Summer 2016
Programming Languages
Homework 4

- This homework is a combination programming and "paper & pencil" assignment.
- Due via NYU Classes on Tuesday, August 2 at 5:00 PM Eastern Time. Due to timing considerations, late submissions will not be accepted.
- For the Prolog questions, you should use SWI Prolog. A link is available on the course page.
- You may collaborate and use any reference materials necessary to the extent required to understand the material. All solutions must be your own, except that you may rely upon and use Prolog code from the lecture if you wish. Homework submissions containing any answers or code copied from any other source, in part or in whole, will receive a zero score.
- Please submit your homework as a zip file, `hw4-<netid>.zip`. For questions 1, 3 and 4, you will submit a PDF containing the written solutions. For question 2, you will submit separate files for rules and queries. Therefore the `.zip` file should contain three files: `hw4-<netid>-rules.pl` and `hw4-<netid>-queries.pl` containing solutions to question 2, and `hw4-<netid>.pdf` containing solutions to questions 1, 3 and 4. Please make sure your code for question 2 compiles before submitting. *If your code does not compile for any reason it may not be graded.*

1. [30 points] **Garbage Collection**

   Consider the following pseudo-code:

```
public class Program
{
  // entry point
  public static void Main ()
  {
    Car car = new Car(Color.Red, 200);

    Initialize(car);

    EventLog log = new EventLog("output.log");

    BeginEventLoop(log);
  }

   // initialize a car
 protected static void Initialize(Car c)
 {
    Carwash w = new Carwash();
    w.Wash(c);

    Garage g = new Garage();
    g.Store(c);
  }

  public static void BeginEventLoop() { ... }
}

public class Car  // occupies 24 bytes
{
    private List<Wheel> wheels = new List<Wheel>();


    Car(Color col, int horsePower) // constructor
      {
        for (int x=0; x < 4; x++)
          { wheels.Add(new Wheel()); }
      }
}

public class Carwash { ... }  // 32 bytes
public class Garage  { ... }  // 32 bytes
public class Wheel { ... } // 12 bytes
public class EventLog { ... } // 8 bytes
```

   You may assume that the program above is single-threaded, follows static scoping rules, and that sub-program `Main` is the entry point. Additionally assume that every heap object will occupy an 8 byte forwarding address *in addition to* the sizes noted above in the comments. Please do the following:

   1. Assuming the "heap pointer" allocation method, draw the contents of the heap upon entry into BeginEventLoop, but before any garbage collection has taken place. Identify and show the address

of each object in the heap. Also, show the address of the heap pointer at that moment. (Assume heap memory begins at address 0 and that the heap pointer is incremented by the size of the allocated object during each allocation.)

2. Assuming the copy garbage collection algorithm discussed in class executes during the call to `BeginEventLoop`, show the state of heap memory after the first collection has concluded. As part of your solution, identify the root pointers and specify the order in which they were processed by the copy collection algorithm to arrive at the heap state you identified. As before, identify and show the address of each object in the heap and show the address of the heap pointer. It is only necessary to draw the "TO" space, since the "FROM" space will appear according to your solution above. You may assume that `BeginEventLoop` causes no change to the heap.

2. [40 points] **Prolog**

Consider the micro-blogging site, Twitter—first conceived at NYU by an undergraduate student and later launched in around 2006. It was originally written in Ruby on Rails, then later ported for reasons of scalability to the cross-paradigm language, Scala. Although Twitter was never implemented in Prolog, we shall see that implementing the majority of Twitter's decision engine in Prolog is surprisingly simple. First, some background.

Users may broadcast short messages called *tweets*. Twitter restricts tweets to 140 characters or less, but we shall ignore this restriction here and instead consider tweets of any arbitrary length. Tweets are broadcast publicly, meaning they can be seen by everybody[1]. A tweet which is directed to a particular user, by convention, begins with the recipient's Twitter user name at the beginning. It's important to remember that despite a tweet being possibly directed to a particular user, the tweet is still viewable by everybody.

If user x is generally interested in what user y has to say, user x may *follow* user y, causing all of y's tweets to appear in x's Twitter *feed*. The feed of user x is a running list of tweets generated by anybody that x follows.

If a user x finds one particular tweet of another user y interesting, she may *retweet* it to their followers. This has the effect of the tweet being visible to the followers of x (in addition to y, as before).

Another way tweets can gain visibility is through searching. Users can search the entire universe of tweets for certain keywords (including, but not limited to, hash tags). This is typically how users develop a following by other users.

Now let's get started:

1. Create a number of Twitter users by stating several `user(U)` facts, where U is the user's name. Note that all Twitter user names begin with the character @. (e.g., @tony)
2. The relation `follows(X,Y)` establishes that user "X *follows* user Y." Note that this relation should not be symmetric (i.e., user x following y does not imply that y follows x). Relate the above users by creating several `follows` facts.
3. The relation `tweet(U,I,M)` represents a tweet broadcast by user U with unique identifier I and message M. The identifier is needed because the same user could tweet the same message twice, but this is considered 2 distinct tweets. We can represent the message M as an array of atoms. We will not concern ourselves with the 140 character limit, nor the length of the array, for this assignment. Create several tweet facts.
4. Create a Prolog relation `retweet(U,I)` which represents user U retweeting the tweet identified by I.
5. Create a Prolog relation `feedhelper(U,F,M,I)` which establishes user U's Twitter feed. F is any user who U follows, M is any message tweeted or retweeted by F and I is a unique tweet identifier. Note that if one queries `feedhelper` by instantiating U to a concrete value, Prolog may output identical instantiations (i.e., bind M, I to the same values more than once), effectively causing duplicate tweets to appear in the feed. This could happen for several reasons, such as if two users $f_1, f_2 \in F$ both retweet the same tweet. We can remedy this situation by defining the Prolog relation `feed(U,M)` as follows:

```
feed(U,M) :- uniquefeed(U,O),remove_ident(O,M).

uniquefeed(U,R) :- setof([I,F|M],feedhelper(U,F,M,I),R).

remove_ident([],[]).
remove_ident([[_|Y]|T1],[H2|T2]) :- Y=H2,remove_ident(T1,T2).
```

[1] A user's Twitter account can also be specified as "private," thereby designating all tweets by that user as non-public. We will not consider this case here.

6. Create a Prolog relation `search(K,U,M)` which searches the universe of tweets for the keyword K. Variables U and M, when uninstantiated, will be bound to each tweet sent by user U whose message is M. For this homework, we will limit searches to single atoms so that finding a particular keyword (i.e., atom) within a tweet amounts to searching the tweet's array of atoms for a match.
7. A tweet identified by j is considered *viral* (as far as this homework is concerned) if there exists some $n \geq 2$ and there also exists a sequence of users $u_1, \ldots, u_n$ such that:
   - u_1 follows u_2.
   - for all $1 < i < n : u_i$ follows u_{i+1} *and* u_i retweets j.
   - u_n tweets j.

   Create a Prolog relation `isviral(S,I,R)`, where I is the unique identifier of a tweet, $S = u_n$ and $R = u_1$. Note that if `isviral` holds for some n, then by definition it also holds for all i such that $1 < i < n$.
8. Create a Prolog relation `isviral(S,I,R,M)`, where I is viral and at least M transitive `follows` relations (beginning from R and terminating at S) exist. When the above relation holds, we say that a tweet is viral with at least M *levels of indirection.*

Also complete the following:

1. Write a query that shows who is following a specified user.
2. Write a query that shows all tweets posted by a specified user.
3. Write a query that shows how many users retweeted a specified tweet.
4. Write a query that shows a particular user's feed. Ensure that tweets of any users that the user follows are visible in the feed.
5. Write a query that searches for a keyword in the universe of tweets.
6. Write a query that shows if a particular tweet is viral between the sender and a specified receiver.
7. Write a query that shows if a particular tweet is viral between the sender and a specified receiver in no less than 3 levels of indirection.

3. [15 points] **Prolog Performance** Consider the following:

```
foo(ashwin).
foo(roberta).

hello(brock).
hello(roberta).
hello(john).

world(ashwin).
world(roberta).

goal(X) :- sub1(X),sub2(X).
sub1(X) :- foo(X).
sub2(X) :- hello(X),world(X).
```

1. Reorder the facts (i.e., not the rules) above to provide faster execution time when querying `goal(X).` List the re-ordered facts.
2. Explain in your own words why reordering the facts affects total execution time. Show evidence of the faster execution time (provide a trace for each).
3. Going back to the *original* ordering of facts, now suppose we rewrite goal `sub1` to read: `sub1(X) :- foo(X),!.` Upon returning to query mode and querying `goal(X)`, the interpreter will display *false.* Explain why.
4. Going back to the *original* ordering of facts, now suppose we rewrite goal `sub2` to read: `sub2(X) :- hello(X),!,world(X).` Upon returning to query mode and querying `goal(X)`, the goal this time will succeed. Explain why.

4. [15 points] **Unification**

   For each pair below that unifies, show the bindings. Circle any pair that doesn't unify and explain why it doesn't.

   1. d(15) & c(X)

   2. a(X, b(3, 1, Y)) & a(4, Y)

   3. a(X, c(2, B, D)) & a(4, c(A, 7, C))

   4. a(X, c(2, A, X)) & a(4, c(A, 7, C))

   5. e(c(2, D)) & e(c(8, D))

   6. X & e(f(6, 2), g(8, 1))

   7. b(X, g(8, X)) & b(f(6, 2), g(8, f(6, 2)))

   8. a(1, b(X, Y)) & a(Y, b(2, c(6, Z), 10))

   9. d(c(1, 2, 1)) & d(c(X, Y, X))