Programação em Lógica

Prof. A. G. Silva

08 de setembro de 2016

Listas (revisão)

- Elementos separados por vírgulas entre colchetes:

```
[a, b, c]
```

- Qualquer termo pode ser componente de uma lista, como variáveis ou outras listas:

```
[o, homem, [gosta, de, pescar]]
[a, V1, b, [X, Y]]
```

- Notação para a decomposição de uma lista em cabeça e cauda:

```
[X|Y]
```

Membro de lista (revisão)

- `member(X, Y)` é verdadeiro se o termo `X` é um elemento da lista `Y`
- Duas condições a verificar:
  - É um fato que `X` é um elemento da lista `Y`, se `X` for igual à cabeça de `Y`
    `member(X, [X|_]).`
  - `X` é membro de `Y`, se `X` é membro da cauda de `Y` (recursão):
    `member(X, [_|Y]) :- member(X, Y).`

Validação de lista (revisão)

- Predicados podem funcionar bem em chamadas com constantes, mas falhar com variáveis. O exemplo:

```
lista([A|B]) :- lista(B).
lista([ ]).
```

é a própria definição de lista, funcionando bem com constantes:

```
?- lista([a, b, c, d]).
true
?- lista([ ]).
true
?- lista(f(1, 2, 3)).
false
```

mas Prolog entrará em *loop* em (inverter a ordem das cláusulas resolve o problema):

```
?- lista(X).
```

Concatenação de listas (revisão)

- Predicado `append` para concatenar duas listas.

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

- Exemplos:

```
?- append([alfa, beta], [gama, delta], X).
X = [alfa, beta, gama, delta]
```

```
?- append(X, [b, c, d], [a, b, c, d]).
X = [a]
```

Acumuladores (revisão)

- Cálculos podem depender do que já foi encontrado até o momento
- A técnica de acumuladores consiste em utilizar um ou mais argumentos do predicado para representar "a resposta até o momento" durante este percurso
- Estes argumentos recebem o nome de acumuladores
- Prolog possui um predicado pré-definido `length` para o cálculo do comprimento de uma lista. Segue uma definição própria, **sem acumuladores**:

```
listlen([ ], 0).
listlen([H|T], N) :- listlen(T, N1), N is N1 + 1.
```

Acumuladores – comprimento de lista (revisão)

- A definição **com acumulador** baseia-se no mesmo princípio recursivo, mas acumula a resposta a cada passo num argumento extra:

```
lenacc([ ], A, A).
lenacc([H|T], A, N) :- A1 is A + 1, lenacc(T, A1, N).

listlen(L, N) :- lenacc(L, 0, N).
```

- Sequência de submetas para o comprimento de `[a, b, c, d, e]`:

```
listlen([a, b, c, d, e], N)
lenacc([a, b, c, d, e], 0, N)
lenacc([b, c, d, e], 1, N)
lenacc([c, d, e], 2, N)
lenacc([d, e], 3, N)
lenacc([e], 4, N)
lenacc([ ], 5, N)
```

Acumuladores – inversão de uma lista

- Acumuladores não precisam ser números inteiros. Segue definição do predicado `rev` (Prolog tem sua própria versão chamada `reverse`) para inversão da ordem dos elementos de uma lista:

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

- O segundo argumento `revacc` serve como acumulador. A sequência de metas para `?- rev([a, b, c, d], L3).`:

```
rev([a, b, c, d], L3)
revacc([a, b, c, d], [ ], L3)
revacc([b, c, d], [a], L3)
revacc([c, d], [b, a], L3)
revacc([d], [c, b, a], L3)
revacc([ ], [d, c, b, a], L3)
```

Breve revisão até agora

- Uma pergunta é a conjunção de várias metas
- A satisfação de uma meta consiste em uma busca no banco de dados, a partir do início, por uma cláusula unificante
  - Se não houver, a meta falha
  - Se houver, marca-se o ponto onde ela ocorre, e instanciam-se e ligam-se as variáveis conforme necessário (diz-se que a meta casou)
  - Se a cláusula unificante for um fato, a meta é satisfeita
  - Se for a cabeça de uma regra, a meta dá origem a um novo nível de submetas (todas as submetas devem ser satisfeitas)
- Ao satisfazer uma meta, passa-se para a próxima. Não havendo próxima, finaliza-se e informa-se o resultado (positivo), com os valores das variáveis da pergunta

Breve revisão até agora (cont...)

- Quando uma meta falha, a meta anterior sofre tentativa de ressatisfação (*backtracking* ou retrocesso). Não havendo meta anterior, finaliza-se e informa-se o resultado (negativo)

- A tentativa de ressatisfação apresenta as seguintes ressalvas:
  - A busca continua do ponto marcado no banco de dados (em vez de começar do início)
  - Desfazem-se as instanciações e ligações causadas pela última unificação desta meta

Backtracking e o corte

- Objetivos da aula:
  - Examinar o *backtracking* com mais detalhe
  - Conhecer o corte, um mecanismo especial que inibe o *backtracking* em certas condições

Gerando múltiplas soluções

- Considerando o banco de dados

```
pai(maria, jorge).
pai(pedro, jorge).
pai(sueli, haroldo).
pai(jorge, eduardo).
pai(X) :- pai(_,X).
```

Há dois predicados `pai`: um binário (pessoa e seu pai) e um unário (diferencia pais de não-pais). A pergunta

```
?- pai(X).
```

produzirá o seguinte resultado (`jorge` aparece duas vezes):

```
X = jorge ;
X = jorge ;
X = haroldo ;
X = eduardo ;
no
```

Gerando múltiplas soluções

- Outra situação: predicado `member` quando há repetições na lista

  ```
  member(a, [a, b, c, a, c, a, d, a, b, r, a])
  ```

  Pode ser satisfeita várias vezes (no caso, cinco)

- Há situações em que gostaríamos que fosse satisfeita uma única vez. Podemos instruir Prolog a descartar escolhas desnecessárias com o uso do corte

- Há casos onde geramos um número infinito de alternativas por não conhecermos de antemão quando aparecerá a alternativa de interesse:

  ```
  inteiro(0).
  inteiro(N) :- inteiro(M), N is M + 1.
  ```

  Gerará todos os inteiros a partir do zero. Pode-se usar outro predicado para selecionar alguns entre estes inteiros para uma dada aplicação

O "corte"

- Mecanismo em Prolog que instrui o sistema a não reconsiderar certas alternativas no processo de *backtracking*
- Pode ser importante para poupar memória e tempo de processamento
- Em alguns casos, o **corte** pode diferenciar um programa que funciona de outro que não funciona
- Sintaticamente, o **corte** é um predicado denotado por `!` (com nenhum argumento)
- Como meta, é sempre satisfeito da primeira vez, e sempre falha em qualquer tentativa de ressatisfação (como efeito colateral, impede a ressatisfação da meta que lhe deu origem ou meta mãe)

O "corte"

- Exemplo de regra:

  ```
  g :- a, b, c, !, d, e, f.
  ```

  Prolog realiza o *backtracking* normalmente entre as metas `a`, `b` e `c` até que o sucesso de `c` cause a satisfação do corte e Prolog passe para a meta `d`

  O processo de *backtracking* ocorre normalmente entre `d`, `e` e `f`, mas se `d`, em algum momento falhar, a meta envolvendo `g` que casou com esta regra também falha imediatamente

Três usos principais do corte

- Indicar que a regra certa foi encontrada
- Combinação corte-falha, indicando negação
- Limitar uma busca finita ou infinita

Confirmando a escolha certa

- Exemplo de fatorial de um número (não é a melhor definição):

```
fat(0, 1) :- !.
fat(N, F) :- N1 is N - 1, fat(N1, F1), F is F1 * N.
```

O corte impede que uma meta da forma `fat(0, F)` case com a segunda cláusula em caso de ressatisfação

Com corte:

```
?- fat(5, F).
F = 120 ;
no
```

Sem corte:

```
?- fat(5, F).
F = 120 ;
```

(*loop* infinito – *out of memory*)

Combinação corte-falha

- Predicado pré-definido sem argumentos chamado fail que sempre falha
- Pode-se usar em seu lugar qualquer meta incondicionalmente falsa como por exemplo $0 > 1$ (sem a elegância do fail). Em combinação com o corte, pode implementar negação
- Exemplo:

```
nonmember(X, L) :- member(X, L), !, fail.
nonmember(_, _).
```

  - Prolog vai tentar a primeira cláusula. Se member(X, L) for satisfeito, o corte será processado e logo a seguir vem fail
  - Devido ao corte, a tentativa de ressatisfação vai fazer a meta nonmember(X, L) falhar sem tentar a segunda cláusula.
  - No caso de member(X, L) falhar, o corte não será processado e será tentada a segunda cláusula que sempre é satisfeita

Outro modo de implementar a negação

- Existe a notação \+ em Prolog para indicar a negação, antecedendo uma meta. Exemplo:

```
nonmember(X, L) :- \+ member(X, L).
```

- Contudo, em geral estas negações só funcionam para metas onde os argumentos sejam todos instanciados

Limitando buscas

- Muitas vezes, usamos um predicado para gerar várias alternativas que serão testadas por um segundo predicado para escolher uma delas

- Em alguns casos, o predicado gerador tem a capacidade de gerar infinitas alternativas, e o corte pode ser útil para limitar esta geração

- Exemplo de divisão inteira (o Prolog possui o operador //):

```
divide(Numerador, Denominador, Resultado) :-
    inteiro(Resultado),
    Prod1 is Resultado * Denominador,
    Prod2 is Prod1 + Denominador,
    Prod1 =< Numerador,
    Prod2 > Numerador,
    !.
```

Sem o corte haveria um *loop* infinito pois há infinitos inteiros, candidatos a quociente, e tentativas de ressatisfação

Cuidados com o corte

- Suponha que queiramos usar `member` apenas para testar se elementos dados pertencem a listas dadas, sem nos importarmos com o número de vezes que aparecem. A definição

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

é apropriada. Porém perdemos a possibilidade das múltiplas alternativas:

```
?- member(X, [b, c, a]).
X = b ;
no
```

Outro exemplo

- Resultado inesperado:

```
pais(adao, 0).
pais(eva, 0).
pais(_, 2).

?- pais(eva, N).
N = 0
?- pais(adao, N).
N = 0
?- pais(eva, 2).
yes
```

- Forma de contornar o problema:

```
pais(adao, N) :- !, N = 0.
pais(eva, N) :- !, N = 0.
pais(_, 2).
```

Conclusão

- Ao introduzir cortes para que o predicado sirva a um certo tipo de meta, não há garantia que ele continuará funcionando a contento para outros tipos de metas

Exercícios

1. Conserte o predicado `fat(N, F)` para que não entre em *loop* em chamadas onde `N` é um número negativo e também em chamadas verificadoras, onde ambos `N` e `F` vêm instanciados.

2. Alguém teve a ideia de usar `nonmember` para gerar todos os termos que não estão na lista `[a, b, c]` com a pergunta

   ```
   ?- nonmember(X, [a, b, c]).
   ```

   Vai funcionar? Por quê?

3. Suponha que alguém queira listar os elementos comuns a duas listas usando a seguinte pergunta:

   ```
   ?- member1(X, [a, b, a, c, a]),
      member2(X, [c, a, c, a]).
   ```

   Quais serão os resultados nas seguinte situações:

Exercícios

(a) `member1` sem corte e `member2` sem corte?
(b) `member1` sem corte e `member2` com corte?
(c) `member1` com corte e `member2` sem corte?
(d) `member1` com corte e `member2` com corte?

Relembrando as definições com e sem corte:

```
member_com(X, [X|_]) :- !.
member_com(X, [_|Y]) :- member_com(X, Y).

member_sem(X, [X|_]).
member_sem(X, [_|Y]) :- member_sem(X, Y).
```