# Elixir Programming

Lucas Larsson

Spring Term 2022

## 1 Introduction

This document is written with the assumption that the reader knows the assignment reported here, and have basic understanding of the Elixir, programming language.

The solution for the assignment Enkla Funktioner is represented below with some code snippets and explanation of the functions used.

## 2 Solution

Not all suggested exercises are solved here since the purpose of the lab not to solve all of them rather just get started with Elixir and LaTeX.

### 2.1 A first program

This is the first function written just to check that the environment is up and running.

```
   # Compute the double of a number
 @spec double(number) :: number
 def double(x) do
   x * 2
 end
```

No need to explain how the function works it just `x * 2` , One thing to observe though is the Elixir syntax. A function is defined only by the key word `"def"` and the code inside the `do .. end` clauses is executed.

### 2.2 Multiplication

Recursion is known as a very important method to do operations that require repetition, which is basically all operations, but it is even more important in Elixir since there is no `While` or `For` loops compared to other programming languages.

the following function shows how multiplication operation is implemented using only addition operation and recursion.

```
# Compute the product of x and y using recursion
@spec prod(number, any) :: number
 def prod(0, _ ) do 0 end
 def prod(1 , x ) do x end

 def prod( x , y ) do
 # print to CLI for debugging
     # IO.puts(x)
     # IO.puts(y)
   x + prod( y - 1 , x)
end
```

The above shown function shows how the product of two numbers `x, y` is computed using the addition operation and a recursive call to the same method. The Most important rule to take into consideration when writing a recursive function is the base case, otherwise it will run forever or until the computer crashes.

The method to compute the product is writing as three clauses for easy readability and since it was this how the author wrote the function, first base case is when one of the arguments is a 0 then the product is also 0, as shown above. Second base case is when one of the arguments is `1` then the method returns the number multiplied by `1`.

The third clause is the clause that is going to get most the calls, and it's the most complicated relative to the other two clauses.

## 2.3 Power Operation

After creating the `"prod"` function i thought of how i could use it even further more to compute the `"exp"` function, i.e x to the power of y.

Using the same method of implementing the base case first where x to the power of 0 is 1, and x to the power of 1 is x, and then x to the power of y is prod(x,x) y times.

```
# Compute m to the power of n
@spec exp(number, non_neg_integer) :: number
def exp(_ , 0) do 1 end
def exp(m , 1) do m end
def exp(m , n) do prod( m ,exp(m, n - 1)) end
```

The code shown above shows the cases mentioned previously, a underscore means that input can by any number, no matter the number if the second argument is 0 then the function should result in 1.

Second clause returns the first argument if the second argument is 1.

The third clause compute the rest of the calculations it takes in two arguments first argument m multiplied by the exp function with the arguments m and n-1, this continues in a loop and for each iteration it get saved to the call stack and jump to the exp function until n = 1 then it results in a return of m and start to get back to the last saved address on the call stack and do the operation there, until it is done all the up to the first call of the function.

## 2.4 C to F and vice versa

Last two functions are inspired by the assignment paper.

The math functions to convert from C to F and vice versa are as follows.

$$C = \frac{5 * (F - 32)}{9}$$

$$F = \frac{C * 9}{5} + 32$$

After considering the mathematical formals above two functions are written as shown below

```
# Compute from F to C
@spec fer_c(number) :: float
def fer_c (x) do
  (x - 32) / 1.8
end

# Compute from C to F
@spec c_fer(number) :: float
def c_fer (x) do
  x  * 1.8 + 32
end
```