CS 420 – Advanced Programming Languages

Assignment 2 – Functional Programming in Haskell – 100 points

The primary goal of this assignment is to help you familiarize with **Functional Programming** paradigms**.** You will learn to:

- use simple function definitions.
- Pattern Matching in Haskell
- Recursion in Haskell
- Folds in Haskell
- Lazy evaluation in Haskell

The assignment has 19 questions and comes in 3 parts with increasing order of difficulty.

Part A: Simple functions in Haskell

For the following problems you are use any Prelude library function on integers but only the following prelude library functions on lists:

length
(++)
(==)

However, you may use the definitions you write to solve the subsequent problems.

Write a Haskell definition called,

1. sumList to compute Sum the elements of a list

```
sumList :: [Int] -> Int
```

2. digitsOfInt – to return `[]` if `n` is not positive, and otherwise returns the list of digits of `n` in the order in which they appear

```
digitsOfInt :: Int -> [Int]
```

3. digits - returns the list of digits of `n`

```
digits :: Int -> [Int]
```

4. additivePersistence – Explaination can be found in the link: http://mathworld.wolfram.com/AdditivePersistence.html

```
additivePersistence :: Int -> Int
```

5. digitalRoot - is the digit obtained at the end of the sequence computing the additive Persistence.

```
digitalRoot :: Int -> Int
```

6. ListReverse – to reverse a list.

```
listReverse :: [a] -> [a]
```

7. Palindrome – check if the provided list is a palindrome.

```
palindrome :: String -> Bool
```

8. rootList – returns the digitalroot of all elements in a list.

```
rootList :: [Int] -> [Int]
```

Once again, the above functions should be implemented without using the prelude library functions. You are allowed to write helper functions for your functions (if need be).

Part B: Intermediate functions in Haskell

9. Without using any built-in functions, write a **tail-recursive** function.

```
assoc :: Int -> String -> [(String, Int)] -> Int
```

such that *assoc def key [(k1,v1), (k2,v2), (k3,v3);...])*

searches the list for the first i such that ki = key. If such a ki is found, then vi is returned. Otherwise, if no such ki exists in the list, the default value def is returned.

Once you have implemented the function, you should get the following behavior:

```
ghci> assoc 0 "william" [("ranjit", 85), ("william",23), ("moose",44)])
23

ghci> assoc 0 "bob" [("ranjit",85), ("william",23), ("moose",44)]
0
```

10. Use the library function ***elem*** to write a function called removeDuplicates to obtain a function of type. You function should be **tail-recursive**.

```
removeDuplicates :: [Int] -> [Int]
```

such that removeDuplicates xs returns the list of elements of xs with the duplicates, i.e. second, third, etc. occurrences, removed, and where the remaining elements appear in the same order as in xs.

Once you have implemented the function, you should get the following behavior:

```
ghci> removeDuplicates [1,6,2,4,12,2,13,12,6,9,13]
[1,6,2,4,12,13,9]
```

11. Without using any built-in functions, write a **tail-recursive** function:

```
wwhile :: (a -> (Bool, a)) -> a -> a
```

such that wwhile f x returns x' where there exist values v_0,...,v_n such that

- `x is equal to v_0`
- `x' is equal to v_n`
- `for each i between 0 and n-2, we have f v_i equals (true, v_i+1)`
- `f v_n-1 equals (false, v_n).`

Once you have implemented the function, you should get the following behavior:

```
ghci> let f x = let xx = x * x * x in (xx < 100, xx) in wwhile f 2
512
```

12. Without using any built-in functions,

```
fixpointL :: (Int -> Int) -> Int -> [Int]
```

The expression fixpointL f x0 should return the list [x_0, x_1, x_2, x_3, ... , x_n] where

- `x = x_0`
- `f x_0 = x_1, f x_1 = x_2, f x_2 = x_3, ... f x_n = x_{n+1}`
- `x_n = x_{n+1}`

When you are done, you should see the following behavior:

```
>>> fixpointL collatz 1
[1]
>>> fixpointL collatz 2
[2,1]
>>> fixpointL collatz 3
[3,10,5,16,8,4,2,1]
>>> fixpointL collatz 4
[4,2,1]
>>> fixpointL collatz 5
[5,16,8,4,2,1]
```

13. Without using any built-in functions, write a definition called fixpointW to obtain a function

```
fixpointW :: (Int -> Int) -> Int -> Int
```

such that fixpointW f x returns the last element of the list returned by fixpointL f x.

Once you have implemented the function, you should get the following behavior:

```
ghci> fixpointW collatz 1
1

ghci> fixpointW collatz 2
1

ghci> fixpointW collatz 3
1

ghci> fixpointW collatz 4
1

ghci> fixpointW collatz 5
1
```

Part C: Advanced functions in Haskell using Folds and Lazy Evaluation

14. Write a function called `sqsum`, which uses `fold` to get a function

```
sqSum :: [Int] -> Int
```

such that `sqSum [x1,...,xn]` returns the integer `x1^2 + ... + xn^2`

Once you have implemented the function, you should get the following behavior:

```
ghci> sqSum []
0

ghci> sqSum [1, 2, 3, 4]
30

ghci> sqSum [(-1), (-2), (-3), (-4)]
30
```

15. Write a function called pipe which uses fold to get a function

```
pipe :: [(a -> a)] -> (a -> a)
```

such that pipe [f1,...,fn] x (where f1,...,fn are functions!) should return f1(f2(...(fn x))).

Once you have implemented the function, you should get the following behavior:

```
ghci> pipe [] 3
3

ghci> pipe [(\x -> x+x), (\x -> x + 3)] 3
12

ghci> pipe [(\x -> x * 4), (\x -> x + x)] 3
24
```

16. Write a function for sepConcat, which uses fold to get a function

```
sepConcat :: String -> [String] -> String
```

Intuitively, the call sepConcat sep [s1,...,sn] where

- sep is a string to be used as a separator, and
- [s1,...,sn] is a list of strings

should behave as follows:

- sepConcat sep [] should return the empty string "",
- sepConcat sep [s] should return just the string s,
- otherwise (if there is more than one string) the output should be the string s1 ++ sep ++ s2 ++ ... ++ sep ++ sn.

Once done, you should get the following behavior:

```
ghci> sepConcat ", " ["foo", "bar", "baz"]
"foo, bar, baz"

ghci> sepConcat "---" []
""

ghci> sepConcat "" ["a", "b", "c", "d", "e"]
"abcde"

ghci> sepConcat "X" ["hello"]
"hello"
```

17. Write a function for `stringOfList`

```
stringOfList :: (a -> String) -> [a] -> String
```

such that `stringOfList f [x1,...,xn]` *should return the string* `"[" ++ (f x1) ++ ", " ++ ... ++ (f xn) ++ "]"`

Hint: This function can be implemented on one line, **without using any recursion** by calling `map` and `sepConcat` with appropriate inputs.

You should get the following behavior:

```
ghci> stringOfList show [1, 2, 3, 4, 5, 6]
"[1, 2, 3, 4, 5, 6]"

ghci> stringOfList (\x -> x) ["foo"]
"[foo]"

ghci> stringOfList (stringOfList show) [[1, 2, 3], [4, 5], [6], []]
"[[1, 2, 3], [4, 5], [6], []]"
```

18. Write functions called clone, padZero and removeZero

```
clone :: a -> Int -> [a]
```

such that clone x n returns a list of n copies of the value x. If the integer n is 0 or negative, then cloneshould return the empty list. You should get the following behavior:

```
ghci> clone 3 5
[3, 3, 3, 3, 3]

ghci> clone "foo" 2
["foo", "foo"]
```

Use clone to write a function,

```
padZero :: [Int] -> [Int] -> ([Int], [Int])
```

which takes two lists: [x1,...,xn] [y1,...,ym] and adds zeros in front of the *shorter* list to make the list lengths equal. Your implementation should **not** be recursive.

You should get the following behavior:

```
ghci> padZero [9, 9] [1, 0, 0, 2]
([0, 0, 9, 9], [1, 0, 0, 2])

ghci> padZero [1, 0, 0, 2] [9, 9]
([1, 0, 0, 2], [0, 0, 9, 9])
```

Next, write a function

```
removeZero :: [Int] -> [Int]
```

that takes a list and removes a prefix of leading zeros, yielding the following behavior:

```
ghci> removeZero [0, 0, 0, 1, 0, 0, 2]
[1, 0, 0, 2]

ghci> removeZero [9, 9]
[9, 9]

ghci> removeZero [0, 0, 0, 0]
[]
```

19. BigInt - The Haskell type Int only contains values up to a certain size. So lets create a new type called BigInt to hold larger numbers.

Let us use the list [d1, d2, ..., dn], where each di is between 0 and 9, to represent the (positive) **big-integer** d1d2...dn.

type BigInt = [Int]

For example, [9, 9, 9, 9, 9, 9, 9, 9, 9, 8] represents the big-integer 9999999998.

Fill out the implementation for

```
bigAdd :: BigInt -> BigInt -> BigInt
```

so that it takes two integer lists, where each integer is between 0 and 9 and returns the list corresponding to the addition of the two big-integers.

You should get the following behavior:

```
ghci> bigAdd [9, 9] [1, 0, 0, 2]
[1, 1, 0, 1]

ghci> bigAdd [9, 9, 9, 9] [9, 9, 9]
[1, 0, 9, 9, 8]
```

Next you will write functions to multiply two big integers. First write a function

```
mulByDigit :: Int -> BigInt -> BigInt
```

which takes an integer digit and a big integer, and returns the big integer list which is the result of multiplying the big integer with the digit. You should get the following behavior:

```
ghci> mulByDigit 9 [9,9,9,9]
[8,9,9,9,1]
```

Now, using `mulByDigit`, fill in the implementation of

```
bigMul :: BigInt -> BigInt -> BigInt
```

Once you are done, you should get the following behavior

```
ghci> bigMul [9,9,9,9] [9,9,9,9]
[9,9,9,8,0,0,0,1]

ghci> bigMul [9,9,9,9,9] [9,9,9,9,9]
[9,9,9,9,8,0,0,0,0,1]
```