CptS355 - Programming Language Design
Spring 2020

Sakire Arslan Ay, PhD

School of Electrical Engineering and Computer Science
Washington State University

Midterm 1 Sample Solutions

Time Limit: 50 minutes

- Print your name and WSU ID below.

Name: ___key__________________________ WSU ID:___key___________

Directions: Answer all of the questions. You may have **one 8 1/2 X 11 sheet of notes** during the exam. You may **not** use a computer, electronic portable device, calculator or phone during the test. Remove all caps and visors.

There are X major problems on Y pages totaling 100 points. **Make sure that you have a complete exam before beginning.**

Write your name on your exam **now**.

Q1 (12pts)	**Q2 (5pts)**	**Q3 (10pts)**	**Q4 (10pts)**	**Q5 (12pts)**	**Q6 (8pts)**	**Q7 (25pts)**	**Q8 (18pts)**	**Total**

1) **[12 pts] Types** - Short answer questions
(3 **pts each**)

a) What is the Haskell type of `(3,[4,5])` ?

Will accept all below answers:

```
(Int, [Int])

(Integer, [Integer])

(Num a1, Num a2) => (a1, [a2])
```

b) What is the Haskell type of function f?

```
  f x y = (x,y)
```

```
  f :: a -> b -> (a, b)
```

c) A Haskell tuple contains:

1) values that must all have the same type
2) values that may have different types
3) exactly two values
4) both 1 and 3
5) both 2 and 3

d) Is `[1,2,3,[4,5]]` a legal list value in Haskell? Why or why not?

No, it is not a legal list. It includes both `Int` and `[Int]` values. The elements of a Haskell list needs to be all of the same type.

2) **[5 pts]** What is the principal difference between statically typed and dynamically typed languages? Mention one statically typed language and one dynamically typed language.

Statically typed -> type checking is done during compile-time (examples: C/C++, Java, ML)
Dynamically typed -> type checking is done during run-time (examples: Python, Scheme/Racket)

3) **[10 pts]** Consider the following Haskell functions:

```
step x y = y + 1
mystery iL = foldr step 0 iL
```

What will the following expression evaluate to?

```
mystery ["ab","bc","cd","bc"]
```

```
1 + mystery ["bc","cd","bc"]
1 + 1 + mystery ["cd","bc"]
1 + 1 + 1 + mystery ["bc"]
1 + 1 + 1 + 1+ mystery []
= 4
```

Mystery returns length of the input list.

4)**[12 pts]** The following Haskell function is expected to apply function argument `op` to every other value in the input list, starting with the first value. For example:

```
everyOther (\x -> 0-x) [1,2,3,4,5,6]  returns  [-1,2,-3,4,-5,6]
```

(a) However when the function is defined as below, it gives a *type error.* Explain what might have caused the error.

```
everyOther [] = []
everyOther op (x:y:xs) = (op x) : y : (everyOther xs)

everyOther op [] = []
everyOther op (x:y:xs) = (op x) : y : (everyOther op xs)
```

(b) Assume you fixed the type error. When you test the everyOther function you realize that the function works with some input lists, but gives an exception for some other inputs. Fix the bug in the code and re-write this function.

The patterns are not exhaustive in the above everyOther definition. The pattern for the input list including a single element is missing. So, the function will give an exception for any input list that includes odd number of values. Below is the complete implementation of everyOther.

```
everyOther op [] = []
everyOther op [x] = (op x): []
everyOther op (x:y:xs) = (op x) : y : (everyOther op xs)
```

5) [**12 pts**] Binary trees with Int data in the interior nodes (and no data at the leaves) can be represented by

```
data BTree = LEAF | NODE Int BTree BTree
```

Give a definition for a function doubleT having type doubleT :: BTree -> BTree, that makes of copy of the tree given as argument but with all the contained Int values doubled.

```
doubleT :: BTree -> BTree
doubleT (LEAF) = LEAF
doubleT (NODE v t1 t2) = NODE (2*v) (doubleT t1) (doubleT t2)
```

6) [**8 pts**] Consider the following myFunction function.

```
myFunction []      = 0
myFunction [(a,b)] = a + b
myFunction ((a,b):(c,d):rest) = b + d + (myFunction rest)
```

(a) **[3 pts]** Is myFunction tail-recursive? Explain why/why not.

No, it is not tail recursive.

(b) Give the value that ans will be assigned to after evaluating the following. Show your work.

```
ans = myFunction ( map (\x -> (1,x)) [1,2,3,4,5] )
```

result of map:
[(1,1),(1,2),(1,3),(1,4),(1,5)]

ans:
1 + 2 + myFunction [(1,3),(1,4),(1,5)]
1 + 2 + 3 + 4 + myFunction [(1,5)]
1 + 2 + 3 + 4 + 1 + 5 = 16

7) **[25 pts]** A Haskell list of tuples can be used to represent a lookup table where the first value of each pair is the key and the second value is the associated value.
For example: `[("a",1),("b",2),("c",3),("a",4)]`
(*Assume the keys in a lookup table are not necessarily unique.*)

We want to define a Haskell function (`lookup k table`) that returns the values for key `k` in the lookup table. If key `k` doesn't appear in table it should return empty list.
Thus,

```
lookup 3   [(1,"4"),(2,"5"),(1,"3")]           returns []
lookup 1   [(1,"4"),(2,"5"),(1,"3")]           returns ["4","3"]
lookup "b" [("a",1),("b",2),("c",3),("a",4)]   returns [2]
```

(a) [**5 pts**] What should be the Haskell type of the `lookup` function ? *(Note: The input list does not necessarily "(Int,String)" or "(String,Int)" tuples.)*

```
Eq a1 => a1 -> [(a1, a2)] -> [a2]
```

(b) [**10 pts**] Give a recursive definition of the `lookup` function. (*Part of the code is given; complete the following.)*

```
lookup k [] =  []
lookup k ((x,y):xs) | (x == k) = y:(lookup k xs)
                    | otherwise = (lookup k xs)
```

(c) [**10 pts**] The following `isKey` function is a predicate function that checks whether a given value `v` is the key in the given pair.

```
  isKey v (x,y) = (v==x)
```

For example : `(isKey 1 (1,"4"))` returns `True`

Now re-define the `lookup` function using `isKey`, `map`, and `filter` functions. Your solution should not use explicit recursion. You may define additional non-recursive helper function(s) if needed.

```
lookup v iL = map snd (filter (isKey v) iL)
```

8) **[18 pts]** The following Haskell function , maxL, returns the maximum value in a given list of positive integers.

```
    maxL iL = foldl max 0 iL
```

We define a function maxLE (standing for "**max**imum of a **L**ist, possibly **E**mpty") that returns **Nothing** when given the empty list and returns **Just v** when given a non-empty list and **v** is the maximum value in that list.
For example (maxLE [2,5,1]) is Just 5.

maxLE can be implemented using foldl like this:

```
    maxLE iL = foldl maxMaybe Nothing iL
```

a) [**8 pts**] Give a definition of the maxMaybe function that will work with the above definition of maxLE.

```
maxMaybe Nothing  y = Just y
maxMaybe (Just x) y = Just (max x y)
```

b) [**5 pts**] Assume we reimplement maxLE using foldr like this:

```
    maxLE2 iL = foldr maxMaybe2 Nothing iL
```

Give a definition of the maxMaybe2 function that will work with foldr ?

```
maxMaybe y Nothing = Just y
maxMaybe y (Just x) = Just (max x y)
```

c) [**5 pts**] Will the above definition of maxLE work for lists that contain only negative integers? Explain your answer.

Yes it will. Even if the list includes negative integers, since the base case (Nothing) is assumed to be less than any value in the list, maxLE will always return the ~~smallest~~ largest value in the list (as a Maybe value)