Assignment 1

due at 23:59 on Wed Sep 13 (80 points)

Software setup

For the first part of this assignment, I'd like you to set up the tools for Haskell programming, both in the lab and on your own machine (laptop or desktop).

Stack and GHC

"Stack" is a build tool and dependency manager for Haskell projects. "GHC" is the Glasgow Haskell Compiler, which Stack can download for you. They work on every popular platform (Windows, Mac, Linux).

Start by following installation instructions here: https://docs.haskellstack.org/en/stable/install_and_upgrade/

For Mac and Linux, that page recommends a `curl` command; you would run that in the Terminal application (in **Applications** » **Utilities** on the Finder menu). For Windows, there's a traditional installation program.

Once installed, you will also use Stack from the Terminal application, called the Command Prompt on Windows. To launch the Windows Command Prompt, open the start menu and type `cmd` – you should see the Command Prompt application appear as the best match. Press enter to open it.

On either platform, verify that Stack is installed correctly by typing `stack` in the terminal, and pressing enter. You should see output similar to what follows:

```
stack - The Haskell Tool Stack

Usage: stack [--help] [--version] [--numeric-version]
             [--hpack-numeric-version] [--docker*] [--nix*]
             ([--verbosity VERBOSITY] | [-v|--verbose] |
              [--silent]) [--[no-]time-in-log]

[...lots more options...]

stack's documentation is available at https://docs.haskellstack.org/
```

Now we need to ask Stack to download the latest version of GHC. Just type this in your terminal:

```
stack setup
```

Once it is successfully installed, you should be able to type:

```
stack ghci
```

and be loaded into the interactive loop. It should look something like this:

```
Configuring GHCi with the following packages:
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from C:\Users\Christopher League\AppData\Local\Temp\ghci5708\ghci-scri
Prelude>
```

To quit GHCi, just type `:q` then enter.

Editor

You will need to use a text editor, ideally one with at least some support for Haskell syntax highlighting. On Windows, a good choice is Notepad++ https://notepad-plus-plus.org/, but here are some others that work on multiple platforms:

- Atom https://atom.io/
- VS Code https://code.visualstudio.com/
- Sublime https://www.sublimetext.com/ – free evaluation, but you're supposed to buy a license ($70) if you continue to use it. I'm not sure this is enforced.

If you experiment with anything else and like it, let us know!

Hello, world!

Now we'll try to compile and run a small "Hello world" program. This should probably be the first program you write with any new language or compiler. And although printing Hello is simple enough in Haskell, we won't use console output (`putStrLn`) much more until we're a few weeks into the course. (Remember, Haskell is a pure language, and console output is a side effect – so the way I/O works is a bit restrictive and unusual.)

Here's how to start. Create a folder somewhere convenient (like Desktop or Documents) – I just called mine `cs695`. **Don't use spaces in your folder or file names** – it makes them really inconvenient to specify from the command line.

Use your editor to type this program – I always suggest typing rather than copy and paste, because you learn the syntax better:

```
main = putStrLn "Hello world"
```

Save that text as a file called `Hello.hs` in your `cs695` folder. **Again, don't use spaces in file names!** Now in your Command Prompt or Terminal, you need to change to the folder where your code is saved:

```
cd Desktop/cs695
```

Once you're there, the simplest way to compile and run your program is:

```
stack runghc Hello.hs
```

You should see your message appear on the console. Congratulations, your first Haskell program!

Now you may want to make a change and run it again. In class, I showed how to sequence together two different print statements, using the `do` keyword:

```
main = do
  putStrLn "Hello grad students"
  putStrLn "Bye again"
```

Try making a similar change and running the program again. **Don't forget to save the file in your editor after making changes!**

Functions

This section serves mainly as rough notes of some of the demonstrations I did in class on 7 September. It also contains instructions for testing your functions interactively.

Let's first define some simple mathematical functions, such as:

$$f(x) = x^2 + 3$$

$$g(x, y) = x - 2y$$

There I'm using math notation, and here is a translation into Haskell:

```
f x = x*x + 3
g x y = x - 2*y
```

Notice that we do not use parentheses or commas to specify functions and arguments. The function `f` applied to `x` is simply written as `f x`.

Interactive testing

To test these functions, we wouldn't use `stack runghc` like we did with the "Hello world" program. They do not do any console output. Instead, we want to load their definitions into the interactive interpreter.

Save them into a `.hs` file (you can just add them to `Hello.hs` if you want), and then in your terminal, run:

```
stack ghci
```

At the `Prelude>` prompt, type `:load Hello.hs`. (Or you can just use `:l` as an abbreviation for `:load`.) Then you can interactively call the functions and inspect the results. Here's a transcript where I did that:

```
C:\Users\Christopher League\Desktop\cs695>stack ghci
Configuring GHCi with the following packages:
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from C:\Users\Christopher League\AppData\Local\Temp\ghci3240\ghci-scri
Prelude> :load Hello.hs
[1 of 1] Compiling Main             ( Hello.hs, interpreted )
Ok, modules loaded: Main.
*Main> f 14
199
*Main> f 21
444
*Main> g 14 21
-28
*Main> g 15 (f 12)
-279
*Main> g (f 15) (f 12)
-66
```

Whenever you make changes to your program, you do *not* have to completely restart GHCi. Instead, just use the `:reload` command (abbreviated `:r`) and it will load the last thing again, and start using the new definitions. (As long as you saved it in your editor!)

Conditions and guards

Some mathematical functions can be defined by specifying multiple cases, where a Boolean expression determines which definition to use. For example, here's a mathematical specification of the function at the heart of the Collatz conjecture.

$$c(n) = \begin{cases} n/2 & \text{if } n \text{ even} \\ 3n+1 & \text{otherwise} \end{cases}$$

We'll look at a few ways to specify that in Haskell. First, we can use the `if-then-else` keywords:

```
collatz1 n =
  if even n then n `div` 2
  else 3*n + 1
```

The function even is built-in, so we don't have to do a module operator like we would in C++/Java. Also note the division operator: we use `div` rather than `/` because we want to stick with integer division; `/` would produce floating-point answers.

The next variation uses two separate *clauses* or cases to specify the function, but adorns one of them with a **guard** – the part between the pipe `|` (which we read as "where") and the equals `=`.

```
collatz2 n | even n = n `div` 2
collatz2 n = 3*n + 1
```

Finally, you can stick to one clause to define the function, but use two different guards. The catch-all one uses the keyword `otherwise`:

```
collatz3 n
  | even n = n `div` 2
  | otherwise = 3*n + 1
```

Try all of these in your GHCi, and ensure you get the expected answers for each one on several different parameters.

Recursion

Recursive definitions are a staple of functional programming – much more common than in typical imperative languages. One of the ways to define the factorial function is recursive:

$$n! = \begin{cases} 1 & \text{if } n \leq 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

Here is a definition in Haskell, using two clauses and a guard:

```
fact n | n <= 0 = 1
fact n = n * fact (n-1)
```

In class, we tried this on numbers large and small. Haskell uses arbitrary-sized integers by default, so it can generate some impressive quantities pretty quickly:

```
*Main> fact 5
120
*Main> fact 10
3628800
*Main> fact 500
1220136825991110068701238785423046926253574342803192842192413588385
8453731538819976054964475022032818630136164771482035841633787220781
7720048078520515932928547790757193933060377296085908627042917454788
2424912726344305670173270769461062802310452644218878789465754777149
8634943677810376442740338273653974713864778784954384895955375379904
2324106127132698432774571554630997720278101456108118837370953101635
6324432987029563896628911658974769572087926928871281780070265174507
7684107196243903943225364226052349458501299185715012487069615681416
2535905669342381300885624924689156412677565448188650659384795177536
0894005745238940335798476363944905313062323749066445048824665075946
7358620746379251842004593696929810222639719525971909452178233317569
3458150855233282076282002340262690789834245171200620771464097945611
6127629145951237229913340169552363850942885592018727433795173014586
3575708283557801587354327688886801203998823847021514676054454076635
3598417443048012893831389688163948746965881750450692636533817505547
8128640000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
```

Exercises

So far, this document has contained setup instructions and notes on using Haskell. Starting in this section, you will do some exercises and turn in your answers. Save your Haskell code for this entire section as `a01.hs` and upload that file to this dropbox.

1. Rewrite the factorial function I defined above to use `if-then-else` instead of the pattern guard. Name it `factIf`. Test that it produces the expected results for some inputs like 5, 6, and 10.

2. Define this mathematical function using Haskell, name it `quadratic`, and test it (correct answers for some inputs are given below).

$$q(a,b,c) = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Note about negative numbers in Haskell: the syntax of the negative sign can be a little confusing; if you're getting errors related to it, you should try surrounding the value with parentheses, such as `(-9)` rather than just `-9`. Also, the square-root function in Haskell is named `sqrt`, and it's built-in (nothing to import).

```
*Main> quadratic 2 9 3
-0.36254139118231254
*Main> quadratic 1 (-1) (-1)
1.618033988749895
```

3. Define a function to calculate the volume of a sphere of a given radius. Name it `sphere`. Below are correct answers for some inputs. To represent π, you should use the built-in constant `pi` (lowercase).

```
*Main> sphere 1
4.1887902047863905
*Main> sphere 3.75
220.8932334555323
```

4. Here is a recursive function in Haskell:

```
slow x y
  | x < y = x
  | otherwise = slow (x-y) y
```

On paper, trace out the steps involved in evaluating `slow 101 21`, the same way we would for an algebraic expression. That is, determine the parameters `x` and `y`, check which case applies, then rewrite it. Repeat until the recursion completes and you get down to a single number.

Then you can transcribe your paperwork into Haskell comment near the function in `a01.hs`. Comments in Haskell are surrounded by `{-` and `-}` (similar to `/*` and `*/` in C++/Java) or single-line comments using `--` (similar to `//` in C++/Java). For example, you can write:

```
{- My evaluation:
   slow 101 21 ==
   _______ ==
   _______ ==
   _______  (fill in the blanks)
 -}
```

5. Read the introduction to the Collatz sequence on Wikipedia and/or watch this Numberphile video about Collatz.

   Then write a recursive Haskell function `collatzCount n` that calculates the number of steps it takes to get from `n` down to `1`. For example, the Wikipedia article states "the sequence for $n = 27$, listed and graphed below, takes 111 steps" so your function output in that case should be:

```
*Main> collatzCount 27
111
```

Your function can begin like this:

```
collatzCount n
  | n <= 1 = 0   -- We're already at one, so zero steps
  | otherwise =  __________ -- fill in the blank
```

and you should feel free to make use of the `collatz1` (or similar) definitions given previously.

Here are a few other correct examples:

```
*Main> collatzCount 99
25
*Main> collatzCount 1024
10
*Main> collatzCount  118
33
```