UMA INTRODUÇÃO A PROLOG

Newton José Vieira

UFMG

18 de novembro de 2011

Introdução

Programação em lógica:

- O ideal: **especificação executável**.
- O problema: explosão combinatória.

Prolog impõe dois tipos de restrições:

- Linguagem: **cláusulas de Horn**.
- Estratégia de busca: não tem.

A programação pode ser feita em duas fases:

- Especificação (programação) declarativa.
- Rearranjo (e talvez modificações) tendo em vista aspectos de controle e eficiência.

O que é um programa

A seguir, cada A_i é uma fórmula atômica.

- **Programa**: sequência de cláusulas de programa.
- **Cláusula de programa**:
  - Fato: A_0.
  - Regra: A_0 :- A_1,A_2,...,A_n. para $n \geq 1$.

    A regra expressa $\forall x_1 \ldots \forall x_k (A_0 \vee \neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n)$.
- **Cláusula meta**: ?- A_1,A_2,...,A_n. para $n \geq 1$.

  A meta expressa a consulta $\exists x_1 \ldots \exists x_k (A_1 \wedge A_2 \wedge \ldots \wedge A_n)$.

  Negando-se a meta: $\forall x_1 \ldots \forall x_k (\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n)$.

Execução de um programa

Método de inferência:

- **resolução de entrada** com a cláusula meta como conjunto de suporte.
- Restrições:
  - **Regra de computação**: escolhe literal mais à esquerda.
  - **Regra de busca**: escolhe a primeira cláusula de entrada (ainda não tentada) na ordem de ocorrência no texto.

Uma implicação:

- Resolventes só têm literais negativos.

A execução de um programa

Seja

?- $A_1, A_2, \ldots, A_k$.

o resolvente atual.

- Pela **regra de computação**: escolhe A_1.
- Pela **Regra de busca**: escolhe a primeira cláusula de entrada (ainda não tentada). Seja B_0 :- $B_1, B_2, \ldots, B_n$ tal cláusula ($n \geq 0$).

O novo resolvente é:

?- $B_1\sigma, B_2\sigma, \ldots, B_n\sigma, A_2\sigma, \ldots, A_k\sigma$.

onde σ é umg de A_1 e B_0.

Obtenção da resposta durante a execução

Se a cláusula meta é:

?- $A_1, A_2, \ldots, A_k$. com variáveis livres $x_1, \ldots, x_n$

será inicialmente anexado um literal resp($x_1, \ldots, x_n$) assim:

resp($x_1, \ldots, x_n$) :- $A_1, A_2, \ldots, A_k$

Seja B_0 :- $B_1, B_2, \ldots, B_n$ a cláusula a resolver, sendo σ umg de A_1 e B_0.

O novo resolvente é:

resp$(x_1, \ldots, x_n)\sigma$:- $B_1\sigma, B_2\sigma, \ldots, B_n\sigma, A_2\sigma, \ldots, A_k\sigma$.

No lugar de $\perp$ será deduzido resp$(x_1, \ldots, x_n)\theta$, e a resposta será:

$x_1 = x_1\theta$
$\vdots$
$x_n = x_n\theta$

Um exemplo de programa em Prolog

```
 1. masc(pedro).
 2. masc(paulo).
 3. masc(lucas).
 4. fem(ana).
 5. fem(maria).
 6. fem(joana).
 7. pai(pedro, lucas).
 8. pai(pedro, joana).
 9. mae(ana, maria).
10. mae(ana, lucas).
11. genit(X,Y) :- pai(X,Y).
12. genit(X,Y) :- mae(X,Y).
13. filho(X,Y) :- genit(Y,X),masc(X).
14. filha(X,Y) :- genit(Y,X),fem(X).
15. irmao(X,Y) :- filho(X,Z),genit(Z,Y),X \= Y.
```


Um exemplo de execução de programa em Prolog

Para a consulta: `?- irmao(lucas,X).`:

```
16. resp(X):- irmao(lucas,X).                                  [consulta]
17. resp(Y):- filho(lucas,Z),genit(Z,Y),lucas \= Y.            [15]
18. resp(Y):- genit(Z,lucas),masc(lucas),genit(Z,Y),lucas \= Y. [13]
19. resp(Y):- pai(Z,lucas),masc(lucas),genit(Z,Y),lucas \= Y.  [11] alt: 12
20. resp(Y):- masc(lucas),genit(pedro,Y),lucas \= Y.           [7] alt: 8
21. resp(Y):- genit(pedro,Y),lucas \= Y.                       [3]
22. resp(Y):- pai(pedro,Y),lucas \= Y.                         [11] alt: 12
23. resp(lucas):- lucas \= lucas.                              [7] alt: 8
      falha ⇒ backtracking
23'. resp(joana):- lucas \= joana.                             [8]
24. resp(joana).                                               [built-in]
```

Resposta:

```
X = joana
```

Após a resposta pode-se teclar ";" ou [enter].

Um outro exemplo

1. `ac(X,Y) :- ac(X,Z),adj(Z,Y).`
2. `ac(X,X).`
3. `adj(a,b).`
4. `adj(b,c).`

Para a consulta `?- ac(a,X).`**:**

5. `resp(X):- ac(a,X).` [consulta]
6. `resp(X):- ac(a,Z),adj(Z,X).` [1] alt: 2
7. `resp(X):- ac(a,Z'),adj(Z',Z),adj(Z,X).` [1] alt: 2
8. `resp(X):- ac(a,Z''),adj(Z'',Z'),adj(Z',Z),adj(Z,X).` [1] alt: 2

⋮

loop!

ERROR: Out of local stack

Solução: trocar cláusulas de programa 1 e 2.

Trocando cláusulas de programa 1 e 2

1. `ac(X,X).`
2. `ac(X,Y) :- ac(X,Z),adj(Z,Y).`
3. `adj(a,b).`
4. `adj(b,c).`

Para a mesma consulta `?- ac(a,X).`**:**

5. `resp(X):- ac(a,X).` [consulta]
6. `resp(a).` [1] alt: 2

Resposta: `X = a;` ⇐pedido de mais resposta

5. `resp(X):- ac(a,X).` [consulta]

backtracking (como se houvesse falhado)

6'. `resp(X):- ac(a,Z),adj(Z,X).` [2]
7. `resp(X):- adj(a,X).` [1] alt: 2
8. `resp(b).` [3]

Resposta: `X = b;` ⇐pedido de mais resposta

Continuando o exemplo

1. `ac(X,X).`
2. `ac(X,Y) :- ac(X,Z),adj(Z,Y).`
3. `adj(a,b).`
4. `adj(b,c).`

5. `resp(X):- ac(a,X).` [consulta]
6'. `resp(X):- ac(a,Z),adj(Z,X).` [2]

backtracking (como se houvesse falhado)

7'. `resp(X):- ac(a,Z),adj(Z,Y),adj(Y,X).` [2]
8'. `resp(X):- adj(a,Y),adj(Y,X).` [1] alt: 2
9. `resp(X):- adj(b,X).` [3]
10. `resp(c).` [4]

Resposta: `X = c;` ⇐pedido de mais resposta

Agora ocorre loop por causa da recursão à esquerda na cláusula 2!

⇒ **Substituir 2 por** `ac(X,Y) :- adj(X,Z),ac(Z,Y).`

(O que contece se a consulta for `?- ac(X,c).`**?)**

Estruturas de dados

Construídas por meio de formas funcionais. Por exemplo:

```
seminario('Logica',data(1,4,2012),prof('Abel'),local('ICEx',2077))
```

Falando sobre objetos geométricos:

1. `horizontal(sr(p(X,Y),p(Z,Y))).`
2. `vertical(sr(p(X,Y),p(X,Z))).`

Algumas consultas:

```
?- horizontal(sr(p(0,1),p(4,X))).
 X = 1

?- vertical(sr(X,p(4,1))).
 X = p(4,Z)

?- vertical(X),horizontal(X).
 X = sr(p(Z,Y),p(Z,Y))
```


Listas

O que é lista:

- `[]` é lista (a lista vazia);
- se L é lista e T é um termo, então $[T|L]$ é lista.

Obs: listas são termos.

Simplificação: $[T_1|[T_2 \mid \ldots [T_n \,|[]] \ldots]] = [T_1, T_2, \ldots, T_n]$

Exemplos de listas:

normal	simplificada
`[a|[]]`	`[a]`
`[a|[b|[c|[]]]]`	`[a,b,c]`
`[[]|[]]`	`[[]]`
`[[1,2]|[3,4]]`	`[[1,2],3,4]`
`[a| X]`	não tem

Manipulação de listas/exemplos

`list(X)`: **X é lista**

```
list([]).
list([X|L]) :- list(L).
```

`member(X,L)`: **X é membro da lista L**

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

`prefix(P,L)`: **a lista P é prefixo da lista L**

```
prefix([],L) :- list(L).
prefix([X|P],[X|L]) :- prefix(P,L).
```

`sufix(S,L)`: **a lista S é sufixo da lista L**

```
sufix(L,L) :- list(L).
sufix(S,[X|L]) :- sufix(S,L).
```


Manipulação de listas/exemplos

`append(L1,L2,L3)`: **L3 é a junção das listas L1 e L2**

```
append([],L,L) :- list(L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

`reverse(L1,L2)`: **a lista L2 é o reverso da lista L1**

```
reverse([],[]).
reverse([X|L1],L3) :- reverse(L1,L2),append(L2,[X],L3).
```

Mais eficiente:

```
reverse(L1,L2) :- revac(L1,[],L2).
revac([],L,L).
revac([X|R1],L1,L2) :- revac(R1,[X|L1],L2).
```


Simulação de autômatos finitos

AF representado por 2 relações:

- `final(E)`: **E é um estado final.**
- `trans(E1,A,E2)`: $\delta(\mathbf{E1}, \mathbf{A}) = \mathbf{E2}$.
  `trans(E1,lambda,E2)`: $\delta(\mathbf{E1}, \lambda) = \mathbf{E2}$.

Exemplo:

```
trans(e1,0,e1).
trans(e1,0,e2).
trans(e1,1,e1).
trans(e2,1,e3).
trans(e2,lambda,e4).
trans(e3,lambda,e1).
trans(e3,1,e4).
final(e3).
```

Simulação de autômatos finitos

`aceita(E,W)`: **aceita a palavra W a partir do estado E.**

```
aceita(E,[]) :  final (E).
aceita(E,[A|Y]) :- trans(E,A,E1), aceita(E1,Y).
aceita(E,W) :- trans(E,lambda,E1), aceita(E1,W).
```

Exemplos de consultas:

```
?- aceita(e1,[0,0,0,1])
   true

?- aceita(E,[0,1])
   E = e1;
   E = e3;
   false

?- aceita(e1,[_,_])
   true
```


Macaco e bananas

Um estado é representado por um termo `state(H,V,C,B)`**, sendo:**

- `H`: **posição horizontal do macaco;**
- `V`: **posição vertical do macaco;**
- `C`: **posição da caixa; e**
- `B`: **macaco tem ou não as bananas.**

Estado meta:

```
state(_,_,_,has)
```

Movimento expresso por: `move(State1,Move,State2)`

Macaco e bananas/cont.

Os 4 operadores (ações do macaco):

1. **pega as bananas (**`grasp`**):**

```
move(state(middle,onbox,middle,hasnot), grasp,
     state(middle,onbox,middle,has))
```

2. **sobe na caixa (**`climb`**):**

```
move(state(P,onfloor,P,H), climb,
     state(P,onbox,P,H))
```

3. **empurra a caixa (**`push`**):**

```
move(state(P1,onfloor,P1,H), push(P1,P2),
     state(P2,onfloor,P2,H))
```

4. **anda (**`walk`**):**

```
move(state(P1,onfloor,B,H), walk(P1,P2),
     state(P2,onfloor,B,H))
```


Macaco e bananas/O programa

A consulta: `?- canget(state(atdoor,onfloor,atwindow,hasnot)).`

Além das 4 cláusulas para `move`**, o programa contém:**

```
canget(state(_,_,_,has)).
canget(State1) :- move(State1,Move,State2), canget(State2).
```

Cuidado: `walk` **não pode vir em primeiro lugar!**

Quicksort em Prolog

quicks(L1,L2): **a lista L2 é a lista L1 ordenada**

```
quicks([],[]).
quicks([X|R],O) :- split(R,X,P1,P2),
                   quicks(P1,O1),
                   quicks(P2,O2),
                   append(O1,[X|O2],O).
split([],X,[],[]).
split([Y|R],X,[Y|R1],P2) :- Y =< X,split(R,X,R1,P2).
split([Y|R],X,P1,[Y|R2]) :- Y > X,split(R,X,P1,R2).
```


O corte

O corte (!):

- **Avaliado com sucesso.**
- **Caso haja retrocesso para o mesmo, nenhuma alternativa à cláusula em que aparece é considerada.**

Exemplos:

merge(L1,L2,L3): **a lista L3 é o merging das listas L1 e L2**

```
merge([X|R1],[Y|R2],[X|R3]) :- X < Y,!,merge(R1,[Y|R2],R3).
merge([X|R1],[Y|R2],[X|[Y|R3]]) :- X = Y,!,merge(R1,R2,R3).
merge([X|R1],[Y|R2],[Y|R3]) :- X > Y,!,merge([X|R1],R2,R3).
merge([],L,L) :- !.
merge(L,[],L) :- !.
```


Outro exemplo com corte

min(X,Y,Z): **Z é o menor valor entre X e Y**

```
min(X,Y,X) :- X =< Y, !.
min(X,Y,Y) :- X > Y, !.
```

Cuidado com o corte:

```
min(X,Y,X) :- X =< Y, !.
min(X,Y,Y).                ⇒ Tente avaliar min(2,4,4).


member(X,[X|R]) :- !.
member(X,[Y|R]) :- member(X,R).

                           ⇒ member(X,[1,2,3]) tem só uma resposta.
```


A falha

A falha (fail):

- **Avaliado com falha.**

Exemplos:

Imprimir os elementos acessíveis a partir de a:

```
?- ac(a,X),write(X),nl,fail.
```

Para entrada de dados:

```
leia :- repita,read(X),processe(X),!.
processe(X) :- final(X),!.
processe(X) :- assert(X),fail.
repita.
repita :- repita.
```

Negação por falha

`not(X)`: `X` **não é dedutível; o mesmo que:**

```
not(X) :- X,!,fail.
not(X).
```

Exemplo:

```
gosta(ana,X) :- animal(X), not(cobra(X)).
```


Cuidado com o not!

Exemplo:

```
estudante(paulo).
casado(pedro).
estsolteiro(X) :- not(casado(X)), estudante(X).
```

Consulta:

```
?- estsolteiro(Paulo).
    true

?- estsolteiro(Pedro).
    false

?- estsolteiro(X).
    false !!!
```


FIM