CS131 Spring 23 - Midterm #1
version 1 rubric

1. [10 points-2 each] For each of the items below, write a Haskell type signature for the expression. If the item is a function then use the standard haskell type notation for a function. If the item is an expression, write the type of the expression (e.g., [Int]). For any answers that use one or more type variables *where concrete types can't be inferred*, use "t1", "t2", etc. for your type variable names.

```
a. foo bar = map (\x -> 2 * x) bar
b. z = \x -> x ++ ["carey"]
c. m = [((\a -> length a < 7) x,take 3 y) |
        x <- ["matt","ashwin","siddarth","ruining"],
        y <- ["monday","tuesday","wednesday","thursday","friday"]]
d. z = foldl (\x y -> x ++ y)
e. foo bar = []
```

Answers:

Any option listed below is worth full credit. Answers that fail to use types t1, t2, etc. but instead use other letters or type names should get full credit if their answer is valid. An answer is either entirely correct or entirely wrong.

a.

```
foo :: Num t1 => [t1] -> [t1]
foo :: [Int] -> [Int]
foo :: [Integer] -> [Integer]
```

full points if they don’t include the function name (e.g. ‘foo ::’)
full points if they don’t use a generic for instance of the Num class (e.g. giving full points for ‘Num => [Num] -> [Num]’

b.

```
z :: [[Char]] -> [[Char]]
z :: [String] -> [String]
```

c.

```
m :: [(Bool, [Char])]
m :: [(Bool, String)]
[(Bool, [Char])]
[(Bool, String)]
```

d.

```
      z :: [t1] -> [[t1]] -> [t1]
      z :: Foldable t2 => [t1] -> t2 [t1] -> [t1]
      note: the second answer involves type classes, which is out of
scope of the class!

e.
      foo :: t1 -> [t2]
```

2. [5 points] Suppose a language does not support closures. Is it possible for the language to still support currying and partial function application? If so, how would this work; if not, explain why not. Note: Limit your answer to five or fewer sentences, and avoid adding superfluous, incorrect answers as this will result in a point deduction.

Answer:

- Any answer that hits the following points gets full credit:

  Currying requires closures to capture free variables used in the inner lambda expressions. If there are no closures then we can't create curried functions or do partial application.

- If the student just says currying/PA is not possible but does not give a rationale they get 1 point
- If the student also includes superfluous answers that are not correct, take one point off for each incorrect answer that is present.
- The minimum score for this problem is zero.

3. [16 points-2/2/2/10] In this question, you will be implementing *directed graph* algebraic data types and algorithms in Haskell. Recall that a graph is composed of nodes that hold values, and edges which connect nodes to other nodes. In a directed graph, edges are unidirectional, so an outgoing edge from node A to node B does NOT imply that B also has a direct connection to A. You may assume that all graphs have at least one node.

a. Show the Haskell definition for an algebraic data type called "Graph" that has a single “Node” variant. Each Node must have one Integer value, and a list of zero or more adjacent nodes that can be reached from the current node.

Answer:

Any of the following answers is OK:

```
data Graph = Node Int [Graph]
data Graph = Node Integer [Graph]
data Graph = Node [Graph] Int
data Graph = Node [Graph] Integer
```

If the student uses a type variable with a type of Integral that's ok too.
All other answers get a zero.

b. Using your Graph ADT, write down a Haskell expression that represents a graph with two nodes:

1. A node with the value 5 with no outgoing edges
2. A node with the value 42, with an outgoing edge to the previous node

Answer:

The students may use any variable names they like (instead of a and b).

Given these definitions from part a:

```
data Graph = Node Int [Graph]
data Graph = Node Integer [Graph]
```

Either of the following options are valid:

```
a = Node 5 []                - 1 point
b = Node 42 [a]              - 1 point

b = Node 42 [Node 5 []]      - 2 points
```

Given these definitions from part a:

```
data Graph = Node [Graph] Int
data Graph = Node [Graph] Integer
```

Either of the following options are valid:

```
a = Node [] 5                - 1 point
b = Node [a] 42              - 1 point

b = Node [Node [] 5] 42      - 2 points
```

c. Carey has designed his own graph ADT, and written a Haskell function that adds a node to the “front” of an existing graph g that is passed in as a parameter:

```
add_to_front g = Node 0 [g]
```

In Haskell, creating new data structures incurs cost. Assuming there are N nodes in the existing graph *g*, what is the time complexity of Carey's function (i.e., the big-O)? Why?

Answer:

O(1) because only a single new node is being created, and it links to the original graph. The existing graph need not be regenerated in any way.

d. Write a function called *sum_of_graph* that takes in one argument (a Graph) and returns the sum of all nodes in the graph. Your function must include a type annotation for full credit. You may assume that the passed-in graph is guaranteed to have no cycles. You may have helper function(s). *Your solution MUST be shorter than 8 lines long.*

Answer:

Any of these answers deserves full marks. They require one of the following type annotations:

```
sum_of_graph :: Graph -> Int
sum_of_graph :: Graph -> Integer

--- carey's solution

sum_of_graph (Node val edges) =
  val + (sumx_aux edges)
 where
```

```
  value (Node val edges) = val + (sumx_aux edges)
  sumx_aux edges =
    sum [value n | n <- edges]

--- matt's solution (map + sum)
sum_of_graph (Node val edges) = val + sum (map sum_of_graph edges)

--- matt's solution (map + fold)
sum_of_graph (Node val edges) = foldl (+) val (map sum_of_graph edges)

--- optionally, students might include a base case like the following, though
observe that it's not strictly necessary
sum_of_graph (Node val []) = val

--- many other solutions are possible. other neat ones we saw included:
--- - parsing the edges node-by-node, which required:
---   - a mutually recursive helper function
---   - or, "recreating" a new node with value 0, and the tail as its edges
--- - foldl/foldr, but calling sum_of_graph on each edge
```

Rubric Sketch (negative grading)

- Perfect (-0 pts)
- Minor syntax error(s) that did not affect understanding (-1 pts)
  - ex: forgetting the parens for an ADT variant argument
- Incorrect base case / singleton node (-2 pts)
  - ex: base case has value 0
- Minor recursive logic bug (-2 pts)
  - requires a fully-working solution, with one minor tweak
  - ex: switching the order of the args to foldl/foldr
  - ex: used sum improperly
- Major recursive logic bug / multiple minor bugs (-5 pts)
  - ex: recursive solution tries to add a Graph and Integer
  - ex: recursive solution tries to call sum_of_graph [Graph]
  - ex: recursive solution misses multiple levels of children
- Incorrect/missing type definition (-1 pts)
  - all-or-nothing; did give ECF
  - if *only* submitted typedef, received 1 pt total
- Did not attempt (-10 pts)

A handful of students noted that our definition of acyclic was unclear (is this acyclic in the undirected sense, or in the directed sense)? The latter has a "diamond problem" built-in; we were expecting solutions for the former.

If you tried to deal with this but had a minor logic bug, we gave you full credit.

4. [10 points] Write a Haskell function called *get_every_nth* that takes in an Integer n and a list of generic types that returns a sublist of every n^{th} element. For example:

```
get_every_nth 2 ["hi","what's up","hello"] should return ["what's up"]
get_every_nth 5 [31 .. 48] should return [35, 40, 45]
```

Your function must have a type signature and must use either filter, foldr or foldl. It must be less than 8 lines long.

Hint: If you use fold, your accumulator doesn't need to be the same type as your returned value or your input, and a tuple might be useful!

Carey's answer:

```
get_every_nth :: Integer -> [a] -> [a]
get_every_nth n lst =
  map (\tup -> snd tup)                    -- or map (\(_,x) -> x)
      (filter (\tup -> (fst tup) `mod` n == 0) (zip [1..] lst))
```

Ruining's answer:

```
get_every_nth :: Integer -> [a] -> [a]
get_every_nth n lst =
 let f = \(res,index) x -> if mod index n == 0 then (x:res, index+1) else
(res,index+1)                                    -- this 'let' is one line
 in reverse( fst (foldl f ([],1) lst))
also acceptable;
(\(x,_) ->x) instead of fst
```

Possible common mistakes:

- 0/10 for empty/ all wrong
- 2/10 for an ok attempt (correct type, usage of functions is alright)
- 5/10 for good idea
- 7/10 for multiple syntax/edge problems
  - indexing off
- 9/10 for syntax/edge problems (might bump up, not sure)
  - wrong operator/symbol
  - reverse
  - convert tuple to list
- 10/10 for perfect

(comprehensions count as filters)

5. [5 points] Write a Haskell comprehension that generates an infinite list of functions, each of which takes a single argument x, such that the k^{th} function returns x^k. You may assume that k starts at 1 for the first generated function, 2 for the second function, etc. For example:

inf_list = [your comprehension here]
head inf_list 9 → returns 9
head (tail inf_list) 9 → returns 81

Answer:

```
inf_list = [\x -> z^x | z <- [1..]]       - carey's answer
```

Possible common mistakes:

- not returning a list of functions ([z^x | z <- [1..]]) -> -3 pts
- using the wrong operator (neither ** or ^) -> -3 pts
- extraneous argument in lambda ([\x k -> x^k | k <- [1..]] -> -1 pt

6. [10 points-5/5] Consider the following Python program:

```
class Comedian:
  def __init__(self, joke):
    self.__joke = joke

  def change_joke(self, joke):
    self.__joke = joke

  def get_joke(self):
    return self.__joke

def process(c):
 # line A
 c[1] = Comedian("joke3")
 c.append(Comedian("joke4"))
 c = c + [Comedian("joke5")]
 c[0].change_joke("joke6")

def main():
 c1 = Comedian("joke1")
 c2 = Comedian("joke2")
 com = [c1,c2]
 process(com)
 c1 = Comedian("joke7")
 for c in com:
  print(c.get_joke())
```

a. Assuming we run the main function, what will this program print out?

Answer:

Partial credit for this part (a) and the next part (b) is only awarded if either the student fails to properly take into account a single valid mutation, or incorporates a single invalid mutation when there wasn't supposed to be one.

Rubric for MT1 6.a

5 points for:
joke6
joke3
joke4

2.5 points for:
joke6
joke2
joke4

2.5 points for:
joke6
joke3

2.5 points for:
joke6
joke3
joke4
joke5

2.5 points for:
joke1
joke3
joke4

2.5 points for:
joke7
joke3
joke4

All other answers get a zero

b. Assuming we removed the comment on line A and replaced it with:

```
    c = copy.copy(c)        # initiate a shallow copy
```

If we run the main function, what would the program print?

Answer:

5 points for:
joke6
joke2

2.5 points for:
joke1
joke2

2.5 points for:
joke3
joke2

7. [9 points-3/3/3] We know that int and float are usually not subtypes of each other. Assume you have a programming language where you can represent integers and floating-point values with infinite precision (call the data types Integer and Float). In this language, the operations on Integer are +, -, *, / and % and the operations on Float are +, -, *, and /.

a. In this language, will Integer be a subtype of Float? Why or why not?

Answer:

3 points for the following:

Yes, Int is a subtype of Float because:

1. Every Integer value can be represented by a Float (with the fractional part being zero).
2. Every operation on Float can also be performed on an Integer.

3 points off if they do not say "Yes"
1 point off for missing either requirement

b. In this language, will Float be a subtype of Integer? Why or why not?

Answer:

3 points for the following:

No, Float is a NOT a subtype of Integer because:

1. Not every Float value can be represented by an Integer
2. Not every operation on an Integer can also be performed on a Float.

3 points off if they do not say "No"
2 points off for missing justification (must mention one of the two)

c. If you discovered that the Float type also supports an additional operator, exponentiation, what effect would that have on your answers to a and b? Why?

Answer:

3 points for the following:

Neither would be subtypes of each other. Both reasons needed for full credit:

1. Floats have an operator that Integers don't support so Integers can't be a subtype of Float
2. Integers have an operator that Floats don't support so Floats can't be a subtype of Integer (or alternatively, not every float value can be an Integer)

2 points off for incorrect/missing justification

8. [9 points-3/3/3] Siddarth has created a compiler for a new language he's invented. He gives you the following code written in the language, with line numbers added for clarity:

```
00  func mystery(x) {
01    y = x;
02    print(f"{y}");
03    y = 3.5;
04    print(f"{y}");
05    return y;
06  }
07
08  q = mystery(10);
09  print(f"{q}");
```

Siddarth tells you that the program above outputs the following:

```
10
3
3
```

a. What can you say about the type system of the language? Is it statically typed or dynamically typed? Explain your reasoning.

Answer:

3 points for something close to this reasoning:

- It is statically typed.
- Why: because y's type remains an integer even when we assign it to a floating point value (3.5)

1 point for saying it's statically typed but without reasoning

b. Is either casting or conversion being performed in this code? If so, what technique are being used on what line(s)? If you do identify any casts/conversions, explain for each if they are narrowing or widening.

Answer:

3 points for an answer that hits both of the items below:

Yes, the code is using a conversion on line 3 where it converts 3.5 into an integer value of 3.

This is a narrowing conversion since it loses precision when it truncates to an integer.

1 point off for saying casting assuming everything else is right
1 point off for saying widening instead of narrowing
1 point off for failing to specify the line number

c. Assuming the language used the opposite type system as the one that you answered in part a, what would the output of the program be?

Answer:

10
3.5
3.5

-1.5 points for the following:
“10”
“3.5”
“3.5”

-1.5 points for the following:
10.0
3.5
3.5

d. Ruining likes the language so much that she built her own compiler for the language, but with some changes to the typing system. Siddarth wants to determine what typing system Ruining chose for her updated language, so he changed eline 3 above to:

y = “3”;

and found that the program still runs and produces the same output. What can we say about the typing system for Ruining's language? Is it static or dynamic? Or is it impossible to tell? Why?

Answer:

3 points for the either of following options:

Ruining's language must be dynamically typed.
Why? Because the language allows the y variable to be assigned to two different types over time.

Ruining's language is statically typed and the string "3" undergoes conversion into an int. (must explicitly mention conversions or coercion)

2 points off if the student doesn't explain why properly.

I gave 3 points for either of the 3 answers (static, dynamic, or impossible to tell), as long as they provided reasonable justification.