

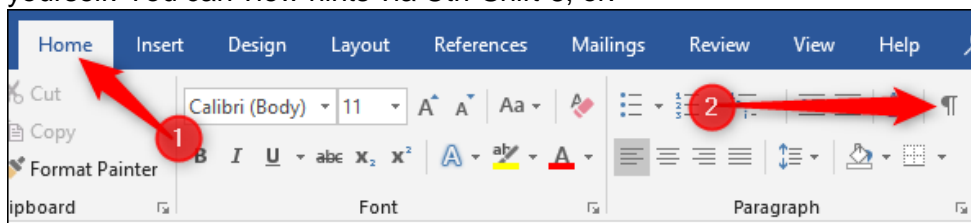
# Module D1. Functional Programming

## Summary

Functional Programming is a programming style where mathematical functions (that is, expressions with input parameters and an output result) are fundamental and running a program basically means evaluating a complex function. When compared to traditional imperative and OO programming styles, FP is much faster to develop and easier to test/debug, and is also very suitable for parallel applications. Lisp, Haskell, Scala, Elm are examples of purely functional programming languages very much used today.

This module aims at exposing you to the functional way of thinking and introducing FP concepts and techniques that have been proven useful in practice, like list comprehensions, generators, map, filter, reduce. We will work with Elm as default language, but will look at functional aspects of other languages as well.

Some exercises only have a brief description, such that you can design the sub functions yourself. You can view hints via Ctrl-Shift-8, or:



## Learning outcomes

### Learning outcome 1. Analysis

You are able to investigate, analyze and translate real world problems into abstract/mathematical models and solve them methodically and creatively.  
*You learn to use the functional programming style confidently, understand the pros and.*

### Learning outcome 2. Design

You are able to design abstract solutions (like algorithms and data structures) and able to reason about their validity/scope.  
*You design programs using functional programming paradigms (modular, reusable, without side-effects).*

### Learning outcome 3. Realization

You are able to translate abstract/mathematical solutions into concrete implementations and reason about the correctness of the implementation.  
*You program with basic functional concepts like filter, map, reduce, recursion.*

### Learning outcome 4. Testing

You are able to design and implement tests on the basis of abstract/mathematical specifications and reason about concepts like the coverage.  
*You reason about the correctness and time/space complexity of your programs, and measure their practical performance.*

### **Learning outcome 5. Research**

You are able to find and compare ideas and solutions from academic sources, such as research papers and are able to translate those abstract models/concepts into sustainable solutions.

*Throughout the course, you adopt a curious attitude and take charge of your learning process.*

### **Learning outcome 6. Interaction**

You are able to effectively interact and communicate with academically trained people on abstract problems and solutions.

*There will be enough opportunity for interaction with your teachers and colleagues.*

### **Learning outcome 7. Communication and Leadership**

You are able to communicate your ideas to other people on an academic level both in presentations and in writing.

*There will be enough opportunity to describe and present your work, for instance in the research assignment.*

## **Teaching and grading**

Weekly exercises + final exam.

Delivery of the home assignments:

- all assignments have to be delivered in Canvas, before the deadline
- to be delivered:
  - a. .elm file with proper test cases  
the filename contains your own name and the assignment's name
  - b. screen shot of the output (and/or video with commentary)
  - c. (small) doc showing your intelligence

# Assignments

## Week 1

### 1. Getting started (not to be submitted in Canvas)

- (30 min.) Read the explanations and follow the tutorial "Learning the basics" section at <https://elmbridge.github.io/curriculum/>. (i.e. the paragraphs "Introduction", "Intermission: Why Elm?", "Getting Started" and The Basic Building Blocks of Elm")
- (10 min.) Do "Lab 1. First Steps Introduction to Functional Programming edX" (available on SharePoint)
- (10 min.) Do the beginning of <https://elmprogramming.com/list.html> (until "Sorting a List")

### 2. Caesar (part 1)

When we talk about cryptography these days, we usually refer to the encryption of digital messages, but encryption actually predates the computer by quite a long period. One of the best examples of early cryptography is the Caesar cipher, named after Julius Caesar because he is believed to have used it, even if he didn't actually invent it. The idea is simple: take the message you want to encrypt and shift all letters by a certain amount between 0 and 26 (called the offset). For example: encrypting the sentence "THIS IS A BIG SECRET" with shifts of 5, would result in "YMNX NX F GNL XJHWJY". See also

<https://cryptii.com/pipes/caesar-cipher>

In this exercise you will be implementing a variant of the Caesar cipher. You can use the list of library functions linked from the course webpage, as well as those in the Appendix of this tutorial sheet.

Write a function

```
encode: Int -> Char -> Char
```

that encrypts the given single character using the given key (i.e. the offset). Note that lower case letters remain lower case, and upper case letters remain upper case. For example:

```
> encode 5 'x'  
'c'  
> encode 7 'T'  
'A'
```

Write a function

```
decode: Int -> Char -> Char
```

that decrypts the given single character using the given offset. For example:

```
> decode 5 'c'  
'x'  
> decode 7 'A'  
'T'
```

Tip:

- use `modBy`, `Char.toCode` and `Char.fromCode` (<https://package.elm-lang.org/packages/elm/core/latest/Char>)
- a look at the ASCII table (<https://ascii-code.net>) might be convenient

- when implementing `decode`: use `encode`
- when you see similar code lines appearing: make new functions

### 3. Pythagoras (part 1)

A Pythagorean triple is a set of three integers (a, b, c) which satisfy the equation  $a^2 + b^2 = c^2$ . For example, (3, 4, 5) is a Pythagorean triple, since  $3^2 + 4^2 = 9 + 16 = 25 = 5^2$ . See also [https://en.wikipedia.org/wiki/Pythagorean\\_triple](https://en.wikipedia.org/wiki/Pythagorean_triple)

Write a function

```
sqr: Int -> Int
```

which calculates a square, and write a function

```
isTriple: Int -> Int -> Int -> Bool
```

which tests for Pythagorean triples. Take care of negative or zero length. Make sure that it returns `True` for numbers that satisfy the equation (such as 3, 4 and 5) and `False` for numbers that don't (such as 3, 4 and 6).

Next we'll create some triples automatically. One formula for finding Pythagorean triples is as follows:  $(x^2 - y^2, 2yx, x^2 + y^2)$  is a Pythagorean triple for all positive integers x and y with  $x > y$ .

Write functions `leg1`, `leg2` and `hyp` that generate the components of Pythagorean triples using the above formulas.

```
leg1 : Int -> Int -> Int
leg2 : Int -> Int -> Int
hyp  : Int -> Int -> Int
```

Write function `pythTriple` with input tuple (x, y), generates a tuple (a, b, c) (according the formula above):

```
pythTriple : (Int, Int) -> (Int, Int, Int)
```

Examples:

```
> leg1 5 4
9
> leg2 5 4
40
> hyp 5 4
41
> isTriple 9 40 41
True
> pythTriple (5,4)
(9,40,41)
```

Write a function

```
isTripleTuple: (Int, Int, Int) -> Bool
```

which has a tuple as an input. Tip: use `isTriple` for its implementation. Examples:

```
> isTripleTuple (9,40,41)
True
> isTripleTuple (pythTriple (5,4))
True
```

## Week 2

### 1. Caesar (part 2)

Write a function

```
normalize: String -> String
```

which removes spaces and all non-letters (like #, =, !, @). Example:

```
> normalize "Hello, Fontys!"
"HelloFontys"
```

Hints
-------

Write the functions

```
encrypt: Int -> String -> String
decrypt: Int -> String -> String
```

to encrypt and decrypt a normalized string. Tip: same as `normalize`. Do not use `String.map`. Example:

```
> encrypt 7 (normalize "Hello, Fontys!")
"OlssvMvuafz"
> decrypt 7 "OlssvMvuafz"
"HelloFontys"
```

### 2. Pythagoras (part 2)

Write functions

```
pythTriplesMap: List (Int, Int) -> List (Int, Int, Int)
pythTriplesRec: List (Int, Int) -> List (Int, Int, Int)
```

which calculates the a-b-c triples for the given x-y values. Make 2 implementations: with `List.map` and recursively. Example:

```
> pythTriplesMap [(5,4), (2,1), (35,7)]
[(9,40,41), (3,4,5), (1176,490,1274)]
> pythTriplesRec [(5,4), (2,1), (35,7)]
[(9,40,41), (3,4,5), (1176,490,1274)]
```

Write functions

```
arePythTriplesFilter: List (Int, Int, Int) -> List (Int, Int, Int)
arePythTriplesRec: List (Int, Int, Int) -> List (Int, Int, Int)
```

which removes all non-Pythagorean triples. Make 2 implementations: with `List.filter` and recursively. Example:

```
> arePythTriplesRec [(1,2,3), (9,40,41), (3,4,5), (100,2,500)]
[(9,40,41), (3,4,5)]
```

## Week 3

### 1. Caesar (part 3)

One kind of brute-force attack on an encrypted string is to decrypt it using each possible key and then search for common English letter sequences in the resulting text. If such sequences are discovered then the key is a candidate for the actual key used to encrypt the plaintext. For example, the words "the" and "and" occur very frequently in English text: in the Adventures of Sherlock Holmes, "the" and "and" account for about one in every 12 words, and there is no sequence of more than 150 words without either "the" or "and".

The conclusion to draw is that if we try a key on a sufficiently long sequence of text and the result does not contain any occurrences of "the" or "and" then the key can be discarded as a candidate.

Of course do not use `String.contains`, `String.startsWith`, `String.endsWith`, `String.indexes`, `String.indices`, `List.any` for any of the functions described below.

Hints

Write a function

```
candidates: List String -> String -> List (Int, String)
-- semantics:
-- candidates canaries decodedText
```

that decrypts the input string with each of the 25 possible keys and, when the decrypted text contains one of the search strings (`canaries`), then include the decryption key and the decrypted text in the output list. Example:

```
> candidates ["THE", "AND"] "DGGADBCOOCZYMJHZYVMTOJOCZHVS"
[(5, "YBBVYWXJJXUTHECUTQHOJEJXUCQN"),
 (14, "PSSMPNOAAOLKYVTLKHYFAVAOLTHE"),
 (21, "ILLFIGHTTHEDROMEDARYTOTHEMAX")]
```

### 2. Validating Credit Card Numbers

Use the template `CreditCard_template.elm`. Make sure that *at least once* you have applied each of the following features:

- function recursion
- list recursion
- lambda expression (aka. anonymous function)
- `List.map`
- `List.filter`
- `List.foldr` or `List.foldl`

and do not use the trivial functions like:

- `List.reverse`
- `List.sum`

Have you ever wondered how websites validate your credit card number when you shop online? They don't check a massive database of numbers, and they don't use magic. In fact, most credit providers rely on a checksum formula for distinguishing valid numbers from random collection of digits (or typing mistakes).

In this exercise, you will implement a **validation algorithm for credit cards**. The algorithm follows these steps:

- Double the value of every second digit beginning with the rightmost.
- Add the digits of the doubled values and the undoubled digits from the original number.
- Calculate the modulus of the sum divided by 10.

If the result equals 0, then the number is valid. See also:

<https://www.creditcardvalidator.org/articles/luhn-algorithm>. Here is an example of the results of each step on the number "401288888881881".

- In order to start with the rightmost digit, we produce a reversed list of digits. Then, we double every second digit.

*Result: [1,16,8,2,8,16,8,16,8,16,2,2,0,8].*

- We sum all of the digits of the resulting list above. Note that we must again split the elements of the list into their digits (e.g. 16 becomes [1, 6]).

*Result: 90.*

- Finally, we calculate the modulus of 90 over 10.

*Result: 0.*

Since the final value is 0, we know that the above number is a valid credit card number. If we make a mistake in typing the credit card number and instead provide 401288888881891, then the result of the last step is 2, proving that the number is invalid.

Hints
-------

Write the function

```
numValid: List String -> Bool
```

that tells how many valid credit card numbers there are in a list.

Calculate and show how many valid credit card numbers are in the given list `creditcards`.

## Week 4

### 1. Sorting

Implement the merge sort (<https://www.geeksforgeeks.org/merge-sort/>) in a proper and efficient FP way; in this case: by *only* using list recursion.

Hints
-------

Write the function

```
msort: List comparable -> List comparable
```

Function `msort` sorts a list via the mergesort algorithm.

### 2. Modelling Math functions (part 1)

Given the type Function, write function

```
print: Function -> String
eval: Float -> Function -> Float
graph: Function -> Int -> Int -> Int -> Int -> String
```

Function `print` gives a human readable infix representation of the Function.

Function `eval` calculates the function values of the given input values.

Function `graph` makes an graph of the function, within the ranges `xmin-xmax` and `ymin-ymax`. You may choose another return type if you prefer. Tip: it's easier to rotate 90 degrees (so the y-axis is from left to right, and the x-axis is from top to bottom). And perhaps you need to enlarge the variable `page_width` of the template.

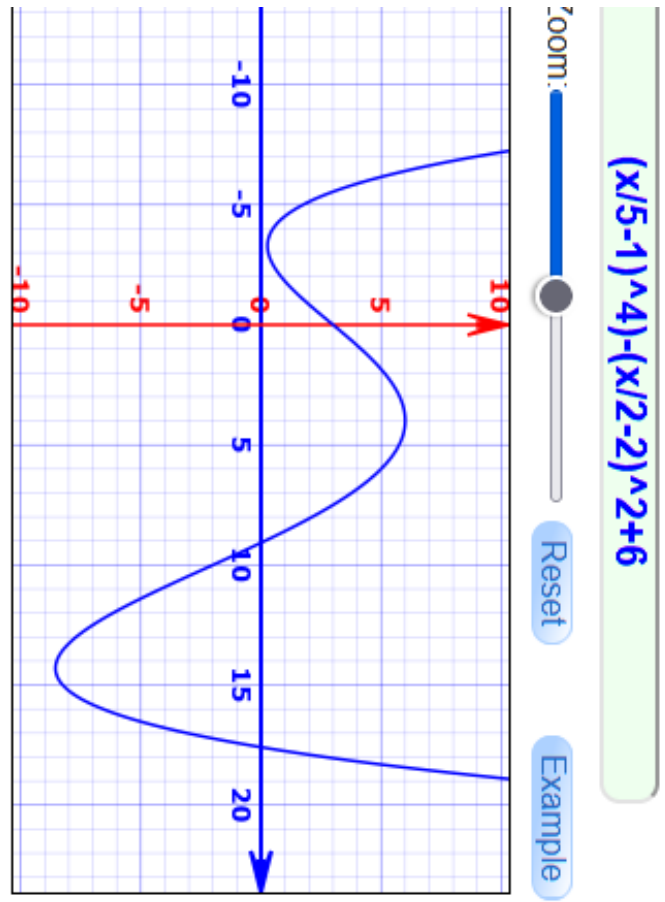
You can use <https://www.mathsisfun.com/calculus/derivative-plotter.html> to create your own functions and to check your results.

Examples:

```
> f = Plus (Mult (Plus (Const 3) X) (Minus X (Poly X 5))) (Const 2)
> print f
"((3 + x) * (x - (x ^ 5))) + 2"

> eval 2 f
-148
```



[illegible]

```
above100: Int -> Bool
above100 x =
```

```
x > 100
```

```
double: Int -> Int
double x =
  x * 2
```

```
repeatUntil above100 double 7
```

In this case, `repeatUntil` is repeatedly applying function `double` on initial value 7 (and its result etc.) until the value is greater than 100 (i.e. the predicate function `above100` holds).

```
above100 7 is False, so apply: double(7) = 14
double(14) = 28
double(28) = 56
double(56) = 112, and now it stops because above100 112 is True
```

So: `repeatUntil p f init` is continuously calculating `f(f...f(f(f(init))))` until predicate `p` holds.

- Check that

```
repeatUntil above100 ((+) 1) 42
```

indeed returns 101.

- Change `above100` into a new function such that it can stop above any given value.
- Calculate  $^3\log 100$  (for natural numbers only). Spoiler: answer=5 because  $3^5=243$  (and  $3^4 = 81$ ).

Make sure that the calculations can be done for other base numbers and target numbers as well.

- Examine if the Collatz sequence of a given number indeed ends at 1 (see [https://en.wikipedia.org/wiki/Collatz\\_conjecture](https://en.wikipedia.org/wiki/Collatz_conjecture)).

Now we don't choose the application function `f` being `Int -> Int` (because the final value is 'always' 1, too boring):

```
repeatUntil ((==)1) (\x->if even x then x//2 else 3x+1) 19
```

but instead: `List Int -> List Int`, collecting the whole sequence. Example:

```
> repeatUntil myPredicate myCollatz [19]
[1,2,4,8,16,5,10,20,40,13,26,52,17,34,11,22,44,88,29,58,19]
```

## 2. Modelling Math functions (part 2)

Given the type Function, write functions

```
derivative: Function -> Function
simplify: Function -> Function
```

Function `derivative` gives the derivative of a function (following all rules like chain-rule, etc.)

Function `simplify` prunes unnecessary branches like `+0`, `*0`, `*1`, `^1`, `^0`, `/1`. It also simplifies the multiplication, addition and subtraction of constants, like `3+6`, `4^5`, `9*5`, `-2--2`.

Examples:

```
> print f
"(((3+x)*(x-(x^5)))+2)"
> print <| derivative f
"((((0+1)*(x-(x^5)))+(3+x)*(1-((5*(x^4))*1))))+0)"
> print <| simplify <| derivative f
"((x-(x^5))+((3+x)*(1-(5*(x^4)))))"
```

Make your own (nasty) test cases to check that it also works flawlessly for all kind of combinations of nested simplifications etc.

Also here: use <https://www.mathsisfun.com/calculus/derivative-plotter.html> to validate your answer: if you have calculated a (simplified) derivative, you can print it in the lower figure (and this screen shot shows that my calculation is not yet correct...)

