CIS 623 Structured programming & formal methods

Spring 2020, Final Examination

Instructions:

1. This test is a take home test. It is originally designed so that a student should be able to answer all the questions in two hours or less when conducted in a regular classroom setting. I anticipate that each student should be able to prepare the submission as an electronic document and submit it within half an hour right after the examination. Due to the time zone issue, students will be given a 4-hour time slot. By doing so, we should be able to accommodate students that are currently residing in other physical locations.
2. **Important note**: *You need to attempt all of the seven questions. Some questions are divided into small parts and the parts are usually related. Please spend sufficient time to study the definitions given in each question carefully. Although the questions looks longer, the actual code you need to write is very short. Most of the required functions is within 4 lines. I expect the student to spend about 15 -18 minutes per question. For each question, I will advise you to spend half of the time to study the definitions given, use papers to sketch/outline your ideas and plans before completing and test the code. Once you pin down the idea, you may need less than five minutes to complete the code for each question.*
3. You are expected to work on the final by yourself only.
4. The test is open book and open notes. You may consult the course notes and the course text (Bird's and Hutton's text) while you are working on the problems. However, using other sources are discouraged because concepts may be defined differently and it may cause unnecessary confusion.
5. The time slot is set up to be 5:15 pm, 5-5-2020 to 9:15 pm, 5-5-2020 (Syracuse time). The examination paper will be released at 5:15 pm, 5-5-2020.
6. Attempt all questions in the test. Partial credits may be given to incomplete answers.
7. You may

   (1) Write the solution to each question on a separate piece of paper. Print your name and the question number at the top of each page, or

   (2) Put the answers to each question in a dot hs file name `final-sol.hs`.
8. After completing the test, please fill in the **required cover sheet**.
9. **Submission:** After completing the examination, complete the (1) cover sheet (provided separately), (2) all of your answers in a folder named `<your-full-name>` and zip it. For example, if I am the student, the zip folder will be named `Andrew-Lee.zip`.
10. Write clearly. Illegible answers will not be graded and will not receive any credits.

Question 1. Recursion (12 point)

Answer each part of this question by using recursion. Do not use list comprehension nor higher order functions.

a. (7 point) Write, by using recursion, a Haskell function

```
keep    :: [a] -> [a]
```

such that it will take a list of type `a` as input, returns a list created by keeping *only* the elements at the positions

$$1, 3, 5, \ldots, (2n+1), \ldots$$

of the input list, whenever possible. For example:

```
*Main> keep []
[]
*Main> keep ['a']
"a"
*Main> keep ['a','b']
"a"
*Main> keep ['a','b','c']
"ac"
*Main> keep ['a','b','c','d']
"ac"
*Main> keep ['a','b','c','d','e']
"ace"
```

b. (5 point) By using the `keep` function, write a Haskell function `keepLess` such that it will take a list of type `a` as input, returns a list created by keeping *only* the elements at the positions

$$1, 5, 9, \ldots, (4n+1), \ldots$$

of the input list, whenever possible. For example:

```
*Main> keepLess []
[]
*Main> keepLess ['a']
"a"
*Main> keepLess ['a','b']
"a"
*Main> keepLess ['a','b','c']
"a"
*Main> keepLess ['a','b','c','d']
"a"
*Main> keepLess ['a','b','c','d','e']
"ae"
```

Question 2. List comprehension (20 point)

Answer each part of this question by using list comprehension. Do not use recursion nor higher order functions.

We use the following definitions in this question. Let k and n be positive integers. (1). The integer k a factor of n if $n = k\,r$ for some positive integer `r`. (2). We say n is a prime number if $n \geq 2$ and the only factors of n are 1 and n, and, (3). We say that n is a *pseudoprime* if n is product of two primes (which are not necessarily distinct). For example, 4 is a pseudoprime because $4 = 2 \times 2$ and 2 is a prime.

a. (4 point) Write a Haskell function:

```
factors :: Integer -> [Integer]
```

such that, if `n` is a positive integer, `factors n` will return the list of *distinct* factors for the positive integer `n`. For example,

```
*Main> factors 4
[1,2,4]
*Main> factors 5
[1,5]
```

b. (8 point) Write a Haskell function:

```
pseudoprime :: Integer -> [Integer]
```

such that, if `k` is a positive integer, `pseudoprime k` will return a list of distinct integers, each of them is a pseudoprime, and, it is less than or equal to k. It is Okay if the list returned is not in sorted order. For example:

```
*Main> pseudoprime 4
[4]
*Main> pseudoprime 10
[4,6,10,9]
 -- This answer is okay even when it is not sorted
```

c. (8 point) By using the `zip` function, write a Haskell function `selectRange`:

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

such that, it will take two integers `i, j` and a list `lst` as input, where

$$\texttt{lst} = [a_1, \ldots, a_i, \ldots, a_j, \ldots, a_n].$$

The function will return the list

$$[a_i, \ldots, a_j].$$

You may assume that the function will return the empty list whenever the conditions: $1 \leq i \leq j \leq l$ fails. Here, l is the length of the input list `lst`. Example:

```
*Main> selectRange 5 3 [1..10]
[]
*Main> selectRange 3 5 [1..10]
[3,4,5]
```

Question 3 Higher order functions (14 point)

a. (8 point) By using the high order function `map`, write a Haskell function `mapThem`:

```
mapThem ::  [a->a] -> [a] -> [a]
```

such that `mapThem` will take a list of functions `fs` (of type `[a -> a]`) and a list `xs` (of type `[a]`) as input, return a list `ys` (of type `[a]`) which is defined, in mathematics notation:

$$\text{If} \quad \texttt{fs} = \{f_1, \ldots, f_n\} \text{ and } \texttt{xs} = \{x_1, \ldots, x_k\}, \text{ then } \texttt{ys} = \{y_1, \ldots, y_k\} \text{ where}$$

$$y_i = f_n(\ldots(f_2(f_1(x_i)))), \quad \text{for } (1 \leq i \leq k)$$

For example, if `f1=(+2)` and `f2=(*3)` then

```
*Main> mapThem [(+2), (*3)] [1,2,3,4]
[9,12,15,18]
*Main> mapThem [(*3), (+2)] [1,2,3,4]
[5,8,11,14]
```

b. (6 point) Let `combine` be the function:

```
combine :: Num a => (a -> a) -> (a -> a) -> (a -> a)
```

where, given `a`, which is a member of the type class `Num` and f, g that are functions of type `a -> a`, the function returns a *function h* of the type `a -> a` so that, in mathematical notation:

$$h(x) = 2 \cdot f(x) + 3 \cdot g(x).$$

For example, if f is the function $f(x) = x + 1$ and g is the function $g(x) = x + 3$, then combine will return the *function* $h = 2(x + 1) + 3(x + 3)$.

Complete the implementation of the Haskell function `combine` by filling in the blanks below:

```
combine :: Num a => (a -> a) -> (a -> a) -> (a -> a)
combine f g =                  --  < Fill in the blanks here
```

Question 4 Recursive data types (18 point)

Let `Tree a` be the following data type:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
            deriving (Show)
```

We say that such a tree *T* is *almost balanced* if, the number of nodes in the left and right subtree of every node differs by at most two. Note that for the Empty tree and the trees with a single node, they are considered as *almost balanced*.

a. (4 point) Following the given definition of Tree, define, in the Haskell language, two trees `tree1` and `tree2` of type `Tree Int` such that `tree1` is almost balanced but `tree2` is not almost balanced. Give a concise explanation why `tree1` is almost balanced but `tree2` is not.

b. (6 point) Write the Haskell function `noOfNodes`

```
noOfNodes :: Tree a -> Int
```

which will take a tree `t` (type `Tree a`) as input, return the number of nodes of `t`.

c. (8 point) By using the function `noOfNodes`, write the Haskell function `abalanced`:

```
abalanced :: Tree a -> Bool
```

which will take a Tree `t` as input, returns `True` if `t` is almost balanced and `False` if otherwise.

Question 5 The foldr operator and its variants (10 point)

Recall that the recursion operator `foldr` is defined as follows:

```
foldr        :: (a -> b -> b) -> b -> [a] -> b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

In the question, you are asked to rewrite a function for computing prefix sums using the `foldr` operator. Let $l = [x_1, x_2, x_3, \ldots, x_n]$ be a list of integers (each of type `Int`). The prefix sums of l is the list $[y_1, y_2, y_3, \ldots, y_n]$ where

$$y_1 = x_1,\ \ y_2 = x_1 + x_2,\ \ y_3 = x_1 + x_2 + x_3,\ \ldots\ , y_n = x_1 + x_2 + \ldots + x_n$$

Write a Haskell function `fun` so that, given `xs` (a list of type Int), `foldr (fun) [] xs` will compute the prefix sums of `xs`. For example,

```
*Main> foldr (fun) [] []
[]
*Main> foldr (fun) [] [1..5]
[1,3,6,10,15]
```

Question 6 IO (16 point)

Let $l = [x_1, x_2, x_3, \ldots, x_n]$ be a list of integers (each of type `Int`). The suffix sums of s is the list $[z_1, z_2, z_3, \ldots, z_n]$ where

$$z_1 = x_1 + x_2 + \ldots + x_n, \;\; z_2 = x_2 + x_3 + \ldots + x_n, \;\; z_3 = x_3 + x_4 + \ldots + x_n, \;\; \ldots \;\; , z_n = x_n.$$

a. (8 point) Write a Haskell function `suffixsum` so that, given `xs` (a list of type Int),

```
suffixsum :: [Int] -> [Int]
```

`suffixsum xs` will compute the suffix sums of `xs`. For example,

```
*Main> suffixsum [1..5]
[15,14,12,9,5]
*Main> suffixsum []
[]
```

b. (8 point) By using the `suffixsum` function in your answers to part 1, or otherwise, Write a Haskell function

```
printSufSum :: [Int] -> IO ()
```

such that

Case 1 `xs` is not empty: `printSufSum xs` will display the suffix sums of the list `xs`, one element per line.

Case 2 `xs` is empty: `printSufSum []` will NOT print anything. For example:

```
*Main> printSufSum []
*Main> printSufSum [1..4]
10
9
7
4
```

Question 7 Miscellaneous (10 point)

Consider the following definition `Prop`, which is to represent well formed formula in propositional logic.

```
data Prop = Const Bool
          | Var Char
          | Not Prop
          | And Prop Prop
          | Or Prop Prop
          | Imply Prop Prop
```

Note that the data constructors Not, And, Or and Imply represent the four logical symbols $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ respectively.

a (3 point) Define, in the Haskell language, a formula `pa` of type `Prop` which uses each of the four symbols once.

b. (7 point) Write a Haskell function `countSym`:

```
countSym :: Prop -> Int
```

which will take a formula p (type `Prop`), returns the number of logical symbols used in the formula p. For example, for the formula `p` you define in part *a*, `countSym pa` will return 4.