

Programação Declarativa

Paradigmas de Programação Avançados

Técnicas de Programação

Listas e recursão

Debugger

~~Aritmética~~

~~Meta-programação~~

~~Termos “abertos” (inc. listas de diferença)~~

Voltando aos nossos problemas...

Os mecanismos principais do Prolog são:

- A unificação
- A recursão
- O backtracking

Deve-se escrever programas que **usem** estes conceitos.

Coisas a fazer e coisas a evitar fazer

Fazer	Não fazer
$p(X, X).$	$p(A, B) :- A=B.$
$p(X, f(X)).$	$p(A, B) :- A=X, B=f(X).$
$p(a1, X) :- pa1(X).$ $p(a2, X) :- pa2(X).$ $p(a3, X) :- pa3(X).$	$p(A, X) :- A=a1, pa1(X).$ $p(A, X) :- X=a2, \dots$
$f(op(A), X) :- ok_p(A), !, p(X).$ $f(op(A), X) :- ok_q(A), !, q(X).$ \dots	$f(A, X) :- A=op(Z), ok_p(Z), !,$ $p(X).$ \dots

Listas

Uma lista é composta por um termo que pode ser

- A lista vazia `[]`, pronunciada “nil”
- Um *par* formado por uma **cabeça** (head) e uma **cauda** (tail).
 - `'.'(CABECA, CAUDA)`
 - `[CABECA | CAUDA]`
 - `[CABECA , .. CAUDA]`

Uma lista é um termo normal, mas com uma sintaxe exterior particular.

Predicado de *tipo* para listas:

```
lista([]).                % caso base
lista([_ | L]) :- lista(L). % caso recursivo
```

Inversão de lista com *acumulador*

Ideia: vamos construindo a lista inversa enquanto atravessamos a de entrada. Quando chegarmos ao fim, a que fomos construindo **já é** a lista invertida!

Código:

```
rev(L, R) :- rev(L, [], R).  
  
rev([], R, R).  
rev([A|B], X, R) :- rev(B, [A|X], R).
```

Modelo:

- Fazer predicado auxiliar (neste caso, rev/3)
- Auxiliar tem mais um parâmetro, dito de acumulação (neste caso, o 2º)
- Auxiliar tem **recursão terminal!**

Trace do rev/3

| ?- rev([1,2,3], X).

```
1 1 Call: rev([1,2,3],_22) ?
2 2 Call: rev([1,2,3],[],_22) ?
3 3 Call: rev([2,3],[1],_22) ?
4 4 Call: rev([3],[2,1],_22) ?
5 5 Call: rev([], [3,2,1],_22) ?
5 5 Exit: rev([], [3,2,1], [3,2,1]) ?
4 4 Exit: rev([3], [2,1], [3,2,1]) ?
3 3 Exit: rev([2,3], [1], [3,2,1]) ?
2 2 Exit: rev([1,2,3], [], [3,2,1]) ?
1 1 Exit: rev([1,2,3], [3,2,1]) ?
```

X = [3,2,1]

Regresso do clássico: **naïve reverse**

Trata-se de inverter uma lista.

Dois casos (lista vazia ou não vazia)

Código:

```
nrev([], []).  
nrev([X|A], B) :-  
    nrev(A, AR),  
    catena(AR, [X], B).      % costuma ser append/3
```

Usar com parcimónia!! Problemas:

- Recursão interna
- Predicado auxiliar “complexo” (catena/3)
- Reconstroi estruturas de dados ([X])

Execução do nrev/2

profundidade da chamada

número da chamada

```
| ?- nrev([1,2,3], X).
```

1	1	Call: nrev([1,2,3],_285) ?
2	2	Call: nrev([2,3],_354) ?
3	3	Call: nrev([3],_378) ?
4	4	Call: nrev([],_402) ?
4	4	Exit: nrev([],[]) ?
5	4	Call: catena([], [3],_430) ?
5	4	Exit: catena([], [3], [3]) ?
3	3	Exit: nrev([3], [3]) ?
6	3	Call: catena([3], [2],_459) ?
7	4	Call: catena([], [2],_446) ?
7	4	Exit: catena([], [2], [2]) ?
6	3	Exit: catena([3], [2], [3,2]) ?
2	2	Exit: nrev([2,3], [3,2]) ?
8	2	Call: catena([3,2], [1],_285) ?
9	3	Call: catena([2], [1],_503) ?
10	4	Call: catena([], [1],_530) ?
10	4	Exit: catena([], [1], [1]) ?
9	3	Exit: catena([2], [1], [2,1]) ?
8	2	Exit: catena([3,2], [1], [3,2,1]) ?
1	1	Exit: nrev([1,2,3], [3,2,1]) ?

X = [3,2,1]

Exemplo sem recursão terminal

Somar elementos numa lista

Recursão simples

```
% somalista(Is,SOMA)

somalista([], 0).
somalista([I|Is], S) :-
    somalista(Is, S0),
    S is I+S0.
```

Exemplo com recursão terminal

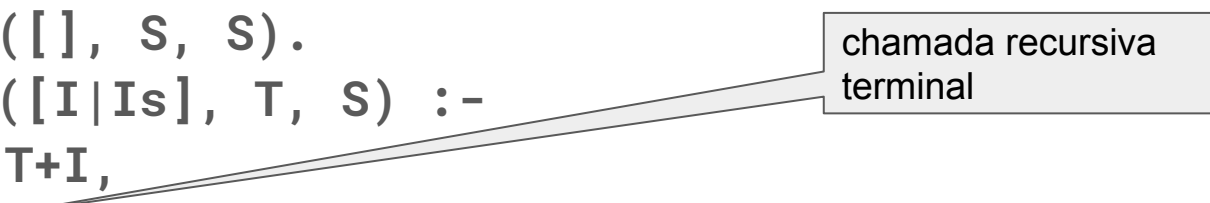
Com recursão terminal

- Predicado auxiliar
- Parâmetro acumulador

```
% somalista(Is,SOMA)
```

```
somalista(Is, S) :-  
    somalista(Is, 0, S).
```

```
somalista([], S, S).  
somalista([I|Is], T, S) :-  
    T1 is T+I,  
    somalista(Is, T1, S).
```



chamada recursiva
terminal

Ordenação ingênua

Digamos que a versão ordenada duma lista é uma qualquer permutação da mesma, desde que **seja** ordenada.

Precisamos então definir duas coisas:

- O que é uma *permutação*.
- Quais são as *condições de ordenação*.

Permutação

Formulação - P é uma permutação da lista L , se:

- L for vazia, sendo P também
- Se retirarmos (não-deterministicamente) um elemento X de L , a permutação será a lista que começa com X e continua com uma permutação do resto da lista L .

Tradução:

```
perm([], []).  
perm(L, [X|LP]) :-  
    sel(X, L, LX),  
    perm(LX, LP).
```

Verificação de ordenação

Uma lista é ordenada se todos os pares de elementos consecutivos estiverem por ordem.

Precisamos do predicado built-in que compara termos: `</2`.

Ficamos com isto:

```
ord([ ]).  
ord([_]).  
ord([A,B|X]) :- A<B, ord([B|X]).
```

Ordenação por geração de permutação ordenada

Dados os predicados que já definimos, podemos dizer isto assim:

```
psort(L, S) :- perm(L, S), ord(S).
```

Será pouco eficiente (tem de gerar todas as permutações até encontrar uma “boa”), mas é **muito simples** de definir, e é obviamente correto.

Ordenação por inserção

Usa a técnica do “acumulador”

Predicado auxiliar (**isort/3**)

Recursão terminal = iteração.

```
isort(I, S) :- isort(I, [], S).
```

```
isort([], S, S).
```

```
isort([X|Xs], SI, SO) :-  
    insord(X, SI, SX),  
    isort(Xs, SX, SO).
```

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

```
insord(X, [A|As], [X,A|As]) :- X=<A.
```

```
insord(X, [A|As], [A|AAs]) :-  
    X>A,  
    insord(X, As, AAs).
```


Execução do isort/2

```
| ?- isort([4,1,2], X).
  1      1      Call: isort([4,1,2],_285) ?
  2      2      Call: isort([4,1,2],[],_285) ?
  3      3      Call: insord(4,[],_380) ?
  3      3      Exit: insord(4,[],[4]) ?
  4      3      Call: isort([1,2],[4],_285) ?
  5      4      Call: insord(1,[4],_433) ?
  6      5      Call: 1=<4 ?
  6      5      Exit: 1=<4 ?
  5      4      Exit: insord(1,[4],[1,4]) ?
  7      4      Call: isort([2],[1,4],_285) ?
  8      5      Call: insord(2,[1,4],_513) ?
  9      6      Call: 2=<1 ?
  9      6      Fail: 2=<1 ?
  9      6      Call: 2>1 ?
  9      6      Exit: 2>1 ?
 10      6      Call: insord(2,[4],_500) ?
 11      7      Call: 2=<4 ?
 11      7      Exit: 2=<4 ?
 10      6      Exit: insord(2,[4],[2,4]) ?
   8      5      Exit: insord(2,[1,4],[1,2,4]) ?
 12      5      Call: isort([], [1,2,4],_285) ?
 12      5      Exit: isort([], [1,2,4], [1,2,4]) ?
   7      4      Exit: isort([2],[1,4],[1,2,4]) ?
   4      3      Exit: isort([1,2],[4],[1,2,4]) ?
   2      2      Exit: isort([4,1,2],[], [1,2,4]) ?
   1      1      Exit: isort([4,1,2],[1,2,4]) ?
```

X = [1,2,4] ?

Execução do isort/2

```
| ?- isort([4,1,2,5,3], X).
```

```
X = [1,2,3,4,5] ?
```

```
| ?- isort([4,1,2,5,3,a], X).
```

```
uncaught exception:  
error(type_error(evaluable,a/0),(<=)/2)
```

```
| ?-
```

Quicksort

Quicksort semelhante, mas vamos partir a lista em duas “metades”, conforme sejam menores or maiores que um elemento “pivot” (o primeiro).

Predicado auxiliar **part/4**.
Divide uma lista em duas “metades”.

```
qsort(L, S) :- qsort(L, [], S).
```

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

```
qsort([X|Xs], L0, L) :-  
    part(Xs, X, MEN, MAI),  
    qsort(MAI, L0, L1),  
    qsort(MEN, [X|L1], L).
```

```
part([], _, [], []).
```

```
part([X|L], Y, [X|L1], L2) :-
```

```
    X <= Y, !,
```

```
    part(L, Y, L1, L2).
```

```
part([X|L], Y, L1, [X|L2]) :-
```

```
    part(L, Y, L1, L2).
```

quicksort

```
qsort(L, S) :- qsort(L, [], S).
```

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

```
qsort([X|Xs], L0, L) :-  
    particao(Xs, X, MEN, MAI),  
    qsort(MAI, L0, L1),  
    qsort(MEN, [X|L1], L).
```

```
particao([], _, [], []).
```

```
particao([X|L], Y, [X|L1], L2) :-  
    X <= Y, !,  
    particao(L, Y, L1, L2).
```

```
particao([X|L], Y, L1, [X|L2]) :-  
    particao(L, Y, L1, L2).
```

Execução do qsort

```
| ?- spy(qsort).  
Spypoint placed on qsort/2  
Spypoint placed on qsort/3  
The debugger will first leap -- showing spyoints (debug)
```

```
yes  
{debug}  
| ?-
```

Execução do qsort

```
| ?- qsort([1,5,2,4,3], X).  
+   1   1   Call: qsort([1,5,2,4,3],_289) ? 1  
+   2   2   Call: qsort([1,5,2,4,3],[],_289) ? 1  
+   8   3   Call: qsort([5,2,4,3],[],_527) ? 1  
+  16   4   Call: qsort([],[],_741) ? 1  
+  16   4   Exit: qsort([],[],[]) ? 1  
+  17   4   Call: qsort([2,4,3],[5],_769) ? 1  
+  21   5   Call: qsort([4,3],[5],_879) ? 1  
+  25   6   Call: qsort([], [5],_985) ? 1  
+  25   6   Exit: qsort([], [5], [5]) ? 1  
+  26   6   Call: qsort([3], [4,5],_1013) ? 1  
+  28   7   Call: qsort([], [4,5],_1065) ? 1  
+  28   7   Exit: qsort([], [4,5], [4,5]) ? 1  
+  29   7   Call: qsort([], [3,4,5],_1093) ? 1  
+  29   7   Exit: qsort([], [3,4,5], [3,4,5]) ? 1  
+  26   6   Exit: qsort([3], [4,5], [3,4,5]) ? 1  
+  21   5   Exit: qsort([4,3], [5], [3,4,5]) ? 1  
+  30   5   Call: qsort([], [2,3,4,5],_1123) ? 1  
+  30   5   Exit: qsort([], [2,3,4,5], [2,3,4,5]) ? 1  
+  17   4   Exit: qsort([2,4,3], [5], [2,3,4,5]) ? 1  
+   8   3   Exit: qsort([5,2,4,3], [], [2,3,4,5]) ? 1  
+  31   3   Call: qsort([], [1,2,3,4,5],_289) ? 1  
+  31   3   Exit: qsort([], [1,2,3,4,5], [1,2,3,4,5]) ? 1  
+   2   2   Exit: qsort([1,5,2,4,3], [], [1,2,3,4,5]) ? 1  
+   1   1   Exit: qsort([1,5,2,4,3], [1,2,3,4,5]) ? 1
```

X = [1,2,3,4,5]

(2 ms) yes

{debug}

| ?-