CSE 341, Winter 2015, Assignment 6
Prolog Project
Due: Fri Feb 27, 10:00pm

55 points total (5 points each for Questions 1, 2, and 5; 10 points each for Questions 3, 4, 6, and 7).

You can use up to 3 late days for this assignment.

Rather than a single large project, for this assignment there are a set of smaller problems that bring out different aspects of the language. The first two problems are basically warmup problems. (Hint: do these soon, even if you don't start on the others.) The next two (family tree and finding paths through a maze) illustrate search and backtracking to find solutions. The last three problems illustrate the use of two of SWI Prolog's constraint libraries. There is a starter program, linked from the assignments page, that includes some starter code for the family tree and maze questions, and another file with starter code for some of the unit tests.

1. Prolog sentences: write a Prolog rule `sentence` that succeeds if the first argument is the definite article (in other words, the atom "the"), the second article is a noun, and the third article is a verb. Define at least four facts about nouns (so that Prolog has at least four possible nouns to use in sentences), and also at least four facts about verbs. For example, your facts might include:

   ```
   noun(octopus).
   verb(swims).
   ```

   Then the goal `sentence(the, octopus, swims).` should succeed.

   There are a few unit tests for these rules in the starter file — modify these as needed if you use other facts. You should have at least 4 unit tests, including one for a goal that fails.

   In addition, try your goal with variables for all three arguments. Backtrack if there are more answers available. You don't need to turn in all the output for this. *However, in a comment in your code, say how many different answers you found, and explain why Prolog found that number of answers.*

2. Write a Prolog rule `average` that computes the average of a list of numbers. Fail if the list is empty. (You don't need to worry about lists of things that aren't numbers.) For this question, use `is` for arithmetic — as a result this will only work with lists that are ground (in other words, no variables in the middle of the list. You can use the built-in `length` predicate if you wish. There are suitable unit tests already in the starter file.

3. Prolog family tree: Write Prolog rules `grandmother`, `son`, and `ancestor`.

   Facts may include

   ```
   man('Haakon VII').
   man('Olav V').
   man('Harald V').
   man('Haakon').
   woman('Martha').
   woman('Mette-Marit').
   woman('Maud').
   woman('Sonja').
   parent('Haakon VII','Olav V').
   parent('Maud','Olav V').
   parent('Olav V','Harald V').
   ```

```
parent('Martha','Harald V').
parent('Harald V','Haakon').
parent('Sonja','Haakon').
```

(If you want, you can use facts other than about the Norwegian royal family — if you do, include at least 4 facts for men, 4 facts for women, and 6 facts for parent.)

For `parent(X,Y)` facts, the definition means X is a parent of Y.

Include a few unit tests to test your rules.

Given the facts above, a few example goals that should succeed are

`grandmother('Martha', 'Haakon').`

`son('Haakon', 'Sonja').`

`ancestor('Haakon VII', 'Haakon').`

`ancestor('Sonja','Haakon').`

4. Write Prolog rules to find paths through a maze. The maze has various destinations (which for some strange reason are named after well-known spots on the UW campus), and directed edges between them. Each edge has a cost. Here is a representation of the available edges:

   ```
   edge(allen_basement, atrium, 5).
   edge(atrium, hub, 10).
   edge(hub, odegaard, 140).
   edge(hub, red_square, 130).
   edge(red_square, odegaard, 20).
   edge(red_square, ave, 50).
   edge(odegaard, ave, 45).
   edge(allen_basement, ave, 20).
   ```

   The first fact means, for example, that there is a edge from the Allen Center basement to the Atrium, which costs $5 (expensive maze). These edges only go one way (to make this a directed acyclic graph) — you can't get back from the Atrium to the basement. There is also a mysterious shortcut tunnel from the basement to the Ave, represented by the last fact.

   You can use these facts directly as part of your program – to avoid the need for copying and pasting, they are in the starter file `hw6.pl`. You should then write rules that define the `path` relation:

   ```
   path(Start, Finish, Stops, Cost) :- ....
   ```

   This succeeds if there is a sequences of edges from `Start` to `Finish`, through the points in the list `Stops` (including the start and the finish), with a total cost of `Cost`. For example, the goal `path(allen_basement, hub, S, C)` should succeed, with `S=[allen_basement, atrium, hub], C=15`. The goal `path(red_square, hub, S, C)` should fail, since there isn't any path from Red Square back to the HUB in this maze.

   The goal `path(allen_basement, ave, S, C)` should succeed in four different ways, with costs of 20, 200, 195, and 210 and corresponding lists of stops. (It doesn't matter what order you generate these in.)

   The starter file includes 3 unit tests, which show different ways of testing one of the goals. Add at least 3 other tests for other goals, including one that fails.

   Hints: try solving this in a series of steps. First, solve a simplified version, in which you omit the list of stops and the cost from the goal. Then modify your solution to include the cost, then after that's

working add the stops. Note that there are no edges from a stop to itself, i.e. there is no implicit rule `edge(allen_basement,allen_basement, 0).` Finally, add other unit tests as needed.

When you add the code to find the stops, you might find your solution almost works, except that your path comes out in reverse order. There are various ways to solve this (including reversing the list). The more elegant approach, however, is to construct the list from the end. (Doing this will likely only require minor changes to your code.) For example, suppose that you are searching for a path from `allen_basement` to `red_square`. Your rule might find an edge from `allen_basement` to `atrium`, and a recursive call to your rule might find a path from `atrium` to `red_square`. Then the path from `allen_basement` to `red_square` can be formed by taking the path from `atrium` to `red_square` (namely `[atrium,hub,red_square]`) and putting the new node on the front of this list to yield the path from `allen_basement` (namely `[allen_basement,atrium,hub,red_square]`).

5. Write an improved version of the Prolog rule `average`, from Question 2, called `better_average`, that uses the clpr library. Your new rule should now be more general, and should work for these goals:

```
better_average([1,2,3],N).
better_average([1,X,3],3).
better_average(Xs,10).
better_average(Xs,A).
```

6. A cryptarithmetic puzzle consists of an equation involving several numbers whose digits are represented by letters. Each digit can map to at most one letter. A solution consists of identifying the digit corresponding to each letter. The classic `SEND+MORE=MONEY` puzzle, by Henry Dudeney, was published in 1924 in *Strand Magazine*. There is a sample Prolog program on the 341 website to solve this puzzle.

```
  S E N D
+ M O R E
---------
M O N E Y
```

For this question, write a Prolog program, using the `clpfd` library, to solve the following cryptarithmetic *multiplication* problem:

```
      B O B
x     B O B
-----------
M A R L E Y
```

(The lower-case `x` denotes multiplication - it is not a digit in the problem.) The leading digits (`M` and `B`) cannot be 0. There are two solutions: find them both.

7. A Wheatstone bridge is an electrical circuit typically used to measure the resistance of a resistor by balancing two legs of a bridge circuit, one leg of which includes the resistor whose resistance isn't known. Figure 1 (from Wikipedia) shows the circuit diagram.

Write a Prolog rule `wheatstone(Vb, R1,R2,R3,Rx, Ig)` that models a Wheatstone bridge. Make use of the clpr library. `Vb` should be the voltage of the battery in volts; `R1`, `R2`, `R3`, and `Rx` should be the resistances of the four resistors in ohms, and `Ig` should be the current through the galvanometer in amperes. Model the galvanometer as a short circuit (so that the voltages at D and B in the diagram are the same). You'll need to use Ohm's Law and Kirchhoff's current law (that the sum of the currents at a node in the circuit is 0).

Using your rule, find the current when `Vb=10`, `R1=100`, `R2=50`, `R3=60`, and `Rx=30`. It should be 0 in this case, since the ratio of `R1` to `R2` is the same as the ratio of `R3` to `Rx`. Next find the current when `Rx=10` (other values the same) – in this case the current should be approximately 0.04546 amperes.

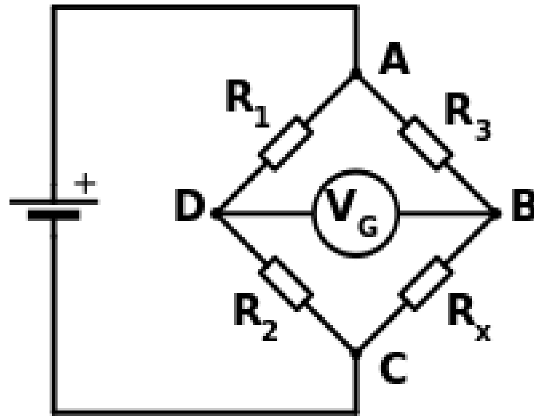

Figure 1: Wheatstone Bridge Circuit (from Wikipedia)

Finally, suppose that the only resistors available are 10, 20, 50, and 100 ohms. You can represent this using four facts, just as with the edges in the maze for Question 4. Find all the sets of those resistors used in the Wheatstone bridge that result in a current of more than 0.4 amperes. (You can use a value more than once, i.e. you've got lots of 10 ohm, 20 ohm, etc. resistors.) You should get three different answers for this. For your unit test, it's OK to just test that you get (nondeterministically) one answer that uses the available resistors and has a current greater than 0.4 amperes.

Turnin: Turn in two files: `hw6.pl`, which should have your rules, and `hw6_tests.pl` with your unit tests. Include appropriate unit tests for Questions 1, 2, 3, 4, 5, and 7. (The starter file already has a few tests to get you started.)

For Question 6 include a goal to find the answers, and include the answers in a comment. As usual, your program should be tastefully commented (i.e. put in a comment before each set of rules saying what they do).