Homework 6: Logic Programming

Due Thursday, June 8 at 11:30pm

Turn in your homework as a text file called `hw6.pl` via the course home page. Please make sure the file has exactly that name.

Make sure the file can be successfully loaded into the `gprolog`. There should be *no compilation errors*; if not you get an automatic 0 for the homework!

Recall the CS131 Academic Honesty Policy! You must list whom you discussed the assignment with at the top of your assignment, and also what other resources you used.

For this assignment you will get some experience programming in a (almost) fully declarative language.

Some rules and advice:

- You may define any number of helper predicates as needed to solve the following problems.
- You may use predicates from the GNU Prolog library (see the GNU Prolog Manual) except where explicitly disallowed.
- Your code should never go into an infinite loop except where explicitly allowed.
- Your code should never use the cut (`!`) operator.
- Except where explicitly indicated, your code need not produce solutions in a particular order, as long as it produces all and only correct solutions and produces each solution the right number of times.
- Some of these problems are computationally hard (e.g., NP-complete). For such problems especially, the order in which you put subgoals in a rule can make a *big* difference on running time. In general the best strategy is to put the *strongest* constraints earliest, i.e., the constraints that will prune the search space the most.

To test your code, make a text file that has a bunch of queries that you want to input at the interpreter. You need to use a predicate like `findall` so that the interpreter will collect all solutions rather than providing them one at a time and asking if you want more. For example, the file could have the following form:

```
consult(hw6).
findall(Y, duplist([1,2,3], Y), L), L = [[1,1,2,2,3,3]].
... a bunch of other queries to use as tests
```

If this file is called hw6Test, then you can do "gprolog < hw6Test" to run all the tests.

Now on to the assignment! **Make sure your predicates have the exact same names and order of arguments as described below. Otherwise our tests will fail and you will get no credit.**

1. Implement a predicate `duplist(X,Y)` which succeeds whenever list `Y` is simply list `X` with each element duplicated. For example:

   ```
   | ?- duplist([1,2,3], Y).

   Y = [1,1,2,2,3,3]
   ```

   Make sure that your predicate works in both directions. For example, you should be able to ask the query `duplist(X, [1,1,2,2,3,3])` and get the right value of `X`.

2. Define a predicate `subseq(X,Y)` that succeeds if list `X` is a *subsequence* of list `Y`, which means that `X` can be obtained by removing zero or more elements from anywhere within `Y`. Note that this means that the elements in `X` have to be in the same order as they are in `Y`. For example, `[1,3]` is a subsequence of `[1,2,3]`, but `[3,1]` is not. **You may not use the built-in predicate `sublist` in your solution.**

   In addition to checking whether one concrete list is a subsequence of another concrete list, your solution should be able to produce all subsequences of a given concrete list (but is allowed to produce those solutions in any order). For example:

   ```
   | ?- subseq(X, [1,2]).

   X = [1,2] ? ;

   X = [1] ? ;

   X = [2] ? ;

   X = [] ? ;
   ```

3. In this problem, you will write a Prolog program to solve a form of the *blocks world* problem, which is a famous problem from artificial intelligence. Imagine you have three stacks of blocks in a particular configuration, and the goal is to move blocks around so that the three stacks end up in some other configuration. You have a robot that knows how to do two kinds of actions. First, the robot can pick up the top block from a stack, as long as that stack is nonempty and the robot's mechanical hand is free. Second, the robot can place a block from its hand on to a stack.

   Implement a Prolog relation `blocksworld` that has three arguments: a start world, a list of actions, and a goal world. The predicate should only succeed if the given list of actions does in fact take you from the start to the goal.

   We will represent a world as a data structure `world` that has four components: three lists to represent the three stacks, and a component to represent the contents of the mechanical hand. That last component either contains a single block or the constant `none`, to represent the fact that the hand is empty. Your implementation should not care what the blocks actually look like; the users can provide any constants they wish to stand for blocks. For example, `world([a,b,c],[],[d],none)` represents a world in which there are four blocks, with three of them in the first stack (in the specified order, with `a` at the top of the stack), the fourth block on the third stack, and nothing in the mechanical hand.

   We will use the constants `stack1`, `stack2`, and `stack3` to represent the three stacks. There are two kinds of actions, `pickup(B,S)` and `putdown(B,S)`, corresponding to the two robot actions described above, where `B` is a block and `S` is one of the three stack constants.

   For example, here is a sample query:

```
| ?- length(Actions,L), blocksworld(world([a,b,c],[],[],none), Actions, world([],[],[a,b,c],none)).

Actions = [pickup(a,stack1),putdown(a,stack2),pickup(b,stack1),putdown(b,stack2),pickup(c,stack1),putdown(c,stack3),pickup(b,stack2),putd
L = 10 ?
```

Notice how I use `length` to limit the size of the resulting list of actions. This is necessary to do when you test your code, in order to prevent Prolog from getting stuck down infinite paths (e.g., continually picking up and putting down the same block).

4. In this problem you will define a predicate to solve verbal arithmetic puzzles. In the special case we'll consider, you're given three words and have to find a digit for each letter such that word1 + word2 = word3. Each letter must have a distinct digit between 0 and 9, and the first letter of each word cannot have the value 0. Your predicate `verbalarithmetic` will take four argument lists (in this order): a list of all the letters in the problem (in any order), followed by three lists representing the three words in the puzzle. For example:

```
| ?- verbalarithmetic([S,E,N,D,M,O,R,Y],[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]).

D = 7
E = 5
M = 1
N = 6
O = 0
R = 8
S = 9
Y = 2 ? ;

| ?- verbalarithmetic([C,O,A,L,S,I],[C,O,C,A],[C,O,L,A],[O,A,S,I,S]).

A = 6
C = 8
I = 9
L = 0
O = 1
S = 2 ? ;
```

Hint: You may find gprolog's `fd_all_different` predicate useful.