Lab 1

Introduction to Functional Programming with Elm

Goals

In this lab you will learn to:

1. Use the Elm REPL (**R**ead-**E**valuate-**P**rint-**L**oop)
2. Define and call basic functions in Elm
3. Understand function signatures
4. Create basic logic using `if` expressions
5. Write recursive and tail recursive functions

Resources

Table 1.1: Lab Resources

Resource	Link
Elm website	`https://elm-lang.org/`
Elm core language overview (up to if expressions)	`https://guide.elm-lang.org/core_language.html`

1.1 Meet Elm

Elm is a Functional Programming language for building *reliable* web applications. It takes an *opinionated* approach to many aspects of programming, which means that is mainly only one *easy way* to achieve things.

A couple of things to note before we get into the details:

1. Elm (as most modern functional languages) is *expression oriented*. This means that we don't have *statements* and every function *must return a value*.
2. Elm (as most modern functional languages) has *type inference*, which means that we can just write a function without type annotations, like in dynamically typed languages and the compiler will *infer* the types for us, which means that programs will be shorter, but will benefit from *type checking* which prevents errors like passing a string to a function which expects a number as an argument.
3. Elm has no (syntax for declaring or throwing/raising) exceptions. This largely means that every error has to be handled in order for the code to compile.
4. Just like Python, Elm (as well as Haskell) also uses indentation to determine where a scope begins and ends.
5. For certain errors, the REPL and compiler will direct you to read one of the files at this link: `https://github.com/elm/compiler/tree/master/hints`. It is worth reading them to learn about the reasoning for some of more opinionated aspects of Elm's design.

1.2 Installation

Follow the instructions for Elm in chapter A (found in `Appendix-A.pdf`).

1.3 The REPL

Concept 1.3.1: REPL

The **R**ead **E**valuate **P**rint **L**oop was first introduced by LISP.

Question 1.3.1 *

Can you name a language which has a REPL?

To start the Elm REPL type the command below:

Shell session

```
> elm repl
---- Elm 0.19.1 ---------------------------------------------------------------
Say :help for help and :exit to exit! More at <https://elm-lang.org/0.19.1/repl>
-------------------------------------------------------------------------------
```

1.4 Simple expressions

You can evaluate simple expressions as you would in any other language:

```
Elm REPL

> 1 + 1
2 : number
> "Hello" ++ " World"
"Hello World" : String
> True && False
False : Bool
```

Notice, that the besides the value response, the REPL also displays its *type*. These types are deduced by the REPL using *type inference*.

Let's see what happens when we try to use `+` on a `String` and a `number`:

```
Elm REPL

> 1 + "Hello"
-- TYPE MISMATCH ------------------------------------------------------- REPL

I cannot do addition with String values like this one:

3|   1 + "Hello"
         ^^^^^^^
The (+) operator only works with Int and Float values.

Hint: Switch to the (++) operator to append strings!
```

Elm will try to provide helpful error message, saying that we can't add a `number` and a `String` using the `+` operator.

We can indeed confirm this by checking the *signature* of the `(+)` operator using:

```
Elm REPL

> (+)
<function> : number -> number -> number
```

Note 1.4.1

To check the signature of a function we can simply type its name, but to get the signature of an *infix operator* (`+`, `++`, `\`, etc.) we need to put it in parentheses.

1.5 Defining functions

Let's try to define some functions:

```
Elm REPL

> increment n = n + 1
<function> : number -> number
> increment 1
2 : number
```

As you can see Elm can also deduce the signature of the functions we declare, by using *type inference*. By using the `+` operator and 1 in `increment`, Elm deduced that the type of `n` must also be `number`.

We can also define *constants*, which can also be seen as functions, which take no parameters and always return the same value:

```
Elm REPL
> favoriteNumber = 42
42 : number
> greeting = "Hello"
"Hello" : String
```

Note 1.5.1

In Elm variable shadowing **is an error**!
Try running the following sequence in the REPL to see the error:

```
Elm REPL
> n = 10
10 : number
> double n = n * 2
```

The simplest solution to fix this error is to type `:reset` in the REPL, which will clear all current bindings.

Let's try some more examples:

```
Elm REPL
doubleString s = s ++ ", " ++ s
<function> : String -> String
> rightTriangle a b c = a*a + b*b == c*c
<function> : number -> number -> number -> Bool
```

1.6 Calling functions

To call a function, just write its name and each of the arguments separated by spaces.

```
Elm REPL
> doubleString "Hello"
"Hello, Hello" : String
> rightTriangle 3 4 5
True : Bool
> rightTriangle 2 2 3
False : Bool
```

1.7 Function signatures

The syntax for a function signature is:

$$functionName : TypeOfParam_1 \to \cdots \to TypeOfParam_n \to ReturnType$$

In the REPL, we can write the function signature, followed by its implementation. After you type the first line, the REPL will automatically print the `|` symbol and the name of the function (i.e. `rightTriangle`) for you.

```
                                                                    Elm REPL
> rightTriangle : Float -> Float -> Float -> Bool
| rightTriangle a b c = a*a + b*b == c*c
|
<function> : Int -> Int
```

Why should we bother to write the function signatures when Elm can infer them anyway? The main reason is to help Elm check that the implementation part of the function corresponds with its signature.

You'll learn more about why functions look like this later.

1.8 If expressions

To do something useful we also need `if` expressions:

```
                                                                    Elm REPL
> boolToAnswer b = if b then "Yes" else "No"
<function> : Bool -> String
```

So far, it looks similar to the `if` statements in other languages like C and Java, but there is one big Caveat: **the `else` branch is mandatory**!

Concept 1.8.1: Expressions and Statements

Expressions **return a value** while statements **have side effects**.

Let's see what happens if we try to skip it:

```
                                                                    Elm REPL
> nope n = if n == False then "Nope"
|
-- UNFINISHED IF ------------------------------------------------------- REPL

I was expecting to see an ‘else‘ branch after this:

3| nope n = if n == False then "Nope"
                                       ^

I know what to do when the condition is True, but what happens when it is False?
Add an else branch to handle that scenario!
```

We can also chain multiple if expressions:

```
Elm REPL
howBig n = if n < 10 then "Small" else if n < 100 then "Medium" else "Large"
<function> : number -> String
> howBig 3
"Small" : String
> howBig 11
"Medium" : String
> howBig 110
"Large" : String
```

You can also write the same function on multiple lines for better readability (the | symbols at the beginning of the lines are printed by the REPL):

```
Elm REPL
> howBig n =
|   if n < 10 then
|   "Small"
|   else if n < 100 then
|   "Medium"
|   else
|   "Large"
|
<function> : number -> String
```

1.9 Recursion

The second ingredient for writing interesting and useful functions is *recursion*. Since in Elm we have no looping mechanisms like `for` or `while`, the only way to iterate is with recursion.

```
Elm REPL
> fact n = if n == 0 then 1 else n * fact (n - 1)
<function> : number -> number
```

Exercise 1.9.1

Consider the expression `fact 5 - 1`. What result do you expect to get?

Note 1.9.1

To avoid unpleasant surprises, use parentheses whenever you're unsure about precedence.

```
Elm REPL
> pow n i = if i == 0 then 1 else n * pow n (i-1)
<function> : number -> number1 -> number
> pow 2 3
8 : number
> pow 3 2
9 : number
```

Tail recursion

As a short reminder, when a function is called, memory has to be allocated for the parameters and local variables of the call. Thus a recursive function can run out of *stack space* by calling itself too many times.

Consider the following, deliberately inefficient function:

```
                                                                    Elm REPL
> slowAdd a b = if b == 0 then a else 1 + slowAdd a (b-1)
<function> : number -> number1 -> number
```

It simply adds `a` and `b` together by successively adding `b` copies of 1 to `a`.

```
                                                                    Elm REPL
> slowAdd 1 100
101 : number
> slowAdd 100 1
101 : number
> slowAdd 1 1000
1001: number
> slowAdd 1000 1
1001 : number
> slowAdd 1 10000
RangeError: Maximum call stack size exceeded
> slowAdd 10000 1
10001 : number
> slowAdd 100000 1
100001 : number
```

As you can see, if we make `b` too large, we will overflow the stack, since `b` determines the number of recursive function calls.

Exercise 1.9.2 *

Find two values for `b`, b_1 and b_2 such that $b_2 = b_1 + 1$ and `slowAdd` overflows for b_2, but it doesn't for b_1.

Concept 1.9.1: Tail recursive functions

A function is tail recursive if it returns either something computed directly or something returned by its recursive call (the last thing it does is to call itself).

To make `slowAdd` tail recursive we can store the current result in `a`, *passing it on* at each function call and returning it when we reach the exit condition.

```
Elm REPL

> betterAdd a b = if b == 0 then a else betterAdd (a+1) (b-1)
<function> : number -> number1 -> number
> betterAdd 1 10000
10001 : number
> betterAdd 1 100000
100001 : number
> betterAdd 1 1000000
1000001 : number
```

As you can see, not we can call `betterAdd` with much larger values for `b`.

Concept 1.9.2: Tail call optimization

Tail call optimization is a *compiler optimization* which ensures that tail recursive functions occupy constant space on the stack, thus allowing a larger number of recursive function calls.
Due to the fact that the last action of the function is to call itself, we know that nothing that is before the recursive call can be used (e.g. like the `1` in `1 + slowAdd a (b-1)` in `slowAdd`) after the recursive call returns, so we can reuse the space (allocated on the stack) for the next recursive call.

As a second example, consider the `fact` function above. Currently it will build up a stack of partial results, each waiting to be evaluated when the function returns. We can rewrite `fact` in a similar way to `betterAdd` to use an additional *accumulator* parameter that will be used to store the partial results between the recursive calls:

```
Elm REPL

> factAcc n acc = if n == 0 then acc else factAcc (n-1) (acc * n)
<function> : number -> number1 -> number -> number
> factAcc 10 1
3628800 : number
> factAcc 5 1
120 : number
```

Note 1.9.2

Notice that each time we call `factAcc` we need to pass it 1 as a starting value for the accumulator. A useful pattern to avoid this is to define an auxiliary function which uses an accumulator, `factAccHelper` and wrap it in a function which passes it the initial parameter values, `factAcc`:

```
Elm REPL

> factAccHelper n acc = if n == 0 then acc else factAccHelper (n-1) (acc * n)
<function> : number -> number -> number
> factAcc n = factAccHelper n 1
<function> : number -> number
```

Finally consider the Fibonacci function, `fib`:

```
Elm REPL
> fib n = if (n == 0) || (n == 1) then 1 else (fib (n-1)) + (fib (n-2))
<function> : number1 -> number
> fib 20
10946 : number
> fib 30
1346269 : number
```

Exercise 1.9.3

Evaluate the following expressions in the REPL: `(fib 40)`, `(fib 45)` and `(fib 50)`. How long does each take?

It should be clear after completing the exercise above, that the `fib` function is very inefficient. We can rewrite it in manner similar to `factAcc`, by using 2 accumulators, `f1` and `f2`, adding `f1` and `f2` to obtain the next Fibonacci number and shifting them on each iteration:

```
Elm REPL
> fibTailHelper f1 f2 n = if n == 0 then f2 else fibTailHelper f2 (f1 + f2) (n - 1)
<function> : number -> number -> number1 -> number
> fibTail n = fibTailHelper 0 1 n
<function> : number -> number
```

Exercise 1.9.4

Evaluate the following expressions in the REPL: `(fibTail 50)`, `(fibTail 100)` and `(fibTail 1000)`. How long does each take?

1.10 Practice problems

Exercise 1.10.1 **

Implement in Elm the **Euclid-style** greatest common divisor function, `gcd a b`, in a tail recursive manner.
Be sure to test your implementation for all of these values! Did you manage to implement the function without getting any compile errors?

```
Elm REPL
> gcd 60 12
12 : Int
> gcd 70 12
2 : Int
> gcd 70 25
5 : Int
> gcd 70 50
10 : Int
```

Exercise 1.10.2 **

Write an Elm function called `ack`, which computes the values of the Ackermann function

(denoted by A), defined as follows:

$$A(n,m) = \begin{cases} m+1 & \text{if n} = 0 \\ A(n-1,1) & \text{if m} = 0 \\ A(n-1,A(n,m-1)) & \text{otherwise} \end{cases}$$

Elm REPL

```
> ack 1 1
3 : number
> ack 2 3
9 : number
> ack 3 3
61 : number
> ack 4 1
RangeError: Maximum call stack size exceeded
> ack 3 10
8189 : number
```

Exercise 1.10.3 **

Write an Elm function called `sudan`, which computes the values of the function of Gabriel Sudan (denoted by S), defined as follows:

$$S(n,x,y) = \begin{cases} x+y & \text{if n} = 0 \\ x & \text{if n} > 0 \text{ and y=0} \\ S(n-1,S(n,x,y-1),y+S(n,x,y-1)) & \text{otherwise} \end{cases}$$

Elm REPL

```
> sudan 1 1 1
3 : number
> sudan 1 2 1
5 : number
> sudan 1 2 2
12 : number
> sudan 2 1 1
8 : number
> sudan 2 2 2
15569256417 : number
```