# UNIVERSIDADE DE SÃO PAULO

## Meta-Interpretadores Prolog[1]

JOÃO LUIZ FRANCO

MARIA CAROLINA MONARD

Nº 6

## NOTAS DIDÁTICAS

**Instituto de Ciências Matemáticas de São Carlos**

ISSN - 0103-2585

# Meta-Interpretadores Prolog[1]

JOÃO LUIZ FRANCO

MARIA CAROLINA MONARD

Nº 6

N O T A S  D I D Á T I C A S

São Carlos (SP)
nov. 1992

# Meta-Interpretadores Prolog [1]

**João Luiz Franco**
Universidade de São Paulo / ILTC
Escola de Engenharia de São Carlos
Departamento de Hidráulica e Saneamento

**Maria Carolina Monard**
Universidade de São Paulo / ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística

## Resumo

Meta-programas são programas que tratam outros programas como dados. A meta-programação em Prolog é muito simples, principalmente devido à equivalência entre programas e dados, pois ambos são termos Prolog, e à capacidade de manipulação simbólica da linguagem Prolog.

Meta-interpretadores são uma classe particular de meta-programas. Um meta-interpretador para uma linguagem é um interpretador para a linguagem escrito na própria linguagem.

Este trabalho didático concentra-se em meta-interpretadores Prolog. Vários meta-interpretadores são apresentados, junto com exemplos de execução, mostrando, a cada passo, como eles podem ser incrementados para aplicações mais complexas.

**Novembro - 1992**

[1] Trabalho realizado com auxílio do CNPQ e RHAE/ILTC.

# Conteúdo

# 1 Introdução

Uma das principais utilidades da linguagem de programação Prolog é a prototipagem rápida, onde há interesse na implementação imediata de novas idéias, para que essas idéias possam ser rapidamente testadas antes de serem implementadas para real utilização [Amble 87]. Em prototipagem, a ênfase é implementar novas idéias com rapidez e a um custo pequeno, não importando muito a eficiência da implementação. Após desenvolvido um primeiro protótipo, o sistema pode ser implementado em uma outra linguagem de programação, caso seja necessário.

A linguagem Prolog é também apropriada para implementar outras linguagens, devido à facilidade de se escrever meta-programas em Prolog [Sterling 84] [Bowen 82].

Nesta Nota Didática apresenta-se o conceito de meta-programação, com ênfase em uma classe particular de meta-programas: os meta-interpretadores. Diversos meta-interpretadores são apresentados, junto com exemplos de execução. Com isto, pode-se ver, de forma detalhada, como eles podem ser incrementados para aplicações mais complexas.

As bases de conhecimento utilizadas para mostrar a ação desses meta-interpretadores são muito simples, a fim de facilitar o entendimento. Entretanto, é importante que o leitor, após entender cada meta-interpretador, desenvolva bases de conhecimento mais complexas e de seu interesse para serem manipuladas por esses meta-interpretadores. Este procedimento auxilia a compreensão da ação de cada meta-interpretador.

O trabalho está organizado da seguinte forma:

Na seção 2 são discutidos o conceito e o funcionamento de meta-programas e meta-interpretadores; na seção 3 são tratados meta-interpretadores Prolog. Na seção 4 são apresentadas diversas implementações, partindo de um meta-interpretador muito simples, que é incrementado passo a passo para manipular aplicações cada vez mais complexas. Na seção 5 é discutida a utilidade de meta-interpretadores na construção de Sistemas Especialistas e, finalmente, na seção 6 apresentamos nossas conclusões.

# 2 Meta-programas e Meta-interpretadores

Meta-programas são programas que tratam outros programas como dados. Eles analisam, transformam e simulam outros programas. Interpretadores e compiladores são exemplos de meta-programas [Monard 90].

A meta-programação em Prolog é muito simples [Furukawa 89], e essa facilidade deve-se, principalmente, a dois fatores. O primeiro deles é a equivalência entre programas e dados: ambos são termos Prolog. A capacidade de manipulação simbólica é outra característica que torna Prolog uma linguagem poderosa para meta-programação [Gallaire 82].

Meta-interpretadores são uma classe particular de meta-programas. Um meta-interpretador para uma linguagem é um interpretador para a linguagem escrito na própria

linguagem [Sterling 87]. Portanto, um meta-interpretador Prolog é um interpretador para a linguagem Prolog escrito em Prolog. A facilidade para escrever meta-interpretadores é uma poderosa característica de uma linguagem de programação, pois possibilita a construção de um ambiente de programação integrado e dá acesso ao processo computacional da linguagem [Sterling 86].

# 3 Meta-Interpretadores Prolog

Um interpretador realiza duas funções. A primeira delas é chamada ação sintática, onde a estrutura dos termos de entrada é analisada para verificar se ela pertence à linguagem com a qual o interpretador trabalha. A segunda função é chamada ação semântica, e corresponde a simular o procedimento operacional associado à estrutura que foi analisada.

Um meta-interpretador Prolog interpreta estruturas que são termos Prolog válidos — regras ou cláusulas de Horn. Estes interpretadores analisam uma regra unificando-a com esqueletos de estruturas aceitáveis pela linguagem, extraindo seus subtermos e analisando-os recursivamente. A fase de análise termina quando um símbolo atômico da linguagem é encontrado, mesmo que a interpretação possa continuar a partir da execução das ações semânticas associadas a este símbolo atômico. Os símbolos atômicos que acarretam interpretação de outras regras são chamados de símbolos não terminais. Os símbolos atômicos terminais, por sua vez, são aqueles que não acarretam interpretação, como, por exemplo, predicados pré-definidos do sistema. Neste caso, eles são considerados símbolos terminais.

Essa fase de análise — ação sintática — introduz um nível extra de interpretação em relação à execução direta do programa sem a utilização de meta-interpretadores. Isto significa que a utilização de meta-interpretadores diminui a eficiência de execução dos programas.

Basicamente, um meta-interpretador Prolog toma um programa Prolog — juntamente com uma meta Prolog — e executa a meta com respeito ao programa, isto é, o meta-interpretador tenta provar que a meta segue-se logicamente do programa. Porém, para ter real interesse, um meta-interpretador Prolog não deve comportar-se exatamente como o interpretador Prolog original, e sim oferecer alguma funcionalidade adicional, tal como gerar uma árvore de prova ou rastrear a execução de programas Prolog [Bratko 90].

A seguir, é apresentada a implementação de diversos meta-interpretadores, onde cada um deles possui alguma capacidade adicional relevante.

# 4 Implementação de Meta-Interpretadores

O programa a seguir — denominado Familia — pode ser considerado como uma Base de Conhecimento muito simples. Ele será utilizado para mostrar a execução dos diversos meta-interpretadores apresentados nas próximas seções.

**Programa 1** *Programa Familia*

```
tem_sobrinho(X) :- tio(X,Y),homem(Y).
tem_sobrinho(X) :- irmao(X,chris).

tio(A,B) :- irmao(A,C),pai(C,B).
tio(johnny,franklin).

pai(scott,rachel).
pai(david,bruce).
pai(bruce,dick).

irmao(clark,bruce).
irmao(alex,scott).
irmao(xavier,chris).
irmao(rick,dick).

homem(dick).
homem(franklin).
homem(bruce).
```

## 4.1 Um Meta-Interpretador Trivial

A relação `prove/1` será utilizada para implementar um meta-interpretador em Prolog. O meta-interpretador mais simples que pode ser escrito em Prolog, apresentado a seguir, usa uma meta-variável:

**Programa 2** *Meta-Interpretador Trivial*

```
prove(Meta) :- Meta.
```

Neste caso, todo o trabalho é deixado para o interpretador de Prolog, e o meta-interpretador Trivial comporta-se exatamente como o interpretador Prolog original. A seguir é mostrado um exemplo de execução utilizando o programa Familia apresentado anteriormente.

```
?- prove(tem_sobrinho(X)).
X = clark ->;

X = johnny ->;

X = xavier ->;
no
```

Com certeza, este meta-interpretador não tem valor prático, pois não fornece nenhuma característica adicional àquelas fornecidas pelo interpretador Prolog. Para que este objetivo possa ser atingido, é necessário um meta-interpretador mais refinado.

## 4.2 Um Meta-interpretador Básico

As diferenças entre meta-interpretadores podem ser caracterizadas em termos de sua *granularidade*, que é a quantidade de detalhes computacionais que são acessíveis ao programador. A granularidade do meta-interpretador Trivial — programa 2 — é muito grossa. Consequentemente, há pouco espaço para sua aplicação. Por outro lado, um meta-interpretador de granularidade muito fina nem sempre é realmente útil, pois o refinamento também sacrifica a eficiência e, na maioria dos casos, a perda de eficiência exigida por meta-interpretadores de níveis muito finos não se justifica.

Utilizando o predicado pré-definido `clause/2` — fornecido na maioria das implementações de Prolog — é possível refinar o meta-interpretador Trivial. Este predicado recupera uma cláusula do programa consultado realizando uma busca em profundidade. Por exemplo, se a interrogação

```
?- clause(Cab,Corpo)
```

é bem sucedida, `Cab` é a cabeça da cláusula recuperada e `Corpo` unifica com o corpo da cláusula. No caso de um fato, `Corpo` unifica com o átomo `true`. No caso de regras, o corpo pode conter uma ou mais sub-metas. Se contiver uma única sub-meta, `Corpo` unificará com ela. Se contiver diversas sub-metas, Corpo unificará com a seguinte estrutura:

```
(Meta1,OutrasMetas)
```

Neste caso, `OutrasMetas` também pode ser tanto uma única sub-meta como diversas sub-metas, estruturadas da mesma maneira. Por exemplo, a execução de `clause/2`, considerando como exemplo o programa Familia, é:

```
?- clause(tem_sobrinho(X),Corpo).

Corpo = tio(X,Y),homem(Y) ->;

Corpo = irmao(X,chris)
yes

?- clause(tio(A,B),Corpo).

Corpo = irmao(A,C),pai(C,B) ->;
```

```
A = johnny
B = franklin
Corpo = true
yes
```

A seguir, é apresentado um meta-interpretador mais interessante que aquele mostrado no programa 2. Este meta-interpretador — que será denominado meta-interpretador Básico — simula o modelo computacional de programas lógicos. Ele é descrito pelas três cláusulas abaixo:

**Programa 3** *Meta-Interpretador Básico*

```
prove(true).

prove((Meta1,Meta2)) :-
  prove(Meta1),
  prove(Meta2).

prove(Meta) :-
  clause(Meta,Corpo),
  prove(Corpo).
```

Este meta-interpretador possui granularidade mais fina que o meta-interpretador Trivial — programa 2. Ele apresenta nível de redução de cláusulas, possuindo granularidade suficiente para um grande número de aplicações.

Declarativamente, o programa pode ser descrito da seguinte forma:

- a constante `true` é verdadeira;
- a conjunção `(Meta1,Meta2)` é verdadeira se `Meta1` é verdadeira e `Meta2` é verdadeira;
- a meta `Meta` é verdadeira se há uma cláusula `Meta :- Corpo` tal que `Corpo` é verdadeira.

O aspecto procedimental do programa pode ser descrito da seguinte forma:

- O fato `prove(true)`, indicando meta vazia — representado em Prolog pelo átomo `true`) —, é resolvido;
- Para resolver uma conjunção de cláusulas `(Meta1,Meta2)`, resolva `Meta1` e resolva `Meta2`;
- No caso geral, para resolver uma meta, escolha uma cláusula do programa cuja cabeça unifica com a meta e recursivamente resolva o corpo da cláusula.

O resultado da execução — com relação ao programa que vem sendo utilizado — é o mesmo do meta-interpretador Trivial. No entanto, este meta-interpretador pode ser facilmente expandido para incorporar algumas outras características, das quais serão apresentadas as de maior interesse.

## 4.3 Um Meta-interpretador para Construir a Árvore de Prova

A técnica utilizada para construir a árvore de prova é similar àquela utilizada na construção de árvores de prova em analisadores de sentenças, através da adição de argumentos extras [Marcus 86].

A relação básica é:

```
prove(Meta,Arv)
```

onde `Arv` é a árvore de prova para resolver a meta `Meta`. Neste meta-interpretador, a árvore de prova será representada pela estrutura `(Meta <== Prova)`, onde `Prova` é a conjunção dos ramos que provam a meta `Meta`.

O programa 4, que implementa prove/2, é uma extensão direta do programa 3. As quatro cláusulas correspondem exatamente às quatro cláusulas do meta-interpretador Básico para Prolog puro, acrescidas somente da variável que deve unificar com a árvore de prova da meta.

**Programa 4** *Meta-Interpretador Arv-Prova*

```
prove(true,true).
prove((Meta1,Meta2),(Arv1,Arv2)) :-
  prove(Meta1,Arv1),
  prove(Meta2,Arv2).
prove(Meta,(Meta <== Arv)) :-
  clause(Meta,Corpo),
  prove(Corpo,Arv).
```

O exemplo a seguir mostra passo a passo como o meta-interpretador atua sobre o programa Familia.

```
?- prove(tem_sobrinho(X),Arv).
```

Neste caso, apenas a terceira cláusula de `prove/2` unifica com a meta acima:

```
#1 prove(tem_sobrinho(X),(tem_sobrinho(X) <== Arv0)) :-
     clause(tem_sobrinho(X),Corpo),
#2   prove(Corpo,Arv0).
```

Na ativação de clause(tem_sobrinho(X),Corpo), ocorre a seguinte unificação:

```
Corpo = (tio(X,Y),homem(Y))
```

A seguir, há a chamada de prove((tio(X,Y),homem(Y)), que irá unificar com a segunda cláusula de prove/2:

```
#2 prove((tio(X,Y),homem(Y)),(Arv1,Arv2)) :-
#3    prove(tio(X,Y),Arv1),
#4    prove(homem(Y),Arv2).
```

Uma vez mais, prove(tio(X,Y),Arv1) unifica com a terceira cláusula de prove/2.

```
#3 prove(tio(X,Y),(tio(X,Y) <== Arv3)) :-
      clause(tio(X,Y),CorpoB),
#5    prove(CorpoB,Arv3).
```

Na ativação de clause(tio(X,Y),CorpoB) tem-se:

```
CorpoB = (irmao(X,Z),pai(Z,Y))
```

Na ativação de prove/2, a unificação será feita com a segunda cláusula:

```
#5 prove((irmao(X,Z),pai(Z,Y)),(Arv4,Arv5)) :-
#6    prove(irmao(X,Z),Arv4),
#7    prove(pai(Z,Y),Arv5).
```

O próximo passo é a ativação de prove(irmao(X,Z),Arv4), que é feito utilizando-se a terceira cláusula de prove/2:

```
#6 prove(irmao(X,Z),(irmao(X,Z) <== Arv6)) :-
      clause(irmao(X,Z),CorpoD),
#8    prove(CorpoD,Arv6).
```

A ativação de clause/2 proporciona a seguinte unificação:

```
clause(irmao(X,Z),CorpoD)
X = clark
Z = bruce
CorpoD = true
```

Desta vez, a ativação de prove/2 irá unificar com a primeira cláusula:

```
#8 prove(true,Arv6).
Arv6 = true
```

Com isto, chega-se ao final da execução do ramo #6 da árvore de prova, com as seguintes unificações:

```
X = clark
Z = bruce
Arv4 = irmao(clark,bruce) <== true
```

O próximo passo é solucionar o ramo #7 da árvore de prova, que começa com a chamada de `prove(pai(Z,Y),Arv5)`. Neste caso, o valor de Z encontrado no ramo #6 é passado para o ramo #7 na ativação de `prove/2`. A unificação será bem sucedida com a terceira cláusula:

```
#7 prove(pai(bruce,Y),(pai(bruce,Y) <== Arv7)) :-
     clause(pai(bruce,Y),CorpoF),
#9   prove(CorpoF,Arv7).
```

A ativação de `clause/2` ocasiona a unificação abaixo:

```
Y = cinza
CorpoF = true
```

A ativação de `prove/2` no ramo #9 utilizará a primeira cláusula:

```
#9 prove(true,Arv7)
Arv7 = true
```

Isto resolve o ramo #7 da árvore de prova, envolvendo as seguintes unificações:

```
Y = dick
Arv5 = pai(bruce,dick) <== true
```

Como os ramos #6 e #7 já estão resolvidos, o ramo #5 também está, com as seguintes unificações:

```
CorpoB = (irmao(clark,bruce),pai(bruce,dick))
Arv3 = (irmao(clark,bruce) <== true , pai(bruce,dick) <== true)
```

A resolução do ramo #5 implica na resolução do ramo #3, com as unificaçes abaixo:

```
X = clark
Y = dick
Arv1 = tio(clark,dick) <== (irmao(clark,bruce) <== true ,
pai(bruce, dick) <== true)
```

O próximo passo é ativar o procedimento `prove(homem(dick),Arv2)`, que utiliza a terceira cláusula de `prove/2`.

```
#4 prove(homem(dick),(homem(dick) <== Arv8)) :-
     clause(homem(dick),CorpoC),
     prove(CorpoC,Arv8).
```

Na ativação de `clause(homem(dick),CorpoC)` haverá uma unificação com a terceira cláusula do procedimento `homem/1`, resultando `CorpoC = true`.

A chamada de `prove(true,Arv8)` utiliza-se da primeira cláusula para fazer `Arv8 = true`. Isto encerra o ramo `#4` da árvore com as seguintes unificações:

```
Y = dick
Arv2 = homem(dick) <== true
```

Isto conclui também o ramo de execução `#1` e `#2`, com as seguintes unificações:

```
X = clark
Corpo = (tio(clark,dick),homem(dick))
Arv0 = (tio(clark,dick) <== (irmao(clark,bruce) <== true ,
pai(bruce, dick) <== true) , homem(dick) <== true)
```

Finalmente, chegando à raiz da árvore, a solução para `prove(tem_sobrinho(X),Arv)` é:

```
X = clark
Arv = tem_sobrinho(clark) <== (tio(clark,dick) <== (irmao(clark,bruce)
      <== true , pai(bruce, dick) <== true) , homem(dick) <== true) ->;
```

Seguindo o mesmo procedimento, o sistema pode apresentar outras duas soluções:

```
X = johnny
Arv = tem_sobrinho(johnny) <== (tio(johnny,franklin) <== true) ,
homem(franklin) <== true) ->;

X = xavier
Arv = tem_sobrinho(xavier) <== (irmao(xavier,chris) <== true) ->;
no
```

## 4.4 Um Meta-interpretador para Rastrear a Execução de Programas

Outra extensão que pode ser feita no meta-interpretador Básico é rastrear a execução de um programa. Neste caso, um parâmetro extra é inserido para indicar a altura da árvore de prova. O valor inicial, que corresponde à altura da raiz na árvore de prova, é zero.

**Programa 5** *Meta-interpretador Rastro*

```
prove(Meta) :- prove(Meta,0).

prove(true,_).

prove((Meta1,Meta2),Comp) :-
  prove(Meta1,Comp),
  prove(Meta2,Comp).

prove(Meta,Comp) :-
  clause(Meta,Corpo),
  mostre(Meta,Comp),
  Comp1 is Comp + 1,
  prove(Corpo,Comp1).

mostre(Meta,Comp) :-
  tab(Comp),
  write(Meta),
  nl.
```

Um exemplo de execução, utilizando a Base de Conhecimento Korez é apresentado a seguir:

```
?- prove(tem_sobrinho(X)).

tem_sobrinho(X)
 tio(X,Y)
  irmao(clark,bruce)
  pai(bruce,dick)
 homem(dick)

X = clark ->;

  irmao(alex,scott)
  pai(scott,rachel)
  irmao(xavier,chris)
  irmao(rick,dick)
 tio(johnny,franklin)
 homem(franklin)

X = johnny ->;

tem_sobrinho(X)
```

```
irmao(xavier,chris)

X = xavier ->;
no
```

Neste meta-programa, o nível de cada nó na árvore de prova é utilizado para imprimir com um certo espaçamento — procedimento **mostre/2** — a última meta provada. Entretanto, ele pode ser também utilizado para suspender a prova nesse ramo da árvore, se o nível da meta em um determinado ponto da execução for maior que um certo valor limite, como é mostrado a seguir.

**Programa 6** *Meta-interpretador Rastro-Corte*

```
prove(Meta,Limite) :- prove(Meta,0,Limite).

prove(true,_,_).

prove(Meta,Limite,Limite) :- !.

prove((Meta1,Meta2),Comp,Limite) :-
  prove(Meta1,Comp,Limite),
  prove(Meta2,Comp,Limite).

prove(Meta,Comp,Limite) :-
  clause(Meta,Corpo),
  mostre(Meta,Comp),
  Comp1 is Comp + 1,
  prove(Corpo,Comp1,Limite).

mostre(Meta,Comp) :-
  tab(Comp),
  write(Meta),
  nl.
```

## 4.5 Árvore de Rastreamento

Outra característica muito interessante para ser incorporada a um meta-interpretador é manter uma árvore que carrega informações sobre como a meta está sendo resolvida. Neste caso, um parâmento extra é inserido tal que, sempre que o procedimento **prove/2** é ativado, o segundo argumento contém as informações sobre os passos que foram executados até chegar ao estado corrente de **prove/2**. Essa árvore é de grande importância na implementação do Mecanismo de Explicação em Sistemas Especialistas, que é a parte do sistema responsável por mostrar ao usuário como se chegou a uma determinada conclusão, bem como explicar porque está a procura de uma determinada

informação, quando o sistema solicita informações externas adicionais [Rodrigues 89] [Rodrigues 90b]. Segue-se o meta-programa que implementa a árvore de rastreamento.

**Programa 7** *Meta-interpretador Arv-Info*

```
prove(true,Arvore).

prove((Meta1,Meta2),Arvore) :-
  prove(Meta1,Arvore),
  prove(Meta2,Arvore).

prove(Meta,Arvore) :-
  clause(Meta,Corpo),
  prove(Corpo,[Meta <== Corpo | Arvore]).
```

Neste caso, o processo de construção da árvore é o processo oposto ao utilizado pelo meta-interpretador Arv-Prova — programa 4 —, no qual as submetas recebem uma variável livre e devolvem-na unificada com os passos que foram percorridos para provar essa sub-meta. Já no caso do meta-interpretador Arv-Info, a meta envia para as submetas uma árvore contendo os passos utilizados até o momento.

Para mostrar a execução do meta-interpretador Arv-Info com relação ao programa Familia, será feita uma pequena alteração no meta-interpretador Arv-Info, acrescentando um comando de impressão à primeira e à terceira cláusula de **prove/2**:

```
prove(true,Arvore) :-
  write(Arvore),nl.

prove((Meta1,Meta2),Arvore) :-
  prove(Meta1,Arvore),
  prove(Meta2,Arvore).

prove(Meta,Arvore) :-
  write(Arvore),nl,
  clause(Meta,Corpo),
  prove(Corpo,[Meta <== Corpo | Arvore]).
```

A execução de **tem_sobrinho(X)** com o novo meta-interpretador resulta[1]:

```
?- prove(tem_sobrinho(X),[]).

[tem_sobrinho(X) <== (tio(X,Y) , homem(Y))]
```

[1]Quando necessário, as respostas às interrogações Prolog foram editadas visando uma melhor visualização

```
[tio(X,Y) <== (irmao(X,Z) , pai(Z,Y)),tem_sobrinho(X) <== (tio(X,Y) ,
 homem(Y))]

[irmao(clark,bruce) <== true,tio(clark,Y) <== (irmao(clark,bruce) ,
 pai(bruce,Y)),tem_sobrinho(clark) <== (tio(clark,Y) , homem(Y))]

[tio(clark,Y) <== (irmao(clark,bruce) , pai(bruce,Y)),
 tem_sobrinho(clark) <== (tio(clark,Y) , homem(Y))]

[pai(bruce,dick) <== true,tio(clark,dick) <== (irmao(clark,bruce) ,
 pai(bruce,dick)),tem_sobrinho(clark) <== (tio(clark,dick) ,
 homem(dick))]

[tem_sobrinho(clark) <== (tio(clark,dick) , homem(dick))]

[homem(dick) <== true,tem_sobrinho(clark) <== (tio(clark,dick) ,
 homem(dick))]

X = clark ->;

[homem(dick) <== true,tem_sobrinho(clark) <== (tio(clark,dick) ,
 homem(dick))]

[pai(bruce,dick) <== true,tio(clark,dick) <== (irmao(clark,bruce) ,
 pai(bruce,dick)),tem_sobrinho(clark) <== (tio(clark,dick) ,
 homem(dick))]

[irmao(clark,bruce) <== true,tio(clark,Y) <== (irmao(clark,bruce) ,
 pai(bruce,Y)),tem_sobrinho(clark) <== (tio(clark,Y) , homem(Y))]

[irmao(alex,scott) <== true,tio(alex,Y) <== (irmao(alex,scott) ,
 pai(scott,Y)),tem_sobrinho(alex) <== (tio(alex,Y) , homem(Y))]

[tio(alex,Y) <== (irmao(alex,scott) , pai(scott,Y)),tem_sobrinho(alex)
 <== (tio(alex,Y) , homem(Y))]

[pai(scott,rachel) <== true,tio(alex,rachel) <== (irmao(alex,scott) ,
 pai(scott,rachel)),tem_sobrinho(alex) <== (tio(alex,rachel) ,
 homem(rachel))]

[tem_sobrinho(alex) <== (tio(alex,rachel) , homem(rachel))]

[pai(scott,rachel) <== true,tio(alex,rachel) <== (irmao(alex,scott) ,
 pai(scott,rachel)),tem_sobrinho(alex) <== (tio(alex,rachel) ,
 homem(rachel))]
```

```
[irmao(alex,scott) <== true,tio(alex,Y) <== (irmao(alex,scott) ,
 pai(scott,Y)),tem_sobrinho(alex) <== (tio(alex,Y) , homem(Y))]

[irmao(xavier,chris) <== true,tio(xavier,Y) <== (irmao(xavier,chris) ,
 pai(chris,Y)),tem_sobrinho(xavier) <== (tio(xavier,Y) , homem(Y))]

[tio(xavier,Y) <== (irmao(xavier,chris) , pai(chris,Y)),
 tem_sobrinho(xavier) <== (tio(xavier,Y) , homem(Y))]

[irmao(xavier,chris) <== true,tio(xavier,Y) <== (irmao(xavier,chris) ,
 pai(chris,Y)),tem_sobrinho(xavier) <== (tio(xavier,Y) , homem(Y))]

[irmao(rick,dick) <== true,tio(rick,Y) <== (irmao(rick,dick) ,
 pai(dick,Y)),tem_sobrinho(rick) <== (tio(rick,Y) , homem(Y))]

[tio(rick,Y) <== (irmao(rick,dick) , pai(dick,Y)),tem_sobrinho(rick)
 <== (tio(rick,Y) , homem(Y))]

[irmao(rick,dick) <== true,tio(rick,Y) <== (irmao(rick,dick) ,
 pai(dick,Y)),tem_sobrinho(rick) <== (tio(rick,Y) , homem(Y))]

[tio(X,Y) <== (irmao(X,Z) , pai(Z,Y)),tem_sobrinho(X) <== (tio(X,Y) ,
 homem(Y))]

[tio(johnny,franklin) <== true,tem_sobrinho(johnny) <== (tio(johnny,
 franklin) , homem(franklin))]

[tem_sobrinho(johnny) <== (tio(johnny,franklin) , homem(franklin))]

[homem(franklin) <== true,tem_sobrinho(johnny) <== (tio(johnny,
 franklin) , homem(franklin))]

X = johnny ->;

[homem(franklin) <== true,tem_sobrinho(johnny) <== (tio(johnny,
 franklin) , homem(franklin))]

[tio(johnny,franklin) <== true,tem_sobrinho(johnny) <== (tio(johnny,
 franklin) , homem(franklin))]

[tem_sobrinho(X) <== (tio(X,Y) , homem(Y))]

[tem_sobrinho(X) <== irmao(X,chris)]

[irmao(xavier,chris) <== true,tem_sobrinho(xavier) <== irmao(xavier,
 chris)]
```

```
X = xavier ->;

[irmao(xavier,chris) <== true,tem_sobrinho(xavier) <== irmao(xavier,
 chris)]
no
```

## 4.6 Raciocínio Incerto

É frequente, nos problemas que manipulam conhecimento, os casos em que a resposta final não é conhecida com certeza absoluta. Tanto as regras do especialista podem ser vagas como o usuário pode não ter certeza ao responder algumas questões. Isto pode ser visto facilmente em sistemas de diagnóstico médico, onde os especialistas nem sempre são capazes de definir a relação entre sintomas e doenças com precisão absoluta. De fato, os médicos podem apontar muitos diagnósticos possíveis para um dado conjunto de sintomas.

Deste modo, torna-se muito importante na construção de Sistemas Especialistas a existência de um mecanismo de raciocínio incerto. Um método frequentemente utilizado para implementar raciocínio incerto é associar um fator de certeza a cada regra na Base de Conhecimento. O fator de certeza indica a certeza do especialista quanto à veracidade de uma determinada regra.

Há vários formalismos para tratar estes fatores de certeza [Monard 89a], mas todos eles têm sido questionados [Pearl 86], [White 85]. Portanto, qualquer método escolhido será aberto a críticas.

Este questionamento pode ser justificado por várias razões. Em primeiro lugar, os Sistemas Baseados em Conhecimento necessitam de um modelo de raciocínio; porém, não se sabe com absoluta certeza como o ser humano pensa e raciocina, pelo menos até o momento. Um segundo ponto que pode ser destacado são os métodos para tratamento de incerteza que vêm sendo utilizados, pois, embora estejam baseados em teorias supostamente corretas, algumas suposições devem ser aceitas para que se possa realizar uma implementação do método que evite a explosão combinatória.

Entre as muitas propostas existentes, o mecanismo escolhido neste trabalho para representar raciocínio incerto é semelhante ao usado em MYCIN [Shortliffe 76], mas não é idêntico. O mecanismo associa a cada regra ou fato um fator de certeza entre 0 e 1.

Um programa lógico com incerteza é um conjunto de pares

```
<Clausula,Fator>,
```

onde `Fator` é o fator de certeza da cláusula `Clausula`. Neste caso, a regra utilizada para realizar o cálculo de incertezas é explicado a seguir.

Considerando que uma cláusula do programa é `<(A :- B) , F>`, o grau de certeza é calculado da seguinte forma:

```
certeza(A) = F * certeza(B)
certeza((M1,M2)) = Min{certeza(M1),certeza(M2)}
```

O meta-interpretador Fator-Certeza é uma extensão do meta-interpretador básico que incorpora um mecanismo para cálculo de incerteza utilizando a regra acima.

**Programa 8** *Meta-interpretador Fator-Certeza*

```
prove(true,1).
prove((Meta1,Meta2),FC) :-
 prove(Meta1,FC1),
 prove(Meta2,FC2),
 minimo(FC1,FC2,FC).
prove(Meta,FC) :-
clause_fc(Meta,Corpo,Fator),
 prove(Corpo,FC1),
 FC is Fator * FC1.
```

O procedimento `minimo(C1,C2,C)` unifica `C` com o menor valor entre `C1` e `C2`:

```
minimo(C1,C2,C1) :- C2 >= C1.
minimo(C1,C2,C2) :- C1 > C2.
```

O procedimento `clause_fc/3` tem função semelhante ao predicado pré-definido de Prolog `clause/2`, somente que, além de unificar com o corpo de uma meta, ele unifica também com o fator de certeza associado a essa regra. Não há grandes dificuldades para a implementação de `clause_cf/3`, pois ela depende apenas da maneira escolhida para representar o fator de certeza das regras. Como exemplo, será considerado o seguinte método:

A representação da regra

```
r(R,S) :- r1(R),r2(S).
```

com fator de certeza FC é

```
r(R,S) :- fc = FC,r1(R),r2(S).
```

A representação do fato

```
f(estrela).
```

com fator de certeza F é

```
f(estrela) :- fc = F.
```

Neste caso, a implementação de `clause_cf/3` seria:

```
clause_cf(Meta,Corpo,Fator) :- clause(Meta,(fc = Fator,Corpo)).
clause_cf(Meta,true,Fator) :- clause(Meta,fc = Fator).
```

Para mostrar como o meta-interpretador Fator-Certeza atua sobre um programa Prolog, será utilizada uma pequena Base de Conhecimento que não possui nenhuma informação contextual em suas regras. Nessa Base, as regras e os fatos são associados a fatores de certeza, como mostrado a seguir.

```
a(X) :- fc = 0.5,b(X,Y),c(Y).
a(X) :- fc = 0.3,d(X,azul).

b(A,B) :- fc = 0.8,d(B,C),f(C,A).
b(preto,branco) :- fc = 0.9.

c(verde) :- fc = 0.6.
c(branco) :- fc = 0.8.
c(laranja) :- fc = 0.5.

d(branco,vermelho) :- fc = 0.87.d(verde,azul) :- fc = 1.
d(vinho,marrom) :- fc = 0.5.
d(cinza,rosa) :- fc = 0.35.

f(vermelho,cinza) :- fc = 0.4.
f(marrom,branco) :- fc = 0.65.
f(rosa,verde) :- fc = 0.9.
```

O resultado da execução do meta-interpretador Fator-Certeza com relação à Base de Conhecimento acima é:

```
?- prove(a(X),FC).

X = cinza
FC = 0.16 ->;

X = preto
FC = 0.4 ->;

X = verde
FC = 0.3 ->;
no
```

## 4.7 Interação com o Usuário

Muitos Sistemas Especialistas devem incorporar mecanismos de interação com o usuário durante o processo de prova de uma meta. Neste caso, quando uma meta não pode ser

provada por nenhuma das cláusulas do meta-interpretador, uma solução é perguntar ao usuário a veracidade ou falsidade da meta. Na implementação mostrada a seguir, a meta será considerada provada se o usuário responder sim, falhando caso contrário. A cláusula **prove/1** abaixo pode ser incorporada ao final do meta-interpretador Básico — Programa 3 — para que possa ser realizada a interação com o usuário:

```
prove(Meta) :-
  pergunte(Meta,Resp),
  verdade(Resp).

pergunte(Meta,Resp) :-
  faz_pergunta(Meta),
  read(Resp).
faz_pergunta(Meta) :-
  write(Meta),
  write(' ? ').

verdade(sim).
```

O mecanismo de interação acima ainda apresenta algumas deficiências. A primeira delas é a repetição da mesma pergunta ao usuário sempre que for necessário repetir a prova de uma meta. A solução para isso é guardar a resposta do usuário para que ela possa ser lembrada. Neste caso, o meta-interpretador deve verificar, antes de perguntar ao usuário, se a pergunta já foi feita anteriormente. Segue-se o meta-interpretador que implementa esta solução.

```
prove(Meta) :-
  ja_perguntado(Meta,Resp),
  verdade(Resp).

prove(Meta) :-
  not ja_perguntado(Meta,_),
  pergunte(Meta,Resp),
  lembre_resposta(Meta,Resp),
  verdade(Resp).

lembre_resposta(Meta,Resp) :-
  assert(ja_perguntado(Meta,Resp)).
```

Uma outra falha apresentada pelo mecanismo de interação descrito acima é que nem sempre uma meta deve ser perguntada ao usuário, pois pode ser que algumas metas, no contexto da prova, devam realmente falhar. Isto é facilmente sanado pela relação **perguntavel/1**, que indica se uma meta deve ou não ser perguntada ao usuário. O Programa 9 contém as cláusulas que compõem o meta-interpretador Interativo.

**Programa 9** *Meta-interpretador Interativo*

```
prove(true).

prove(Meta) :-
  ja_perguntado(Meta,Resp),
  verdade(Resp).

prove((Meta1,Meta2)) :-
  prove(Meta1),
  prove(Meta2).

prove(Meta) :-
  clause(Meta,Corpo),
  prove(Corpo).

prove(Meta) :-
  perguntavel(Meta),
  not ja_perguntado(Meta,_),
  pergunte(Meta,Resp),
  lembre_resposta(Meta,Resp),
  verdade(Resp).
```

Para mostrar a execução do meta-interpretador Interativo, será feita uma pequena alteração na Base de Conhecimento original — Programa 1 — para transformar os fatos `homem/1` em relações perguntáveis.

O programa a seguir é o programa Familia com as alterações mencionadas acima.

```
tem_sobrinho(X) :- tio(X,Y),homem(Y).
tem_sobrinho(X) :- irmao(X,chris).

tio(A,B) :- irmao(A,C),pai(C,B).
tio(johnny,franklin).

pai(scott,rachel).
pai(scott,nathan).
pai(david,bruce).
pai(bruce,dick).

irmao(clark,bruce).
irmao(alex,scott).
irmao(xavier,chris).
irmao(rick,dick).

perguntavel(homem(X)).
```

Abaixo são mostradas duas interrogações utilizando este meta-interpretador, considerando as respostas dadas pelo usuário, distintas em cada uma delas. Pode-se observar que, neste caso particular, a resposta do sistema é diferente em cada caso.

- Exemplo 1:

```
?- prove(tem_sobrinho(X)).

homem(dick)? sim.

X = clark ->;

homem(rachel)? nao.
homem(nathan)? sim.

X = alex ->;

homem(franklin)? sim.

X = johnny ->;

X = xavier ->;
no
```

- Exemplo 2:

```
?- prove(tem_sobrinho(X)).

homem(dick)? nao.
homem(rachel)? sim.

X = alex;

homem(nathan)? sim.

X = alex ->;

homem(franklin)? nao.

X = xavier ->;
no
```

Deve ser observado que, após a execução, as respostas são lembradas na Base de Dados. Portanto, se a meta prove/2 é ativada uma segunda vez, nenhuma pergunta será feita ao usuário, obtendo-se a mesma solução. Deste modo, para a obtenção de um resultado

correto de uma nova prova, independente de respostas anteriores, é necessário eliminar da base todas as cláusulas `ja_perguntado(Meta,Resp)`. O procedimento pré-definido `abolish(Meta/Aridade)` deve ser utilizado para essa finalidade.

## 4.8 Relações do Sistema

Muitas vezes, é interessante que um meta-interpretador manipule certas características da linguagem que não pertencem ao Prolog puro, como, por exemplo, os vários predicados pré-definidos do sistema. Estes predicados não são definidos por cláusulas no programa e, portanto, precisam ser tratados de forma diferente, pois eles devem ser executados diretamente pelo Prolog.

Em Prolog, o predicado pré-definido **system(P/N)** determina se um predicado P com N argumentos é ou não um predicado pré-definido. O procedimento **sistema/1** abaixo — que utiliza **system/2** — é bem sucedido quando um predicado é pré-definido do sistema.

```
sistema(Predicado) :-
  functor(Predicado,Nome,Aridade),
  system(Nome/Aridade).
```

O predicado **functor/3** é pré-definido de Prolog.Ele quebra uma estrutura, retornando seu nome e aridade. Por exemplo:

```
?- functor(jogo(bola,sapato,mel),Nome,Aridade).

Nome = jogo
Aridade = 3
```

A cláusula a seguir incorpora os predicados de sistema ao meta-interpretador Básico apresentado no programa 3:

```
prove(Meta) :- sistema(Meta), Meta.
```

Esta cláusula extra faz com que os predicados do sistema sejam invisíveis ao meta-interpretador **prove/1**. Porém, há alguns predicados fornecidos pelo sistema que deveriam ser visíveis, como por exemplo o predicado **not** — negação por falha de Prolog. Para melhor manipular esses predicados, cada um deles deverá ter uma cláusula especial no meta-interpretador, tornando mais fina sua granularidade. No caso do predicado not, a implementação seria:

```
prove(not Meta) :- not prove(Meta).
```

O programa a seguir possui o predicado pré-definido **write/2** e a negação **not**. Este programa é uma pequena Base de Conhecimento para receitar medicamentos, e uma Base de Dados de pacientes [2] para exemplificá-la.

**Programa 10** *Programa Farmacia*

```
receita(Paciente) :-
   apresenta(Paciente,Sintoma),
   combate(Remedio,Sintoma),
   not contra_indicado(Paciente,Remedio),
   write('Deve tomar '),write(Remedio),nl.

contra_indicado(Paciente,Remedio) :-
   agrava(Remedio,Condicao),
   sofre(Paciente,Condicao).

combate(aspirina,gripe).
combate(lomotil,diarreia).
combate(coristina,gripe).
combate(floratil,diarreia).

agrava(aspirina,ulcera).
agrava(lomotil,figado).

apresenta(diamantino,diarreia).
apresenta(geraldo,gripe).

sofre(diamantino,figado).
```

O meta-interpretador a seguir é o meta-interpretador Arv-Prova extendido para reconhecer funções do sistema.

**Programa 11** *Meta-interpretador Arv-Sistema*

```
prove(true,true).

prove((Meta1,Meta2),(Arv1,Arv2)) :-
  prove(Meta1,Arv1),
  prove(Meta2,Arv2).

prove(Meta,sistema(Meta)) :-
  sistema(Meta),
  Meta.
```

[2] A Base de Dados é formada pelas relações **apresenta/2** e **sofre/2**

```
prove(Meta,(Meta <== Arv)) :-
  clause(Meta,Corpo),
  prove(Corpo,Arv).
```

Segue-se o resultado da execução do meta-interpretador acima com relação ao programa Farmacia.

```
?- prove(receita(diamantino),Arv).

Deve tomar floratil

Arv = receita(diamantino) <== (apresenta(diamantino,diarreia) <== true,
      combate(floratil,diarreia) <== true, sistema(not contra_indicado(
      diamantino,floratil)) , sistema(write(Deve tomar )) , sistema(
      write(floratil)) , sistema(nl))
```

Como já foi mencionado anteriormente, pode-se desejar que alguns predicados do sistema não sejam invisíveis ao meta-interpretador. Para que isto possa ser exemplificado, a cláusula abaixo é adicionada como terceira cláusula no meta-interpretador Arv-Sistema. Neste caso, a árvore de prova será diferente da anterior.

```
prove(not Meta , nao(Meta)) :- not prove(Meta,Arv).

?- prove(receita(diamantino),Arv).

Deve tomar floratil

Arv = receita(diamantino) <== (apresenta(diamantino,diarreia) <== true,
      combate(floratil,diarreia) <== true , nao(contra_indicado(
      diamantino,floratil)) , sistema(write(Deve tomar )) , sistema(
      write(floratil)) , sistema(nl))
```

## 4.9 Procedimentos Invisíveis

Da mesma maneira em que há interesse em tornar alguns predicados do sistema visíveis ao meta-interpretador, pode ser também necessário tornar invisíveis alguns predicados, mesmo que estes não sejam predicados do sistema. Neste caso, pode ser utilizado um procedimento análogo ao utilizado na definição de relações do sistema. Isto pode ser feito definindo-se um predicado para indicar todas as relações que serão invisíveis ao meta-interpretador. No exemplo a seguir, `primitiva(Meta)` define `Meta` como uma relação invisível ao meta-interpretador.

```
prove(Meta) :- primitiva(Meta), Meta.
```

As relações invisíveis ao meta-interpretador são executadas diretamente pelo motor de inferência de Prolog.

O meta-interpretador Cristalino abaixo é uma extensão do meta-interpretador Arv-Prova com as seguintes características adicionais: relações do sistema, negação de metas e relações primitivas — invisíveis ao meta-interpretador.

**Programa 12** *Meta-interpretador Cristalino*

```
prove(true,true).

prove((Meta1,Meta2),(Arv1,Arv2)) :-
  prove(Meta1,Arv1),
  prove(Meta2,Arv2).

prove(Meta,sistema(Meta)) :-
  sistema(Meta),
  Meta.

prove(Meta,primitiva(Meta)) :-
  primitiva(Meta),
  Meta.

prove(not Meta,nao(Meta)) :- not prove(Meta,Arv).

prove(Meta,(Meta <== Arv)) :-
  clause(Meta,Corpo),
  prove(Corpo,Arv).
```

Se a relação

```
primitiva(b(Arg1,Arg2))
```

for adicionada ao programa Familia — página 3 — e o meta-interpretador Cristalino for interrogado com relação a essa base, a resposta é:

```
?- prove(tem_sobrinho(X),Arv).

X = clark
Arv = tem_sobrinho(clark) <== (primitiva(tio(clark,dick)) ,
      homem(dick) <== true)
```

```
X = johnny
Arv = tem_sobrinho(johnny) <== (primitiva(tio(johnny,franklin)) ,
      homem(franklin) <== true)

X = xavier
Arv = tem_sobrinho(xavier) <== (irmao(xavier,chris) <== true)
```

Pode-se observar que os detalhes da execução de `tio(X,Y)` foram omitidos na árvore de prova, ou seja, os detalhes da prova de `tio(X,Y)` tornaram-se invisíveis ao meta-interpretador.

# 5 Meta-Interpretadores e Sistemas Especialistas

Uma das principais utilidades de meta-interpretadores é na construção de Sistemas Especialistas.

Geralmente, um Sistema Especialista é decomposto em Motor de Inferência e Base de Conhecimento. Algumas Bases de Conhecimento podem ser escritas diretamente em Prolog, pois a linguagem Prolog possui seu próprio Motor de Inferência. Neste caso, tem-se uma Base de Conhecimento executável.

Entretanto, existem diversas características importantes na implementação de Sistemas Especialistas que não são fornecidas pela linguagem Prolog, tais como mecanismo de explicação ou raciocíno com incerteza [Clocksin 87] [Merrit 89] [Monard 89a], bem como raciocínio não monotônico [Monard 86] [Monard 87].

Uma possível solução para manipular raciocínio com incerteza é incluir estas características diretamente à Base de Conhecimento. Neste caso, alguns argumentos extras são adicionados a cada predicado para armazenar a informação necessária, tal como grau de certeza ou árvore de prova [Monard 89c].

Essa solução mantém a eficiência da execução, pois a Base de Conhecimento continua sendo uma Base de Conhecimento executável. No entanto, a modularidade e a clareza da Base de Conhecimento são sacrificadas, pois um programa que é puramente declarativo é modificado e passa a conter detalhes procedimentais, como por exemplo o cálculo do fator de certeza, bem como argumentos adicionais que, em geral, tornam as regras um tanto confusas [Walker 87].

Um método alternativo é baseado em meta-programação [Rodrigues 90c]. Neste caso, a Base de Conhecimento é mantida intacta e as características adicionais são incorporadas a meta-interpretadores. Este método mantém a clareza e modularidade da Base de Conhecimento, facilitando sua implementação e a manutenção do sistema.

# 6 Conclusões

Antigamente, as grandes preocupações dos implementadores eram:

- economia no armazenamento em memória primária
- otimização na execução dos programas.

Com o desenvolvimento tecnológico, as empresas podem agora dispor de equipamentos mais rápidos e com grande quantidade de memória. Deste modo, a preocupação maior nos dias de hoje é rapidez no desenvolvimento dos sistemas e facilidade na manutenção.

Por estes motivos, dá-se muita importância à expressividade das linguagens, pois quanto maior o poder de expressão da linguagem, mais fácil será a tarefa do programador.

A linguagem de programação lógica Prolog tem um considerável poder de expressão. Além disso, ela apresenta facilidades para a utilização de meta-programação, que pode ser utilizada para aumentar ainda mais a expressividade da linguagem.

Neste trabalho foram apresentados diversos meta-interpretadores e a implementação de cada um deles. A apresentação foi feita em ordem crescente de complexidade, de modo a facilitar o entendimento e capacitar ao leitor construir seu próprio meta-interpretador, segundo suas necessidades.

Deve-se ressaltar, no entanto, que a utilização de meta-interpretadores diminui a eficiência dos sistemas desenvolvidos, devido à fase de análise feita pelo interpretador. No entanto, a técnica da Avaliação Parcial[Franco 92] [Franco 91] pode ser utilizada para recuperar a eficiência original após a utilização de meta-interpretadores. Deste modo, pode-se desenvolver sistemas de forma modular e rapidamente e, quando o sistema está pronto para ser implantado, utiliza-se a Avaliação Parcial para gerar um código eficiente.

# Referências


[Amble 87] AMBLE, T. *Logic Programming and Knowledge Engineering.* Addison-Wesley, 1987.

[Bowen 82] BOWEN, K.; KOWALSKI R. *Amalgamating Language and Metalanguage in Logic Programming.* Logic Programming Clark,K. and Tärnlund S. (Eds) ; Academic Press, 1982.

[Bratko 90] BRATKO, I. *Prolog Programming for A.I.* Addison-Wesley, 2a. Ed., 1990.

[Clocksin 87] CLOCKSIN, W.F.; MELLISH, C.S. *Programming in Prolog.* Springer-Verlag, 3a. Ed. , Germany, 1987.

[Franco 91] FRANCO, J. L. *Sobre Avaliação Parcial de Meta-Interpretadores e Geração de Núcleos Específicos de Sistemas Especialistas.* Tese de Mestrado, ICMSC-USP, 1991

[Franco 92] FRANCO, J. L.; MONARD, M.C. *Avaliação Parcial de Programas Prolog e sua Utilização na Meta-Programação.* ILTC, 1992.

[Furukawa 89] FURUKAWA, K.;FUJITA, H. *Deriving an Efficient Production System by Partial Evaluation.* ICOT Technical Report: TR-456, March 1989.

[Gallaire 82] GALLAIRE, H.; LASSERRE, C. *Metalevel Control for Logic Programs.* Logic Programming, Clark,K. and Tärnlund S. (Eds); Academic Press, 1982.

[Marcus 86] MARCUS, C. *Prolog Programming - Applications for Database Systems, Expert Systems and Natural Language Systems.* Addison-Wesley Publishing Company, 1986.

[Merrit 89] MERRIT, D. *Building Expert Systems in Prolog.* Springer-Verlag, 1989.

[Monard 86] MONARD, M. C. *Um Sistema de Representação de Conhecimento e Raciocínio por Default.* Tese de Livre Docência, ICMSC-USP, 1986.

[Monard 87] MONARD, M. C.,HEBIHARA, S. M. *Um Sistema que Utiliza Raciocínio por "Default".* Anais IV SBIA, pp 143-156, Outubro 1987.

[Monard 89a] MONARD, M. C.; PRADO, A.H.A. *Uso de Incerteza em Sistemas Baseados em Conhecimento.* ILTC, Maio 1989.

[Monard 89c] MONARD, M. C.; RODRIGUES, S. R. *Desenvolvimento de Sistemas Especialistas em Prolog.* ILTC, Julho 1989.

[Monard 90] MONARD, M. C.; FRANCO, J. L. *Uso de Programação Lógica no Desenvolvimento de Compiladores.* Relatório Técnico do ICMSC-USP, Nº8, 1990

[Pearl 86] PEARL, J. *Bayes Decision Methods.* Technical Report CSD-850023. University of California at Los Angeles, January 1986

[Rodrigues 89] RODRIGUES, S. R.; MONARD, M. C. *Um Subsistema de um Núcleo de Sistemas Especialistas para Interrogar o Usuário Admitindo Diversas Categorias de Perguntas.* Anais do 6ºSimpósio Brasileiro em Inteligência Artificial, pp 78-93, 1989.

[Rodrigues 90b] RODRIGUES, S. R.; MONARD, M. C. *Implementação Lógica de um Módulo Coletor de Dados para a Construção de Sistemas Especialistas.* Notas do ICMSC-USP, Nº63, 1990.

[Rodrigues 90c] RODRIGUES, S. R.; MONARD M. C. *Uso de Meta-Interpretadores no Desenvolvimento de Núcleos de Sistemas Especialistas.* ILTC, Agosto 1990.

[Shortliffe 76] SHORTLIFFE, E.H. *Computer-Based Medical Consultation Mycin.* American Elsevier, New York, 1976.

[Sterling 84] STERLING; L. *Logical Levels of Problem Solving.* Journal of Logic Programming, Nº2, pp 151-163, 1984.

[Sterling 86] STERLING, L.; BEER, R. *Incremental Flavor-Mixing of Meta-Interpreters for Expert System Construction.* Proceedings Symposium on Logic Programming, pp 20-27, 1986.

[Sterling 87] STERLING, L. ; SHAPIRO, E. *The Art of Prolog.* The Mit Press, 1987.

[Walker 87] WALKER, A. (ED.); MCCORD, M.; SOWA, J. E.; WILSON, W. G. *Knowledge Systems and Prolog. A Logical Approach for Expert Systems and Natural Language Processing.* Addison-Wesley, 1987.

[White 85] WITHE, A.P. *Inference Deficiencies in Rule-based Expert Systems* Research and Development in Expert Systems, Cambridge University Press, pp 39-50, 1985.