Applied Functional Programming Assignment Report

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

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

February 19, 2022

Contents

1 Caesar Week 2 1

2 Pythagoras Week 2 2

1 Caesar Week 2

This weeks assignment essentially worked to expand what was implemented in the first week. So rather than now encoding and decoding a character, we had to expand it to whole strings.

Before we could implement the string encoding function, we had to define the normalise function that would remove all non-alphabetical characters from the code. We first had to convert the input string into a list of characters such that we may use the Elm list-filter operation.

```
normalize: String -> String
normalize string =
    fromList (filter sanitize (toList string))
```

The sanitize function was also defined, and it returns a Boolean based on if the character is within the 'correct' bounds on the ASCII table. (Only returning true for the 'alphabetical' characters).

Having that, we now could actually implement the string encoding and decoding methods. To do this, we actually reused the character encoding/decoding functions from last week. We also did it recursively using case matching. **Note: The code for encode is shown below. But decode works on the same principle. It was not included for brevity.** The input is normalized.

```
encode: Int -> String -> String
encode offset string =
    case (toList (normalize string)) of
        [] ->
            ""

        x :: xs ->
            (fromChar (encodeChar offset x)) ++ (encode offset (fromList xs))
```

2 Pythagoras Week 2

This week we were tasked to extend our Pythagoras solution. Particularly having a possibility to pass a list to functions *pythTriple* and *isPythTriple*. To do so we use two approaches, implementation with built-in functionality (map, filter) and our custom recursive implementations.

We decided to reuse our solutions from week 1 to reuse the code for basic methods. However, in order to keep *main* file clean we decided to decompose our previous solutions to modules "Caesar" and "Pythagoras" respectively. Since using the map and filter functions is fairly simple and outcome is quite obvious it is worth focusing on our custom implementation more.

```
pythTriplesRec list =
    case list of
        [] ->
            []
        x :: xs ->
            pythTriple x :: pythTriplesRec xs
```

In the code above you may notice that we use "case list of" statement to break list into smaller pieces and apply certain operations to it. In a case list is empty, empty list is returned. For all the other cases list is split into parts **x** (single element, tuple) and **xs** (the rest of the list). As can be seen, result of the operation *PythTriple (see Week 1)* is concatenated with the rest of the new list with the use of "::" operator. The result of the function is identical to the result of "map" function if applying *PythTriple.*

```
x :: xs ->
        if isTripleTuple x then
            x :: (arePythTriplesRec xs)
        else (arePythTriplesRec xs)
```

In general, the same logic was applied for "arePythTriplesRec". Similarly to the case above we applied an "case list of" statement for the list of tuples. Again, function *isTripleTuple* was reused. Important detail is "if .. else ..." statement. It is used to either concatenate the resulting list with input value or not depending on the outcome of *IsTripleTuple* function. This is equivalent to "filter" function in elm.