**IMPLEMENTAÇÃO LÓGICA DO
NÚCLEO DE UM SISTEMA ESPECIALISTA**

Sandra Mari Hebihara

Orientadora:

Profa.Dra. Maria Carolina Monard

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

**- São Carlos -
Abril de 1988**

A meus pais

AGRADECIMENTOS

Agradeço à *Prof. Dra. Maria Carolina Monard* pela orientação, incentivo e dedicação recebidos durante o desenvolvimento deste trabalho.

Agradeço à *FAPESP* e *CNPq* pelo auxílio financeiro.

Agradeço aos amigos que tiveram participação constante na realização deste trabalho; em particular, à *Marta Bueno de Moraes* e com especial carinho a *Celso Nagata*.

ÍNDICE

INTRODUÇÃO.....	01
-----------------	----

CAPÍTULO 1

SISTEMAS ESPECIALISTAS

1.1	DEFINIÇÃO E CARACTERÍSTICAS.....	03
1.2	ESTRUTURA BÁSICA.....	04
1.2.1	A Base de Conhecimento.....	05
1.2.2	O Engenho de Inferência.....	06
1.2.3	A Base de Dados.....	08
1.2.4	O Módulo de Coleta de Evidências.....	08
1.2.5	Interface de Linguagem Natural.....	08
1.3	LINGUAGENS DE PROGRAMAÇÃO.....	10

CAPÍTULO 2

LÓGICA NÃO-MONOTONICA

2.1	LÓGICA DE PRIMEIRA ORDEM E MONOTONICIDADE.....	13
2.2	LÓGICA NÃO-MONOTONICA.....	15
2.3	RACIOCÍNIO DEFAULT.....	19

CAPITULO 3

PROVA AUTOMATICA DE TEOREMAS

3.1	PROVADOR AUTOMÁTICO DE TEOREMAS EM PROLOG.....	22
3.1.1	Introdução.....	22
3.1.2	Implementação.....	27
3.2	EXEMPLOS.....	29
3.2.1	Exemplo 1.....	29
3.2.2	Exemplo 2.....	32
3.2.3	Exemplo 3.....	34
3.2.4	Exemplo 4.....	36
3.2.5	Exemplo 5.....	38

CAPITULO 4

DESCRICAO DO SISTEMA

4.1	INTRODUÇÃO.....	44
4.2	DIAGNOSE E APRENDIZAGEM.....	47
4.3	REPRESENTAÇÃO DE CONHECIMENTO.....	48
4.4	LINGUAGEM DE INTERAÇÃO.....	53
4.5	"ESQUELETO" DE UM SISTEMA ESPECIALISTA.....	55

CAPITULO 5

IMPLEMENTAÇÃO DO SISTEMA

5.1	INTRODUÇÃO.....	56
5.2	O MÓDULO DE AQUISIÇÃO DE CONHECIMENTO.....	56
5.3	O MÓDULO DE COLETA DE EVIDÊNCIAS.....	65
5.4	O NÚCLEO DO SISTEMA ESPECIALISTA.....	68
5.5	EXEMPLO.....	75
CONCLUSÃO.....		82
REFERÊNCIAS.....		84
APÊNDICE.....		86

RESUMO

Neste trabalho apresentamos a construção e implementação do Núcleo de um Sistema Especialista que usa raciocínio por "default" e representa o conhecimento usando "frames".

O raciocínio "default" está associado com conhecimento incompleto, isto é, possíveis hipóteses (defaults) podem ser usadas para raciocinar, sempre que elas forem consistentes com os demais conhecimentos do problema em questão.

A idéia geral utilizada é que os seres humanos constroem uma "teoria científica" para predizer observações e que esta teoria pode ser justificada.

Mostramos também uma forma simples de selecionar estas teorias a fim de obter aprendizagem ou diagnose.

O sistema tem sido implementado em uma extensão da linguagem de programação lógica PROLOG.

ABSTRACT

In this work we show how to build and implement an Expert System inference engine using default reasoning and frames as knowledge representation.

Default reasoning is associated with incomplete knowledge where possible hypothesis (default) can be used to reason, provided they are consistent with the knowledge in the Knowledge Base.

The general idea is that human beings try to build a scientific theory to predict observations and furthermore, this theory can be justified. We also show a simple approach to select from these theories such that we can obtain learning or diagnosis.

The implementation language is an extension of the logic programming language PROLOG.

INTRODUÇÃO

Grande parte dos problemas do nosso dia-a-dia requer senso comum. A maior dificuldade de se modelar esse tipo de problema em Prolog encontra-se na monotonicidade da Lógica de Primeira Ordem.

A idéia fundamental deste trabalho, baseada no sistema Theorist de Poole et.al. [Poole-86a,b], é construir teorias científicas baseadas em possíveis hipóteses para predizer observações ao invés de retirar deduções monotonicamente do nosso conhecimento.

A construção automática de teorias científicas requer uma linguagem em que seja possível derivar a negação de uma fórmula, para que as teorias inconsistentes sejam rejeitadas.

O que se torna necessário então, é estender o Prolog para um Provador Automático de Teoremas de Primeira Ordem. Neste trabalho, apresentamos uma extensão e implementação baseada no procedimento MESON de Loveland [Loveland-78].

Assim, este sistema pode ser utilizado na implementação do Núcleo de um Sistema Especialista onde é possível fazer inferências através do raciocínio default. Defaults são possíveis hipóteses que podem fornecer uma explicação consistente para as possíveis observações.

Apresentaremos no Capítulo 1 deste trabalho a conceitualização sobre Sistemas Especialistas e principais características, e também a descrição de algumas linguagens de programação usuais

em Inteligência Artificial.

Introduzimos no Capítulo 2 os conceitos de lógica não-monotônica e raciocínio por default e suas principais dificuldades.

No Capítulo 3 mostramos como estender Prolog para um Provador de Teoremas de Primeira Ordem, a teoria em que essa extensão foi baseada e alguns exemplos de aplicação.

No Capítulo 4 fazemos a descrição da estrutura do sistema proposto apresentando também a representação de conhecimento por nós utilizada.

No Capítulo 5 descrevemos detalhes de implementação do sistema, onde os programas são apresentados e explicitados.

Finalmente, apresentamos nossas conclusões e descrevemos algumas sugestões para trabalhos futuros.

CAPÍTULO 1

SISTEMAS ESPECIALISTAS

1.1 DEFINIÇÃO E CARACTERÍSTICAS

Em meados da década de 50, a Inteligência Artificial emergia como um novo campo de pesquisa. Passaram-se quase trinta anos e ainda hoje várias são as conceituações para o termo. Fundamentalmente, Inteligência Artificial é a parte da Ciência de Computação que se preocupa em elaborar sistemas de computação inteligentes, ou seja, sistemas que exibem características que normalmente são associadas com inteligência no comportamento humano.

Um dos segmentos bastante desenvolvido das pesquisas contemporâneas de Inteligência Artificial é constituído pelos Sistemas Especialistas.

Um Sistema Especialista - S.E. - é um programa "inteligente" que usa conhecimentos e procedimentos de inferência na solução de problemas que normalmente só podem ser resolvidos por especialistas humanos. Sendo assim, estes programas realizam muitas das funções secundárias de um especialista, tais como perguntar questões relevantes e explicar seu raciocínio.

A maioria dos problemas em um domínio de especialidade não tem uma solução baseada em algoritmos, devido a que se

originam num contexto muito complexo, sem descrições ou especificações exatas, isto é, muitas vezes a solução está baseada em resultados empíricos.

Portanto, o trabalho que um computador pode desenvolver a nível de especialista depende principalmente dos conhecimentos que tem incorporados e da manipulação desses conhecimentos. É importante o papel do Engenheiro de Conhecimento que se responsabiliza pela transferência dos conhecimentos do especialista humano para a Base de Conhecimento. São várias as tarefas que ele deve realizar: ele deve extrair o conhecimento do especialista humano, organizá-lo e modelá-lo.

1.2 ESTRUTURA BÁSICA

Um Sistema Especialista é composto fundamentalmente pelos seguintes sub-sistemas básicos:

- Uma Base de Conhecimento que contém o conhecimento e heurísticas utilizadas pelo especialista.
- Um Motor de Inferências ou Núcleo do Sistema Especialista que interpreta os dados da Base de Conhecimento.
- Uma Base de Dados que contém os dados relativos ao domínio que está sendo explorado.
- Um Módulo de Coleta de Evidências que mantém a comunicação entre o usuário e o Sistema Especialista.

1.2.1 A Base de Conhecimento

A Base de Conhecimento - B.C. - consta dos fatos relativos a área de especialização, de regras que descrevem relações no domínio, e possivelmente de métodos, heurísticas e idéias que resolvem problemas nesse domínio.

Muitas regras estão calcadas na experiência ou intuição dos especialistas humanos, ou seja, as regras são empíricas e aproximadas. Nestes casos é comum atribuir-se um valor numérico que indica o grau de certeza à regra considerada. Por exemplo, o sistema MYCIN desenvolvido na Universidade de Stanford pelo médico E.H.Shortliffe [Shortliffe-76] é um sistema capaz de diagnosticar doenças infecciosas do sangue selecionando os antibióticos adequados para o respectivo tratamento. Este programa foi escrito em LISP e continha aproximadamente 600 regras do tipo:

```
SE      a infecção é bacteremia primaria E
        o local da cultura é um dos lugares esteréis E
        a porta de entrada suspeita é o trato gastrointestinal
```

```
ENTAO existe uma evidência sugestiva (0.7) de que a categoria
        do organismo é bacteróides.
```

MYCIN interage com o usuário, pedindo informação adicional sobre o paciente para aplicar outras regras, sugerindo, às vezes, a realização de análises clínicas. Baseado nas regras e nos dados do paciente, o sistema chega a uma conclusão e utiliza estes níveis de confiança visando simular raciocínio típico de

médicos. Este coeficiente de certeza é a base dos controles heurísticos que guia o processo de busca do programa até uma inferência plausível.

1.2.2 O Núcleo do Sistema Especialista

O Núcleo do Sistema Especialista processa as representações da Base de Conhecimento determinando uma linha de raciocínio para chegar à conclusão do problema que está sendo analisado. Esse processamento pode ser executado pelas seguintes estratégias:

. - Encadeamento para trás ("backward chaining") - pesquisa na Base de Conhecimento uma lista de hipóteses e procura reunir fatos que permitam concluir que uma das hipóteses é verdadeira. Portanto, o que devemos ter inicialmente é um conjunto de hipóteses. Alguns fatos deverão estar armazenados inicialmente no sistema, ou, como é frequente neste tipo de encadeamento, o sistema deverá interagir com o usuário durante uma consulta pedindo informações necessárias para provar as hipóteses. Por exemplo, o sistema MYCIN descrito anteriormente utiliza este tipo de raciocínio regressivo.

Essa estratégia seguiria uma pergunta do tipo: "Nós poderemos provar as hipóteses a partir dos dados que temos?".

- Encadeamento para frente ("forward chaining") - parte dos fatos e, a seguir, vai deduzindo outros fatos até chegar à solução do problema na qual se está trabalhando. Para que o

sistema possa trabalhar seguindo essa estratégia, deve ter disponível na Base de Conhecimento um conjunto de fatos e um conjunto de regras. Quando as condições das regras forem provadas pelo conjunto de fatos, a conclusão da regra será considerada verdadeira e adicionada ao conjunto de fatos. Por exemplo, considere o seguinte conjunto de fatos:

"o leão está faminto"

"o rato está perto do leão"

E o seguinte conjunto de regras:

"SE o leão está faminto E
o rato está perto do leão
ENTAO o leão comerá o rato"

"SE o gato está faminto E
o rato está perto do gato
ENTAO o gato comerá o rato"

Assim, inferimos o fato "o leão comera o rato" e o adicionaremos na Base de Conhecimento.

Resumindo, essa estratégia nos formularia uma pergunta do tipo: "O que nós podemos concluir a partir dos dados que temos?".

- A terceira possibilidade, frequentemente usada em problemas mais complexos, é uma estratégia em que tanto o "for-

ward" quanto o "backward chaining" são usados quando apropriados.

1.2.3 A Base de Dados

A Base de Dados serve para seguir o problema através de todas as suas fases. Contém os dados de entrada do problema e leva a sequência histórica de tudo que se tem feito, com o objetivo de poder justificar, se solicitado, os passos da linha de raciocínio.

1.2.4 Módulo de Coleta de Evidências

O Módulo de Coleta de Evidências é responsável pela comunicação entre o Sistema Especialista e o usuário. Este módulo fará as perguntas necessárias ao usuário, cujas respostas serão usadas pelo Núcleo do Sistema Especialista para inferir algumas deduções.

1.2.5 Interface de Linguagem Natural

É desejável que exista no sistema uma interface de comunicação de entrada/saída em linguagem natural para permitir maior facilidade de interação entre homem e máquina.

Entretanto, nos dias de hoje, ainda são raros os S.E. que possuem tal interface, pois a Linguagem Natural é um dos grandes desafios em estudo pelas pesquisas de I.A.

Dessa forma, o sistema pode ser representado pela figura seguinte:

ESTRUTURA DE UM SISTEMA ESPECIALISTA

1.3 LINGUAGENS DE PROGRAMAÇÃO

Embora seja possível escrever qualquer tipo de programa em qualquer linguagem de programação, o processo de construção de sistemas de I.A. pode ser consideravelmente facilitado se for usada uma linguagem que ofereça suporte para uma variedade de estruturas, tanto para dados quanto para controle.

Grande parte dos problemas de Inteligência Artificial foram implementados principalmente nas linguagens de programação LISP e PROLOG.

LISP (List Processing) foi desenvolvido no final da década de 50 por John McCarthy [McCarthy-60], baseado no Cálculo Lambda de Church. Os programas mais importantes de Inteligência Artificial até o momento foram escritos em LISP. Porém, nos últimos anos, PROLOG está tomando grande impulso em relação a LISP. Os trabalhos de Inteligência Artificial que estão sendo desenvolvidos na Europa tendem a ser implementados em PROLOG, assim como o projeto japonês de quinta geração.

PROLOG (Logic Programming) [Bratko-86], [Clocksin-81], [Colmerauer-85] surgiu em 1970 da idéia de usar lógica como linguagem de programação. É uma linguagem baseada no Cálculo de Predicados de Primeira Ordem e foi desenvolvido principalmente por Alain Colmerauer (implementação), da Faculdade de Ciências de Marseilles na França, Robert Kowalski (parte teórica) do Imperial College na Inglaterra e Maarten Van Emden (demonstração experimental), da Universidade de Waterloo, Canadá. Os programas Prolog consistem de fatos (por exemplo: pai_de(joao,maria)),

regras (por exemplo: $\text{ama}(X,Y) \text{ :- pai_de}(X,Y)$) e perguntas (por exemplo: $\text{ama}(X,\text{maria})$).

Os fatos e regras são usados para responder perguntas provando relações entre objetos (no exemplo anterior, os fatos e as regras são usados para fornecer a resposta de que quem ama maria é $X=\text{joao}$).

O mecanismo usado para realizar essas provas está fundamentado no princípio de resolução [Robinson-65] que é um método de demonstração automática de teoremas.

As classes de fórmulas lógicas que constituem os programas PROLOG são chamadas cláusulas de Horn, ou seja, são expressões da forma:

$$P \leftarrow P_1 \ \& \ P_2 \ \& \ \dots \ \& \ P_n$$

ou, equivalentemente, uma disjunção com no máximo um literal positivo:

$$P \vee \neg P_1 \vee \neg P_2 \vee \dots \vee \neg P_n$$

Esse tipo de cláusula é apropriada para representar um determinado domínio que não requer conhecimento incompleto ou negativo. Dessa forma, PROLOG é uma implementação de procedimento de prova para uma lógica simples, onde o conhecimento sobre o mundo é especificado em termos, as variáveis são quantificadas universalmente e as respostas são consequências lógicas do conhecimento.

No nosso trabalho utilizaremos a linguagem de programação Prolog, ou especificamente, o compilador Arity-Prolog [Arity-86].

CAPÍTULO 2

LÓGICA NÃO-MONOTONICA

2.1 LÓGICA DE PRIMEIRA ORDEM E MONOTONICIDADE

Os conhecimentos que usamos no nosso dia-a-dia envolvem grande quantidade de senso comum. Observe que todos nós sabemos como calçar os sapatos, sabemos como atravessar a rua, sabemos como tomar um banho; enfim, sabemos milhares de coisas de senso comum que frequentemente estamos realizando.

Porém, o raciocínio de senso comum é uma das coisas mais difíceis de se modelar no computador, uma vez que está diretamente ligado a aspectos subjetivos como vivência e senso prático. Por exemplo, a afirmação "o leão comeu o rato" é satisfatória para o homem e para a máquina. Mas a afirmação "o rato comeu o leão", embora aceitável para a máquina, é improvável ser aceita pelo homem sem questionamento. É necessário então repassar à máquina certas regras que não aceitem esse tipo de informação.

A primeira dificuldade com que nos confrontamos para formalizar problemas de senso comum é a propriedade de monotonicidade da Lógica de Primeira Ordem no sentido de que o que é válido num conjunto de fórmulas de primeira ordem continua válido quando novas informações são adicionadas a ele. Ou seja:

- Se A e B são conjuntos de fórmulas de primeira ordem

e

- $A \vdash q$ (A prova q)

então

- Se A está contido em B

então

- $B \vdash q$ (B prova q)

Realmente, uma prova a partir das premissas A é uma sequência finita de sentenças sendo que cada uma é ou uma premissa, ou um axioma ou segue de um subconjunto das sentenças que ocorreram antes na prova por uma das regras de inferência. Portanto, uma prova a partir de A pode também servir como uma prova a partir de B; nós dizemos que q é dedutível a partir de A ($A \vdash q$) se q é verdade em todos os modelos de A. Mas se $A \vdash q$ e A está contido em B, então qualquer modelo de B é também um modelo de A o que mostra que $B \vdash q$.

É bastante confortável trabalhar em sistemas monotônicos, pois eles apresentam muitas vantagens de confiabilidade. Por exemplo:

- Quando um novo fato é adicionado ao sistema preservando a consistência, novas verdades são obtidas, mas as verdades obtidas anteriormente serão preservadas.

- Não é preciso lembrar para cada declaração que foi provada a lista das outras declarações, pois não existe nenhum risco das declarações desaparecerem.

Os problemas do mundo real geralmente utilizam raciocínio não-monotônico.

Por exemplo, podemos aceitar tranquilamente que

SE é de manhã ENTÃO está claro

mas não aceitaremos se acrescentarmos a expressão anterior o seguinte fato:

SE é de manhã E ocorre uma eclipse solar ENTÃO está claro

Portanto, raciocínio de senso comum é não monotônico pois frequentemente extraímos novas conclusões quando sobre informações parciais são dadas informações mais completas.

2.2 LÓGICA NAO-MONOTONICA

Vejamos um exemplo interessante dado por Dov Gabbay [Gabbay-82]:

A WTA (World Travel Agency) oferece pacotes de turismo de Nova York a Paris e Stuttgart. Todos os domingos passageiros embarcam num avião WTA em Nova York; alguns passageiros desembarcam em Paris, enquanto outros continuam para Stuttgart. O escritório da agência de Stuttgart reserva hotéis na Europa. Porém, às 12:00 hs, o escritório de Stuttgart se informa de que todos os voos para Paris serão redirecionados, pois o aeroporto de Paris foi tomado por terroristas. O escritório de Stuttgart tinha que decidir o que fazer com o grupo WTA. Eles não podiam entrar em contato com a WTA de Nova York pois as linhas telefônicas estavam interrompidas. O único fato concreto é que o avião WTA havia

decolado em Nova York. O escritório de Stuttgart deduziu então que o avião seria redirecionado para Londres onde os passageiros de Paris desembarcariam, enquanto os outros prosseguiriam para Stuttgart. Gabbay representa esse raciocínio da seguinte maneira:

(a) Fatos concretos às 12:00 hs em Stuttgart:

A = avião levantou vôo em Nova York.

B = Terroristas atacaram o aeroporto de Paris.

(b) Verdade assumida às 12:00 hs baseado no procedimento da companhia e nenhuma evidência contrária:

C = Vôo foi redirecionado para Londres.

(c) Conclusão às 12:00 hs:

D = Os passageiros de Stuttgart chegam hoje a noite em Londres.

Este drama mostra que o inesperado acontece. Às 14:00 hs o escritório de Stuttgart consegue uma ligação telefônica com Nova York. O que eles ficam sabendo provoca preocupações: o avião atrasou duas horas e meia para decolar. Isso muda a figura consideravelmente. O escritório de Stuttgart agora deduz (pela comunicação com Nova York) que o comandante, quando ficou sabendo do ataque a Paris, decidiu retornar a curta distância para Nova York. Este raciocínio pode ser expressado como se segue:

(a') Fato às 14:00 hs:

A, B e

(A') = O vôo saiu atrasado em Nova York.

(b') Verdade assumida baseada no procedimento da companhia e nenhuma evidência contrária:

(C') = O avião retorna a Nova York.

(c') Conclusão às 14:00 hs:

~D = Os passageiros de Stuttgart não chegam hoje a noite em Londres.

Este exemplo nos leva à natureza da inferência não-monotônica.

Às 12:00 o escritório de Stuttgart fez certas suposições plausíveis, e essas suposições permitiram extrair certas inferências. Infelizmente, as suposições tornaram-se menos plausíveis dando-se informações adicionais. Subsequentemente, a conclusão D não estava mais garantida.

Sistemas lógicos "não-monotônicos" são lógicas em que a introdução de nova informação (axiomas) pode invalidar teoremas antigos [McDermott-80].

Assim, em nosso exemplo, antes da informação A, D era provável, mas depois que A foi adicionado, ~D tornou-se provável.

Em outras palavras, um sistema é dito monotônico se para cada par de teorias S1 e S2,

se

S1 está contido em S2

então

Th(S1) está contido em Th(S2)

onde

$$\text{Th}(S1) = \{A: S1 \vdash A\}$$

O sistema que não satisfizer essa propriedade é dito não-monotônico

Como dizem McDermott e Doyle [McDermott-80], o fenômeno de não-monotonicidade está em toda parte da I.A. Por exemplo:

- estudos de percepção visual e representação de conhecimento sintetizam mecanismos, de uma forma ou outra, cuja tarefa é compensar a tomada de decisões incorretas ainda que aparentemente apropriadas;

- senso comum ignora milhares de exceções especiais: observe que diariamente deparamo-nos com vários problemas e para resolvê-los frequentemente tomamos decisões baseadas em "regras" gerais que não são universalmente verdadeiras. Com pouco esforço, podemos nos lembrar de inúmeras situações em que qualquer fato ou informação nova contraindica decisões assumidas anteriormente. Porém, as pessoas não ficam paralizadas por causa disso; elas continuam a agir e decidir apesar dessa aparente inconsistência.

Na verdade, o que acontece geralmente é que o que sabemos sobre um mundo é "quase sempre" verdade com poucas exceções. Tais fatos assumem usualmente a forma "A maioria dos P's são Q's" ou "A maioria dos P's tem a propriedade Q". Por exemplo, a maioria dos passaros pode voar com exceção dos pinguins, avestruzes, etc; ou seja, se nós sabemos que Tico é um passaro, nós assumimos, na ausência de evidência contrária que ele pode voar. Se, entretanto, soubermos posteriormente que Tico é um pinguim, nós desconsideraremos a hipótese anterior.

Esse exemplo clássico nos mostra a inadequação da lógica tradicional para resolver certos tipos de problemas não-monotônicos, pois uma representação natural de primeira ordem seria:

$$\text{voa}(X) \text{ if } \text{passaro}(X) \wedge \sim \text{pinguim}(X) \wedge \sim \text{avestruz}(X) \wedge \dots$$

Mas, com essa representação, ninguém poderia concluir que um passaro "qualquer" voa. Ou seja, considere uma tentativa de provar que Tico voa, sendo que a única informação que temos sobre ele é que ele é um pássaro. Observe que nós deveríamos provar cada uma das seguintes submetas:

$$\sim \text{pinguim}(\text{Tico}) \wedge \sim \text{avestruz}(\text{Tico}) \wedge \dots$$

Porém, isso não é possível, pois não temos nenhuma dessas informações sobre Tico. Isso faz com que fiquemos bloqueados de concluir que Tico voa.

2.2 RACIOCÍNIO DEFAULT

Uma maneira de contornar a dificuldade do exemplo anterior é permitir exceções implícitas ao domínio de quantificação universal. Assim a declaração "todos os passaros voam" pode inicialmente ser dada sem listar qualquer exceção. A Lógica Default desenvolvida por Reiter [Reiter-80] propõe um mecanismo para afirmar conclusões na ausência de conhecimento específico.

No exemplo dado, diríamos que Tico voa "por default". A interpretação dada para esse default é algo como: "Se x é um pássaro, então na ausência de qualquer informação contrária, infira que x pode voar"; ou ainda: "Se x é um pássaro e é consistente assumir que x pode voar, então infira que x pode voar". A representação formal para regra default é a seguinte:

passaro(X): M voa(X)

voa(X)

onde M é lido como "é consistente assumir". As exceções podem ser declaradas na seguinte representação de primeira ordem:

~voa(X) if pinguim(X)

~voa(X) if avestruz(X)

As asserções inferidas por default tem o "status" de uma crença, pois podem ser modificadas ou rejeitadas por observações ou descobertas subsequentes. É essa propriedade que leva à não-monotonicidade da lógica de defaults (isto é, a adição de uma nova informação pode forçar a remoção de uma outra).

Por exemplo, suponhamos que Pedro queira visitar uma grande amiga. Pela regra geral, sabemos que a maioria das pessoas gosta de flores e então, para agradá-la, Pedro leva para ela um perfumado "bouquet". Se, ao receber as flores, ela começar a espirrar, Pedro deve abandonar sua crença prévia de que essa pessoa gosta de flores.

Defaults servem como mais uma regra de inferência somando-se àquelas fornecidas normalmente pela Lógica de Primeira

Ordem. Sobre certas condições, eles permitem inferências que não podem ser feitas dentro de uma estrutura de primeira ordem. Se suas premissas são conhecidas e suas justificativas são "consistentes" (i.e., suas negações não são prováveis), então seus consequentes podem ser inferidos. Apresentaremos mais adiante no Capítulo 5 a implementação de um sistema que utiliza raciocínio default.

CAPÍTULO 3

PROVA AUTOMÁTICA DE TEOREMAS

3.1 PROVADOR AUTOMÁTICO DE TEOREMAS EM PROLOG

3.1.1 Introdução

O raciocínio default pode ser considerado uma "teoria científica" que além de tirar dedução do nosso conhecimento, pode explicar os resultados.

Teorias científicas são interpretadas como sistemas de informação que organizam experiências passadas e predizem observações. Uma teoria pode ser deduzida a partir das experiências (asserções) ou de hipóteses (default).

De experiências ambíguas e hipóteses alternativas podem resultar diferentes teorias. Se uma alternativa tornar uma teoria inconsistente, faz-se necessário restringir ou modificar convenientemente as premissas que contribuíram para a contradição.

Neste trabalho, nos propomos a construir teorias científicas, deduzindo contradições e rejeitando teorias inconsistentes do nosso sistema.

Infelizmente, não podemos usar PROLOG diretamente na implementação do nosso sistema, pois a teoria das cláusulas de Horn são sempre consistentes, isto é, existe sempre uma interpretação verdadeira para qualquer conjunto de cláusulas de Horn.

Isso porque as teorias das cláusulas de Horn não podem derivar a negação de uma formula, exceto a negação por falha.

Como uma alternativa para tratar essa dificuldade, implementaremos uma extensão do PROLOG para um provador de teoremas da Lógica de Primeira Ordem.

Existem muitas maneiras de se implementar um provador automático de teoremas. Os principais métodos utilizados nos últimos anos são resolução e dedução natural. O "problema de redução de formato" desenvolvido por Loveland [Loveland-78] é similar aos sistemas baseados em dedução natural, tendo uma estrutura de meta-submeta. Ou seja, "o problema de redução de formato" é um dos métodos de prova por organização de busca que permite uma estrutura na qual o problema pode ser expresso, e um processo que pode levar a mostrar que a conclusão segue das premissas. Este método é importante para a implementação deste trabalho.

A idéia fundamental usada nessa implementação pode ser entendida considerando o exemplo seguinte:

Suponhamos as clausulas:

- 1 - Pedro está na praia.
- 2 - Se Pedro está na praia e não é um dia chuvoso
então Pedro nada.
- 3 - Se Pedro nada
então Pedro se molha.
- 4 - Se chove
então Pedro se molha.

É evidente para nós que "Pedro se molha" é verdade considerando que todas as cláusulas acima são verdadeiras.

Entretanto, o programa PROLOG correspondente a este problema não consegue realizar esta prova:

- 1 - na_praia(Pedro)
- 2 - nada(Pedro) if na_praia(Pedro) and
not dia(chuvoso)
- 3 - se_molha(Pedro) if nada(Pedro)
- 4 - se_molha(Pedro) if dia(chuvoso)

Pela árvore de prova temos:

- 1 - A
- 2 - B if A and $\sim C$
- 3 - D if B
- 4 - D if C

Segundo Loveland [Loveland-78], o que está faltando aqui é a negação para expressar as formas contrapositivas das afirmações dadas.

Definição: Dada uma cláusula

$A \text{ if } A_1 \text{ and } A_2 \dots \text{ and } A_n$

com A_i ($1 \leq i \leq n$) e A literais, então a sua contrapositiva

geral é a cláusula

$$B \text{ if } B_1 \text{ and } B_2 \dots \text{ and } B_n$$

onde B é $\sim A_1$ e cada B_i é A_j com $j \leq n$, ou $\sim A$ e cada $\sim A_i$ aparece exatamente uma vez na expressão.

Por exemplo:

$$A \text{ if } B \text{ and } C$$

tem como contrapositivas

$$\sim B \text{ if } \sim A \text{ and } C$$
$$\sim C \text{ if } \sim A \text{ and } B$$

Proposição: Se B é uma contrapositiva geral de uma asserção A , então B e A são logicamente equivalentes.

Demonstração: Se A é um literal, então B é falso se $\sim A$, e A e B são logicamente equivalentes pois

$$L \text{ s.s.s. } \sim L \text{ então falso}$$

é uma tautologia (onde L é literal).

Se a fórmula for

$$1 - A \text{ if } A_1 \text{ and } A_2 \text{ and } \dots \text{ and } A_n$$

e uma das contrapositivas é

$$2 - \sim A_n \text{ if } \sim A \text{ and } A_1 \text{ and } A_2 \text{ and } \dots \text{ and } A_{(n-1)}$$

então vê-se que (1) s.s.s. (2) é tautologia. Não há perda de generalidade, pois as outras contrapositivas são obtidas pela comutatividade quando A_1 e A são complementados e trocados de lugar.

Portanto, para o exemplo anterior, acrescentando as contrapositivas, teremos o seguinte conjunto de asserções:

1 - A

2 - B 1f A and $\sim$ C

3 - D 1f B

4 - D 1f C

5 - $\sim$ A 1f $\sim$ B and $\sim$ C

6 - C 1f $\sim$ B and A

7 - $\sim$ B 1f $\sim$ D

8 - $\sim$ C 1f $\sim$ D

A estratégia produzirá agora o novo ramo de prova:

...que pode ser marcado como sucesso em virtude da prova por redução ao absurdo.

Redução ao absurdo é a seguinte regra de inferência:

de A $\rightarrow$ B

e

A $\rightarrow$ $\sim$ B

inferimos

$\sim$ A

explique(come_animal(X,Y)).

Explicacao:

come_animal(passaro,lagarta)
multo_menor(lagarta,passaro)
multo_menor1(lagarta,passaro)
come_planta(lagarta)
come(lagarta,planta)

Voce quer outra solucao? sim.

Explicacao:

come_animal(raposa,lagarta)
multo_menor(lagarta,raposa)
multo_menor1(lagarta,passaro)
multo_menor(passaro,raposa)
multo_menor1(passaro,raposa)
come_planta(lagarta)
come(lagarta,planta)

Voce quer outra solucao? sim.

Explicacao:

come_animal(lobo,lagarta)
multo_menor(lagarta,lobo)
multo_menor1(lagarta,passaro)
multo_menor(passaro,lobo)
multo_menor1(passaro,raposa)
multo_menor(raposa,lobo)
multo_menor1(raposa,lobo)
come_planta(lagarta,planta)
come(lagarta,planta)

Voce quer outra solucao? sim.

Explicacao:

come_animal(raposa,minhoca)
multo_menor(minhoca,raposa)
multo_menor1(minhoca,passaro)
multo_menor(passaro,raposa)
multo_menor1(passaro,raposa)
come_planta(minhoca,planta)
come(minhoca,planta)

Voce quer outra solucao? sim.

Explicacao:

come_animal(lobo,minhoca)
multo_menor(minhoca,lobo)
multo_menor1(minhoca,passaro)

```
 muito_menor(passaro,lobo)
 muito_menor1(passaro,raposa)
 muito_menor(raposa,lobo)
 muito_menor1(raposa,lobo)
 come_planta(minhoca,planta)
 come(minhoca,planta)
```

Voce quer outra solucao? sim.

... Sem solucao!

O sistema encontrou 5 soluções para este problema. Para cada solução ele exhibe na lista de explicações o caminho percorrido; por exemplo, a primeira solução encontrada nos mostra que pássaros comem lagartas, porque elas são muito menores que os pássaros e também porque lagartas comem plantas. Dessa forma é possível perceber que as soluções satisfazem as condições do problema proposto.

3.2.2 Exemplo 2

Alguém que morava na Mansão da Família Oliveira matou a Sra. Isabela. As únicas pessoas que moravam na mansão eram Isabela, seu neto Carlos e o mordomo. Um assassino sempre odeia a sua vítima e nunca é mais rico que sua vítima. Carlos não odiava ninguém que Isabela odiava. Isabela odiava todos com exceção do mordomo. O mordomo odiava todos que eram menos ricos que Isabela e também odiava todos que ela odiava. Ninguém odeia todo mundo. Quem matou Isabela?

Formalizando o problema para o provador temos:

```
expl:mora_na_mansao(isabela).
expl:mora_na_mansao(mordomo).
expl:mora_na_mansao(carlos).
```

```

expl:matou(X,isabela) ->
    mora_na_mansao(X),
    assassino(X,isabela).

expl:assassino(X,Y) ->
    odela(X,Y),
    n_mais_rico(X,Y).

expl:odela(X,Y) ->
    not_prove(n_odela(X,Y)).

expl:n_odela(carlos,Y) ->
    odela(isabela,Y).

expl:odela(isabela,Y) ->
    diferente(Y,mordomo).

expl:odela(mordomo,Y) ->
    n_mais_rico(Y,isabela).

expl:odela(mordomo,Y) ->
    odela(isabela,Y).

expl:n_mais_rico(X,X).

if_needed:diferente(X,Y).

pro:diferente(X,Y) -> on(X,[isabela,mordomo,carlos]),
    on(Y,[isabela,mordomo,carlos]),
    X \= Y.

```

As cláusulas acima representam claramente a descrição do problema. O mistério será desvendado se fizermos a seguinte pergunta ao sistema:

```
explique(matou(X,isabela)).
```

```

Explicacao:
matou(isabela,isabela)
mora_na_mansao(isabela)
assassino(isabela,isabela)
odela(isabela,isabela)
n_mais_rico(isabela,isabela)

```

Voce quer outra solucao? sim.

...Sem solucao!

Concluimos portanto, que Isabela suicidou-se!

3.2.3 Exemplo 3

Vamos mostrar agora a prova do seguinte teorema de geometria:

- Um triângulo isóceles possui dois ângulos iguais que são opostos aos lados iguais do triângulo.

Construímos a Base de Conhecimento para a resolução do problema:

```
expl:seg_iguais(X,Y,X,X).
expl:seg_iguais(X,Y,U,V) -> seg_iguais(U,V,X,Y).
expl:seg_iguais(V,U,X,Y) -> seg_iguais(U,V,X,Y).
expl:triangulo(Y,Z,X) -> triangulo(X,Y,Z).
expl:triangulo(Y,X,Z) -> triangulo(X,Y,Z).

expl:ang_iguais(U,V,W,X,Y,Z) -> congruente(U,V,W,X,Y,Z).
expl:ang_iguais(U,V,W,Y,X,Z) -> congruente(U,V,W,X,Y,Z).
expl:ang_iguais(U,W,V,X,Z,Y) -> congruente(U,V,W,X,Y,Z).

expl:congruente(U,V,W,X,Y,Z) -> triangulo(U,V,W),
                                triangulo(X,Y,Z),
                                seg_iguais(U,V,X,Y),
                                seg_iguais(V,W,Y,Z),
                                seg_iguais(U,W,X,Z).
```

As asserções acima representam as seguintes propriedades:

seg_iguais(U,V,X,Y): segmento UV é igual ao segmento XY.

triangulo(X,Y,Z): triângulo formado pelos pontos X, Y e Z.

ang_iguais(U,V,W,X,Y,Z): ângulo UVW é igual ao ângulo XYZ.

congruente(U,V,W,X,Y,Z): o triângulo UVW é congruente com o triângulo XYZ.

Considerando os fatos:

expl:triangulo(a,b,c).

expl:seg_iguais(a,c,b,c).

Queremos provar:

ang_iguais(c,a,b,c,b,a).

O provador consegue demonstrar este teorema de diversas maneiras. Seguem abaixo algumas resoluções:

explique(ang_iguais(c,a,b,c,b,a)).

Explicacao:

ang_iguais(c,a,b,c,b,a)
congruente(c,a,b,c,b,a)
triangulo(c,a,b)
triangulo(b,c,a)
triangulo(a,b,c)
triangulo(c,b,a)
triangulo(a,c,b)
triangulo(b,a,c)
seg_iguais(c,a,c,b)
seg_iguais(c,b,c,a)
seg_iguais(b,c,c,a)
seg_iguais(c,a,b,c)
seg_iguais(a,c,b,c)
seg_iguais(a,b,b,a)
seg_iguais(a,c,c,b)
seg_iguais(c,b,a,c)
seg_iguais(b,c,a,c)

Voce quer outra solucao? sim.

Explicacao:

ang_iguais(c,a,b,c,b,a)
congruente(c,a,b,c,b,a)
triangulo(c,a,b)
triangulo(b,c,a)
triangulo(a,b,c)
triangulo(c,b,a)
triangulo(a,c,b)
triangulo(b,a,c)
seg_iguais(c,a,c,b)
seg_iguais(c,b,c,a)
seg_iguais(b,c,c,a)
seg_iguais(c,a,b,c)
seg_iguais(a,c,b,c)
seg_iguais(a,b,b,a)

Voce quer outra solucao?sim.

Explicacao:

ang_iguais(c,a,b,c,b,a)
congruente(c,a,b,c,b,a)
triangulo(c,a,b)
triangulo(b,c,a)
triangulo(a,b,c)
triangulo(c,b,a)
triangulo(a,c,b)
triangulo(b,a,c)
seg_iguais(c,a,c,b)
seg_iguais(c,b,c,a)
seg_iguais(b,c,c,a)
seg_iguais(c,a,b,c)
seg_iguais(a,c,b,c)
seg_iguais(a,b,b,a)
seg_iguais(b,a,a,b)
seg_iguais(a,c,c,b)
seg_iguais(c,b,a,c)
seg_iguais(b,c,a,c)

Voce quer outra solucao? sim.

Explicacao:

ang_iguais(c,a,b,c,b,a)
congruente(c,a,b,c,b,a)
triangulo(c,a,b)
triangulo(b,c,a)

triangulo(a,b,c)
triangulo(c,b,a)
triangulo(a,c,b)
triangulo(b,a,c)
seg_iguais(c,a,c,b)
seg_iguais(c,b,c,a)
seg_iguais(b,c,c,a)
seg_iguais(c,a,b,c)
seg_iguais(a,c,b,c)
seg_iguais(a,b,b,a)
seg_iguais(b,a,a,b)

Voce quer outra solucao?nao.

3.2.4 Exemplo 4

Dados os axiomas a seguir da Teoria dos Grupos queremos provar que a identidade à esquerda é também identidade à direita:

Axiomas:

1. $f(f(x,y),z) = f(x,f(y,z))$	associatividade
2. $f(e,x) = x$	identidade à esquerda
3. $f(g(x),x) = e$	inverso à esquerda
4. $x = x$	reflexividade
5. $x = y \rightarrow y = x$	simetria
6. $(x = y \ \& \ y = z) \rightarrow x = z$	transitividade
7. $(u = x \ \& \ v = y) \rightarrow f(u,v) = f(x,y)$	substituição por f
8. $x = y \rightarrow g(x) = g(y)$	substituição por g

Os termos acima são:

constante: e - identidade do grupo

funções: $f(x,y)$ - operação binária do grupo

$g(x)$ - operação inversa do grupo

O predicado "=" representa a relação de igualdade.

A representação dos axiomas é a seguinte:

```
expl:igual(f(f(X,Y),Z),f(X,f(Y,Z))).
expl:igual(f(e,X),X).
expl:igual(f(g(X),X),e).
expl:igual(X,X).
expl:igual(X,Y) -> igual(Y,X).
expl:igual(X,Z) -> igual(X,Y),igual(Y,Z).
expl:igual(f(U,V),f(X,Y)) -> igual(U,X),igual(V,Y).
expl:igual(g(X),g(Y)) -> igual(X,Y).
```

O predicado *igual* representa a relação de igualdade.

Este problema é relativamente fácil de ser resolvido. Porém, o nosso sistema não é capaz de resolvê-lo por causa do método de prova utilizado. Para fazer a prova perguntamos:

Para resolver este problema consideramos o estado inicial, um conjunto de movimentos possíveis dentro das condições impostas até que se chegue ao estado final desejado.

Um estado qualquer do problema é representado por:

[X,Y]

onde X representa o conteúdo (em litros) do recipiente maior e Y representa o conteúdo do recipiente menor.

Os movimentos permitidos pelo problema foram especificados pela relação *pode_despejar* onde:

pode_despejar([X,Y],[7,Y]).

para indicar que é possível encher o recipiente maior.

pode_despejar([X,Y],[X,5]).

para indicar que é possível encher o recipiente menor.

pode_despejar([X,Y],[0,Y]).

para indicar que é possível esvaziar o recipiente maior.

pode_despejar([X,Y],[X,0]).

para indicar que é possível esvaziar o recipiente menor.

*pode_despejar([X,Y],[0,Z]) ->
soma(X,Y,Z),
menor_igual(Z,5).*

para transferir a água do recipiente maior para o menor até que o recipiente maior fique vazio.

```
expl:pode_despejar([X,Y],[Z,0]) ->  
    soma(X,Y,Z),  
    menor_igual(Z,7).
```

para transferir a água do recipiente menor para o maior até que o recipiente menor fique vazio.

```
expl:pode_despejar([X,Y],[7,Z]) ->  
    soma(X,Y,X1),  
    menor(7,X1),  
    soma(7,Z,X1).
```

para transferir a água do recipiente menor para o maior até que o recipiente maior fique cheio.

```
expl:pode_despejar([X,Y],[Z,5]) ->  
    soma(X,Y,X1),  
    menor(5,X1),  
    soma(Z,5,X1).
```

para transferir a água do recipiente maior para o menor até que o recipiente menor fique cheio.

Se usarmos esta especificação para resolver este problema diretamente por PROLOG, o programa entra num "loop" que nunca termina, pois a estratégia de retrocesso ("backtracking") do PROLOG, não permite detectar estes "loops".

O nosso provador consegue resolver este problema verificando as metas antepassadas à meta que está sendo provado, descartando, desta forma, possíveis "loops".

A Base de Conhecimento fornecida para o provador é a seguinte:

```
expl:permitido([X,Y],[X1,Y1]) ->  
    pode_despejar([X,Y],[X1,Y1]).
```

```

expl: permitido([X,Y],[X1,Y1]) ->
    pode_despejar([X,Y],[X0,Y0]),
    permitido([X0,Y0],[X1,Y1]).

```

```

expl: pode_despejar([X,Y],[7,Y]).
expl: pode_despejar([X,Y],[X,5]).
expl: pode_despejar([X,Y],[0,Y]).
expl: pode_despejar([X,Y],[X,0]).

```

```

expl: pode_despejar([X,Y],[0,Z]) ->
    soma(X,Y,Z),
    menor_igual(Z,5).

```

```

expl: pode_despejar([X,Y],[Z,0]) ->
    soma(X,Y,Z),
    menor_igual(Z,7).

```

```

expl: pode_despejar([X,Y],[7,Z]) ->
    soma(X,Y,X1),
    menor(7,X1),
    soma(7,Z,X1).

```

```

expl: pode_despejar([X,Y],[Z,5]) ->
    soma(X,Y,X1),
    menor(5,X1),
    soma(Z,5,X1).

```

```

if_needed: menor(X,Y).
if_needed: menor_igual(X,Y).
if_needed: soma(X,Y,Z).

```

```

pro: menor(X,Y) -> X < Y.

```

```

pro: menor_igual(X,Y) -> X =< Y.

```

```

pro: soma(X,Y,Z) -> nonvar(X),
                    nonvar(Y),
                    var(Z),
                    Z is X+Y.

```

```

pro: soma(X,Y,Z) -> nonvar(X),
                    nonvar(Z),
                    var(Y),
                    Y is Z-X.

```

...e algumas das soluções possíveis são:

`explique(permitido([0,0],[4,Y])).`

Explicação:

```
permitido([0,0],[4,5])
pode_despejar([0,0],[7,0])
permitido([7,0],[4,5])
pode_despejar([7,0],[7,5])
permitido([7,5],[4,5])
pode_despejar([7,5],[0,5])
permitido([0,5],[4,5])
pode_despejar([0,5],[5,0])
permitido([5,0],[4,5])
pode_despejar([5,0],[5,5])
permitido([5,5],[4,5])
pode_despejar([5,5],[7,3])
permitido([7,3],[4,5])
pode_despejar([7,3],[0,3])
permitido([0,3],[4,5])
pode_despejar([0,3],[3,0])
permitido([3,0],[4,5])
pode_despejar([3,0],[3,5])
permitido([3,5],[4,5])
pode_despejar([3,5],[7,1])
permitido([7,1],[4,5])
pode_despejar([7,1],[0,1])
permitido([0,1],[4,5])
pode_despejar([0,1],[1,0])
permitido([1,0],[4,5])
pode_despejar([1,0],[1,5])
permitido([1,5],[4,5])
pode_despejar([1,5],[6,0])
permitido([6,0],[4,5])
pode_despejar([6,0],[6,5])
permitido([6,5],[4,5])
pode_despejar([6,5],[7,4])
permitido([7,4],[4,5])
pode_despejar([7,4],[0,4])
permitido([0,4],[4,5])
pode_despejar([0,4],[4,0])
permitido([4,0],[4,5])
pode_despejar([4,0],[4,5])
```

Voce quer outra solucao? sim.

Explicação:

```
permitido([0,0],[4,0])
pode_despejar([0,0],[7,0])
permitido([7,0],[4,0])
pode_despejar([7,0],[7,5])
```

```
permitido([7,5],[4,0])
pode_despejar([7,5],[0,5])
permitido([0,5],[4,0])
pode_despejar([0,5],[5,0])
permitido([5,0],[4,0])
pode_despejar([5,0],[5,5])
permitido([5,5],[4,0])
pode_despejar([5,5],[7,3])
permitido([7,3],[4,0])
pode_despejar([7,3],[0,3])
permitido([0,3],[4,0])
pode_despejar([0,3],[3,0])
permitido([3,0],[4,0])
pode_despejar([3,0],[3,5])
permitido([3,5],[4,0])
pode_despejar([3,5],[7,1])
permitido([7,1],[4,0])
pode_despejar([7,1],[0,1])
permitido([0,1],[4,0])
pode_despejar([0,1],[1,0])
permitido([1,0],[4,0])
pode_despejar([1,0],[1,5])
permitido([1,5],[4,0])
pode_despejar([1,5],[6,0])
permitido([6,0],[4,0])
pode_despejar([6,0],[6,5])
permitido([6,5],[4,0])
pode_despejar([6,5],[7,4])
permitido([7,4],[4,0])
pode_despejar([7,4],[0,4])
permitido([0,4],[4,0])
pode_despejar([0,4],[4,0])
permitido([4,0],[4,0])
pode_despejar([4,0],[4,0])
```

Voce quer outra solucao? nao.

CAPÍTULO 4

DESCRIÇÃO DO SISTEMA

4.1 INTRODUÇÃO

Vamos apresentar a seguir um sistema de representação de conhecimento e raciocínio, por nós desenvolvido, que pode ser usado na construção do Núcleo de um Sistema Especialista.

Este sistema, baseado no sistema Theorist de Poole et al [Poole-86a], utiliza um mecanismo de raciocínio dedutivo a partir da construção automática de teorias científicas; isto é, ao invés de deduzir teoremas monotonicamente de uma teoria lógica fixada, o sistema distingue asserções de hipóteses e tenta usar dedução para construir teorias consistentes onde observações são consequências lógicas.

- asserções são "fórmulas bem formadas" que sabemos que são verdadeiras no domínio que está sendo representado.

- possíveis hipóteses são expressões que estamos preparados a aceitar como parte de uma explicação. Se estas hipóteses são "default", obtemos "raciocínio por default". Se elas são possíveis problemas obtemos diagnose e, se permitirmos que sejam generalizações de observações ou leis derivadas de observações, obtemos aprendizagem

- observações são observações a serem explicadas.

Para representarmos isso mais formalmente, considere:

F o conjunto de asserções,

H o conjunto de hipóteses

Dizemos que G é o conjunto de observações explicáveis se existir um conjunto finito D de instâncias de elementos de H (D H) tal que:

F uniao D é consistente e

F uniao D $\models$ G

D é dito a teoria que explica G, ou seja, D é a teoria científica que é parte da teoria "lógica" que é o fecho dedutivo de F unido D.

Segue abaixo um diálogo com o sistema, usando o exemplo de "passaros voam" (citado no Capítulo 2). A Base de Conhecimento é formada pelas seguintes declarações:

default: voa(X) -> passaro(X).

n voa(X) -> pinguim(X).

passaro(tico).

Se interrogarmos ao sistema se "tico" voa obteremos uma resposta afirmativa:

explique(voa(tico)).

Teoria: voa(tico).

Voce quer outra teoria? sim.

... Sem solucao.

Existe uma explicação consistente para *voa(tico)*, porque *voa(tico)* pode ser explicado inicialmente pela teoria *voa(X) -> passaro(X)*.

Porém, se adicionarmos agora a asserção de que tico é um pinguim, esta teoria não é mais consistente, uma vez que a sua negação *n voa(tico)* pode ser provada. Nos não podemos dar nenhuma explicação de que tico voa, como mostra o programa a seguir:

default: voa(X) -> passaro(X).

n voa(X) -> pinguim(X).

passaro(tico).

pinguim(tico).

explique(voa(tico)).

... Sem solucao!

4.2 DIAGNOSE E APRENDIZAGEM

A maioria dos sistemas de diagnose baseados em regras usam o seguinte conhecimento:

SE observação ENTÃO problema.

Por exemplo, alguns sistemas baseados em diagnósticos médicos expressam o conhecimento em regras do tipo:

SE sintoma ENTÃO doença

Entretanto, a diagnose é um problema oposto a esse caso: um diagnóstico não é consequência lógica de nossas observações sobre um paciente mas sim as observações que devem ser as consequências lógicas do nosso conhecimento e diagnóstico.

O sistema que estamos propondo pode ser utilizado para diagnose se considerarmos as doenças e anomalias como hipóteses (default) que estamos preparados a aceitar como um diagnóstico e os sintomas como observações. As teorias são descrições detalhadas do que está acontecendo para produzir os sintomas.

Sendo assim, uma diagnose é a teoria mais simples que

explica as observações.

Uma teoria A é dita "mais simples" que outra teoria B se A é subconjunto de B. Dessa forma, a diagnose exclui as teorias que contém hipóteses estranhas considerando apenas o conjunto mínimo de hipóteses necessárias para explicar as observações.

Se estivermos agora interessados em acumular regras para explicar novas observações, certamente não consideraremos a teoria mais específica, como no caso da diagnose, e sim a teoria mais geral. Podemos perceber, portanto, que a aprendizagem é um problema oposto ao de diagnose.

A noção de construção de teorias é usada em vários modelos de aprendizagem de máquina, onde a maior ênfase está em estratégias para gerar e utilizar hipóteses úteis.

4.3 REPRESENTAÇÃO DE CONHECIMENTO

O conhecimento do nosso sistema será formalizado através de "frames" [Minsky-75]. As descrições dos slots são as seguintes:

- *asser*
- *contrapo*
- *default*
- *ako*
- *if_needed*
- *if_ask*

(a) *asser* - para representar fórmulas verdadeiras e consistentes

da B.C.

(b) *contrapo* - para as contrapositivas geradas pelo sistema.

(c) *default* - para representação de conhecimento inexato ou incompleto.

Por exemplo, consideramos geralmente que uma pessoa é de origem alemã se possuir determinadas características:

```
default: alemao(X) -> cor_olhos(X,azul),  
                loiro(X),  
                pele_clara(X).
```

Se tivermos uma asserção do tipo:

```
gosta_de(maria,X) -> alemao(X),  
                estuda(X,quimica),  
                toca(X,piano).
```

podemos explicar a observação "gosta_de(maria,X)" usando este *default* para a prova. O sistema deverá então executar a prova mostrando as teorias consistentes.

(d) *ako* - abreviação de "a kind of" - usado para representação de relações hierárquicas.

Por exemplo, a classificação de animais obedece a seguinte hierarquia:

```
animals { vertebrados  
          invertebrados
```

vertebrados	{	mamíferos
		aves
		peixes
		répteis
		anfíbios

mamíferos	{	carnívoros
		quirópteros
		.
		primatas

carnívoros	{	cachorro
		gato
		.
		lobo

Podemos representar essas relações hierárquicas na B.C. da seguinte maneira:

```
vertebrados(ako,animal)
mamíferos(ako,vertebrados)
carnívoros(ako,mamíferos)
cachorros(ako,carnívoro)
```

Suponhamos que temos uma asserção do tipo:

mamiferos(reproducao ,sexuada)

na B.C. e interrogamos ao sistema a observação

cachorros(reproducao,X).

Através de *ako* o sistema sabe que "cachorros" herda as propriedades de seus antepassados, ou seja, cachorros possui todas as propriedades de carnívoros que por sua vez possui todas as propriedades de mamíferos,...etc. Assim, a prova é bem sucedida com *X = sexuada*.

A descrição *ako* facilitará a transferencia de conhecimento de um nível para outro.

(e) *if_needed* - para utilizar programas PROLOG.

Por exemplo, para que um aluno seja aprovado em determinada disciplina, a média aritmética de suas notas não pode ser inferior a 5.0. Representariamos esse conhecimento da seguinte maneira:

aprovado(X) -> media_aritmetica(X,Y),

Y => 5.0.

Os programas que compõem o corpo desta cláusulas deverão ser descritos na B.C. como *if_needed* significando que eles são necessários para explicar uma observação, isto é, eles serão provados diretamente por Prolog:

```

if_needed:media_aritmetica(X,Y) -> notas(X,Lista),
                                   soma(Lista>Total),
                                   length(Lista,N),
                                   Y is Total/N.

if_needed:Y => X.

```

Os programas *length* e *is* são pré-definidos do Prolog, mas *notas* e *soma* são programas Prolog definidos por nós. Por isso, guardaremos os programas usando a descrição *pro*:

```

pro:notas(andre,[6.5,8.0,4.3,5.5]).
pro:notas(juca,[2.0,9.0,7.5,6.0]).
pro:soma([],0).
pro:soma([Cab:Cauda],S) -> soma([Cauda],S1),
                           S is S1 + Cab.

```

(f) *if_ask* - para indicar a necessidade de se fazer perguntas ao usuário.

Por exemplo, se o sistema deseja saber se o paciente é alérgico a algum antibiótico, a B.C. deve conter uma descrição do tipo:

```

if_ask:alergia_paciente(X).
pro:pergunta([alergia,X],[ 'A que antibiotico o paciente e
                           alergico? ' ],[u,critica_alergia]).

```

O Módulo de Coleta de Evidências é responsável pela manipulação deste "slot". Os argumentos e significado da relação "pergunta" são explicados em detalhe no Capítulo 5, seção 5.3.

4.4 LINGUAGEM DE INTERAÇÃO

Os dados são introduzidos na B.C. através do Modulo de Aquisição de Conhecimento. A linguagem usada como interface é semelhante à linguagem do Calculo de Predicados de Primeira Ordem, ou seja, o sistema aceitara formulas do tipo:

$$L_1 ; L_2 ; \dots ; L_n \rightarrow L_{n+1} , L_{n+2} , \dots L_{n+m}.$$

onde cada L_i é um literal arbitrário (negativo ou positivo).

Os conectivos logicos são definidos pelos seguintes operadores:

n indicando negação
, para conjunção
; para disjunção exclusiva
 $\rightarrow$ indicando if

Na notação BNF, a gramatica da nossa linguagem sera:

$\langle \text{sentenca} \rangle$	::=	$\langle \text{expldef} \rangle \langle \text{clgeral} \rangle ;$ $\langle \text{if} \rangle \langle \text{cabecapositiva} \rangle ;$ $\langle \text{pro} \rangle \langle \text{clausula} \rangle ; \langle \text{clgeral} \rangle$
$\langle \text{clgeral} \rangle$	::=	$\langle \text{disjuntiva} \rangle ; \langle \text{disjuntiva} \rangle \rightarrow \langle \text{corpo} \rangle$
$\langle \text{disjuntiva} \rangle$	::=	$\langle \text{cabecageral} \rangle ;$ $\langle \text{cabecageral} \rangle ; \langle \text{disjuntiva} \rangle$
$\langle \text{corpo} \rangle$	::=	$\langle \text{cabecageral} \rangle ; \langle \text{cabecageral} \rangle , \langle \text{corpo} \rangle$
$\langle \text{cabecageral} \rangle$	::=	$n \langle \text{cabecapositiva} \rangle ; \langle \text{cabecapositiva} \rangle$
$\langle \text{cabecapositiva} \rangle$	::=	$\langle \text{relacao} \rangle \langle \text{lista} \rangle ; \langle \text{relacao} \rangle$
$\langle \text{clausula} \rangle$	::=	$\langle \text{cabecapositiva} \rangle ;$ $\langle \text{cabecapositiva} \rangle \rightarrow \langle \text{corpo} \rangle$

<code><expldef></code>	<code>::=</code>	<code>expl ; default ; (expl,default)</code>
<code><if></code>	<code>::=</code>	<code>if_needed ; if_ask</code>
<code><pro></code>	<code>::=</code>	<code>pro ; prolog</code>
<code><relacao></code>	<code>::=</code>	<code>nome de uma relação na sintaxe Arity-PROLOG</code>
<code><lista></code>	<code>::=</code>	<code>lista não vazia</code>

Dessa forma, as asserções deverão ser definidas por
general.

Se desejarmos introduzir a asserção na lista de explicações devemos defini-la da seguinte maneira:

```
expl: <clgeral>.
```

As hipóteses serão definidas na forma:

```
default:<clgeral>.
```

ou ainda:

```
(expl,default):<clgeral>.
```

para as hipóteses que serão incluídas na lista de explicações.

As relações perguntáveis deverão ser expressões da forma:

```
if ask: <cabecapositiva>.
```

As regras que devem ser provadas diretamente por Prolog
deverão ser definidas por:

If needed: <cabecapositiva>.

Os programas Prolog serão introduzidos na forma:

pro:<clausula> ou

prolog: <clausula>.

4.5 "ESQUELETO" DO SISTEMA ESPECIALISTA

Com base na descrição dada acima e no Provador Automático de Teoremas apresentado no Capítulo 3, vamos apresentar no Capítulo seguinte o Núcleo do Sistema Especialista por nós implementado.

A Base de Conhecimento deste sistema deverá ser introduzida pelo Engenheiro de Conhecimento utilizando-se a linguagem apresentada na seção anterior.

Através das relações perguntáveis, o Módulo de Coleta de Evidências se encarregará de interagir com o usuário coletando informações para a Base de Dados.

A partir dessas informações e dos dados da Base de Conhecimento, o Núcleo do Sistema Especialista deverá efetuar diversas inferências buscando soluções para o problema.

CAPÍTULO 5

IMPLEMENTAÇÃO DO SISTEMA

5.1 INTRODUÇÃO

O sistema proposto foi implementado seguindo o modelo apresentado em [Monard-86]. Como foi citado anteriormente, a linguagem de programação utilizada foi o Arity-PROLOG, onde os programas pré-definidos desta linguagem usados na implementação são descritos no APENDICE deste trabalho.

Os programas definidos por nós serão apresentados a seguir cujos argumentos serão descritos da seguinte maneira:

[E]. <arg-n>: n-ésimo argumento de entrada.

[S]. <arg-n>: n-ésimo argumento de saída.

[A]. <arg-n>: n-ésimo argumento de entrada e saída.

5.2 MÓDULO DE AQUISIÇÃO DE CONHECIMENTO

Os dados da Base de Conhecimento são coletados pelo Módulo de Aquisição de Conhecimento, seja através da console ou através de arquivos em disco. Este Módulo se encarregará de gerar e armazenar as contrapositivas de cada termo lido registrando todas as informações numa representação em que o Núcleo do Siste-

ma Especialista ira reconhecer e manipular.

O programa *consulte* lê todos os termos de um determinado arquivo atraves do programa *leia*:

```
consulte(Arq):-  
    open(H,Arq,r),  
    read(H,Termo),  
    leia(H,Termo),  
    close(H).
```

O programa *leia* possui dois argumentos:

[E]. <arg-1>: refere-se ao "handle" do arquivo.
[A]. <arg-2>: ultimo termo lido.

```
1- leia(H,end_of_file):-!,  
2- leia(H,(pro:Termo)):-  
    !,  
    transforma(Termo,Termo1),  
    assertz(Termo1),  
    read(H,ProxTermo),  
    leia(H,ProxTermo).  
3- leia(H,Termo):-  
    str(Termo),  
    read(H,ProxTermo),  
    leia(H,ProxTermo).
```

1- Quando o ultimo termo lido for "end_of_file", a execução de *leia* termina.

2- Se o termo lido for um termo Prolog, esta clausula devera substituir a negação verdadeira pela negação de Prolog e grava este termo diretamente na B.C. A seguir lê o proximo termo.

3- Esta clausula gera as contrapositivas do ultimo termo lido e armazena tudo na B.C. atraves do programa *str*. Em seguida, lê o proximo termo e repete-se o processo ate o ultimo termo do arquivo.

```

transforma((A -> B),(A1:-B1)) :-
    !,
    substneg(A,A1),
    substneg(B,B1).

transforma(A,A1) :-
    substneg(A,A1).

```

O programa *transforma* traduz o termo da sintaxe de leitura para a sintaxe de Prolog puro, substituindo a negação verdadeira pela negação de Prolog.

O programa *substneg* se incumbirá de realizar a substituição da negação verdadeira pela negação do Prolog:

```

substneg((A;B),(A1;B1)) :-
    !,
    substneg(A,A1),
    substneg(B,B1).

substneg((A,B),(A1,B1)) :-
    substneg(A,A1),
    substneg(B,B1).

substneg(n(A),not(A)) :- !.

substneg(A,A).

```

O programa *str* gera uma cláusula logicamente equivalente de Termo, atribui uma chave para ela e a armazena na B.C. sob esta chave. Em seguida o programa falha a fim de gerar todas as cláusulas logicamente equivalentes deste Termo.

```

str(Termo):-
    clausulas(Termo,C),
    chave(C,Ch),
    recordz(Ch,C,Ref),
    fail.

str(Termo).

```

O programa *clausulas* é:

- 1- *clausulas*((*default*:*B*),*default*(*X*,*Y*,*n*)):-
 equivalentes(*B*,*claus*(*X*,*Y*)).
- 2- *clausulas*((*expl*,*default*):*B*),*default*(*X*,*Y*,*s*)):-
 equivalentes(*B*,*claus*(*X*,*Y*)).
- 3- *clausulas*((*expl*:*B*),*asser*(*X*,*Y*,*s*)):-
 conjuncao(*B*,*C*),
 ass(*C*,*claus*(*X*,*Y*)).
- 4- *clausulas*((*expl*:*B*),*contrapo*(*X*,*Y*,*s*)) :-
 conjuncao(*B*,*C*),
 contrapositivas(*C*,*claus*(*X*,*Y*)).
- 5- *clausulas*(*B*,*asser*(*X*,*Y*,*n*)) :-
 B $\models$ (*F*:*G*),
 conjuncao(*B*,*C*),
 ass(*C*,*claus*(*X*,*Y*)).
- 6- *clausulas*(*B*,*contrapo*(*X*,*Y*,*n*)):-
 B $\models$ (*F*:*G*),
 conjuncao(*B*,*C*),
 contrapositivas(*C*,*claus*(*X*,*Y*)).
- 7- *clausulas*((*if_needed*:*B*),*if_needed*(*B*)) :- *true*.
- 8- *clausulas*((*if_ask*:*B*),*if_ask*(*B*)) :- *true*.

onde:

1- refere-se aos termos *default*. Todas as cláusulas *default* serão geradas através do programa *equivalentes*. Elas serão guardadas na forma *default*(*X*,*Y*,*n*) onde *X* é o conseqüente das cláusulas *default* e *Y* a lista dos antecedentes. O terceiro parâmetro "*n*" indicará que as cláusulas *default* não constam na lista de explicações.

2- refere-se aos termos (*expl*,*default*). As contrapositivas serão guardadas da mesma forma que os termos *default*, com a única diferença que elas deverão constar na lista de explicações.

3- refere-se aos termos *expl*. Será gerada aqui apenas uma cláusula com um consequente. Ela será guardada na forma *asser(X,Y,s)* onde "s" indica que essa cláusula consta na lista de explicações.

4- Serão geradas aqui as contrapositivas da asserção gerada pela cláusula anterior. Elas também serão guardadas na lista de explicações.

5- Refere-se também as asserções. Procede de maneira análoga a (3). A diferença é que a cláusula não constará na lista de explicações.

6- Análogo a (4); porém as contrapositivas não constarão na lista de explicações.

7- refere-se aos termos *if_needed*. Gera cláusulas Prolog para o N.S.E.

8- refere-se aos termos *if_ask*. Semelhante a *if_needed*.

O programa *equivalentes* transforma as disjunções do termo lido em implicações da seguinte maneira:

```
1- equivalentes(A,claus(A,[ ])):-  
    A \= (B->C),  
    A \= (D;E),  
    !.
```

```
2- equivalentes(A,Equiv) :-  
    conjuncao(A,Conj),  
    def(Conj,Equiv).
```

onde:

1- significa que não há nada a ser transformado (a lista vazia do

segundo parâmetro indica que a cláusula gerada não possui nenhum antecedente).

2- O programa *conjuncao* transforma a cláusula na forma de uma conjunção de literais. Feito isso, o programa *def* irá construir as fórmulas equivalentes através das negativas de cada literal.

Para guardar uma asserção na B.C., o termo lido será transformado em conjunção de literais através do programa *conjuncao* e o programa *ass* irá então gerar apenas uma cláusula com um conseqüente:

```
ass([A;B],claus(NA,B)) :-  
    neg(A,NA).
```

O programa *def* gerará as cláusulas para default:

```
def([A;B],claus(C,B)) :-  
    A = (n C).  
  
def([A;B],claus(ND,[A;C])) :-  
    def(B,claus(ND,C)).
```

O programa *conjuncao* é:

```
1- conjuncao(( A-> B ),Conj) :-  
    !,  
    lado_direito(B,ListaB),  
    lado_esquerdo(A,ListaB,Conj).  
  
2- conjuncao(A,Conj) :-  
    lado_esquerdo(A,[],Conj).
```

onde:

1- refere-se a uma <clgeral>. A cláusula será transformada numa lista de conjunções através da concatenação da lista de conjun-

pões do lado direito da cláusula com a lista de conjunções do lado esquerdo da cláusula.

2- refere-se a uma <disjuntiva>. A lista de conjunções é construída de forma similar ao caso 1.

O programa *lado_direito* simplesmente transforma o <corpo> de uma cláusula numa lista de conjunções:

```
1- lado_direito( ( A, B ),[A/C]) :-  
    !,  
    lado_direito(B,C).  
2- lado_direito(A,[A]).
```

onde:

1- refere-se a <corpo> com mais de uma <cabecageral>.
2- refere-se a <corpo> com somente uma <cabecageral>.

O programa *lado_esquerdo* nega os termos da <disjuntiva> e constroi a lista de conjunções juntamente com os termos de *lado_direito*:

```
1- lado_esquerdo(( A, B ),Lista,[NA/Resto]) :-  
    !,  
    neg(A,NA),  
    lado_esquerdo(B,Lista,Resto).  
2- lado_esquerdo( A, Lista, [ NA/Lista ]) :-  
    neg( A, NA ).
```

onde:

1- refere-se a <disjuntiva> com mais de uma <cabecageral>.
2- refere-se a <disjuntiva> com somente uma <cabecageral>.

Por fim, o programa *contra* gera as contrapositivas

negando cada um dos termos da lista de conjunções:

```
contra([A;B],claus(NA,B)) :-  
    neg(A,NA).
```

```
contra([A;B],claus(NW,[A;S])) :-  
    contra(B,claus(NW,S)).
```

O programa *neg* gera a negativa de um termo:

```
neg(A,n(A)) :- A\=n(Y).  
neg(n(A),A) :- A\= n(Y).
```

Como foi citado anteriormente, as cláusulas serão armazenadas na B.C. sob uma determinada chave. As chaves de cada caso particular são determinadas abaixo pelo segundo argumento:

```
chave(asser(n(C),X,Y),C) :- !.  
chave(asser(C,X,Y),C) :- !.  
chave(default(n(C),X,Y),C) :- !.  
chave(default(C,X,Y),C) :- !.  
chave(contrapo(n(C),X,Y),C) :- !.  
chave(contrapo(C,X,Y),C) :- !.  
chave(if_needed(A),A) :- !.  
chave(if_ask(A),A) :- !.
```

Ativaremos as cláusulas da B.C através das chaves pelo comando *recorded*:

```
asser(n(X),Y,Z) :-  
    recorded(X,asser(n(X),Y,Z),Ref).  
  
asser(X,Y,Z) :-  
    recorded(X,asser(X,Y,Z),Ref).
```

```

default(n(X),Y,Z) :-
    recorded(X,default(n(X),Y,Z),Ref).

default(X,Y,Z) :-
    recorded(X,default(X,Y,Z),Ref).

contrapo(n(X),Y,Z) :-
    recorded(X,contrapo(n(X),Y,Z),Ref).

contrapo(X,Y,Z) :-
    recorded(X,contrapo(X,Y,Z),Ref).

if_needed(A) :-
    recorded(A,if_needed(A),Ref).

if_ask(A) :-
    recorded(A,if_ask(A),Ref).

```

O programa *consulte_console* lê termos diretamente da console através do programa *leia*:

```

consulte_console(Tipo) :-
    cls,
    repeat,
    write(Tipo),
    write( :),
    read(Sentenca),
    ((Sentenca = end,!);
    (Clausula=Tipo:Sentenca,
    leia(Clausula),fail)).

```

O programa *leia* lê os termos da console gerando e armazenando todas as suas contrapositivas (da mesma forma que foi feito para termos de um arquivo) até que o termo lido seja "end".

```

leia(end) :- !.
leia((pro:Termo) :-
    !,
    transforma(Termo,Termo1),
    assertz(Termo1),
    consulte_console.
leia(Termo) :-
    str(Termo),
    consulte_console.

```

5.3 O MÓDULO DE COLETA DE EVIDÊNCIAS

Dado um determinado problema, o N.S.E. deverá buscar todas as informações necessárias para chegar a algumas conclusões.

O Módulo de Coleta de Evidências é o meio de comunicação entre o usuário e o Sistema Especialista. Este módulo interage com o usuário formulando as perguntas necessárias, cujas respostas serão utilizadas pelo Núcleo do Sistema Especialista para inferir certas deduções.

O Núcleo do Sistema Especialista ativa o programa *faz_pergunta* sempre que necessitar de alguma informação declarada anteriormente como *if_ask*:

```
faz_pergunta(Goal,GoalAnt,Classe):-  
    pergunta(Goal,Perg ,Tcritica),  
    pergunte(Goal,GoalAnt,Perg,Tcritica,Classe).
```

Este programa possui três argumentos:

[A]. <arg-1>: a meta a ser provada.

[E]. <arg-2>: a meta anterior à meta a ser provada usada para justificar o porque da pergunta

[S]. <arg-3>: classe da resposta dada pelo usuário. As respostas se classificarão em:

sim - no caso do usuário responder a todas as perguntas ou no caso da resposta ser afirmativa.

nao - no caso da pergunta ser negada.

nao-sei - no caso do usuário responder que desconhece alguma resposta.

O programa *faz_pergunta* irá verificar se existe alguma cláusula *pergunta* na Base de Conhecimento. Esta cláusula deverá possuir os seguintes argumentos:

[A]. <arg-1>: a meta a ser provada que será preenchida com as respostas do usuário.

[E]. <arg-2>: o texto da pergunta a ser formulada.

[E]. <arg-3>: o tipo de resposta e os nomes das funções de crítica.

Os tipos de respostas são:

(1) s-n ou sn	para respostas do tipo sim ou não.
(11) u ou unica	para respostas únicas.
(111) v ou varias	para várias respostas.

(1) Tipo de resposta s-n: para perguntas que necessitam de respostas afirmativas ou negativas.

(11) Tipo de resposta unica: para perguntas que devem ser feitas somente uma vez.

(111) Tipo de resposta varias: para perguntas que admitem muitas respostas. A pergunta é feita várias vezes até que o usuário responda basta ou pare.

Por exemplo:

(1) Se quiséssemos saber se um paciente tem febre poderíamos ter a seguinte cláusula *pergunta*:

```
if_ask: tem_febre.  
pro: pergunta([tem_febre],[O paciente tem febre?],[s-n]).
```

(11) Se quiséssemos saber o tipo sanguíneo do paciente poderia-

mos ter:

```
if_ask: tipo_sanguineo(Tipo,Rh).
pro: pergunta([tipo_sanguineo,Tipo,Rh],[Qual e o tipo de
sangue e o tipo de Rh do paciente?'],
[u,critica_tipo,critica_rh]).
```

As funcoes de crítica são fornecidas ao sistema como programas PROLOG:

```
critica_tipo(Tipo) :- on(Tipo,[a,b,ab,o]),
!.
```

```
critica_tipo(Tipo) :- write('Resposta deve ser a,b,ab ou o'),
fail.
```

```
critica_rh(Rh) :- on(Rh,[+,-]),
!.
```

```
critica_rh(Rh) :- write('Resposta deve ser + ou -'),
fail.
```

(111) Se quisermos saber quais os órgãos danificados de um paciente poderíamos ter:

```
if_ask:orgao(Org).
pergunta([orgao,Org],[Qual o orgao danificado?'],
[v,critica_orgao]).
```

```
critica_orgao(Org) :- on(Org,[rim,coracao,pulmao,...]),
!.
```

```
critica_orgao(Org) :- write('Respostas validas sao rim,
coracao,...'),
fail.
```

Após o programa *faz_pergunta* verificar se existe alguma cláusula *pergunta* na B.C., o programa *pergunte* será ativado para fazer as perguntas ao usuário. Os detalhes de implementação deste Módulo são apresentados em [Monard-88].

5.4 NÚCLEO DO SISTEMA ESPECIALISTA

O Núcleo do Sistema Especialista processará os dados da Base de Conhecimento buscando conclusões alternativas para o problema em questão. O núcleo é ativado pelo programa *explique*:

```
explique(G) :-
    lado_direito(G,Lista),
    simpl_neg(Lista,Listal),
    prove_resto(Listal,[],[],[],T,[],E),
    write('Teoria: '),
    write(T),
    nl,
    write('Explicacao:'),
    escreva_invertido(E),
    nl,
    write('Voce quer outra teoria?'),
    read(R),
    on(R,[nao,no,n]),
    !.
```

```
explique(G) :-
    write(' ...Sem solucao!'),
    nl.
```

O programa *prove_resto* determina o caminho para se chegar a conclusão do problema que está sendo analisado. Seus argumentos são:

[A]. <arg-1>: conjunto de metas a ser provado.

[A]. <arg-2>: lista de metas antepassadas.

[A]. <arg-3>: classes antepassadas (ako) da submeta a ser provada.

[A]. <arg-4>: teorias (default) usadas para provar a meta principal.

[S]. <arg-5>: teorias usadas para provas bem sucedidas.

[A]. <arg-6>: teorias da classe expl e (expl,default) para provar a meta principal.

[S]. <arg-7>: lista de explicações para provas bem sucedidas.

1- *prove_resto*([],Antec,Ako,Teoria,Teoria,Expl,Expl):- true.

2- *prove_resto*([Goal|Goals],Antec,Ako,Teoria1,Teoria3,Expl1,Expl3) :-
 prove(Goal,Antec,Ako,Teoria1,Teoria2,Expl1,Expl2),
 prove_resto(Goals,Antec,Ako,Teoria2,Teoria3,Expl2,Expl3).

1- nesta cláusula a lista de metas está vazia significando que não há mais nada a provar.

2- esta cláusula prova um conjunto de metas provando a primeira meta e em seguida provando o restante do conjunto de metas.

O programa *prove* provará as metas uma a uma. Dessa forma possui os mesmos argumentos de *prove_resto*:

1- *prove*(Goal,Antec,Ako,Teoria,Teoria,Expl,Expl) :-
 neg(Goal,NotGoal),
 on(NotGoal,Antec).

2- *prove*(Goal,Antec,Ako,Teoria,NovaTeoria,ExplParcial,ExplFinal) :-
 asser(Goal,Body,InsFlag),
 not(*on*(Goal,Antec)),
 insira_expl(Goal,ExplParcial,Expl,InsFlag),
 prove_resto(Body,[Goal|Antec],Ako,Teoria,NovaTeoria,
 Expl,ExplFinal).

3- *prove*(Goal,Antec,Ako,Teoria,NovaTeoria,ExplParcial,ExplFinal) :-
 contrapo(Goal,Body,InsFlag),
 not(*on*(Goal,Antec)),
 insira_expl(Goal,ExplParcial,Expl,InsFlag),
 prove_resto(Body,[Goal|Antec],Ako,Teoria,NovaTeoria,
 Expl,ExplFinal).

4- *prove*(Goal,Antec,Ako,Teoria,NovaTeoria,Expl,ExplFinal) :-
 default(Goal,Body,Flag),
 not(*on*(Goal,Antec)),
 insira_expl(Goal,Expl,ExplParcial,Flag),
 prove_default(Goal,Antec,Ako,Body,Teoria,NovaTeoria,
 ExplParcial,ExplFinal).

```

5- prove(Goal,Antec,Ako,Teoria,NovaTeoria,ExplParcial,ExplFinal) :-
    find_ako(Goal,Ako,NovoGoal),
    prove(NovoGoal,Antec,Ako,Teoria,NovaTeoria,
        ExplParcial,ExplFinal).

6- prove(Goal,Antec,Ako,Teoria,NovaTeoria,ExplParcial,ExplFinal) :-
    entra_ako(Goal,Ako,AkoGoal,NovoAko),
    prove(AkoGoal,Antec,NovoAko,Teoria,NovaTeoria,
        ExplParcial,ExplFinal).

7- prove(Goal,Antec,Ako,Teoria,NovaTeoria,ExplParcial,ExplFinal) :-
    if_needed(Goal),
    not(on(Goal,Antec)),
    call(Goal),
    prove_resto([], [Goal|Antec], Ako, Teoria,
        NovaTeoria, ExplParcial, ExplFinal).

8- prove(Goal,Antec,Ako,Teoria,NovaTeoria,ExplParcial,ExplFinal) :-
    if_ask(Goal),
    resp(Goal,Flag),
    Flag = sim,
    not(on(Goal,Antec)),
    prove_resto([], [Goal|Antec], Ako, Teoria, NovaTeoria,
        ExplParcial, ExplFinal).

9- prove(Goal,Antec,Ako,Teoria,NovaTeoria,ExplParcial,ExplFinal) :-
    if_ask(Goal),
    Goal =.. [Rel|Args],
    not(resp([Rel|X],Flag)),
    faz_pergunta([Rel|X],Antec,Classe),
    assert(resp([Rel|X],Classe)),
    Goal =.. [Rel|X],
    Classe = sim,
    not(on(Goal,Antec)),
    prove_resto([], [Goal|Antec], Ako, Teoria, NovaTeoria,
        ExplParcial, ExplFinal).

```

onde:

- 1- confirma uma meta que é o complemento de uma meta antepassada.
- 2- prova asser. Verifica se a meta a ser provada não é iden-
tica a alguma meta antepassada; aqui é feita a detecção de possí-
veis "loops". A seguir insere esta meta na lista de explicações
através do programa *insira_expl*, insere esta meta na lista de
metas antepassadas e tenta provar o corpo da regra.

3- prova *contrapo*. Semelhante a cláusula 2; só que esta cláusula tenta fazer uma prova a partir de contrapositivas.

4- prova *default*. Semelhante a cláusula 2. Para realizar a prova, o programa *prove_default* será ativado.

5- prova *ako*. Verifica se existe um *ako* antepassado desta meta. Se existir, esta nova meta que contém a meta original deverá ser provada através do programa *prove*.

6- Se não existir um *ako* antepassado desta meta, o programa *entra_ako* será ativado a fim de procurar um *ako* da meta na B.C. A seguir, insere a meta na lista *ako* e chama *prove* para provar esta nova meta.

7- prova *if_needed*. Esta cláusula ativa um programa PROLOG.

8- prova *if_ask*. Se a meta a ser provada é uma relação perguntável e já consta na lista de respostas, o programa *prove_resto* será ativado incluindo esta meta entre as antepassadas sendo que a nova meta a ser provada é a lista vazia.

9- prova *if_ask* também. Porém, se a meta a ser provada não constar na lista de respostas, o programa *faz_pergunta* será ativado e a resposta do usuário será armazenado na Base de Dados. Se a resposta do usuário for confirmada, então *prove_resto* será ativado incluindo esta meta entre as antepassadas e nova meta a provar vazia.

O programa *prove_default* constrói as teorias através das hipóteses. Este programa possui oito argumentos:

[A]. <arg-1>: conjunto de metas a ser provado.

- [A]. <arg-2>: metas antepassadas.
- [A]. <arg-3>: classes antepassadas da submeta a ser provada.
- [E]. <arg-4>: corpo da meta default.
- [A]. <arg-5>: teorias usadas na prova da meta principal.
- [S]. <arg-6>: teorias usadas em provas bem sucedidas.
- [A]. <arg-7>: lista de explicações.
- [S]. <arg-8>: lista de explicações para uma prova bem sucedida.

```

1- prove_default(Goal,Antec,Ako,Body,Teoria,NovaTeoria,
                  ExplParcial,ExplFinal) :-
    on(Goal,Teoria),
    prove_resto(Body,Antec,Ako,Teoria,NovaTeoria,
                ExplParcial,ExplFinal).

2- prove_default(Goal,Antec,Ako,Body,Teoria,NovaTeoria,
                  ExplParcial,ExplFinal) :-
    not(on(Goal,Teoria)),
    prove_resto(Body,Antec,Ako,[Goal|Teoria],
                NovaTeoria,ExplParcial,ExplFinal),
    neg(Goal,NotGoal),
    not(prove(NotGoal,[],Ako,Teoria,Teoria,ExplParcial,E)).

```

onde:

- 1- verifica se a meta consta na lista de teorias antepassadas e ativa o programa *prove_resto* para provar o corpo da meta.
- 2- se a meta não constar na lista de teorias antepassadas, insere esta nova meta *default* na lista de teorias e chama *prove_resto* para provar o corpo da regra. A seguir verifica que não é possível provar a negação da meta usando a mesma teoria o que prova a consistência da teoria.

As metas do tipo *asser* que possuírem como terceiro argumento o parametro "s", deverão constar na lista de explica-

ções do caminho de prova. O programa *insira_expl* se encarrega de inserir as metas nesta lista:

```
insira_expl(Goal,ExplParcial,[Goal|ExplParcial],S) :-  
    not(on(Goal,ExplParcial)),  
    !.
```

```
insira_expl(Goal,Expl,Expl,_).
```

O programa *find_ako* verifica se a lista *ako* contém um antepassado da meta a ser provada. Se contiver, a meta antepassada passa a ser a nova meta que deverá ser provada. Este programa possui três argumentos:

[E]. <arg-1>: meta a ser provada.

[E]. <arg-2>: classes antepassadas usadas na prova.

[S]. <arg-3>: nova meta a ser provada.

```
find_ako(Goal,Ako,NovoGoal) :-  
    Goal = [GoalFun|GoalArgs],  
    on(ako(GoalFun,Pal,X),Ako),  
    NovoGoal = [Pal|GoalArgs],  
    NovoGoal \= X.
```

O programa *entra_ako* verifica se há uma cláusula *asser* na B.C. com o mesmo nome da relação da meta a ser provada de tipo *ako*. A seguir, verifica que esta nova meta não está na lista *ako* e a insere na lista *ako*.

Os argumentos de *entra_ako* são os mesmos de *find_ako* mais o seguinte argumento:

[S]. <arg-4>: nova lista de classes antepassadas.

```

entra_ako(Goal,Ako,AkoGoal,[ako(AkoFun,GoalFun,Goal),Ako]) :-
    Goal = [GoalFun;GoalArgs],
    asser([[GoalFun,Xako,AkoFun;Body]],InsFlag),
    Xako = ako,
    nonvar(AkoFun),
    not(on(ako(AkoFun,GoalFun,Goal)),Ako),
    AkoGoal = [AkoFun;GoalArgs].

```

Programas auxiliares:

```

escreva_invertido([]) :- !.
escreva_invertido([A;B]) :-
    escreva_invertido(B),
    write(A),
    nl.

```

```

not_prove(P) :-
    prove(P,[],[],[],T,[],E),
    !,
    fail.
not_prove(P) :-
    true.

```

```

simpl_neg([],[]) :- !.
simpl_neg([A;B],[C;D]) :-
    simples(A,C),
    simpl_neg(B,D).

```

```

simples(A,A):- A \= n(X).
simples(n A,n A) :- A \= n(X),!.
simples(n(n A),C) :- simples(A,C).

```

5.5 EXEMPLO

Vamos apresentar a seguir um exemplo de aplicação do sistema utilizando o raciocínio default. Considere o circuito de somador completo da figura abaixo:

Descrevemos através das asserções abaixo os tipos de portas da figura:

```
expl:porta(x1,xor).  
expl:porta(x2,xor).  
expl:porta(a1,and).  
expl:porta(a2,and).  
expl:porta(o1,or).
```

onde:

<arg-1> representa o nome da porta

<arg-2> especifica o tipo da porta

Chamamos de "c" o circuito total. As conexões entre as portas podem ser representadas da seguinte maneira:

```
expl:conectado(entrada(1,c),entrada(1,x1)).
expl:conectado(entrada(1,c),entrada(1,a1)).
expl:conectado(entrada(2,c),entrada(2,x1)).
expl:conectado(entrada(2,c),entrada(2,a1)).
expl:conectado(entrada(3,c),entrada(1,a2)).
expl:conectado(entrada(3,c),entrada(2,x2)).
expl:conectado(saida(1,x1),entrada(1,x2)).
expl:conectado(saida(1,x1),entrada(2,a2)).
expl:conectado(saida(1,a1),entrada(2,o1)).
expl:conectado(saida(1,a2),entrada(1,o1)).
expl:conectado(saida(1,x2),saida(1,c)).
expl:conectado(saida(1,o1),saida(2,c)).
```

As portas podem funcionar de diversas maneiras, dependendo do seu tipo. As asserções `tabela(T,E1,E2,S)` descrevem que a porta de tipo T com entradas E1 e E2 produzem saída S:

```
expl:tabela(and,on,on,on).
expl:tabela(and,off,on,off).
expl:tabela(and,off,off,off).
expl:tabela(and,on,off,off).
expl:tabela(or,off,off,off).
expl:tabela(or,on,off,on).
expl:tabela(or,off,on,on).
expl:tabela(or,on,on,on).
expl:tabela(xor,off,off,off).
expl:tabela(xor,on,on,off).
expl:tabela(xor,off,on,on).
expl:tabela(xor,on,off,on).
```

Vamos considerar agora a hipótese de que a saída das portas pode falhar. Podemos declarar essa hipótese no nosso sistema através da seguinte cláusula default:

```
default:falha(X).
```

O problema consiste em avaliar os valores das saídas do circuito considerando também as hipóteses de que as saídas das

portas podem falhar. Os valores das entradas deverão ser fornecidas pelo usuário quando o sistema solicitar. As regras a seguir mostram como o sistema faz as inferências:

- 1- *expl:valor(entrada(X,c),Y) ->
 entre(X,Y).*
- 2- *expl:valor(saida(1,D),Saida1) ->
 porta(D,T),
 tabela(T,Entrada1,Entrada2,Saida1),
 valor(entrada(1,D),Entrada1),
 valor(entrada(2,D),Entrada2).*
- 3- *expl:valor(saida(1,D),Saida1) ->
 porta(D,T),
 falha(D),
 tabela(T,Entrada1,Entrada2,Saida2),
 oposto(Saida2,Saida1),
 valor(entrada(1,D),Entrada1),
 valor(entrada(2,D),Entrada2).*
- 4- *expl:valor(Y,Z) ->
 conectado(X1,Y),
 valor(X1,Z).*

onde:

- 1- O valor de alguma entrada no circuito será solicitada pelo sistema através da relação perguntável *entre*.
- 2- Através do tipo de porta e dos valores das entradas da porta é possível avaliar o valor da saída da porta.
- 3- Nesta cláusula o sistema considera "por default" que uma determinada porta pode falhar. Se isso ocorre, o valor da saída é o oposto.
- 4- Quando há conexão entre entradas e/ou saídas das portas os valores são preservados.

Completamos com as cláusulas abaixo a Base de Conhecimento para o problema:

```
expl:oposto(on,off).  
expl:oposto(off,on).
```

```
if_ask:entre(X,Y).
```

```
pro:pergunta([entre,X,Y],[Qual o numero de entrada e o valor? ''],  
[V,critica_num,critica_entr]).
```

```
pro:critica_num(X) ->  
on(X,[1,2,3]),  
!.
```

```
pro:critica_num(X) ->  
write('Respostas validas sao 1,2 ou 3'),  
fail.
```

```
pro:critica_entr(Y) ->  
on(Y,[on,off]),  
!.
```

```
pro:critica_entr(Y) ->  
write('Respostas validas sao on ou off'),  
fail.
```

Se fornecermos por exemplo que todas as entradas são "off" e perguntar ao sistema por que todas as saídas do circuito são "on", o sistema deverá explicar:

```
explique((valor(saida(1,c),on),valor(saida(2,c),on))).
```

```
Qual o numero de entrada e o valor?  
1.  
off.
```

```
Qual o numero de entrada e o valor?  
2.  
off.
```

```
Qual o numero de entrada e o valor?  
3.  
off.
```

Qual o numero de entrada e o valor?
basta.

Teoria: [falha(a2),falha(x1)]
Explicacao:valor(saida(1,c),on)
conectado(saida(1,x2),saida(1,c))
valor(saida(1,x2),on)
porta(x2,xor)
tabela(xor,on,off,on)
valor(entrada(1,x2),on)
conectado(saida(1,x1),entrada(1,x2))
valor(saida(1,x1),on)
porta(x1,xor)
tabela(xor,off,off,off)
oposto(off,on)
valor(entrada(1,x1),off)
conectado(entrada(1,c),entrada(1,x1))
valor(entrada(1,c),off)
valor(entrada(2,x1),off)
conectado(entrada(2,c),entrada(2,x1))
valor(entrada(2,c),off)
valor(entrada(2,x2),off)
conectado(entrada(3,c),entrada(2,x2))
valor(entrada(3,c),off)
valor(saida(2,c),on)
conectado(saida(1,o1),saida(2,c))
valor(saida(1,o1),on)
porta(o1,or)
tabela(or,on,off,on)
valor(entrada(1,o1),on)
conectado(saida(1,a2),entrada(1,o1))
valor(saida(1,a2),on)
porta(a2,and)
tabela(and,off,on,off)
valor(entrada(1,a2),off)
conectado(entrada(3,c),entrada(1,a2))
valor(entrada(2,a2),on)
conectado(saida(1,x1),entrada(2,a2))
valor(entrada(2,o1),off)
conectado(saida(1,a1),entrada(2,o1))
valor(saida(1,a1),off)
porta(a1,and)
tabela(and,off,off,off)
valor(entrada(1,a1),off)
conectado(entrada(1,c),entrada(1,a1))
valor(entrada(2,a1),off)
conectado(entrada(2,c),entrada(2,a1))

Voce quer outra teoria?sim.

Teoria: [falha(a2),falha(x1)]
Explicacao:valor(saida(1,c),on)
conectado(saida(1,x2),saida(1,c))
valor(saida(1,x2),on)

```

porta(x2,xor)
tabela(xor,on,off,on)
valor(entrada(1,x2),on)
conectado(saida(1,x1),entrada(1,x2))
valor(saida(1,x1),on)
porta(x1,xor)
tabela(xor,off,off,off)
oposto(off,on)
valor(entrada(1,x1),off)
conectado(entrada(1,c),entrada(1,x1))
valor(entrada(1,c),off)
valor(entrada(2,x1),off)
conectado(entrada(2,c),entrada(2,x1))
valor(entrada(2,c),off)
valor(entrada(2,x2),off)
conectado(entrada(3,c),entrada(2,x2))
valor(entrada(3,c),off)
valor(saida(2,c),on)
conectado(saida(1,o1),saida(2,c))
valor(saida(1,o1),on)
porta(o1,or)
tabela(or,on,off,on)
valor(entrada(1,o1),on)
conectado(saida(1,a2),entrada(1,o1))
valor(saida(1,a2),on)
porta(a2,and)
tabela(and,off,off,off)
valor(entrada(1,a2),off)
conectado(entrada(3,c),entrada(1,a2))
valor(entrada(2,a2),off)
conectado(saida(1,x1),entrada(2,a2))
valor(saida(1,x1),off)
valor(entrada(2,o1),off)
conectado(saida(1,a1),entrada(2,o1))
valor(saida(1,a1),off)
porta(a1,and)
valor(entrada(1,a1),off)
conectado(entrada(1,c),entrada(1,a1))
valor(entrada(2,a1),off)
conectado(entrada(2,c),entrada(2,a1))

```

Voce quer outra teoria?sim.

O sistema consegue encontrar dessa forma 16 teorias para explicar as observações. As teorias encontradas são listadas pela tabela a seguir:

	a1	a2	o1	x1	x2
1		falha		falha	
2		falha		falha	
3	falha			falha	
4	falha			falha	
5	falha	falha		falha	
6	falha	falha		falha	
7			falha	falha	
8			falha	falha	
9		falha		falha	falha
10		falha			falha
11	falha			falha	falha
12	falha				falha
13	falha	falha		falha	falha
14	falha	falha			falha
15			falha	falha	falha
16			falha		falha

Podemos observar pela tabela que não temos para este problema nenhum diagnóstico, ou seja não existe aqui nenhuma teoria que seja mais simples que todas as outras teorias.

CONCLUSÕES

Apresentamos neste trabalho a implementação do Núcleo de um Sistema Especialista que permite declarações do tipo default onde se torna possível a construção automática de teorias científicas.

Seguindo esta idéia, foi implementado um Provador Automático de Teoremas através de uma extensão do Prolog. Como foi mostrado no Capítulo 3, este provador é capaz de solucionar problemas descritos através de uma linguagem de representação natural da Lógica de Primeira Ordem e onde o mecanismo de inferência possui um detector de possíveis "loops".

O provador por nós implementado foi baseado na idéia de Loveland que sugere a geração de contrapositivas para cada cláusula e sobre elas uma busca com algumas verificações significativas.

Para implementar este método podemos gerar todas as contrapositivas das cláusulas da Base de Conhecimento no início da execução ou também é possível gerar somente as contrapositivas que são necessárias durante uma prova. Este último método economiza memória na máquina mas consome maior tempo de execução.

Neste trabalho implementamos o primeiro método, pois embora as contrapositivas ocupem um pouco mais de memória na máquina, achamos mais interessante que elas estejam incorporadas na Base de Conhecimento.

Uma sugestão para futuros trabalhos é tentar outros métodos de implementação para a construção do Provador de Teoremas.

O tempo de execução do sistema é sempre razoável, se não for usada nenhuma teoria durante a prova. No caso de teorias serem usadas e dependendo da B.C. sentimos dificuldade na prova de consistências, pois a busca torna-se exponencial e a execução consideravelmente lenta.

Outro trabalho a ser prosseguido é a implementação do Módulo de Coleta de Evidências apresentado brevemente na seção 5.3. Algumas alterações já estão sendo estudadas, como por exemplo, o tratamento de resposta "nao-sei" dada pelo usuário. Esse tipo de resposta necessita de um tratamento especial, pois a interpretação para ela depende muito do contexto e do tipo da pergunta que está sendo feita.

A diagnose é também um problema a ser estudado com mais cautela. A idéia apresentada por Poole [Poole-86b] em que a diagnose é a teoria mais simples que explica as observações parece ser uma idéia muito simplista, no sentido de que todas elas são consideradas com a mesma importância. Esse tipo de comparação pode ser comprometedor, dependendo do diagnóstico que está sendo feito. Além disso, em casos reais, dificilmente encontramos a teoria mais simples de um dado problema.

Da mesma forma, pode-se discutir também o problema da aprendizagem que é um problema ainda mais complexo.

REFERENCIAS

[Arity-86]

Arity Corporation. The Arity/Prolog Programming Language. 1986

[Bratko-86]

Bratko,I. Prolog Programming for Artificial Intelligence. Addison-Wesley. 1986.

[Clocksin-81]

Clocksin,W.F.;Mellish,C.S. Programming in Logic. Springer-Verlag, 1981

[Colmerauer-85]

Colmerauer,A. Prolog in 10 Figures. CACM, Vol 18, No 12, pp 1297-1310, December 1985

[Etherington-87]

Etherington,D.W. Formalizing Nonmonotonic Reasoning Systems. Artificial Intelligence, Vol. 31, pp 41-85, 1987

[Gabbay-82]

Gabbay,D. Intuitionistic basis for non-monotonic logic. Proc. Conf. on Automated Deduction, Vol 6, pp 260-273, 1982

[Loveland-78]

Loveland,D.W. Automated Theorem Proving: A Logical Basis. North-Holland, 1978

[McCarthy-80]

McCarthy,J. Circumscription - A Form of Nonmonotonic Reasoning. Artificial Intelligence, Vol 13, pp 27-39, 1980

[McDermott-80]

McDermott,D.V.;Doyle,J. Nonmonotonic Logic I. Artificial Intelligence, Vol 13, pp 41-72, 1980

[Minsky-75]

Minsky,M. A Framework for Representing Knowledge. The Psychology of Computer Vision. McGraw-Hill, N.Y., 1975

[Moore-85]

Moore,R.C. Semantical Considerations on Nonmonotonic Logic. Artificial Intelligence, Vol 25, pp 75-94, 1985

[Monard-86]

Monard,M.C. Um Sistema de Representação de Conhecimento e Raciocínio por Default. Tese de Livre Docência do ICMSC-USP,1986

[Monard-88]

Monard,M.C.; Hebihara,S.M. Implementação Lógica de Um Módulo de Coleta de Evidências. Notas Técnicas do ICMSC [em preparação].

[Pelletier-86]

Pelletier,F.J. Seventy-five problems for testing Automatic Theorems Provers. J. of Automated Reasoning 2,pp 191-216, 1986

[Poole-85a]

Poole,D.;Goebel,R. On Eliminating Loops in Prolog. Thecnical Report CS-85-27, Logic Programming and A.I. Group, Dept. of Computer Science, University of Waterloo, Canada, August 1985

[Poole-85b]

Poole,D.;Goebel,R. Gracefully Adding Negation and Disjunction to Prolog. Thecnical Report CS-85-51, Logic Programming and A.I. Group, Dept. of Computer Science, University of Waterloo, Canada, December 1985

[Poole-86a]

Poole,D;Goebel,R;Aleliunas,R. Theorist: a Logical Reasoning System for Default and Diagnosis. Thecnical Report CS-86-06, Logic Programming and A.I. Group, Dept. of Computer Science, University of Waterloo, February 1986

[Poole-86b]

Poole,D. Default Reasoning and Diagnosis as Theory Formation. Thecnical Report CS-86-08, Logic Programming and A.I. Group, Dept. of Computer Science, University of Waterloo, Canada, March 1986

[Reiter-80]

Reiter,R. A Logic for Default Reasoning. Artificial Intelligence, Vol 13, pp 81-132, 1980

[Reiter-87]

Reiter,R. A Theory of Diagnosis From First Principles. Artificial Intelligence, Vol 32, pp 57-95, 1987

[Robinson-65]

Robinson,J.A. A Machine-oriented Logic Based on the Resolution Principle. JACM, Vol 12, No 1, pp 23-41, January 1965

[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

APENDICE - ARITY / PROLOG

Listamos neste apêndice os programas Arity-PROLOG pré-definidos utilizados durante o desenvolvimento do trabalho apresentado.

1) CONTROLE E ESTRUTURA DA LINGUAGEM

P,P

Especifica uma conjunção de metas.

P;P

Especifica uma disjunção de metas.

!

Símbolo de corte: controla e limita as buscas.

T 1 = T 2

Tentativa de unificar T1 com T2.

T 1 \ = T 2

Falha se T1 unifica com T2.

c a l l (P)

Executa um programa P (se P estiver instanciado); falha se P não está instanciado.

f a i l

Sempre falha.

n o t (P)

Falha se a meta P for satisfeita; caso contrário, será sucesso.

r e p e a t

Origina uma série de metas que serão executadas até uma condição verdadeira ser detectada.

t r u e

Sempre sucesso.

2) DEFINIÇÃO DE OPERADORES

O intervalo de definição de operadores em Arity/Prolog é [1;1200]. O predicado:

current_op(Prec,Assoc,Op)

retorna, através de retrocesso, as definições de operadores correntes na base de dados.

3) FUNÇÕES ARITMETICAS

Na tabela seguinte, X e Y são operandos de uma expressão aritmética:

EXPRESSÃO	OPERAÇÃO
$X + Y$	Adição
$X - Y$	Subtração
$X * Y$	Multiplicação
X / Y	Divisão

4) PREDICADOS ARITMETICOS AVALIAVEIS

Nas descrições seguintes, E1 e E2 são expressões aritméticas como descritas anteriormente na 'Definição de Operadores':

$E1 > E2$

Verifica se o valor de E1 é maior que o valor de E2.

$E1 < E2$

Verifica se o valor de E1 é menor que o valor de E2.

$E1 \geq E2$

Verifica se o valor de E1 é maior ou igual ao valor de E2.

$E1 \leq E2$

Verifica se o valor de E1 é menor ou igual ao valor de E2.

E1 ::= *E2*

Verifica se os valores de *E1* e de *E2* são iguais.

E1 \::= *E2*

Verifica se os valores de *E1* e de *E2* são diferentes.

X is *E*

Avalia *E* e unifica o valor com *X*.

5) COMPARAÇÃO DE TERMOS

Os termos são comparados de acordo com o padrão de ordenação de termos, como segue, do menor ao maior:

* Variáveis, numa ordem arbitrária.

* Inteiros e números com ponto flutuante, de -oo ao +oo.

* Átomos e strings, na ordem ASCII.

* Termos compostos, ordenados primeiro pela aridade, depois pelo nome do functor principal e recursivamente por seus argumentos.

Nas descrições seguintes, *T1* e *T2* são termos em Prolog:

T1 == *T2*

Verifica se os termos *T1* e *T2* são equivalentes.

T1 \== *T2*

Verifica se os termos *T1* e *T2* não são equivalentes.

6) CONVERSAO DE TERMOS

Struct = *..list*

Refere-se como "univ". Converte uma estrutura numa lista ou uma lista numa estrutura.

arg(N,Term,X)

Unifica *X* com o valor do *N*-ésimo argumento de um termo. (Os argumentos são numerados a partir de 1.)

argO(N,Term,X)

Unifica X com o valor do (N+1)-ésimo argumento de um termo. (Os argumentos são numerados a partir de 0.)

functor(Estr,X,N)

Retorna um nome de estrutura e sua aridade.

length(L,N)

Retorna o comprimento N de uma lista L de caracteres ou átomos.

7) CLASSIFICAÇÃO DE TERMOS

atom(X)

Verifica se X é um átomo.

atomic(X)

Verifica se X é um tipo de dado atômico.

float(X)

Verifica se X é um número com ponto flutuante.

integer(X)

Verifica se X é um inteiro.

nonvar(X)

Verifica se X está instanciado.

number(X)

Verifica se X é um inteiro ou número com ponto flutuante.

string(X)

Verifica se X é uma "string".

var(X)

Verifica se X é uma variável não instanciada.

8) COLEÇÃO DE TERMOS

bagof(X,P,Bag)

Coleta todos os casos de X onde a meta P está satisfeito e retorna estes casos de X numa lista desordenada (Bag).

findall(X,P,Set)

Coleta todos os casos de X onde a meta P está satisfeita e retorna esses casos numa lista não classificada contendo duplicatas (Set).

setof(X,P,Set)

Coleta todos os casos de X onde a meta P está satisfeita e retorna esses casos numa lista ordenada sem duplicatas (Set).

9) CONTROLE DA BASE DE DADOS

abolish(Name/Arity)

Remove da base de dados todas as cláusulas com o especificado nome e número de argumentos.

assert(Clause)

Adiciona uma cláusula no final de um predicado.

asserta(Clause)

Adiciona uma cláusula no início de um predicado.

assertz(Clause)

Adiciona uma cláusula no final de um predicado.

clause(Head,Body)

Retorna os metas com uma dada cabeça.

current_predicate(P)

Retorna através de "backtracking" os predicados que estão correntemente definidos na base de dados.

erase(Ref)

Remove o termo armazenado sob o dado numero de referência.

eraseall(Key)

Remove todos os termos armazenados sob a chave.

instance(Ref,Term)

Retorna o termo associado com o número de referência.

key(Key,Ref)

Retorna o número de referência da chave.

keys(Name/Arity)

Retorna, através de "backtracking", o nome da função e o número de argumentos de todas as chaves armazenadas na base de dados.

listing

Escreve, com saída padrão, todos os predicados da base de dados.

listing(Name)

Escreve, com saída padrão, todos os predicados da base de dados armazenados sob dado nome com qualquer número de argumentos.

nref(Ref,Next)

Retorna o número de referência do próximo termo da cadeia de termos que possuem a mesma chave.

recorda(Key,Term,Ref)

Adiciona um termo no início de uma cadeia de termos com a mesma chave e retorna o número de referência atribuído ao termo.

recorded(Key,Term,Ref)

Retorna através de "backtracking" os termos e associado, os números de referência armazenados sob uma chave dada.

recordz(Key,Term,Ref)

Adiciona um termo no final de uma coleção de termos com mesma chave e retorna o número de referência associado ao termo.

ref(X)

Determina se X é um número de referência.

retract(Clause)

Remove a cláusula da base de dados.

10) ENTRADA E SAÍDA PADRAO

currente_op(Prec,Assoc,Op)

Retorna através de "backtracking" as definições de operadores correntes na base de dados.

nl

Escreve uma linha em branco no artifício de saída padrão.

op(Prec,Assoc,Op)

Define ou retorna o precedente, a posição e a associatividade de um operador.

read(Term)

Le um termo da console.

write(Term)

Escreve o termo na console.

11) ENTRADA E SAIDA DE ARQUIVOS

close(Handle)

Fecha o arquivo especificado por Handle.

open(Handle,Filespec,Access)

Abre um arquivo existente.

read(Handle,Term)

Le um termo do arquivo especificado.

stdin(Filespec,Goal)

Redireciona a entrada padrão para a duração da meta.

stdinout(InFile,OutFile,Goal)

Redireciona a entrada e saída padrão para a duração da meta.

stdout(Filename,Goal)

Redireciona a saída padrão para a duração da meta.

write(Handle,Term)

Escreve o termo no arquivo especificado.

12) EXECUÇÃO DAS FUNÇÕES DO SISTEMA

[Filespec]

Lê as cláusulas de um arquivo, em "blocos" com o predicado "fast_consult".

consult(Filespec)

Lê as cláusulas de um arquivo acrescentando-as no fim da base de dados.

halt

Sai do interpretador ou da aplicação compilada.

reconsult(Filespec)

Lê as cláusulas da base de dados de um arquivo, trocando qualquer predicado que já está presente na base de dados com aquelas cláusulas que também ocorrem no arquivo.

save(Name)

Guarda o programa corrente em forma binária no arquivo especificado.

shell

Permite-se interagir com o sistema DOS sem sair do interpretador. Digitar "exit" do MS-DOS para retornar ao interpretador.

shell(DOSCommand)

Executa um comando MS-DOS e retorna ao interpretador logo após o comando ser executado.