CS 430 Programming Languages
Spring 2017
Module 11 Lab

1. Go to a directory where you can save files. In an editor, open a file called `lab11.hs` in one window. Open another window, go to the same directory, and start `ghci`.
2. In your lab file, write a function called `listOf4` that takes four parameters and combines them into a list, so that, for example, `listOf4 3 1 6 7` is `[3,1,6,7]`. Load this file into `ghci` and make sure it works, then use `:t` to determine its type. Unsurprisingly, it should be `listOf4 :: a -> a -> a -> a -> [a]`. We have talked before about how in the type signature, the last type is the function result and all the rest are the parameter types. But why all the arrows? Actually, the arrows indicate functions, and they are right associative. Thus the type of `listOf4` is really `a -> (a -> (a -> (a -> [a])))`, and this says that `listOf4` takes a parameter of type `a` and returns a function. This result function takes a parameter of type `a` and returns a function. This result function takes a parameter of type `a` and returns a function. This result function takes a parameter of type `a` and returns a list of values of type `a`. This idea of thinking of a function of multiple parameters as really a function of a single parameter returning another function of a single parameter, which itself returns a function of a single parameter, and,so on, is called *currying*, named after Haskell Curry (yes, that Haskell). (Curry did not actually come up with currying, but it is named after him anyway.)
3. Functions are represented internally in Haskell as curried functions. So, for example, `listOf4 8` returns a function. This function tacks on an 8 at the start of a list of three elements that it makes into a list. So we could define the function `listOf4With8 = listOf4 8`; similarly, we can make the function `listOf4With88 = listOf4 8 8`. And so on. Try them and see.
4. So what? This has a consequence that is useful for defining functions. If we have a function of several values and need a function just like it but with fixed parameters, we can make one out of it by currying. For example, we might define a `double` function as `double n = 2*n`. But with currying, we can just define this as `double = (2*)`. Defining a function without mentioning its parameters is called a *point-free definition*. A function that is a partial application of an infix operator (like `(2*)` or `(+8)`) is called a *section*. Try it.
5. Functions are first-class entities in Haskell. This means that they can be returned as results (which is what currying does). This also means that they can be parameters. Suppose we want to do something to every element of a list. For example, suppose we want to double every element of a list. We could make the following definition.
   ```
   doubleList [] = []
   doubleList (n:ns) = (double n):(doubleList ns)
   ```
   This works fine, as you will see if you try it.
6. But suppose we want to triple every element of a list, or do something else to every element of a list? Then we have to write another function every time. But what if defined

a function like the following?

```
applyToList :: (a -> a) -> [a] -> [a]
applyToList _ [] = []
applyToList f (n:ns) = (f n):(applyToList f ns)
```

Notice the type signature: `applyToList` takes a function as its first argument, followed by a list, and returns a list. The definition makes clear that `applyToList` forms a new list by applying the argument function `f` to every element of the argument list. Now we can write `applyToList double list`, `applyToList triple list`, and so forth.

7. We don't need to write our own functions to do this. And since we have currying and sections, we don't even have to write the double and triple functions. We can write things like the following.

   ```
   applyToList (2*) [1..10]
   applyToList (3*) [1..10]
   ```

   Try these and see. Write code to generate the remainder of the numbers from 1 to 20 when divided by three.

8. It should not be a surprise that Haskell already has a function like `applyToList`. It is called `map`. Use `:t` to find the type of `map`. Note that it has a slightly different type than `applyToList`. This is because you can `map` a function that produces values of a type different from its argument type. For example, try `map odd [1..10]`.

9. Use `map` to generate a list of the lengths of a list of strings.

10. Haskell has other very useful functions that have functions as arguments (sometimes called *higher order functions*). Two of the most useful are `foldl` and `foldr`. These functions produce a result by applying a binary function to successive elements of a list. For example, suppose you want to add up the elements of a list. You need an accumulator that starts at 0, and then you want to add each element of the list to the accumulating value and return it as the result. Both `foldl` and `foldr` take three arguments: a binary function, a starting accumulator value, and list, and return the result of applying the function to successive elements of the list. The only difference is that `foldl` goes left to right in the list, and `foldr` goes right to left (which matters in some cases). Use a fold function to sum the first 100 numbers. Use one of them to compute 100! (remember to enclose operators in parentheses to refer to them as functions).

11. The functions `foldl1` and `foldr1` don't have a starting accumulator value—they just use the first (or last) value in the list as the starting accumulator value. Use `foldl1` or `foldr1` to find the maximum value in a list of numbers. Use a fold to `&&` together all the values in a list of Booleans.

12. Suppose we want to find the longest string in a list of strings. This is like finding the maximum of a list of numbers except that type of the value is different from the type of the compared value (that is, we are comparing string lengths, not strings). We have to write our own function for this comparison. Write a function `maxString :: [Char] -> [Char] -> [Char]` that return the longest of two strings. Then use a fold function to find the longest string in a list of strings.

13. Now suppose that we want to write a function `maximumString :: [[Char]] -> [Char]` that finds the largest string in a list. We can write this function easily using a fold function and the `maxString` function from above. We can make the `maxString` function local to `maximumString` using either a `where` or a `let` clause. A `let` clause can be used anywhere an expression can be used. It introduces temporary name bindings (like a block in Java or C). For example, we could write `maximumString` using a `let` as follows.
```
maximumString :: [[Char]] -> [Char]
maximumString =
 let
  maxString s t
   | (length s) < (length t) = t
   | otherwise = s
 in foldl maxString ""
```
In this construction, `maxString` is defined first and then used in the expression following the "in". Type this in and try it.

14. Alternatively, consider the following definition.
```
maximumString' :: [[Char]] -> [Char]
maximumString' = foldl maxString ""
 where
  maxString s t
   | (length s) < (length t) = t
   | otherwise = s
```
Here the `where` introduces a definition that applies in the context immediately preceding it. Type this in and try it. Although there are some subtle differences between these two constructs, they can mostly be used as alternatives to one another.

15. Note the indentation in the previous examples. We have not mentioned it before, but indentation is important in Haskell because, like Python, Haskell relies on indentation to determine when expressions end. The "golden rule" of Haskell indentation is that code that is part of an expression must be indented further in than the start of the expression. The second rule of indentation is that all grouped expression must be exactly aligned. To make matters even more difficult, Haskell includes non-whitespace as part of the indentation. To illustrate, the following is ok.
```
let x = 4
    y = 7
```
Here the `x = 4` is indented from the `let` and the `y = 7` is exactly aligned with it. But the following is not ok.
```
let
x = 4
y = 7
```
The expressions are not indented from the `let`. These are also not ok.
```
let x = 4
  y = 7

let x = 4
      y = 7
```
Here the grouped expressions are not exactly aligned. If you only use spaces for indentation, and you align things nicely, you should be ok. (But you must indent `then`

and `else` from the `if` in an `if` expression, which is unlike imperative languages conventions.) I think it is also a good idea to put things like `let`, `where`, `in`, and so on alone on a line. However, if you get a weird compiler message for code that looks perfectly ok, it could be an indentation problem. (Actually, you can use curly braces and semicolons to make everything work, but this is frowned on in the Haskell community.) Be careful of tabs—a tab is counted as a single character in Haskell, so things may not be indented as they appear is you have both spaces and tabs on a line.

16. Write a function `myReverse` that uses a fold to reverse a list. Write the argument function as a local function using a let or where.

17. Yet another higher order function is one that selects or filters a list based on a test function. The built-in filter function has type `filter :: (a -> Bool) -> [a] -> [a]`. The first argument is a test function applied to every element of a list. If this test function returns `True`, then the elements is part of the result list; otherwise it is tossed out. For example, `filter odd [1..10]` returns a list of odd number between 1 and 10.

18. Write a function `partitionLess :: Ord a => a -> [a] -> [a]` that takes a value `v` and a list `l` and returns a list of elements of `l` that are less than or equal to `v`. Now write a function `partitionMore :: Ord a => a -> [a] -> [a]` that return the elements of `l` greater than `v`.

19. Now write quicksort. Now make the partition functions local. Now write quicksort without any helper functions.

20. We have discussed various types, and sometimes they can be quite complicated. Haskell provides a way to abbreviate types for ease of typing and documentation purposes. The statement type T = <def> can be used to define a new type. For example, `type String = [Char]` defines a `String` to be a list of `Chars`. This definition is already built into Haskell, so we could have been doing this all along.

21. Type definitions can also have parameters. For example, `type Pair a = (a,a)` defines `Pair` to be a two-tuple of something. `Pair Int` would then be an abbreviation for `(Int,Int)`, `Pair String` and abbreviation for `(String,String)`, and so on.

22. One last thing: function application has high precedence and it is left associative. This can lead to lots of parentheses, such as `odd (mod 8 (div 9 3))`. The `$` operator is right associative, has lower precedence than function application, but means the same as function application (so `f $ x` means the same as `f x`). Since we have currying, this means we can use it to eliminate some parentheses, as in `odd $ mod 8 $ div 9 3`.