

COMP210: Artificial Intelligence

Lecture 6. Prolog: Recursive Definitions

Trevor Bench-Capon
Room 215, Ashton Building

<http://www.csc.liv.ac.uk/~tbc/COMP210>

Last Week

- A Prolog program is constructed from facts and rules.
- Facts describe things that are true without conditions
 - like data in a database.
- Rules describe things that hold depending on certain conditions.
- Prolog programs can be queried using questions.
- Prolog clauses are facts, rules and questions.
- Queries are answered by instantiating variables, creating new subgoals from rules and matching with facts.

Recursion

- Aims:-
 - to be able to write recursive rules in Prolog;
 - to be able to evaluate recursive rules.
 - to be able to recognise (and avoid) infinite recursion;
 - to appreciate the difference between declarative and procedural meanings for Prolog programs.

The Ancestor Relation

- Consider the family tree from the previous lecture. We'd like to be able to define ancestor:
 - a parent, or
 - a parent of a parent, or
 - a parent of a parent of a parent, or
 - . . .

First Attempt

```
ancestor(X,Z):-
    parent(X,Z).
ancestor(X,Z):-
    parent(X,Y),
    parent(Y,Z).
ancestor(X,Z):-
    parent(X,Y1),
    parent(Y1,Y2),
    parent(Y2,Z).
```

- Problems:
 - Lengthy (not important)
 - If we use only the first two of these (or three or four etc) we get a finite depth on our search for ancestors

Recursion

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

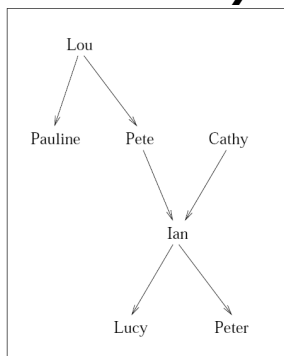
Base case

Recursive Call

Predicate in head and body

- This type of definition is very important in Prolog. A set of clauses referring to the same relation is known as a procedure. This is a *recursive procedure*

Example: A Family Tree



Questioning Recursive Definitions

• The question *Who are the ancestors of Lucy?* is posed as follows.

| ?- ancestor(X,lucy).

- X=ian ? ;
- X=pete ? ;
- X=lou ? ;
- X=cathy ? ;
- no

Semi-Colon for Next answer
Full stop for Enough

How this Works

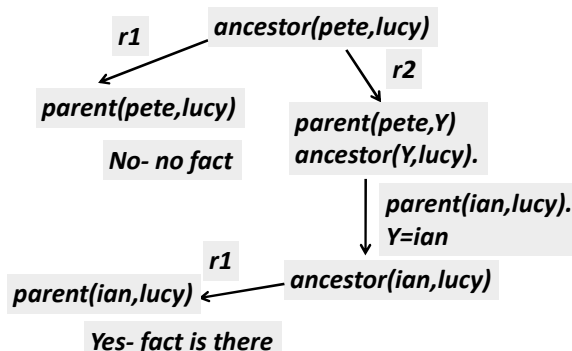
```

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

```

- ancestor(pete,lucy)?
- X=pete Y= lucy
- No clause for parent(pete,lucy) so first clause fails
- Try second clause:
- X= pete, Z= lucy
- parent(pete,Y)
- parent(pete,ian)
- X=pete, Z=lucy,Y=ian
- ancestor(ian,lucy)?
- Try first clause
- parent(ian,lucy)
- Success! So Prolog answers yes

How This Works 2



Recursion and Search

- Recursion is a powerful construct essential to Prolog.
- Prolog searches for an answer through several possibilities
 - Uses depth first search
- This search method can be applied to search problems
- But first we have to learn about structures

Structures

- Structures are useful data structure in Prolog.
- They are objects that have several components and a name (functor) that associates them together.
 - date(5, february, 2002)
 - location(depot1, manchester)
 - id_no(rajeev, gore, 02571)
 - state(ontable,onblock)
- Cf database tuples: functor is table name, arguments are attributes

Example: Monkey and Banana

- "A monkey at the door into a room. In the middle of the room a banana is hanging from the ceiling. The monkey is hungry and wants the banana, but cannot stretch high enough from the floor. At the window there is a box the monkey can use."
- The monkey can
 - walk
 - climb
 - push the box (if at box)
 - grasp the banana (if standing on box under banana)
- Can the monkey get the banana?

States

- States are represented by the structure state(X,Y,Z,U)
 - X: Horizontal position of monkey
 - (atdoor, middle, atwindow)
 - Y: Vertical position of monkey
 - (onfloor, onbox)
 - Z: Position of box
 - U: Monkey has or has not banana
- Goal state: state(_, _, _, has)
- The underscore _ stands for an *anonymous variable* — we do not care what the value is

Moves

- Monkey can
 - Grasp
 - Climb
 - Push
 - Walk
- Moves change states, for example,
 - move(state(middle, onbox, middle, hasnot), grasp, state(middle, onbox, middle, has)).

State problem with Recursion

- Base Case – monkey already has the banana
 - canget (state(_, _, _, has)) .
- there is a move which gets the monkey the banana
 - canget (State1) :-
 - move (State1, Move, State2),
 - canget (State2) .

The Program

Tries moves in order: grasp, climb, push, walk

```

% move( State1, Move, State2): making Move in State1 results in State2,
% a state is represented by a structure:
% state( MonkeyHorizontal, MonkeyVertical, BoxPosition, HasBanana)

move( state( middle, onbox, middle, hasnot), % Before move
     grasp, % Grasp banana
     state( middle, onbox, middle, has) ). % After move

move( state( P, onfloor, P, H),
     climb, % Climb box
     state( P, onbox, P, H) ).

move( state( P1, onfloor, P1, H),
     push( P1, P2), % Push box from P1 to P2
     state( P2, onfloor, P2, H) ).

move( state( P1, onfloor, B, H),
     walk( P1, P2), % Walk from P1 to P2
     state( P2, onfloor, B, H) ).

canget( state( _, _, _, has) ). % can 1: Monkey already has it
canget( State1 ) :- % can 2: Do some work to get it
    move( State1, Move, State2), % Do something
    canget( State2). % Get it now
    
```

Can't grasp, climb or push, so walks

```

?- canget(state(atdoor, onfloor, atwindow, hasnot)).
Call: (7) canget(state(atdoor, onfloor, atwindow, hasnot)) ?
Call: (8) move(state(atdoor, onfloor, atwindow, hasnot), _L230, _L203) ?
Exit: (8) move(state(atdoor, onfloor, atwindow, hasnot), walk(atdoor, _G608), state(_G608, onfloor, atwindow, hasnot)) ?
Call: (8) canget(state(_G608, onfloor, atwindow, hasnot)) ?
Call: (9) move(state(_G608, onfloor, atwindow, hasnot), _L237, _L207) ?
Exit: (9) move(state(atwindow, onfloor, atwindow, hasnot), climb, state(atwindow, onbox, atwindow, hasnot)) ?
Call: (9) canget(state(atwindow, onbox, atwindow, hasnot)) ?
Call: (10) move(state(atwindow, onbox, atwindow, hasnot), _L274, _L257) ?
Fail: (10) move(state(atwindow, onbox, atwindow, hasnot), _L274, _L257) ?
Fail: (9) canget(state(atwindow, onbox, atwindow, hasnot)) ?
Redo: (9) move(state(_G608, onfloor, atwindow, hasnot), _L237, _L220) ?
Exit: (9) move(state(atwindow, onfloor, atwindow, hasnot), push(atwindow, _G616), state(_G616, onfloor, _G616, hasnot)) ?
Call: (9) canget(state(_G616, onfloor, _G616, hasnot)) ?
Call: (10) move(state(_G616, onfloor, _G616, hasnot), _L275, _L258) ?
Exit: (10) move(state(_G616, onfloor, _G616, hasnot), climb, state(_G616, onbox, _G616, hasnot)) ?
Call: (10) canget(state(_G616, onbox, _G616, hasnot)) ?
Call: (11) move(state(_G616, onbox, _G616, hasnot), _L312, _L295) ?
Exit: (11) move(state(middle, onbox, middle, hasnot), grasp, state(middle, onbox, middle, has)) ?
Call: (11) canget(state(middle, onbox, middle, has)) ?
Exit: (11) canget(state(middle, onbox, middle, has)) ?
Exit: (10) canget(state(middle, onbox, middle, hasnot)) ?
Exit: (9) canget(state(middle, onfloor, middle, hasnot)) ?
Exit: (8) canget(state(atwindow, onfloor, atwindow, hasnot)) ?
Exit: (7) canget(state(atdoor, onfloor, atwindow, hasnot)) ?
true.
    
```

Tries climbing

Not under banana

Pushes instead

Can't grasp, so climbs

Can grasp now

Unwind the recursion

Comments

- Prolog backtracked only once
- Good ordering of clauses in the move procedure
- With a different order may never terminate!
 - Example: if push is the first line in the move procedure, the money will go to the box and then push it around aimlessly
 - Always needs to try to grasp, and see whether climbing helps

Infinite Loops

- Take care when using recursion as it is easy to get in an infinite loop.
- If we add the following rule to our program


```
silly(X):- silly(X)
```
- meaning something like people are silly if they are silly.
- Querying this program


```
| ?- silly(boris).
```
- is matched to the head of the above rule, X is instantiated by boris and the new subgoal is silly(boris). This is matched a second time to the head of the above clause, a new subgoal is generated etc but the program never returns a solution, because there are no facts for silly.

Infinite Loops with ancestor

- If we change the definition of ancestor to predecessor as follows we have problems.

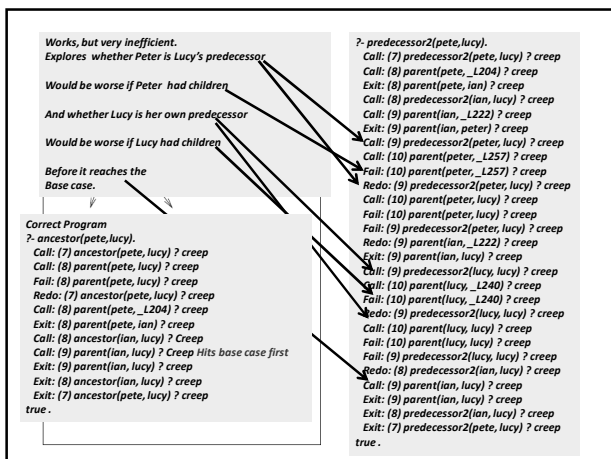
```
predecessor(X,Z):-
    predecessor(Y,Z),
    parent(X,Y).
predecessor(X,Z):-
    parent(X,Z).
```

We never hit the base case! We call predecessor(_,_) for ever

Avoiding Infinite Looping

- A general rule to avoid such problems is to ensure that the base case is the first clause - try the simplest idea first.
- Must hit a fact to terminate


```
predecessor2(X,Z):-
    parent(X,Y),
    predecessor2(Y,Z).
predecessor2(X,Z):-
    parent(X,Z).
```
- Does this cause infinite looping?



Declarative and Procedural Meaning

- Two interpretations of meaning of Prolog programs: -
 - declarative (logical)
 - procedural
- The declarative meaning determines what will count as an answer. It is concerned only with the relations that have been defined in the program.
- The procedural meaning also involves how this output is obtained. This means that the order of clauses is significant.

Declarative and Procedural Meaning

- Consider the query `parent(X,ian)`.
- The declarative meaning tells us that both `cathy` and `pete` can be successfully instantiated as `X`.
- The procedural meaning tells us if the fact `parent(pete,ian)` occurs in the program before `parent(cathy,ian)`, the first answer returned
- will be `X=pete` (but we can also obtain the second answer by typing a semi-colon).

Declarative and Procedural Meaning

- Remember Prolog uses depth first search to order its goals
- If the wrong node is chosen, the path may be infinite (so the program does not terminate)
- Or the path may be very long and ultimately unsuccessful
- So it is important that we order the clauses so the best path will be tried first

Declarative and Procedural Meaning

- Ancestor and both versions of predecessor are declaratively the same.
- Yet we have shown that predecessor does not return an answer, i.e. it is not procedurally correct.
- It can be shown that ancestor and predecessor2 both reach an answer for any query.
 - But predecessor2 is procedurally inefficient

Declarative and Procedural Meaning

- Another version of predecessor

```
predecessor3(X,Z):-
  parent(X,Z).
predecessor3(X,Z):-
  predecessor3(Y,Z),
  parent(X,Y).
```

Need to instantiate Y before the recursive call

Ground (no variables) queries will all be easier to solve (path will be shorter)

- Doesn't give an answer for like
- `?- predecessor3(lucy,peter).`

Summary

- Recursion is a powerful construct essential to Prolog.
- Take care with recursive rules to avoid an infinite sequence of recursive calls.
- The order of clauses and goals does matter.
- Programs which are declaratively correct may not be procedurally correct (and so will not work in practice).