COMP210: Artificial Intelligence

Lecture 9. Prolog: Lists and operations

Trevor Bench-Capon
Room 215, Ashton Building

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

Recap: 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)`

Structures

- Both `location(depot1, manchester)` and `manchester` known as *terms.*
- Components of structured objects can themselves be structured eg
  - `id_no(name(rajeev,gore),02571)`
- Can contain variables:
  - `location(X, manchester)` could be used in a program to mean any depot in Manchester.

Using Structured Object in Procedures

```
/****************************************/
% alternative_depot(Loc1,Loc2).
% takes two locations (location(Depot,City))
% Loc1 and Loc2 and succeeds if the the City
% component of each is the same.
/*****************************************/
alternative_depot(location(X,Z),location(Y,Z)).
/****************************************/
% move(State1,State2).
% takes a state (state(LocBlock1,LocBlock2))
% State1 and returns a new state State2
% dependent on action taken.
/*****************************************/

move(state(ontable,ontable),
          state(ontable,inhand)):-
    pickup(block2).
```

Today

- Aims:-
- to be familiar with the syntax of lists;
- to be able to write procedures involving lists;
- to know and be able to use built in operators for arithmetic and comparison
- to carry out simple debugging.

Lists

- Lists are the most commonly used data structure in Prolog.
- Lists are represented as
  - `[clare,sean,richard,paula]`
- The first item in a list is known as the ***head*** of the list.
- The remaining part is the ***tail****.*
- Note tail is a list whereas head is an element of a list.
  - In the above list `clare` is the head and
  - `[sean,richard,paula]` is the tail.
- We can also represent the list as
  - `[clare |[sean,richard,paula]]`
  - which would match `with [Head|Tail]`.

Pipe symbol **|** separates head from rest

Some Queries on List Patterns

```
?- spectrum(X).
X = [red, orange, yellow, green, blue, indigo, violet].
spectrum([red,orange,yellow,green,blue,indigo,violet]).
?- spectrum([X,Y]).
false.
?- spectrum(X).
X = [red, orange, yellow, green, blue, indigo, violet].
?- spectrum([X | Y]).
X = red,
Y = [orange, yellow, green, blue, indigo, violet].
?- spectrum([ [X] | Y]).
false.
?- spectrum([X,Y,Z | T]).
X = red,
Y = orange,
Z = yellow,
T = [green, blue, indigo, violet].
```

The argument is a ***list***

Asked for a list with ***exactly two terms***

The variable ***following pipe*** binds to a list

The first term is ***not a list***

Succeeds if the list contains ***at least three*** terms

Member Succeeding

```
member(H,[H|Tail]).
member(X,[H|Tail]):-
        member(X,Tail).
```

H is a member of a list if it is the 1st term

X is a member of a list if it is on the tail

- The query
  - | ?- `member(richard,[clare,sean,richard,paula]).`
- doesn't match with the first clause but does with the second,
  - `X=richard, H=clare,` and
  - `Tail=[sean,richard,paula]` to create the new subgoal `member(richard,[sean,richard,paula]).`
  - Next `X = richard, H = sean`
    `Tail = [richard,paula]`
  - giving the subgoal, `member(richard,[richard,paula]).`
  - Now matches the base case and succeeds.

Member Failing

- The query
  `?- member(john,[clare,sean,richard,paula]).`
- keeps recursively calling `member` as follows.
  `member(john,[sean,richard,paula]).`
  `member(john,[richard,paula]).`
  `member(john,[paula]).`
  `member(john,[]).`
- There is no rule to apply to the empty list so `member(john,[])` fails
- `member(X,[X|T])` needs ***at least one*** element to match the X

Debugging Prolog Programmes

- The debugging tool called tracing is invoked by typing trace. at the prompt. Several options but pressing return allows you to see the program trace.
- Advice:
  - Test smaller units e.g individual procedures.
  - Look out for infinite recursions.

Tracing a Successful Goal

```
?- trace.

Yes
[trace]  ?- member(richard,[clare,sean,richard,paula]).
   Call: (7) member(richard, [clare, sean, richard, paula]) ? creep
   Call: (8) member(richard, [sean, richard, paula]) ? creep
   Call: (9) member(richard, [richard, paula]) ? creep
   Exit: (9) member(richard, [richard, paula]) ? creep
   Exit: (8) member(richard, [sean, richard, paula]) ? creep
   Exit: (7) member(richard, [clare, sean, richard, paula]) ? creep

Yes
```

Tracing a Failing Goal

```
?- trace.

Yes
[trace]  ?- member(john,[clare,sean,richard,paula]).
   Call: (7) member(john, [clare, sean, richard, paula]) ? creep
   Call: (8) member(john, [sean, richard, paula]) ? creep
   Call: (9) member(john, [richard, paula]) ? creep
   Call: (10) member(john, [paula]) ? creep
   Call: (11) member(john, []) ? creep
   Fail: (11) member(john, []) ? creep
   Fail: (10) member(john, [paula]) ? creep
   Fail: (9) member(john, [richard, paula]) ? creep
   Fail: (8) member(john, [sean, richard, paula]) ? creep
   Fail: (7) member(john, [clare, sean, richard, paula]) ? creep

No
```

Append

- Append is an useful example of list processing.

```
/*************************************/
% append(L1,L2,L3)
% takes two lists L1 and L2 and returns a
% list L3 which is the result of appending
% L2 to L1
/*************************************/

append([],L2,L2).

append([H1|L1],L2,[H1|L3]):-
    append(L1,L2,L3).
```

Appending a list to the empty list gives that list

If the list is not empty:
The appended list is:
The head, followed by the tail appended to the second list

Append in Action

Base case

```
?- myappend([a,b],[c,d],E).
  Call: (7) myappend([a, b], [c, d], _G559) ?
  Call: (8) myappend([b], [c, d], _G668) ?
  Call: (9) myappend([], [c, d], _G671) ?
  Exit: (9) myappend([], [c, d], [c, d]) ?
  Exit: (8) myappend([b], [c, d], [b, c, d]) ?
  Exit: (7) myappend([a, b], [c, d], [a, b, c, d]) ?
E = [a, b, c, d].
```

Append Again

Last argument Instantiated. What lists append to give this one?

```
?-  myappend(A,B,[a,b,c]).
  Call: (8) myappend(_G762, _G763, [a, b, c]) ?
  Exit: (8) myappend([], [a, b, c], [a, b, c]) ?
A = [],
B = [a, b, c] ;
  Redo: (8) myappend(_G762, _G763, [a, b, c])
?
  Call: (9) myappend(_G876, _G763, [b, c]) ?
  Exit: (9) myappend([], [b, c], [b, c]) ?
  Exit: (8) myappend([a], [b, c], [a, b, c]) ?
A = [a],
B = [b, c] ;
  Redo: (9) myappend(_G876, _G763, [b, c]) ?
  Call: (10) myappend(_G879, _G763, [c]) ?
  Exit: (10) myappend([], [c], [c]) ?
  Exit: (9) myappend([b], [c], [b, c]) ?
  Exit: (8) myappend([a, b], [c], [a, b, c]) ?
A = [a, b],
B = [c] ;
Redo: (10) myappend(_G879, _G763, [c]) ?
  Call: (11) myappend(_G882, _G763, []) ?
  Exit: (11) myappend([], [], []) ?
  Exit: (10) myappend([c], [], [c]) ?
  Exit: (9) myappend([b, c], [], [b, c]) ?
  Exit: (8) myappend([a, b, c], [], [a, b, c]) ?
A = [a, b, c],
B = [] .
```

Arithmetic

- Prolog has several built in operators for arithmetic.
  - + addition
  - - subtraction
  - * multiplication
  - / division
  - ** power
  - // integer division
  - mod modulo, the remainder of integer division

Numeric comparison

- Similarly there are several built in comparison operators.
  - > greater than
  - < less than
  - >= greater than or equal to
  - =< less than or equal to
  - =:= is equal to
  - = \= is not equal to

Evaluating Arithmetic Operators

- The query

```
| ?- X = 3 + 5
X = 3 + 5
```

- does not evaluate the addition operation. The = instantiates the RHS to the LHS
- However the following evaluates the RHS.

```
| ?- X is 3 + 5
X = 8
```

Arithmetic is Procedural

- In order to use is we must have the right hand side instantiated
- Arithmetic is procedural: it wont tell us what numbers make 5.

```
mysum(A,B,C):- var(A),A is B + C.
mysum(A,B,C):- var(B),B is A - C.
mysum(A,B,C):- var(C),A is A - B.
```

- Still requires two arguments instantiated.

Example: Calculating the length of a list

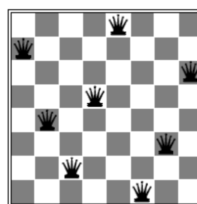
```
listlength([],0).
listlength([H|Tail],Len1):-
        listlength(Tail,Len2),
        Len1 is Len2 + 1.
```

- *How is length([a,b,c],X) calculated?*

List Length

```
Call: (7) listlength([a, b, c], _G598) ? creep
  Call: (8) listlength([b, c], _L205) ? creep
  Call: (9) listlength([c], _L224) ? creep
  Call: (10) listlength([], _L243) ? creep
  Exit: (10) listlength([], 0) ? creep
^  Call: (10) _L224 is 0+1 ? creep
^  Exit: (10) 1 is 0+1 ? creep
  Exit: (9) listlength([c], 1) ? creep
^  Call: (9) _L205 is 1+1 ? creep
^  Exit: (9) 2 is 1+1 ? creep
  Exit: (8) listlength([b, c], 2) ? creep
^  Call: (8) _G598 is 2+1 ? creep
^  Exit: (8) 3 is 2+1 ? creep
  Exit: (7) listlength([a, b, c], 3) ? creep
N = 3.
```

Example ProblemL N Queens



$n = 8$.

- This is a problem from chess.
- Place n queens on chess board so that no queen can be taken by another.

(A queen attacks any piece in the same row, column or diagonal.)

Problem Formulation

- States: List of positions of queens:
  - [(p(1, 7), p(2, 4), p(3, 5) ... p(8,1)]
- Initial state: Empty list []
- Goal: A list of n positions such that no queen can be taken by another

Operations

- Assume we have placed k queens on the board correctly represented by the list Others
- We form the new list
  - [p(X,Y) | Others]
- such that:
  - X and Y must be integers between 1 and 8.
  - A queen at square (X, Y) must not attack any of the queens in the list Others
- Start with the empty list []

n Queen Program

```
solution( [] ).

solution( [p(X,Y) | Others] )  :-      % First queen at p(X,Y),
  solution( Others),                   % other queens at Others
  member( Y, [1,2,3,4,5,6,7,8] ),
  member( X, [1,2,3,4,5,6,7,8] ),
  noattack( p(X,Y), Others).           % First queen does not attack others

noattack( _, [] ).                   % Nothing to attack

noattack( p(X,Y), [p(X1,Y1) | Others] )  :-
  Y =\= Y1,                          % Different Y-coordinates
  X =\= X1,                          % Different X-coordinates
  Y1-Y =\= X1-X,                     % Different diagonals
  Y1-Y =\= X-X1,
  noattack( p(X,Y), Others).
```

But this does not tell us that we have to place exactly 8 queens

A Solution Template

- Provide a template for the solution:
  - `Template1 ([p(X1,Y1),p(X2,Y2),p(X3,Y3),p(X4,Y4), p(X5,Y5),p(X6,Y6),p(X7,Y7),p(X8,Y8)] ).`
  - `?- template1(S),solution(S).`
- Eventually:
- `S = [p(4, 8), p(2, 7), p(7, 6), p(3, 5), p(6, 4), p(8, 3), p(5, 2), p(1, 1)]`

Better Solution

- `template2( [p(1,Y1),p(2,Y2),p(3,Y3),p(4,Y4), p(5,Y5),p(6,Y6),p(7,Y7),p(8,Y8)] ).`

We can use this because we know we cant have two queens in the same column

Avoids a lot of doomed attempts

Summary

- Lists are common data structures in Prolog.
- Procedures related to lists are often recursive.
  - Do it to the head, then do it to the rest
- Tracing a procedure can help you see what your program is doing.
- There are several built in operations for comparison and arithmetic.
  - To evaluate arithmetic you need the *is* construct. This is procedural only and the RHS must be fully instantiated