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 12, 2022

Contents

1 Caesar Week 1

To encode a symbol with Caesar cipher we decided to use modulo operation.

Let X be the symbol's code and Y be the encrypted symbol's code and m be the offset.
Then,

$$Y_i = (X_i + m) \ \% \ 26 \tag{1}$$

However, since modern computers are using ASCII table to encode characters, we therefore must make sure that when we apply the offset, the character falls within the English alphabet. There is also a difference in codes between lower and upper case characters. Thus, our formula needs to be slightly modified. We first shift first letter of the alphabet to 0 and then bring it back to the original code after the operation.

Let β be the code of the first letter of English alphabet *('A' or 'a' depending on the case.)*
Then,

$$Y_i = ((X_i - \beta + m) \ \% \ 26) + \beta \tag{2}$$

In code we of course do have an *if then else* statement to distinguish between lower and upper case.

Function $decode(x_i)$ refers to $encode(x_i)$ function passing $0 - \beta$ *(negative offset)* as a parameter. By this we reuse the function $encode(x_i)$ which is obviously beneficial.

2 Hypotenuse Week 1

To solve the week 1 part of the Hypotenuse assignment. We set out a function for everything as instructed.

Note: The sqr, leg1, leg2 and hypotenuse functions are simple, as they only compute a given formula and therefore wont be explained. All their inputs (x , y) are verified using the same validXY function, that checks whether parameters are than 0, and x > y. As per the formula.

The implementation of the isTriple pythagorean test function works in the following way: In the reader we are given conditions that the parameters must be meet so that the function returns a bool value depending on if the given inputs are a triple. This means we need not use if else statements. Rather we can use the logical and operator to evaluate the result.

```
isTriple: Int -> Int -> Int -> Bool
isTriple adj op hyp =
    adj > 0 && op > 0 && hyp > 0 && ((sqr adj + sqr op) == sqr hyp)
```

.

The next notable function is the pythTriple function. Who's input is a tuple (x, y), and returns a tuple (a,b,c) based on given formula. The first parameter as the name suggests, takes the value of the first element in the input tuple.

```
pythTriple: (Int, Int) -> (Int, Int, Int)
pythTriple (first, second) = (leg1 first second,
                              leg2 first second,
                              hypotenuse first second)
```

.

The final function is similar to the previous one, but now instead of a tuple with two elements, we input a 3 element tuple and then uses the isTriple function, to evaluate whether the elements make up a pythagorean triple.

```
isTripleTuple: (Int, Int, Int) -> Bool
isTripleTuple (adj,op,hyp) =
    isTriple adj op hyp
```

3 Template explanation

When first learning about Elm, we were recommended to complete a an Elm practice program. We found that it was really easy to use, and the biggest benefit is that visually we would see, if a function was implemented correctly, or not.

We decided to adapt the original example Found here. It works on the premise where there are defined 'checker' functions that are called depending on the argument count. These functions take in the to-be-tested function, the arguments and then the expected output.

The checking is done using a function. Where the first parameter is the name to be displayed, and the second a list of Exercise runners. Within which we define which function is to be tested, using what inputs. And what the expected output is to be

```
caesar1 : List ( String, List ExerciseRunner.Example )
caesar1 =
    [ ( "Caesar (part 1)"
      , [ ExerciseRunner.functionExample2 "encode"
          encode
          [ ( (5, 'x'), 'c' )
          , ( (7, 'T'), 'A' )
          ]
      ]
      ),
    ]
```

After this is done, then the excercise function is added to the main HTML function.

```
caesar1
|> List.map (\( title, x ) ->
ExerciseRunner.viewExampleSection title x) |> Html.div []
```

To potentially extend the input parameter count, you must add a new function within the *ExerciseRunner.elm* file.

This functionality is shown in the image below: