CSE 341, Winter 2010, Assignment 6
CLP(R) Part 2
Due: Monday Feb 22, 10:00pm

12 points total (3 points each Questions 1 and 3; 6 points Question 2)

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

1. Write CLP(R) 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 CLP(R), 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. 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
   ```

2. Write CLP(R) 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 – copy them and paste them into the start of your solution. You should also 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.)

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)`.

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]`).

3. The program `grid.clpr` (available on attu at `~borning/clpr/grid.clpr`) 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. Now import the grid program into your own homework solution by including this goal in your own program:

   ```
   :- consult(grid).
   ```

   Try the goals `grid1` and `grid2` to make sure it's working. (You don't need to include this part in the script you hand in.)

   Now, using `grid.clpr`, model a 9×9 plate and print 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 left side and the center is 30℃
   - the temperature at the point midway between the center and the right side is 70℃
   - 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)

   Here is the expected output:

   ```
     4.53  27.26  27.26  27.26  27.26  27.26  27.26  27.26  95.47
     4.53  18.27  25.16  29.82  33.98  38.84  46.23  60.58  95.47
     4.53  16.13  25.30  32.88  40.00  47.90  58.23  73.34  95.47
     4.53  16.41  27.03  36.40  45.23  54.53  65.46  79.08  95.47
     4.53  17.96  30.00  40.47  50.00  59.53  70.00  82.04  95.47
     4.53  20.92  34.54  45.47  54.77  63.60  72.97  83.59  95.47
     4.53  26.66  41.77  52.10  60.00  67.12  74.70  83.87  95.47
     4.53  39.42  53.77  61.16  66.02  70.18  74.84  81.73  95.47
     4.53  72.74  72.74  72.74  72.74  72.74  72.74  72.74  95.47
   ```

Turnin: Turn in your CLP(R) program and sample output showing it running on some well-chosen test cases. 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. (1 point) The rules for queues in Question 1 are simple but probably inefficient, in that your rule for putting something at the end of a queue ends up copying 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.