CS 430 Programming Languages
Spring 2016
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*)`. This is called a *point-free definition*. 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, 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 `findMaxString :: [[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 `findMaxString` 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. For example, we could write

findMaxString using a let as follows.

```
findMaxString :: [[Char]] -> [Char]
findMaxString =
 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.

```
findMaxString' :: [[Char]] -> [Char]
findMaxString' = 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.)

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

17. We have discussed various composite types such as lists and tuples. An important and interesting composite type that allows us to create our own data type is the `data` declaration. The types constructed by a data declarations are called *algebraic data types*. The simplest algebraic data types ressemble enumerations.
```
data Color = Blue
           | Green
           | Yellow
           | Orange
           | Red
```
This type has the designated values, as one would expect.

18. But algebraic data types can also have values associated with identifiers as in the following (this makes it clear that algebraic data types are discriminated unions).
```
data Shape = Square Float
           | Circle Float
           | Rectangle Float Float
           | Triangle Float Float deriving Show
```
The identifiers are called *constructors* and behave like functions. For example, `Rectangle 5 3` creates a new `Shape` value. (The deriving Show bit at the end is so we can print out Shapes.) Type this declaration into your lab file, load it in `ghci`, and use `:t` to find the type of `Rectangle 5 3`. Also, find the type of `Rectangle` by itself.

19. We can get values "out of" an algebraic data type using pattern matching. For example, the pattern `(Circle r)` would match an instance of `Shape` created using the `Circle` constructor. Write a function `area :: Shape -> Float` to compute the area of a shape (`pi` is defined in the the Prelude).

20. Sometimes it is useful to be able to refer to an entire value when recognizing which of several alternatives obtains. This can be done with *as-patterns*, which consist of an identifier followed by an `@` followed by the rest of the pattern. For example `c@(Circle _)` will cause `c` to refer to a `Shape` value that is a `Circle`. Write a function `extractCircles` that takes a list of `Shapes` and returns a list of just the `Circles` from the original list.

21. Algebraic data types can also have type parameters. An excellent example of this, and a type useful in its own right, is the predefined type `Maybe`, defined as follows:
```
data Maybe a = Just a | Nothing
```
This type is used when the result of a function might not be defined. For example, `head` and `tail` are predefined in Haskell. Try `head [1..10]` and `tail [1..10]`. Now try `head []` and `tail []`. This is not a good outcome. We can write our own `safeHead :: [a] -> Maybe a` and `safeTail :: [a] -> Maybe [a]` functions that return `Nothing` when given the empty list as a parameter and `Just` whatever when given a non-empty list. Write these functions.

22. The Prelude also has a built-in function `maybe :: b -> (a -> b) -> Maybe a -> b`. This function takes a default value, a function, and a `Maybe` value. If the `Maybe` value is `Nothing`, it return the default value; if not, it extracts the value from the `Just` constructor and applies the function to it. For example, `maybe False odd (Just 5)` is `True` and `maybe False odd Nothing` is `False`. Write a function `addMaybe` that sums the values in a list of `Maybe Integers`, using 0 as

the Nothing value. Hint: use map to convert the list from Maybe Integers to Integers (the id function just returns its argument), and then use a fold function (or sum, if you want) to add them up.

23. How would we write the maybe function? One way would be as follows:

```
myMaybe d _ Nothing = d
myMaybe _ f (Just x) = f x
```

This works (you can try it), but it is hard to read. How about the following alternative?

```
myMaybe' d f m =
 case m of
  Nothing -> d
  (Just n) -> f n
```

This definition uses a case expression. Case expressions use the same pattern matching mechanisms as function definitions. The general form of a case expression is the following.

```
case (x1, x2, … xk) of
 p1 -> exp1
 p2 -> exp2
 …
 pn -> expn
```

Where x1, x2, … xk are variables, p1, p2, … pn are patterns, and exp1, exp2, … expn are expressions. The values of the variables are matched against the patterns from first to last, and the first one that matches determines the expression used as the value of the entire case construct. As in defining functions, it is often a good idea to use otherwise as the last pattern. Rewrite the myMaybe' function above using otherwise.

24. Modules are used to control name spaces, create abstract data types, and hide implementations. Module names must be capitalized. A module can be pulled into a code file for use using the line

```
import M
```

where M is a module. This line must appear before any definitions.
If you only want to import a few things from a module, you can use the line

```
import M (a,b,c, …, k)
```

where a, b, c, … k are names of items in M to be imported.

25. A lot of modules reuse the same names, causing names clashes that must be resolved by explicitly indicating the module where the desired item comes from (using dot notation as in M.item), which is a pain. To avoid this, you can import only the items you need (as above), or you can do a qualified import, as follows.

```
import qualified M
```

Now the items with clashing names can be referred to as before, and all newly imported items must be referred to using their module name. This is also a pain, but we can minimize it by using an abbreviated module name, as follows.

```
import qualified M as B
```

Now, for example, if we import Data.Map.Strict as Map, we can refer to Data.Map.Strict.size as Map.size, which is still a pain, but less of a pain.