Homework 3 Solutions

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

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.

Solution:

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

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.

Solution:

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

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"`.

Solution:

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

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`.

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
```

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

Solution:

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

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**.

Solution:

```haskell
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
```

HASK12: Variable Capture (7 min)

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)
```

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

Solution:

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).

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

Solution:

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

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

Solution:

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.

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.

Solution:

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.

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)?

Solution:

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).

HASK14: Currying, Partial Function Application (11 min)

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

Solution:

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.

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?

Solution:

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.

Part 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:

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

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.

Solution:

bar :: (Integer -> a) -> [a]

Why?

The original foo function takes four arguments: three Integer values (x, y, z) and a function (t) of type Integer -> a.

The call bar = foo 1 2 3 partially applies the first three arguments (1, 2, and 3) to foo, leaving the last argument (t) to be provided.

Therefore, bar is a function that takes a function t (of type Integer -> a) and returns a list of type [a].

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`.

Solution:

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

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.

Solution:

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

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.

Solution:

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

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`).

Solution:

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

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).

Solution:

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

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`).

Solutions:

```haskell
-- 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
```

```haskell
-- 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
```

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?

Solution:

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

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

Consider the following Haskell data type:

```haskell
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`.

Solution:

```haskell
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
```

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.

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.

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.

Solution:

```
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))
```

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?

Solution:

Given the immutability of Haskell's linked lists:

- **Nodes Created: If you need to insert a new value at position P in a list of N nodes, you must create P+1 new nodes. This is because the list must be reconstructed from the beginning up to the point where the new value is inserted. Each of these nodes will reference the original sub-list starting from the next node.**
  For example, if you insert a new node at position 2 in a list, first we'll create a new version of node 1, and point it at the new node 2. Then we'll have to create a new version of node 0, and point it at the new node 1. Thus, the nodes at positions 0 and 1 must be newly created, along with the new node at position 2. The nodes after position 2 are reused from the original list.
- **Nodes Reused: N-P nodes can be reused from the original list, where P is the position of the insertion. This is because the list nodes after the insertion point do not need to be modified and can be shared between the old and new lists.**

Example:

Consider inserting a new value into position P=2 in a list of N=5 nodes:

- **The first 2 nodes (positions 0 and 1) need to be recreated because the new list will reference them as part of its structure.**
- **The new node for the inserted value will also be created.**
- **The last 3 nodes (positions 3 to 5) can be directly reused in the new list.**

Thus, you end up creating 2+1=3 new nodes and reusing 5−2=3 nodes from the original list.