Logic Programming and Prolog (conclusion)

Finish reading: Scott, Chapter 12

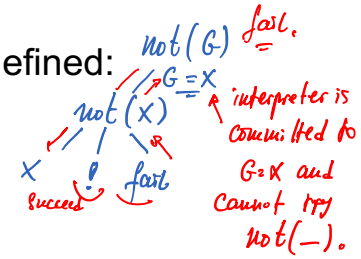
Lecture Outline

- *Prolog programming*
  - *Parsing and language processing*
  - *Arithmetic*
  - *Imperative control flow*
  - *Negation by failure*
  - *Generate and test paradigm*

Negation by Failure: `not(X)`, `\+(X)`

- **`not(G)`** succeeds when goal `G` cannot be made to succeed
  - Called negation by failure, defined:

```
not(X) :- X,!,fail.
not(_).
```



- Not the same as negation in logic ¬G!
- In Prolog, we can assert that something is true, but we cannot assert that something is false

Exercise

```
takes(jane, his).
takes(jane, cs).
takes(ajit, art).
takes(ajit, cs).
classmates(X,Y) :- takes(X,Z),takes(Y,Z).

?- classmates(jane,Y).
```

What are the bindings of `Y`?

Exercise

```
p(X) :- q(X), not(r(X)).
r(X) :- w(X), not(s(X)).
q(a). q(b). q(c).
s(a). s(c).
w(a). w(b).
```

- Evaluate:
  - `?- p(a).`
  - `?- p(b).`
  - `?- p(c).`

Lecture Outline

- *Prolog programming*
  - *Parsing and language processing*
  - *Arithmetic*
  - *Imperative control flow*
  - *Negation by failure*
  - *Generate and test paradigm*

Generate and Test Paradigm

- Search in space
- Prolog rules to generate potential solutions
- Prolog rules to test potential solutions for desired properties
- Easy prototyping of search

```
solve(P) :- generate(P), test(P).
```


A Classical Example: n Queens

- Given an n by n chessboard, place each of n queens on the board so that no queen can attack another in one move
  - Queens can move either vertically,
  - horizontally, or
  - diagonally.
- A classical generate and test problem

n Queens

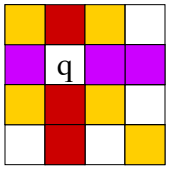
```
my_not(X):- X, !, fail.   %same as not
my_not(_).
in(H,[H|_]).              %same as member
in(H,[_|T]):- in(H,T).


nums(H,H,[H]).
nums(L,H,[L|R]):- L<H, N is L+1, nums(N,H,R).



queen_no(4).
```


n Queens (ii)

```
ranks(L):- queen_no(N), nums(1,N,L).
files(L):- queen_no(N), nums(1,N,L).


rank(R):- ranks(L), in(R,L).




file(F):- files(L), in(F,L).
```

n Queens (iii)

```
attacks((R,_),(R,_)).
attacks((_,F),(_,F)).  %a Prolog tuple
attacks((R1,F1),(R2,F2)):-
                    diagonal((R1,F1),(R2,F2)).
%%%can decompose a Prolog tuple by unification
  (X,Y)=(1,2) results in X=1,Y=2; tuples have fixed
  size and there is not head-tail type construct for
  tuples
```



same rank
same file
same diagonal

What is safe placement for next queen on board?

12

12

n Queens (iv)

```
%%% Two squares are on the same diagonal if the slope of
  the line between them is 1 or -1.  Since / is used, real
  number values for 1 and -1 are needed.

diagonal((X,Y),(X,Y)). %degenerate case
diagonal((X1,Y1),(X2,Y2)):-N is Y2-Y1,D is X2-X1,
  Q is N/D, Q is 1 . %diagonal needs bound
  arguments!
diagonal((X1,Y1),(X2,Y2)):-N is Y2-Y1,D is X2-X1,
  Q is N/D, Q is -1 .
%%%because of use of "is", diagonal is NOT invertible.
```


13

n Queens (v)

```
%%% This solution works by generating every list of
  squares, such that the length of the list is the same
  as the number of queens, and then checks every list
  generated to see if it represents a valid placement of
  queens to solve the N queens problem;
  assume list length function
queens(P):- queen_no(N), length(P,N),
  placement(P), ok_place(P).
```

"generate" code given first "test" code follows

n Queens (vi)

```
placement([]).
placement([(R,F)|P]):- placement(P), rank(R), file(F).
```

n Queens (vii)

```
ok_place([]).
ok_place([(R,F)|P]):- no_attacks((R,F),P),ok_place(P).


no_attacks(_,[]).
no_attacks((R,F),[(R2,F2)|P]):-
  my_not(attacks((R,F),(R2,F2))), no_attacks((R,F),P).
```


16

The End

17