CS348

Summer 2001

MP 2: Logic Programming

Due: Thursday, July 12

Name: Joseph S Kim

DESCRIPTION

The goal of this assignment is to write six small programs in PROLOG. Briefly, they are *Reverse*, *Substitute*, *Insert*, *Inorder*, *Path*, and *ShortestPath*. There are many commercial and free prolog programs but we used GNU prolog. And, I refer to several on-line manuals as a reference,
http://pauillac.inria.fr/~diaz/gnu-prolog/,
http://www-courses.cs.uiuc.edu/~cs348/prolog/doc/Html/,
http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html,

IMPLEMENTATION

1. Reverse
   The reverse(List1, List2) predicate is a built-in predicate in GNU Prolog.
   Re-implement reverse(), calling it reverse2(), without the use of any built-in predicates.

```
% reverse(List1,List2) :- L2 is the list obtained from L1 by
%                          reversing the order of the elements.

reverse(L1,L2) :- reverse2(L1,L2,[]).

reverse2([],L2,L2) :- !.
reverse2([X|Xs],L2,Int) :- reverse2(Xs,L2,[X|Int]).
```

2. Substitute
   Substitute(Atom1, Atom2, List1, List2); If called with Atom1 and Atom2 as constants, List1 as a constant list, and List2 as a variable, will bind List2 to a list with all instances of Atom1 in List1 replaced with List2.

```
% substitute(A1,A2,L1,L2) is true if the list L2 is the result of
% substituting A2 for all occurrences of A1 in the list L1.

substitute(A1,A2,[],[]).      %base case when L1 & L2 are empty.

substitute(A1,A2,[A1|L1],[A2|L2]):-substitute(A1,A2,L1,L2).
substitute(A1,A2,[Z|L1],[Z|L2]):- A1\=Z, substitute(A1,A2,L1,L2).
```

3. Insert (BST)

Implement the insert(Node, Tree1, Tree2) predicate for a Binary Search Tree (BST). Each node is defined by the predicate "node(Root, Left, Right)", or by the constant "empty". For example:

```
| ?- insert(a, empty, T).
T = node(a, empty, empty)
| ?- insert(3, node(2,empty,empty), T).
T = node(2,empty,node(3,empty,empty))
| ?- insert(1.5, node(3,node(1,empty,empty),node(5,empty,empty)), T).
T = node(3,node(1,empty,node(1.5,empty,empty)),node(5,empty,empty))
```

I separated five separate cases, as provided:

Case 1 – Tree1 is empty
Case 2 – N is smaller than the root, and left is empty
Case 3 – N is smaller than the root, but left is a tree
Case 4 – N is greater/equal the root, and right is empty
Case 5 – N is greater/equal the root, and right is a tree

```
% insert(X,Tree1,Tree2) is true if Tree2 is the result of inserting
% the element X into the ordered node Tree1.

insert(N, empty, node(N, empty, empty)). %base case: Tree1 is empty
insert(N, node(N,L,R), node(N,L,R)).

insert(N, node(Y,L,R), node(Y,L1,R)):-
  N < Y,                    % node(N) is smaller than the Root(Y)
  insert(N,L,L1).

insert(N, node(Y,L,R),node(Y,L,R1)):-
  N > Y,                    % node(N) is greater than the Root(Y)
  insert(N,R,R1).
```

4. Inorder

A set of rules to do an inorder traversal of the BST structure used in the previous problem. The predicate should be defined as inorder(Tree, List). But, I didn't use the *append()* predicate to keep track of the list.

```
% inorder(Tree,List) is true if List is an inorder traversal of BST

inorder(Tree, List) :- inorder(Tree, List, []).
inorder(node(X,L,R), Xs, Zs) :-
     inorder(R, Ys, Zs), inorder(L, Xs, [X|Ys]).
inorder(empty,Xs,Xs).              % base case that tree is empty
```

5. Path

Define a predicate, path(From, To, Path) which is true if there is a path from location From to location To via the locations in Path. The predicate should also work to return all possible paths. And also, I define several additional small predicates such as *Connected(), Travel(), Reverse()* , in addition to *path()*.

```
connected(X,Y) :- edge(X,Y) ; edge(Y,X).

path(A,B,Path) :-
       travel(A,B,[A],Q),
       reverse(Q,Path).

travel(A,B,P,[B|P]) :-
       connected(A,B).
travel(A,B,Visited,Path) :-
       connected(A,C),
       C \== B,
       \+member(C,Visited),
       travel(C,B,[C|Visited],Path).
```

6. Shortest Path[1]

Now define a predicate called shortestpath(From, To, Route) which is true if Route is the shortest path from From to To. Assume all edges are of length 1. The "best" way is to implement a breadth-first-search in Prolog, but that's a pain. Given that Prolog does a depth-first-search, a related search algorithm that finds optimal goals when the costs are of constant length is iterative deepening search algorithm.

```
shortestpath(Start,Goal,Route) :-
  shortestpath1([[Start]],Goal,R),reverse(R,Route).

shortestpath1([First| Rest],Goal,First) :- First = [Goal| _].

shortestpath1([[Last| Trail]| Others],Goal,Route) :-
  findall([Z,Last| Trail],
  legalnode(Last,Trail,Z),List),
  append(Others,List,Newroutes),
  shortestpath1(Newroutes,Goal,Route).

legalnode(X,Trail,Y) :- (a(X,Y);a(Y,X)),not(member(Y,Trail)).
```

[1] I had a hint from http://www.lh.com/~oleg/ftp/Prolog/shortest_path_weight1.prl .