INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Computer Science Department

Ch 6 – More on Lists

Mireille Ducassé

Last revision April 2023

Hindsight

- The most important design pattern for list processing:

do_list([], <base result>).

sometimes
do_list([X], [<base_result>]).

do_list([Head | Tail], [Head_Res |Tail_Res]) :-
do_one(Head , Head_Res),
do_list(Tail, Tail_Res).

- End result is concatenated **at the end of the recursions**

Recursion

- Replaces iteration of imperative programming
- Much safer to program with
  - … once well understood ☺

Exercise 3.12: reverse/2

- **Using the previous pattern and predicate append/3**, write predicate reverse/2 that changes a list [a, b, c, d, e] into a list [e, d, c, b, a]

- This predicate is useful because Prolog only gives easy access to the head of a list
  - sometime what is interesting is at the end of a list

Exercise 3.12: naïve_reverse/2 (bis)

```
naive_reverse([], []).
naive_reverse([H|T], R):-
    naive_reverse(T, RT),
    append(RT, [H], R).
```

Exercise 3.13: Efficiency of append/3

- **Which goal will take the least time to execute**?

  ?- append([a, b, c, d, e, f, g, h, i], [j, k,l], R).

  ?- append([j, k, l], [a, b, c, d, e, f, g, h, i], R).

Ex. 3.12: Efficiency of append/3(bis)

- Which goal will take the least time to execute?

  ?- append([a, b, c, d, e, f, g, h, i], [j, k,l], R).

  9 recursive calls

  ?- append([j, k, l], [a, b, c, d, e, f, g, h, i], R).

  3 recursive calls

- The goal with the **shortest list as first argument will be the fastest** because the recursion works on the first argument

Ex 3.13: Efficiency of naïve reverse

- Why do you think that this version of reverse/2 is called “naïve” ?
- What is the source of inefficiency ?

```
[eclipse]: naive_reverse([a, b, c, d], L).
  (1) 1 CALL  naive_reverse([a, b, c, d], L)
  (2) 2 CALL  naive_reverse([b, c, d], _282)
  (3) 3 CALL  naive_reverse([c, d], _368)
  (4) 4 CALL  naive_reverse([d], _454)
  (5) 5 CALL  naive_reverse([], _540)
  (5) 5 EXIT  naive_reverse([], [])
  (6) 5 CALL  my_append([], [d], _454)
  (6) 5 EXIT  my_append([], [d], [d])
  (4) 4 EXIT  naive_reverse([d], [d])
  (7) 4 CALL  my_append([d], [c], _368)
  (8) 5 CALL  my_append([], [c], _995)
  (8) 5 EXIT  my_append([], [c], [c])
  (7) 4 EXIT  my_append([d], [c], [d, c])
  (3) 3 EXIT  naive_reverse([c, d], [d, c])
  (9) 3 CALL  my_append([d, c], [b], _282)
  (10) 4 CALL  my_append([c], [b], _1363)
  (11) 5 CALL  my_append([], [b], _1451)
  (11) 5 EXIT  my_append([], [b], [b])
  (10) 4 EXIT  my_append([c], [b], [c, b])
  (9) 3 EXIT  my_append([d, c], [b], [d, c, b])
  (2) 2 EXIT  naive_reverse([b, c, d], [d, c, b])
  (12) 2 CALL  my_append([d, c, b], [a], L)
  (13) 3 CALL  my_append([c, b], [a], _1883)
  (14) 4 CALL  my_append([b], [a], _1971)
  (15) 5 CALL  my_append([], [a], _2059)
  (15) 5 EXIT  my_append([], [a], [a])
  (14) 4 EXIT  my_append([b], [a], [b, a])
  (13) 3 EXIT  my_append([c, b], [a], [c, b, a])
  (12) 2 EXIT  my_append([d, c, b], [a], [d, c, b, a])
  (1) 1 EXIT  naive_reverse([a, b, c, d], [d, c, b, a])
```

trace

```
[eclipse]: naive_reverse([a, b, c, d], L).
  (1) 1 CALL  naive_reverse([a, b, c, d], L)
  (2) 2 CALL  naive_reverse([b, c, d], _282)
  (3) 3 CALL  naive_reverse([c, d], _368)
  (4) 4 CALL  naive_reverse([d], _454)
  (5) 5 CALL  naive_reverse([], _540)
  (5) 5 EXIT  naive_reverse([], [])
  (6) 5 CALL  my_append([], [d], _454)
  (6) 5 EXIT  my_append([], [d], [d])
  (4) 4 EXIT  naive_reverse([d], [d])
  (7) 4 CALL  my_append([d], [c], _368)
  (8) 5 CALL  my_append([], [c], _995)
  (8) 5 EXIT  my_append([], [c], [c])
  (7) 4 EXIT  my_append([d], [c], [d, c])
  (3) 3 EXIT  naive_reverse([c, d], [d, c])
  (9) 3 CALL  my_append([d, c], [b], _282)
  (10) 4 CALL  my_append([c], [b], _1363)
  (11) 5 CALL  my_append([], [b], _1451)
  (11) 5 EXIT  my_append([], [b], [b])
  (10) 4 EXIT  my_append([c], [b], [c, b])
  (9) 3 EXIT  my_append([d, c], [b], [d, c, b])
  (2) 2 EXIT  naive_reverse([b, c, d], [d, c, b])
  (12) 2 CALL  my_append([d, c, b], [a], L)
  (13) 3 CALL  my_append([c, b], [a], _1883)
  (14) 4 CALL  my_append([b], [a], _1971)
  (15) 5 CALL  my_append([], [a], _2059)
  (15) 5 EXIT  my_append([], [a], [a])
  (14) 4 EXIT  my_append([b], [a], [b, a])
  (13) 3 EXIT  my_append([c, b], [a], [c, b, a])
  (12) 2 EXIT  my_append([d, c, b], [a], [d, c, b, a])
  (1) 1 EXIT  naive_reverse([a, b, c, d], [d, c, b, a])
```

trace

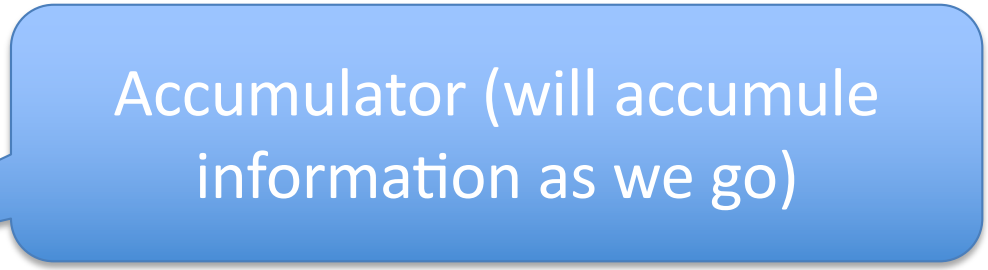
At each recursive call of naïve_reverse/2 append/3 has to traverse a longer list

Hence the cost of the execution is not linear.

Another design pattern: using an accumulator

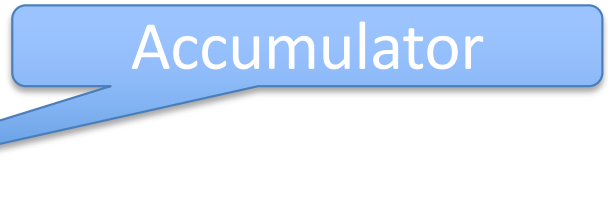
- An accumulator is an extra argument
  - a list (or a number)
- It stores partial result(s) **before recursive calls**
  - not waiting for the end of the recursions
- The pattern requires an extra predicate
  - with one more argument: the accumulator
  - in general initialized to empty list (or to 0)

reverse/2 with accumulator




```
reverse(L1,L2):-
      accReverse(L1, [ ], L2).

accReverse([ ], L, L).
accReverse([H|T], Acc, Rev):-
      accReverse(T, [H|Acc], Rev).
```



List: [a,b,c,d] Accumulator: []
List: [b,c,d] Accumulator: [a]
List: [c,d] Accumulator: [b,a]
List: [d] Accumulator: [c,b,a]
List: [] Accumulator: [d,c,b,a]

trace 1/2

This time the cost of the execution is linear with respect to the number of elements of the list.

```
[eclipse]: my_reverse([a, b, c, d], L).
  (1) 1 CALL  my_reverse([a, b, c, d], L)
  (2) 2 CALL  accReverse([a, b, c, d], [], L)
  (3) 3 CALL  accReverse([b, c, d],    [a], L)
  (4) 4 CALL  accReverse([c, d],       [b, a], L)
  (5) 5 CALL  accReverse([d],          [c, b, a], L)
  (6) 6 CALL  accReverse([],           [d, c, b, a], L)
  (6) 6 EXIT  accReverse([],           [d, c, b, a], [d, c, b, a])
  (5) 5 EXIT  accReverse([d],          [c, b, a], [d, c, b, a])
  (4) 4 EXIT  accReverse([c, d],       [b, a], [d, c, b, a])
  (3) 3 EXIT  accReverse([b, c, d],    [a], [d, c, b, a])
  (2) 2 EXIT  accReverse([a, b, c, d], [], [d, c, b, a])
  (1) 1 EXIT  my_reverse([a, b, c, d], [d, c, b, a])


L = [d, c, b, a]
```

trace 2/2

It also works with first list not instantiated

```
[eclipse]: my_reverse(L, [a, b, c, d]).
  (1) 1 CALL  my_reverse(L, [a, b, c, d])
  (2) 2 CALL  accReverse(L,    [], [a, b, c, d])
  (2) 2 NEXT  accReverse(L,    [], [a, b, c, d])
  (3) 3 CALL  accReverse(_369, [_368], [a, b, c, d])
  (3) 3 NEXT  accReverse(_369, [_368], [a, b, c, d])
  (4) 4 CALL  accReverse(_459, [_458, _368], [a, b, c, d])
  (4) 4 NEXT  accReverse(_459, [_458, _368], [a, b, c, d])
  (5) 5 CALL  accReverse(_549, [_548, _458, _368], [a, b, c, d])
  (5) 5 NEXT  accReverse(_549, [_548, _458, _368], [a, b, c, d])
  (6) 6 CALL  accReverse(_639, [_638, _548, _458, _368], [a, b, c, d])
  (6) 6 *EXIT  accReverse([],  [a, b, c, d], [a, b, c, d])
  (5) 5 *EXIT  accReverse([a], [b, c, d], [a, b, c, d])
  (4) 4 *EXIT  accReverse([b, a], [c, d], [a, b, c, d])
  (3) 3 *EXIT  accReverse([c, b, a], [d], [a, b, c, d])
  (2) 2 *EXIT  accReverse([d, c, b, a], [], [a, b, c, d])
  (1) 1 *EXIT  my_reverse([d, c, b, a], [a, b, c, d])


L = [d, c, b, a]
```

Back to Ex 3.11 Georgian routing: update 3


```
road(tbilisi, rustavi).
road(tbilisi, mtskheta).
road(tbilisi, gurjaani).
road(tbilisi, akhmeta).
road(mtskheta, gori).
road(gurjaani, telavi).
road(akhmeta, telavi).
road(gori, khashuri).
```

- Write a version that **takes into account that roads go both ways.**

```
?- route3(tbilisi, telavi, P).
?-route3(tbilisi, Y, P).
?-route3(telavi, Y, P).
?-route3(Y, Y, P.
```

- **use an accumulator and check that the routes do not loop**
- **use member/2 and not/1**

Accumulator

The top predicate is

```
route3(X, Y, Path) :-
        route3_do(X, Y, [X], Path).
```

There are 92 solutions for `?-route3(X, Y, P)`

Back to Ex 3.11
Georgian routing: update 3 (bis)


```
route3(X, Y, Path) :-
    route3_do(X, Y, [X], Path).

route3_do(X, Y, Done, []) :-
    road_sym(X, Y),
    not member(Y, Done).
route3_do(X, Y, Done, [Z | Path]) :-
    road_sym(X, Z),
    not member(Z, Done),
    route3_do(Z, Y, [Z | Done], Path).

road_sym(X, Y) :-
    road(X, Y).
road_sym(X, Y) :-
    road(Y, X).
```

- A version that **takes into account that roads go both ways.**

```
?-
route3(tbilisi,telavi,P)
.
?-route3(tbilisi,Y,P).
?-route3(telavi,Y,P).
?-route3(X, Y, P).
```

Hindsight

- Why is there no problem with
  not(member(…))
  even when the cities are initially not ground ?

Hindsight (bis)

```
route3_do(X, Y, Done, []) :-
        road_sym(X, Y),
        not member(Y, Done).
route3_do(X, Y, Done, [Z  | Path]) :-
        road_sym(X, Z),
        not member(Z, Done),
        route3_do(Z, Y, [Z  | Done], Path).
```

- Why is there no problem with
  not(member(…))
  even when the cities are initially not ground ?
- because road_sym/2 calls road/2 that returns always ground arguments.
- and the first value of the accumulator, [X], is instantiated by unification after the success of road_sym/2
- Hence both arguments of member are ground at call time