Logic Programming: Recursion, lists, data structures

Alan Smaill

Sep 28 2015

Today

- Recursion
  - proof search
  - practical concerns
- List processing
- Programming with terms as data structures.

Recursion

So far the rules we have seen have been (mostly) non-recursive.
This is a limit on what can be expressed.

Without recursion, we cannot define transitive closure
eg define `ancestor/2` in terms of `parent/2`.

Recursion ctd

In recursive use, the same predicate is used in the head (lhs) of the rule as in the body (rhs)
(in the second clause below):

```
ancestor(X,Y) :- parent(X,Y).

ancestor(X,Y) :- parent(X,Z),
                 ancestor(Z,Y).
```

Recursion ctd

In recursive use, the same predicate is used in the head (lhs) of the rule as in the body (rhs)
(in the second clause below):

```
ancestor(X,Y) :- parent(X,Y).

ancestor(X,Y) :- parent(X,Z),
                 ancestor(Z,Y).
```

This is a fine declarative description of what it is to be an ancestor.

But watch out for the traps!!!

Reminder: depth-first search

Prolog searches **depth-first** in program order ("top to bottom"):

- Regardless of context
- ...even if there is an "obvious" solution elsewhere in the search space.

Reminder: depth-first search

Prolog searches **depth-first** in program order ("top to bottom"):

- Regardless of context
- ... even if there is an "obvious" solution elsewhere in the search space.

```
p :- p.
p.

?- p.
```

— the query will **loop** on the first clause, and fail to terminate.

Recursion: order can matter

Take the program for `ancestor/2` with clauses in the opposite order:

```
ancestor(X,Y) :- parent(X,Z),
                 ancestor(Z,Y).

ancestor(X,Y) :- parent(X,Y).
```

Recursion: order can matter

Take the program for ancestor/2 with clauses in the opposite order:

```
ancestor(X,Y) :- parent(X,Z),
                 ancestor(Z,Y).

ancestor(X,Y) :- parent(X,Y).
```

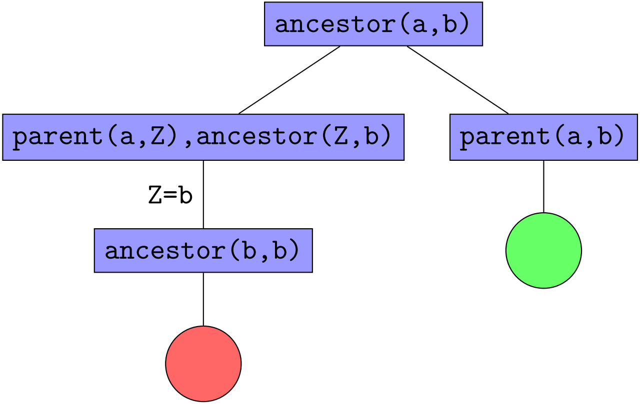
This may be less efficient – looks for **longest** path first.

More likely to loop – if the parent/2 relation has cycles.

HEURISTIC: write base cases first (ie non-recursive cases).

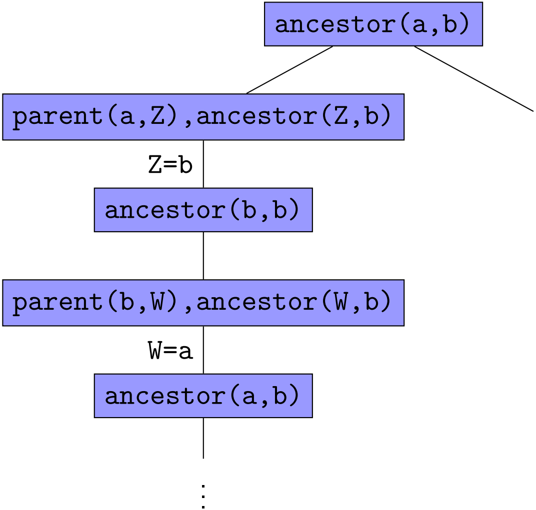
Rule order affects search

```
parent(a,b).
parent(b,c).
```



Rule order affects search

```
parent(a,b).
parent(b,a).
```



Recursion again

Goal order can matter!

```
ancestor3(X,Y) :- parent(X,Y).
ancestor3(X,Y) :- ancestor3(Z,Y),
                  parent(X,Z)
```

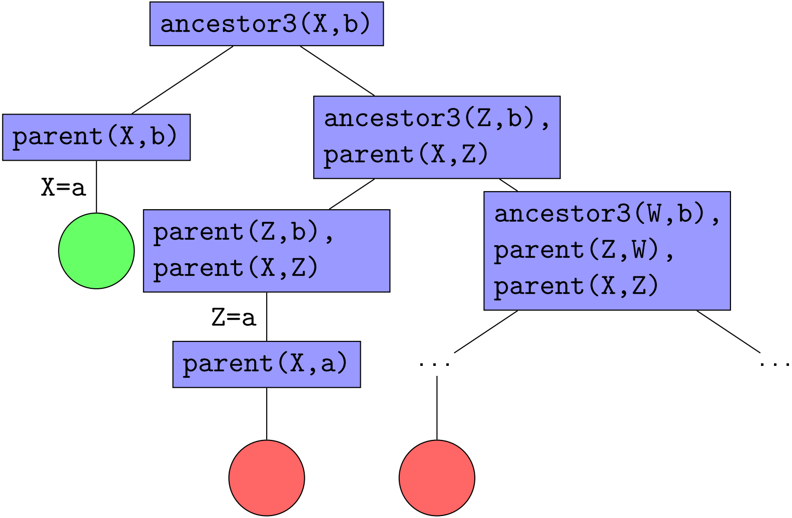
Goal order can matter!

```
ancestor3(X,Y) :- parent(X,Y).
ancestor3(X,Y) :- ancestor3(Z,Y),
                  parent(X,Z)
```

This returns all solutions, then loops, eg with the following facts:

```
parent(a,b).
parent(b,c).
```

Goal order affects search



Clause order can matter.

```
ancestor4(X,Y) :- ancestor4(Z,Y),
                   parent(X,Z).
ancestor4(X,Y) :- parent(X,Y).
```

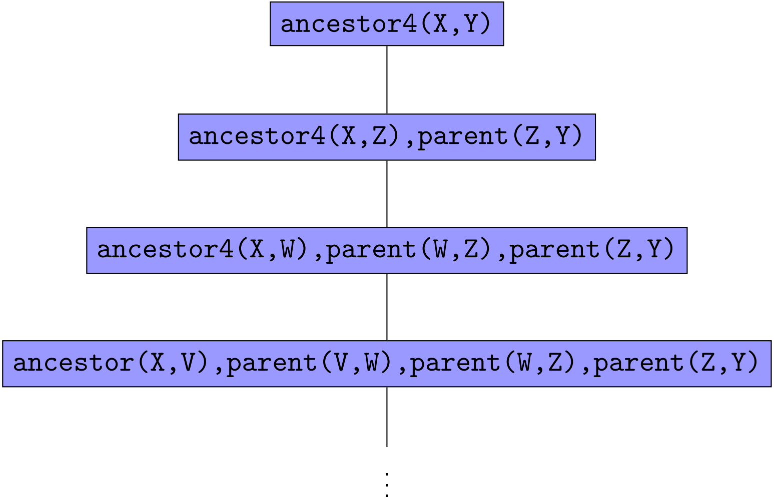
More recursion

Clause order can matter.

```
ancestor4(X,Y) :- ancestor4(Z,Y),
                   parent(X,Z).
ancestor4(X,Y) :- parent(X,Y).
```

This will always loop.
Heuristic: put non-recursive goals first.

Goal order matters



Recursion and terms

- Terms can be arbitrarily nested
- Example: unary natural numbers

```
nat(z).
nat(s(X)) :- nat(X).
```

- Terms can be arbitrarily nested
- Example: unary natural numbers

```
nat(z).
nat(s(X)) :- nat(X).
```

- To do interesting things, we need recursion.

Addition, subtraction

- Addition:

```
add(z,N,N).
add(s(N),M,s(P)) :- add(N,M,P).
```

- Addition:

```
add(z,N,N).
add(s(N),M,s(P)) :- add(N,M,P).
```

  - Run in reverse to get all M,N that sum to P:

Addition, subtraction

- Addition:

```
add(z,N,N).
add(s(N),M,s(P)) :- add(N,M,P).
```

  - Run in reverse to get all M,N that sum to P:

    ```
    ?- add(X,Y,s(s(s(z)))).
    X=z,Y=s(s(s(z)));
    X=s(Z),Y=s(s(z));
    ...
    ```

Addition, subtraction

- Addition:

```
add(z,N,N).
add(s(N),M,s(P)) :- add(N,M,P).
```

  - Run in reverse to get all M,N that sum to P:

```
?- add(X,Y,s(s(s(z)))).
X=z,Y=s(s(s(z)));
X=s(Z),Y=s(s(z));
...
```

  - Use to define `leq/2`:

```
leq(M,N) :- add(M,_,N).
```

Addition, subtraction

- Addition:

```
add(z,N,N).
add(s(N),M,s(P)) :- add(N,M,P).
```

  - Run in reverse to get all M,N that sum to P:

    ```
    ?- add(X,Y,s(s(s(z)))).
    X=z,Y=s(s(s(z)));
    X=s(Z),Y=s(s(z));
    ...
    ```

  - Use to define `leq/2`:

    ```
    leq(M,N) :- add(M,_,N).
    ```

Here "_" is a so-called **anonymous** variable;
use to avoid warning of *singleton variable* in Prolog programs.
Can also use, for example, `_X`, `_Anon`.

Multiplication

Now define multiplication:

```
multiply(z,N,z).   % or: multiply(z,_,z).

multiply(s(N),M,P) :-
    multiply(N,M,Q), add(M,Q,P).

square(N,M) :- multiply(N,N,M).
```

List Processing

- Recall built-in list syntax:

```
list([]).
list([X|L]) :- list(L).
```

List Processing

- Recall built-in list syntax:

```
list([]).
list([X|L]) :- list(L).
```

- Examples: list append

```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).
```

append in action

- Forward direction:

```
?- append([1,2],[3,4],X).

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

append in action

- Forward direction:

```
?- append([1,2],[3,4],X).

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

- Backward direction

```
?- append(X,Y,[1,2,3,4]).
X=[], Y=[1,2,3,4];
X=[1],Y=[2,3,4];
...
```

Mode annotations

These are recognised ways of indicating properties of Prolog procedures.

- Notation: `append(+,+,-)`
  - Expect to be called with the first two arguments ground, and third a variable (which we normally expect to bound after the call)

Mode annotations

These are recognised ways of indicating properties of Prolog procedures.

- Notation: `append(+,+,-)`
  - Expect to be called with the first two arguments ground, and third a variable (which we normally expect to bound after the call)
- Similarly, `append(-,-,+)`
  - Call with last argument ground, first two as variables (which we normally expect to be bound after the call).

Mode annotations

These are recognised ways of indicating properties of Prolog procedures.

- Notation: `append(+,+,-)`
  - Expect to be called with the first two arguments ground, and third a variable (which we normally expect to bound after the call)
- Similarly, `append(-,-,+)`
  - Call with last argument ground, first two as variables (which we normally expect to be bound after the call).
- Not "code", but often used in annotations
- "?" annotation used where any term may appear — i.e. ground, variable, or compound term with variables.

When is something a member of a list?

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

When is something a member of a list?

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

Typical modes:

```
member(+,+)
member(-,+)
```

- Removing an element of a list:

```
remove(X, [X|L], L).
remove(X, [Y|L], [Y|M]) :- remove(X, L, M).
```

List processing ctd

- Removing an element of a list:

```
remove(X, [X|L], L).
remove(X, [Y|L], [Y|M]) :- remove(X, L, M).
```

  NB: removes one occurrence of `X`;
  fails if `X` is not a member of the list.

- Typical mode:
  `remove(+,+,-)`

- Zip: pairing of corresponding elements of lists: assumed to be of same length.

```
zip([],[],[]).
zip([X|L], [Y|M], [(X,Y)|N]) :- zip(L, M, N).
```

- Typical modes:

```
zip(+,+,-).
zip(-,-,+).  % unzip
```

List flattening

- Write a **flatten** predicate `flatten/2` that
  - Given a list of (lists of ...)
  - Produces a list of individual elements in the original order.

- Write a **flatten** predicate `flatten/2` that
  - Given a list of (lists of ...)
  - Produces a list of individual elements in the original order.
  - Examples:

```
?- flatten([[1,2],[3,4]], L).
   L = [1,2,3,4]

?- flatten([[1,2],[3,[4,5]],6],L).
   L = [1,2,3,4,5,6]

?- flatten([3,X,[4,5]],L).
   L = [3,X,4,5]
```

List flattening

```
flatten([], []).

flatten([H|T], M) :- flatten(H, Hf),
                     flatten(T, Tf),
                     append(Hf, Tf, M).

flatten(X, [X]) :-   ???
    % non-list case;  how treat variables?!?!
```

Records

- Can use terms to define data structures:

```
pb([entry(alan, '156-675'),...]).
```

- Can use terms to define data structures:

```
pb([entry(alan, '156-675'),...]).
```

- and operations on them:

```
pb_lookup(pb(B), P, N) :-
        member(entry(P,N), B).

pb_insert(pb(B), P, N, pb([entry(P,N) | B])).

pb_remove(pb(B), P, pb(B2)) :-
        remove(entry(P,_), B, B2).
```

Trees

We can define (binary) trees with data (at the nodes).

```
tree(leaf).
tree(node( Data, LT, RT )) :- tree(LT), tree(RT).
```

Trees

We can define (binary) trees with data (at the nodes).

```
tree(leaf).
tree(node( Data, LT, RT )) :- tree(LT), tree(RT).
```

Data membership in a tree —
using ";" for alternatives in the body of a clause.

```
mem_tree(X, node(X, _, _)).
mem_tree(X, node(_, LT, RT)) :-
      mem_tree(X, LT)  ;
      mem_tree(X, RT).
```

Preorder traversal

Pick up the data in a particular order:
start at root, traverse recursively left subtree, then right subtree.

```
preorder(leaf, []).

preorder(node(X, LT, RT), [X|N]) :-
  preorder(LT, LO),
  preorder(RT, RO),
  append( LO, RO, N).
```

Pick up the data in a particular order:
start at root, traverse recursively left subtree, then right subtree.

```
preorder(leaf, []).

preorder(node(X, LT, RT), [X|N]) :-
  preorder(LT, LO),
  preorder(RT, RO),
  append( LO, RO, N).
```

What happens if we run this in reverse?

- The tutorial questions are on the web page;
  you should work through these before the tutorial.
- It's recommended to use the sicstus emacs mode to interact with Prolog and edit source code. This mode is invoked automatically when editing Prolog files (with suffix `.pl`) on DICE.
  (See sicstus documentation if you want to set this up for yourself.)

  You can find out about the mode by "`C-h m`" in emacs when the mode is in use, or via sicstus documentation.

Coming Attractions

- Non-logical features:
  - Expression evaluation
  - I/O
  - "cut" (pruning proof search)
- Further reading
-   - Learn Prolog Now, ch 3–4
- Tutorial questions on web page.