CS 430 Programming Languages
Spring 2017
Module 12 Lab

1. We have seen how to do information hiding with local entities using `let` and `where`. We can also control which entities defined in a file can be exported to other files. We can make the code in a file into a nameable unit by making it into a module. Make a file called `MyModule.hs`. Open the file and make the first non-comment, non-blank line `module MyModule where`. Below this in the file make `myLength :: [a] -> Int` and `myHead :: Ord a => [a] -> a` functions.

2. Make a `lab12.hs` file in the same directory as `MyModule.hs` and at the top type `import MyModule`. Write some code that used `myLength` and `myHead` from `MyModule`. In `ghci`, load `lab12.hs`. You should see a message that both `MyModule.hs` and `lab12.hs` are being compiled, and you should be able to run your code from `lab12.hs`. Note that the module name must match the file name.

3. To control what gets exported from a module, we can make a list of exported names. In `MyModule.hs`, change the module declaration to the following.
   ```
   module MyModule
   (myLength)
   where
   ```
   The names between the parentheses are exported (they are separated by commas if there are more than one of them). If you save this change and then try to reload `lab12.hs` in `ghci`, you will get a message that `myHead` is not in scope (because it was not included in the lsti of exported names).

4. You can also control which names exported by a module are imported by listing the desired names in parentheses after the module name in the `import` statement. And there is a lot more control available too: you can specify that imported names can only be referenced as qualified names (such as `MyModule.myLength`), that the qualifier be renamed, and that some imported names be hidden. Also, modules can be put into directories that can be part of the qualified name (as in Java). But we won't need all this fancy stuff.

5. Haskell has built-in functions for converting from string to other values and back, called `read` and `show`. In `ghci`, type `read "8" + 5`. Now try `read "[1,2,3]" :: [Int]`. But try `read "8"` by itself. This does not work because `read` cannot tell the type of thing you are trying to read. If you put a string representation of a Haskell value between double quotes and the context disambiguates the type, then `read` will convert it to a value.

6. `show` does the opposite of `read`. Typing `show 8` in `ghci` does not tell us much because just typing `8` in `ghci` also prints the string version of 8. This is because behind the scenes `ghci` is using `show` to convert values to strings for us. But try this in `ghci`: `"Pair " ++ show (9,2)`. This would not work if `show` was not converting the tuple to a string that could be concatenated to `"Pair "`.

7. The main topic we want to address in this lab is input and output. So far we have been doing everything in the ghci, and our programs have not read any data from anywhere. The interpreter has handled whatever IO was necessary. But clearly Haskell programs need to be able to do more than this. IO is a special sort of thing in Haskell because it cannot be referentially transparent. Remember that an expression is *referentially transparent* if we can execute it anywhere in a program and get the same results; or, in other words, if you give an expression the same arguments, you will always get the same results. This has been true of all the Haskell we have done so far. And this has worked because, at bottom, none of the code we have used so far has had *side-effects*, that is, no non-local resource have been changed. But to do IO we **must** give up referential transparency and accept side-effects: if we write some output, then we have changed the world outside our program (a side-effect), and if we read some input, then we won't get the same value every time (so our expressions won't always have the same values). In Haskell, referentially transparent parts of programs are called "pure" while non-referentially transparent parts are "impure." Haskell programs segregate their pure and impure parts, with the goal being to make the impure parts as small as possible so that most of the program is pure and so can be reasoned about more thoroughly.

8. The type of impure parts of programs is `IO a` where a is some type. A sub-program whose result type is `IO a` for some `a` is by definition impure. Sub-programs with these types are called *actions* rather than functions. Every program that does IO must have a special action called `main :: IO ()` that will be executed when the program is run. Although arbitrary actions can be defined in a Haskell program, none will actually be executed unless they are called (directly or indirectly) from `main`.

9. To get started with actions, add the following line to `lab12.hs`
   ```
   main = putStrLn "Hello"
   ```
   At the command line, type `runhaskell lab12.hs`. The `runhaskell` command compiles, links, and then executes a program. The `putStrLn` action has type `String -> IO ()`; in other words, it is an IO action. It causes its `String` argument to be written to the terminal.

10. We can read from the terminal too. The `getLine :: IO String` action reads a single line of text from the terminal. But it returns it inside an `IO` context, which means it is not immediately accessible. However, we can extract values from IO contexts inside a construct called a *do block* using a special <- operator. Modify your `main` action to look like the following:
    ```
    main = do
      putStrLn "Your name?"
      name <- getLine
      putStrLn ("Hello " ++ name)
    ```
    A do block allows us to stitch together a sequence of IO actions. (Note that the very idea of sequencing is new because order does not matter when you have referential transparency.) The line `name <- getLine` removes the `String` returned by `getLine` from its IO context, so the type of `name` is `String`. Hence we can use `++` on it to form the `String` output in the last line of the do block.

11. Although the code above makes the <- look sort of like an assignment statement, the <- operator is really doing a special job (removing a value from an IO context). Try the following and see what happens.

```
main = do
  putStrLn "Your name?"
  name <- getLine
 response <- "Hello" ++ name
  putStrLn response
```

The problem is that <- expects an IO something on its right, but `"Hello" ++ name` has type `String`. However, we can make regular assignments as in pure code using a `let`, with the added advantage that we don't need the `in` keyword. We can just put `let` and then follow it with one or more assignments; these will be in effect until the end of the do block. Fix the code above so it works by using `let` to assign a value to `response`.

12. We can make IO actions as separate pieces of code and then use them in `main`. We can also use recursion and other Haskell constructs in actions as in pure Haskell functions. For example, Haskell has a built-in action `getContents :: IO String` that reads everything in the standard input stream (stdin) until the end of the stream. We can build an approximation of this action using `getLine`.

13. First lets see what `getContents` does. Replace the `main` action in `lab12.hs` with the following:

```
main = do
  input <- getContents
  putStr input
```

You can probably guess that this program just echoes everything in stdin. The program stops at the end of the input. To test it, make a file called `lab12.txt` with a few lines in it, and run `runhaskell lab12.hs <lab12.txt` at the command line.

14. Now lets write `getTilBlank :: IO String` that reads stdin until it encounters a blank line. Because `getTilBlank` is an action, its body must be a do block. It can read a line from stdin using `getLine` and then check to see whether the result is the empty string `""`. If so, it can simply return the empty string in an IO context. How do we do that? There is a built-in action called `return` that does this (this is NOT like a return in Java or Ruby because it does not affect control flow—it just takes a value and puts it in an IO context). If the input line is not the empty string, then, in a do block, we can apply `return` to the line concatenated to a newline concatenated to the result of getting the rest of the input. How do we get the rest of the input? Call `getTilBlank` again, of course (that is, use recursion). So we can do things in actions and do blocks that we do in regular functions.

15. Finish writing `getTilBlank`. modify `main` to use it, modify `lab12.txt` to have a blank line at the end, and test your code.