ICOM 4036: PROGRAMMING LANGUAGES

Lecture 5
Logic Programming

5/11/2004
What is Prolog

- Prolog is a ‘typeless’ language with a very simple syntax.
- Prolog is declarative: you describe the relationship between input and output, not how to construct the output from the input (“specify what you want, not how to compute it”)
- Prolog uses a subset of first-order logic
First-Order Logic

Simplest form of logical statement is an *atomic formula*. An assertion about objects.

Examples:

- is-man(tom)
- is-woman(mary)
- married-to(tom, mary)
- mother-of(mary, john)
More complex formulas can be built up using *logical connectives*:

- **Men and Women are humans**
  - \(\forall X [\text{is-men}(X) \lor \text{is-woman}(X) \rightarrow \text{is-human}(X)]\)

- **Somebody is married to Tom**
  - \(\exists X \ \text{married-to}(\text{tom}, X)\)

- **Some woman is married to Tom**
  - \(\exists X [\text{married-to}(\text{tom}, X) \land \text{is-woman}(X)]\)

- **John has a mother**
  - \(\exists X \ \text{mother-of}(X, \text{john})\)

- **Two offspring of the same mother are siblings**
  - \(\forall X \ \forall Y \ \forall Z [\text{mother-of}(Z, X) \land \text{mother-of}(Z, Y) \rightarrow \text{siblings}(X, Y)]\)

- \(\exists\) is the Existential quantifier
- \(\forall\) is the Universal quantifier
Logical Inference

Example 2: Given these facts:

- is-man(carlos)
- is-man(pedro)

and this rule:

\[ \forall X \ [\text{is-mortal}(X) \leftarrow \text{is-man}(X)] \]

derive:

- is-mortal(carlos), is-mortal(pedro).
Logic programming is based on a simple idea: From facts and inferences try to prove more facts or inferences.
A rule:
\[ \forall X \ [p(X) \leftarrow (q(X) \land r(X))] \]

is written as
\[ p(X) \leftarrow q(X), r(X). \]

Prolog conventions:
- *variables* begin with upper case (A, B, X, Y, Big, Small, ACE)
- *constants* begin with lower case (a, b, x, y, plato, aristotle)
/* list of facts in prolog, stored in an ascii file, ‘family.pl’*/

mother-of(mary, ann).
mother-of(mary, joe).
mother-of(sue, mary).
father-of(mike, ann).
father-of(mike, joe).
grandparent-of(sue, ann).

/* reading the facts from a file */
?- consult ( ‘family.pl’ ).

family.pl compiled, 0.00 sec, 828 bytes
Prolog Evaluation

?- mother-of(sue, mary).
Yes

?- mother-of(sue, ann).
no

?- father-of( X, Y ).
X = mike;
Y = joe ;
no

% Prolog returns these solutions one at a time, in the order it finds them. You can press semicolon (;) to repeat the query and find the next solution. Prolog responds “no” when no more valid variable bindings of the variables can be found.
/* Rules */

parent-of( X , Y ) :- mother-of( X , Y ).
% if mother(X,Y)  then  parent(X,Y)

parent-of( X , Y ) :- father-of( X , Y ).
% if father(X,Y)  then  parent(X,Y)

grandparent( X , Z ) :- parent-of( X , Y ), parent-of(Y, Z ).
% if parent(X,Y) and parent(Y,Z) then grandparent(X,Z)

:= means ←
?- parent-of( X , ann), parent-of( X , joe).
X = mary;
X = mike;
no

?- grandparent-of(sue, Y ).
Y = ann;
Y = joe;
no
/* specification of factorial n! */
factorial(0,1).
factorial(N, M) :- N1 is N - 1,
                 factorial(N1, M1),
                 M is N*M1.

Takes 1 assertion and 1 inference
Factorial in Prolog - Evaluation

?- factorial (2, X).
  factorial (0, 1). /* fails */
  factorial (2, M) :- 1 is 2 - 1,
                   factorial (1, M1),
                   M is N * M1.

M = X, N = 2, N1 = 1

?- factorial (1, X1).
  (X1 is the M1 above)
  factorial (0, 1). /* fails */
  factorial (1, M) :- 0 is 1 - 1,
                   factorial (0, M1),
                   M is N * M1.

M = X1, N = 1, N1 = 0

?- factorial (0, X2).
  (X2 is the M1 above)
  factorial (0, 1). /* succeeds */
  /* after the first rule succeeds, the second rule is not used */
mylength([], 0).
mylength([X | Y], N):− mylength(Y, Nx), N is Nx+1.

? – mylength([1, 7, 9], X).
X = 3

? - mylength(jim, X).
No

? - mylength(Jim, X).
Jim = []
X = 0
**List Membership**

mymember( X , [X | Y] ).

?- mymember(a, [b, c, 6] ).
no

?- mymember(a, [b, a, 6] ).
yes

?- mymember( X , [b, c, 6] ).
X = b;
X = c;
X = 6;
no
Appending Lists

The Problem: Define a relation $\text{append}(X, Y, Z)$ as $X$ appended to $Y$ yields $Z$
Appending Lists

The Problem: Define a relation append(X, Y, Z) to mean that X appended to Y yields Z

append([], Y, Y).
append([H|X], Y, [H|Z]) :-
    append(X, Y, Z).
Appending Lists

?- append([1,2,3,4,5],[a,b,c,d],Z).
Z = [1,2,3,4,5,a,b,c,d];
no

?- append(X,Y,[1,2,3]).
X = [] Y = [1,2,3];
X = [1] Y = [2,3];
X = [1,2] Y = [3];
X = [1,2,3] Y = [];
no

Prolog Computes ALL Possible Bindings!
Control in Prolog

Prolog tries to solve the clauses from left to right. If there is a database file around, it will be used in a similarly sequential fashion.

1. Goal Order: Solve goals from left to right.
2. Rule Order: Select the first applicable rule, where first refers to their order of appearance in the program/file/database.
The actual search algorithm is:
1. start with a query as the current goal.
2. WHILE the current goal is non-empty
   DO choose the leftmost subgoal;
   IF a rule applies to the subgoal
   THEN select the first applicable rule;
   form a new current goal;
   ELSE backtrack;
ENDWHILE
SUCCEED
Control in Prolog

- Thus the order of the queries is of paramount importance.
- The general paradigm in Prolog is Guess then Verify: Clauses with the fewest solutions should come first, followed by those that filter or verify these few solutions.
Fibonacci in Prolog

fib1(1, 1).
fib1(2, 1).
fib1(N1, F1) :-
    N1 > 2,
    N2 is N1 - 1,
    N3 is N1 - 2,
    fib1(N2, F2),
    fib1(N3, F3),
    F1 is F2 + F3.
More List Processing

remove(X, L1, L2) ~ sets L2 to the list obtained by removing the first occurrence of X from list L1

remove(X, [X|Rest], Rest).
remove(X, [Y|Rest], [Y|Rest2]) :-
    X \== Y,
    remove(X, Rest, Rest2).
replace(X, Y, L1, L2) ~ sets L2 to the list obtained by replacing all occurrences of X in list L1 with Y

replace(_, _, [], []). replace(_, _, L1, L2) :- replace(X, Y, X|Rest, Y|Rest2). replace(X, Y, Z|Rest, Z|Rest2) :- Z \== X, replace(X, Y, Rest, Rest2).

replace(X, Y, [X|Rest], [Y|Rest2]) :- replace(X, Y, Rest, Rest2).
Write a predicate `insert(X, Y, Z)` that can be used to generate in `Z` all of the ways of inserting the value `X` into the list `Y`.

\[
\begin{align*}
\text{insert}(X, [], [X]). \\
\text{insert}(X, [Y|\text{Rest}], [X,Y|\text{Rest}]). \\
\text{insert}(X, [Y|\text{Rest}], [Y|\text{Rest2}]) :&- \\
&\text{insert}(X, \text{Rest}, \text{Rest2}).
\end{align*}
\]
Write a predicate permutation(X, Y) that can be used to generate in Y all of the permutations of list X

permutation([], []).

permutation([X|Rest], Y) :-
    permutation(Rest, Z), insert(X, Z, Y).
Graphs in Prolog

path(a,b).
path(b,c).
path(c,d).
path(d,b).
path(a,c).

Write a predicate route(X,Y) that succeeds if there is a connection between X and Y.

Route(X,X).
Route(X,Y):- path(X,Z), route(Z,Y).
Binary Search Trees in Prolog

\[
\text{\texttt{bstree}}::= \text{empty}
\]

\[
\quad \text{node}(<\text{number}>, \text{bstree}, \text{bstree})
\]

\[
\text{node}(15, \text{node}(2, \text{node}(0, \text{empty}, \text{empty}), \text{empty}),
\quad \text{node}(10, \text{node}(9, \text{node}(3, \text{empty}, \text{empty}),
\quad \quad \quad \quad \text{empty}),
\quad \quad \quad \quad \text{node}(12, \text{empty}, \text{empty}))),
\quad \text{node}(16, \text{empty}, \text{node}(19, \text{empty}, \text{empty})))
\]
isbtree(empty).
isbtree(node(N,L,R)):- number(N), isbtree(L), isbtree(R), smaller(N,R), bigger(N,L).

smaller(N,empty).
smaller(N, node(M,L,R)):- N < M, smaller(N,L), smaller(N,R).

bigger(N, empty).
bigger(N, node(M,L,R)):- N > M, bigger(N,L), bigger(N,R).
Binary Search Trees

?- [btree].
?- isbtree(node(6,node(9,empty,empty),empty)). no

?- isbtree(node(9,node(6,empty,empty),empty)). yes
Define a relation which tells whether a particular number is in a binary search tree.

\texttt{mymember(N,T)} should be true if the number \texttt{N} is in the tree \texttt{T}.

\begin{verbatim}
mymember(K,node(K,_,_)).
mymember(K,node(N,S,_)) :- K < N, mymember(K,S).
mymember(K,node(N,_,T)) :- K > T, mymember(K,T).
\end{verbatim}
?-
    mymember(3, node(10, node(9, node(3, empty, empty), empty), empty), node(12, empty, empty)).

yes
Sublists (Goal Order)

myappend([], Y, Y).
myappend([H|X], Y, [H|Z]) :-
    myappend(X, Y, Z).

myprefix(X, Z) :- myappend(X, Y, Z).
mysuffix(Y, Z) :- myappend(X, Y, Z).

Version 1
sublist1(S, Z) :-
    myprefix(X, Z), mysuffix(S, X).

Version 2
sublist2(S, Z) :-
    mysuffix(S, X), myprefix(X, Z).
Sublists

?- [sublist].

?- sublist1([e], [a,b,c]).
  no

?- sublist2([e], [a,b,c]).
  Fatal Error: global stack
  overflow ...
So what’s happening? If we ask the question:

\(\text{sublist1([e], [a,b,c])}.\)

this becomes

\(\text{prefix(X,[a,b,c]), suffix([e],X)}.\)

and using the *guess-query* idea we see that the first goal will generate four guesses:

\[
[\text{}] \quad [\text{a}] \quad [\text{a,b}] \quad [\text{a,b,c}]
\]

none of which pass the *verify* goal, so we fail.
On the other hand, if we ask the question:
\[
\text{sublist2([e], [a,b,c]).}
\]
this becomes
\[
\text{suffix([e],X),prefix(X,[a,b,c]).}
\]
using the guess-query idea note that the goal will generate an \textit{infinite} number of guesses.
\[
[e] \ [_,e] \ [_,_,e] \ [_,_,_,e] \ [_,_,_,_,e]
\]
None of which pass the verify goal, so we never terminate!!
You can move N disks from A to C in three general recursive steps.

- Move N-1 disks from A pole to the B pole using C as auxiliary.
- Move the last (Nth) disk directly over to the C pole.
- Move N-1 disks from the B pole to the C pole using A as auxiliary.
Towers of Hanoi

loc := right; middle; left

hanoi(integer)
move(integer, loc, loc, loc)
inform(loc, loc)

inform(Loc1, Loc2):-
  write("\nMove a disk from ", Loc1, " to ", Loc2).
hanoi(\(N\)):-
    move(\(N\),left,middle,right).

move(1,A,_,C) :- inform(A,C),!.
move(\(N\),A,B,C):-
    N1 is \(N-1\),
    move(N1,A,C,B),
    inform(A,C),
    move(N1,B,A,C).
construct an exclusive OR circuit from AND, OR, and NOT circuits, and then check its operation
**Logic Circuits: Prolog Model**

- `not_(D,D)`
- `and_(D,D,D)`
- `or_(D,D,D)`
- `xor_(D,D,D)`

- `not_(1,0).`  `not_(0,1).`
- `and_(0,0,0).`  `and_(0,1,0).`
- `and_(1,0,0).`  `and_(1,1,1).`
- `or_(0,0,0).`  `or_(0,1,1).`
- `or_(1,0,1).`  `or_(1,1,1).`

- `xor_(Input1,Input2,Output):-`
  - `not_(Input1,N1),`
  - `not_(Input2,N2),`
  - `and_(Input1,N2,N3),`
  - `and_(Input2,N1,N4),`
  - `or_(N3,N4,Output).`
Symbolic Differentiation

\[
\text{EXP} := \text{var(STRING)}; \ 
\text{int(INTEGER)}; \ 
\text{plus(EXP, EXP)}; \ 
\text{minus(EXP, EXP)}; \ 
\text{mult(EXP, EXP)}; \ 
\text{div(EXP, EXP)}; \ 
\text{ln(EXP)}; \ 
\text{potens(EXP, EXP)}; \ 
\text{cos(EXP)}; \ 
\text{sin(EXP)}; \ 
\text{tan(EXP)}; \ 
\text{sec(EXP)}. \]

Symbolic Differentiation

d(int(_,_,int(0))).

d(var(X),X,int(1)) :- !.

d(var(_,_,int(0))).

d(plus(U,V),X,plus(U1,V1)) :- d(U,X,U1), d(V,X,V1).

d(minus(U,V),X,minus(U1,V1)) :- d(U,X,U1), d(V,X,V1).
Symbolic Differentiation

d(mult(U,V),X,plus(mult(U1,V),mult(U,V1))):-
   d(U,X,U1),
   d(V,X,V1).

d(div(U,V),X,div(minus(mult(U1,V),mult(U,V1)),mult(V,V))) :-
   d(U,X,U1),d(V,X,V1).

d(ln(U),X,mult(div(int(1),U),U1)) :- d(U,X,U1).

d(potens(E1,int(I)),X,mult(mult(int(I),potens(E1,int(I1))),EXP)) :-
   I1=I-1,
Symbolic Differentiation

d(E1,X,EXP).

d(sin(U),X,mult(cos(U),U1)) :- d(U,X,U1).

d(cos(U),X,minus(int(0),mult(sin(U),U1))) :- d(U,X,U1).

d(tan(U),X,mult(potens(sec(U),int(2)),U1)) :- d(U,X,U1).
**Insertion Sort**

\[
\text{isort}([ ],[ ]). \\
\text{isort}([X|UnSorted],AllSorted) :- \\
\quad \text{isort(UnSorted,Sorted),} \\
\quad \text{insert(X,Sorted,AllSorted).} \\
\]

\[
\text{insert}(X,[ ],[X]). \\
\text{insert}(X,[Y|L],[X,Y|L]) :- X \leq Y. \\
\text{insert}(X,[Y|L],[Y|IL]) :- X > Y, \quad \text{insert}(X,L,IL). 
\]
Tail Recursion

Recursive

reverse([ ],[ ]). reverse([XL],Rev) :- reverse(L,RL), append(RL,[X],Rev).

Tail Recursive (Iterative)

reverse([ ],[ ]). reverse(L,RL) :- reverse(L,[ ],RL). reverse([ ],RL,RL). reverse([XL],PRL,RL) :- reverse(L,[XPRL],RL).
Prolog Applications

- **Aviation, Airline and Airports**
  - Airport Capacity Planning (SCORE)
  - Aircraft Rotation Schedule Optimization (OPUS)
  - Resource Optimization for Ramp Handling (LIMBO II)
  - Baggage Sorter Planning (CHUTE)

- **Industry, Trade**
  - Shop Floor Scheduling (CAPS)
  - Shop Floor Scheduling (CIM.REFLEX)
  - Production, Stock & Transportation Planning (LOGIPLAN)

- **Health, Public**
  - Staff Scheduling (STAFFPLAN)
  - Hospital Resource Management & Booking (IDEAL)
  - Integrated Hospital Resource Management (REALISE)