Functional Programming Assignment Report

Bohdan Tymofieienko B.S.
b.tymofieienko@student.fontys.nl
4132645

Mikolaj Hilgert M.H.
m.hilgert@student.fontys.nl
4158385

March 5, 2022

Contents

1 Caesar Week 3

This weeks assignment essentially worked to expand what was implemented in the first and second week. We were tasked to implement a sort of brute force way to decoding strings encoded with the Caesar cipher.

This brute force approach works on the concept that we pass through canaries and we try all the possible offset values and see in which of those the canaries (key words) show up. To implement this in Elm, recursion was used.

```
candidates: List String -> String -> List (Int, String)
candidates canaries text =
    (repeat 0 canaries text)
```

The repeat function is called which takes in the offset it should use, the wanted canaries and the encrypted text. This function then goes through each of the canaries we are searching for, and we check if with the given offset the decoded text (using a function made in last weeks assignment) contains the canary. If it does, then it is appended to a list. Then the next offset is inspected, until 25, after which the next canary is tested.

```
repeat: Int -> List String -> String -> List(Int,String)
repeat n canary text =
    case canary of
    (x :: xs) ->
        if n == 26 then
            repeat 0 xs text
        else
            if containsCanary x (decode n text) x then
                (n,(decode n text)) :: repeat (n+1) canary text
            else
                repeat (n+1) canary text
    [] ->
        []
```

Since we were not allowed to use the inbuilt String.Contains function, we had to implement our own implementation using case matching and recursion. This is the containsCanary function which takes in the canary and decoded text as well as an immutable copy of the canary and returns a Bool.

```
containsCanary: String -> String -> String -> Bool
containsCanary canary text reset =
    case (toList canary, toList text) of
    (x :: xs, y :: ys) ->
        if x == y then
            containsCanary (fromList xs) (fromList ys) reset
        else
            containsCanary reset (fromList ys) reset
    ( [], _ ) -> True
    ( _, [] ) -> False
```

This function works by recursively going through each character in the decoded string, if the consecutive characters of the input string match the consecutive characters in the canary, then the function will return True. For example, If a letter matches the first letter of the canary, then a recursive call is made with the rest of the canary and string. If not, then the function is called again using the rest of the decoded string, but instead of the rest of the canary being used, the entire canary is used.
This process is repeated until one of the following cases occurs:
- The canary list is empty, that means that the canary WAS present in the string. Thus, True is returned.
- If the decoded string runs out but the canary is not yet empty then that means that the canary is not present in the given string. Thus, the function returns False.

2 Credit card validation

This week the task was to implement the algorithm for credit card validation. In essence we followed the steps proposed in the document. Firstly we designed a function that converts string to list of digits. In this case we decided to use recursion with "case of" operator to split a string into list of characters and then convert them to integers one by one. As you can see "withDefault" function is necessary to assure that return type is list of integers and not "Maybe Int".

```
toDigits: String -> List Int
toDigits string =
    case (toList string) of
         [] -> []
         (x::xs) -> withDefault 0 (toInt (fromChar x)) :: toDigits (fromList xs)
```

We then simply reverse the list of integers with the use of "foldl" operator (Can be observed in the code). According to algorithm we have to double every second digit and take in consideration that if product exceeds ten it has to be decomposed into two digits. Again with the use of recursion we split the list into smaller pieces and apply certain logic. We decided to use a "trigger" value isSecond to indicate is the digit multiplied by two or not. Obviously after each operation we switch the trigger and achieve desired result.

```
doubleSecond: Bool ->List Int -> List Int
doubleSecond isSecond list =
    case list of
        [] -> []
        (x::xs) ->
            if isSecond == True then
                if 2*x >= 10 then
                  1::(remainderBy 10 (2*x))::(doubleSecond False xs)
                else 2*x::(doubleSecond False xs)
            else x::(doubleSecond True xs)
```

Moving further, we have to sum all the digits and decide if the result is dividable by 10 without a remainder. For that we apply "foldr" operation. Lastly to verify weather the card is valid or not we apply the chain of functions and check if the result is 0.

```
isValid: String -> Bool
isValid card =
    (remainderBy  10 (sumDigits (doubleSecond False (toDigitsRev card)))) == 0
```

In this particular case we were tasked to verify a set of credit card numbers. According to our result 94 cards are valid.