CSE 341, Autumn 2015, Assignment 6
Prolog Project
Due: Wed Nov 18, 10:00pm

50 points total (5 points each for Questions 1, 2, 6, and 7); 10 points each for Questions 3, 4, and 5); plus up to 2 points extra credit.

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 additional warmup problems. (Hint: do these soon, even if you don't start on the others right away.) The next two (family tree and finding paths through a maze) illustrate search and backtracking to find solutions. The derivatives problem 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 two of SWI Prolog's constraint libraries. Finally there is an extra credit problem that involves difference lists. 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. 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.

2. Write Prolog rules to manipulate a queue, including rules for `enqueue` (put an element at the end of the queue), `dequeue` (remove the first element), `head` (find the element at the head of the queue), and `empty` (succeeds if the expression is an empty queue).

   To express this in Prolog, your rules should be of the following form. (Some of these may just be facts, i.e. there won't be a `...` part. Also, you can use expressions in place of the variables like `Q1` to simplify your program.)

   ```
   /* if Q1 is a queue, Q2 is a new queue that has the same elements as
      Q1 plus X at the end of the queue */
   enqueue(Q1,X,Q2) :- .....

   /* if Q1 is a queue, X will be the first element in the queue,
   and Q2 is a new queue that results from removing X */
   dequeue(Q1,X,Q2) :- .....

   /* if Q1 is a queue, X will be the first element in the queue.  Fail
   if Q1 is empty. */
   head(Q1,X) :- .....

   /* succeed if Q1 is the empty queue; otherwise fail */
   empty(Q1) :- .....
   ```

   Hints: represent a queue as an ordinary list. You can use the built-in `append` rule. This is a simple program — the sample solution was 5 lines, plus comments and a helper rule. Here is an example of using these rules:

```
?- empty(Q1), enqueue(Q1,squid,Q2), enqueue(Q2,clam,Q3), dequeue(Q3,X,Q4).
```

This starts `Q1` as an empty queue, adds `squid` to `Q1` to yield `Q2`, adds `clam` to `Q2` to yield `Q3`, and dequeues the first element of `Q3` to yield `Q4`. This goal should succeed with

```
Q1 = []
Q2 = [squid]
Q3 = [squid, clam]
Q4 = [clam]
X = squid
```

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).` This avoids cycles. 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 a Prolog version of the "deriv" program you wrote in Racket to do symbolic differentiation. The starter file `deriv.pl`, linked from the Prolog pages on the 341 website, 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.

6. Write an improved version of the Prolog rule `average`, from Question 1, 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).
```

7. 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.

Extra Credit. (2 points) The rules for queues in Question 2 are simple but probably inefficient, in that your rule for putting something at the end of a queue probably copies the old queue for each new element. Write a different version of the queue rules that uses difference lists, so that all operations on queues are constant time operations.

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 1–6. (The starter file already has a few tests to get you started, and there are also some unit tests with the "deriv" program.)

For Question 7 include a goal to find the answers, and include the answers in a comment. Check the answers with a calculator if you wish.

As usual, your program should be tastefully commented (i.e. put in a comment before each set of rules saying what they do).