Some list processing examples

Marek Sergot
Department of Computing
Imperial College, London

Autumn 2012

1 Example 1

`last(List, X)` – `X` is the last element of the list `List`

```
last(List, X) :- append(_, [X], List).

append([], X, X).
append([U|X], Y, [U|Z]) :- append(X, Y, Z).
```

Trace the computation:

```
?-  last([a,b,c], X).
|
?-  append(_1, [X], [a,b,c]).
|      _1 = [a|_2]
?-  append(_2, [X], [b,c]).
|      _2 = [b|_3]
?-  append(_3, [X], [c]).
/|     _3 = [], X = c
/ ?   (empty)    Solution:  X = c
|
 \
  |    _3 = [c|_4]
  ?-  append(_4, [X], [])
  |
      (fails)
```


Formulated directly, without reference to `append`:

```
last([X], X).
last([_|Rest], X) :-
  last(Rest, X).
```

Compare the computation:

```
?-  last([a,b,c], X).
|
?-  last([b,c], X).
|
?-  last([c], X).
/|      X = c
/ ?   (empty)    Solution:  X = c
|
 \
  |
  ?-  last([], X)
  |
      (fails)
```

Alternatively (very slightly better)

```
last([X], X).
last([_,U|Rest], X) :-
  last([U|Rest], X).
```

In this last version, the two clauses for `last` are mutually exclusive, and cover the two (mutually exclusive) cases of a list with exactly one element, and a list with at least two elements. This is usually better if it can be done, though it makes very little difference in this example.

2 Example 2

Given a list of terms of the form `(Person, Sex, Age)`, find:

- maximum Age
- average Age
- both (in one pass)

2.1 First attempt

(not tail recursive)

If you are used to writing procedural programs only and you like to think *procedurally*, you could write:

```
max_age(List, Max) :-
   List = [X],
   X = (_, _, Age),
   Max = Age.
max_age(List, Max) :-
   List = [X|Rest],
   X = (_, _, Age),
   max_age(Rest, MaxRest)
   bigger_of(MaxRest, Age, Max).

bigger_of(X, Y, Y) :- Y > X.
bigger_of(X, Y, X) :- Y =< X.
```

Read procedurally the two clauses for `max_age` are:

- if `List` has exactly one element `X`:
  - get the `Age` parameter from `X`
  - set (unify) `Max` to `Age`, and return;
- split `List` into head element `X` and tail `Rest`:
  - get the `Age` parameter from `X`
  - compute the max age of `Rest`; call it `MaxRest`
  - compute `Max` as the bigger of `Age` and `MaxRest`, and return.

This works, but it's unnecessary to have separate unify/split/set calls as above. Instead use a *pattern directed* style of programming/computation and write:

Pattern-directed style

```
max_age([ (_, _, Age) ], Age).
max_age([ (_, _, Age) |Rest], Max) :-
   max_age(Rest, MaxRest)
   bigger_of(MaxRest, Age, Max).
```

Finally (cf. `last`) you could make the two clauses mutually exclusive:

```
max_age([ (_, _, Age) ], Age).
max_age([ (_, _, Age), Y|Rest], Max) :-
   max_age(Rest, MaxRest)
   bigger_of([Y|MaxRest], Age, Max).
```

Or you might find the following easier to read.

```
max_age([X], Age) :-
   X = (_, _, Age).
max_age([X,Y|Rest], Max) :-
   X = (_, _, Age),
   max_age(Rest, MaxRest)
   bigger_of([Y|MaxRest], Age, Max).
```

(I prefer the first version.)

Note (1) We want `max_age(List, Age)` to fail if `List` is empty. Some Prolog hackers are tempted to stick in a first clause like this:

```
max_age([ ], _) :- !, fail.
```

This is pointless. (I don't know why some people do it.)

Note (2) Some Prolog programmers might want to make `bigger_of` slightly more efficient by using a 'cut', like this:

```
bigger_of(X, Y, Y) :- Y > X, !.
bigger_of(X, _, X).
```

If `Y > X` succeeds we don't need to try the second clause; hence the 'cut'. If `Y > X` fails and we have to try the second clause, we don't need to test `Y =< X`; we already know that `Y > X` fails. You have to decide for yourself whether you think the saving is worth it.

2.2 Second attempt

(much better – tail recursive)

```
max_age([X|Rest], Max) :-
   max_age([X|Rest], 0, Max).
% the above fails if input List is empty


% ------ max_age/3

max_age([], MaxSofar, MaxSofar).

max_age([(_, _, Age)|Rest], MaxSofar, Max) :-
   bigger_of(MaxSofar, Age, NextMax),
   max_age(Rest, NextMax, Max).
```

Some programmers put a 'cut' after the call to `bigger_of`. This doesn't affect the answer but it helps some Prolog compilers to detect an opportunity to optimise ('last call optimisation' – described in Sicstus manual). I'm not sure the Sictus compiler needs it, but anyway there is no benefit in a small example like this.

If `bigger_of` is not used anywhere else one can avoid defining it by using Prolog's `...-> ...; ...` ('if ... then ... else ...') construction, like this:

```
max_age([], MaxSofar, MaxSofar).

max_age([(_, _, Age)|Rest], MaxSofar, Max) :-
   (
    Age > MaxSofar
    ->
       NextMax = Age
    ;
       NextMax = MaxSofar
   ),
   max_age(Rest, NextMax, Max).
```


2.3 Compute average age

Assuming no duplicate `Person` entries in `List` we could write

```
average_age(List, Average) :-
  sum_ages(List, Sum),
  length(List, N),
  N > 0,
  Average is Sum / N.
```

But that requires two passes through `List`, once to compute `Sum` and another to compute the number of elements `N`.

Here is a one-pass, tail recursive program:

```
average_age([X|Rest], Average) :-
  age_stats([X|Rest], 0, 0, N, SumAges),
  Average is SumAges / N.
% the above fails if first argument is an empty list


% the second and third arguments are 'accumulators'
age_stats([], N, Sum, N, Sum).

age_stats([(_,_,Age)|Rest], N, Acc, N_final, Sum) :-
  NextAcc is Acc + Age,
  NextN is N + 1,
  age_stats(Rest, NextN, NextAcc, N_final, Sum).
```

2.4 Max and Avg together

```
max_avg_age([X|Rest], Max, Avg) :-
   age_stats([X|Rest], 0, 0, 0, Max, N, Sum),
   Avg is Sum / N.
% fails if first argument is empty list

age_stats([], MaxSofar, N, Sum, MaxSofar, N, Sum).

age_stats([(_,_,Age)|Rest], MaxSofar, N, Acc, Max, N_final, Sum) :-
   (
    Age > MaxSofar
    ->
       NextMax = Age
    ;
       NextMax = MaxSofar
   ),
  NextAcc is Acc + Age,
  NextN is N + 1,
     % can put a cut here
  age_stats(Rest, NextMax, NextN, NextAcc, Max, N_final, Sum).
```


3 Example 3

Given a list of terms of the form `(Person, Sex, Age)`, find

- the list of oldest `Person` (there may be more than one)

The program should fail if the list is empty.

Tail recursive version:

```
oldest_people([(Person,_,Age) | Rest], Oldest) :-
   oldest_people(Rest, Age, [Person], Oldest).

%% The second and third arguments of oldest_people/4 are the
%% maximum age so far and the list of persons so far with that age.

oldest_people([], _, Oldest, Oldest).
oldest_people([(Person,_,Age) | Rest], MaxSofar, Oldest, Result) :-
   compare_ages(Age, MaxSofar, Person, Oldest, NextMax, OldestNext),
     % can put a cut here
   oldest_people(Rest, NextMax, OldestNext, Result).


compare_ages(Age, MaxSofar, Person, Oldest, NextMax, OldestNext) :-
   Age = MaxSofar,       % include Person
   NextMax = MaxSofar,
   OldestNext = [Person|Oldest].
compare_ages(Age, MaxSofar, _Person, Oldest, NextMax, OldestNext) :-
   MaxSofar > Age,       % ignore Person
   NextMax = MaxSofar,
   OldestNext = Oldest.
compare_ages(Age, MaxSofar, Person, Oldest, NextMax, OldestNext) :-
   Age > MaxSofar,       % reset max age found so far
   NextMax = Age,
   OldestNext = [Person].
```

Or (more compact, but perhaps not as clear):

```
% Age of Person same as MaxSofar
compare_ages(MaxSofar, MaxSofar, Person, Oldest, MaxSofar, [Person|Oldest]).

% Person's Age < MaxSofar -- ignore Person
compare_ages(Age, MaxSofar, _Person, Oldest, MaxSofar, Oldest) :-
   MaxSofar > Age.

% Person is oldest so far
compare_ages(Age, MaxSofar, Person, _, Age, [Person])
   Age > MaxSofar.
```

As usual, one can use the if-then-else construction `...-> ...; ...` instead of defining an auxiliary predicate `compare_ages`.

```
oldest_people([], _, Oldest, Oldest).
oldest_people([(Person,_,Age) | Rest], MaxSofar, Oldest, Result) :-
  (
      Age = MaxSofar
      ->
         NextMax = MaxSofar,
         OldestNext = [Person|Oldest]
    ;
      MaxSofar > Age
      ->
         NextMax = MaxSofar,
         OldestNext = Oldest
    ;
      % otherwise
         NextMax = Age,
         OldestNext = [Person]
  ),
       % can put a cut here (but Sicstus doesn't need it)
   oldest_people(Rest, NextMax, OldestNext, Result).
```

Notice that if-then-else `...-> ...; ...` can be nested, as above. The layout (white space, new lines, indentation) doesn't matter. It's personal style.

4 Example 4

Given a list of terms of the form `(Person, Sex, Age)`, as in previous examples, split the list into the list of males and the list of females. Each of these should be a list of terms of the form `(Person, Age)`.

Assume first that the list does not contain duplicates entries for a person.

4.1 Version 1

(Some Prolog textbooks say this is not tail-recursive. It is.)

```
split_sex([], [], []).

split_sex([(P,male,Age)|Rest], [(P,Age)|Males], Females) :-
   split_sex(Rest, Males, Females).

split_sex([(P,female,Age)|Rest], Males, [(P,Age)|Females]) :-
   split_sex(Rest, Males, Females).
```

4.2 Version 2

This is a style recommended in some books.

```
split_sex([], [], []).

split_sex([(P,Sex,Age)|Rest], Males, Females) :-
   insert_item(Sex, (P,Age), Males, Females, RestMales, RestFemales),
     % can put a cut here
   split_sex(Rest, RestMales, RestFemales).


insert_item(male, X, [X|RestMales], RestFemales, RestMales, RestFemales).
insert_item(female, X, RestMales, [X|RestFemales], RestMales, RestFemales).
```

Alternatively, using if-then-else `...-> ...; ...` instead of defining an auxiliary predicate `insert_item`:

```
split_sex([], [], []).

split_sex([(P,Sex,Age)|Rest], Males, Females) :-
   (
    Sex = male
    ->
       Males = [(P,Age)|RestMales], Females = RestFemales
    ;
       Males = RestMales, Females = [(P,Age)|RestFemales]
   ),
     % can put a cut here
   split_sex(Rest, RestMales, RestFemales).
```

Remark I tried both versions in Sicstus Prolog. Somewhat surprisingly, I found that the simpler Version 1 is actually slightly *faster* than the more complicated Version 2 (at least in Sicstus).

4.3 Variation: eliminate duplicates

Same again, but this time suppose we need to remove duplicate entries for a person.

This will be a tail recursive program with two 'accumulator' arguments representing the list of males and list of females found so far.

```
split_sex_nodup(List, Males, Females) :-
   split_nodup(List, [], [], Males, Females).


split_nodup([], MalesSofar, FemalesSofar, MalesSofar, FemalesSofar).

split_nodup([(P,Sex,Age)|Rest], MalesSofar, FemalesSofar, Males, Females) :-
   insert_item(Sex, (P,Age), MalesSofar, FemalesSofar,
                             MalesSofarX, FemalesSofarX ),
   split_nodup(Rest, MalesSofarX, FemalesSofarX, Males, Females).



insert_item(male, X, MalesSofar, FemalesSofar, MalesSofar, FemalesSofar) :-
   member(X, MalesSofar).
insert_item(male, X, MalesSofar, FemalesSofar, [X|MalesSofar], FemalesSofar) :-
   \+ member(X, MalesSofar).
insert_item(female, X, MalesSofar, FemalesSofar, MalesSofar, FemalesSofar).
   member(X, FemalesSofar).
insert_item(female, X, MalesSofar, FemalesSofar, MalesSofar, [X|FemalesSofar]).
   \+ member(X, FemalesSofar).
```

Clearly the above has some redundant calls to `member`: not a problem if the lists are short but very inefficient nevertheless.

Some programmers would use the 'cut' like this

```
split_nodup([], MalesSofar, FemalesSofar, MalesSofar, FemalesSofar).

split_nodup([(P,Sex,Age)|Rest], MalesSofar, FemalesSofar, Males, Females) :-
   insert_item(Sex, (P,Age), MalesSofar, FemalesSofar,
                             MalesSofarX, FemalesSofarX ),
   !,
   split_nodup(Rest, MalesSofarX, FemalesSofarX, Males, Females).


% the following only works because insert_item call is followed by !
insert_item(male, X, MalesSofar, FemalesSofar, MalesSofar, FemalesSofar) :-
   member(X, MalesSofar).
insert_item(male, X, MalesSofar, FemalesSofar, [X|MalesSofar], FemalesSofar).
insert_item(female, X, MalesSofar, FemalesSofar, MalesSofar, FemalesSofar) :-
   member(X, FemalesSofar).
insert_item(female, X, MalesSofar, FemalesSofar, MalesSofar, [X|FemalesSofar]).
```

I don't like the style above. It's very unclear. If the recomputation of `member` is an issue, I would use an if-then-else, as follows.

Using if-then-else to avoid recomputation of `member`:

```
insert_item(male, X, MalesSofar, FemalesSofar, MalesSofarX, FemalesSofar) :-
   (
    member(X, MalesSofar)
    ->
       MalesSofarX = MalesSofar
    ;
       MalesSofarX = [X|MalesSofar]
   ).
insert_item(female, X, MalesSofar, FemalesSofar, MalesSofar, FemalesSofarX) (
    member(X, FemalesSofar)
    ->
       FemalesSofarX = FemalesSofar
    ;
       FemalesSofarX = [X|FemalesSofar]
   ).
```

Or, alternatively, without the `insert_item`:

```
split_nodup([], MalesSofar, FemalesSofar, MalesSofar, FemalesSofar).

split_nodup([(P,Sex,Age)|Rest], MalesSofar, FemalesSofar, Males, Females) :-
   (Sex = male
    ->
        FemalesSofarX = FemalesSofar,
        (member((P,Age), MalesSofar)
         ->
            MalesSofarX = MalesSofar
         ;
            MalesSofarX = [(P,Age)|MalesSofar]
        )
    ;
        % else Sex = female
        MalesSofarX = MalesSofar,
        (member((P,Age), FemalesSofar)
         ->
            FemalesSofarX = FemalesSofar
         ;
            FemalesSofarX = [(P,Age)|FemalesSofar]
        )
   ),
   split_nodup(Rest, MalesSofarX, FemalesSofarX, Males, Females).
```

Finally, you might prefer this one (which runs just as fast, at least in Sicstus):

```
split_nodup([], MalesSofar, FemalesSofar, MalesSofar, FemalesSofar).

split_nodup([(male,Sex,Age)|Rest], MalesSofar, FemalesSofar, Males, Females) :-
   (member((P,Age), MalesSofar)
    ->
        MalesSofarNew = MalesSofar
    ;
        MalesSofarNew = [(P,Age)|MalesSofar]

   ),
   split_nodup(Rest, MalesSofarNew, FemalesSofar, Males, Females).

split_nodup([(female,Sex,Age)|Rest], MalesSofar, FemalesSofar, Males, Females) :-
   (member((P,Age), FemalesSofar)
    ->
        FemalesSofarNew = FemalesSofar
    ;
        FemalesSofarNew = [(P,Age)|FemalesSofar]

   ),
   split_nodup(Rest, MalesSofar, FemalesSofarNew, Males, Females).
```

(Or with an auxiliary 'helper' predicate instead of the if-then-else.)