276 Introduction to Prolog

Miscellaneous Notes (Part 2)

Marek Sergot
Department of Computing
Imperial College, London

Autumn 2013

Here are some more assorted remarks, comments, examples.

1 A simple (but useful) procedure

```
?- p(X), write(X), nl, fail.
```

A slightly more complicated version

```
?- p(X), format("Answer is ~w~n", [X]), fail.
```

Prolog's I/O primitives are not an examinable part of this course.

2 Example

```
path(X,Y) :- arc(X,Y).
path(X,Y) :-
  arc(X,Z), path(Z,Y).
```

```
path(X,Y) :- arc(X,Y).
path(X,Y) :-
  path(Z,Y), arc(X,Z).
```

```
path(X,Y) :- arc(X,Y).
path(X,Y) :-
  arc(Z,Y), path(X,Z).
```

```
path(X,Y) :- arc(X,Y).
path(X,Y) :-
  path(X,Z), path(Z,Y).
```


The first version:

```
path(X,Y) :- arc(X,Y).
path(X,Y) :-
  arc(X,Z), path(Z,Y).
```

Procedurally: searches *forward* from given `X`.

```
    ?- path(a,Y).
   |
    ?- arc(a,Y).
 / |             Y = b (say)
 |   done
 |
 \
   ?- arc(a,Z), path(Z,Y).
  |              Z = b (say)
   ?- path(b,Y).
  |
    ... continues
```

Alternatively (equivalently):

```
path(X,Y) :- arc(X,Y).
path(X,Y) :-
  arc(Z,Y), path(X,Z).
```

Procedurally: searches *backwards* from given `Y`.

```
    ?- path(X,g).
   |
    ?- arc(X,g).
 / |             X = f (say)
 |   done
 |
 \
   ?- arc(Z,g), path(X,Z).
  |              Z = f (say)
   ?- path(X,f).
  |
    ... continues
```

Alternatively (equivalently):

```
path(X,Y) :- arc(X,Y).
path(X,Y) :-
  path(X,Z), path(Z,Y).
```

Procedurally: sometimes works, but usually – *AWFUL*

```
    ?- path(d,Y).
   |
    ?- arc(d,Y).
 / |
|    fails    (say)
|
 \
    ?- path(d,Z), path(Z,Y).
   |
    ?- arc(d,Z), path(Z,Y).
 / |
|     fails    (say)
|
 \
    ?- path(d,Z'), path(Z',Z), path(Z,Y).
   |
    ?- arc(d,Z'), path(Z',Z), path(Z,Y).
 / |
|     fails    (say)
|
 \
    ?- path(d,Z''), path(Z'',Z'), path(Z',Z), path(Z,Y).
   |
    ?- arc(d,Z''), path(Z'',Z'), path(Z',Z), path(Z,Y).
   |
      fails    (say)

     ... etc etc
```

(Actual behaviour depends on the graph, obviously)

Other graph-searching algorithms (e.g. Dijskra)?

Need a *different representation* of graphs in Prolog.

(There are several.)

3 Example (Exercises 3, Q2)

Some people I saw wrote something like this:

```
pairs([], []).
pairs([H|T], Paired) :-
  pairs(T, PairedTail),
  Less is H - 1, More is H + 1,
  append([(Less,More)], PairedTail, Paired).
```

Two things bad about this program. Bad, not fatal.

They are very common faults.

(a) pointless append

You wouldn't write:

```
pairs([], Result) :- append([], [], Result).
```

`append([Item], X, Result)` is just `Result = [Item|X]`.

Could have written:

```
pairs([H|T], Paired) :-
  pairs(T, PairedTail),
  Less is H - 1, More is H + 1,
  Paired = [(Less,More)| PairedTail].
```

But even the explicit call to `=` is unnecessary. Use pattern matching (unification).

```
pairs([], []).
pairs([H|T], [(Less,More)| PairedTail]) :-
  pairs(T, PairedTail),
  Less is H - 1, More is H + 1.
```

(Sometimes – if there are many arguments or the list patterns are very complicated – it is clearer to write it using `=`. But not usually.)

(b) make it tail recursive

Make your programs tail recursive if possible. *Much* more efficient.

Not always easy. *Very* easy in this example.

```
pairs([], []).
pairs([H|T], [(Less,More)| PairedTail]) :-
  Less is H - 1, More is H + 1,
  pairs(T, PairedTail).
```

Generally speaking, if you can:

- base case(s) first
- recursive call last (if possible)

4 Example

A coursework exercise from previous years requires as a first step a program

```
makeList(N, Item, List)
```

which, given an integer `N` and some (ground) term `Item` will generate a list `List` consisting of `N` copies of `Item`.

A common solution:

```
makeList(0, _, []).
makeList(N, Item, [Item|Rest]) :-
  M is N-1,
  makeList(M, Item, Rest).
```

Fine, except it loops if it backtracks:

```
?- makeList(2, a, X).
   X = [a,a] ;    % backtrack
                  % loops
```

Here is why ...

```
  ?- makeList(2, a, X).
  |
  ?- M is 2 - 1, makeList(M, a, Rest).
  |
  ?- makeList(1, a, Rest).
  |
  ?- M' is 1 - 1, makeList(M', a, Rest').
  |
  ?- makeList(0, a, Rest'').
 /|
|    (done) Rest'' = [], X = [a,a]
|
\
  ?- M'' is 0 - 1, makeList(M'', a, Rest').
  |
  ?- makeList(-1, a, Rest').
  |
   ...
  |
  ?- makeList(-2, a, Rest'').
  |
        etc, etc, etc
```

Solution (1): make the clauses mutually exclusive

```
makeList(0, _, []).
makeList(N, Item, [Item|Rest]) :-
  N > 0,     % better than N \= 0
  M is N-1,
  makeList(M, Item, Rest).
```

Solution (2): use a ‘cut’

```
makeList(0, _, []) :- !.    % pronounced 'cut'
makeList(N, Item, [Item|Rest]) :-
  M is N-1,
  makeList(M, Item, Rest).
```

The ’cut’ `!` hacks off all existing backtracking points.

Procedurally, you can read it as:

- if `!` is executed, no other clause for `makeList` (in this example) will be tried.

You can only read `!` procedurally – it has no sensible declarative reading.

It is horrible but very heavily used (especially by bad programmers).

More about ‘cut’ presently.

5 Example

```
all_bs([]).
all_bs([b|X]) :- all_bs(X).
```

Alternatively:

```
all_bs(List) :-
  forall( member(X, List),
          X = b ).
```

Equivalently:

```
all_bs(List) :-
  \+ ( member(X, List),
       X \= b ).
```

A common style of Prolog program:

```
all_bs(List) :-
  member(X, List),
  X \= b, !,    % 'cut'
  fail.
all_bs(_).
```

(The last version is no more efficient than the previous two.)

Note that the recursive program generates solutions to:

```
?- all_bs(X).
```

The iterative ‘forall’ (double negation) versions do not.

Some Prologs (e.g. Sicstus) don’t have `forall` built-in. You can define it yourself:

```
forall(P, Q) :- \+ ( call(P), \+ call(Q) ).
```

You can also write:

```
forall(P, Q) :- \+ ( P, \+ Q ).
```

6 Example

Recursively:

```
contained_in([],_).
contained_in([X|Rest], List) :-
   member(X, List),
   contained_in(Rest, List).
```

Alternatively:

```
contained_in(SubList, List) :-
  forall( member(X, SubList),
          member(X, List) ).
```

Equivalently:

```
contained_in(SubList, List) :-
  \+ ( member(X, SubList),
       \+ member(X, List) ).
```

A common hack using 'cut':

```
contained_in(SubList, List) :-
  member(X, SubList),
  \+ member(X, List), !,
  fail.
contained_in(_, _).  % horrible !
```

(The last version is no more efficient than the previous two.)