Logic Programming and Prolog

Finish reading: Scott, Chapter 12
Lecture Outline

- Quiz 3
- Prolog
  - Imperative control flow
  - Negation by failure
  - Generate and test paradigm

\( \neg t(x) \)
Imperative Control Flow

- Programmer has **explicit control** on backtracking process

**cut (!)**

- ! is a subgoal

- As a goal it succeeds, but with a **side effect**:
  - Commits interpreter to **all bindings** made since unifying left-hand side of current rule with parent goal
Cut (!) Example

rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).

?- snowy(C).
?- false.
Cut (!) Example

rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).

_x = X
C = _C

GOAL FAILS.
Cut (!) Example 2

rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).
snowy(troy).

?- snowy(C).
Cut (!) Example 2

rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).
snowy(troy).

2 committed OR bindings:
_\_C = _\_X
and X = seattle

GOAL FAILS.

How about query ?- snowy(troy)?
?-. true.
rainy(seattle) :- !.
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
snowy(troy).

?- snowy(C).
?- C = Troy.
Cut (!) Example 3

rainy(seattle) :- !.
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
snowy(troy).

How about query? - snowy(rochester)?
Cut (!) Example 4

rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- !, rainy(X), cold(X).

?- snowy(C).
?- rochester.
Cut (!) Example 4

rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- !, rainy(X), cold(X).

X = seattle
rainy(seattle) OR rainy(rochester)
cold(seattle) fails; backtrack.

X = rochester
cold(rochester)
snowy(X) AND

_C = _X

success
Cut (!) Example 5

rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X), !.

?- snowy(C).
?- C = rochester.
Cut (!) Example 5

rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X), !.

编程语言 CSCI 4430，A. Milanova
Negation by Failure: \( \text{not}(X), \ \backslash + (X) \)

- \( \text{not}(C) \) succeeds when \( C \) fails
  - Called *negation by failure*, defined:
    \[
    \text{not}(X) :- X!,!,\text{fail}.
    \]
    \[
    \text{not}(\_).
    \]
  - don't care.

- Not the same as negation in logic \( \neg X \! \)

- In Prolog, we can assert that something is true, but we **cannot** assert that something is false
  \[
  \text{likes}(\text{eve}, \text{pie}). \ \neg \text{likes}(\text{eve}, \text{pie}).
  \]
Exercise

takes(jane, his).
takes(jane, cs).
takes(ajit, art).
takes(ajit, cs).

classmates(X,Y) :- takes(X,Z), takes(Y,Z),
\text{not}(X = Y).

?- classmates(jane, Y).

What are the bindings of $Y$?

?- $Y = \text{jane}$ ; // $Y = \text{jane}$, $Z = \text{his}$
?- $Y = \text{jane}$ ; // $Z = \text{cs}$
?- $Y = \text{ajit}$ ; // $Z = \text{cs}$
Exercise

- \( p(X) :- q(X), \neg r(X) \).
- \( r(X) :- w(X), \neg s(X) \).
- \( q(a) \), \( q(b) \), \( q(c) \).
- \( s(a) \), \( s(c) \).
- \( w(a) \), \( w(b) \).

Evaluate:

- \( ?- p(a) \). true
- \( ?- p(b) \). false
- \( ?- p(c) \). true
Lecture Outline

- Prolog
  - Imperative control flow
  - Negation by failure
  - Generate and test paradigm
Generate and Test Paradigm

- Search in space

- Prolog rules to \textit{generate} potential solutions
  
- Prolog rules to \textit{test} potential solutions for desired properties

- Easy prototyping of search
  
  ```prolog
  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]).          %result of toplevel call
nums(L,H,[L|R]):- L<H, N is L+1, nums(N,H,R).

%%%nums generates a list of integers between two other
    %numbers, L,H by putting the first number at the front of the list returned by a recursive call with a number 1 greater than the first. It works when L and H are bound to integers. It stops when it gets to the higher number

queen_no(4).

%%%The number of queens/size of board - use 4

nums(1,4,R).
R = [1,2,3,4].
n Queens (ii)

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

%%% ranks and files generate the x and y axes of the chess board. Both are lists of numbers up to the number of queens; that is, ranks(L) binds L to the list [1,2,3,...,#queens].

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

%%% R is a rank on the board; selects a particular rank R from the list of all ranks L.

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

%%% F is a file on the board; selects a particular file F from the list of all files L.
n Queens (iii)

%% Squares on the board are (rank, file) coordinates.

attacks decides if a queen on the square at rank R1, file F1 attacks the square at rank R2, file F2 or vice versa. A queen attacks every square on the same rank, the same file, or the same diagonal.

attacks((R, _), (R, _)).
attacks((_, F), (_, F)).
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

What is safe placement for next queen on board?
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.
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
%%placement can be used as a generator. If placement is called with a free variable, it will construct every possible list of squares on a chess board.

The first predicate will allow it to establish the empty list as a list of squares on the board. The second predicate will allow it to add any (R,F) pair onto the front of a list of squares if R is a rank of the board and F is a file of the board.

placement first generates all 1 element lists, then all 2 element lists, etc. Switching the order of predicates in the second clause will cause it to try varying the length of the list before it varies the squares added to the list.

placement([]).

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

\[ P = [x_1, x_2, x_3, x_4] \]
n Queens (vii)

%%%these two routines check the placement of the next queen

%%%Checks a list of squares to see that no queen on any of them would attack any other. does by checking that position j doesn’t conflict with positions (j+1),(j+2) etc.

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

%%% Checks that a queen at square (R,F) doesn't attack any square (rank,file pair) in list L; uses attacks predicate defined previously

no_attacks(_,[]).
no_attacks((R,F),[(R2,F2)|P]):— my_not(attacks((R,F),(R2,F2))), no_attacks((R,F),P).
Homework Solution Structure

- Typical Prolog homework: search in space (e.g., paths in a maze, paths in graph, parsing sequences, various puzzles)

- Typical solution:

```prolog
search(F, Partial, Total) :-
  final(F), ... % get Total from Partial

search(C, Partial, Total) :-
  generate(C, N), % generate next position
  valid(N), ... % test if N is a valid position
  augment(Partial, New_partial),
  % augment Partial solution with N, typically we would need not(member(N, Partial)) too.
  search(N, New_partial, Total).
```
A Harder Exercise

- Remember grammar from one of the quizzes...
  1. \( S \rightarrow aSbS \)
  2. \( S \rightarrow bSaS \)
  3. \( S \rightarrow \varepsilon \)

- Write a **top-down depth-first** parser in Prolog:
  ```prolog
  ?- parse([a,b,a,b], R).
  R = [1, 2, 3, 3, 3] ; // seq. of productions
  R = [1, 3, 1, 3, 3] ; // different seq
  false. // no more seqs
  ```

- Hint: break list into constituent parts
\[ \text{Search} \left( P, \text{Final} \right) \leftarrow \text{queens\_no} \left( N \right), \text{length} \left( P, N \right), \text{Final} = P. \]

\[ \% \text{ Search} \left( P, P \right) \leftarrow \text{queens\_no} \left( N \right), \text{length} \left( P, N \right). \]

\[ \text{Search} \left( P, \text{Final} \right) \leftarrow \text{range} \left( R \right), \text{file} \left( F \right), \text{no\_attacks} \left( \left( R, F \right), P \right), \text{Search} \left( \left[ \left( R, F \right) \mid P \right], \text{Final} \right). \]

\[ \text{Solve} \left( P \right) \leftarrow \text{Search} \left( [], P \right). \]