Dados compostos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Listas

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

Listas

- Exemplos

```
?- L0 = vazio, L1 = cons(3, vazio),
       L2 = cons(3, cons(4, vazio)).
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 elemento 3 e 4

Listas

- Vamos definir um predicado para verificar se um termo é uma lista

Listas

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

lista(vazia).

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

Listas

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

Listas

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

Listas

- O Prolog já "entende" uma definição de lista semelhante a nossa
  - `[]` ao invés de `vazia`
  - `'.'` ao invés de `cons`

Atenção

Nas versões mais recentes do SWI-Prolog o termo `'[|]'` é usando ao invés de `'.'`, mas a ideia é a mesma

Listas

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

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

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

Listas

- 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 definir listas

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

Composição

- Podemos utilizar uma lista já existente para definir 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].
```

Composição

- A notação `[ A | B ]` é equivalente a `'.'(A, B)`

```
?- X = '.'(3, '.'(4, [])),
   Y = [3 | [4 | []]],
   Z = [3, 4].
X = Y, Y = Z, Z = [3, 4].
```

Decomposição

- 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

Decomposição

```
?- '.'(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].
```

Decomposição

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

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

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

Receita de projeto

- Para definir muitos tipos de predicados podemos seguir as mesmas receitas de projeto que usávamos para escrever funções em Racket
- De forma semelhante ao Racket, para cada tipo de dado, usamos um modelo
  - Para definir alguns tipos de predicado não tem receita de projeto
  - Nestes casos, a experiência ajuda bastante

Receita de projeto para listas

- 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

Exemplo 3.1

Defina 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`).

Exemplo 3.1

```
:- 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).
```

Exemplo 3.1

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

tamanho([], 0).
```

Exemplo 3.1

```
% Leitura em português do predicado

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

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

Exemplo 3.1

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

Exemplo 3.2

Defina 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 3.2

```
%% 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, _).

% Usamos o argumento [fail] para testar que o predicado falha

:- end_tests(kesimo).
```

Exemplo 3.2

```
% Fizemos, mais ou menos, a combinação de
% modelos de lista e número natural

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

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

Exemplo 3.2

```
% 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 3.2

```
?- 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!?

Predicados (semi) determinísticos

Predicados (semi) determinísticos

- Por padrão, os testes 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.
```

Predicados (semi) determinísticos

- Porque o predicado `tamanho` não apresentou este problema?
- Teoricamente o predicado `tamanho` também deveria apresentar este 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

Predicados (semi) determinísticos

- Isto não acontece 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 cláusula 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 cláusula que o primeiro argumento seja uma estrutura com o mesmo functor da estrutura da consulta ou uma variável

Predicados (semi) determinísticos

- Considerando a definição do predicado `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`

Predicados (semi) determinísticos

- 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 este problema?

Predicados (semi) determinísticos

- 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).
```

Predicados (semi) determinísticos

- Veremos em outro momento todos os usos e o funcionamento detalhado do operador de corte
- Observe que o operador de corte não faz parte do paradigma lógico, mas é uma adição útil
- Outra alternativa seria colocar o `K` como primeiro argumento. Porque isto também resolveria o problema?

Exemplo 3.3

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

Exemplo 3.3

```
% 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, 3, 3, 4, 4, 3, 5, 5, 5],
                       [3, 4, 3, 5]).

:- end_tests(comprimida).
```

Exemplo 3.3

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

comprimida([], []).

% Elemento repetido
comprimida([X | XS], YS) :-
    comprimida(XS, YS),
    [X | _] = YS.

% Elemento distinto
comprimida([X | XS], [X | YS]) :-
    comprimida(XS, YS),
    [X | _] \= YS.
```

Exemplo 3.3

```
?- 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.
```

Exemplo 3.3

Solução usando corte para tornar o predicado semi determinístico

```
% comprimida(+XS, ?YS) is semidet
%
% Verdadeiro se XS comprimida é YS, isto é, sem elementos
% repetidos consecutivos.
comprimida([], []).
% Elemento repetido
comprimida([X | XS], YS) :-
    comprimida(XS, YS),
    [X | _] = YS,
    !.
% Elemento distinto
comprimida([X | XS], [X | YS]) :-
    comprimida(XS, YS),
    [X | _] \= YS.
```

Exemplo 3.3

Solução alternativa

```
% comprimida2(+XS, ?YS) is semidet
%
% Verdadeiro se XS comprimida é YS, isto é, sem elementos
% repetidos consecutivos.

:- begin_tests(comprimida2).

test(t0) :- comprimida2([], []).
test(t1) :- comprimida2([x], [x]).
test(t2) :- comprimida2([3, 3, 4, 4, 4, 3, 5, 5, 5],
                        [3, 4, 3, 5]).

:- end_tests(comprimida2).
```

Exemplo 3.3

```
comprimida2([], []).

comprimida2([X], [X]) :- !.

comprimida2([X, X | XS], YS) :-
    comprimida2([X | XS], YS),
    !.

comprimida2([X, Y | XS], [X | YS]) :-
    X \== Y,
    comprimida2([Y | XS], YS).
```

Exemplos

Variáveis instanciadas vs não instanciadas

- Vamos voltar ao predicado `tamanho`

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

tamanho([], 0).
```

- O que acontece na consulta?

```
?- tamanho(XS, 3).
```

Variáveis instanciadas vs não instanciadas

- Entra em laço. A consulta unifica com a primeira cláusula, a primeira meta unifica novamente com a primeira cláusula e assim por diante
- Neste caso o operador de corte sozinho não resolve o problema é necessário fazer duas definições
  - Uma para o caso que `XS` está instanciada e outro para o caso que `XS` não está instanciada
  - Usamos os predicados pré-definidos `nonvar` e `var`, respectivamente

Variáveis instanciadas vs não instanciadas

```
tamanho([], 0) :- !.

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

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

Variáveis instanciadas vs não instanciadas

- O que acontece nas consultas?

```
?- length(XS, T).
...
?- tamanho(XS, T).
...
```

- Como alterar a definição de `tamanho` para funcionar da mesma forma que `length`?

Exemplo 3.4

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`)

Exemplo 3.4

```
% 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).
```

Exemplo 3.4

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

Exemplo 3.4

```
?- 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] ;
...
```

Exemplo 3.4

- 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

Exemplo 3.4

```
% 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).
```

Exemplo 3.4

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

ou

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

Exemplo 3.5

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 3.5

```
% 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(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).
```

Exemplo 3.5

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

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

Exemplo 3.5

```
?- 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.
```

Exemplo 3.5

- Porque?
  - Na consulta `concatenacao(XS, [3], [1, 2, 4, 3])` são testadas as duas cláusulas, gerando a escolha
- Solução
  - Adicionar o operador de corte na primeira cláusula faz com que o teste `t3` falhe ...
  - Adicionar `nondet` ao teste `t1`

Exemplo 3.5

```
% 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).
```

Listas aninhadas

Listas aninhadas

- Uma lista aninhada é
  - `[]`; ou
  - `[X | XS]`, onde `X` não é uma lista aninhada e `XS` é uma lista aninhada; ou
  - `[X | XS]`, onde `X` e `XS` são listas aninhadas

Listas aninhadas - Modelo

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

pred_lista_aninhada([X | XS], ...) :-
    \+ is_list(X),
    pred_lista_aninhada(XS, ...),
    ??? X.

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

Exemplo 3.6

Defina um predicado `super_soma(XS, S)` que é verdadeiro se `S` é soma de todos os elementos da lista aninhada `XS`.

Exemplo 3.6

```
%% super_soma(XS, S) is semidet
%
%  Verdadeiro se S é a soma de todos elementos da
%  lista aninhada XS.

:- begin_tests(super_soma).

test(t0) :- super_soma([[], [], [[], []]], 0).
test(t1) :- super_soma([[1], [2, 3], [4, [5, 6], 7]], 28).
test(t2, S == 36) :-
    super_soma([[1, 2], 3, [4, [5, 6, [7]], 8]], S).

:- end_tests(super_soma).
```

Exemplo 3.6

```
super_soma([], 0).

super_soma([X | XS], S) :-
    \+ is_list(X), !,
    super_soma(XS, S1),
    S is X + S1.

super_soma([X | XS], S) :-
    super_soma(X, S1),
    super_soma(XS, S2),
    S is S1 + S2.
```

Árvore binária

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

Árvore binária - Modelo

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

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

Exemplo 3.7

Defina um predicado `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

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

Acumuladores

- Exemplo 3.8, tamanho de uma lista.
- Exemplo 3.9, reverso de uma lista.

Diferenças de listas

- Exemplo 3.10, 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