Homework 3

This homework covers the following topics from class:

- Functional Programming, part 3
    - Higher-order Functions, Map/filter/reduce, Lambdas/Closures, Partial Function Application
- Functional Programming, part 4
    - Currying, **Algebraic Data Types, Immutable Data Structures***

*** We covered the first part of algebraic data types and will finish on Monday. You should have enough knowledge to attempt the last two problems that cover this topic with the help of slides and online resources like ChatGPT.**

Each problem below has a time associated with it. This is the amount of time that we expect a thoroughly-prepared student would take to solve this problem on an exam. We understand, however, that as you learn these concepts, these questions may take more time to solve, and we don't want to overwhelm you with homework. For that reason, only **required** questions need to be completed when you submit this homework. That said, the exams will cover all materials covered (a) in class, (b) from class projects, (c) in the required homework problems, and (d) in the optional homework problems, so at a minimum, please review the optional problems and/or use them as exam prep materials.

For questions where you are writing Python code, you **do not** need to perform type-checking of parameters unless explicitly instructed to.

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.

Also, for all Haskell functions, please make sure to include a type signature at the top of the function!

For homework #3, you must complete the following **required** problems:

- HASK10: Map, Filter, Reduce (6 min)
- HASK11: First-class Functions, Higher-order Functions, Recursion (11 min)
- HASK12: Variable Capture (7 min)
- HASK13: Closures (5 min)
- HASK14: Currying, Partial Function Application (11 min)
- HASK15: Algebraic Data Types (16 min)
- HASK16: Algebraic Data Types, Immutable Data Structures (13 min)

For homework #3, you may optionally complete the following additional problems:

- N/A

HASK10: Map, Filter, Reduce (6 min)

Part A: (2 min.) Use the `map` function to write a Haskell function named `scale_nums` that takes in a list of `Integers` and an `Integer` named `factor`. It should return a new list where every number in the input list has been multiplied by `factor`.

Example:
`scale_nums [1, 4, 9, 10] 3` should return `[3, 12, 27, 30]`.

You may not define any nested functions. Your solution should be a single, one-line map expression that includes a lambda.

Your answer:

Part B: (2 min.) Use the `filter` and `all` functions to write a Haskell function named `only_odds` that takes in a list of `Integer` lists, and returns all lists in the input list that only contain odd numbers (in the same order as they appear in the input list). Note that the empty list vacuously satisfies this requirement.

Example:
`only_odds [[1, 2, 3], [3, 5], [], [8, 10], [11]]` should return `[[3, 5], [], [11]]`.

You may not define any nested functions. Your solution should be a single, one-line `filter` expression that includes a lambda.

Your answer:

Part C: (2 min.) In Homework 1, you wrote a `largest` function that returns the larger of two words, or the first if they are the same length:

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

Use one of `foldl` or `foldr` and the `largest` function to write a Haskell function named `largest_in_list` that takes in a list of `String`s and returns the longest `String` in the list. If the list is empty, return the empty string. If there are multiple strings with the same maximum length, return the one that appears first in the list. **Do not use the `map`, `filter` or `maximum` functions in your answer.**

Your answer should be a single, one-line `fold` expression.

Example:

`largest_in_list ["how", "now", "brown", "cow"]` should return `"brown"`.

`largest_in_list ["cat", "mat", "bat"]` should return `"cat"`.

Your answer:

HASK11: First-class Functions, Higher-order Functions, Recursion (11 min)

Part A: (5 min.) Write a Haskell function named `count_if` that takes in a predicate function of type `(a -> Bool)` and a list of type `[a]`. It should return an `Int` representing the number of elements in the list that satisfy the predicate. **Your solution must use recursion. Do not use the `map`, `filter`, `foldl`, or `foldr` functions in your solution.**

Examples:

`count_if (\x -> mod x 2 == 0) [2, 4, 6, 8, 9]` should return `4`.

`count_if (\x -> length x > 2) [“a”, “ab”, “abc”]` should return `1`.

Your answer:

Part B: (3 min.) Now, reimplement the same function above (call it `count_if_with_filter`), **but use the `filter` function in your solution**.

Your answer:

Part C: (3 min.) Now, reimplement the same function above (call it `count_if_with_fold`) **but use either `foldl` or `foldr` in your solution**.

Your answer:

HASK12: Variable Capture (7 min)

Consider the following Haskell function:

```
f a b =
  let c = \a -> a   -- (1)
      d = \c -> b   -- (2)
  in \e f -> c d e  -- (3)
```

Part A: (1 min.) What variables (if any) are captured in the lambda labeled (1)?

Your answer:

Part B: (1 min.) What variables (if any) are captured in the lambda labeled (2)?

Your answer:

Part C: (1 min.) What variables (if any) are captured in the lambda labeled (3)?

Your answer:

Part D: (4 min.) Suppose we invoke `f` in the following way:

```
f 4 5 6 7
```

What are the names of the variables that the passed in values (4, 5, 6, and 7) are bound to? Which of the values/variables (if any) are actually referenced in the implementation of `f`? Explain.

Your answer:

HASK13: Closures (5 min)

(5 min.) C allows you to point to functions, as in the following code snippet:

```c
int add(int a, int b) {
  return a + b;
}

int main() {
  // Declare a pointer named `fptr` that points to a function
  // that takes in two int arguments and returns another int
  int (*fptr)(int, int);

  // Assign the address of the `add` function to `fptr`
  fptr = &add;

  // Invoke the function using `fptr`. This returns 8.
  (*fptr)(3, 5);
}
```

Function pointers are first-class citizens like any other pointer in C: they can be passed as arguments, used as return values, and assigned to variables. As a reminder, however, C does not support nested functions.

Compare function pointers in C with closures in Haskell. Are Haskell closures also first-class citizens? What (if any) capabilities do function pointers have that closures do not (and vice versa)?

Your answer:

HASK14: Currying, Partial Function Application (11 min)

Part A: (3 min.) Explain the difference between currying and partial application.

Your answer:

Part B: (4 min.) Suppose we have a Haskell function with type `a -> b -> c`. Consider the other two function types:

i. `(a -> b) -> c`
ii. `a -> (b -> c)`

Is `a -> b -> c` equivalent to i, ii, both, or neither? Why?

Your answer:

Part C: (2 min.) Consider the following Haskell function:

```
foo :: Integer -> Integer -> Integer -> (Integer -> a) -> [a]
foo x y z t = map t [x,x+z..y]
```

Rewrite the implementation of `foo` as a chain of lambda expressions that each take in **one** variable to demonstrate the form of a curried function.

Your answer:

Part D: (2 min.) Given the function in part C, assuming we call it like this:

bar = foo 1 2 3

give the type signature for the partially-applied function referred to by bar.

Your answer:

HASK15: Algebraic Data Types (16 min)

Haskell allows you to define your own custom data types. In this question, you'll look at code examples and use them to write your own (that is distinct from the ones shown).

Consider the following code example:

```
data Triforce = Power | Courage | Wisdom

wielder :: Triforce -> String
wielder Power = "Ganon"
wielder Courage = "Link"
wielder Wisdom = "Zelda"

princess = wielder Wisdom
```

Part A: (2 min.) Define a new Haskell type `InstagramUser` that has two value constructors (without parameters) - `Influencer` and `Normie`.

Your answer:

Part B: (2 min.) Write a function named `lit_collab` that takes in two `InstagramUsers` and returns `True` if they are both `Influencers` and `False` otherwise.

Your answer:

Consider the following code example:

```
data Character = Hylian Int | Goron | Rito Double | Gerudo |
Zora

describe :: Character -> String
describe (Hylian age) = "A Hylian of age " ++ show age
describe Goron = "A Goron miner"
describe (Rito wingspan) = "A Rito with a wingspan of " ++ show
wingspan ++ "m"
describe Gerudo = "A mighty Gerudo warrior"
describe Zora = "A Zora fisher"
```

Part C: (2 min.) Modify your `InstagramUser` type so that the `Influencer` value constructor takes in a list of Strings representing their sponsorships.

Your answer:

Part D: (3 min.) Write a function `is_sponsor` that takes in an `InstagramUser` and a `String` representing a sponsor, then returns `True` if the user is sponsored by `sponsor` (this function always returns `False` for `Normies`).

Your answer:

For parts e-g, consider the following code example:

```
data Quest = Subquest Quest | FinalBoss

count_subquests :: Quest -> Integer
count_subquests FinalBoss = 0
count_subquests (Subquest quest) = 1 + count_subquests quest
```

Part E: (2 min.) Modify your `InstagramUser` type so that the `Influencer` value constructor also takes in a list of other `InstagramUsers` representing their followers (after their sponsors).

Your answer:

Part F: (3 min.) Write a function `count_influencers` that takes in an `InstagramUser` and returns an `Integer` representing the number of `Influencers` that are following that user (this function always returns `0` for `Normies`).

Your answer:

Part G: (2 min.) Use GHCi to determine the type of `Influencer` using the command `:t Influencer`. What can you infer about the type of custom value constructors?

Your answer:

HASK16: Algebraic Data Types, Immutable Data Structures (13 min)

Consider the following Haskell data type:

```
data LinkedList = EmptyList | ListNode Integer LinkedList
  deriving Show
```

Part A: (3 min.) Write a function named `ll_contains` that takes in a `LinkedList` and an `Integer` and returns a `Bool` indicating whether or not the list contains that value.

Examples:

`ll_contains (ListNode 3 (ListNode 6 EmptyList)) 3`

should return `True`.

`ll_contains (ListNode 3 (ListNode 6 EmptyList)) 4`

should return `False`.

Your answer:

Part B: (3 min.) We want to write a function named `ll_insert` that inserts a value at a given zero-based index into an existing `LinkedList`. Provide a type definition for this function, explaining what each parameter is and justifying the return type you chose.

Your answer:

Part C: (5 min.) Implement the `ll_insert` function. If the insertion index is 0 or negative, insert the value at the beginning. If it exceeds the length of the list, insert the value at the end. Otherwise, the value should have the passed-in insertion index after the function is invoked.

Your answer:

Part D: (2 min.) If there are N nodes in a list and you are asked to insert a new value into position P, how many new nodes must be created in order to insert the value? How many nodes, if any, can be reused from the original list?

Your answer: