Homework 1 – Spring 2023

This first homework is meant to get you set up for the first few weeks of class, so you can breathe easy! We'll make sure you have Haskell (GHC/GHCi) and Python 3 properly installed; there are also a few Haskell exercises to help you warm up! Each question has a time estimate; you'll know you're ready for the exam when you can solve them roughly within their time constraints.

We understand, however, that as you learn these questions may take more time. For that reason, only **starred red** questions need to be completed when you submit this homework (the rest should be used as exam prep materials).

For every homework, you must turn in a PDF file with your answers via Gradescope. You may include both typed and hand-written solutions, so long as they are legible and make sense to our TAs. Make sure to clearly label each answer with the problem number you're solving so our TAs can more easily evaluate your work. As a reminder, homework is graded on effort, not correctness – please try every question!

Installing Haskell and GHCi

You can download the Haskell compiler and interpreter, GHC and GHCi, from the following link: https://www.haskell.org/ghc/download.html. On OSX and Linux, you can also use Homebrew: `brew install ghc`

Note in particular that the macOS distribution also requires Xcode (and its command line tools) to be installed. You can download Xcode from the App Store.

You have completed this task once you are able to run ghc/ghci and not get a file-not-found error.

Hello Haskell

Create a new file named *hello.hs*, and place the following expression in it:

```
greeting = "Hello, world!"
```

Now run the Haskell interpreter by typing “ghci” on your command line. You should now be in an interactive environment known as a REPL (read-eval-print loop). Although Haskell is typically compiled, GHCi allows you to write Haskell commands and execute them as if it were an interpreted one.

You can use the `:load` command to load Haskell code in files in the working directory. Run the following command in GHCi:

```
:load hello.hs
```

Now, using the GHCi interpreter, figure out an expression that gives the length of the greeting string. You’ll have completed this task once you can successfully get the length of the greeting string via typing an expression in GHCi.

Installing Python 3

Next, we'll ensure you have a working version of Python 3. Some OS distributions (Linux, macOS) already have Python 3 bundled in. To check, the following commands can be used to locate it:

macOS/Linux:

```
which python3
```

Windows:

```
where python3.exe
```

If Python is already installed, then run it to determine the version. Your version should be 3.9 or greater.

If you don't have Python installed, you can download the latest Python 3 distribution for your OS here: https://www.python.org/downloads/.

Once you have installed Python3 v3.9 or higher and can print "Hello world!", you will have completed this task.

Haskell Warmup

Note: questions #2, #4, and #5 rely on control flow topics (pattern matching, conditionals, guards) that have not been covered in Week 1. We'll cover these on Monday, so don't stress if you can't do them yet; you are of course welcome to read ahead and give them a shot.

You may also find Haskell's List library documentation to be particularly helpful.

For all questions where you are writing Haskell code, you should give your functions a type definition. You'll learn more about type definitions on Monday.

1. ** (2 min.) Write a Haskell function named `largest` that takes in 2 `String` arguments and returns the longer of the two. If they are the same length, return the first argument.

   Example:
   `largest "cat" "banana"` should return `"banana"`.
   `largest "Carey" "rocks"` should return `"Carey"`.

   Solution:

```
largest :: String -> String -> String
largest first second =
    if length first >= length second then first else second
```

2. ** (4 min.) Barry Snatchenberg is an aspiring Haskell programmer. He wrote a function named `reflect` that takes in an `Integer` and returns that same `Integer`, but he wrote it in a very funny way:

```
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
  | num < 0 = (-1) + reflect num+1
  | num > 0 = 1 + reflect num-1
```

He finds that when he runs his code, it always causes a stack overflow (infinite recursion) for any non-zero argument! What is wrong with Barry's code (i.e. can you fix it so that it works properly)?

Barry is missing parentheses around num+1 and num-1. In its current state, the program is functionally equivalent to:

```
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
  | num < 0 = (-1) + (reflect num) + 1
  | num > 0 = 1 + (reflect num) - 1
```

which causes an infinite loop.

To fix the program, we need only to add some parentheses to fix the associativity:

```
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
  | num < 0 = (-1) + reflect (num+1)
  | num > 0 = 1 + reflect (num-1)
```

3. **

a) (2 min.) Write a Haskell function named `all_factors` that takes in an `Integer` argument and returns a list containing, in ascending order, all factors of that integer. You may assume that the argument is always positive. **Your function's implementation should be a single, one-line list comprehension.**

Example:

`all_factors 1` should return `[1]`.

`all_factors 42` should return `[1, 2, 3, 6, 7, 14, 21, 42]`.

Solution:

```haskell
all_factors :: Integer -> [Integer]
all_factors num =
  [x | x <- [1..num], num `mod` x == 0]
```

b) (3 min.) A perfect number is defined as a positive integer that is equal to the sum of its proper divisors (where "proper divisors" refers to all of its positive whole number factors, excluding itself). For example, 6 is a perfect number because its proper divisors are 1, 2 and 3 and 1 + 2 + 3 = 6.

Using the `all_factors` function, write a Haskell expression named `perfect_numbers` whose value is a **list comprehension** that generates an infinite list of all perfect numbers (even though it has not been proved yet whether there are infinitely many perfect numbers 😉).

Example:
`take 4 perfect_numbers` should return `[6, 28, 496, 8128]`.

Hint: You may find the init and sum functions useful.

Solution:

```
perfect_numbers :: [Integer]
perfect_numbers =
  [x | x <- [1..], sum (init (all_factors x)) == x]
```

4. ** (4 min.) Write a pair of Haskell functions named `is_odd` and `is_even` that each take in 1 `Integer` argument and return a `Bool` indicating whether the integer is odd or even respectively. You may assume that the argument is always positive.

   You may not use any builtin arithmetic, bitwise or comparison operators (including `mod`, `rem` and `div`). You may only use the addition and subtraction operators (+ and -) and the equality operator (==).

   You must implement THREE versions of these functions: (1) with regular if statements, (2) using guards, and (3) using pattern matching.

   Example:
   `is_even 8` should return `True`.
   `is_odd 8` should return `False`.

   Hint: The functions can call one another in their implementations. (This is called mutual recursion).

   Solutions:

```haskell
-- with if statements
is_odd :: Integer -> Bool
is_odd x =
 if x == 0 then False else is_even (x-1)

is_even :: Integer -> Bool
is_even x =
 if x == 0 then True else is_odd (x-1)
```

```haskell
-- with guards
is_odd :: Integer -> Bool
is_odd x
 | x == 0 = False
 | otherwise = is_even (x-1)

is_even :: Integer -> Bool
is_even x
 | x == 0 = True
 | otherwise = is_odd (x-1)
```

```haskell
-- with pattern matching
is_odd :: Integer -> Bool
is_odd 0 = False
is_odd x = is_even (x-1)

is_even :: Integer -> Bool
is_even 0 = True
is_even x = is_odd (x-1)
```

5. (6 min.) Write a function named `count_occurrences` that returns the number of ways that all elements of list a_1 appear in list a_2 in the same order (though a_1's items need not necessarily be consecutive in a_2). The empty sequence appears in another sequence of length `n` in `1` way, even if `n` is `0`.

   Examples:

   `count_occurrences [10, 20, 40] [10, 50, 40, 20, 50, 40, 30]` should return 1.

   `count_occurrences [10, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 2.

   `count_occurrences [20, 10, 40] [10, 50, 40, 20, 50, 40, 30]` should return `0`.

   `count_occurrences [50, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 3.

   `count_occurrences [] [10, 50, 40, 20, 50, 40, 30]` should return 1.

   `count_occurrences [] []` should return 1.

   `count_occurrences [5] []` should return `0`.

   Solution:

```haskell
count_occurrences :: [Integer] -> [Integer] -> Integer
count_occurrences [] _ = 1
count_occurrences _ [] = 0
count_occurrences (x:xs) (y:ys)
  | x == y = count_occurrences xs ys + other_occurrences
  | otherwise = other_occurrences
  where other_occurrences = count_occurrences (x:xs) ys
```