FUNCTIONAL PROGRAMMING

2019-2020 SPRING FINAL TAKE-HOME EXAM

This take-home exam consists of three parts. You are expected to submit the following files to Ninova:

- A Haskell source code file for Part 1.

- A Haskell source code file for Part 2.

- A Haskell source code file for Part 3.

- A report file (if necessary).

The grading weights for the individual parts are 25% for part 1, 35% for part 2, and 40% for part 3. Grading will be based on both the submitted code and the report about the code. You have to explain every part of your code in detail. You can write your explanations as inline comments in your code file, or as part of the report file. For each part, you will get a code grade between 0 and 100, and a report grade between 0 and 1. The report grade will be a factor which will be multiplied with your code grade to determine the final grade for that part. For example, a code that passes every test but has no report will get 0.

Your code will be graded using the automatic input-output checker Calico (https://calico.readthedocs.io/). YOUR CODE MUST FOLLOW THE INSTRUCTIONS EXACTLY, and you are expected to run the Calico tests yourself before submitting your code. If your operating system doesn't support Calico, you can find instructions about setting up a virtual machine here: https://github.com/itublg/itucs-vmimage

You are not allowed to use any technique that was not covered in the course. You can use standard library functions that were not covered but you will first need to explain the function in general and then explain your usage in your code. You are allowed to add helper functions for implementing the requested functions.

Part 1: Counting Sundays

Given that 1 Jan 1900 was a Monday, how many Sundays fell on the first of the month during the 20th century (1 Jan 1901 to 31 Dec 2000)?[1]

Two solutions will be considered for this problem. The solutions (along with their JavaScript implementations) are explained on the following page:

https://www.xarg.org/puzzle/project-euler/problem-19/

Python implementations for these solutions can be found under the class files section.

1. Write a function "dayOfWeek" that, given a year, a month and a day, returns which day of the week the date is. Use Zeller's congruence. Note that Haskell's type system requires explicit type conversions as in:

```
t1 = floor (fromIntegral (13 * (m' + 1)) / 5.0)
```

2. Fill in the Haskell code below to calculate the result.

```
sundays1 :: Integer -> Integer -> Integer
sundays1 start end = sundays' ? ?
  where
    sundays' :: Integer -> Integer -> Integer
    sundays' y m
      | y > end   = ?
      | otherwise = if dayOfWeek y m 1 == 1 then rest + 1 else rest
      where
        nextY = ?
        nextM = ?
        rest = ?
```

   - What does the helper function (sundays') calculate?
   - What if you don't define a "rest" and use its expression where it's needed?

3. Write a tail recursive function of "sundays1" and name it “sundays1tr”.

4. Write the "leap" and "daysInMonth" functions as given in the Python source. Using these, implement "sundays2".

5. Bring all of it together so that the following `main` function will work:

```
getFunction :: String -> (Integer -> Integer -> Integer)
getFunction name
  | name == "sundays1"   = sundays1
  | name == "sundays1tr" = sundays1tr
  | name == "sundays2"   = sundays2
  | otherwise            = error "unknown function"

main :: IO ()
main = do
 line <- getLine
 let [f, start, end] = words line
 putStrLn $ show $ (getFunction f) (read start :: Integer) (read end :: Integer)
```

1 This exercise is taken from the Project Euler site: https://projecteuler.net/problem=19

A sample run of the program should look as follows (first line is user input, second line is output):

```
sundays1tr 1901 2000
171
```

6. Name your source file *part1.hs* and make sure that the following command creates an executable (you might have to remove the module definition at the top of your source file):

```
ghc part1.hs -o part1
```

7. Run your code through the supplied Calico test file:

```
calico part1.yaml
```

8. (math question) Is the number of weeks in 400 years an integer value? In other words, is the number of days in 400 years a multiple of 7? If so, what is the possibility that a certain day of a month (such as 1 Jan, or your birthday) is a Sunday (or some other day)? Are all days equally possible?

Part 2: Custom Solitaire

This problem involves a solitaire card game invented just for this exercise.[2] You will write a program that tracks the progress of a game.

A game is played with a *card-list* and a *goal*. The player has a list of *held-cards*, initially empty. The player makes a move by either *drawing*, which means removing the first card in the card-list from the card-list and adding it to the held-cards, or *discarding*, which means choosing one of the held-cards to remove. The game ends either when the player chooses to make no more moves or when the sum of the values of the held-cards is greater than the goal.

The objective is to end the game with a low score (0 is best). Scoring works as follows: Let *sum* be the sum of the values of the held-cards. If *sum* is greater than *goal*, the *preliminary score* is three times *sum* − *goal*, else the preliminary score is *goal* − *sum*. The score is the preliminary score unless all the held-cards are the same color, in which case the score is the preliminary score divided by 2 (and rounded down as usual with integer division).

The types are defined as follows (you will probably want to derive from `Eq` and `Show`):

```
data Color = Red | Black
data Suit = Clubs | Diamonds | Hearts | Spades
data Rank = Num Int | Jack | Queen | King | Ace
data Card = Card { suit :: Suit, rank :: Rank }
data Move = Draw | Discard Card
```

1. Write a function `cardColor`, which takes a card and returns its color (spades and clubs are black, diamonds and hearts are red).
2. Write a function `cardValue`, which takes a card and returns its value (numbered cards have their number as the value, aces are 11, everything else is 10).
3. Write a function `removeCard`, which takes a list of cards `cs`, and a card *c*. It returns a list that has all the elements of `cs` except `c`. If `c` is in the list more than once, remove only the first one. If `c` is not in the list, raise an error.
4. Write a function `allSameColor`, which takes a list of cards and returns `True` if all the cards in the list are the same color and `False` otherwise.
5. Write a function `sumCards`, which takes a list of cards and returns the sum of their values. *Use a locally defined helper function that is tail recursive.*
6. Write a function `score`, which takes a card list (the held-cards) and an int (the goal) and computes the score as described above.
7. Define a type for representing the state of the game. (What items make up the state of the game?)
8. Write a function `runGame` that takes a *card list* (the card-list) a *move list* (what the player "does" at each point), and an *int* (the goal) and returns the score at the end of the game after processing (some or all of) the moves in the move list in order. Use a locally defined recursive helper function that takes the current state of the game as a parameter. As described above:
    - The game starts with the held-cards being the empty list.
    - The game ends if there are no more moves. (The player chose to stop since the *move list* is empty.)
    - If the player discards some card `c`, play continues (i.e., make a recursive call) with the held-cards not having `c` and the card-list unchanged. If `c` is not in the held-cards, raise an error.
    - If the player draws and the card-list is empty, the game is over. Else if drawing causes the sum of the held-cards to exceed the goal, the game is over. Else play continues with a larger held-cards and a smaller card-list.

2 This exercise is adapted from an online course by Prof. Dan Grossman.

In the second part, arrange the code to get the cards, the moves and the goal from the user.

9. Write a function `convertSuit` that takes a character c and returns the corresponding suit that starts with that letter. For example, if `c` is `'d'` or `'D'`, it should return `Diamonds`. If the suit is unknown, raise an error.
10. Write a function `convertRank` that takes a character c and returns the corresponding rank. For face cards, use the first letter, for "Ace" use '1' and for 10 use 't' (or 'T'). You can use the `isDigit` and `digitToInt` functions from the *Data.Char* module. If the rank is unknown, raise an error.
11. Write a function `convertCard` that takes a suit name (char) and a rank name (char), and returns a card.
12. Write a function `readCards` that will read (and return) a list of cards from the user. On each line, the user will type a card as two letters (such as "hq" for "Queen of Hearts" or "s2" for "Two of Spades"). The user will end the sequence by typing a single dot.
13. Write a function convertMove that takes a move name (char), a suit name (char), and a rank name (char) and returns a move. If the move name is `'d'` or `'D'` it should return `Draw` and ignore the other parameters. If the move is `'r'` or `'R'`, it should return `Discard c` where `c` is the card created from the suit name and the rank name.
14. Write a function `readMoves` that will read (and return) a list of moves from the user. On each line, the user will type a move as one or three letters (such as "d" for "Draw" or "rhq" for "Discard Queen of Hearts"). The user will end the sequence by typing a single dot.
15. Bring all of it together so that the following `main` function will work:

```
main = do putStrLn "Enter cards:"
 cards <- readCards
 -- putStrLn (show cards)

 putStrLn "Enter moves:"
 moves <- readMoves
 -- putStrLn (show moves)

 putStrLn "Enter goal:"
 line <- getLine

 let goal = read line :: Int

 let score = runGame cards moves goal
 putStrLn ("Score: " ++ show score)
```

A sample run of the program should look as follows:

```
Enter cards:
c1
s1
c1
s1
.
Enter moves:
d
d
d
d
d
.
Enter goal:
42
Score: 3
```

A sample run with an error:

```
Enter cards:
cj
s8
.
Enter moves:
d
rhj
.
Enter goal:
42
part2: card not in list
```

9. Name your source file *part2.hs* and make sure that the following command creates an executable:

```
ghc part2.hs -o part2
```

10. Run your code through the supplied Calico test file:

```
calico part2.yaml
```

Part 3: Anagrams

In this assignment, you will solve the combinatorial problem of finding all the anagrams of a sentence.[3]

An anagram of a word is a rearrangement of its letters such that a word with a different meaning is formed. For example, if we rearrange the letters of the word "Elvis" we can obtain the word "lives", which is one of its anagrams. In a similar way, an anagram of a sentence is a rearrangement of all the characters in the sentence such that a new sentence is formed. The new sentence consists of meaningful words, the number of which may or may not be the same as the number of words in the original sentence. For example, tan anagram of the sentence "I love you" is "You olive". In this exercise, we will consider permutations of words to be anagrams of the sentence. In the above example, "You I love" is considered to be a separate anagram. When producing anagrams we will ignore the character casing and the punctuation symbols.

Your goal is to implement a program which takes a sentence and lists all its anagrams of that sentence. You will be given a dictionary that contains all meaningful words.

The general idea is to examine the list of characters. To find the anagrams of a word, we will find all the words from the dictionary which have the same character list. To finding anagrams of a sentence, we will extract subsets of characters to see if we can form meaningful words. For the remaining characters we will solve the problem recursively and then combine the meaningful words we have found.

For example, consider the sentence "You olive". The list of characters in this sentence is "eiloouvy". We start by selecting some subset of the characters, say "i". We are left with the characters "eloouvy". Checking the dictionary we see that "i" corresponds to the word "I", so we've found one meaningful word. We now solve the problem recursively for the rest of the characters "eloouvy". This will give us a list of solutions: [["love", "you"], ["you, "love"]]. We then combine "I" with that list to obtain the sentences "I love you" and "I you love", which are both valid anagrams.

The types are as follows:

- A "word" is a string that contains lowercase and uppercase characters, but no whitespace, punctuation or other special characters.
- A "sentence" is a list of words.
- A "character count" is a mapping from characters to integer numbers. It can be represented using a list of `(Char,Int)` pairs, or the `Map` type in `Data.Map`. Examine especially the `fromList` and `fromListWith` functions. Whether the order of the entries in the mapping is significant or not is up to your design.

The steps of the assignment are outlined below:

1. Write a function `wordCharCounts` that takes a word and returns its character counts. For example, if the word is "mississippi" the result should report that there are 1 of 'm', 4 of 'i', 4 of 's', and 2 of 'p'. Try to express this function as some combination of higher-order list functions like `map` and `filter`. *Hint*: You can use prelude functions like `toLower` or `nub`.
2. Write a function `sentenceCharCounts` that takes a list and returns its character counts. Try to express it as a composition of functions.
3. Write a function `dictCharCounts` that takes a list of dictionary words and returns a mapping from words to their character counts. For example, if the dictionary contains the words "eat", "all", and "tea", this should report that the word "eat" has 1 of 'e', 1 of 'a', 1 of 't'; the word "all" contains 1 of 'a', 2 of 'l'; the word "tea" contains 1 of 't', 1 of 'e', 1 of 'a'.
4. Write a function `dictWordsByCharCounts` which takes in the result of the previous function and returns a map of words which have the same character counts. For the example above the result should map the character counts 1 of 'a', 2 of 'l' to the word list ["all"], and the character counts 1 of 'a', 1 of 'e', 1 of 't' to the word list ["tea", "eat"].
5. Write a function `wordAnagrams` which takes a word and the result of the previous function and returns a list of anagrams for the given word. For example, if the parameters are `"ate"` and the result given above, it should return the words "tea" and "eat".

3 This exercise and is text is taken from an online course by Prof. Martin Odersky with some minor changes.

6. Write a function `charCountsSubsets` that takes character counts and returns all possible subsets of these counts. For example, the character counts of the word "all" is 1 of 'a', 2 of 'l', and the subsets of the character counts should be (listed in no specific order):
   - empty
   - 1 of 'a'
   - 2 of 'l'
   - 1 of 'l'
   - 1 of 'a', 1 of 'l'
   - 1 of 'a', 2 of 'l'
7. Write a function `subtractCounts` that takes two mappings of character counts and returns a new mapping where counts in the second map are subtracted from the counts in the first map. For example, if your first map is 1 of 'a', 3 of 'e', 2 of 'l', and your second map is 1 of 'a', 1 of 'l', the result should be 3 of 'e', 1 of 'l'. You can assume that the second map is a subset of the first map.
8. Write a function `sentenceAnagrams` that takes a sentence and returns a list of sentences which are anagrams of the given sentence. Test your function on short sentences, no more than 10 characters. The search space gets huge very quickly as your sentence gets longer, so the program may run for a very long time. However, for short sentences such as "Linux rulez", "I love you", or "Mickey Mouse" the program should end fairly quickly.
9. Write a `main` function that takes a short sentence as a command line parameter and prints its anagrams. The program should generate its dictionary by reading the given dictionary text file (extracted from the zip file).
10. Name your source file *part3.hs* and make sure that the following command creates an executable:

    ```
    ghc part3.hs -o part3
    ```

    Then it should be runnable as explained above:

    ```
    ./part3 "i love you"
    ```

11. Run your code through the supplied Calico test file:

    ```
    calico part3.yaml
    ```