

FUNCTIONAL PROGRAMMING 2019-2020 SPRING FINAL TAKE-HOME EXAM

Cihat Akkiraz,150180704, Istanbul Technical University

16/06/2020

Part 1 : Counting Sundays

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: $t1 = \text{floor}(\text{fromIntegral}(13 * (m' + 1)) / 5.0)$

Listing 1: Haskell Code for Finding Day of Week

```
1 dayOfWeek :: Integer -> Integer -> Integer -> Integer
2 dayOfWeek y m d =
3     (d + t1 + k + t2 + t3 + 5 * j) 'mod' 7
4     where
5         dayOfWeek' :: Integer -> Integer -> [Integer]
6         dayOfWeek' y m
7             | m <= 2 = [y-1,m+12]
8             | otherwise = [y,m]
9         [y',m'] = dayOfWeek' y m
10        j = y' 'div' 100
11        k = y' 'mod' 100
12        t1 = floor (fromIntegral (13 * (m' + 1)) / 5.0)
13        t2 = floor (fromIntegral (k) / 4.0)
14        t3 = floor (fromIntegral (j) / 4.0)
```

This function takes day, month and year then returns day of week. I have designed this function using one helper function. 'dayOfWeek' helper function takes year and month, if month is equal or less than 2 it returns [y-1,m+12] else [y,m]. Using this helper function the new year and month is calculated. Then j,k,t1,t2,t3 are calculated using given formulas and the day of week is found with summing d,t1,t2,t3,k,5*j and making mod operations with 7.

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

Listing 2: Filled Function that Given at the homework Booklet

```
1 sundays1 :: Integer -> Integer -> Integer
2 sundays1 start end = sundays' start 1
3     where
4         sundays' :: Integer -> Integer -> Integer
5         sundays' y m
```

```

6         | y > end    = 0
7         -- If it is sunday increase rest by 1
8         | otherwise = if dayOfWeek y m 1 == 1 then rest + 1 else rest
9     where
10         nextM = (m `mod` 12) + 1
11         nextY = if m == 12 then y+1 else y
12         rest = sundays' nextY nextM

```

- What does the helper function (sundays') calculate? : Helper function takes year and month. If first day of month is sunday it returns functions itself with increasing rest by 1. Until y greater than end year, it iterates recursively. Year and month are updated using mod operation.
- What if you don't define a "rest" and use its expression where it's needed? : When we are using recursive approach we need rest. If we don't use rest, we can use tail recursive approach. We can give number of years as parameter to the function. At the end of the iterations we return directly that parameter.

3. Write a tail recursive function of "sundays1".

Listing 3: Tail Recursive Version Of Sundays1 Function

```

1 sundays1tr :: Integer -> Integer -> Integer
2 sundays1tr start end = tailRecursiveSundays' start 1 0
3     where
4         tailRecursiveSundays' :: Integer -> Integer -> Integer -> Integer
5         tailRecursiveSundays' y m rest
6             | y > end    = rest
7             | otherwise = if dayOfWeek y m 1 == 1 then ↵
8                             tailRecursiveSundays' nextY nextM (rest+1) else ↵
9                             tailRecursiveSundays' nextY nextM rest
10        where
11            nextM = (m `mod` 12) + 1
12            nextY = if m == 12 then y+1 else y

```

This function tail recursive version of above function. It takes numberofsundays as a parameter. Every iteration if conditions are valid, it increases by 1. At the final iteration the function returns that parameter(rest).

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

Listing 4: Leap

```

1 leap :: Integer -> Bool

```

```

2 leap y = ((y 'mod' 4) == 0 && (y'mod' 100) /= 0 || (y 'mod' 400) == 0)
3
4 days_in_month :: Integer -> Integer -> Integer
5 days_in_month m y
6     | m == 2 =
7         case leap(y) of
8             True  -> 29
9             False -> 28
10    | m == 4 || m == 6 || m == 9 || m == 11 = 30
11    | otherwise = 31
12
13 sundays2 :: Integer -> Integer -> Integer
14 sundays2 start end = sundays2' start 1 2 0
15     where
16         sundays2' :: Integer -> Integer -> Integer -> Integer -> Integer
17         sundays2' y m weekday n
18             | y > end = n
19             | (new_weekday 'mod' 7) == 0 = sundays2' new_year new_month ↵
20               new_weekday (n+1)
21             | otherwise = sundays2' new_year new_month new_weekday (n)
22         where
23             days = days_in_month m y
24             new_month = ((m) 'mod' 12) + 1
25             new_year = if m == 12 then y+1 else y
26             new_weekday = weekday + (days 'mod' 7)

```

'leap' function takes an integer and returns whether given conditions are true. 'days_in_month' function takes month and year than returns there are how many days in that month. 'sundays2' function makes same thing with sunday1 function. I have used tail recursive approach for this function.

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

Listing 5: Given Main Code In the Homework

```

1 getFunction :: String -> (Integer -> Integer -> Integer)
2 getFunction name
3     | name == "sundays1"    = sundays1
4     | name == "sundays1tr" = sundays1tr
5     | name == "sundays2"   = sundays2
6     | otherwise             = error "unknown function"
7
8 main :: IO ()
9 main = do
10 line <- getLine
11 let [f, start, end] = words line

```

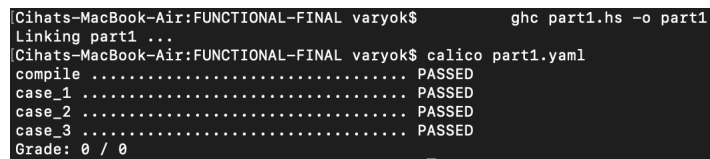
```
12 putStrLn $ show $ (getFunction f) (read start :: Integer) (read end :: Integer)
```

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
```



```
Cihats-MacBook-Air:FUNCTIONAL-FINAL varyok$ ghc part1.hs -o part1
Linking part1 ...
Cihats-MacBook-Air:FUNCTIONAL-FINAL varyok$ calico part1.yaml
compile ..... PASSED
case_1 ..... PASSED
case_2 ..... PASSED
case_3 ..... PASSED
Grade: 0 / 0
```

Figure 1: Result of Calico

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?

There are 366 or 365 days in a year. If year 'mod' 4 is equal zero there are 365 days else 366 days. In 400 years there are $300 \cdot 366 + 100 \cdot 365$ days. So, there are 146300 days. 146300 'mod' 7 is equal to zero. The possibility of a certain day of a month is a Sunday is $1/7$. All days are equally possible.

Part 2: Custom Solitaire

Listing 6: Haskell Code for Defining Data Types

```
1 data Color = Red | Black deriving (Eq, Show)
2 data Suit = Clubs | Diamonds | Hearts | Spades deriving (Eq, Show)
3 data Rank = Num Int | Jack | Queen | King | Ace deriving (Eq, Show)
4 data Card = Card { suit :: Suit, rank :: Rank } deriving (Eq, Show)
5 data Move = Draw | Discard Card deriving (Eq, Show)
```

1. Write a function `cardColor`, which takes a card and returns its color (spades and clubs are black, diamonds and hearts are red).

Listing 7: Haskell Code for Finding Color of The Card

```
1 cardColor :: Card -> Color
2 cardColor crd
3   | ((suit crd == Clubs) || (suit crd == Spades)) = Black
4   | ((suit crd == Diamonds) || (suit crd == Hearts)) = Red
```

This function returns color of the card according to suit of the card. I have designed using guards. If suit of the card is Clubs or Spades it returns Black. If suit the card is Diamonds or Hearts it returns 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, If rank is number it returns numbers itself, everything else is 10.).

Listing 8: Haskell Code for Finding Value of The Card

```
1 cardValue :: Card -> Int
2 cardValue crd
3   | (rank crd == Ace) = 11
4   | (rank crd == King) = 10
5   | (rank crd == Queen) = 10
6   | (rank crd == Jack) = 10
7   | otherwise = cardValue' (rank crd)
8   where
9       cardValue' :: Rank -> Int
10      cardValue' (Num i) = i
```

This function returns value of the card according to rank of the card. I have designed using guards. If rank of the card is Ace it returns 11 otherwise it returns 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.

Listing 9: Haskell Code for Removing Card from the List

```
1 removeCard :: [Card] -> Card -> [Card]
2 removeCard cs c
3     -- If the element is in the list
4     | ((filter (\x -> x/=c) cs) /= cs) = (filter (\x -> x/=c) cs)
5     -- If the element is not in the list
6     | otherwise = []
```

This function removes the given card from the given list. Then returns new list. I have designed using guards. If the element is in the list (First case -> Exp: initial list not equals new list) it returns updated list. Otherwise it returns empty list to show error end of the program.

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.

Listing 10: Haskell Code for Finding Whether All Cards Have Same Color

```
1 allSameColor :: [Card] -> Bool
2 allSameColor (c:cs)
3     | length(c:cs) <= 1 = True -- Final Iteration
4     | otherwise = (cardColor c == cardColor (head cs)) && allSameColor cs
```

This function returns whether cards of the given card list have same color. I have designed using guards. If list has less than 1 element it returns `True`, otherwise recursively call function to control colors of all elements.

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.

Listing 11: Haskell Code for Calculating Sum of Ranks of Cards

```
1 sumCards :: [Card] -> Int
2 sumCards cs = sumOfRanks cs 0
3     where
4         sumOfRanks :: [Card] -> Int -> Int
5         sumOfRanks [] total = total
6         sumOfRanks cardList total
7             | length cardList == 0 = total -- Last Iteration
8             | otherwise = sumOfRanks (tail cardList) (total + (cardValue <->
9                 (head cardList)))
```

This function returns sum of values of the given card list. I have create one helper function 'sumOfRanks'. 'sumOfRanks' function calculates total value using tail recursive approach.

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.

Listing 12: Haskell Code for Calculating Total Score of the Game

```
1 score :: [Card] -> Int -> Int
2 score [] _ = -1
3 score crdList goal
4   | sumOfTheValues > goal = -- If sum greater than goal
5       let preliminary_score = 3*(sumOfTheValues-goal)
6       in calculateRealScore crdList preliminary_score
7   | otherwise =
8       let preliminary_score = (goal - sumOfTheValues)
9       in calculateRealScore crdList preliminary_score
10  where
11      sumOfTheValues = sumCards crdList
12      calculateRealScore :: [Card] -> Int -> Int
13      calculateRealScore crdList preliminary_score
14          | not (allSameColor crdList) = preliminary_score
15          -- If all the cards have same color, divide preliminary score ↵
16              by 2.
17          | otherwise = preliminary_score `div` 2
```

This function returns score of the game according to goal and values of held-cards. I have used guards and two helper functions. If there is no held_card it returns -1. Why there is no held cards because one of the moves of player asked to remove the card she/he does not have. Then removecard function returns empty list and score returns -1 to notify main program.

'calculateRealScore' helper function controls whether held cards have same colors. If they have same colors it divides preliminary score by 2 else it returns preliminary score. This is real score.

'sumOfTheValues' helper function add values of the cards. It iterates using tail recursive approach.

Totalscore is calculated according to given cases.

7. Define a type for representing the state of the game. (What items make up the state of the game?)

Listing 13: Haskell Code for Defining Game State Types

```
1 data State = Start | End | WrongCard deriving (Eq, Show)
```

This types represents state of the game. If the state of the game is 'Start' -> Game continues.

else if the state of the game is 'End' -> Game overs. else if the state of the game is 'WrongCard'
-> Player asked to remove the card he/she does not have.

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:

Listing 14: Haskell Code for Running Game

```
1 runGame :: [Card] -> [Move] -> Int -> Int
2 runGame cardList moveList goal =
3     let heldCards = []
4     in score (runGame' Start heldCards cardList moveList) goal
5     where
6         runGame' :: State -> [Card] -> [Card] -> [Move] -> [Card]
7         runGame' WrongCard _ _ _ = [] -- Wrongcard error
8         runGame' End held_cards _ _ = held_cards -- End
9         runGame' _ held_cards _ [] = held_cards -- There is no move
10        runGame' Start held_cards cardList moveList
11            -- End Condition
12            | ((sumCards held_cards) > goal) = runGame' End held_cards <-
13                cardList moveList
14            -- If Move is Draw and there is no enough card.
15            | (head moveList == Draw && (length cardList) == 0) = runGame' <-
16                End held_cards cardList moveList -- If
17            -- If Move is Draw, add first card of cardList to held cards
18            | (head moveList == Draw) =
19                let l = head cardList
20                in runGame' Start (held_cards++[l]) (tail cardList) (tail <-
21                    moveList)
22            -- One of two conditions below are for case of Discard move.
23            | length (removeCard held_cards (discardCard (head moveList))) <-
24                == 0 =
25                let l = discardCard (head moveList)
26                in runGame' (WrongCard) (removeCard held_cards l) (<-
27                    cardList) (tail moveList)
28            | otherwise =
29                let l = discardCard (head moveList)
30                in runGame' (Start) (removeCard held_cards l) (cardList) (<-
31                    tail moveList)
32        where
33            discardCard :: Move -> Card
34            discardCard (Discard crd) = crd
```

This function gets heldcardLists, movelist and goal. Then returns score of the game. I have

used one helper function. 'runGame' function gets state, cardList, heldcardLists and movelist. It returns heldCardList.

If the state of the game is wrongCard it returns empty list to notify main program about error.

If the state of the game is End it returns heldCards to calculate score of the game.

If the current move is Draw and there is no card in the card list, the function overs the game to calling itself with 'End' status.

If the current move is Draw and there are cards in the card list, the head of the card list is added to held card and the function calls itself with updatedcardlists and updated movelists.

If the game are not okay for above conditions, it means the next move of the player is Discard.

If the card that will be removed is not in the card list, the function calls itself with 'wrongCard' status. And game overs.

If the card that will be removed is in the card list, the function removes card and, calls itself with updatedcard lists and updated movelists.

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.

Listing 15: Haskell Code for Converting Char to Suit

```
1 convertSuit :: Char -> Suit
2 convertSuit c
3     | c == 'd' || c == 'D' = Diamonds
4     | c == 'h' || c == 'H' = Hearts
5     | c == 'c' || c == 'C' = Clubs
6     | c == 's' || c == 'S' = Spades
7     | otherwise = error "Invalid Suit"
```

This function gets a char and returns Suit according to that char. The char should be first letter of one of the suits. Otherwise it raises 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.

Listing 16: Haskell Code for Converting Char to Rank

```
1 convertRank :: Char -> Rank
2 convertRank c
3     | c == '1' = Ace
4     | c == 't' || c == 'T' = Num 10
5     | c == 'J' || c == 'j' = Jack
6     | c == 'Q' || c == 'q' = Queen
```

```

7      | c == 'K' || c == 'k' = King
8      | isDigit c = Num (digitToInt c)
9      | otherwise = error "Invalid Rank"

```

This function gets a char and returns Rank according to that char. If the char is '1' it returns Ace, if the car is 't' or 'T' it returns 10. If the char is 'j','J' or 'q','Q' or 'k','K', it returns 'Jack','Queen','King' according to first letter of ranks. It the char is digit it returns as integer. Otherwise it raises error.

11. Write a function `convertCard` that takes a suit name (char) and a rank name (char), and returns a card.

Listing 17: Haskell Code for Finding Day of Week

```

1  convertCard :: Char -> Char -> Card
2  convertCard suitName rankName = Card {rank = convertRank rankName, suit = ↵
    convertSuit suitName}

```

This function gets two char(suitName,rankName) and returns new card. This cards is created converting rankName to rank and converting suitName to suit.

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.

Listing 18: Haskell Code for Reading Cards from Player

```

1  readCards :: IO([Card])
2  readCards = readCards' []
3      where
4          readCards' :: [Card] -> IO([Card])
5          readCards' cardList = do
6              input <- getLine
7              case (length input) == 2 of
8                  True -> do
9                      let s = head input -- Suit
10                         r = head (tail input) -- Rank
11                         new_card = (convertCard s r)
12                         readCards' (cardList ++ [new_card]) -- Add Card to ↵
13                         CardList
14                  False -> do
15                      case (input == ".") of
16                          True -> do
17                              return(cardList) -- Finish reading cards
18                          False -> do
19                              putStrLn("Invalid Card, Enter Again")

```

This function read cards from the player and saves them to list. I have used one helper function. 'readCards' helper function getline from the user. If length of the line is 2, that means player will enter a information about card. First letter - suitName, second letter - rankName. Card is created according to these. The card is added to list and the function makes call to the functions itself to get new input from the player. If length of the line is 1, the function controls whether it is '.'. If it is '.' the function returns cardList. Otherwise the function asked to enter again

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.

Listing 19: Haskell Code for Converting Inputs to Move

```

1 convertMove :: Char -> Char -> Char -> Move
2 convertMove moveName suitName rankName
3     | (moveName == 'd') || (moveName == 'D') = Draw
4     | (moveName == 'r') || (moveName == 'R') = Discard (convertCard <-
        suitName rankName)

```

This function gets moveName,suitName and rankName as chars. Then returns move. if the type of move(moveName) is 'd' or 'D' it returns Draw. if the type of move(moveName) is 'r' or 'R' it returns Discard with creating new card.

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.

Listing 20: Haskell Code for Reading Moves from Player

```

1 readMoves :: IO([Move])
2 readMoves = readMoves' []
3     where
4         readMoves' :: [Move] -> IO([Move])
5         readMoves' moveList = do
6             input <- getLine
7             case (length input) == 3 of
8                 True -> do
9                     let m = head input -- Move
10                        s = head(tail input) -- Suit
11                        r = head(tail (tail input)) -- Rank
12                        new_move = (convertMove m s r)
13                        readMoves' (moveList ++ [new_move]) -- Add Move to <-

```

```

MoveList
14     False -> do
15         case ((input == "d") || (input == "D")) of
16             True -> do
17                 let new_move = convertMove 'd' 'd' 'd' -- Add ↵
18                     Draw
19                 readMoves' (moveList ++ [new_move]) -- Add Move↵
20                     to MoveList
21             False -> do
22                 case (input == ".") of
23                     True -> do
24                         return(moveList) -- Finish reading ↵
25                         cards
26                     False -> do
27                         putStrLn("Invalid Card, Enter Again")
28                         readMoves' moveList

```

This function read moves from the player and saves them to list. I have used one helper function. 'readMoves' helper function getline from the user. If length of the line is 3, that means player will enter a information about card. First letter - moveName, second letter - suitName, third letter rankName. Move is created according to these. The move is added to list and the function makes call to the functions itself to get new input from the player. If length of the line is 1, the function controls whether it is . or 'd','D'. If it is '.' the function returns moveList. If it is 'd' or 'D' it returns new Move(Draw) and make call to the functions itself to get new input from the player.

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

Listing 21: Haskell Code for Main Program

```

1  main = do
2      putStrLn "Enter cards:"
3      cards <- readCards
4      --putStrLn (show cards)
5
6      putStrLn "Enter moves:"
7      moves <- readMoves
8      --putStrLn (show moves)
9
10     putStrLn "Enter goal:"
11     line <- getLine
12
13     let goal = read line :: Integer
14     let score = runGame cards moves goal
15     case score == -1 of
16         True -> error ("part2: card not in list")

```

This function main function, readCards,moveCards, functions are called. Then Goal is gotten as an input. Then game starts, the result of the runGame function is score. If score is -1, player asked to remove the card that she/he does not have. So one of the moves are wrong. If score is another thing, it returns the score of the play.

16. Name your source file part2.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 part2.hs -o part2`

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

`calico part2.yaml`

```
Cihats-MacBook-Air:FUNCTIONAL-FINAL varyok$ ghc part2.hs -o part2
Linking part2 ...
Cihats-MacBook-Air:FUNCTIONAL-FINAL varyok$ calico part2.yaml
compile ..... PASSED
case_1 ..... PASSED
case_2 ..... PASSED
Grade: 0 / 0
```

Figure 2: Result of Calico

Part 3: Anagrams

Listing 22: Haskell Code for Defining Data Types and Empty Lists

```
1 type CharacterCount = (Char,Int)
2 type DictionaryWordCharacterCount = ([Char],[CharacterCount])
3 type DictionaryCharacterCountWord = ([CharacterCount],[[Char]])
4
5 character_count_list :: [CharacterCount]
6 character_count_list = []
7
8 dict_word_char_counts :: [DictionaryWordCharacterCount]
9 dict_word_char_counts = []
10
11 dict_char_counts_word :: [DictionaryCharacterCountWord]
12 dict_char_counts_word = []
```

In this question three custom types are created. 'CharacterCount' type is mapping from characters to integers. 'DictionaryWordCharacterCount' type is mapping from word to CharacterCount. 'DictionaryCharacterCountWord' type is mapping from CharacterCount to word. Empty lists are created for three of them for future use.

0. In this problem data map module is not used and custom types are used. Therefore, functions are written for inserting,union,sorting etc.

Listing 23: Haskell Code for Inserting Character Counts to Character Counts List

```
1 insertCharacterCountList :: (CharacterCount) -> [CharacterCount] -> [↔
    CharacterCount]
2 insertCharacterCountList character_count character_count_list
3   | length (filter (\x -> (fst x) == fst character_count) ↔
    character_count_list) /= 0 =
4     let current_count = snd (head(filter (\x -> (fst x) == fst ↔
    character_count) character_count_list))
5     new_count = (snd character_count) + current_count
6     list_without_that_char = filter (\x -> (fst x) /= (fst ↔
    character_count)) character_count_list
7     new_list = list_without_that_char ++ [(fst character_count,↔
    new_count)]
8     in new_list
9   | otherwise = character_count_list ++ [character_count]
```

This function inserts CharacterCount(Char,Int) to CharacterCountList. It uses tail recursive approach. If the character is not in the CharacterCountList it adds directly. But if character is in the CharacterCountList the function should be update count only. So firstly, finds current count

and adds it new element's count. Deletes old element's from list and adds new element to the list.

Listing 24: Haskell Code for Union of Two Character Count List

```
1 unionTwoCharacterCountList :: [CharacterCount] -> [CharacterCount] -> [↔
    CharacterCount]
2 unionTwoCharacterCountList list1 list2
3     | length list1 == 0 = list2
4     | otherwise = unionTwoCharacterCountList (tail list1) (↔
    insertCharacterCountList (head list1) list2)
```

This function adds the first list to the second list and finally returns list2. In every iteration, inserts head of list1 to list2 using insertCharacterCountList function.

Listing 25: Haskell Code for Quick Sort

```
1 quicksort :: (Ord a) => [a] -> [a]
2 quicksort [] = []
3 quicksort (x:xs) = (quicksort [a|a<-xs, a<=x]) ++ [x] ++ (quicksort [a|a<-↔
    xs, a>x])
```

Listing 26: Haskell Code for Converting (Char to Character Count List) to (Character Count to [

```
1 convertCtoWDict :: ([Char],[Character_count_ls]) -> ([Character_count_ls↔
    ],[[Char]])
2 convertCtoWDict element = (snd element,[fst element])
```

This function takes one DictionaryWordCharacterCount and returns one DictionaryCharacterCountWord changing place of first element and place of second element.

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.

Listing 27: Haskell Code for Finding Character Counts of Word

```
1 wordCharCounts :: [Char] -> [CharacterCount]
2 wordCharCounts w = wordCharCounts' w character_count_list
3     where
```

```

4      wordCharCounts' :: [Char] -> [CharacterCount] -> [CharacterCount]
5      wordCharCounts' word character_count
6          | length word == 0 = quicksort character_count -- Final ↵
              Iteration
7          | otherwise =
8              let (reduced_word,new_character_list) = ↵
                  calculateAndRemoveChar' (map toLower word) ↵
                  character_count
9              in wordCharCounts' reduced_word new_character_list
10     calculateAndRemoveChar' :: [Char] -> [CharacterCount] -> ↵
        DictionaryWordCharacterCount
11     calculateAndRemoveChar' word character_count_list =
12         let hd = head word -- First character
13             num_of_c = length (filter (==hd) word) -- number of repeat↵
                  in string
14         in ((filter (/=hd) word), insertCharacterCountList (hd,↵
            num_of_c) character_count_list)

```

```

Cihats-MacBook-Air:FUNCTIONAL-FINAL varyok$ ghci part3.hs
GHCi, version 8.8.3: https://www.haskell.org/ghc/ :? for help
[1 of 1] Compiling Main                ( part3.hs, interpreted )
Ok, one module loaded.
*Main> wordCharCounts "Entrepreneur"
[( 'e',4), ( 'n',2), ( 'p',1), ( 'r',3), ( 't',1), ( 'u',1)]

```

Figure 3: Part 3-1 Result

This function gets a string and returns its character counts. I have created 2 helper functions. 'wordCharCounts' helper function takes word and empty list, it iterates using tail recursive approach and returns character counts. If it is final iteration, the function returns sorted list to future use. 'calculateAndRemoveChar' helper function takes word and character count list. Controls first letter of word. It finds out how many times repeated first letter and deletes all repeats and itself. Then add char and number of repeats to character count list. It returns reduced word and new character count list.

2. Write a function sentenceCharCounts that takes a list and returns its character counts. Try to express it as a composition of functions.

Listing 28: Haskell Code for Finding Character Counts of Sentence

```

1 sentenceCharCounts :: [[Char]] -> [CharacterCount]
2 sentenceCharCounts all_sentences = sentenceCharCounts' all_sentences ↵
    character_count_list
3     where
4         sentenceCharCounts' :: [[Char]] -> [CharacterCount] -> [↵
            CharacterCount]
5         sentenceCharCounts' word_list character_list

```



```

6      | length(word_list) == 0 = quicksort character_list -- Final ↵
      | Iteration
7      -- Add characterCount list of first word of wordlist to ↵
      | charactercountlist.
8      | otherwise = sentenceCharCounts' (tail word_list) (↵
      | unionTwoCharacterCountList character_list (wordCharCounts ↵
      | (head word_list)))

```

```

*Main> sentenceCharCounts ["The", "secret", "of", "getting", "ahead", "is", "getting", "started"]
[('a',3),('c',1),('d',2),('e',7),('f',1),('g',4),('h',2),('i',3),('n',2),('o',1),('r',2),('s',3),('t',8)]

```

Figure 4: Part 3-2 Result

This function takes sentence(set of words) and returns CharacterCount list. 'sentenceCharCounts' helper function takes sentence and empty charactercount list then returns charactercount list. It uses tail recursive approach. If it is final iteration, the function returns sorted list to future use. This function calls unionTwoCharacterCountList and merges new character list and total character list.

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'.

Listing 29: Haskell Code for Finding dictionary that Word to Character Counts

```

1 dictCharCounts :: [[Char]] -> [DictionaryWordCharacterCount]
2 dictCharCounts word_list = dictCharCounts' word_list dict_word_char_counts
3   where
4     dictCharCounts' :: [[Char]] -> [DictionaryWordCharacterCount] -> ↵
      [DictionaryWordCharacterCount]
5     dictCharCounts' word_list dict_character_list
6     | length word_list == 0 = quicksort dict_character_list -- ↵
      | Final Iteration
7     -- Add first word of word list and its charactercount to ↵
      | dictionary.
8     | otherwise =
9       let word = head word_list
10        character_list = wordCharCounts word
11        in dictCharCounts' (tail word_list) (↵
        insertCharacterCountListAndWordToDict word ↵
        character_list (dict_character_list))
12    -- Inserting new element to dictionary
13    insertCharacterCountListAndWordToDict :: [Char] -> [CharacterCount↵
      ] -> [DictionaryWordCharacterCount] -> [↵
      DictionaryWordCharacterCount]

```

```

14     insertCharacterCountListAndWordToDict word <-
        new_character_count_list dictionary =
15         let new_element = (word,new_character_count_list)
16         new_dictionary = dictionary ++ [new_element]
17         in new_dictionary

```

```

*Main> dictCharCounts ["eat","all","tea"]
[("all",[( 'a',1),('l',2)]),("eat",[( 'a',1),('e',1),('t',1)]),("tea",[( 'a',1),('e',1),('t',1)])]

```

Figure 5: Part 3-3 Result

This function takes sentence(set of words) and returns DictionaryWordCharacterCount list. dictCharCounts' helper function takes sentence and empty list. It iterates using tail recursive approach. Except final iteration, it looks first word of word list(every iteration it reduces) and find character counts of that word. 'insertCharacterCountListAndWordToDict' helper function inserts new element to dictionary and returns new dictionary.

4. 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'.

Listing 30: Haskell Code for Finding dictionary that Character Counts to Word

```

1 dictWordsByCharCounts :: [DictionaryWordCharacterCount] -> [DictionaryCharacterCountWord]
2 dictWordsByCharCounts dictWordToChar = dictWordsByCharCounts' dictWordToChar
3   where
4     dictWordsByCharCounts' :: [DictionaryWordCharacterCount] -> [DictionaryCharacterCountWord]
5     dictWordsByCharCounts' dictWordToChar = mergeSameThings (map convertCtoWDict dictWordToChar)
6     -- Merging same elements(have same characterCount) on the dictionary.
7     mergeSameThings :: [DictionaryCharacterCountWord] -> [DictionaryCharacterCountWord]
8     mergeSameThings dictCharToWord dictUnique
9       | length dictCharToWord == 0 = dictUnique -- Final Iteration
10      -- If there is more than one word for one countList in the dictionary.
11      | length (filter (\x -> (fst x) == (fst (head dictCharToWord))) dictCharToWord) /= 1 =
12          let element = head dictCharToWord
13          hs = filter (\x -> (fst x) == (fst element))

```

```

dictCharToWord
14 list_without_that_elements = filter (\x -> (fst x) /= ←
    (fst element)) dictUnique
15 new_word_list = (map convertWords hs)
16 new_unique_dict = list_without_that_elements ++ [(fst ←
    element,new_word_list)]
17 in mergeSameThings (list_without_that_elements) ←
    new_unique_dict
18 | otherwise = mergeSameThings (tail dictCharToWord) dictUnique

```

```

*Main> dictWordsByCharCounts (dictCharCounts ["eat","all","tea"])
[[('a',1),('l',2)],["all"]],[(('a',1),('e',1),('t',1)),["eat","tea"]]]

```

Figure 6: Part 3-4 Result

This function takes dictionary(Word to characterCount) and converts it new dictionary(CharacterCount to Word) dictWordsByCharCounts' function firstly changes order of word and character count using convertCtoWDict function. Then make calls to mergeSameThings helper function to merge words that have same character counts. Until this time we done other parts as sorted because to do this part easily. MergeSameThings function uses tail recursive approach. In every iteration it finds words that have same character counts, then removes them from list and merges that word. Then it adds merged elements to list.

5. 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’.

Listing 31: Haskell Code for Finding Anagram of Given Word

```

1 wordAnagrams :: [Char] -> [DictionaryCharacterCountWord] -> [[Char]]
2 wordAnagrams word dictionary =
3     let wordDictionary = wordCharCounts word
4         -- All words that has given wordCharCounts in the dictionary.
5         samePatternWords = filter (\x -> (fst x) == wordDictionary) ←
            dictionary
6         all_words = snd (head samePatternWords)
7     in all_words

```

```

*Main> wordAnagrams "ate" (dictWordsByCharCounts (dictCharCounts ["eat","all","tea"]))
["eat","tea"]

```

Figure 7: Part 3-5 Result

This function takes word and charater count to word dictionary, then returns anagrams of that word in dictionary. This function first finds characterCounts of given word then look dictionary

to find words that have same character counts. And it returns words that have same character counts

6. 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’.

Listing 32: Haskell Code for Finding Subsets of Given Character Counts

```
1 charCountsSubsets :: [CharacterCount] -> [[CharacterCount]]
2 charCountsSubsets cc =
3     --Convert given characterCount to a word.
4     let word = convertToWorld ([],cc)
5         -- Find all subsets of that list of chars. And remove duplicates ←
6         -- from list.
7         all_subsets = removeDuplicates (subsets word)
8         -- Convert all subsets on the list to characterCount list.
9         all_subsets_list = map (\x -> convertToCharacterCountList x []) ←
10         all_subsets
11     in all_subsets_list
12 where
13     convertToWorld :: DictionaryWordCharacterCount -> [Char]
14     convertToWorld (current_word,ch_list)
15         | length ch_list == 0 = current_word -- Final Iteration
16         | otherwise =
17             let element = head ch_list
18                 ch = fst element
19                 numberOfRepeat = snd element
20                 -- Duplicate given char
21                 str = createString (ch:[]) numberOfRepeat
22                 -- Add new string to list and call functions itself.
23                 in convertToWorld ((current_word ++ str),(tail ch_list))
24     -- Duplicate given char(converted string) to multiple char(←
25     -- according to n).
26     -- if string "a" and n = 4, the function returns aaaa.
27     createString :: String -> Int -> String
28     createString string n = concat $ replicate n string
29     -- Find subsets of list
30     subsets :: [a] -> [[a]]
31     subsets [] = [[]]
32     subsets (x:xs) = [zs | ys <- subsets xs, zs <- [ys, (x:ys)]]
33     -- Remove duplicats of the list.
34     removeDuplicates :: (Ord a, Eq a) => [a] -> [a]
35     removeDuplicates xs = remove $ quicksort xs
36     where
```

```

34         remove [] = []
35         remove [x] = [x]
36         remove (x1:x2:xs)
37             | x1 == x2 = remove (x1:xs)
38             | otherwise = x1 : remove (x2:xs)
39 -- Convert given string to characterCount list
40 convertToCharacterCountList :: [Char] -> [CharacterCount] -> [↵
    CharacterCount]
41 convertToCharacterCountList word character_count_list
42     | length word == 0 = character_count_list -- Final Iteration
43 -- Add convert first character of word to characterlist and it↵
    to list.
44     | otherwise = convertToCharacterCountList (tail word) (↵
        convertToCharacterCountList' character_count_list (head ↵
        word))
45 -- Add given char to characterCount to characterCount list.
46 convertToCharacterCountList' :: [CharacterCount] -> Char -> [↵
    CharacterCount]
47 convertToCharacterCountList' character_count_list ch =
48     insertCharacterCountList (head (wordCharCounts (ch:[]))) ↵
        character_count_list

```

```

[*Main> charCountsSubsets (wordCharCounts "all")
[[[]],[('a',1)],[('a',1),('l',1)],[('a',1),('l',2)],[('l',1)],[('l',2)]]

```

Figure 8: Part 3-6 Result

This function creates subset according to given charactercount list. Firstly, this function converts charactercountlist to a set of chars using 'convertToWord' and 'createString' helper functions. Then the function creates all set of words using 'subsets' helper function. These set have some same elements because of design but with 'removeDuplicates' helper function we are deleting duplicate elements. Then all subsets are being converted to characterCount lists using 'convertToCharacterCountList' and 'convertToCharacterCountList' helper functions.

7. 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'.

Listing 33: Haskell Code for Subtracting two Character Counts

```

1 subtractCounts :: [CharacterCount]->[CharacterCount]->[CharacterCount]
2 subtractCounts mapping1 mapping2 = subtractCounts' mapping1 mapping2 []
3     where
4         subtractCounts' :: [CharacterCount]->[CharacterCount]->[↵
            CharacterCount]->[CharacterCount]

```

```

5      subtractCounts' mapping1 mapping2 resultmapping
6      | length mapping1 == 0 = resultmapping -- Final Iteration
7      -- If first mapping and second mapping have same character
8      | length (filter (\x -> (fst x) == (fst (head mapping1))) <-
        mapping2) == 1 =
9          let element = head mapping1
10         decrease = snd (head (filter (\x -> (fst x) == (fst (head mapping1))) mapping2))
11         new_list = filter (\x -> (snd x) /= 0) (resultmapping <-
        ++ [(fst element, (snd element) - decrease)])
12         in subtractCounts' (tail mapping1) mapping2 new_list
13     -- First one has but second one does not have.
14     | otherwise =
15         let element = head mapping1
16         new_list = resultmapping ++ [(fst element, snd element) <-
        ]
17     in subtractCounts' (tail mapping1) mapping2 new_list

```

```

[*Main> subtractCounts [('a',1),('e',3),('l',2)] [('a',1),('l',1)]
[('e',3),('l',1)]

```

Figure 9: Part 3-7 Result

This function takes 2 characterCounts (Second one set of first one) and subtracts second one from first one. 'subtractCounts' helper function iterates until length of first one is not 0. Finally it returns resultmapping. This function uses tail recursive approach. It iterates all characters of mapping1. If mapping2 also have this character, the function subtracts character counts of second one from first one. Then make calls to itself with reduced mapping1 and updated resultmapping. If mapping1 has character but mapping2 has not that character, the function adds directly character and character counts to resultmapping.

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.

Listing 34: Haskell Code for Finding Anagrams of Sentence

```

1 sentenceAnagrams :: [Char] -> [DictionaryCharacterCountWord] -> [[Char]]
2 sentenceAnagrams sentence dictionary = ["I", "couldnt", "do", "it"]
3     where
4         l = charCountsSubsets (wordCharCounts (parseSentence sentence []))
5         -- sentenceAnagrams' :: [Char] -> [Char] -> [[Char]] -> [[Char]]
6         -- sentenceAnagrams' remaining_characters new_sentence <-
        all_sentences =

```

```

7      --      | length remaining_characters == 0 =
8      --      |
9      parseSentence :: [Char] -> [Char] -> [Char]
10     parseSentence sentence new_string
11         | length sentence == 0 = new_string
12         | head sentence /= ' ' = parseSentence (tail sentence) (↔
            new_string ++ [head sentence])
13         | otherwise = parseSentence (tail sentence) (new_string)

```

Until the eighth question, all functions work as desired. My computer was freezing while creating dictionary for these words because the number of words in the given text file was too high. Despite my efforts to try and fix everything, I could not find a solution. And finally I had to give up. Functional programming has been great, but this question has aged me a lot.

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).

Listing 35: Haskell Code for Main Program

```

1  main = do
2      handle <- readFile "words.txt"
3      let allWords = lines handle
4      args <- getArgs
5      print(args)
6      let dictionary = dictWordsByCharCounts (dictCharCounts allWords)
7      let allAnagrams = sentenceAnagrams (head args) dictionary
8      print(allAnagrams)

```

Until the eighth question, all functions work as desired. My computer was freezing while creating dictionary for these words because the number of words in the given text file was too high. Despite my efforts to try and fix everything, I could not find a solution. And finally I had to give up. Functional programming has been great, but this question has aged me a lot.

10. Name your source file part3.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 part3.hs -o part3
```

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

```
calico part3.yaml
```

```
(Cihats-MacBook-Air:FUNCTIONAL-FINAL varyok$ calico part3.yaml
compile ..... FAILED
Grade: 0 / 0
(Cihats-MacBook-Air:FUNCTIONAL-FINAL varyok$ ghc part3.hs -o part3
(Cihats-MacBook-Air:FUNCTIONAL-FINAL varyok$ calico part3.yaml
compile ..... PASSED
case_1 ..... FAILED
case_2 ..... FAILED
case_3 ..... FAILED
case_4 ..... FAILED
case_5 ..... FAILED
case_6 ..... FAILED
Grade: 0 / 0
```

Figure 10: Result of Calico