Functional Programming
Assignment 1

Ulfet CETIN - 391819
Alisafa GASIMOV - 392560
Orkhan HUSEYNOV - 374297
Shreya KAR - 392325
(names sorted on last name basis)

April 24, 2019

Note: respective solutions could be found in the archive file we send
(filenames: exercise1.hs, exercise2.hs, exercise3.hs, exercise4.hs)

Exercise 1 (Function types): ((0.5 + 0.5 + 1) + 1 = 3 points))

a) Give examples of Haskell function declarations with the following types and briefly explain their semantics. Your solutions must not ignore any of their arguments completely.

i) Bool -> Bool -> Int

```
-- counts how many of the the two inputs is true
f1 :: Bool -> Bool -> Int
f1 True True = 2
f1 True False = 1
f1 False True = 1
f1 _ _ = 0
```

ii) [Int] -> [Bool] -> Int

```
-- assumes that the first and the second argument is of same size
-- check, in how many of elements of the first argument,
--   the truth value of condition (greater than zero, in our case)
--     is the same as the respective element of second argument
f2 :: [Int] -> [Bool] -> Int
f2 [] [] = 0
f2 (x:xs) (y:ys)
  | (x > 0) == y = 1 + (f2 xs ys)
  | otherwise = 0 + (f2 xs ys)
```

iii) [Bool] -> (Bool -> Int) -> [Int]

```
-- maps each bool value in first argument using the second argument
-- second argument should be a function that takes as input a bool and produces an int
--   an example for second argument could be a voltageMapper function given below
-- example usage:
--   f3 [True,False,True,True,False] voltageMapper
-- answer:
--   [5,0,5,5,0]
f3 :: [Bool] -> (Bool -> Int) -> [Int]
f3 [] mapFunc = []
f3 (x:xs) mapFunc = (mapFunc x):(f3 xs mapFunc)

-- maps voltages into the values that system accepts
-- for computer systems, it is either 5V or 12V, we preferred 12V
voltageMapper :: Bool -> Int
voltageMapper True = 5
voltageMapper False = 0
```

b) Suppose that f has the type Bool -> [Int] -> Int.

What is the type of x y -> f ((f True x)>0) [y]?

```
-- \x y -> f ((f True x)>0) [y]?
--            (f True x) evaluates to Int
--           (     Int    >0) evaluates to Bool
--         f      Bool       [Int] evaluates to Int, as definition of f shows
-- type of " \x y -> f ((f True x)>0) [y] " is Int
```

Exercise 2 (Lists): (2 + 3 = 5 points)

a) For each of the following equations, if possible, give pairwise different example values for x, y, and z such that the equation holds. Otherwise explain why such an assignment is not possible.

i) [[x],[y]] == [y]:z

```
x = 1
y = 1
z = [[1]]
```

ii) ([x] ++ z):y == (x:z):y

```
x = []
y = []
z = []
```

iii) [[ ]] ++ ([x]:y) == ([x]:z)

```
If we simplify the left side of the giving equation, then we get the following:
[[]] ++ ([x]: y) = [[]] ++ [[x], ...] == [[], [x], ...]

If we look at the right-hand side of the giving equation and compare it with the
simplified expression, then we can easily see that first elements of the lists are
different. Because of that we can say that in any values of x, y and z such an
assignment is not possible.

The simplified expression: [[], [x], ...]
The right-hand side of the given equation: ([x], ...) == [[x], ...]
So, [[], [x], ...] can not be equal to [[x], ...].
```

iv) (x:y):z == (y ++ [x]):z

```
x = 5
y = []
z = []
(x:y):z == (y ++ [x]):z
```

b) Consider the following patterns

p1) ([x]++y):ys

p2) (x:y)++ys

and the following terms:

t1) [[]]

t2) [[1,2],[3]]

For each pair of a pattern and a term, indicate whether the pattern matches the term. If so, provide the appropriate matching substitution. Otherwise, explain why the pattern does not match the term. Does there exist a term that is matched by p1 but not by p2? Justify your answer.

– p1 - t1

```
Lets assign x=[] empty list
then inside of bracket we will get [[]] in
    very simple case.
In second part of this expression, achieved
    result inside of bracket ([[]]) should be
    added to the ys element which should
    append to the new nested list,
so in very simple case,
    we can get [[[]],ys element]
    which can not be equalised with [[]]
```

– p2 - t1

```
p2) (x:y)++ys
t1) [[]]

x = []
y = []
ys = []

does match
```

– p1 - t2

```
p1) ([x]++y):ys
t2) [[1,2],[3]]

x = 1
y = [2]
ys = [[3]]

does match
```

– p2 - t2

```
p2) (x:y)++ys
t2) [[1,2],[3]]

x = [1,2]
y = []
ys = [[3]]

does match
```

Exercise 3 (Programming): $(2 + 2 + 3 + 3 = 10$ points)

Note that you may use constructors like [], :, True, False in all of the following subexercises. You may also write auxiliary functions if needed or reuse functions from earlier subexercises (even if you did not manage to implement them).

a) Write a Haskell–function myrem, where myrem x y is the remainder of the integer division when dividing x by y. So for example, myrem 14 3 == 2. If y == 0 then myrem x 0 == x. If y < 0 then myrem x y == myrem x (-y). myrem :: Int -> Int -> Int

   You may not use any predefined functions except comparisons, +, and -.

```
myrem :: Int -> Int -> Int
myrem x 0 = x
myrem x y
  | y < 0 = myrem x (-y)
  | x >= y = myrem (x-y) y
  | otherwise = x
```

b) Write a Haskell-function count that given a list xs and an element x returns the number of occurences of x in xs. E.g., count 2 [0,2,2,0,2,5,0,2] == 4 wheras count (-7) [0,2,2,0,2,5,0,2] == 0.
   count :: Int -> [Int] -> Int

   You may not use any predefined functions except comparisons and +.

```
count :: Int -> [ Int ] -> Int
count _ [] = 0
count wanted (x:xs)
  | wanted == x = 1 + count wanted xs
  | otherwise = count wanted xs
```

c) Write a Haskell-function simplify that given a list xs returns a list of pairs as follows. The resulting list contains the pair (x,n) if and only if x occurs in xs n times and n > 0. E.g., simplify

   [0,2,2,0,2,5,0,2] == [(0,3),(2,4),(5,1)].

   simplify :: [Int] -> [(Int ,Int)]

   You may not use any predefined functions except comparisons.

```
-- is the first argument the smallest element of the second argument?
isTheSmallest :: Int -> [Int] -> Bool
isTheSmallest _ [] = True
isTheSmallest elem (x:xs)
  | elem <= x = isTheSmallest elem xs
  | otherwise = False

-- my personal sort function to be used as helper
mySort :: [Int] -> [Int]
mySort [] = []
mySort (x:xs)
  | isTheSmallest x xs = x:(mySort xs)
  | otherwise = mySort (xs ++ [x])

-- helper function(s) for part c
-- finds whether element is in the list
isIn :: Int -> [Int] -> Bool
isIn _ [] = False
```

```haskell
isIn elem (x:xs)
  | elem == x = True
  | otherwise = isIn elem xs

-- find unique elements of a list
-- second argument should start as empty list, i.e. []
findUniquesHelper:: [Int] -> [Int] -> [Int]
findUniquesHelper [] resultList = resultList
findUniquesHelper (x:xs) resultList
  | isIn x resultList = findUniquesHelper xs resultList
  | otherwise = findUniquesHelper xs (resultList ++ [x])

-- main findUniques function making use of the helper function above
findUniques :: [Int] -> [Int]
findUniques list = mySort ( findUniquesHelper list [] )

-- counts how many times an element appears in a list
countSingleElement :: Int -> [Int] -> Int
countSingleElement _ [] = 0
countSingleElement elem (x:xs)
  | elem == x = 1 + countSingleElement elem xs
  | otherwise = countSingleElement elem xs

-- first argument should be the unique elements of the second list
simplifyHelper :: [Int] -> [Int] -> [(Int, Int)]
simplifyHelper [] _ = []
simplifyHelper (unique:uList) targetList =
  [(unique, countSingleElement unique targetList)] ++ simplifyHelper uList targetList

-- counts which element appeared how many times in the list
simplify :: [ Int ] -> [( Int , Int )]
simplify [] = []
simplify list = simplifyHelper (findUniques list) list
```

d) Write a Haskell-function multUnion that given two lists of pairs xs and ys concatenates these lists where each "multiple occurence" is simplified as follows: If xs contains a pair (x,n) and ys contains (x,m), then the result contains (x,n+m). You may assume that in both xs and ys an integer occurs at most once as first entry of a pair. Moreover, assume that the lists are sorted in ascending order w.r.t. the first entry of the pair. Make sure that the resulting list is sorted in ascending order w.r.t. the first entry of the pair as well. E.g., multUnion[(0,3),(2,4),(5,1)] [(-1,1),(0,4)] == [(-1,1),(0,7),(2,4),(5,1)].

multUnion :: [(Int ,Int)] -> [(Int ,Int)] -> [(Int ,Int)]

You may not use any predefined functions except comparisons and +

```haskell
-- get all unique elements from that two list
findUniquesTupleVersionHelper :: [(Int, Int)] -> [Int] -> [Int]
findUniquesTupleVersionHelper [] resultList = resultList
findUniquesTupleVersionHelper ( (key,count):list ) resultList
  | isIn key resultList = findUniquesTupleVersionHelper list resultList
  | otherwise = findUniquesTupleVersionHelper list (key:resultList)

-- takes two lists as an argument and finds the unique keys
findUniquesTupleVersion :: [(Int, Int)] -> [(Int, Int)] -> [Int]
findUniquesTupleVersion list1 list2 = mySort (findUniquesTupleVersionHelper (list1 ++
    list2) [] )

-- write a retriever from the lists
countRetriever :: Int -> [(Int, Int)] -> Int
countRetriever key [] = 0
countRetriever key ((key1, count1):xs)
  | key == key1 = count1
  | otherwise = countRetriever key xs

multUnionHelper :: [Int] -> [(Int, Int)] -> [(Int, Int)] -> [(Int, Int)]
multUnionHelper [] _ _ = []
multUnionHelper (u:uniqueList) list1 list2 =
  let summed = ( countRetriever u list1 ) + ( countRetriever u list2 )
  in ( (u,summed):(multUnionHelper uniqueList list1 list2) )

multUnion :: [(Int, Int)] -> [(Int, Int)] -> [(Int, Int)]
multUnion list1 list2 = multUnionHelper (findUniquesTupleVersion list1 list2) list1
    list2
```

Exercise 4 (Infix Operators): (2+1* points)

Define a Haskell function ^^^ in infix notation with the type declaration
(^^^) :: [Int] -> [Int] -> Int
such that the following holds for lists of equal length:

- The function call xs ^^^ ys evaluates to xs to the power of ys interpreted as vectors, where the negative entries of ys are ignored. In other words, [x1, x2, ..., xn] ^^^ [y1, y2, ..., yn] == x1 ^ y1 * x2 ^ y2 * ... * xn ^ yn.
- For example [1, 4, 5] ^^^ [7, 2, 3] evaluates to 1 ^ 7 * 4 ^ 2
  * 5 ^ 3 == 2000 and [1, 4, 5] ^^^ [5, -1, 0] evaluates to 1 ^ 5 * 5 ^ 0 == 1.
- xs ^^^ ys * z, where xs and ys have type [Int] and z has type Int, is a valid expression.

  The function ^^^ may behave arbitrarily if the two arguments have different lengths. You may not use any predefined functions except *, ^, and comparisons. You may, of course, use constructors like [] and :.

  You can get one bonus point if you solve the exercise even without using the predefined function ^.

```haskell
-- equivalent of ^ operator, user defined for bonus points
myPowOp :: Int -> Int -> Int
myPowOp _ 0 = 1
myPowOp a b = a * (myPowOp a (b-1))

(^^^) :: [Int] -> [Int] -> Int
[] ^^^ _ = 1
_ ^^^ [] = 1
(x:xs) ^^^ (y:ys)
  | y <0 = xs ^^^ ys
  | otherwise = (myPowOp x y) * (xs ^^^ ys)

infixl 9 ^^^
```