Fall 2013
Programming Languages
Homework 4

- Due via NYU Classes on Tuesday, December 3 at 3:00 PM Eastern Daylight Time.
- This homework is a combination programming and "paper & pencil" assignment. All short answer solutions and Prolog solutions should be submitted in a PDF document. C++ questions should be answered by submitting a single header file and implementation file containing the solutions to both.
- For the Prolog questions, you should use SWI Prolog, for which a download link is available on the course page. For the C++ questions, you may use any standards-compliant C++ compiler. GNU C++ (g++) is both free and recommended.
- 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 and be subject to disciplinary action.

Prolog Translation [20] Encode the following sentences as Prolog clauses (facts, rule, or query). To differentiate facts from queries, please write the Prolog query prompt `?-` before queries. Be sure your solutions reflect the difference between assertions and questions.

- All pigs roll in mud
- It is sunny on Thursday
- If it is sunny or party cloudy, then I go walking
- All humans are mammals
- Jessica is a human
- Some mammal is a human
- C#, ML, and Ada are programming languages
- There is a book that I like
- I love all books by Stephen Hawking
- If something walks like a duck and talks like a duck, then it's a duck
- If something walks like a duck, but does not talk like a duck, then it's a frog.
- Everything walks like a duck
- Something walks like a duck
- Does something walk like a duck?
- Who walks like a duck?
- Googly is a muppet
- Are Chomki or Googly muppets?
- Is Ted a frog?

Prolog Relations [15] Define the following relations. It's okay to define other relations to support these, if necessary. (The number after the slash represents the arity, or number of parameters, of the relation):

- `parent/2`
- `father/2`
- `mother/2`
- `son/2`
- `sibling/2`
- `brother/2`
- `sister/2`
- `grandson/2`
- `granddaughter/2`
- `cousin/2`
- `niece/2`
- `nephew/2`
- `ancestor/2` (a recursive `parent` relation)

Write several facts using these relations. Then write several queries demonstrating the use of these relations—use each relation at least once in a query.

Prolog Query Processing [20] Consider the following facts and rules:

```
blue(boat).

yellow(car).
yellow(plane).
yellow(boat).

red(car).
red(boat).

orange(X) :- red(X),green(X).
green(X) :- yellow(X),blue(X).
cyan(X) :- green(X),blue(X).
```

1. What instantiations of `X` exist for the query `orange(X)`?
2. Reorder the facts (i.e., not the rules) above to provide faster execution time when querying `orange(X)`. List the re-ordered facts.
3. 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).
4. Going back to the *original* ordering of facts, now suppose we rewrite goal `orange/1` to read: `orange(X) :- red(X),!,green(X).` Upon returning to query mode and querying `orange(X)`, the interpreter will display *false*. Explain in plain terms why this is the case.

Unification[15] For each pair below that unifies, show the bindings. Circle any pair that doesn't unify and explain why it doesn't.

1. a(X) & a(42)

2. a(X,12) & a(7,Y)

3. a(X) & b(Y)

4. a(a(5,X),X) & a(Y,Z)

5. c(X) & c(c(c(X)))

6. c(X) & c(X)

7. a(b(c(X))) & c(b(a(3)))

8. b(1,2) & b(1,6)

9. d(c(A),2) & d(c(2),B)

10. b(X, c(2, X, Z)) & b(4, c(W, 7, Y))

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

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, using a random access container (i.e., deque, vector, primitive array) of integers, generates the initial contents, randomly shuffles the contents, then sorts the contents in ascending order. All three actions listed above should be carried out primarily with built-in STL functions.

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

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

using namespace std;

int main()
{
 int myArray[10];   // primitive array
 const int N = 10;

 // Note: Primitive arrays don't have a begin() or end() function.
 const int* begin = myArray;
 const int* end = myArray + N;

 // Write your code here...
 ...

// Print the sorted sequence to standard out
 copy(begin,end, ostream_iterator<int>(cout,"\n"));

 return 0;
}
```

2. Write a generic algorithm `foldl`, which has the same semantics as we've seen in the functional languages. It should accept the following parameters (all of whose types should be generic):

   - A Forward Iterator pointing to the beginning of a first sequence
   - A Forward Iterator pointing to the end of a first sequence
   - A Binary Function
   - A seed value

   Your `foldl` implementation should successfully operate on Forward Containers of any *any type*. Make sure to write your algorithm in a header file.

See STL documentation for the meaning of Forward Iterator, Binary Function, and Forward Container. For a head start, you may begin with the code below. Your header file `file_containing_foldl_definition.h` (rename it anything you wish) might look as follows:

```
template <typename Iter, typename BinF, typename Res>
Res foldl (Iter first, Iter last, BinF func, Res seed)
{
   ... Insert definition here ...
}
```

Your implementation file `file_containing_foldl_calls.cc` (again, rename as desired) might look as follows:

```
#include "file_containing_foldl_definition.h"

int add (int x, int y) { return x+y; }

int main()
{
 // array, Random Access container (a refinement of Forward Container)
 int begin[] = { 1, 2, 3, 4, 5 };
 int N = sizeof(begin) / sizeof(int);

 // Should output 15
 cout << foldl( begin, begin+N, add, 0 ) << endl;

 // Linked list, a Sequence (refinement of Forward Container)
 list<int> l;
 l.push_back(1);
 l.push_back(2);
 l.push_back(3);
 l.push_back(4);
 l.push_back(5);

 // Should output 15
 cout << foldl( l.begin(), l.end(), add, 0 ) << endl;
 return 0;
}
```

Memory Alloction [10] Do this problem **after** the lecture on garbage collection and memory allocation. In class we studied *first-fit* and *best-fit* allocation strategies. There's also a *worst-fit* strategy, which is quite simple: for a given request, use the largest available block of free memory possible, provided one exists. The idea behind this approach is to reduce fragmentation.

Assume a free list consisting of blocks of the following size (expressed in kilobytes): 6, 10, 3, 2, 7, and 12. Now assume allocation requests are made of sizes 2, 5, 8, 3, 7, in that order.

For each of the three allocation strategies:

1. explain whether the allocation requests were successful.
2. show what the free list looks like after the requests have been processed. If unsuccessful, show what the free list looks like as of the last successful request.