Prolog

Lógica Computacional (Práticas)
2005/2006

Lic. Matemática e Ciências da Computação

Maria João Frade (**mjf@di.uminho.pt**)

Departamento de Informática
Universidade do Minho

Bibliografia

Prolog Programming for Artificial Intelligence - (2nd edition).
Ivan Bratko, Addison-Wesley, 1993.

The Art of Prolog : advanced programming techniques - (2nd edition).
L. Sterling & E. Shaphiro, MIT Press, 1994.

Essentials of Logic Programming.
Christopher John Hogger. Oxford University Press, 1990.

SICStus Prolog – *User's Manual*
http://www.sics.se/sicstus/docs/latest/html/sicstus.html/index.html

PROLOG Uma linguagem de PROgramação em LÓGica.

A linguagem Prolog surgiu no início da década de 70.

O Prolog é uma linguagem de declarativa que usa um fragmento da lógica de 1ª ordem (as *Cláusulas de Horn*) para representar o conhecimento sobre um dado problema.

Um programa em Prolog é um "conjunto" de axiomas e de regras de inferência (definindo relações entre objectos) que descrevem um dado problema. A este conjunto chama-se normalmente *base de conhecimento*.

A execução de um programa em Prolog consiste na dedução de consequências lógicas da base de conhecimento.

O utilizador coloca questões e o "motor de inferência" do Prolog pesquisa a base de conhecimento à procura de axiomas e regras que permitam (por dedução lógica) dar uma resposta. O motor de inferência faz a dedução aplicando o algoritmo de resolução de 1ª ordem.

Exemplo de um **programa** Prolog (um conjunto de *Cláusulas de Horn*).

Factos

Regras

```
mae(sofia,joao).
mae(ana,maria).
mae(carla,sofia).

pai(paulo,luis).
pai(paulo,sofia).
pai(luis,pedro).
pai(luis,maria).

progenitor(A,B) :- pai(A,B).
progenitor(A,B) :- mae(A,B).

avo(X,Y) :- progenitor(X,Z), progenitor(Z,Y).
```

Átomos

Variáveis (lógicas)

$\forall A\ \forall B.$ progenitor(A,B) $\Leftarrow$ pai(A,B)

$\forall A\ \forall B.$ progenitor(A,B) $\Leftarrow$ mae(A,B)

$\forall X\ \forall Y.$ avo(X,Y) $\Leftarrow \exists Z.$ progenitor(X,Z) $\wedge$ progenitor(Z,Y)

Cláusulas de Horn são fórmulas da forma $p \Leftarrow q1 \wedge q2 \wedge \ldots \wedge qn$

representadas em Prolog por `p :- q1, q2, ..., qn`

<cabeça da cláusula> `:-` *<corpo da cláusula>*

Notas:

Os **factos** são cláusulas de Horn com o corpo vazio.

As variáveis que aparecem nas cláusulas são quantificadas universalmente e o seu âmbito é toda a cláusula, mas podemos ver as variáveis que ocorrem apenas no corpo da cláusula (mas não na cabeça), como sendo quantificadas existencialmente dentro do corpo da cláusula.

As **questões** são cláusulas de Horn com cabeça vazia.

As questões são um meio de extrair informação de um programa. As variáveis que ocorrem nas questões são quantificadas existencialmente.

Exemplos de **questões** à base de conhecimento.

```
| ?- avo(ana,joao).
no
```

```
| ?- progenitor(luis,maria).
yes
```

```
| ?- pai(X,maria), pai(X,pedro).
X = luis ?
yes
```

```
| ?- progenitor(X,Y).
X = paulo,
Y = luis ? ;
X = paulo,
Y = sofia ? ;
X = luis,
Y = pedro ? ;
X = luis,
Y = maria ? ;
X = sofia,
Y = joao ? ;
X = ana,
Y = maria ? ;
X = carla,
Y = sofia ? ;
no
```

```
| ?- avo(paulo,X).
X = pedro ?
yes
```

```
| ?- avo(paulo,X).
X = pedro ? ;
X = maria ? ;
X = joao ? ;
no
```


Responder a uma questão é determinar se a questão é uma consequência lógica do programa.

Responder a uma questão com variáveis é dar uma instânciação da questão (representada por uma substituição para as variáveis) que é inferível do programa.

O utilizador ao digitar "`;`" força o motor de inferência a fazer *backtracking*. Ou seja, pede ao motor de inferência para construir uma prova outra prova (alternativa) para a questão que é colocada. Essa nova prova pode dar origem a uma outra substituição das variáveis.

A arte de programar em lógica está, em parte, na escolha da axiomatização mais elegante, clara e completa para os problemas.

Prolog (em resumo)

SICStus Prolog – *User's Manual*
http://www.sics.se/sicstus/docs/latest/html/sicstus.html/index.html

O interpretador

```
> sicstus
SICStus 3.11.0 (x86-linux-glibc2.3): Mon Oct 20 15:59:37 CEST 2003
Licensed to di.uminho.pt
| ?-
```

Notas: As extensão usual dos ficheiros (sicstus) prolog é `.pl`

O *ponto final* assinala o final das cláusulas.

Carregar ficheiros

```
| ?- consult(file).
| ?- consult('file.pl').
| ?- [file1, 'file2.pl', '/home/abc/ex/file3'].
```

Listar a base de conhecimento que está carregada `| ?- listing.`

Para sair do interpretador `| ?- halt.`

Termos

O **termos** são as entidades sintácticas que representam os objectos (do universo de discurso da lógica de 1ª ordem).
Os termos podem ser *constantes*, *variáveis* ou *termos compostos*.

Constantes

Inteiros

Base 10	5	0	27	-48
Outras bases (de 2 a 36)	2'101	8'175		
Código ASCII	0'A	0'z	*(65 e 122 respectivamente)*	

Reais

`2.0  -5.71  47.0E3  -0.9e-6`

Átomos

Qualquer sequência de caracteres alfanuméricos começada por letra minúscula

Qualquer sequência de `+ - * / \ ^ >< = ~ : . ? @ # $ &`

Qualquer sequência de caracteres entre `' '`

`! ; [] {}`

Variáveis

Qualquer sequência de caracteres alfanuméricos iniciada com letra maiúscula ou `_` .

Exemplos: `X  A  _  Valores  _aa  _5  RESP  _VAL`

variável anónima

Termos Compostos

São objectos estruturados da forma *f(t1,...,tn)* em que

`f` é o **functor** do termo e `t1, ..., tn` são termos *(subtermos)*.
O functor `f` tem **aridade** `n`.

O nome do functor é um átomo.

Exemplos: `suc(0)` `suc(suc(0))` `data(15,abril,2006)`

Há functores que podem ser declarados como operadores *(infixos, prefixos, posfixos)*.

Exemplo: `+(X,Y)` ⇨ `X+Y`

Listas

Listas são termos gerados à custa do átomo `[]` *(a lista vazia)*

e do functor `.(H,T)` *(o cons)*

Exemplo: O termo `.(1,.(2,.(3,[])))` representa a lista `[1,2,3]`

O Prolog tem uma notação especial para listas `[` *head* `|` *tail* `]`

Exemplo:

`.(a,.(b,.(c,[]))) = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]] = [a,b,c]`

String

Uma string é a lista de códigos ASCII dos seus caracteres.

Exemplo: `"PROLOG" = [80,82,79,76,79,71]`

Note que `"PROLOG"` é diferente de `'PROLOG'`

Programas

Um programa é um conjunto de *cláusulas de Horn*. Ou seja, fórmulas da forma

$p \Leftarrow q1 \wedge q2 \wedge \ldots \wedge qn$

representadas em Prolog por `p :- q1, q2, ..., qn`

<cabeça da cláusula> `:-` *<corpo da cláusula>*

Factos são cláusula só com cabeça e de corpo vazio.
Regras são cláusula com cabeça e corpo não vazio.

Exemplo:

comentários

```
% grafo orientado
caminho(a,b).
caminho(b,c).

/*  verifica se existe ligação entre
dois nodos dos grafo */
ligacao(X,Y) :- caminho(X,Y).
ligacao(X,Y) :- caminho(X,Z), ligacao(Z,Y).
```

Os factos e regras com o mesmo nome (à cabeça) definem um ***predicado*** (ou ***procedimento***). Neste exemplo definimos os predicados `caminho/2` e `ligacao/2` *(nome/aridade)*. Note que é possível ter predicados distintos com o mesmo nome mas aridades diferentes.

Questões são cláusulas de Horn com cabeça vazia.

As questões são um meio de extrair informação de um programa. As variáveis que ocorrem nas questões são quantificadas existencialmente.

Responder a uma questão é determinar se ela é uma consequência lógica do programa.

Exemplo:

```
| ?- ligacao(a,K).
K = b ?
yes
```

```
| ?- ligacao(a,K).
K = b ? ;
K = c ?
yes
```

```
| ?- ligacao(a,K).
K = b ? ;
K = c ? ;
no
```

Força o *backtracking*

i.e., pede para serem produzidas novas provas (alternativas)

Justifique esta resposta desenhando a *árvore de procura* de `? ligacao(a,K).`

Estratégia de prova do motor de inferência do Prolog

Assuma que o objectivo a provar (o *goal*) é: `? G1, G2, ..., Gn`

O motor de inferência pesquisa a base de conhecimento (de cima para baixo) até encontrar uma regra cuja cabeça *unifique* com `G1`. Essa unificação produz uma *substituiçao* (o *unificador mais geral*) θ

- Se `C :- P1, ..., Pm` é a regra encontrada. θ é tal que $C\theta = G1\theta$.

  O novo objectivo a provar é agora `? P1θ, ..., Pmθ, G2θ, ..., Gnθ`

- Se a regra encontrada é um facto `F`. θ é tal que $F\theta = G1\theta$.

  O novo objectivo a provar é agora `? G2θ, ..., Gnθ`

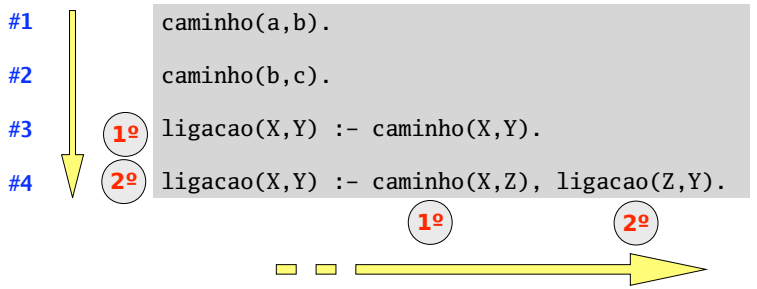
➡ A prova termina quando já não há mais nada a provar (o *goal* é vazio). O interpretador responde à questão inicial indicando a substituição a que têm que ser sujeitas as variáveis presentes na questão inicial, para produzir a prova.

Estratégia de Procura de Soluções

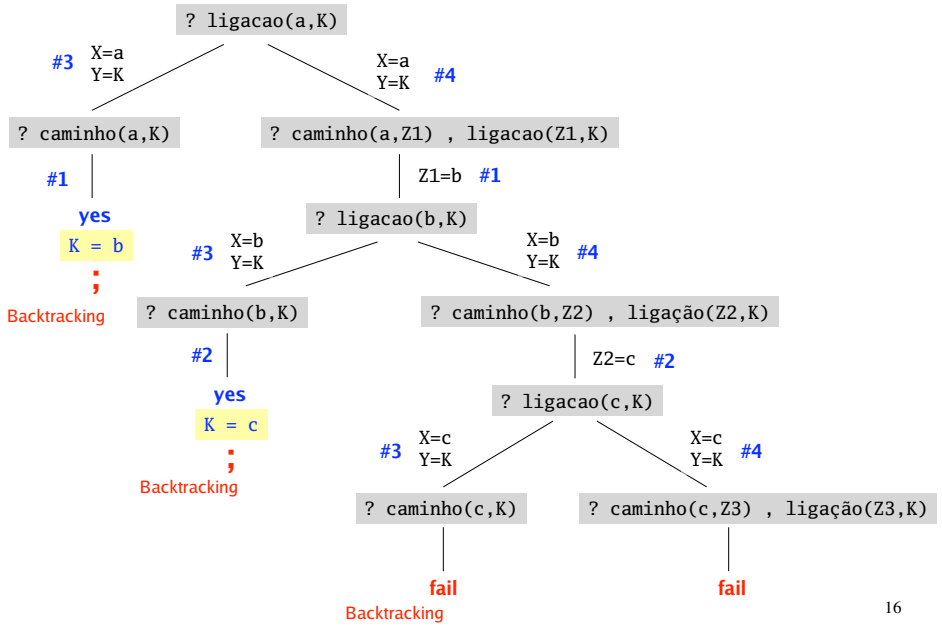
Top Down (de cima para baixo)

Depth-First (profundidade máxima antes de tentar um novo ramo)

Backtracking (volta a tentar encontrar uma prova alternativa)



Árvore de Procura



Observação:

Se o programa fosse

```
cam(a,b).
cam(b,c).

lig(X,Y) :- cam(X,Y).

lig(X,Y) :- lig(Z,Y), cam(X,Z).
```

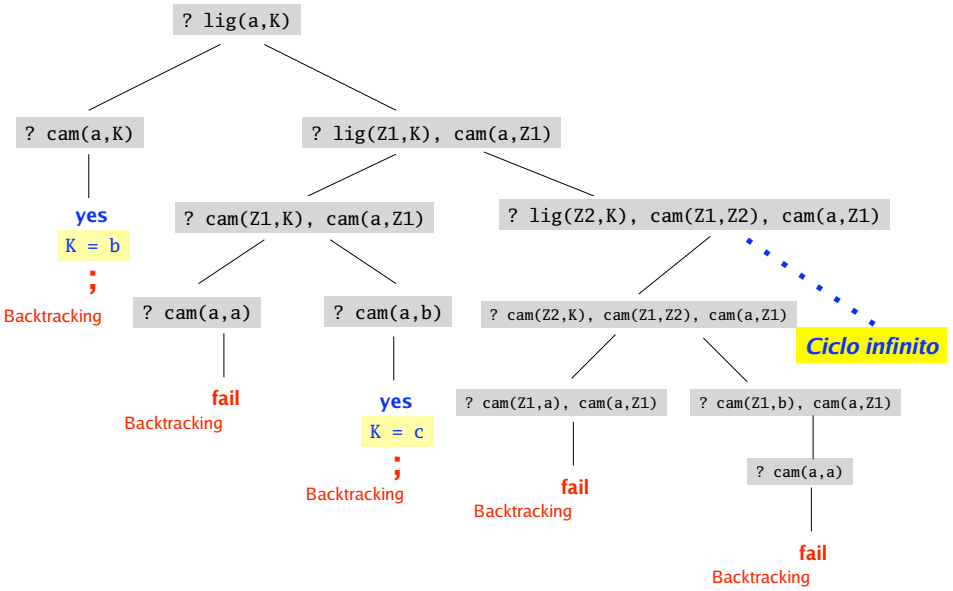
teriamos uma árvore de procura *infinita*.

Note que a única diferença entre o `ligação` e `lig` é a ordem em que aparecem os predicados no corpo da 2ª cláusula.

```
ligacao(X,Y) :- caminho(X,Z), ligacao(Z,Y).

lig(X,Y) :- lig(Z,Y), cam(X,Z).
```


Árvore de Procura



Exemplos com listas

```
% pertence(X,L) indica que X é um elemento da lista L
pertence(X,[X|_]).
pertence(X,[_|T]) :- pertence(X,T).
```

```
% prefixo(L1,L2) indica que a lista L1 é prefixo da lista L2
prefixo([],_).
prefixo([X|Xs],[X|Ys]) :- prefixo(Xs,Ys).
```

```
% sufixo(L1,L2) indica que a lista L1 é sufixo da lista L2
sufixo(L,L).
sufixo(Xs,[Y|Ys]) :- sufixo(Xs,Ys).
```

```
% concatena(L1,L2,L3) indica qua a lista L1 concatenada com a lista L2 é a lista L3
concatena([],L,L).
concatena([H|T],L1,[H|L2]) :- concatena(T,L1,L2).
```

Exercício:

Carregue estas definições no interpretador e interrogue a base de conhecimento. Por exemplo:

```
? pertence(2,[1,2,3]).
? pertence(X,[1,2,3]).
? pertence(3,L).
? pertence(X,L).
```


Exercícios:

1. Considere a representação dos números naturais baseada nos construtores 0 e sucessor:

   0, suc(0), suc(suc(0)), suc(suc(suc(0))), ...

   O predicado `nat` que testa se um termo é um número natural.

   ```
   nat(0).
   nat(suc(X)) :- nat(X).
   ```

   Defina, usando o functor `suc`, predicados que implementem as seguintes relações:

   a) menor ou igual
   b) mínimo
   c) soma
   d) multiplicação
   e) factorial
   f) exponenciação

2. Defina o predicado `last(X,L)` que testa se `X` é o último elemento da lista `L`.

3. Defina a relação `divide(L,L1,L2)` que divide a lista `L` em duas listas `L1` e `L2` com aproximadamente o mesmo tamanho.