

Dados compostos

Marco A L Barbosa

malbarbo.pro.br

Departamento de Informática

Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-Compartilhual 4.0 Internacional.

<http://github.com/malbarbo/na-proglog>

Listas

Podemos representar uma lista em Prolog de forma semelhante a listas em Racket.

Uma lista é

- `vazia`; ou
- `cons(A, B)`, onde `A` é um termo qualquer e `B` é uma lista.

```
?- L0 = vazia, L1 = cons(3, vazia),  
    L2 = cons(3, cons(4, vazia)).
```

```
L0 = vazia,  
L1 = cons(3, vazia),  
L2 = cons(3, cons(4, vazia)).
```

A lista **L0** é vazia, a lista **L1** contém apenas o elemento 3 e a lista **L2** contém os elementos 3 e 4

Vamos definir um predicado que é verdadeiro se seu argumento é uma lista

```
%% lista(?X) is nondet
```

```
%
```

```
% Verdadeiro se X é uma lista.
```

```
?- lista(vazia).
```

```
true.
```

```
?- lista(cons(3, vazia)).
```

```
true.
```

```
?- lista(cons(3, cons(4, vazia))).
```

```
true.
```

```
?- lista(cons).
```

```
false.
```

```
?- lista(cons(3, 4)).
```

```
false.
```

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

```
X = vazia ;
```

```
X = cons(_, vazia) ;
```

```
X = cons(_, cons(_, vazia)) ;
```

```
X = cons(_, cons(_, cons(_, vazia))) ;
```

```
...
```

```
%% lista(?X) is semidet
%
% Verdadeiro se X é uma lista.
lista(vazia).

lista(cons(_, B)) :- lista(B).
```

Não existe nada diferente do vimos anteriormente. O “truque” é que estamos usando estruturas com autorreferência.

O Prolog já “entende” uma definição de lista semelhante a nossa

- `[]` ao invés de `vazia`
- `'.'` (no Prolog “clássico” e `'[]'` no SWI-Prolog) ao invés de `cons`

```
?- L0 = [], L1 = '.'(3, []), L2 = '.'(3, '.'(4, [])).
```

```
L0 = [],
```

```
L1 = [3],
```

```
L2 = [3, 4].
```

```
?- write('L0 = '), write_canonical([]).
```

```
L0 = []
```

```
?- write('L1 = '), write_canonical('.'(3, [])).
```

```
L1 = '.'(3, [])
```

```
?- write('L2 = '), write_canonical('.'(3, '.'(4, []))).
```

```
L2 = '.'(3, '.'(4, []))
```

Para exibir uma lista o Prolog utiliza uma notação mais amigável, mas podemos ver que a representação interna utiliza termos '.'.

Podemos utilizar esta notação amigável para construir listas

?- L0 = [], L1 = [3], L2 = [3, 4].

L0 = [],

L1 = [3],

L2 = [3, 4].

Podemos utilizar uma lista já existente para construir outra lista

```
?- X = [1, 2, 3], Y = '.'(5, X).
```

```
X = [1, 2, 3],
```

```
Y = [5, 1, 2, 3].
```

ou usando uma sintaxe mais amigável

```
?- X = [1, 2, 3], Y = [ 5 | X ], Z = [3, 4 | X].
```

```
X = [1, 2, 3],
```

```
Y = [5, 1, 2, 3],
```

```
Z = [3, 4, 1, 2, 3].
```

A notação $[A \mid B]$ é equivalente a $'.'$ (A, B)

?- $X = '.'(3, '.'(4, []))$, $Y = [3 \mid [4 \mid []]]$, $Z = [3, 4]$.

$X = Y$, $Y = Z$, $Z = [3, 4]$.

Como obter os componentes de uma lista?

- Da mesma forma que obtemos os componentes de outras estruturas, usando unificação
- Lembre-se, uma lista é um termo e pode ser usado da mesma forma que qualquer outro termo

?- '.'(A, B) = '.'(7, '.'(3, '.'(4, []))).

A = 7,

B = [3, 4].

?- [A | B] = '.'(7, '.'(3, '.'(4, []))).

A = 7,

B = [3, 4].

?- [A | B] = [7 | [3 | [4 | []]]].

A = 7,

B = [3, 4].

$$?- [A \mid B] = [7, 3, 4].$$

$$A = 7,$$

$$B = [3, 4].$$

$$?- [A, B \mid C] = [7, 3, 4].$$

$$A = 7,$$

$$B = 3,$$

$$C = [4].$$

$$?- [A, B, C] = [7, 3, 4].$$

$$A = 7,$$

$$B = 3,$$

$$C = 4.$$

Para projetar muitos tipos de predicados podemos usar a mesma ideia que usávamos para escrever funções em Racket.

Como a definição de lista tem dois casos, o modelo para lista também têm dois casos

```
pred_lista([], ...) :- ???.
```

```
pred_lista([X | XS], ...) :-  
    pred_lista(XS, ...),  
    ??? X.
```

Quando a lista não é vazia, pode ser necessário criar outros casos.

Projete um predicado `tamanho(L, T)` que é verdadeiro se a quantidade de elementos na lista `L` é `T`. (Veja o predicado pré-definido `length/2`).

```
:- use_module(library(plunit)).

%% tamanho(+XS, ?T) is semidet
%
% Verdadeiro se o tamanho de XS é T.

:- begin_tests(tamanho).

test(t0) :- tamanho([], 0).
test(t1) :- tamanho([4], 1).
test(t2, T == 2) :- tamanho([7, 2], T).

:- end_tests(tamanho).
```

```
tamanho([_ | XS], T) :-  
    tamanho(XS, T0),  
    T is T0 + 1.  
  
tamanho([], 0).
```

```
% Leitura em português do predicado

% O tamanho da lista [ _ | XS ] é T se
%   T0 é o tamanho da lista XS e
%   T é T0 + 1
tamanho([_ | XS], T) :-
    tamanho(XS, T0),
    T is T0 + 1.

% O tamanho da lista [] é 0.
tamanho([], 0).
```

Resultado dos testes

```
?- run_tests(tamanho).  
% PL-Unit: tamanho ... done  
% All 3 tests passed  
true.
```

Projete um predicado `kesimo(XS, K, N)` que é verdadeiro se `N` é o k -ésimo elemento da lista `XS`. (Veja o predicado pré-definido `nth0/3`)

Exemplo - k -ésimo

```
%% kesimo(+XS, +K, ?N) is semidet
%
% Verdadeiro se N é o K-ésimo elemento da lista XS.

:- begin_tests(kesimo).

test(t0) :- kesimo([5, 3, 10], 0, 5).
test(t1) :- kesimo([5, 3, 10], 1, 3).
test(t2, N == 10) :- kesimo([5, 3, 10], 2, N).
test(t4, fail) :- kesimo([5, 3, 10], 4, _).

:- end_tests(kesimo).
```

```
kesimo([X | _], 0, X).
```

```
kesimo([_ | XS], K, X) :-  
    K > 0,  
    K0 is K - 1,  
    kesimo(XS, K0, X).
```

Note que fizemos, mais ou menos, a combinação de modelos de lista e número natural


```
% Leitura em português do predicado

% X é 0-ésimo elemento da lista [X | _].
kesimo([X | _], 0, X).

% X é o K-ésimo elemento da lista [_ | XS] se
%   K > 0 e
%   K0 é K - 1 e
%   X é o K0-ésimo elemento da lista XS
kesimo([_ | XS], K, X) :-
    K > 0,
    K0 is K - 1,
    kesimo(XS, K0, X).
```

Exemplo - k -ésimo

Resultado dos testes

```
?- run_tests(kesimo).
```

```
% PL-Unit: kesimo
```

```
Warning: /home/malbarbo/desktop/x.pl:39:
```

```
    PL-Unit: Test t0: Test succeeded with choicepoint
```

```
Warning: /home/malbarbo/desktop/x.pl:40:
```

```
    PL-Unit: Test t1: Test succeeded with choicepoint
```

```
Warning: /home/malbarbo/desktop/x.pl:41:
```

```
    PL-Unit: Test t2: Test succeeded with choicepoint
```

```
. done
```

```
% All 4 tests passed
```

```
true.
```

(What!?)

Os testes (por padrão) esperam que o predicado não ofereça escolha mas depois de ser satisfeito uma vez, o predicado `kesimo` está oferecendo a possibilidade de ressatisfação, ou seja, ele é não determinístico

```
?- kesimo([5, 3, 10], 1, 3).  
true ;  
false.
```

Porque o predicado **tamanho** não funcionou desse forma?

Teoricamente o predicado **tamanho** também deveria funcionar dessa problema, isto porque após a consulta ser satisfeita unificando com a primeira cláusula do predicado **tamanho**, o Prolog deveria oferecer a possibilidade de continuar a busca e tentar a unificação com a segunda cláusula, o que criaria o ponto de escolha.

Porque o predicado `tamanho` não funcionou desse forma?

Isto não aconteceu porque o SWI-prolog faz uma otimização. Ele só faz a busca entre as cláusulas do predicado que tenham o primeiro argumento “compatível” com a consulta

- Se o primeiro argumento da consulta é uma constante, ele tenta as cláusulas que o primeiro argumento seja a mesma constante da consulta ou uma variável
- Se o primeiro argumento da consulta é uma variável, ele tenta todas as cláusulas
- Se o primeiro argumento da consulta é uma estrutura, ele tenta as cláusulas que o primeiro argumento seja uma estrutura com o mesmo functor da estrutura da consulta ou uma variável

Considerando a definição de tamanho

```
tamanho([_ | XS], T) :-  
    tamanho(XS, T0),  
    T is T0 + 1.
```

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

Observamos que o primeiro argumento da primeira cláusula é a estrutura '.', e o primeiro argumento da segunda cláusula é a constante [].

Seguindo a otimização do SWI-Prolog, em uma consulta tamanho com o primeiro argumento [], o interpretador tentará a unificação apenas com a segunda cláusula. Em uma consulta com uma lista não vazia como primeiro argumento, o interpretador tentará a unificação apenas com a primeira cláusula.

Vamos ver o que acontece com a definição de `kesimo`

Considerando a definição

```
kesimo([X | _], 0, X).
```

```
kesimo([_ | XS], K, X) :-  
    K > 0,  
    K0 is K - 1,  
    kesimo(XS, K0, X).
```

Neste caso o primeiro argumento das duas cláusulas é a estrutura `'.'`. Isto implica que em qualquer consulta que o primeiro argumento seja uma lista, o interpretador tentará as duas cláusulas, o que pode gerar um ponto de escolha.

Conceitualmente o predicado `kesimo` é semi determinístico, mas a nossa implementação é não determinística, como resolver essa situação?

O Prolog tem o operador de corte (!) que pode ser usado, entre outras coisas, para confirmar uma escolha e evitar que o interpretador faça outras tentativas

```
% usamos o operador de corte para confirmar a escolha,  
% desta forma, se uma consulta for satisfeita unificando com  
% a primeira cláusula, a segunda não será considerada  
kesimo([X | _], 0, X) :- !.
```

```
kesimo([_ | XS], K, X) :-  
    K > 0,  
    K0 is K - 1,  
    kesimo(XS, K0, X).
```


Veremos em outro momento todos os usos e o funcionamento detalhado do operador de corte.

O operador de corte não faz parte do paradigma lógico.

Outra alternativa seria colocar o **K** como primeiro argumento. Porque isto também resolveria o problema?

Projete um predicado `comprimida(XS, YS)` que é verdadeiro se lista `YS` é a lista `XS` comprimida, isto é, sem elementos repetidos consecutivos.

Exemplo - comprimida

```
% comprimida(+XS, ?YS) is semidet
%
% Verdadeiro se XS comprimida é YS, isto é,
% sem elementos repetidos consecutivos.

:- begin_tests(comprimida).

test(t0) :- comprimida([], []).
test(t1) :- comprimida([x], [x]).
test(t2) :- comprimida([3, 4, 4, 1, 5, 5], [3, 4, 1, 5]).
test(t2, X == [4, 1, 5]) :- comprimida([4, 4, 4, 1, 5, 5], X).
test(t2, X == [3, 4, 1, 5]) :- comprimida([3, 4, 4, 1, 5, 5], X).

:- end_tests(comprimida).
```

```
comprimida([], []).
```

```
comprimida([X], [X]).
```

```
comprimida([X, X | XS], YS) :-  
    comprimida([X | XS], YS).
```

```
comprimida([X, Y | XS], [X | YS]) :-  
    dif(X, Y),  
    comprimida([Y | XS], YS).
```

Usamos o modelo para listas. Caso a lista não seja vazia, existem duas alternativas, o primeiro elemento é repetido ou não.

Resultado dos testes

```
?- run_tests(comprimida).
```

```
% PL-Unit: comprimida .
```

```
Warning: /home/malbarbo/desktop/x.pl:71:
```

```
    PL-Unit: Test t1: Test succeeded with choicepoint
```

```
Warning: /home/malbarbo/desktop/x.pl:72:
```

```
    PL-Unit: Test t2: Test succeeded with choicepoint
```

```
done
```

```
% All 3 tests passed
```

```
true.
```

Solução usando corte para tornar o predicado semi determinístico

```
comprimida([], []) :- !.
```

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

```
comprimida([X, X | XS], YS) :-  
    comprimida([X | XS], YS), !.
```

```
comprimida([X, Y | XS], [X | YS]) :-  
    dif(X, Y),  
    comprimida([Y | XS], YS).
```

Defina um predicado `membro(X, XS)` que é verdadeiro se `X` é membro de `XS`. Defina uma versão que seja não determinística e outra que seja semi determinística. (Veja os predicados pré-definido `member/2` e `memberchk/2`)

```
% membro(?X, ?XS) is nondet
%
% Verdadeiro se X é um elemento de XS.

:- begin_tests(membro).

test(t0, nondet) :- membro(1, [1, 1, 3, 7]).
test(t1, nondet) :- membro(3, [1, 3, 7]).
test(t2, nondet) :- membro(7, [1, 3, 7]).
test(t3, all(X == [1, 3, 7])) :- membro(X, [1, 3, 7]).

:- end_tests(membro).
```



```
membro(X, [X | _]).  
membro(X, [_ | XS]) :-  
    membro(X, XS).
```

Exemplos

```
?- membro(X, [1, 3, 7]).
```

```
X = 1 ;
```

```
X = 3 ;
```

```
X = 7 ;
```

```
false.
```

```
?- membro(X, L).
```

```
L = [X|_G279] ;
```

```
L = [_G67, X|_G279] ;
```

```
L = [_G67, _G69, X|_G279] ;
```

```
...
```

Podemos observar que o predicado **membro** é não determinístico.

- Para testar predicados não determinísticos usados o argumento **nondet**
- Para testar todas as respostas de um predicado não determinísticos usamos o termo **all**
- Como definir uma versão semi determinística deste predicado?

Usando o operador de corte

```
% membrochk(+X, ?XS) is semidet
%
% Verdadeiro se X é um elemento de XS.

:- begin_tests(membrochk).

test(t0) :- membrochk(1, [1, 3, 7]).
test(t1) :- membrochk(7, [1, 3, 7]).
test(t2, X == 1) :- membrochk(X, [1, 3, 7]).
test(t3, fail) :- membrochk(5, [1, 3, 7]).

:- end_tests(membrochk).
```

Usando o corte

```
membrochk(X, [X | _]) :- !.  
membrochk(X, [_ | XS]) :-  
    membrochk(X, XS).
```

Usando o predicado pré-definido **once**

```
membrochk(X, XS) :-  
    once(membro(X, XS)).
```

Defina um predicado `concatenacao(XS, YS, ZS)` que é verdadeiro se `ZS` é a concatenação de `XS` com `YS`. (Veja o predicado pré-definido `append/3`)

Exemplo - concatenação

```
% concatenacao(?XS, ?YS, ?ZS) is nondet
%
% Verdadeiro se ZS é a concatenação de XS com YS.

:- begin_tests(concatenacao).
test(t0) :- concatenacao([1, 2], [3, 4, 5], [1, 2, 3, 4, 5]).
test(t1, XS == [1, 2, 4]) :- concatenacao(XS, [3], [1, 2, 4, 3]).
test(t2, YS == [4, 3]) :- concatenacao([1, 2], YS, [1, 2, 4, 3]).
test(t3, all([XS, YS] == [
    [], [1, 2, 3]],
    [[1], [2, 3]],
    [[1, 2], [3]],
    [[1, 2, 3], []]])) :-
    concatenacao(XS, YS, [1, 2, 3]).
:- end_tests(concatenacao).
```

```
concatenacao([], YS, YS).
```

```
concatenacao([X | XS], YS, [X | XSYS]) :-  
    concatenacao(XS, YS, XSYS).
```


Resultado dos testes

```
?- run_tests(concatenacao).  
% PL-Unit: concatenacao .  
Warning: /home/malbarbo/desktop/ex_dados.pl:46:  
    PL-Unit: Test t1: Test succeeded with choicepoint  
.. done  
% All 4 tests passed  
true.
```

Porque?

Na consulta `concatenacao(XS, [3], [1, 2, 4, 3])` são testadas as duas cláusulas, gerando a escolha.

Como podemos resolver essa situação?

- Adicionar o operador de corte na primeira cláusula faz com que o teste `t3` falhe ...
- Adicionar `nondet` ao teste `t1`

Exemplo - concatenação

```
% concatenacao(?XS, ?YS, ?ZS) is nondet
%
% Verdadeiro se ZS é a concatenação de XS com YS.

:- begin_tests(concatenacao).
test(t0) :- concatenacao([1, 2], [3, 4, 5], [1, 2, 3, 4, 5]).
test(t1, [nondet, XS == [1, 2, 4]]) :-
    concatenacao(XS, [3], [1, 2, 4, 3]).
test(t2, YS == [4, 3]) :- concatenacao([1, 2], YS, [1, 2, 4, 3]).
test(t3, all(p(XS, YS) == [
    p([], [1, 2, 3]),
    p([1], [2, 3]),
    p([1, 2], [3]),
    p([1, 2, 3], [])])) :- concatenacao(XS, YS, [1, 2, 3]).
:- end_tests(concatenacao).
```

Árvore binária

Uma árvore binária é:

- `nil`; ou
- `t(X, L, R)` onde `X` é o elemento raiz e `L` é a sub árvore a esquerda e `R` é a sub árvore a direita

Modelo

```
pred_arvore_binaria(nil, ...) :- ???.
```

```
pred_arvore_binaria(t(X, L, R), ...) :-  
  pred_arvore_binaria(L, ...),  
  pred_arvore_binaria(R, ...),  
  ??? X.
```

Defina um predicado $\text{altura}(T, H)$ que é verdadeiro se H é altura da árvore binária T . A altura de uma árvore binária é a distância entre a raiz e o seu descendente mais afastado. Uma árvore com um único nó tem altura 0.

Otimizações

Assim como o Racket, o Prolog faz otimizações das chamadas recursivas em cauda.

As vezes é necessário utilizar acumuladores para transformar uma recursão em recursão em cauda.

Uma outra técnica de otimização comum em Prolog é a utilização de diferenças de listas.

- Exemplo tamanho de uma lista.
- Exemplo reverso de uma lista.

- Exemplo concatenação de listas.

Referências

Referências

- Capítulo 3 e sessão 7.5 do livro Programming in Prolog
- Capítulo 4 da apostila Paradigmas de programação - Prolog
- Capítulos 4 e 6 do livro Learn Prolog Now