CSE 3302 Assignment 3
Due: 30 Mar 2010, 11am

Print out a hard copy of your answers, including code, and submit in class on the due date, or email them to me before the deadline. This is a hard deadline—extensions will not be granted. Type or write neatly.

1 Functions
[6 pts] Write the following anonymous functions in Scala:

(a) A function that takes a String and returns it.

(b) A function that returns a function that returns 2.

(c) A function that takes an Int n and a function f from Int to Int and applies f to n.

(d) The curried version of (c).

(e) A function that takes an Int n and two functions f and g from Int to Int and applies f to the result of g applied to n.

(f) The curried version of (e).

2 CPS
[20 pts] Convert the following functions to continuation-passing style (cps). Every function should take an additional argument—the continuation—that represents the return context and which should be invoked to return from the function. If the function returns type T, the cps version of the function should take a continuation of type T => T and should return T. No function should return a value directly; instead, it should invoke its continuation, which will in turn return a value.

(a) def zero(): Int = 0

(b) def plus1(x: Int): Int = x+1

(c) def reciprocal(n: Float): Option[Float] = if (n == 0) None else Some(1/n)

(d) def andmap(f: Int => Boolean, ls: List[Int]): Boolean = ls match {
   case Nil => true
   case x::xs => andmap(f, ls) && f(x)
   }

(e) def take(n: Int, xs: List[Int]): List[Int] = {
   if (n == 0) xs
   else xs match {
      case Nil => Nil
      case y::ys => take(n-1, ys)
   }
   }

(f) def downto(n: Int): List[Int] = {
    if (n == 0) 0::Nil
    else n::downto(n-1)
}

(g) def sum(xs: List[Int]): Int = xs match {
    case Nil => 0
    case y::ys => y + sum(ys)
}

(h) def collatz(n: Int): List[Int] = {
    if (n == 1) 1::Nil
    else if (n % 2 == 0) n::collatz(n/2)
    else n::collatz(n*3+1)
}

(i) def fib(n: Int): Int = {
    if (n <= 1) 1
    else fib(n-1) + fib(n-2)
}

(j) The fold function in the following code:

```scala
class Tree
object Leaf extends Tree
case class Node(x: Int, left: Tree, right: Tree) extends Tree

def fold(t: Tree, f: (Int,Int,Int) => Int, z: Int): Int = t match {
    case Leaf => z
    case Node(x, l, r) = f(x, fold(l, f, z), fold(r, f, z))
}
```

3 Failure continuations

[4 pts] Rewrite the reciprocal function above with both a success and a failure continuation. The success continuation should be invoked when returning a normal value. The failure continuation should be invoked when dividing by 0. The success continuation should have type Float => Float. The failure continuation should take an error message ("div by 0") and should be of type String => Float. The function might be invoked as:

```scala
scala> reciprocal(10, x => x, msg => { println(msg); Float.NaN })
res11: Float = 0.1
scala> reciprocal(0, x => x, msg => { println(msg); Float.NaN })
div by 0
res12: Float = NaN
```

4 Iterators

[5 pts] In this problem, we’ll characterize coroutine iterators in terms of continuations. Consider the following binary tree iterator:
class Tree<A> {
    Tree<A> left, right;
    A value;
    ...

    elements() yields A {
        if (left != null) for (v : left.elements()) { yield v; }
        yield value;
        if (right != null) for (v : right.elements()) { yield v; }
    }
}

And consider the following for loop:

for (v : tree.elements()) {
    println(v);
}

[2 pts] When is a continuation taken and what does it do when invoked? In the body of the iterator `elements`, when are continuations taken and what do they do when invoked?

[3 pts] Trace through the control flow of the loop above applied to the tree:

```
4
 / \
2 5
 / \
1 3
```

Show how control moves between the for loop and the active invocations of the `elements` iterators and back.