Kelsey Cairns

Cpts 580 Final

The timestamps are not very accurate for the longer problems because I would continually get distracted. Also, any overlapping timestamps are due to doing problems on different days.

Problem 1

Total Time: 17:35 - 17:45

```
T -> TE | E
E -> <t> | <t> ^ E
```

Kelsey Cairns

Problem 2

Reading: 13:10 - 13:45 Writing: Intermittently between 14:30 and 16:00.

The availability of first class function values in a given languages boils down to the management of referencing environments. All first class values need to be stored somewhere, and in the case of first class functions, not only does the function requiring storage space, but its referencing environment does as well. For stack based languages, values passed to a function or created within it are not inherently persistent. This presents the problem of where to store the function and its referencing environment. Unless the languages uses shallow binding, the values in the referencing environment could potentially change between the function instance's creation and application. This creates the need to store the referencing environment is a location that persists across program execution. The only two places the referencing environment could be stored are the stack and the heap. Using the stack would be possible, but impractical. The entire referencing environment would have to be copied each time a function instance was passed to or from a function call. Using the heap is far more practical, but some circumstances require special attention to deallocate the space.

Languages that use a lot of heap storage are generally easier to use with garbage collection, but the problem of heap management in the presence of first class functions spreads beyond ease of use into the practical realm. For instance, assume a language supporting nested function definitions with deep binding, and imagine that a function instance f is created within and returned from another function g, and f's referencing environment relies on values within g's scope. Since the spaced used by g become fair game once g returns, the values required by f must be preserved in the heap. This brings deallocation into question. A garbage collector can track the reference to f, deallocating the heap space once f has been applied for the last time. Manual deallocation is far less likely to happen correctly. In a more extreme example, imagine a line a functional language code that creates an anonymous function instance and applies it, without ever storing a reference to the function instance. If the function's referencing environment was stored on the heap, what reference would the programmer use to deallocate it? Garbage collection is practically required in this case to prevent the heap from overflowing with dead referencing environments. These examples are specific, but they serve to show how special measures must be taken to support of first class functions in the absence of garbage collection.

Problem 3

Coding: 16:40 - 16:50 Writing: 17:10 - 17:25 Documentation reading: 17:25 - 17:30

The language used tested in this question was C, compiled by gcc version 4.6.1 on Ubuntu 11.10.

The code is included in `ordertest.c` and is reproduced here along with the output.

ordertest.c:

```
#include <stdio.h>

void fun(int a, int b, int c){
    printf("a = %d, b = %d, c = %d\n", a, b, c);
}

void main(){
    int i = 5;
    fun(i++, i++, i++);
}
```

Output:

```
kels@Lilac:~/5801/final$ gcc -o ordertest ordertest.c
kels@Lilac:~/5801/final$ ./ordertest
a = 7, b = 6, c = 5
```

In this test the side effects of using the `++` operator cause the function arguments to incrementally increase. The evaluation order determines whether this increase will be appear to be left-to-right or right-to-left. As evidenced by this program, the evaluation order here is right-to-left. The initial value of `i` is 5. The value printed by the function for the first argument is 7. The only way for the first argument (`i++`) to result in 7 is if the remaining arguments had already been evaluated, incrementing `i`. This shows that arguments must be evaluated in right-to-left.

This result is specific to the language and compiler implementation. The document *Using the GNU Compiler Collection* found at
`http://gcc.gnu.org/onlinedocs/gcc-4.6.3/gcc.pdf`
demonstrates a very similar function, stating (on page 637):

> There is no guarantee (in either the C or the C++ standard language definitions) that the increments will be evaluated in any particular order.

And I trust the GCC documenters as a reliable source.

Problem 4

Reading and Coding: Between 12:30 and 14:00 Writing: 4:30-17:20

There is nothing tricky or clever about this code. The `IteratorHelper` class simply manages another thread that runs the iterator's run method. Each time a request is made for an iterator value, the helper class orchestrates a hand-off between the calling thread and the thread running in the iterator loop. The calling thread is paused until the iterating thread yields a value, and the iterating thread is then pause end the original thread becomes executable again. This way the iterator thread can simply pause between iterations, and resume execution at the instruction where it left off.

The thread control is handled in `IteratorHelper.get()` and `IteratorHelper.yield()`. The `get()` method gets a reference to the currently running thread (the calling thread) and saves it. The iterator thread is `notify()`'d, and the current thread is `wait()`'d. The reverse operation happens in `yield()`, `notify()`'ing the saved thread and `wait()`'ing the iteration thread. Synchronized blocks ensure no interference between the two threads. Whenever one thread `notify()`'s another, it acquires the lock associated with the thread being `notify()`'d. This ensures that that no thread can be `notify()`'d unless it is sleeping. The threads may run concurrently after `notify()`'ing each other, but the immediate action of the thread responsible for the `notify()` is to `wait()`. It will not wake up again until signaled by the other thread, guaranteeing it will not interfere with the other thread's execution.

Note that this iterator is not safe in the context of several threads, where multiple threads could access the iterator or the threads could receive a `notifyAll()`.

This code is incomplete. The `hasNext()` function checks to see if the iterator thread is still active or not. However, this is not an actual measure of weather or not the iteration has completed because the run method will be alive until it returns. After the last value is yielded, the thread will be alive until it is allowed to return method. I felt this was more of an implementation detail that would probably detract from the clarity of the code more than it would contribute to the point of the assignment, so I left it as is.

Along with the Java documentation, I looked at a few online sources, mostly to remind me how to use threads and generics:

Java in a Nutshell
`http://onjava.com/onjava/excerpt/javaian5_chap04/index1.html`

Thinking in Java: Wait and Notify
`http://www.linuxtopia.org/online_books/programming_books/thinking_in_java/TIJ315_016.htm`