Announcements

- HW1 is graded
  - Re-grade requests until Friday
- Quiz 1 graded too
- HW3 due Friday

Last Class

- Prolog
  - Language constructs: facts, rules and queries
  - Search tree, rule ordering, unification, backtracking, backward chaining
  - Lists

Today’s Lecture Outline

- Prolog
- Lists
- Arithmetic
- Imperative control flow (backtracking cut)
- Generate and test paradigm

Logic Programming and Prolog

Keep reading: Scott, Chapter 12

Member_of “Procedure”

?- member(a,[a,b]).
   true.
?- member(a,[b,c]).
   false.

?- member(X,[a,b,c]).
   X = a ;
   X = b ;
   X = c ;
   false.

Append “Procedure”

append([], A, A).
append([A|B], C, [A|D]) :- append(B,C,D).

- Build a list:
  ?- append([a,b,c],[d,e],Y).
  Y = [a,b,c,d,e]
- Break a list into constituent parts:
  ?- append(X,Y,[a,b]).
  X = []; Y = [a,b]; X = [a], Y = [b]; X = [a,b], Y = []; false.
Unbounded Arguments

- Generating an unbounded number of lists

```
?- append(X, [b], Y).
X = [ ]
Y = [ b] ;
X = [ _G604]
Y = [ _G604, b] ;
X = [ _G604, _G610]
Y = [ _G604, _G610, b] ;
Etc.
```

- Be careful when using append with 2 unbounded arguments!!!

Common Structure

- "Processing" a list:

```
proc([], []).
proc([H|T], [H1|T1]) :- f(H, H1), proc(T, T1).
```

- Base case: we have reached the end of list.
  In our case, the result for [ ] is [ ].
- Recursive case: result is [H1|T1]. H1 was obtained by calling f(H, H1) --- processes element H into result H1. T1 is the result of recursive call of proc on T.

Lecture Outline

- Prolog
  - Lists
  - Arithmetic
  - Imperative control flow
  - Generate and test paradigm

Arithmetic

- Prolog has all arithmetic operators
- Built-in predicate is
  - is(X, 1+3) or more commonly we write
  - X is 1+3
  - is forces evaluation of 1+3:
    - ?- X is 1+3
    - X = 4
  - = is unification not assignment!
    - ?- X = 4-1.
    - X = 4-1 % unifies X with 4-1!!!

Arithmetic: Pitfalls

- is is not invertible! That is, arguments on the right cannot be unbound!
  - 3 is 3 - X.
  - ERROR: is/2: Arguments are not sufficiently instantiated

- This doesn’t work either:
  - ?- X is 4, X = X+1.
  - false.
  - Why? What is going on here?
Exercise

- Write `sum`, which takes a list of integers and computes the sum of the integers. E.g.,
  \[
  \text{sum}([1,2,3], R).
  \]
  \?- R = 6.

- How about if the integers are arbitrarily nested? E.g.,
  \[
  \text{sum}([[1],[[2]],3], R).
  \]
  \?- R = 6.

Exercise

- Write `plus10`, which takes a list of integers and computes another list, where all integers are shifted +10. E.g.,
  \[
  \text{plus10}([1,2,3], R).
  \]
  \?- R = [11,12,13].

- Write `len`, which takes a list and computes the length of the list. E.g.,
  \[
  \text{len}([1,[2],3], R).
  \]
  \?- R = 3.

Exercise

- Write `atoms`, which takes a list and computes the number of atoms in the list. E.g.,
  \[
  \text{atoms}([a,[b,[[c]]]], R).
  \]
  \?- R = 3.

- Hint: built-in predicate \texttt{atom(X)} yields true if \texttt{X} is an atom (i.e., symbolic constant such as \texttt{x}, \texttt{abc}, \texttt{tom}).

Lecture Outline

- Prolog
  - Lists
  - Arithmetic
  - Imperative control flow
  - Generate and test paradigm

Imperative Control Flow

- Programmer has explicit control on backtracking process

\texttt{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

\[
\begin{align*}
\text{rainy}(\text{seattle}). \\
\text{rainy}(\text{rochester}). \\
\text{cold}(\text{rochester}). \\
\text{snowy}(X) & : = \text{rainy}(X),!, \text{cold}(X).
\end{align*}
\]

\?- \text{snowy}(C).
Cut (!) Example

\[
\text{rainy(seattle).} \\
\text{rainy(rochester).} \\
\text{cold(rochester).} \\
\text{snowy(X) :- rainy(X), !, cold(X).} \\
\]

GOAL FAILS.

How about query \(-\text{snowy(rochester)?}\)

Cut (!) Example 2

\[
\text{rainy(seattle).} \\
\text{rainy(rochester).} \\
\text{cold(rochester).} \\
\text{snowy(X) :- rainy(X), !, cold(X).} \\
\text{snowy(troy).} \\
\]

?\(-\text{snowy}(C).\)

Cut (!) Example 3

\[
\text{rainy(seattle) :- !.} \\
\text{rainy(rochester).} \\
\text{cold(rochester).} \\
\text{snowy(X) :- rainy(X), cold(X).} \\
\text{snowy(troy).} \\
\]

?\(-\text{snowy}(C).\)

Cut (!) Example 4

\[
\text{rainy(seattle).} \\
\text{rainy(rochester).} \\
\text{cold(rochester).} \\
\text{snowy(X) :- !, rainy(X), cold(X).} \\
\text{snowy(troy).} \\
\]

?\(-\text{snowy}(C).\)
Cut (!) Example 4

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

Cut (!) Example 5

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

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?
How can we change rule classmates(X, Y) to prevent binding Y=jane?
Lecture Outline

- Prolog
- Lists
- Arithmetic
- Imperative Control Flow
- 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
  \[
  \text{solve}(F) :- \text{generate}(F), \text{test}(F).
  \]

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

\[
\text{my_not}(X) : :- X, !, \text{fail}. \quad \text{\# same as not}
\]
\[
\text{my_not}(_) .
\]
\[
\text{in}(H, [H|\_]) . \quad \text{\# same as member}
\]
\[
\text{in}(H, [\_|T]) :- \text{in}(H, T).
\]
\[
\text{nums}(H,N,[H]).
\]
\[
\text{nums}(L,N,[L|R]) :- L<H, N is L+1, \text{nums}(N,H,R).
\]

\[
\text{n Queens}
\]
\[
\text{my_not}(X) : :- X, !, \text{fail}. \quad \text{\# same as not}
\]
\[
\text{my_not}(\_).
\]
\[
\text{in}(H, [H|\_]) . \quad \text{\# same as member}
\]
\[
\text{in}(H, [\_|T]) :- \text{in}(H, T).
\]
\[
\text{nums}(H,N,[H]).
\]
\[
\text{nums}(L,N,[L|R]) :- L<H, N is L+1, \text{nums}(N,H,R).
\]

\[
\text{n Queens (ii)}
\]
\[
\text{ranks}(L) : :- \text{queen_no}(N), \text{nums}(1,N,L).
\]
\[
\text{files}(L) : :- \text{queen_no}(N), \text{nums}(1,N,L).
\]

\[
\text{rank}(R) : :- \text{ranks}(L), \text{in}(R,L).
\]
\[
\text{file}(F) : :- \text{files}(L), \text{in}(F,L).
\]

\[
\text{n Queens (iii)}
\]
\[
\text{attacks}((R_1,F_1),(R_2,F_2)) :- \text{same diagonal}
\]
\[
\text{attacks}((R_1,F_1),(R_2,F_2)) :- \text{same rank}
\]
\[
\text{attacks}((R_1,F_1),(R_2,F_2)) :- \text{same file}
\]

\[
\text{What is safe placement for next queen on board?}
\]
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.

\[\text{diagonal}((X,Y),(X,Y)). \]  
\[\text{degenerate case} \]
\[\text{diagonal}((X1,Y1),(X2,Y2)):-\text{N is Y2-Y1,D is X2-X1,} \]
\[\text{Q is N/D, Q is } 1 \].  
\[\text{diagonal needs bound arguments!} \]
\[\text{diagonal}((X1,Y1),(X2,Y2)):-\text{N is Y2-Y1,D is X2-X1,} \]
\[\text{Q is N/D, Q is } -1 \].  
\[\text{because of use of "is", diagonal is NOT invertible.} \]

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;

\[\text{assume list length function} \]
\[\text{queens(P):- queen_no(N), length(P,N),} \]
\[\text{placement(P), ok_place(P).} \]

“generate” code given first  “test” code follows

n Queens (vi)

%%% 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 to 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. The third predicate will allow it to try varying the length of the list before it varies the squares added to the list:

\[\text{placement}([\]). \]
\[\text{placement}([R,F]\text{|P}):-\text{placement(P), rank(R), file(F).} \]

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.

\[\text{ok_place([\]).} \]
\[\text{ok_place([R,F]\text{|P}):-\text{no_attacks((R,F),P),ok_place(P).}} \]

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

\[\text{no_attacks((R,F),(R2,F2)|P):- my_not(attacks((R,F),(R2,F2))), no_attacks((R2,F2)|P).} \]

A Harder Exercise

- Remember the grammar...
  1. \(S \rightarrow aSbS\)
  2. \(S \rightarrow bSaS\)
  3. \(S \rightarrow \epsilon\)

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

- Hint: break the list into constituent parts
Next class

- New topic: Binding, naming and scoping
- Review for Exam 1