Spring 2013
Programming Languages
Homework 4

- This homework is a combination programming and "paper & pencil" assignment.
- Due via NYU Classes on Tuesday, April 30 at 5:00 PM Eastern Time. Submit your programming files as attachments.
- For the Prolog questions, submit **separate** files for rules and queries. In a comment above each rule and query, write the number of the Prolog question you are answering. Submit your short answers either inline or in a PDF attachment.
- For the Prolog questions, you should use SWI Prolog. A link is available on the course page. For the C++ questions, you may use any standards-compliant C++ compiler. GNU C++ (g++) is recommended.
- Due to timing considerations, there will be no point recovery for this assignment. However, you may earn up to 110 points out of 100—you can lose up to 10 points and still earn a perfect homework score.
- 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 and C++ 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.

C++ Standard Template Library [20] Refer to the Standard Template Library (STL) documentation. A link is available under the Resources folder of the course page. Please answer the following questions:

1. Write a C++ program that will use an STL function to iterate through an array of integers and compute a running average. After the average is computed, the program should print the average. Pass a function object of type *Average* to the STL function for this purpose. See slide 17 of the generic programming lecture for ideas on how to get started.

   Here's a skeleton program to start with. Replace "..." with your own code.

```
#include <iostream>
#include <algorithm>

using namespace std;

// Function object
class Average
{
   ...
};

int main()
{

 int myArray[] = { 5, 1, 27, 29, 17, 0, 15, 92 };
 const int N = sizeof(myArray) / sizeof(int);

 // Note: Primitive arrays don't have a begin() or end() function.
 //       Since they are random access containers, use these instead.
 const int* begin = myArray;
 const int* end = myArray + N;

 Average avg;

 // Call the STL function here.
 ...

 cout << "Average is: " << avg.getAverage() << endl;
 return 0;
}
```

2. Write a generic algorithm `combine` which accepts the following parameters, whose types should all be generic. The first three parameters should be the same generic type:

- A ForwardIterator pointing to the beginning of a first sequence
- A ForwardIterator pointing to the end of a first sequence
- A ForwardIterator pointing to the beginning of a second sequence
- A BinaryFunction which accepts 2 integer inputs and returns an integer output.
- An OutputIterator for writing each result.

You are encouraged to look at the STL documentation for ideas, including looking at how STL functions are implemented.

The `combine` function should take the first item from each sequence and feed them as input to the binary function, and write the output of the binary function through the OutputIterator—then repeat with the next item in both sequences. Your algorithm should stop iterating as soon as the end of the first sequence is reached. You may assume a precondition that the second sequence is at least as long as the first.

For example, assuming `begin` and `end` are defined as above and you've written a function object `adder` that accepts two arguments of a specified generic type and returns the sum, the call:

```
combine( begin, end, begin, adder<int>(),
                  ostream_iterator<int>(cout, " ") );
```

should output: `10 2 54 58 34 0 30 184`.

Prolog [50] Consider the fictitious business-oriented social network, JoinedIn. This network allows users to "join" with other users for the purposes of professional networking and job seeking.

JoinedIn works as follows: after you sign up, you browse the JoinedIn network for users you already know. You then ask each user you know to connect with you. If the user accepts your request, they *join* your network and become a *first-degree* connection. The relation is symmetric, so you also become their first-degree connection.

Let $u_1 \leftrightarrow u_2$ denote the join relationship. A user u_i is said to be "in the network of u_1" if there exists a path of join relations: $u_1 \leftrightarrow \ldots \leftrightarrow u_i$.

A user u_i is said to be "an n-th degree connection of u_1" if the shortest path from u_1 to u_i is precisely n join relations apart. It follows from the symmetry property that u_i is also an n-th degree connection of u_1.

You may share an update by writing a brief message. First degree connections can view your messages on their *feed* and also assert that they *like* your update. When a message is "liked," by a user, it is propagated to the feed of the "liker's" first-degree connections. Those connections may also "like" the message, etc. This may cause a message authored by a user to propagate to many people who are not first-degree connections to that user. (For those familiar with Twitter, this concept is similar to "retweeting" a message.)

In the questions below, "stating a fact" means that you are writing a relation with no subgoals, parameterized by all atoms. For example, `single(joe)`, and `married(lisa,david)` are facts. "Writing a relation" means that you are writing one or more rules, usually having variable parameters and subgoals. For example, `daughter(X,Y) :- female(X),mother(Y,X)`. Facts and relations are both written in user mode. The phrase "write a query" means go to query mode and query the database using the relations you stated earlier. Now let's get started:

1. Create a number of JoinedIn users by stating several `user(U)` facts, where U is the name of the user.

2. State several `requestJoin(U1,U2)` facts signifying that $U1$ has asked $U2$ to join their network (i.e., become a first-degree connection).

3. State several `acceptJoin(U2,U1)` facts, which accept a request to join. `U2` is the user who is accepting and `U1` is the user whose request is being accepted. (It is not necessary that you enforce that the request occur before the accept, as long as accepting without a request does not have the affect of creating a connection.)

4. Write a relation `join(U1,U2)` whichs asserts a first-degree connection between user `U1` and `U2`. This relation should be symmetric: i.e., `join(a,b)` $\Leftrightarrow$ `join(b,a)`.

5. Write a query demonstrating the `join` rule.

6. Write the relation `connection(U1,U2)` which asserts that `U2` is an n-th degree connection of `U1`, for any n.

   Here is one wrong answer:

   ```
   connection(U1,U2) :- join(U1,U2).
   connection(U1,U2) :- join(U1,UM),connection(UM,U2).
   ```

   Try this in SWI Prolog. Explain your observations.

7. To fix the problem above, we introduce the following relations:

   ```
   travel(U1,U2,Visited,1) :- join(U1,U2), \+member(U2,Visited).
   travel(U1,U2,Visited,L) :-
       join(U1,UM), \+member(UM,Visited),
       travel(UM,U2,[UM|Visited],L1),
       L is 1+L1.

   shortest(U1,U2,N) :-
       setof(L,travel(U1,U2,[U1],L),Set),
       min_list(Set,N).

   connection(U1,U2,N) :- shortest(U1,U2,N), \+ (shortest(U1,U2,N1), N1 < N).
   ```

   We have redefined `connection` so that it seeks the shortest path. Using this new version of `connection` above, write a query that shows all users to whom a particular user is connected (in any degree).

8. Write a query that, given a user and some number $n > 0$, will show all n-degree connections.

9. Write several facts `update(U,M)` asserting that JoinedIn user `U` has posted a message `M` (represented by a list of words). An example update looks like this: `update(bob,[this,is,a,message]).` Remember, don't capitalize words or they will be treated as variables. For the sake of simplicity, we shall assume that every message is unique.

10. Write several facts `like(U,UA,M)` which assert that user `U` "likes" a message `M` written by user `UA`.

11. Create a Prolog relation `feed(U,M)` which establishes user U's feed. M represents any message appearing in the feed. Recall that the feed should consist of any message written by any of U's first-degree connections or any messages "liked" by U's first-degree connections. (If you wish, you may prepend each message with the name of the author.)

12. Write a query that shows the feed of a specified user.

13. Create your own Prolog relation that makes non-trivial use of the items above, then write a query using that relation. Also briefly explain the meaning of your relation.

More Prolog [15] 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.

Unification[10] 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))

ELIZA[15] Prior to Apple's popular intelligent personal assistant Siri, there was ELIZA—a well-known program written in the mid 1960's featuring a psychoanalyst named Eliza who carries on a conversation with the user. The application was later ported to Prolog, a variation of which is posted on the course page. Please read the source code, experiment with the program, and then answer the following questions:

1. At the bottom of the source code listing is the rule `eliza` whose subgoals serve as the "main" program. Explain at a high level how ELIZA processes user input.
2. Near the top of the source code listing, there are a number of simplification rules named `sr`. Briefly explain the general purpose of these rules.
3. Run ELIZA and type "everyone always hates me", then observe ELIZA's response. Now go to the source code and identify the `rules` fact which caused the response.
4. Note that the phrase "everyone always hates me" contains multiple keywords that have corresponding `rules` facts. Briefly explain how ELIZA picks which one to use.
5. Add a new `rules` fact and give an example user input which causes it to be invoked. Write the fact and user input here.