There are three problems on this assignment, worth 10 points each. The first two involve a bit of Scala programming. You can use the interpreter to develop and test your code. You can download the Scala compiler and interpreter from scala-lang.org, or use it on omega.uta.edu—the interpreter can be invoked with:

$ ˜nystrom/3302/scala/bin/scala
scala>

Or, to run on a file containing your code:

$ ˜nystrom/3302/scala/bin/scala Foo.scala

To test your code, you can just type your functions into the interpreter and then invoke them from the prompt. Or, you can put the code into a file with invocations of the code at the end. Use println to print values.

Print out a hard copy of your answers, including code, and submit in class on 2 March, or email them to me before the deadline. This is a hard deadline—extensions will not be granted. Type or write neatly.

1 Tail recursion

The familiar Fibonacci sequence $F(n)$ is given by the inductive definition:

\[
F(0) = 1 \\
F(1) = 1 \\
F(n) = F(n-2) + F(n-1)
\]

This definition leads immediately to a straightforward algorithm for computing $F(n)$. This algorithm can be implemented in Scala as follows:

```scala
def fib(n: Int): Int = if (n <= 1) 1 else fib(n-1) + fib(n-2)
```

While simple, the algorithm runs in exponential time. A linear algorithm takes advantage of the fact that fib(n-1) calls fib(n-2), and reuses the result when computing fib(n).

1. Write a recursive Scala function with the following signature:

```scala
def fib2(n: Int): (Int, Int)
```

that computes both fib(n) and fib(n-1) in linear time. The function should evaluate as follows:

\[
\begin{align*}
\text{fib2}(0) &= (1,1) \\
\text{fib2}(1) &= (1,1) \\
\text{fib2}(2) &= (2,1) \\
\text{fib2}(3) &= (3,2) \\
\text{fib2}(4) &= (5,3) \\
\end{align*}
\]

...
The `fib` function can then be implemented efficiently using `fib2` and selecting the first element of the pair returned:

```scala
def fib(n: Int) = fib2(n)._1
```

2. Try running `fib(100000)` in the Scala interpreter. What happens? Is the function `fib2` you wrote in part (a) tail recursive? Explain why or why not. If not already tail recursive, write a new version of `fib2`, called `fib3`, that is tail recursive. `fib3` might possibly with a different signature than `fib2`. Show how to implement `fib` using `fib3`.

2 Lists

Scala defines a generic `List` class. For any type `A`, `List[A]` is a list of `A`. `Nil` is the empty list. A new list can be constructed by prepending an element to an existing list: if `x` is an `A` and `xs` is `List[A]`, then `x :: xs` is the list consisting of `x` followed by all the elements of `xs`. For example, using the scala interpreter:

```
scala> Nil // the empty list
res0: scala.collection.immutable.Nil.type = List()

scala> 1 :: Nil // the List[int] consisting of just the int 1
res1: List[Int] = List(1)

scala> 1 :: 2 :: Nil // the List[int] consisting of 1 and 2
res2: List[Int] = List(1, 2)

scala> List(1, 2) // the List[int] consisting of 1 and 2
res3: List[Int] = List(1, 2)
```

Note that lists are immutable: performing an operation on a list returns a new list; lists are never modified in-place.

Write a function `scan` with the following signature:

```scala
def scan[A](f: (A, A) => A)(list: List[A]): List[A]
```

`scan` takes a function `f` of type `(A, A) => A` and a `List[A]` and returns a new `List[A]` of the same length such that:

- The result list is empty if the input list is empty.
- The first element of the output list is copied from the first element of the input list.
- The `i`th element of the output list is the result of applying the function `f` to the the `(i-1)`th element of the output list and the `i`th element of the input list.

Note that the function is curried; that is, applying the function to a single argument returns a new function. For example, `scan[Int](._+_.)` returns a function that takes a list of integers and returns a new list where element `i` is the sum of elements 0 through `i` of the original list. Here are some examples of its usage:

```
scala> scan[Int](._+_.)(List(1, 2, 3, 4))
res35: List[Int] = List(1, 3, 6, 10)

scala> scan[Int](._*_.)(List(1, 2, 3, 4))
res36: List[Int] = List(1, 2, 6, 24)

scala> scan[Int](._ min _)(List(3, 1, 4, 1, 5, 9, 2, 6, 5, 4))
res37: List[Int] = List(3, 1, 1, 1, 1, 1, 1, 1)
3 Scoping

In this problem we'll look at a few variations of a simple program. Scala syntax is used, but the semantics are different. Consider the following program:

```scala
def f = {
  val a = 1
  (x: Int) => x + a
}
val a = 20
println(f(300))
```

What does the program print if static scoping semantics are used? What about dynamic scoping semantics?

Now, consider this slightly modified version of the program:

```scala
def f = {
  (x: Int) => x + a
}
val a = 20
println(f(300))
```

What happens with static scoping? Dynamic scoping? Does it make a difference if errors are caught at compile time versus at run time?

Finally, consider this version:

```scala
def f = {
  val a = 1
  (x: Int) => x + a
}
println(f(300))
```

What happens with static scoping? Dynamic scoping? Again, does it make a difference if errors are caught at compile time versus at run time?