# *CSCI 365 Problem Set 6: I/O*

*due Friday, 1 March 2024*

---

## *Specification*

To receive credit for this problem set:

- You must complete at least 8 out of 11 exercises.
- Any code you write must adhere to the Haskell style guide linked from the course web page.

Haskell code should be submitted in one or more `.hs` files. Written exercises may be submitted either as a PDF, or as comments in one of the `.hs` files.

## *Practical I/O*

Note that to test any `IO` action, you can simply enter it at the GHCi prompt. GHCi has a special case to *execute* `IO` actions entered at the prompt rather than just *evaluating* them.

**Exercise 1** Implement `wc :: IO ()` which behaves somewhat like the Unix command-line utility `wc`. It should:

- Prompt the user for a file name
- Print out the number of lines, words, and characters found in the file.

(It is OK for your program to crash if the file does not exist.) For example:

```
Enter a file name: IO.lhs
170  805  5760
```

**Exercise 2** Extend `wc` from Exercise 1 so that (like the real Unix `wc` command) it gets the file name as a command-line argument rather than by prompting the user. To access command-line arguments, you can use the `getArgs` function from the `System.Environment` module.

To test your `wc`, you will need to define `main = wc`, and then run your program from the command line, like so:

```
$ runhaskell PS6.hs somefilename.txt
```

(Note the `$` denotes a command-line prompt and should not be typed.) If you have any trouble getting this to work, just ask!

**Exercise 3** Extend `wc` from Exercise 1 so that it does not crash when given a file that does not exist, but instead prints a suitable error message. Consult the documentation for the `catch` function from the `Control.Exception` module.

(Note that this exercise and Exercise 2 are independent; you may complete either one without the other, or both.)

**Exercise 4** Implement `playGuessNum :: IO ()` to play guess-the-number with a human (with the computer in the role of guesser). In other words, the human should pick a number between 1 and 100 and then the computer should iteratively guess the number, with the human responding "high", "low", or "correct" based on whether each guess is too high, too low, or correct. *Hint*: write a recursive helper function `guess :: Int -> Int -> IO ()` which guesses a number between its two arguments.

## *Utilities*

For each of the exercises in this section, you are **only** allowed to use `return`, `(>>)`, and `(>>=)` (or `do`-notation). In particular you may not use any other standard library functions.

**Exercise 5** Implement `mapResult :: (a -> b) -> IO a -> IO b`.

**Exercise 6** Implement `applyIO :: IO (a -> b) -> IO a -> IO b`.

**Exercise 7** Implement `whenIO :: IO Bool -> IO () -> IO ()` which decides whether to run the second computation or not depending on the result of the first computation.

## *do-notation*

**Exercise 8** Translate the following code to use `do`-notation:

```
foo :: IO ()
foo =
  readFile "foo.txt" >>=
    (\f -> readFile "bar.txt" >>=
      (\b -> writeFile "foobar.txt" (f ++ b) >> putStrLn "Done!"))
```

**Exercise 9** Read the formal definition of `do`-notation in the Haskell 2010 report: `https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-470003.14` Look particularly at the box labelled "Translation", and notice how `do`-notation is defined via *rewrite rules* that show how to incrementally turn `do`-notation into code without it.

In class, we saw that

```
do x <- m
   ms
```

is equivalent to `m >>= \x -> ms`. However, the translation in the report does not look exactly like this. Explain what is going on with the `ok` function defined in the `p <- e` case of the report, and explain why it is equivalent to the version from class in the case that `p` is a variable (*hint*: `p` can actually be any *pattern*).

## *Further Exploration*

**Exercise 10** Skim the introduction and read sections 2, 2.1, and 2.2 of the 1993 paper "Imperative Functional Programming" by Simon Peyton Jones and Phil Wadler (`https://dl.acm.org/doi/pdf/10.1145/158511.158524`).[1]

Write a short (1 or 2 paragraph) response explaining anything interesting you learned (either technical or historical), how it relates to what we have covered in class, *etc*.

[1] Peyton Jones, Simon L., and Philip Wadler. "Imperative functional programming." In Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 71-84. 1993.

**Exercise 11** Look up the documentation for the `unsafePerformIO` (*e.g.* via Hoogle), and explain what is "unsafe" about it.

For a good laugh, you should also look at `https://hackage.haskell.org/package/bytestring-0.12.1.0/docs/Data-ByteString-Internal.html` and scroll down to the section titled "Deprecated and unmentionable".