CSE 341, Autumn 2012, Assignment 6
Prolog Project
Due: Wed Nov 14, 10:00pm
Version of Nov 7 (corrected typo)

45 points total (10 points each Questions 1, 3, and 4; 15 points Question 2); max 5 points for the extra credit problem.

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

Rather than a single large project, for this assignment there are four different smaller problems that bring out different aspects of the language. The first problem (finding paths through a maze) illustrates search and backtracking to find solutions. The second problem (derivatives) revisits the Racket program for this, illustrating similarities and differences between the languages and demonstrating the use of unification for various kinds of pattern matching. The last two problems illustrate the use of one of SWI Prolog's constraint libraries. There are starter programs for the first three questions, linked from the course website.

1. 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, there is a starter file `maze.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]`).

2. Write a Prolog version of the "deriv" program you wrote in Racket to do symbolic differentiation. The starter file `deriv.pl` contains rules for finding the derivative of constants, variables, the sum of two expressions, and the product of two expressions, for simplification, and unit tests. Add rules for minus, sin and cos, and raising an expression to an integer power. The formulae for these, and the simplification rules, should be the same as in the Racket assignment. Add suitable unit tests for your new rules.

3. The program `grid.pl` uses the clpr library in SWI Prolog. It models a metal plate as a 2d grid of variables representing the temperatures at different points on the grid. The temperature at each interior point is constrained to be the average of its four neighbors (above, below, to the left, and to the right). Copy the program to your own directory.

   Try the goal `grid1` to make sure it's working. Now add a rule `grid2` that models a 9×9 plate and prints out the temperatures at each point. We know the following information about the temperatures:

   - the temperature at the center is 50℃
   - the temperature at the point midway between the top and the center is 40℃
   - the temperature at the point midway between the center and the bottom is 60℃
   - the temperature at the point midway between the center and the right side is 65℃
   - the temperatures at every exterior point on the left side is the same (including the corners)
   - the temperatures at every exterior point on the right side is the same (including the corners)
   - the temperatures at every exterior point on the top side is the same (excluding the corners)
   - the temperatures at every exterior point on the bottom side is the same (excluding the corners)

   You don't need to include unit tests for this question, just the source file.

4. 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).

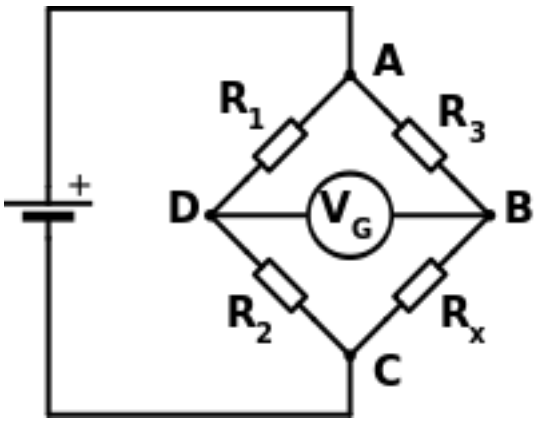

Figure 1: Wheatstone Bridge Circuit (from Wikipedia)

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.

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 1. 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 four different files, one for each question. Include appropriate unit tests for Questions 1, 2, and 4. As usual, your program should be tastefully commented (i.e. put in a comment before each set of rules saying what they do).

Extra Credit. (5 points max) SWI Prolog includes other constraint libraries in addition to clpr, including clpfd (constraints over finite domains). Learn about finite domain constraints and how to use them, identify or devise an interesting problem for which they work well, and write a Prolog + clpfd program that solves that problem. (There is lots of material on the internet about finite domain constraints and solvers. It's fine to use this material for inspiration or as a starting point for your own problem and solution. If you do use existing code as a starting point, say where you found it, what you reused, and what you changed.)