

Programação Prolog

April 13, 2016

Programação Prolog

- Programas Prolog são representados por **cláusulas de Horn**, um subconjunto de **lógica de primeira ordem**, onde cada cláusula só pode conter no máximo um literal positivo na *cabeça* da cláusula (máximo um consequente positivo).
- em Prolog, cada cláusula é representada por **cabeça :- corpo**.
- Prolog: linguagem declarativa.

Programação Prolog

- Programa: conjunto de **fatos** e/ou **regras** que definem relações entre objetos.
- *Fatos*: relações consideradas sempre verdadeiras (axiomas).
- *Regras*: Relações que são verdadeiras ou falsas dependendo de outras relações.
- Computação de um programa em lógica é a dedução dos consequentes do programa.

Programação Prolog

- Exemplos:

%Fatos:

valioso(ouro).

sexo_feminino(jane).

pai(john,mary).

humano(socrates).

ateniense(socrates).

Regras:

gosta(john,X) :-

gosta(X,vinho).

passaro(X) :-

animal(X),

tem_penas(X).

irma(X,Y) :-

sexo_feminino(X),

pais(M,F,X),

pais(M,F,Y).

- Atenção à sintaxe!

Programação Prolog: A Linguagem – Sintaxe

- Termos:
 - ▶ Variáveis: X, Y, C1, _ABC, Input
 - ▶ Constantes: prolog, a, 123, 'rio de janeiro'
 - ▶ Estruturas (termos compostos):
`dono(john,livro(ulysses,autor(james,joyce)))`
- Caracteres: letras maiúsculas, letras minúsculas, dígitos, sinais do teclado.
- Símbolos especiais: :- ; , .
- Comentários:
 - ▶ linha: `% isto e' um comentario.`
 - ▶ texto: `/* este tambem e' um comentário */`

Programação Prolog: A Linguagem – Sintaxe

- Operadores: +, -, *, / etc.
- Igualdade e “matching”:
 $a(b,c,d(e,F,g(h,i,j))) = a(B,C,d(E,f,g(H,i,j)))$
- Aritmética números: =, \= , <, >, =<, >=
- Aritmética strings/termos: ==, \== @<, @>
- Observação: Prolog **não avalia** expressões que não apareçam explicitamente no corpo da cláusula no contexto do operador especial **is**.
 - ▶ $p(2+3,4*5)$.
 - ▶ Estas operações não são avaliadas!!!
 - ▶ Para obrigar a avaliação: $p(X,Y) :- X \text{ is } 2+3, Y \text{ is } 4*5$.

Programação Prolog: Exemplo simples

```
parent(C,M,F) :- mother(C,M),father(C,F).
```

```
mother(john,ann).
```

```
mother(mary,ann).
```

```
father(mary,fred).
```

```
father(john,fred).    female(mary).
```

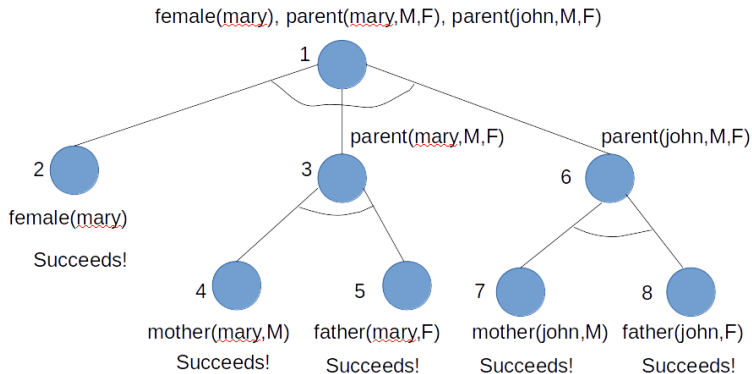
Consulta:

```
?- female(mary),parent(mary,M,F),parent(john,M,F).
```

Programação Prolog: Exemplo simples – observações

- Interpretação declarativa da consulta: “dado que `mary` é do sexo feminino, `mary` e `john` são irmãos”?
- Interpretação procedural: “Para resolver `parent/3`, precisamos resolver `mother/2` & `father/2`”.
- NB: `pred/2` é uma notação sintática para representar um predicado e sua aridade – número de argumentos.
- Mecanismo procedural de execução constrói uma *árvore de execução*.

Árvore de Execução para exemplo simples



Programação Prolog: Outras estruturas de Dados

- Listas: estrutura de dados especial em Prolog.
- Exs:
 - ▶ []: lista vazia.
 - ▶ [the,men,[like,to,fish]]
 - ▶ [a,V1,b,[X,Y]]
- Estrutura geral de lista não vazia: [**Head**|**Tail**]
- Head: primeiro elemento da lista (pode ser de qualquer tipo).
- Tail: lista restante (tipo é **obrigatoriamente** uma lista).

Programação Prolog: Outras estruturas de Dados

- Exemplos:

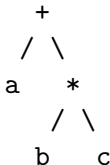
Lista	Head	Tail
[a,b,c]	a	[b,c]
[a]	a	[]
[[the,cat],sat]	[the,cat]	[sat]
[the,[cat,sat]]	the	[[cat,sat]]
[X+Y,x+y]	X+Y	[x+y]
[]	no head	no tail

Programação Prolog: igualdades de listas

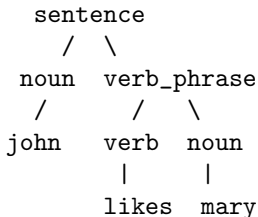
<code>[X,Y,Z]</code>	<code>=</code>	<code>[john,likes,fish]</code>	<code>X=john, Y=likes, Z=fish</code>
<code>[cat]</code>	<code>=</code>	<code>[X Y]</code>	<code>X=cat, Y=[]</code>
<code>[]</code>	<code>=</code>	<code>[X Y]</code>	sempre falha!!!

Programação Prolog: Árvores

- $a+b*c$, $+(a,*(b,c))$



- `sentence(noun(john),verb_phrase(verb(likes),noun(mary)))`



Programação Prolog: obtenção de múltiplas soluções

- *Backtracking* (retrocesso) é utilizado em Prolog na presença de falhas ou para obtenção de múltiplas soluções.
- Quando tenta satisfazer um objetivo, Prolog marca aquele objetivo como sendo um ponto de escolha.
- Se o objetivo falhar, Prolog desfaz todo o trabalho feito até o ponto em que criou o ponto de escolha.
- A partir daí, começa novamente procurando outra alternativa para o objetivo.

Backtracking – Exemplo

```
avo_ou_avo'(X,Y) :- pai_ou_mae(X,Z), pai_ou_mae(Z,Y).
```

```
pai_ou_mae(jane,charles).
```

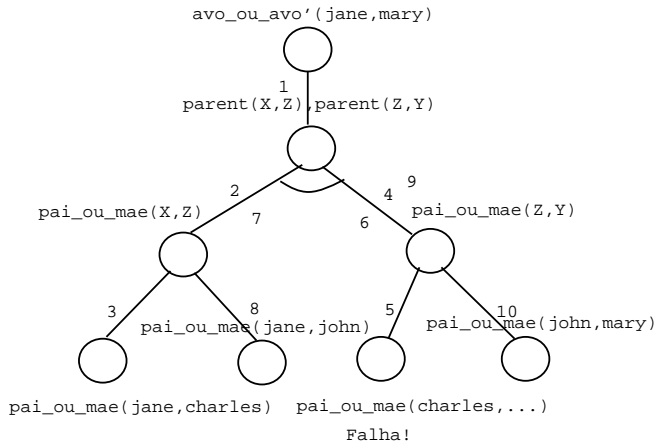
```
pai_ou_mae(john,mary).
```

```
pai_ou_mae(fred,jane).
```

```
pai_ou_mae(jane,john).
```

```
consulta: ?- avo_ou_avo'(jane,mary).
```

Backtracking – Árvore de execução do Exemplo



Apos passo 5 -> backtracking! Passo 6: desfaz computacao
 Passo 7: inicia nova solucao Passo 8: encontra nova solucao
 Passo 9: inicia nova execucao de pai_ou_mae(john,Y)
 Passo 10: encontra solucao!

O Operador de Corte – ! (*cut*)

- Controlado pelo programador.
- Reduz o espaço de busca.
- Implementa exceções.
- Em combinação com **fail**.
- Exemplos:

```
append([],X,X) :- !.
```

```
append([A|B],C,[A|D]) :- append(B,C,D).
```

```
not(P) :- call(P),!,fail.
```

```
not(P).
```

O Operador de Corte – comentários sobre exemplos

- No primeiro exemplo: redução do espaço de busca.
- Utilizado quando o utilizador entra com consulta do tipo: `append(L1,L2,[a,b,c])`, onde há várias soluções (há várias sublistas L1 e L2 que quando concatenadas resultam em `[a,b,c]`).
- Segundo exemplo implementa combinação cut-fail. Implementa *negação por falha finita*. Se P for verdadeiro, isto é, Prolog consegue satisfazer P, então `not(P)` retorna falha (falso). Se Prolog não conseguir satisfazer P, então `not(P)` retorna verdadeiro através da segunda cláusula.

Alguns Predicados Pré-definidos

- Entrada/Saída.
 - ▶ Abrindo e fechando ficheiros em disco: `open(Fd,name,rwa)`, `close(Fd)`.
 - ▶ Lendo ficheiros: `see`, `seeing`, `seen`
 - ▶ Escrevendo em ficheiros: `tell`, `telling`, `told`
 - ▶ Escrevendo termos e caracteres: `nl`, `tab`, `write`, `put`, `display`, `format`
 - ▶ Lendo termos e caracteres: `get`, `get0`, `read`, `skip`
 - ▶ Consultando (carregando) programas: `[ListOfFiles]`, `consult`, `reconsult`
 - ▶ Compilando programas: `compile(file)`
 - ▶ nb.: predicados dinâmicos e estáticos – `:-dynamic pred/n`, `static pred/n`

Alguns Predicados Pré-definidos

- Declaração de operadores.
 - ▶ posição
 - ▶ classe de precedência
 - ▶ associatividade
- Operadores possíveis: ('f' é a posição do operador, 'y' representa expressões que podem conter operadores com ordem de precedência maior ou igual do que a precedência do operador definido, 'x' representa expressões que podem conter operadores com ordem de precedência menor do que a precedência do operador definido)
 - ▶ binários: `xfx`, `xfy`, `yfx`, `yfy`
 - ▶ unários: `fx`, `fy`, `xf`, `yf`
- Exemplos:

```
:- op(255,xfx,':-').
```

```
:- op(40,xfx,'=').
```

```
:- op(31,yfx,'-').
```

Entrada de Novas cláusulas

- Forma 1: consultar ficheiro editado off-line.
- Forma 2: [user]. No modo consulta, o sistema coloca outro 'prompt' e espera o utilizador entrar com definições de novas cláusulas. Para sair do modo de entrada de cláusulas: **Ctrl-D** em unix e **Ctrl-Z** em DOS.
- Forma 3: utilização de predicado pré-definido: **assert**.
 - ▶ **asserta(p(a,b))** insere antes do primeiro predicado **p/2**, uma nova cláusula **p(a,b) ..**
 - ▶ **assertz(p(a,b))** insere após o último predicado **p/2**, uma nova cláusula **p(a,b) ..**

Outros predicados pré-definidos

- Sucesso e falha: `true` e `fail`.
- Classificação de termos: `var(X)`, `atom(X)`, `nonvar(X)`, `integer(X)`, `atomic(X)`.
- Meta-programação: `clause(X,Y)`, `listing(A)`, `retract(X)`, `abolish(X)`, `setof`, `bagof`, `findall`.
- Mais meta-programação: `functor(T,F,N)`, `arg(N,T,A)`, `name(A,L)`, `X =.. L`.
- Afetando backtracking: `repeat`.
- Conjunção, Disjunção e execução de predicados: `X, Y`, `X; Y`, `call(X)`, `not(X)`.
- Depurando programas: `trace`, `notrace`, `spy`.

Manipulação de Listas

- Relação de pertinência a conjuntos: relação `pertence`, número de argumentos necessários: 2, o elemento que se procura e a lista (conjunto de elementos).

```
/* X foi encontrado, portanto pertence 'a lista */
pertence(X, [X|_]).
```

```
/* X ainda nao foi encontrado, portanto
   pode pertencer ao resto da lista */
pertence(X, [Y|L]) :- pertence(X,L).
```

Interpretação declarativa:

- ▶ X pertence a uma lista se for o primeiro elemento
- ▶ Se X pertence a lista L então X continua pertencendo a esta lista se lhe for adicionado mais um elemento.

Manipulação de Listas

- Concatenação de duas listas: relação `concat`, número de argumentos necessários: 3, duas listas de entrada e a lista resultante da concatenação.

```
/* a concatenacao de uma lista vazia com qualquer lista
   e' a propria lista */
```

```
concat([],L,L).
```

```
/* o resultado da concatenacao de uma lista
   nao vazia L (representada por [H|L1] com
   outra lista qualquer L2 e' uma lista que
   contem o primeiro elemento da primeira lista (H)
   e cuja cauda e' o resultado da concatenacao da
   cauda da primeira lista com a segunda lista.
```

```
*/
```

```
concat([H|L1],L2,[H|L3]) :- concat(L1,L2,L3).
```


Manipulação de Listas

Interpretação declarativa:

- a concatenação da lista vazia com qualquer lista é a própria lista
- Se a concatenação das listas L1 e L2 é L3, então se adicionarmos um elemento a L1, este elemento também deve ser adicionado à lista resultante L3.

Manipulação de Listas

- Encontrar o último elemento de uma lista: relação `last`, número de argumentos necessários: 2, o elemento que se procura e a lista (conjunto de elementos). Semelhante ao programa `pertence`.

```
/* X e' o ultimo elemento, pois
   a lista contem um unico elemento */
last(X,[X]).
```

```
/* X nao e' o ultimo, pois a lista contem
   mais elementos, para ser ultimo tem que estar
   na cauda da lista */
last(X,[_|L]) :- last(X,L).
```

Interpretação declarativa da segunda cláusula: se `X` for o último elemento de `L` então `X` continua sendo o último elemento de `L` com mais um elemento.

Manipulação de Listas

- Reverso de uma lista: relação `rev`, número de argumentos necessários: 2, uma lista de entrada e a lista resultante reversa.

```
/* o reverso de uma lista vazia e' a lista vazia */
rev([], []).
```

```
/* o reverso de uma lista nao vazia [H|L1] e' obtido
   atraves da concatenacao do reverso da cauda desta
   lista (L1) com o primeiro elemento da lista (H)
```

```
*/
rev([H|L1],R) :- rev(L1,L2), concat(L2,[H],R).
```

Interpretação declarativa da segunda cláusula: Se o reverso de $L1$ é $L2$ e a concatenação de $L2$ com um elemento qualquer H é R , então o reverso da lista $[H|L1]$ é R .

Manipulação de Listas

- Tamanho de uma lista: relação `tamanho`, número de argumentos: 2, a lista e o argumento de saída correspondente ao tamanho da lista. Idéia: o tamanho de uma lista `L` é obtido através do tamanho da lista menor `L'` sem o primeiro elemento mais 1.

```
/* o tamanho da lista vazia e' zero */
```

```
tam([],0).
```

```
/* o tamanho da lista nao vazia L ([H|L1]) e' obtido
   atraves do tamanho da lista menor L1 (sem o primeiro
   elemento) mais 1.
```

```
*/
```

```
tam([H|L1],N) :- tam(L1,N1), N is N1 + 1.
```

Manipulação de Listas

- Remoção de um elemento X de uma lista L: relação `remove`, número de argumentos: 3, a lista de entrada, o elemento a ser removido, e a lista de saída.

```
/* remove X de lista vazia e' lista vazia */  
remove([],X,[]).
```

```
/* remove X de lista que contem X e' a lista  
sem X
```

```
*/  
remove([X|L],X,L).
```

```
/* ainda nao encontrou X. Continua procurando */  
remove([Y|L],X,[Y|L1]) :- remove(L,X,L1).
```

Manipulação de Listas

- Como modificar este programa para remover **todos** os elementos X de uma lista qualquer?
- Posso alterar a ordem das cláusulas?

Árvore Binária: Busca e Inserção

- Dicionário ordenado binário:
 - ▶ relação lookup,
 - ▶ número de argumentos: 3
 - a chave a ser inserida ou procurada,
 - o dicionário,
 - a informação resultante sobre a chave consultada.
 - ▶ estrutura de dados para o dicionário: `arvbin(K,E,D,I)`
 - K: chave.
 - E: sub-árvore da esquerda.
 - D: sub-árvore da direita.
 - I: informação sobre a chave consultada.

Dicionário binário ordenado

```
/* chave foi encontrada ou e' inserida */  
lookup(K,arvbin(K,_,_,I),I).  
/* chave ainda nao foi encontrada, pode estar na  
   sub-arvore da esquerda, se for menor do que  
   a chave corrente, ou na sub-arvore da direita,  
   se for maior do que a chave corrente.  
*/  
lookup(K,arvbin(K1,E,_,I1),I) :-  
    K < K1, lookup(K,E,I).  
lookup(K,arvbin(K1,_,D,I1),I) :-  
    K > K1, lookup(K,D,I).
```


Meta-interpretador Prolog para Prolog

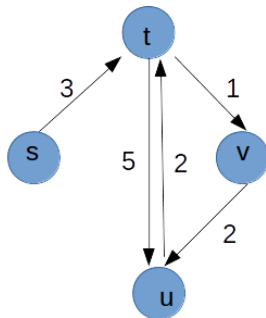
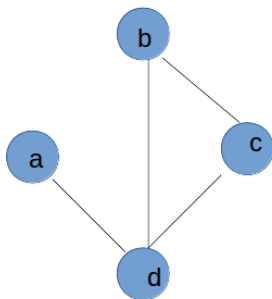
- Um meta-interpretador é um programa que executa outros programas.
- O nosso exemplo implementa um interpretador de Prolog escrito em Prolog.
- Relação: `interp`, número de argumentos: 1, termo Prolog a ser interpretado.

Meta-interpretador Prolog para Prolog

```
/* a interpretacao do termo basico true
   deve ser sempre satisfeita (axioma)
*/
interp(true).
/* a interpretacao de uma conjuncao e'
   satisfeita, se a interpretacao de cada
   termo/literal da conjuncao for satisfeita
*/
interp((G1,G2)) :-
    interp(G1),
    interp(G2).
/* a interpretacao de uma disjuncao e'
   satisfeita, se pelo menos uma interpretacao
   de alguma disjuncao for satisfeita
*/
interp((G1;G2)) :-
    interp(G1);
    interp(G2).
/* a interpretacao de um literal simples
   e' satisfeita, se o literal existir no
   programa e seu antecedente for satisfeito
*/
interp(G) :-
    clause(G,B),
    interp(B).
```

Busca em Grafos

- Representação de grafos: conjunto de nós e conjunto de ramos/arestas/conexões/arcs. Arcos são geralmente representados por pares ordenados.
- Exemplos:



Busca em Grafos: representação de um grafo em Prolog

- Forma 1:

% Grafo nao direcionado	Grafo direcionado
conectado(a,b).	arco(s,t,3).
conectado(b,c).	arco(t,v,1).
conectado(c,d).	arco(u,t,2).
...	...

- Forma 2:

```

grafo([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
digrafo([s,t,u,v],
[a(s,t,3),a(t,v,1),a(t,u,5),a(u,t,2),a(v,u,2)])
    
```

- Forma 3:

```

[a->[b], b->[a,c,d], c->[b,d], d->[b,c]]
[s->[t/3], t->[u/5,v/1], u->[t/2],v->[u/2]]
    
```

Atenção à utilização dos símbolos -> e /.

Busca em Grafos

- Operações típicas em grafos:
 - ▶ encontrar um caminho entre dois nós do grafo,
 - ▶ encontrar um sub-grafo de um grafo com certas propriedades.
- Programa exemplo: encontrar um caminho entre dois nós do grafo.

Busca em Grafos

- **G**: grafo representado com forma 1.
- **A** e **Z**: dois nós do grafo.
- **P**: caminho **acíclico** entre **A** e **Z**.
- **P** representado como uma lista de nós no caminho.
- Relação **path**, número de argumentos: 4, nó fonte (**A**), nó destino (**Z**), caminho parcial percorrido (**L**) e caminho total percorrido (**P**).
- um nó só pode aparecer uma única vez no caminho percorrido (não há ciclos).

Busca em Grafos

- Método para encontrar um caminho acíclico P , entre A e Z , num grafo G .
- Se $A = Z$, então $P = [A]$,
- senão, encontre um caminho parcial de A a Y e encontre um caminho acíclico P_1 do nó Y ao nó Z , evitando passar por nós já visitados em L (caminho parcial).

Busca em Grafos: Programa

```

/* Chegar de A a Z, retornando caminho percorrido P,
   começando de um caminho vazio
*/
path(A,Z,P) :- path1(A,Z,[],P).

/* Se ja' cheguei ao destino, parar a busca e incluir
   destino na lista contendo o caminho percorrido */
path1(Z,Z,L,[Z|L]).

/* Se ainda nao cheguei no destino, encontrar caminho
   parcial de A para Y, verificar se este caminho
   e' valido (isto e', ja' passei por Y antes?). Se for
   um caminho valido, adicionar 'a lista contendo o
   caminho parcial e continuar a busca a partir de Y
*/
path1(A,Z,L,P) :-
    (conectado(A,Y);      /* encontra caminho parcial */
     conectado(Y,A)),    /* de A para Y ou de Y para A */
    \+ member(Y,L),     /* verifica se ja passei por Y */
    path1(Y,Z,[Y|L],P). /* encontra caminho parcial de */
                        /* Y a Z */

```