Homework 2 – Fall 2022 (due: April 19, 2023)

This homework is meant to prepare you for upcoming exams. You will be expected to be familiar with the concepts covered in this homework and be able to write grammatically-correct Haskell code. 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.

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.

For all questions where you are writing Haskell code, you should give your functions a type definition.

1.

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.

Solution:

```haskell
scale_nums :: [Integer] -> Integer -> [Integer]
scale_nums xs factor = map (\x -> x * factor) xs
```

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.

Solution:

```haskell
-- note the partial application with all
only_odds :: [[Integer]] -> [[Integer]]
only_odds xs = filter (all (\x -> mod x 2 /= 0)) xs
```

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:

```haskell
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"`.

Solution:

```haskell
largest_in_list :: [String] -> String
largest_in_list xs = foldl largest "" xs
```

2.

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.`

Solution:

```
count_if :: (a -> Bool) -> [a] -> Int
count_if predicate [] = 0
count_if predicate (x:xs)
    | (predicate x) = 1 + count_if predicate xs
    | otherwise = count_if predicate xs
```

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

Solution:

```
count_if_with_filter :: (a -> Bool) -> [a] -> Int
count_if_with_filter predicate xs = length (filter predicate xs)
```

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

Solution:

```
count_if_with_fold :: (a -> Bool) -> [a] -> Int
count_if_with_fold predicate xs =
  let count acc x = if predicate x then acc + 1 else acc
  in foldl count 0 xs
```

3.

a) (3 min.) Explain the difference between currying and partial application.

Currying is the process of taking a function of `n` arguments and equivalently transforming it into a chain of functions that each only take one argument. Partial application, instead, refers to the process of passing `k` arguments, where `0 < k < n`, to a curried function that takes `n` arguments. This yields another function that accepts `n-k` arguments.

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?

It is equivalent to ii. The key is to recognize that Haskell automatically curries functions; if `f :: a -> b -> c`, then invoking `f` with only one argument will yield another function of type `b -> c` (partial application).

Therefore, `a -> b -> c` is not equivalent to i because i describes a function that accepts a single argument (another function of type `a -> b`), whereas `a -> b -> c` accepts two arguments. However, it is equivalent to ii because ii is simply the curried form of `a -> b -> c` and, as previously discussed, Haskell performs currying automatically.

c) (2 min.) Consider the following Haskell function:

```haskell
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.

Solution:

```haskell
foo = \x -> \y -> \z -> \t -> map t [x,x+z..y]
```

4. Consider the following Haskell function:

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

a) (1 min.) What variables (if any) are captured in the lambda labeled `(1)`?

There are no captured variables. The parameter to the lambda (a in `\a`) "shadows" (hides) the outer a parameter that's passed into `f`. So the outer a is not captured. The line is effectively equivalent to "c = \x -> x" (or any other name for the bound variable).

b) (1 min.) What variables (if any) are captured in the lambda labeled `(2)`?

b is captured (the second parameter of `f`). c is not captured; the line is effectively equivalent to "d = \y -> b".

c) (1 min.) What variables (if any) are captured in the lambda labeled `(3)`?

c and d are captured (from the `let` declarations). We can understand this as c,d being replaceable with the implementations defined by the `let` declarations. Note that the f in this line has nothing to do with the name of the function being f.

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

```haskell
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.

This answer is as instructive as we can make it - this level of detail will not be required on the exam.

After `f` is invoked with two arguments, the lambda function `(\e f -> c d e)` is returned, which accepts another two arguments. So, 4 is bound to a, 5 is bound to b, and then the values of 6 and 7 are passed to the returned lambda. 6 is then bound to e, and 7 is bound to `f`. Only the variables bound to 5 and 6 are actually referenced. In lambda `(1)`, the a in the return expression refers to the lambda parameter a, not the function parameter bound to 4. Lambda `(2)` uses the captured variable b, the variable bound to 5. Lambda `(3)` uses e, the variable bound to 6, but not `f`, the one bound to 7.

What's going on with `\e f -> c d e`? Both c and d are lambda functions, so are we passing d as the parameter to c? Yes! Function c (`c = \a -> a`) takes one parameter, a, and returns the value of that same parameter. In this case, the argument to c is the function d, in the lambda `\e f -> c d e`. So, the call `c d e` passes d as an argument to c, which just re-returns d. This results in the remaining expression of `d e`. Haskell then calls lambda d (`d = \c -> b`) passing e as the argument. The d function finally just returns the captured value of b, which is 5 (without using the value of e at all!). This is a tricky question, but hopefully it got your gears turning.

Alternative explanation:

Calling "f 4 5 6 7" binds in order: a=4, b=5, e=5, f=7. Replacing each line with what we worked out from parts a-c, we have:

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

Evidently, a,f are unused since they are not referenced in the function body.

c is the identity function, d is a function that returns 5 (the value of b) for all inputs.

Evaluating line 3 gives us that we don't care about the 2nd parameter of the lambda and that (c d) = d, \e -> d e =5.

Thus, b and e are referenced but only b actually affects the outcome of the function.

5. (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)?

Closures are also first-class citizens in Haskell. Lambda expressions and nested functions that capture variables can be passed, returned, and assigned. However, Haskell closures can capture local variables (and extend the lifetime of the captured variables in memory) whereas C function pointers cannot. Function pointers are merely addresses to executable code; they have no notion of an environment. As such, you cannot capture variables allocated on the stack - which include function parameters and local variables (but you can in Haskell, due to its use of garbage collection - it keeps captured variables around as long as necessary).

6. 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:

```haskell
data Triforce = Power | Courage | Wisdom

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

```haskell
princess = wielder Wisdom
```

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

Solution:

```haskell
data InstagramUser = Influencer | Normie
```

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.

Solution:

```haskell
lit_collab :: InstagramUser -> InstagramUser -> Bool
lit_collab Influencer Influencer = True
lit_collab _ _ = False
```

Consider the following code example:

```haskell
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"
```

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

Solution:

```haskell
data InstagramUser = Influencer [String] | Normie
```

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`).

Solution:

```haskell
is_sponsor :: InstagramUser -> String -> Bool
is_sponsor Normie _ = False
is_sponsor (Influencer sponsors) sponsor =
  sponsor `elem` sponsors
```

Consider the following code example:

```haskell
data Quest = Subquest Quest | FinalBoss

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

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).

Solution:

```haskell
data InstagramUser = Influencer [String] [InstagramUser] | Normie
```

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`).

Solutions:

```
-- foldl-based solution
count_influencers :: InstagramUser -> Integer
count_influencers Normie = 0
count_influencers (Influencer _ followers) =
  let count acc (Influencer _ _) = acc + 1
      count acc Normie = acc
  in foldl count 0 followers
```

```
-- hand-coded recursive solution
count_influencers:: InstagramUser -> Integer
count_influencers Normie = 0
count_influencers (Influencer _ followers) =
  let count ((Influencer s f):xs) = 1 + count xs
      count (_:xs) = count xs
      count [] = 0
  in count followers
```

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?

Value constructors are just functions that return an instance of the custom data type!

7. Consider the following Haskell data type:

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

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`.

Solution:

```
ll_contains :: LinkedList -> Integer -> Bool
ll_contains EmptyList _ = False
ll_contains (ListNode x next) y =
  if x == y then True else ll_contains next y
```

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.

Solution:

```
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
```

Given the definition of the `LinkedList` data type, there is no way for us to modify a `LinkedList` passed to `ll_insert`. Therefore, what makes most sense is for `ll_insert` to return a new `LinkedList` where the value has been

inserted at the desired index. The first parameter of our function will be the `LinkedList` to insert into, the second the value, and the third the index.

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.

```
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
ll_insert EmptyList x _ = ListNode x EmptyList
ll_insert (ListNode y rest) x index
  | index <= 0 = ListNode x (ListNode y rest)
  | otherwise = ListNode y (ll_insert rest x (index-1))
```

8. In this question, we'll examine how the choice of programming language/paradigm can affect the difficulty of a given task.

a) (5 min.) Using C++, write a function named `longestRun` that takes in a `vector` of booleans and returns the length of the longest consecutive sequence of `true` values in that vector.

Examples:
Given `{true, true, false, true, true, true, false}`, `longestRun(vec)` should return `3.`
Given `{true, false, true, true}`, `longestRun(vec)` should return `2.`

Solution:

```
int longestRun(vector<bool> vec) {
  int currentMax = 0;
  int currentRun = 0;
  for (bool element : vec) {
    if (element) currentRun++;
    else {
      currentMax = max(currentMax, currentRun);
      currentRun = 0;
    }
  }
  return max(currentMax, currentRun);
}
```

b) (10 min.) Using Haskell, write a function named `longest_run` that takes in a list of `Bool`s and returns the length of the longest consecutive sequence of `True` values in that list.

Examples:
`longest_run [True, True, False, True, True, True, False]` should return `3`.
`longest_run [True, False, True, True]` should return `2`.

Solution using helper parameters:

```haskell
longest_run :: [Bool] -> Int
longest_run xs =
  let
    longest_run_helper [] current_run current_max =
      max current_run current_max
    longest_run_helper (x:xs) current_run current_max =
      if x
        then longest_run_helper xs (current_run+1) current_max
        else longest_run_helper xs 0 (max current_run current_max)
  in
    longest_run_helper xs 0 0
```

Fancy solutions using scanl courtesy of Timothy Gu:

```
longest_run :: [Bool] -> Int
longest_run l = maximum (scanl (\p x -> if x then p + 1 else 0) 0 l)
```

```
longest_run :: [Bool] -> Int
longest_run = maximum . scanl (\p x -> if x then p + 1 else 0) 0
```

c) (10 min.) Consider the following C++ class:

```
#import <vector>
using namespace std;

class Tree {
public:
  unsigned value;
  vector<Tree *> children;

  Tree(unsigned value, vector<Tree *> children) {
    this->value = value;
    this->children = children;
  }
};
```

Using C++, write a function named `maxTreeValue` that takes in a `Tree` pointer `root` and returns the largest value within the tree. If `root` is `nullptr`, return `0`.

This function may not contain any recursive calls.

Solution:

```
#import <queue>

unsigned maxValue(Tree *root) {
  unsigned currentMax = 0;
  queue<Tree *> nodesToExplore;
  nodesToExplore.push(root);

  while (!nodesToExplore.empty()) {
    Tree *current = nodesToExplore.front();
    nodesToExplore.pop();
    if (!current) continue;

    unsigned value = current->value;
    vector<Tree *> children = current->children;

    if (value > currentMax)
```

```
      currentMax = value;
    for (Tree *child : children)
      nodesToExplore.push(child);
  }

  return currentMax;
}
```

d) (5 min.) Consider the following Haskell data type:

```
data Tree = Empty | Node Integer [Tree]
```

Using Haskell, write a function named `max_tree_value` that takes in a `Tree` and returns the largest `Integer` in the `Tree`. Assume that all values in the tree are non-negative. If the root is `Empty`, return `0`.

Example:
`max_tree_value (Node 3 [(Node 2 [Node 7 []]), (Node 5 [Node 4 []])])` should return 7.

Solutions:

```
-- hand-coded recursive solution
max_tree_value :: Tree -> Integer
```

```haskell
max_tree_value Empty = 0
max_tree_value (Node val []) = val
max_tree_value (Node val lst) = max val (max_aux lst)
  where
    max_aux [] = 0
    max_aux (x:xs) = max (max_tree_value x) (max_aux xs)
```

```haskell
-- map-based solution
max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node value children) =
  case children of
    [] -> value
    xs -> max value (maximum (map max_tree_value xs))
```

9. (10 min.) Write a Haskell function named `fibonacci` that takes in an `Int` argument n. It should return the first n numbers of the Fibonacci sequence (for this problem, we'll say that the first two numbers of the sequence are 1, 1).

   Examples:

   `fibonacci 10` should return `[1,1,2,3,5,8,13,21,34,55]`.

   `fibonacci -1` should return `[]`.

   Hint: You may find it easier to build the list in reverse in a right-to-left manner, then use the reverse function.

   Solutions:

```haskell
-- solution where we build the list in reverse
```

```haskell
fibonacci :: Int -> [Integer]
fibonacci n =
  let fib_rev 1 = [1]
      fib_rev 2 = [1, 1]
      fib_rev n =
        let prev_fib_rev = fib_rev (n-1)
            first = head prev_fib_rev
            second = head (tail prev_fib_rev)
        in (first + second) : prev_fib_rev
  in reverse (fib_rev n)
```

```haskell
-- less-efficient solution building the list forward
-- second_last is O(n)
fibonacci 1 = [1]
fibonacci 2 = [1, 1]
fibonacci n =
    let second_last xs
          | length xs == 2 = head xs
          | otherwise = second_last (tail xs)
        prev_fib = fibonacci (n-1)
    in prev_fib ++ [last prev_fib + second_last prev_fib]
```

```haskell
-- fancy solution using list comprehension that generates an
-- infinite list.
-- this works because of Haskell’s lazy evaluation.
fibonacci n =
  let fib = 1 : 1 : [a+b | (a,b) <- zip fib (tail fib)]
  in take n fib
```

10. (20 min.) Super Giuseppe is a hero trying to save Princess Watermelon from the clutches of the evil villain Oogway. He has a long journey ahead of him, which is comprised of many events:

```
data Event = Travel Integer | Fight Integer | Heal Integer
```

Super Giuseppe begins his adventure with 100 hit points, and may never exceed that amount. When he encounters:

A `Travel` event, the `Integer` represents the distance he needs to travel. During this time, he heals for ¼ of the distance traveled (floor division).

A `Fight` event, the `Integer` represents the amount of hit points he loses from the fight.

A `Heal` event, the `Integer` represents the amount of hit points he heals after consuming his favorite power-up, the bittermelon.

If Super Giuseppe has 40 or fewer life points after an Event, he enters defensive mode. While in this mode, he takes half the damage from fights (floor division), but he no longer heals while traveling. Nothing changes with Heal events. Once he heals **above** the 40 point threshold following an Event, he returns to his normal mode (as described above).

Write a Haskell function named `super_giuseppe` that takes in a list of `Events` that comprise Super Giuseppe's journey, and returns the number of hit points that he has at the end of it. If his hit points hit 0 or below at any point, then he has unfortunately Game Over-ed and the function should return `-1`.

Examples:
`super_giuseppe [Heal 20, Fight 20, Travel 40, Fight 60, Travel 80, Heal 30, Fight 40, Fight 20]` should return `10`.
`super_giuseppe [Heal 40, Fight 70, Travel 100, Fight 60, Heal 40]` should return `-1`.

Solutions:

```haskell
-- applying each event recursively
super_giuseppe xs =
  let normal_step (Travel distance) hp =
        min 100 (hp + distance `div` 4)
      normal_step (Fight loss) hp = hp - loss
      normal_step (Heal amount) hp = min 100 (hp + amount)
      defensive_step (Travel _) hp = hp
      defensive_step (Fight loss) hp = hp - (loss `div` 2)
      defensive_step (Heal amount) hp = min 100 (hp + amount)
      stepper [] hp = hp
      stepper (x:xs) hp =
        let new_hp = if hp <= 40 then defensive_step x hp
                                 else normal_step x hp
        in if new_hp <= 0 then (-1) else stepper xs new_hp
  in stepper xs 100
```

```haskell
-- using mutual recursion
super_giuseppe xs =
 let normal_mode [] hp = hp
     normal_mode ((Travel distance):xs) hp =
     normal_mode xs (min (hp + (distance `div` 4)) 100)
     normal_mode ((Fight loss):xs) hp
       | new_hp <= 0 = (-1)
       | new_hp <= 40 = defensive_mode xs new_hp
       | otherwise = normal_mode xs new_hp
       where new_hp = hp - loss
     normal_mode ((Heal amount):xs) hp =
       normal_mode xs (min (hp + amount) 100)
     defensive_mode [] hp = hp
     defensive_mode ((Travel _):xs) hp = defensive_mode xs hp
     defensive_mode ((Fight loss):xs) hp =
       let new_hp = hp - (loss `div` 2)
       in if new_hp <= 0 then (-1) else defensive_mode xs new_hp
     defensive_mode ((Heal amount):xs) hp =
       let new_hp = min (hp + amount) 100
       in if new_hp > 40 then normal_mode xs new_hp
                         else defensive_mode xs new_hp
 in normal_mode xs 100
```