COMP219: Artificial Intelligence

Lecture 12: Prolog – Cut

Overview

- Last time
  - Lists in Prolog and debugging
- Today:
  - Built in operators for arithmetic and comparison
  - Another example with lists: the n-queens problem
  - Preventing backtracking (cut)
    - Green cuts and red cuts
- Learning outcome covered today:
  Understand and write Prolog code to solve simple knowledge-based problems.

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

```
X is 3 + 5.  X = 8.
X is 3 + Y.  Error:
```

- Arithmetic is procedural – it won't tell us what numbers make 5, which is a declarative notion

```
5 is X + Y.  Error:
```

Still requires two arguments instantiated on the right hand side

Example: Calculating the Length of a List

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

- How is `tlength([a,b,c],X)` calculated?

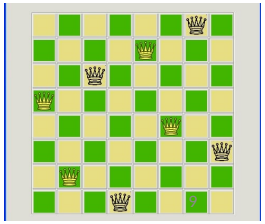
List Length

```
?- trace,tlength([a,b,c],X).
   Call: (7) tlength([a, b, c], _G1720) ? creep
   Call: (8) tlength([b, c], _G1846) ? creep
   Call: (9) tlength([c], _G1846) ? creep
   Call: (10) tlength([], _G1846) ? creep
   Exit: (10) tlength([], 0) ? creep
   Call: (10) _G1848 is 0+1 ? creep
   Exit: (10) 1 is 0+1 ? creep
   Exit: (9) tlength([c], 1) ? creep
   Call: (9) _G1851 is 1+1 ? creep
   Exit: (9) 2 is 1+1 ? creep
   Exit: (8) tlength([b, c], 2) ? creep
   Call: (8) _G1720 is 2+1 ? creep
   Exit: (8) 3 is 2+1 ? creep
   Exit: (7) tlength([a, b, c], 3) ? creep
X = 3.
```

Return to a familiar example: n-Queens

- Recall:
  - This is a problem from chess.
  - In the 8-queens version, place 8 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.
  - Has served as a useful test scenario for search algorithms.

- We will now look at how we can write a Prolog program to represent and solve this problem…
  (based on an example from Chapter 4 of the Prolog book by Bratko).



Problem Formulation

- States: List of positions of queens in terms of vertical and horizontal position. Each element of the list is a position of a queen
  - [p(1,2), p(2,5), p(3,7), p(4,4), p(5,1), p(6,8), p(7,6), p(8,3)]

  As no solution has queens in the same vertical, use a template to simplify the search space
  - [p(1,Y1), p(2,Y2), p(3,Y3), p(4,Y4), p(5,Y5), p(6,Y6), p(7,Y7), p(8,Y8)]

- Initial state: Empty list [], no queens on board, so no queen attacks any other queen

- Goal: A list of *n* positions such that no queen is attacked by another; that is, there are no two queens in the same column (done), row (easy), or diagonal (bit more complex)

Actions

- Suppose there are queens on the board, then how can we add another? The new one is p(X,Y), and the present queens is the list of Others. So, we form the new list:
  - [p(X,Y) | Others]
- such that:
  - No queen in Others attacks another queen in Others
  - X and Y must be integers between 1 and 8
  - A queen at p(X,Y) must not attack any of the queens in the list Others

Actions (continued)

- What is the attack relation?
- tnoattack(Q,Qlist)
  - If Qlist is empty, then there are no queens to attack, so this will be true
  - If Qlist is not empty, then Qlist has the form [Q1|Qlist1] where in tnoattack(Q,[Q1|Qlist1])
    - the queen at Q does not attack Q1, and
    - the queen at Q does not attack any queen in Qlist1
- Since we use a template, we only need to look at attacks with respect to Y so
  - the Y coordinates must be different
  - the queens are not in the same diagonal (the distance in the X-direction is not equal to the distance in the Y-direction)

n-Queens Program

```
tqueensolution([]).
tqueensolution([p(X,Y)|Others]):-          % Queen at p(X,Y), other queens at Others
  tqueensolution(Others),                  % No attacks between Others
  member(Y,[1,2,3,4,5,6,7,8]),             % Y must be integer between 1 - 8
  tnoattack(p(X,Y),Others).                % Queen at X,Y does not attack a queen in Others

tnoattack(_,[]).                           % Nothing to attack
tnoattack(p(X,Y), [p(X1,Y1)|Others]):-
  Y=\=Y1,                                  % Queens have different Ys
  Y1-Y=\=X1-X,                             % Queens have different diagonals
  Y1-Y=\=X-X1,
  tnoattack(p(X,Y),Others).

template([p(1,Y1), p(2,Y2), p(3,Y3), p(4,Y4), p(5,Y5), p(6,Y6), p(7,Y7), p(8,Y8)]).
                                           % A solution template to simplify the search space
```


Example

The Solution variable is shared by both functors

```
?- template(Solution),tqueensolution(Solution).

Solution = [p(1, 4), p(2, 2), p(3, 7), p(4, 3), p(5, 6), p(6, 8), p(7, 5), p(8, 1)] ;
Solution = [p(1, 5), p(2, 2), p(3, 4), p(4, 7), p(5, 3), p(6, 8), p(7, 6), p(8, 1)] ;
Solution = [p(1, 3), p(2, 5), p(3, 2), p(4, 8), p(5, 6), p(6, 4), p(7, 7), p(8, 1)] ;
Solution = [p(1, 3), p(2, 6), p(3, 4), p(4, 2), p(5, 8), p(6, 5), p(7, 7), p(8, 1)] ;
Solution = [p(1, 5), p(2, 7), p(3, 1), p(4, 3), p(5, 8), p(6, 6), p(7, 4), p(8, 2)] ;
and so on....
```


Search and Backtracking

- Prolog does depth-first search
- If a goal fails, Prolog will automatically *backtrack* and explore another possibility
  - Useful programming concept
  - BUT uncontrolled backtracking may lead to inefficiency (or worse)

www.runnersgoal.com

Example: Tax Rates

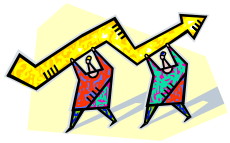
Fotolia

- Consider the following program that sets tax rates (0, 1, 2) for particular incomes (X)

```
tax_rate(X,0):- X < 10000.
tax_rate(X,1):- 10000 =< X, X < 30000.
tax_rate(X,2):- 30000 =< X.
```

- Note, we assume that X is instantiated to a number, as required by the comparison operators
- The goal `tax_rate(2000,Z)` will succeed with `Z=0`. That is, for an income of 2000, the tax rate is 0

Backtracking



- Now consider the query

  `?- tax_rate(2000,Z), 1 < Z.`

  'What is the tax rate where the income is 2000 and the tax band is greater than 1?'

- When executing `tax_rate(2000,Z)`, `Z` is instantiated as 0 since `tax_rate(2000,Z).` That is, it matches the first clause of the program `tax_rate(X,0):- X < 10000,` `X` = 2000
- The second goal of the query becomes 1 < 0, which fails
- At this point Prolog will try to backtrack and match the first goal with each of the other definitions for tax rate, which will also fail
- But, why backtrack at all?

```
?- trace,tax_rate(2000,Z), 1<Z.
   Call: (7) tax_rate(2000, _G1711) ? creep
   Call: (8) 2000<10000 ? creep
   Exit: (8) 2000<10000 ? creep
   Exit: (7) tax_rate(2000, 0) ? creep
   Call: (7) 1<0 ? creep
   Fail: (7) 1<0 ? creep
   Redo: (7) tax_rate(2000, _G1711) ? creep
   Call: (8) 10000=<2000 ? creep
   Fail: (8) 10000=<2000 ? creep
   Redo: (7) tax_rate(2000, _G1711) ? creep
   Call: (8) 30000=<2000 ? creep
   Fail: (8) 30000=<2000 ? creep
   Fail: (7) tax_rate(2000, _G1711) ? creep
false.
```

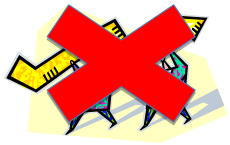
Introduces an internal variable _G1711 for Z, matches to the head of the first rule, then tests the body. The body is true, so we make Z = 0

Tests second part of query, where 1 < 0. Fails. Could stop here, but backtracks to try to find another solution

It tries the body of the 2nd clause, which fails

It tries the body of the 3rd clause, which fails again

Problem



- The problem is that the set of rules for tax_rate are *mutually exclusive*, i.e. for any value of X, the body of only one rule can succeed
- So we know that once 1 < 0 fails, there is no point in trying any other clause for tax_rate
- We do not want to backtrack after a solution is found because this is inefficient
- We ought to encode our knowledge in our rules
- Can be tricky where we have a very complicated set of rules

Cut

! means if you have tried this clause, and the previous body goal X < 10000 succeeds, then don't backtrack. Just give the result relative to this clause

- We can tell Prolog explicitly not to backtrack by using ***cut***, written as !
- The program with cuts becomes the following

```
tax_rate(X,0):- X < 10000, !.
tax_rate(X,1):- 10000 =< X, X < 30000, !.
tax_rate(X,2):- 30000 =< X.
```

No ! here since either succeeds or fails, and nothing to backtrack to

- Now with the query

  `?- tax_rate(2000,Z), 1 < Z.`
- We try the first clause, where `2000 < 10000` succeeds, then we get to !. We have `tax_rate(2000,0), Z = 0.` We test the second goal of the query, `1 < 0`, which fails
- But, since we have hit a cut, the program does not backtrack to explore the branches from rules 2 and 3. No other values for Z are considered, and the program is more efficient – it fails earlier

Cut

- Take the program

```
tax_rate(X,0):- X < 10000, !.
tax_rate(X,1):- 10000 =< X, X < 30000, !.
tax_rate(X,2):- 30000 =< X.
```

- With the query

```
?- tax_rate(40000,Z), 1 < Z.
```

The execution tries the first clause of the program. Since 40000 < 10000 fails, we do not reach the cut in the first clause. We do, then, backtrack to try the second and the third clauses. Since 30000 =< 40000 succeeds, we assign Z = 2, and test the second goal of the query, 1 < 2, which succeeds. So, we have found the solution Z =2

- The order of goals within a clause and cut may be important
- Experiment with different orders or having/not having cuts

Green Cuts and Red Cuts

- With this type of cut the program gives the same results as the version without cuts but is, in general, more efficient. Such cuts are called *green cuts*. They are a procedural convenience.

- The above is not always the case; using cuts can also change the results of the program - they can change the declarative meaning of the program. These are known as *red cuts*. Examples of red cuts occurring are where explicit conditions in a rule are omitted – exercise caution.

Red Cuts - Tax Program

- Now we remove the condition X >= 10000 in the second rule, the idea being that if we get to try the second rule then X < 10000 must have already failed

```
tax_rate(X,0):- X < 10000, !.   First rule
tax_rate(X,1):- X < 30000, !.   Half of second rule
tax_rate(X,2):- 30000 =< X.     Third rule
```

- Note that because of the cut the query

```
?- tax_rate(500,Z).
```

will only produce one result, i.e. Z=0.

- What would happen if we posed the same query to a program like the above, but without the cuts?
- Shows the difference in declarative meaning we can have with and without cuts

Summary

- We have looked at the use of cut to improve the efficiency of programs by stopping backtracking
- Green cuts prune branches (affect the procedural behaviour of the program) without changing the results of the program
- Red cuts both prune branches and affect the results
- Cuts may destroy the correspondence between the declarative and procedural semantics
- Use cuts with care. Only use cut with good reason

- Next time
  - Applying search to games