Homework 7: The Future of Programming?

Due Thursday, March 13 at 11:30pm

Turn in your homework as a text file called `hw7.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.
- 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` or `bagof` so that the interpreter will collect all solutions rather than providing them one at a time and asking if you want more. You can then use `permutation` to see if you got the expected answer, allowing for solutions to be produced in any order. For example, the file could have the following form:

```
consult(hw7).
findall(K, get(K,[[2,hello],[1,hello]],hello), L), permutation(L, [1,2]).
... a bunch of other queries to use as tests
```

If this file is called hw7Test, then you can do "gprolog < hw7Test" 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 the tests will fail and you will get no credit.**

1. Let's implement everyone's favorite data structure...wait for it...*dictionaries*! We'll represent a dictionary as a list of pairs, where each pair is represented as a two-element list. Implement a predicate `put(K,V,D1,D2)` that succeeds if `D2` is the dictionary that results from mapping key `K` to value `V` in dictionary `D1`. If the key `K` is already mapped in `D1`, then `put` should (conceptually) replace it with the new key. In other words, `D2` should only ever have at most one entry for a given key (assuming `D1` satisfies this property). Also implement a predicate `get(K,D,V)` that succeeds if `K` is mapped to `V` in `D`. This being Prolog, these predicates are much cooler than the versions we implemented in other languages, because they can be used to answer a wide variety of queries. For example, `get` can not only get the value associated with a key but also get all keys associated with a particular value and also iterate over all key-value pairs in the dictionary. Some examples:

   ```
   | ?- put(1,hello,[[2,two]],D).

   D = [[2,two],[1,hello]] ?

   | ?- put(1,hello,[[2,two],[1,one]],D).

   D = [[2,two],[1,hello]] ?

   | ?- put(1,hello,D,[[2,two],[1,hello]]).

   D = [[2,two]] ? ;

   D = [[2,two],[1,_]] ? ;

   | ?- get(1,[[2,two],[1,hello]],V).

   V = hello ?
   ```

```
| ?- get(K,[[2,hello],[1,hello]],hello).

K = 2 ? ;

K = 1 ? ;

| ?- get(K,[[2,two],[1,hello]],V).

K = 2
V = two ? ;

K = 1
V = hello ? ;
```

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. Consider the 4x4 version of the game Sudoku. In this version, you are given a 4x4 grid with numbers between 1 and 4 filled in some squares. The goal is to fill in the remaining squares so that each row contains the numbers 1-4 in some order, each column contains the numbers 1-4 in some order, and each 2x2 quadrant contains the numbers 1-4 in some order.

   Define a predicate `sudoku(Initial, Final)` that succeeds if `Final` is the solution to the Sudoku puzzle defined by `Initial`. We will represent a 4x4 grid as a list of four lists, each inner list representing one row of the grid. You may assume that the given `Initial` list is a list of exactly four lists and that each inner list has exactly four elements.

   Here's an example of what your predicate should be able to do:

```
| ?- sudoku([[2,1,_,_],
             [4,_,_,_],
             [_,_,_,4],
             [_,_,1,_]], Solution).

Solution = [[2,1,4,3],[4,3,2,1],[1,2,3,4],[3,4,1,2]] ? ;
```

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

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.

5. The Tower of Hanoi is another classic mathematical puzzle. It consists of three pegs and a number of disks of different sizes. The puzzle starts with the disks stacked on peg 1, in order of size with the largest disk at the bottom of the stack. The goal is to relocate the entire stack to peg 2, using a sequence of *moves*. Each move can simply take the top disk from one peg (if that peg is nonempty) and move it to the top of another peg. However, a disk can never be placed on top of a smaller disk.

   Define a predicate `towerOfHanoi(Init, Goal, Moves)` that succeeds if we can get from the state of the world represented by `Init` to the state of the world `Goal` by executing the moves represented by `Moves` in order. Of course, those moves must all satisfy the rules defined above for a legal move. We will represent a state of the world as a list containing three lists, each inner list representing the contents of one of the pegs from top to bottom. We will represent disks as integers, with smaller integers representing smaller disks. Your code should work for any number of disks. The six possible moves are represented by the terms `to(peg1,peg2)`, `to(peg1,peg3)`, `to(peg2,peg1)`, `to(peg2,peg3)`, `to(peg3,peg1)`, and `to(peg3,peg2)`, which respectively denote moving a disk from peg 1 to peg 2, from peg 1 to peg 3, etc. Note that `peg1`, `peg2`, and `peg3` are constants representing the three pegs.

   Here's an example of what your predicate should be able to do:

```
| ?- length(Moves, L), L < 8, towerOfHanoi([[1,2,3],[],[]], [[],[1,2,3],[]], Moves).

L = 7
Moves = [to(peg1,peg2),to(peg1,peg3),to(peg2,peg3),to(peg1,peg2),to(peg3,peg1),to(peg3,peg2),to(peg1,peg2)] ? ;
```

   You can assume that the intial and goal states will always be fully specified as part of a query and that they represent legal states of the puzzle.

   Notice how I use `length` to limit the size of the resulting list of moves. This is necessary to do when you test your code, in order to prevent Prolog from getting stuck down infinite paths (e.g., continually transferring the same disk back and forth between two pegs). Prolog will still go into an infinite loop (and overflow the stack) eventually if you keep asking for more solutions, but it should not do that until after it has produced all valid solutions.