Comp24112: Symbolic AI
Lecture 3: Prolog III

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2016–17

Outline

- Prolog has a special predicate, `not`.
- The goal `not(`*goal*`)` succeeds when *goal* fails. Given:

```
parent(sue,noel).
parent(chris,noel).
parent(noel,ann).
parent(ann,dave).
```

- we have the following behaviour

```
?-  not(parent(sue, noel)).

No

?-  not(parent(ann, noel)).

Yes
```

- Note that `not` in Prolog is different to ‘not’ in English.
- For example, the above program says nothing about Dougal and Zebedee, yet

```
?-  not(parent(dougal, zebedee)).

Yes
```

- In these contexts `not` means ‘not as far as I know’.
- `not` is not part of pure logic programming, and has no (clear) declarative meaning

- To think about: What happens with programs like these?

  ```
  p:- not(p).
  ```

  or

  ```
  p:- not(q).
  q:- not(p).
  ```

- `not` can be useful in various predicate definitions
- A **set** is like a list except that the order of elements is unimportant, and there are no repeated elements.
- The following predicate computes the union of two sets

```
union([],S,S).
union([X|S],S1,S2):-
   member(X,S1),
   union(S,S1,S2).

union([X|S],S1,[X|S2]):-
   not(member(X,S1)),
   union(S,S1,S2).
```

- It is **not** what you need for the first lab!

- There is a useful predicate \= (read **not equal**)
- X \= Y is the same as `not(`X`=` Y`)`:

```
?- a \= a.
no

?- a \= b.
yes
```

- There is a deadly trap involving `not`
- The call

  ```
  ?- not(parent(X,noel)).
  ```

  will **fail** (given the above program).
- It will not find a value for `X` which is not one of Noel's parents, eg. Ann!

- The same applies to `\=`.
- The call

  `?- X \= a.`

  will always **fail**.
- It will not find a value for `X` which is not equal to a!

- To understand what is going on with `not`, we need to understand `!`, or ‘cut’:
  - `!` always succeeds
  - Once an instance of `!` has succeeded, Prolog is committed to all choices made between the matching of the clause containing that instance of `!` and the instance of `!` itself
  - This includes other declarations for proving the same clause. This is very useful as we will now see.

- Example:

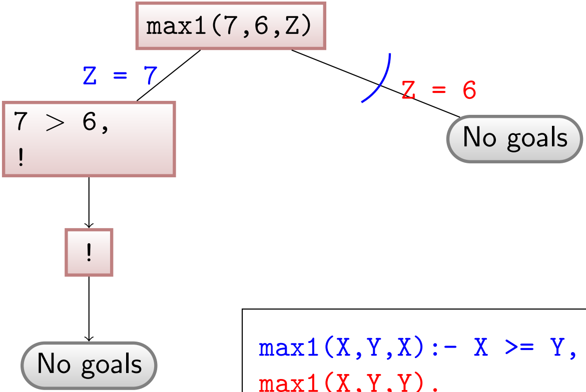
  ```
  max1(X,Y,X):- X >= Y.
  max1(X,Y,Y):- X < Y.
  ```

  is (well, sort of) equivalent to

  ```
  max1(X,Y,X):- X >= Y,!.
  max1(X,Y,Y).
  ```

- **Warning: deliberate error.**

- It helps to consider the search tree




```
max1(X,Y,X):- X >= Y,!.
max1(X,Y,Y).
```

- Why does the program

  ```
  max1(X,Y,X):- X >= Y,!.
  max1(X,Y,Y).
  ```

  contain an error?
- Because all arguments might be instantiated:

  ```
  2 ?- max1(3,2,X).

  X = 3 ;

  No
  5 ?- max1(3,2,2).

  Yes
  ```

- How do we fix this?
- By delaying the instantiation of the third variable in the first rule:

```
max2(X,Y,Z):- X >= Y,!,X=Z.
max2(X,Y,Y).
```

- This produces the correct behaviour:

```
18 ?-  max1(3,2,2).

Yes
19 ?-  max2(3,2,2).

No
```

- Here is another example of cut:

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

one_member(X,[X|L]):- !.
one_member(X,[Y|L]):- one_member(X,L).
```

- This is just like ordinary `member` except that it does not find repeated solutions

- Thus:

```
?- member(a, [a, b, a, c]).
yes
?- member(X, [a, b, a, c]).
X = a ;
X = b ;
X = a ;
X = c ;
No
?- one_member(a,[a, b, a, c]).
Yes
?- one_member(X,[a, b, a, c]).
X = a ;
No
```

- The relationship between `not` and `!`:
- Note that `call(`*term*`)` calls *term* as if it were a a goal in its own right.

```
?- ancestor(sue,N).
N = noel

Yes

?- call(ancestor(sue,N)).
N = noel

Yes
```

- We can define `not` in terms of `!`.

```
not(Goal):- call(Goal), !, fail.
not(Goal).
```

Outline

- Time to worry about efficiency
- Consider again the definition of append/3

```
append([X|L1],L2,[X|L3]):-
   append(L1,L2,L3).
append([],L,L).
```

- Calls to append/3 evidently involve a number of goal calls which is **linear** in the first argument.

- Consider again the definition of reverse/2

```
rev1([X|L], L_ans):-
    rev1(L,L_ans1),
    append(L_ans1,[X],L_ans).

rev1([],[]).
```

where append/3 is as defined above

- Suppose the list we are reversing has N elements
- For the kth item in the list, we perform an `append` on the end of an $N-k$ element list.
- This append takes $N-k+1$ reductions
- So the whole operation takes

$$(N+1)+N+\ldots+1=\frac{1}{2}(N+1)(N+2)$$

  reductions
- That is, it is quadratic in N.
- Although `rev1` is very easy to understand, quadratic behaviour seems unacceptable.

- Here is a better program:

```
rev2(L,L1):-
   rev_acc(L,[],L1).

rev_acc([X|L],L_acc,L_ans):-
   rev_acc(L,[X|L_acc],L_ans).

rev_acc([],L_acc,L_acc).
```

- In operation:

```
?-  rev2([a, b, c, d, e], L).

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

- The middle variable in `rev2` is called an **accumulator**

- We can see better what is going on by tracing:

```
call  rev2([a, b, c], _873)
 call  rev_acc([a, b, c], [], _873)
  call  rev_acc([b, c], [a], _873)
   call  rev_acc([c], [b, a], _873)
    call  rev_acc([], [c, b, a], _873)
    exit  rev_acc([], [c, b, a], [c, b, a])
   exit  rev_acc([c], [b, a], [c, b, a])
  exit  rev_acc([b, c], [a], [c, b, a])
 exit  rev_acc([a, b, c], [], [c, b, a])
exit  rev2([a, b, c], [c, b, a])
```

- Obviously, the number of reductions performed by `rev2` is linear in the length of the list.

- Here is another way to think about the same thing.
- Now consider the predicate `append_dfl`:

```
append_dfl(L1/L2,L2/L3,L1/L3).
```

- The time taken to query `append_dfl` is constant
- The program clearly appends lists (look at the variable `L`):

```
:-  append_dfl([a, b, c| X]/X, [d, e, f| Y]/Y, U/[])

N1 X = [d, e, f],  Y = [],  U = [a, b, c, d, e, f]
```

- We are interested in structures of the form

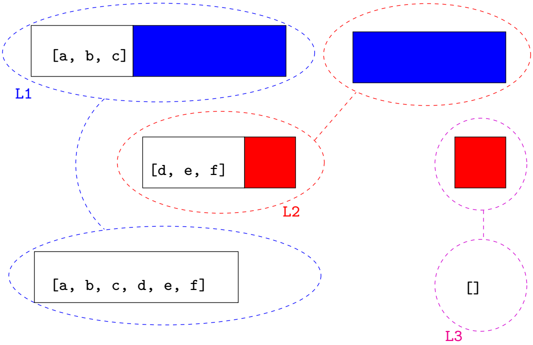
$$[x_1, \ldots, x_i, x_{i+1}, \ldots, x_n]/[x_{i+1}, \ldots, x_n]$$

- This represents the list

$$[x_1, \ldots, x_i]$$

  i.e. the difference between the two lists.
- We are particularly interested in difference lists such as `[a, b, c| X]/X`.
- Such data-structures are said to be **incomplete**

- The following picture illustrates what is going on with a call to the goal `append_dfl([a,b,c|X]/X,[d,e,f|Y]/Y,U/[])` given the program `append_dfl(L1/L2,L2/L3,L1/L3).`



- We can also write a fast list-reverse program using difference lists

```
rev3(L,L1):-
   rev_dfl(L,L1/[]).

rev_dfl([X|L],L_ans/L_ans_tail):-
   rev_dfl(L,L_ans/[X|L_ans_tail]).

rev_dfl([],L_ans_tail/L_ans_tail).
```

- This is just like `rev1`, except that the method of appending used in `append_dfl` has been ‘built in’ to the definition

- The program certainly works:

```
?-  rev3([1, 2, 3], L)

L = [3, 2, 1]
```

```
call  rev3([1, 2, 3], _897)
 call  rev_dfl([1, 2, 3], _897/[])
  call  rev_dfl([2, 3], _897/[1])
   call  rev_dfl([3], _897/[2, 1])
    call  rev_dfl([], _897/[3, 2, 1])
    exit  rev_dfl([], [3, 2, 1]/[3, 2, 1])
   exit  rev_dfl([3], [3, 2, 1]/[2, 1])
  exit  rev_dfl([2, 3], [3, 2, 1]/[1])
 exit  rev_dfl([1, 2, 3], [3, 2, 1]/[])
exit  rev3([1, 2, 3], [3, 2, 1])
```

- If this looks unfamiliar, it shouldn't.

```
rev3(L,L1):-
   rev_dfl(L,L1/[]).
rev_dfl([X|L],L_ans/L_ans_tail):-
   rev_dfl(L,L_ans/[X|L_ans_tail]).
rev_dfl([],L_ans_tail/L_ans_tail).


rev2(L,L1):-
   rev_acc(L,[],L1).
rev_acc([X|L],L_acc,L_ans):-
    rev_acc(L,[X|L_acc],L_ans).
rev_acc([],L_acc,L_acc]).
```

- As a final example, recall

```
factorial(0,1).

factorial(N,F):-
   N > 0,
   N1 is N - 1,
   factorial(N1,F1),
   F is N * F1.
```

- The Prolog interpreter has to store the program state on the stack before making each recursive call.

- Here is an alternative using a numerical accumulator:

```
factorial2(N,F):-
   factorial2_acc(N,1,F).

factorial2_acc(N,Acc,F):-
   Acc1 is Acc * N,
   N1 is N - 1,
   factorial2_acc(N1,Acc1,F).

factorial2_acc(0,Acc,Acc).
```

- The Prolog interpreter can forget about the program state on the stack before making each recursive call.

- These two programs require the same number of calls as each other
- Nevertheless, the second is more efficient
- This is because Prolog does not need to keep a stack of the `factorial2`-goals, and can reclaim the space taken up by each.
- This space reclamation is handled by the Prolog compiler, and is called **tail-recursion optimization**.

Outline

- Summary:
  - Negation: `not` and \=
  - Flow of control: the cut !
  - Accumulators
  - Incomplete data structures: difference lists
- What should I do next?
  - Revise Chh. 6, 10 of *Learn Prolog Now!*.
  - Read Introduction and Chh. 1,5 of *Representation and inference for natural Language* for next lecture.