Lab 4

Elm - Modules, Higher order functions and list processing

Goals

In this lab you will learn to:

1. Expose (export and import) only certain functions and types from modules
2. Use Debug.todo to temporarily make your code compile
3. Define and use lambda expressions
4. Work with higher order functions
5. Use curried functions
6. Update all elements in a list using the `map` function
7. Return elements that match a predicate from a list using the `filter` function
8. Process all elements in a list using the `fold` function
9. Work with `String`s

Resources

Table 4.1: Lab Resources

Resource	Link
Elm core language overview	`https://guide.elm-lang.org/core_language.html`
Elm core library	`https://package.elm-lang.org/packages/elm/core/1.0.5/`
Lists module from the core library	`https://package.elm-lang.org/packages/elm/core/1.0.5/List`

4.1 Modules, visibility and qualified imports

In lab 2 we learned how to create and load modules in the REPL, but the `exposing (..)` part was not detailed.

4.1.1 Controlling exported items

To control what functions and types (further referred to as *symbols*) are exported from a module, we write the names of the exported symbols in the list after the `exposing` keyword.

For example let's create `Date` module, which defines the types `Month` and `Date` and the functions `createDate`, `daysInMonth`, `monthToInt` and `compareMonth`.

To export the `daysIntMonth` function, we just add its name to the `exposing` list.

Listing 4.1.1: Exposing a function — Elm code

```elm
module Date exposing (daysInMonth)
```

To export the `Month` type and *all its variants*, we write the name of type (`Month`) and add `(..)` after it:

Listing 4.1.2: Exposing a type and all of its variants — Elm code

```elm
module Date exposing (daysInMonth, Month(..))
```

We can also export only a type name, without its variants, which will prevent the respective type from being instantiated. To do this for the `Date` type we add only the type name to the list:

Listing 4.1.3: Exposing only a type — Elm code

```elm
module Date exposing (daysInMonth, Month(..), Date)
```

Question 4.1.1 *

How can we prevent a type from being instantiated in uncontrolled ways in Java?

4.1.2 Open imports

Until now whenever we imported a module we always used the `exposing (..)` after it. We can actually omit this part and there will be no errors. If we try to use the function `daysInMonth` like we did until now, we get an error:

Elm REPL

```
> import Date
> daysInMonth
-- NAMING ERROR ---------------------------------------------------------- REPL

I cannot find a `daysInMonth` variable:

4|   daysInMonth
     ^^^^^^^^^^^
These names seem close though:

    Date.daysInMonth
    Basics.not
    asin
    isInfinite

Hint: Read <https://elm-lang.org/0.19.1/imports> to see how `import`
declarations work in Elm.
```

This error already gives us the solution to the problem: we need to refer to the exported symbols by their *fully qualified name*.

Elm REPL

```
> Date.daysInMonth
<function> : Date.Month -> Int
```

4.1.3 Controlling imported items

To import a symbol in the *global scope* (i.e. so that it can be used everywhere), the syntax is the same as for exports:

Listing 4.1.4: Importing a function — Elm code

```elm
import Date exposing (daysInMonth)
```

Listing 4.1.5: Importing a type and all of its variants — Elm code

```elm
import Date exposing (daysInMonth, Month(..))
```

Listing 4.1.6: Importing only a type — Elm code

```elm
import Date exposing (daysInMonth, Month(..), Date)
```

4.1.4 Qualified imports

You can also *rename* imported modules, using the `as` keyword:

Elm REPL

```
> import Date as D
> D.daysInMonth
<function> : D.Month -> Int
```

We can also mix open imports with qualified imports:

```
Elm REPL
> import Date as D exposing (Month(..))
> D.daysInMonth Jan
31 : Int
```

Note 4.1.1

Qualified imports are the preferred importing method in Elm. They avoid name conflicts and allow you to easily determine which module exports a given function or type.

4.2 Making your code compile with `Debug.todo`

Sometimes when you want to write a complex function which uses multiple helper functions, you might want to test your implementation for some of the functions, but you can't load a module in the REPL if it contains any errors (undefined functions or syntax errors).

To indicate to the Elm compiler that you want to implement a function, but won't use it right now you can use the `Debug.todo` function which when applied to a `String`, can replace any function or argument.

Note 4.2.1

If the `Debug.todo` function gets evaluated at runtime, the program will crash.

4.3 Higher order functions

Since in Functional Programming languages functions are first class citizens, we can easily pass functions around. The simplest example is function that applies a function to one argument and then applies it again on the result:

```
Elm REPL
> applyTwice f x = f (f x)
<function> : (a -> b) -> a -> b
```

Notice the parentheses around the first `a -> b`, which denote that it is a function.

4.3.1 Partial application and Currying

Functions can also return functions, but there is a crucial twist: every function, when applied to fewer arguments than the number of parameters returns a function!

Concept 4.3.1: Currying

A curried function can take its arguments, one at a time.
Each time we provide one or more (but not all) arguments, the function will return a new function which "expects" the remaining arguments. This process is repeated (think of it as recursive definition) until all arguments are provided, when the value computed by the function is returned.

To explain, lets consider again the tail recursive Fibonacci function:

Elm REPL

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

To avoid passing 0 and 1 to `f1` and `f2`, auxiliary functions were presented in Note 1.9.2 on page 15 and further refined to be hidden using `let ... in` in 2.4 on page 26.

The final refinement is to make use of *partial application* to make the code even shorter[1]:

Listing 4.3.1 of `Functions.elm (fibTailPf)` Elm code

```
fibTailPf : Int -> Int
fibTailPf =
  let
    fibTailHelper f1 f2 i = if i == 0 then f2 else fibTailHelper f2 (f1 + f2) (i - 1)
  in
    fibTailHelper 0 1
```

Here we use a style called *point-free*, where the main goal is to hide the parameters (points) the function is applied to. In this case the argument would be the index of the desired Fibonacci number, `n` which is skipped in the listing above, which should be passed as third argument (i.e. for `i`) to `fibTailHelper`. Since `fibTailHelper` is curried, we can omit the third argument to return a function that takes one argument (i.e. `fibTailHelper 0 1` returns a function of type `number -> number`) and returns the n^{th} Fibonacci number.

Note 4.3.1

The point-free style should (and throughout this lab guide will) be used sparingly, because it can easily obfuscate the meaning of a function.

Note 4.3.2

The order of parameters influences greatly the cases when we can use partial application in a concise matter!

4.3.2 Lambdas and closures

The final tool to make working with functions ergonomic and concise are *lambda expressions*. These are anonymous (nameless) functions, that are meant to be **short** and **used as arguments to other functions**.

Consider how we would use the function `applyTwice` defined above:

Elm REPL

```
> let f x = x + 1 in (applyTwice f 1)
3 : number
```

We had to define the function `f` in a local scope using a `let ... in` expression to avoid shadowing issues in the REPL.

[1]Yes, by 3 whole characters. Functional programmers are **very** stingy with characters.

Using lambda expressions, the code is much shorter and clearer:

```
Elm REPL
> applyTwice (\x -> x + 1) 1
3 : number
```

The general syntax for lambda expression is (in the code we use the \ symbol for λ):

$$\lambda\ param_1 \to \lambda\ param_2 \to \ldots \to \lambda\ param_n \to body$$

Which also shows that lambdas also use partial application to build lambdas with multiple curried parameters:

```
Elm REPL
> (\x -> \y -> \z -> z + y + z) 1 2 3
6 : number
```

So `(\x -> \y -> \z -> z + y + z) 1 2 3` is read as "A function which has a parameter named `x` and returns a function which has a parameter named `y` which returns a function which has a parameter named `z` that return the sum of `x`, `y` and `z`."

There is also a shorthand notation for lambdas with multiple parameters:

$$\lambda\ param_1\ param_2\ \ldots\ param_n \to body$$

So `\x -> \y -> \z -> z + y + z` could also be written as `\x y z -> z + y + z`.

4.3.3 Closures

Concept 4.3.2: Closures

A closure is function that *captures its environment* when it is created.
In FP languages, closures must be local definitions as the environment they can capture consists of the parameters and local definitions of the function they are defined in.

An important aspect to note is that the `x` and `y` are both available to use in the last lambda expression (i.e. the one with the `z` parameter). This means this last function has access to data that wasn't passed to it directly. This can be used to create functions that "set up" other functions with some (but not all) arguments to shorten some operations:

Listing 4.3.2 of Functions.elm (betweenClo) — Elm code

```
betweenClo : Int -> Int -> Int -> Bool
betweenClo lo hi =
  let
    betweenInner n = (lo <= n) && (n <= hi)
  in
    betweenInner
```

The `betweenClo` function takes 2 parameters that are used directly (not passed as arguments) by the `betweenInner` function that is defined locally and returns this inner function, which has to be applied to only one argument.

Elm REPL

```
> import Functions exposing (..)
> positive = betweenClo 0 ((2 ^ 31) - 1)
<function> : Int -> Bool
> positive 10
True : Bool
> positive -1
False : Bool
```

Here we use the `betweenClo` function to "set up" a new function, `positive`, which takes as argument a number and checks whether the number is between the 0 and $2^{31} - 1$. We can say that we captured the concept of a positive number by using more abstract concept of "number in a range".

4.3.4 Combinator functions

Combinator functions are functions with no free variables, or in other words a function that only refers to its arguments.

The `const` function

The simplest example is the `const` function: it takes one argument and returns a function which always returns this argument (i.e. the same result).

Listing 4.3.3 of `Functions.elm (const)` Elm code

```
const : a -> (b -> a)
const x = \_ -> x
```

The `flip` function

In Note 4.3.2 we noted that the order of parameters greatly influences how easily we can use partial application with some functions. Since we can't always control the order of parameters of a function (i.e. the function is defined by someone else) we need a convenience function to reverse the order of parameters to make our lives easier.

The `flip` function takes a function as argument and returns a function which takes the arguments of the first function in reverse order.

Listing 4.3.4 of `Functions.elm (flip)` Elm code

```
flip : (a -> b -> c) -> (b -> a -> c)
flip f = \x -> \y -> f y x
```

Consider the function `pow n i` which raises `n` to the i^{th} power. We would like to create two functions: `square` and `qube`, which are *particular cases* of `pow` with $i = 2$ and $i = 3$, respectively.

The `square` function is defined in the normal, "simple" style, while `qube` is defined using `flip` in a point-free style.

Listing 4.3.5 of `Functions.elm` (`pow, square, cubePf`) Elm code

```
pow : Int -> Int -> Int
pow n i = if i == 0 then 1 else n * pow n (i - 1)
square : Int -> Int
square n = pow n 2
cubePf : Int -> Int
cubePf = (flip pow) 3
```

The `uncurry` function

The `uncurry` function takes a curried function, which takes 2 arguments and returns a function which takes a 2-tuple:

Listing 4.3.6 of `Functions.elm` (`uncurry`) Elm code

```
uncurry: (a -> b -> c) -> ((a, b) -> c)
uncurry f = \(x, y) -> f x y
```

We will use it in the next section, for processing lists of tuples.

4.4 Lists - part 2

Building on the previous section about lists, here we will study some of the functions that are provided by the `List` module of the Elm core library.

Note 4.4.1

For clarity, most of the functions will be given in their non tail-recursive form.

4.4.1 Obtaining or removing a fixed length prefix: `take` and `drop`

The simplest way to generalize the `head` and `tail` functions we saw in section 3.4 on page 42 is add an additional parameter that determines how many elements to `take` or `drop`:

Listing 4.4.1 of `Lists.elm` (`take, drop`) Elm code

```
take : Int -> List a -> List a
take n l =
  if n <= 0 then
    []
  else
    case l of
      [] -> []
      x::xs -> x :: take (n - 1) xs
drop : Int -> List a -> List a
drop n l =
  if n <= 0 then
    l
  else
    case l of
      [] -> []
      _::xs -> drop (n - 1) xs
```

```
Elm REPL
> import Lists as L
> L.take 3 [1, 2, 3, 4, 5]
[1,2,3] : List number
> L.take 10 [1, 2, 3, 4, 5]
[1,2,3,4,5] : List number
> L.drop 3 [1, 2, 3, 4, 5]
[4,5] : List number
> L.drop 10 [1, 2, 3, 4, 5]
[] : List number
```

4.4.2 Obtaining or removing a prefix: `takeWhile` and `dropWhile`

We can generalize `take` and `drop` further to keep or remove elements while a *predicate* function, `p` returns `True`:

Listing 4.4.2 of `Lists.elm (takeWhile, dropWhile)` Elm code

```
takeWhile : (a -> Bool) -> List a -> List a
takeWhile p l =
  case l of
    [] -> []
    x::xs ->
      if p x then
        x :: takeWhile p xs
      else
        []
dropWhile : (a -> Bool) -> List a -> List a
dropWhile p l =
  case l of
    [] -> []
    x::xs ->
      if p x then
        dropWhile p xs
      else
        x::xs
```

```
Elm REPL
> import Lists as L
> L.takeWhile (\x -> x < 3) [1, 2, 3, 4, 1, 2]
[1,2] : List number
> L.dropWhile (\x -> x < 3) [1, 2, 3, 4, 1, 2]
[3,4,1,2] : List number
```

Here we used a lambda expression to provide the predicate. Notice that the elements are kept or discarded until the first time the predicate returns `False`.

Elm REPL

```
> import Lists as L
> import Functions as F
> L.takeWhile (F.const True) [1, 2, 3, 4, 1, 2]
[1,2,3,4,1,2] : List number
> L.dropWhile (F.const True) [1, 2, 3, 4, 1, 2]
[] : List number
> L.takeWhile (F.const False) [1, 2, 3, 4, 1, 2]
[] : List number
> L.dropWhile (F.const False) [1, 2, 3, 4, 1, 2]
[1,2,3,4,1,2] : List number
```

Here we used the combinator function `const` to create a function that always returns `True` and function that always return `False`.

4.4.3 Working with lists of tuples: `zip` and `unzip`

Sometimes we have two lists which are related in some way (e.g., a list containing ASCII codes and their corresponding characters) and we would like to create a single list from these lists, which preserves this relationship. We can use `zip` to create a list of tuples from two such lists, provided that the elements are "lined up" (i.e. order of elements is that same).

We can also obtain the individual lists from a list of tuples using the `unzip` function.

Listing 4.4.3 of `Lists.elm` `(zip, unzip)` Elm code

```elm
zip : List a -> List b -> List (a, b)
zip lx ly =
  case (lx, ly) of
    (x::xs, y::ys) -> (x, y)::(zip xs ys)
    _ -> []
unzip : List (a, b) -> (List a, List b)
unzip l =
  case l of
    [] -> ([], [])
    (x, y)::ls ->
      let
        (xs, ys) = unzip ls
      in
        (x::xs, y::ys)
```

Elm REPL

```
> import Lists as L
> L.zip [97, 98, 99] ['a', 'b', 'c']
[(97,'a'),(98,'b'),(99,'c')] : List ( number, Char )
> L.unzip [("Romania", "Bucharest"), ("Germany", "Berlin"), ("France", "Paris")]
(["Romania","Germany","France"],["Bucharest","Berlin","Paris"])
    : ( List String, List String )
```

4.4.4 Transforming lists with `map`

A very common operation on lists is changing each element by applying a function to it. To transform a list with elements of type `a` into a list with elements of type `b` we can use the `map` function:

Listing 4.4.4 of `Lists.elm (map)` Elm code

```elm
map : (a -> b) -> List a -> List b
map fn l =
  case l of
    [] -> []
    x::xs -> (fn x)::map fn xs
```

Elm REPL

```
> import Lists as L
> import Functions as F
> L.map (\x -> x + 1) [1, 2, 3]
[2,3,4] : List number
> L.map (L.take 1) [[1, 2], [3, 4, 5], [6, 7]]
[[1],[3],[6]] : List (List number)
> L.map (F.uncurry (+)) [(1, 2), (3, 4), (5, 6)]
[3,7,11] : List number
```

4.4.5 Filtering lists with `filter`

The second common operation on lists is to remove some elements that don't match a predicate. To obtain a new list with only the elements that match a certain predicate, we can use the `filter` function:

Listing 4.4.5 of `Lists.elm (filter)` Elm code

```elm
filter : (a -> Bool) -> List a -> List a
filter pred l =
  case l of
    [] -> []
    x::xs ->
      if (pred x) then
        x::filter pred xs
      else
        filter pred xs
```

Elm REPL

```
> import Lists as L
> import Functions as F
> L.filter (\x -> x < 3) [1, 2, 3, 4, 1, 2]
[1,2,1,2] : List number
> L.filter (\x -> x >= 3) [1, 2, 3, 4, 1, 2]
[3,4] : List number
> L.filter (F.const True) [1, 2, 3, 4, 1, 2]
[1, 2, 3, 4, 1, 2] : List number
```

4.4.6 Processing lists with `foldl` and `foldr`

The final common operation on lists is to "summarize" the data in the list, by processing each element and returning one result. `foldl` and `foldr` take an operation and a starting value as parameters, then apply the `op` on the first or last element and the starting value, then apply `op` on the next element and the result of the first application, until only one result remains.

Listing 4.4.6 of `Lists.elm` (`foldr, foldl`) Elm code

```elm
foldr : (a -> b -> b) -> b -> List a -> b
foldr op start l =
  case l of
    [] -> start
    x::xs -> op x (foldr op start xs)
foldl : (a -> b -> b) -> b -> List a -> b
foldl op start l =
  case l of
    [] -> start
    x::xs -> foldl op (op x start) xs
```

Question 4.4.1 *

Are `foldr` and `foldl` tail recursive?

Elm REPL

```
> import Lists as L
> L.foldl (::) [] [1, 2, 3]
[3,2,1] : List number
> L.foldr (::) [] [1, 2, 3]
[1,2,3] : List number
> sum = L.foldl (+) 0
<function> : List number -> number
> sum [1, 2, 3]
6 : number
```

4.4.7 Checking if all or any elements match a predicate with `all` and `any`

Sometimes we only need to know if at least one element or all elements in a list match a given predicate. The `all` and `any` functions are defined exactly for this purpose:

Listing 4.4.7 of `Lists.elm` (`all, any`) Elm code

```elm
all : (a -> Bool) -> List a -> Bool
all pred l =
  case l of
    [] -> True
    x::xs ->
      if pred x then
        all pred xs
      else
        False
any : (a -> Bool) -> List a -> Bool
any pred l =
  case l of
    [] -> False
    x::xs ->
      if pred x then
        True
      else
        any pred xs
```

```
Elm REPL
> import Lists as L
> L.all (\x -> x > 1) []
True : Bool
> L.any (\x -> x > 1) []
False : Bool
> L.all (\x -> x > 3) [4, 5, 6]
True : Bool
> L.any (\x -> x > 3) [1, 2, 3]
False : Bool
```

4.4.8 Sorting

Quicksort

Perhaps Quicksort is used as one of the most prominent examples of algorithms implemented in Functional Programming languages. Notice that the implementation focuses on the description of the algorithm itself and not on implementation details, like indices or element swaps.

Listing 4.4.8 of `Lists.elm (partition, quicksort)` Elm code

```
partition : comparable -> List comparable -> (List comparable, List comparable)
partition pivot l =
  (filter (\x -> x < pivot) l, filter (\x -> x >= pivot) l)
quicksort : List comparable -> List comparable
quicksort l =
  case l of
    [] -> []
    x::xs ->
      let
        (less, greater) = partition x xs
      in
        (quicksort less) ++ [x] ++ (quicksort greater)
```

4.5 Strings

In Elm, a string is a sequence of Unicode characters. The most important function that we will use on `String`s is `String.toList` which returns a `List Char`. Then we can process this list using the functions defined above. Some of them will also be defined directly on the `String` type, like `map`, `filter`, `foldl`, `foldr`, `all` and `any`.

4.6 Practice problems

Exercise 4.6.1 *

Implement a function `enumerate`, with the signature `enumerate : List a -> List (Int, a)` which returns a list of tuples, where the first member of the tuple is the index of the element in the list and second member is the element at that position in the list.

```
Elm REPL
> enumerate [1, 2, 3]
[(0, 1), (1, 2), (2, 3)] : List (Int, number)
> type Day = Mon | Tue | Web | Thu | Fri | Sat | Sun
> enumerate [Mon, Tue, Web, Thu, Fri, Sat, Sun]
[(0, Mon), (1, Tue), (2, Web), (3, Thu), (4, Fri), (5, Sat), (6, Sun)]
    : List (Int, Day)
```

Exercise 4.6.2 *

Implement a function `repeat n elem`, with the signature `repeat : Int -> a -> List a` which returns a list that contains `elem` n times.

```
Elm REPL
> repeat 4 1
[1, 1, 1, 1] : List number
> repeat 2 "Hello"
["Hello", "Hello"] : List String
```

Exercise 4.6.3 *

Implement a function `countVowels`, with the signature `countVowels : String -> Int` which returns the number of vowels in a `String`.

Hint:
Use the `String.toList` function to obtain the characters of a `String`.

Exercise 4.6.4 **

Implement the `partition` function without using any other functions defined on lists (i.e. from scratch).

Exercise 4.6.5 **

Write a function with the signature `countriesWithCapital : List (String, String) -> (String -> Bool) -> List String` that takes a list of `(Country, Capital)` tuples and returns the list of countries whose capital city name matches a given predicate.

```
Elm REPL
> countries = [("Romania", "Bucharest"), ("Germany", "Berlin"), ("France", "Paris")]
> countriesWithCapital countries (\s -> (String.left 1 s)  == "B")
["Romania", "Germany"]
> countriesWithCapital countries (\s -> (String.length s)  <= 5)
["France"]
```

Exercise 4.6.6 **

Implement a function with the signature `filterMap : (a -> Maybe b) -> List a -> List b` which combines the functionality of `filter` and `map`: it applies the function received as first arguments to each element an only keeps the elements that are wrapped in the `Just` variant.

```
Elm REPL

> oddLastDigit x = if modBy 2 x == 1 then Just (modBy 10 x) else Nothing
<function> : Int -> Maybe Int
> filterMap oddLastDigit [1, 2, 3]
[1, 3] : List number
> filterMap oddLastDigit [21, 2, 13]
[1, 3] : List number
```

Exercise 4.6.7 **

Implement the `all` and `any` functions by using only `foldl`.

Exercise 4.6.8 **

Implement a function `chunks n l` with the signature `chunks: Int -> List a -> List (List a)` which splits the list `l` in chunks of length `n`.

```
Elm REPL

> chunks 2 [1, 2, 3, 4, 5, 6]
[[1, 2], [3, 4], [5, 6]] : List (List number)
> chunks 3 [1, 2, 3, 4]
[[1, 2, 3], [4]]: List (List number)
```

Exercise 4.6.9 **

You may have noticed that the `createDate` function is not always correct: it doesn't handle leap years!

1. Modify the `Date.elm` module, by adding a new function with the signature `isLeapYear : Int -> Bool`, that checks if a given year between 1970 and 3000 is a leap year.
2. Then modify the `daysInMonth` function to account for leap years.
3. Finally update the `createDate` function to use the corrected version of `daysInMonth`.

Hint:
You should change the signature to `daysInMonth : Month -> Int -> Int`, where the second parameter is the year.

Exercise 4.6.10 ***

Implement the `enumerate` by using only `foldl`.

Exercise 4.6.11 ***

Implement a function `collect l` with the signature `collect : List (Result err ok) -> Result err (List ok)` which takes a list of `Result`s and returns the first element that is in the `Err` variant, or if all elements are in the `Ok` variant, then list of all unwrapped values, with the list wrapped in the `Ok` variant.

Elm REPL

```
> collect [Ok 1, Ok 2]
Ok [1, 2] : Result error (List number)
> collect [Ok 1, Err 2, Ok 3]
Err 2
```