao Pergunta),not(PredProlog),Expl,CFs,CFs) :-
    Pergunta =.. [Predicado,Valor],
    perg(Predicado,_,escolha_certa),!,
    PredProlog =.. [Predicado,Valor,[EIExpl]].

trad1(E,pergunte(Pergunta),PredProlog,Expl,CFs,CFs) :-
    Pergunta =.. [Predicado,Valor],
    perg(Predicado,_,escolha_certa),!,
    PredProlog =.. [Predicado,Valor,[EIExpl]].

trad1(E,pergunte(nao Pergunta),(PredProlog,CFY=1-CFX),Expl,CFs,
                                           [CFY|CFs]):-
    Pergunta =.. [Predicado,Valor],
    perg(Predicado,_,escolha_incerta), !,
    PredProlog =.. [Predicado,CFX,Valor,[EIExpl]].

trad1(E,pergunte(Pergunta),PredProlog,Expl,Cfs,[CF|CFs]) :-
    Pergunta =.. [Predicado,Valor],
    perg(Predicado,_,escolha_incerta), !,
    PredProlog =.. [Predicado,CF,Valor,[EIExpl]].

trad1(E,(Pred), PredProlog, Expl, CFs,[CF|CFs]) :-
    /*para predicados definidos em Prolog*/
    Pred =.. [Fn,Valor],
    PredProlog =.. [Fn,CF,Valor,[EIExpl]].

trad1((Pred),Pred,_,CFs,CFs).
    /* para predicados pre-definidos do Prolog */
```

```

trad_perg(Perg) :-
    guarde(Perg),
    completa_perg(Perg).

completa_perg(perg(Pergunta,L,sn)) :- !,
    Pred1 =.. [Pergunta,Expl],
    Pred2 =.. [pergunta,Pergunta,_,Certeza,Expl],
    guarde((Pred1 :- Pred2,Certeza=0,! ,fail)),
    guarde(Pred1).

completa_perg(perg(Pergunta,L,escolha_certa)) :- !,
    Pred1 =.. [Pergunta,Valor,Expl],
    Pred2 =.. [pergunta,Pergunta,Valor1,_,Expl],
    guarde((Pred1 :- Pred2,Valor=Valor1,!)),
    guarde((Pred1 :- fail)).

completa_perg(perg(Pergunta,L,escolha_incerta)) :- !,
    Pred1 =.. [Pergunta,CFX,Valor,Expl],
    Pred2 =.. [pergunta,Pergunta,Valor1,CFX,Expl],
    guarde((Pred1 :- Pred2, Valor=Valor1, !)),
    guarde((Pred1 :- fail)).

tradcauda(E,Expl,CF,CR,CFs,P1 e P2, (Prlg1,Prlg2)) :- !,
    trad1(E,P1,Prlg1,Expl,CFs,NovoCFs),
    tradcauda(E,Expl,CF,CR,NovoCFs,P2,Prlg2).

tradcauda(E,Expl,CF,CR,CFs,P,Prlg):-
    trad1(E,P,Prlg1,Expl,CFs,NovoCFs),
    addmin(Prlg1,CF,CR,NovoCFs,Prlg).

addmin(Prlg,CF,CF,[],Prlg) :- !.
addmin(Prlg,CF,1,[CF],Prlg) :- !.
addmin(Prlg,CF,N,[CF1],(Prlg, CF is N*CF1)) :- !.

addmin(Prlg,CF,1,L,(Prlg; min(CF,L))) :- !.
addmin(Prlg,CF,N,L,(Prlg,min(CFM,L), CF is N*CFM)).

trad(E,Cab com CR se Cauda, (CabPrlg :- CaudaPrlg)) :-
    Cab =.. [Fn,Valor],
    CabPrlg =.. [Fn,possivel,c(CF,Valor), Expl],
    tradcauda(E,Expl,CF,CR,[],Cauda,CaudaPrlg).

cabecalho(Cabeca,
    (CabPrlg :- findall(V, Pred, L),
    max(Fn,c(CFX,Valor),L))) :-
    Cabeca =.. [Fn1_],
    CabPrlg =.. [Fn,CFX,Valor,Expl],
    Pred =.. [Fn, possivel, V, Expl].

elimine(Fn, Ar) :- abolish(Fn/Ar) , !.
elimine(Fn, Ar).

cabeca(C com Cz se Cauda, C) :- !.

```

```

guarde_regra(Fn, Ar, E:Regra, Regra1) :-
    cabeca(Regra, C),
    functor(C,Fn,Ar), !,
    trad(E,Regra, Regra_Prolog),
    guarde(Regra_Prolog),
    read(Prox_Regra),
    guarde_regra(Fn,Ar,Prox_Regra, Regra1).

guarde_regra(Fn, Ar, Regra,Regra).

armazene_regras(end_of_file) :- !.

armazene_regras(perc(Perg,Dpcoes,Tipo)) :-
    !, trad_perc(perc(Perg,Dpcoes,Tipo)),
    read(Regra),
    armazeneregras(Regra).

armazene_regras(explicacao(E,Expl)) :-
    !, guarde(explicacao(E,Expl)),
    read(Regra),
    armazeneregras(Regra).

armazene_regras(E:Regra) :-
    cabeca(Regra,C),
    functor(C,Fn,Ar),
    elimine(Fn,3),
    cabecalho(C,Cabecalho),
    guarde(Cabecalho),
    guarde_regra(Fn,Ar,E:Regra,Regra1),
    armazeneregras(Regra1).

reconsulte(Arq) :- see(Arq),
    read(Regra),
    armazeneregras(Regra),
    seen.

guarde(R) :- assertz(R).

```

A seguir listamos os programas chamados durante a execução do Sistema Especialista compilado, que independem do do domínio da Base de Conhecimento:

```

maior(c(X,_),c(Y,_)) :- X>Y.
maior_igual(c(X,_),c(Y,_)) :- X >= Y.
minimo(M,X) :- min(M,1,X).

```



```

min(M,M,[]).

min(M,M1,[XIY]) :-
    M1 = ( X , ! ,
    min(M,M1,Y).

min(M,_,[XIY]) :-
    min(M,X,Y).

max(Mensagem,X,L) :-
    merge(L,[YILM]),
    maximo(LM,Y,[],X,_).

maximo([],M,Rs,M,Rs).

maximo([XIL],C,R,M,Rs) :-
    maior(X,C),
    maximo(L,X,[CIR],M,Rs).

maximo([XIL],C,R,M,Rs) :-
    maior_igual(C,X),
    maximo(L,C,[XIR],M,Rs).

maximo([],_,[CIR],M,Rs) :-
    maximo(R,C,[],M,Rs).

merge([],[]) :- !.

merge([XIL],[YIS]) :-
    merge1(X,L,Y,LM),
    merge(LM,S).

merge1(CertVal,CertVals,MergCertVal,RestoCertVals) :-
    merge1_aux(CertVals,CertVal,[],MergCertVal,RestoCertVals).

merge1_aux([],XM,LM,XM,LM) :- !.

merge1_aux(L,XP,LP,XM,LM) :-
    comb(L,XP,LP,ProxL,ProxXP,ProxLP),
    merge1_aux(ProxL,ProxXP,ProxLP,XM,LM).

comb([c(CFX,V)IL],c(CFY,V),LP,L,c(CF,V),LP) :-
    !, CF is CFX + CFY - CFX * CFY.

comb([CIL],CP,LP,L,CP,[CILP]).

inic_dados :-
    retract(resp(,_,_)), fail.

inic_dados.

```

```

index(Perg,[X],1,X,[E|Expl]) :-
    X = porque, !,
    explicacao(E,Texto),
    write(Texto),nl,
    pergunta(Perg,_,_,Expl),
    fail.

index(Perg,[X],1,X,[]) :-
    X = porque, !,
    write('Nenhuma outra explicacao'), nl,
    pergunta(Perg,_,_,[]),
    fail.

index(_,[_],1,X,_) :- !.

index(Perg,[_],I,Z,E) :-
    J is I - 1,
    index(Perg,Y,J,Z,E).

pergunta(Perg,Resp,Certeza,_) :-
    resp(Perg,Resp,Certeza), !.

pergunta(Perg,Resp,Certeza,E) :-
    perg(Perg,Opcoes,escolha_incerta),
    write(Perg), nl,
    write(Opcoes), nl,
    read(Resp),
    teste(Resp,Perg,E), !,
    write('Certeza'), nl,
    write([certeza_absoluta,
           quase_certo,
           talvez,
           nao_sei]), nl,
    read(CF),
    conv_cert(CF,Certeza),
    assertz(resp(Perg,Resp,Certeza)).

pergunta(Perg,Resp,1,E) :-
    perg(Perg,Opcoes,escolha_certa),
    write(Perg),nl,
    write(Opcoes),nl,
    read(Resp),
    teste(Resp,Perg,E), !,
    assertz(resp(Perg,Resp,1)).

pergunta(Perg,Resp,Certeza,E) :-
    perg(Perg,Opcoes,sn),
    write(Perg),
    write(Opcoes),nl,
    read(Resp),
    teste(Resp,Perg,E), !,
    conv_cert(Resp,Certeza),
    assertz(resp(Perg,Resp,Certeza)).

```

```

pergunta(Perg,Resp,Certeza,Expl) :-
    /* Esta clausula eh para ser usada, quando
       foram dadas explicacoes (a chamada teste falha) */
    resp(Perg,Resp,Certeza).

conv_cert(sim,1).
conv_cert(nao,0).
conv_cert(certeza_absoluta,1).
conv_cert(quase_certo,0.8).
conv_cert(talvez,0.7).
conv_cert(nao_sei,0.5).

teste(Resp,Perg,[]) :-
    Resp = porque, !,
    write('Nenhuma outra explicacao'),nl,
    pergunta(Perg,_,_,[]),
    fail.

teste(Resp,Perg,[E|Expl]) :-
    Resp = porque, !,
    explicacao(E,Texto),
    write(Texto), nl,
    pergunta(Perg,_,_,Expl),
    fail.

teste(Resp,Perg,Expl).

```

Uma Base de Conhecimento sobre Vinhos

```
perg(tem_vitela,[sim,nao,porque],sn).
perg(tem_peru,[sim,nao,porque],sn).
perg(tem_molho,[sim,nao,porque],sn).
perg(prato_principal,[carne,peixe,ave,porque],escolha_incerta).
perg(molho,[muito_temperado,doce,cremoso,tomate,porque],
      escolha_incerta).
perg(sabor,[delicado,medio,forte,porque],escolha_incerta).
perg(docura_preferida,[doce,medio,seco,porque],escolha_certa).
perg(cor_preferida,[tinto,branco,porque],escolha_certa).
perg(corpo_preferido,[leve,medio,pesado,porque],escolha_certa).
gol([vinho]).

e1:vinho('Gamay') com 1 se
    cor_recomendada(tinto) e
    corpo_recomendado(medio) e
    docura_recomendada(doce).

e1:vinho('Chablis') com 1 se
    cor_recomendada(branco) e
    corpo_recomendado(leve) e
    docura_recomendada(seco).

e1:vinho('Savignon Blanc') com 1 se
    cor_recomendada(branco) e
    corpo_recomendado(medio) e
    docura_recomendada(seco).

e1:vinho('Valpolicella') com 1 se
    cor_recomendada(tinto) e
    corpo_recomendado(leve).

e1:vinho('Pinot Noir') com 1 se
    cor_recomendada(tinto) e
    corpo_recomendado(medio) e
    docura_recomendada(medio).

e1:vinho('Burgundy') com 1 se
    corpo_recomendado(pesado).
```

e1:vinho('Soave') com 1 se
 cor_recomendada(branco) e
 corpo_recomendado(leve) e
 docura_recomendada(seco).

e1:vinho('Soave') com 1 se
 cor_recomendada(branco) e
 corpo_recomendado(leve) e
 docura_recomendada(medio).

e1:vinho('Riesling') com 1 se
 cor_recomendada(branco) e
 corpo_recomendado(medio) e
 docura_recomendada(doce).

e1:vinho('Chardonnay') com 1 se
 cor_recomendada(branco) e
 corpo_recomendado(medio) e
 docura_recomendada(medio).

e1:vinho('Chardonay') com 1 se
 cor_recomendada(branco) e
 corpo_recomendado(medio) e
 docura_recomendada(seco).

e1:vinho('Chenin Blanc') com 1 se
 cor_recomendada(branco) e
 corpo_recomendado(leve) e
 docura_recomendada(doce).

e1:vinho('Geverztraminer') com 1 se
 cor_recomendada(branco) e
 corpo_recomendado(pesado).

e1:vinho('Cabernet Sauvignon') com 1 se
 cor_recomendada(tinto) e
 docura_recomendada(seco).

e1:vinho('Zinfandel') com 1 se
 cor_recomendada(tinto) e
 docura_recomendada(medio).

e1:vinho('Zinfandel') com 1 se
 cor_recomendada(tinto) e
 docura_recomendada(leve).

e2:cor_recomendada(tinto) com 0.8 se
 melhor_cor(tinto).

e2:cor_recomendada(branco) com 0.8 se
 melhor_cor(branco).

e3:cor_recomendada(tinto) com 1 se
 pergunte(cor_preferida(tinto)) e
 melhor_cpr(tinto).

e3:cor_recomendada(branco) com 1 se
pergunte(cor_preferida(branco)) e
melhor_cor(branco).

e4:melhor_cor(tinto) com 0.9 se
pergunte(prato_principal(carne)) e
pergunte(nao_tem_vitela).

e4:melhor_cor(branco) com 0.6 se
pergunte(prato_principal(carne)) e
pergunte(tem_vitela).

e5:melhor_cor(branco) com 1 se
pergunte(prato_principal(peixe)).

e6:melhor_cor(tinto) com 0.8 se
pergunte(prato_principal(ave)) e
pergunte(tem_peru).

e6:melhor_cor(branco) com 0.5 se
pergunte(prato_principal(ave)) e
pergunte(tem_peru).

e6:melhor_cor(branco) com 0.9 se
pergunte(prato_principal(ave)) e
pergunte(nao_tem_peru).

e6:melhor_cor(tinto) com 0.3 se
pergunte(prato_principal(ave)) e
pergunte(nao_tem_peru).

e7:melhor_cor(tinto) com 1 se
pergunte(nao_prato_principal(peixe)) e
pergunte(tem_molho) e
pergunte(molho(tomate)).

e8:corpo_recomendado(leve) com 0.7 se
melhor_corpo(leve).

e8:corpo_recomendado(medio) com 0.7 se
melhor_corpo(medio).

e8:corpo_recomendado(pesado) com 0.7 se
melhor_corpo(pesado).

e9:corpo_recomendado(leve) com 1 se
melhor_corpo(leve) e
pergunte(corpo_preferido(leve)).

e9:corpo_recomendado(medio) com 1 se
melhor_corpo(medio) e
pergunte(corpo_preferido(medio)).

e9:corpo_recomendado(pesado) com 1 se
melhor_corpo(pesado) e
pergunte(corpo_preferido(pesado)).

e9:corpo_recomendado(medio) com 0.8 se
melhor_corpo(leve) e
pergunte(corpo_preferido(pesado)).

e9:corpo_recomendado(medio) com 0.8 se
melhor_corpo(pesado) e
pergunte(corpo_preferido(leve)).

e10:melhor_corpo(leve) com 0.8 se
pergunte(sabor(delicado)).

e10:melhor_corpo(leve) com 0.3 se
pergunte(sabor(medio)).

e10:melhor_corpo(medio) com 0.6 se
pergunte(sabor(medio)).

e10:melhor_corpo(pesado) com 0.3 se
pergunte(sabor(medio)).

e10:melhor_corpo(medio) com 0.4 se
pergunte(sabor(forte)).

e10:melhor_corpo(pesado) com 0.8 se
pergunte(sabor(forte)).

e11:melhor_corpo(pesado) com 1 se
pergunte(tem_molho) e
pergunte(molho(muito_temperado)).

e11:melhor_corpo(medio) com 0.4 se
pergunte(tem_molho) e
pergunte(molho(cremoso)).

e11:melhor_corpo(pesado) com 0.6 se
pergunte(tem_molho) e
pergunte(molho(cremoso)).

e12:docura_recomendada(seco) com 0.7 se
melhor_docura(seco).

e12:docura_recomendada(medio) com 0.7 se
melhor_docura(medio).

e12:docura_recomendada(doce) com 0.7 se
melhor_docura(doce).

e13:docura_recomendada(seco) com 1 se
melhor_docura(seco) e
pergunte(docura_preferida(seco)).

e13:docura_recomendada(medio) com 1 se
melhor_docura(medio) e
pergunte(docura_preferida(medio)).

e13:docura_recomendada(doce) com 1 se
melhor_docura(doce) e
pergunte(docura_preferida(doce)).

e13:docura_recomendada(medio) com 0.8 se
melhor_docura(seco) e
pergunte(docura_preferida(doce)).

e13:docura_recomendada(medio) com 0.8 se
melhor_docura(doce) e
pergunte(docura_preferida(seco)).

e13:docura_recomendada(seco) com 1 se
pergunte(docura_preferida(seco)).

e13:docura_recomendada(medio) com 1 se
pergunte(docura_preferida(medio)).

e13:docura_recomendada(doce) com 1 se
pergunte(docura_preferida(doce)).

e14:melhor_docura(doce) com 0.9 se
pergunte(tem_molho) e
pergunte(molho(doce)).

e14:melhor_docura(medio) com 0.4 se
pergunte(tem_molho) e
pergunte(molho(doce)).

explicacao(e1,'Para encontrar um vinho provavel, preciso
determinar a cor recomendada, o corpo re-
comendado e a docura recomendada').

explicacao(e2,'Para determinar a cor recomendada, preciso
determinar a melhor cor para a sua refeicao')

explicacao(e3,'Para determinar a cor recomendada, preciso
determinar a melhor cor para a sua refeicao
e qual a cor de sua preferencia').

explicacao(e4,'Para determinar a melhor cor, preciso saber
se o prato principal eh carne e se tem vitela

explicacao(e5,'Para determinar a melhor cor, preciso saber se o prato principal eh peixe').

explicacao(e6,'Para determinar a melhor cor, preciso saber se o prato principal eh ave e se tem peru').

explicacao(e7,'Para determinar a melhor cor, preciso saber se o prato principal nao eh peixe, se o molho eh de tomate').

explicacao(e8,'Para determinar o melhor corpo, preciso determinar o melhor corpo para sua refeicao').

explicacao(e9,'Para determinar o corpo recomendado, preciso determinar o melhor corpo para sua refeicao e qual o corpo de sua preferencia').

explicacao(e10,'Para determinar o melhor corpo, preciso saber como eh o sabor da sua refeicao').

explicacao(e11,'Para determinar o melhor corpo, preciso saber se tem molho e como ele eh').

explicacao(e12,'Para determinar a docura recomendada, preciso determinar a melhor docura para a sua refeicao')

explicacao(e13,'Para determinar a docura recomendada, preciso determinar a melhor docura para a sua refeicao e qual a docura de sua preferencia').

explicacao(e14,'Para determinar a melhor docura, preciso saber se tem molho e como ele eh').

Como Usar o Compilador de Regras

Tomemos como exemplo a compilação da base de conhecimento sobre vinhos, cujo nome do arquivo que a contém é 'vinhos.db'.

Inicialmente, devemos carregar o arquivo que contém o compilador. Em seguida, ativamos a compilação da seguinte forma:

```
reconsulte('vinhos.db').
```

Para completar o SE devemos carregar o arquivo que contém as chamadas do SE que são independentes da compilação e assim, podemos fazer a consulta:

```
vinho.
```

REFERENCIAS BIBLIOGRAFICAS

[Buchanan-78] |

Buchanan, B.G.; Feigenbaum, E.A. Dendral and Meta-Dendral: Their Applications Dimension. Artificial Intelligence, Vol 11, pp 5-24, 1978

[Buchanan-84] |

Buchanan, B.G.; Shortliffe, E.H. Editores. Rule-Based Expert Systems. The Addison-Wesley Series in Artificial Intelligence, 1984

[Clark-84] |

Clark, K.L.; McCabe, F.G. micro-PROLOG: Programming in Logic. Prentice-Hall, 1984

[Duda-79] |

Duda, R.D.; Hart, P.E.; Konolige, K.; Rebo, R.A. Computer-based Consultant for Mineral Exploration. Technical Report SRI International, Sept. 1979

[Ernst-69] |

Ernst, G.; Newell, A. GPS: A Case Study in Generality and Problem Solving. ACM Monograph. Academic Press, 1969

[Hammond-82] |

Hammond, P. Logic Programming for Expert Systems. Technical Report DDC 82/4. Department of Computing, Imperial College of Science and Technology, University of London, March 1982

[Hayes-Roth-83] |

Hayes-Roth, F.; Waterman, D.A.; Lenat, D.B. An Overview of Expert Systems. Building Expert Systems, pp 3-29, Addison-Wesley 1983

[Kowalski-79] |

Kowalski, R.A. Logic for Problem Solving. North Holland Elsevier, 1979

[Barros-86] |

Barros, N.L. Mini-dissertação para Exame de Qualificação apresentada ao Instituto de Ciências Matemáticas de São Carlos - USP.

[Lindsay-80] |

Lindsay, R.K.; Buchanan, B.G.; Feigenbaum, E.A.; Lederberg, J. Applications of Artificial Intelligence for Organic Chemistry: The

Dendral Project. McGraw-Hill, 1980

[McDermott-80] :

McDermott, J. R1: A Rule-Based Configurator of Computer Systems. Computer Science Department, Carnegie-Mellon University, 1980

[Merrit-86] :

Merrit, Dennis. Forward Chaining in PROLOG. AI EXPERT the Magazine for the Artificial Intelligence Community. November, 1986.

[Michaelson-85] :

Michaelson, R.H.; Michie, D.; Boulanger, A. The Technology of Expert Systems. Byte, pp 303-312, April 1985

[Nilsson-82] :

Nilsson, N.J. Principles of Artificial Intelligence. Springer-Verlag, 1982

[Rich-83] :

Rich, E. Artificial Intelligence. McGraw Hill, 1983

[Robinson-65] :

Robinson, J.A. A Machine Oriented Logic Based on the Resolution Principle. Journal of the Association for Computing Machinery. 12, 23-41.

[Sell-85] :

Sell, P.S. Expert Systems: A Practical Introduction. MacMillan Computer Science Series, 1985

[Shortliffe-76] :

Shortliffe, E.H. Computed-Based Medical Consultation: MYCIN, American Elsevier, New York, 1976

[Tärnlund-77] :

Tärnlund, S.A. Horn Clause Computability. Bit 17, pp 215-226.

[Thompson-85] :

Thompson, B.A.; Thompson, W.A. Inside an Expert System. Byte, pp 315-330, April 1985

[Winston-84] :

Winston, P.H. Artificial Intelligence. Addison-Wesley, 1984