Ryan Goodfellow - CS 580 Final Exam - Written Part

Problem 1 *Grammars*

$$
\begin{aligned}
&< mul > \coloneqq < mul >< exp > \\
&\qquad\quad \mid \ < t >
\end{aligned}
$$

$$
\begin{aligned}
&< exp > \coloneqq < t > \ \hat{} \ < exp > \\
&\qquad\quad \mid < t >
\end{aligned}
$$

Problem 2 *Function closures and storage management*

There are two primary issues that come to mind when thinking about higher order functions and storage management. These issues are tied specifically to functions that capture their referencing environment when created (closures). Dealing *variable lifetime* and *scope* in this case becomes problematic without garbage collection. When a first class function is defined as a closure, it has visibility of the variables within the referencing environment in which it is defined. Because the function is a first class function it may be arbitrarily passed around the program just like any other variable. This means that the referencing environment in which the function was defined may be entered from any other arbitrary referencing environment within the program. Thus the variables within the closure of the function have a lifetime that is is tied to that of the function itself and also have unlimited scope.

It is easy to see how this creates an utter mess in a language without garbage collection. A simple example; the programmer must ensure that any variables within a closure are allocated on the heap. Imagine the alternative, function local variables declared on that functions stack, are lost immediately when the function returns. What if these variables are within the closure of some first-class function and get called later. In C++ we call this 'undefined behavior' in the general case and a segmentation fault in the usual case (bad). As you can see this takes the danger of a pointer escaping a referencing environment and extends it to the entire environment (yikes!). Therefore, the language must make special accommodations for closures ensuring that this does not happen (and indeed C++11 does with the new lambda expressions). This is a prime example of both the memory issue (variable lifetime) as well as variables escaping their referencing environment (unlimited scope). This problem may be manageable my careful programming on a small scale by 1 or a very small group of programmers sticking to tight *conventions* for development. However, in the general case, higher order functions especially ones with closures raise very difficult issues in medium to large scale collaborative software development.

Problem 3 *Argument evaluation order*

Language: C++
Implementation: *OS* : Mac OS X 10.7
Compiler : g++, clang++
Code:

```
#include<iostream>
#include<string>

using std::cout;
using std::endl;

void f(int exp1, int exp2, int exp3) { return; }
int i() { cout << " I "; return 0;}
int like() { cout << " like "; return 0;}
int muffins() { cout << " muffins "; return 0;}

int main(void)
{
  cout << "Hi my name is Ryan and - ";
  f(i(),like(),muffins());
  cout << " ! " << endl;

  cout << "Hi my name is Ryan and - ";
  f(muffins(),like(), i());
  cout << " ! " << endl;

  cout << "Hi my name is Ryan and - ";
  f(like(),i(),muffins());
  cout << " ! " << endl;

}
```

Output:

clang++ Compiler

```
Hi my name is Ryan and -  I  like  muffins  !
Hi my name is Ryan and -  muffins  like  I  !
Hi my name is Ryan and -  like  I  muffins  !
```

g++ Compiler

```
Hi my name is Ryan and -  muffins  like  I  !
Hi my name is Ryan and -  I  like  muffins  !
Hi my name is Ryan and -  muffins  I  like  !
```

Discussion: Well this is fun. We have the same code running on the same platform, but executables made by different compilers and we have a different result! clang++ has chosen to evaluate arguments left to right and g++ right to left. If you look at the C++ standard this fact does not come as a surprise. C++ does not specify argument evaluation order, it is left up to the implementor, in this case the GNU folks had a different take than the LLVM folks. If you want to try clang/clang++ on Ubuntu it is one apt-get away. I have included a little bash script that will compile using both assuming g++ and clang++ are in your path.

Problem 4 *True Iterators using Threads*

The bulk of the work here is in the code which is included. I don't think there is anything sneaky going on. I generalized a bit by allowing non-integer indexes (why I don't know but hey seemed neat at the time). The bulk of what is happening is implemented in the IteratorHelper class. This class owns and controls the iterator thread and its interaction with the main thread. I believe that this is in the spirit of the 'true iterator' in the fact that the only state that is maintained is the index that iterates over the object which it is iterating. Any other state is not necessary. To compile just run javac on the Program.java file.

```
//Ryan Goodfellow CS580 Final
public class IteratorHelper<V,I> {

    I place; //The current index
    Itr<V,I> goose; //The object under iteration
    Thread t; //The thread being used for the iteration
    I last; //The last index used for yielding an element

    public void start() {
        last = null; //initialize the most recent item produced to nil
        t = new Thread(goose); //create a new thread for iteration
        t.start(); //start the iteration thread
    }

    public IteratorHelper (Itr<V,I> g) {
        this.goose = g; // Init member vars
        this.place = g.min();
    }

    public void yield(I i) {
        this.place = i; //update the current place
        t.suspend(); //suspend the iteration thread
    }

    public synchronized V get() {
       // It is possible that the iterator thread has not had a chance yet to iterate
       //if this is the case wait for it
       while(last != null && last == place) {
                t.resume();
                Thread.yield();
       }
       last = place; //update the last used index
       V i = goose.elementAt(place); //get the value at the current index
       t.resume(); //resume the iterator thread
       Thread.yield(); //Try to yield to the iterator thread to let it do it's thing
       return i; //return the value at the current index of the object under iteration
   }

    public boolean done() {
        //We are done when we have reached the maximum value of the object under iteration
        return place.equals(goose.max());
    }

}
```