Debuggers and Concurrency Exercises
CS 4610 — Spring 2017

This Review Set asks you to prepare written answers to questions on debuggers and concurrency. Each of the questions has a short answer. You may discuss this Review Set with other students and work on the problems together.

1 Debuggers

1. Recall that a compiler can generate several mappings to help support breakpoints and single-stepping in high-level source code (e.g. Cool). Use the `list.cl` example code to formulate answers for the following parts. You can use the `--asm` flag to the reference compiler to generate Cool assembly.

   (a) Create a mapping from Cool source code line number to ranges of Cool assembly for each executable line in the `Main` class.

   (b) Create a mapping from Cool source code line number to in-scope variables for each executable line in the `Main` class. If you're feeling bold, also include the locations (in memory or registers) for each variable as well.

2. It can be nice to step backwards through code sometimes. For example, you might single-step "too far" and wish to go back one line. Unfortunately, it is difficult to "undo" instructions because of side-effects. Using your knowledge of debugger implementation, describe how you would implement a debugger that gives the illusion of "stepping backwards". (Hint: This feature is often called *replay debugging*.)

2 Concurrency

1. One way to gain parallelism in a program is to identify loops in which there are no dependencies between iterations. These loops iterations can be executed in parallel. Suppose we wish to implement a custom language structure to capture this parallelism:

```
1 parallel_for( Foo x : Collection<Foo> ) {
2     ... body ...
3 }
```

This `parallel_for` performs the same computation on `x` for each value in the collection. Propose two different techniques for using fork/join task creation to spawn tasks to perform each loop iteration in parallel. Which of your proposed techniques is more efficient? Why?

2. Consider the following function written in the Cilk language:

```
int fib(int n)
{
    if (n < 2)
        return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

Draw a directed acyclic graph (DAG) for a call to `fib(5)` (fork/join operations are indicated by nodes; all other instructions are on edges). How many strands are there?

3. Explain why big-step operational semantics (the kind of semantics used to describe the Cool language) are not sufficient for modeling concurrency. What is one approach that provides a better model?