CS 81, Logic and Computability
Problem Set 4: Prolog Games and Puzzles

In this assignment, you'll be using Prolog to solve a number of interesting problems. You're welcome to use the built-in predicates `member` and `append` and any other "helper" predicates that you wish, including those that you write yourself and those that we developed in class. **Do not change the names of the starter files, as the autograder relies on filenames!**

Note: Online Prolog interpreters such as SWISH (https://swish.swi-prolog.org/) are available. Some students prefer those environments to running `swipl` locally.

Challenge 1: More Trees! [15 Points]

Write a predicate called `insert(E, Tree, NewTree)` that is true if and only if the result of inserting element `E` into the binary search tree `Tree` results in the new binary search tree `NewTree`. Trees are represented in `[Root, Left, Right]` format as described above and insertion is the standard binary tree insertion that inserts a new element at a leaf in the location that preserves the binary search tree property. (Thankfully, we're reasoning here about standard binary search trees, not red-black trees, splay trees, etc.!) Here is an example (but test your code on other inputs too).

```
?- tree(T), insert(200, T, New).
T = [42, [5, [], []], [47, [], [50, [], []]]],
New = [42, [5, [], []], [47, [], [50, [], [...|...]]]]] [write]
HERE, PROLOG IS SHOWING ... PRESS THE w KEY TO SEE IT ALL!

T = [42, [5, [], []], [47, [], [50, [], []]]],
New = [42, [5, [], []], [47, [], [50, [], [200, [], []]]]]
```

Challenge 2: Graphs [10 Points]

Write predicate `path(Start, End, Graph, Path, Budget)` that is true if and only if `Path` is a list of vertices comprising a path from vertex `Start` to vertex `End` in the given weighted directed `Graph` such that the sum of the costs of the edges in the path does not exceed the given `Budget`. Let's break that down! First, here's an example of a directed weighted graph. Notice that the graph is represented as a list of triples where a triple `[x, y, w]` indicates that there is an edge from vertex `x` to vertex `y` with weight `w`. (*Note:* `member(X, Y)` is built-in to Prolog. It is true if and only if item X is a member of list Y.)

```
graph(  [ [a,b,1], [b,c,2], [c,b,1], [b,a,5], [c,d,2],
          [b,w,2], [w,x,3], [x,z,2], [b,z,5] ] ).
```

The weights will always be positive integers. The graph can have cycles as in this example (e.g., an edge from `a` to `b` and an edge from `b` to `a`). Moreover, the path from start to end need not be "simple"; that is, it may visit some vertices multiple times. Here are a few examples:

```
?- graph(G), path(a, z, G, P, 5).
false.  % There is no path from a to z of cost 5 or less

?- graph(G), path(a, z, G, P, 6).
G = [[a, b, 1], [b, c, 2], [c, b, 1], [b, a, 5], [c, d, 2], [b, w, 2], [w, x|...], [x|...], [...|...]],
P = [a, b, z] ;  % The path a, b, z costs 6, so it's within budget.  Any more?
false.

?- graph(G), path(a, z, G, P, 8).
G = [[a, b, 1], [b, c, 2], [c, b, 1], [b, a, 5], [c, d, 2], [b, w, 2], [w, x|...], [x|...], [...|...]],
P = [a, b, w, x, z] ;  % any more?
G = [[a, b, 1], [b, c, 2], [c, b, 1], [b, a, 5], [c, d, 2], [b, w, 2], [w, x|...], [x|...], [...|...]],
P = [a, b, z] ;  % any more?
false.

?- graph(G), path(a, z, G, P, 10).
G = [[a, b, 1], [b, c, 2], [c, b, 1], [b, a, 5], [c, d, 2], [b, w, 2], [w, x|...], [x|...], [...|...]],
P = [a, b, c, b, z] ; % any more?
G = [[a, b, 1], [b, c, 2], [c, b, 1], [b, a, 5], [c, d, 2], [b, w, 2], [w, x|...], [x|...], [...|...]],
P = [a, b, w, x, z] ; % any more?
G = [[a, b, 1], [b, c, 2], [c, b, 1], [b, a, 5], [c, d, 2], [b, w, 2], [w, x|...], [x|...], [...|...]],
P = [a, b, z] ; % any more
false.
```

Challenge 3: A Number Puzzle! [25 Points]

This problem is motivated by the famous "Four fours puzzle." An *arithmetic expression tree* is a binary tree that is either a single number or has arithmetic operations (+, *, -, or /) at its internal nodes and numbers at its leaves. The value of the tree is computed recursively by computing the value of the left subtree, the value of the right subtree, and then using the arithmetic operation at the root of that tree to combine the values of the left and right subtrees. For example, the value of the tree `[+, 3, 5]` is 8, the value of the tree `[-, 3, 5]` is -2 and the value of the tree `[*, [+, 4, 2], [-, 9, 2]]` is 42. We'll use the `/` symbol to represent division, but on your Prolog platform the operation may be `//` instead (there seems to be a difference between Mac and Windows according to some students, so check if `//` is integer division or normal division on your platform). So, the value of `[/, 7, 2]` may be 3 or 3.5 depending on your system. You'll want to use whatever corresponds to regular division (not integer division). My own solution (which works for the autograder) uses `//`.

Now, imagine that you are given a list of arithmetic operators which is some subset of `[+, *, -, /]`, a list of integers L, an integer "target" X, and an arithmetic expression tree. The question that we ask is: Does the arithmetic tree use only the operations in the given list, *each occurrence of a number in L appears exactly once* (in any order), and the tree evaluates to the target value X? Here are some examples:

```
?- solve([+, *], [1, 2, 3], 7, [+, 1, [*, 3, 2]]).
true

?- solve([+, -], [1, 2, 3], 7, [+, 1, [*, 3, 2]]).
false  <-- Note that the arithmetic tree used * which is not in [+, -]

?- solve([+, *], [4, 4, 4, 4], 24, X).  <-- Four fours, each one must be used once!
X = [+, 4, [+, 4, [*, 4, 4]]] ;
X = [+, 4, [+, [*, 4, 4], 4]] ;

?- solve([+, *, /], [5, 2, 1], 7, X).
X = [+, 5, [*, 2, 1]] ;
X = [+, 5, [/, 2, 1]] ;
X = [*, [+, 5, 2], 1] ;
X = [/, [+, 5, 2], 1] ;
...
X = [*, 1, [+, 2, 5]] ;
MANY MORE FOLLOW, BUT WE'VE CUT IT OFF HERE
```

Notice that each number must be used exactly once but the operations can be used any number of times. Write the `solve` predicate using the starter file `problem3.pl` that we've provided with the assignment instructions. Please add code to that file and submit it online.

Challenge 4: Fox, Hare, Lettuce, Man [25 Points]

The Fox-Hare-Lettuce Puzzle goes like this: A man, a fox, a hare, and lettuce are initially on the left bank of a river. The goal is to transport all four to the right bank of the river. Fortunately, there is a canoe available that can transport the man and up to one item from one bank to the other. (The man is permitted to cross the river by himself if he wishes, as long as he doesn't leave the fox and hare together nor does he leave the hare and lettuce together.) Only the man can operate the canoe, so the man will always need to be doing the transporting between the banks. (The man has been trying to teach the fox to paddle, but it hasn't been so successful yet.) Unfortunately, if the fox and hare are left alone, the hare will be eaten. If the hare and lettuce are left alone, the lettuce will be eaten. Your objective is to write a Prolog predicate called `solve` that encodes the starting configuration, ending configuration, and the rules of the puzzle (but no extra insights that you inferred on your own) and solves the puzzle. The solver is not required to find the shortest solution (the one that uses the fewest river crossing), just a valid solution.

Please use the provided starter file `problem4.pl` that came with the assignment instructions. A configuration of the puzzle will be represented as a list `[Left, Right]` where `Left` and `Right` are lists that represent who is present on the left and right side of the river. We'll keep those lists sorted in the order man before fox before hare before lettuce. So, for example, the initial configuration is `[[man, fox, hare, lettuce], []]` and the final configuration is `[[], [man, fox, hare, lettuce]]`. The configuration `[[man, hare], [fox, lettuce]]` is valid because both lists are sorted by man-before-fox-before-hare-before-lettuce and because man and hare can be together and so can fox and lettuce. But, the configuration `[[hare, fox], [man, lettuce]]` is not valid for two reasons: First, `[hare, fox]` is not in the right order and, second, those two can't be left on the same bank without the man.

We've defined some predicates in the starter file, including one called `initial` which defines the initial configuration. This will allow you to run your code this way:

```
?- initial(C), solve(C, X).
C = [[man, fox, hare, lettuce], []],
X = [man_takes_hare_right, man_goes_left, man_takes_fox_right,
man_takes_hare_left, man_takes_lettuce_right, man_goes_left,
man_takes_hare_right] ;
MORE SOLUTIONS POSSIBLE
```

The names of some of the moves are in the solution list above and all the moves are enumerated in the `problem4.pl` starter file.