Lab 1

- ++ to concatenate strings ex. “Hello” ++ “World” => “Hello World”
- Elm has type inference
- Variable shadowing is an error
- Function signatures
    - funcName : Pram1Type -> … -> ParamNType -> ReturnType
- If experssions
    - Else branch is mandatory!!!!!
- Tail recursion
    - Concept: 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)

Lab 2

- Tuples
    - Are limited by design to contain at most 3 items
    - Can contain different types of data
    - Help us keep related data close together, or pair up related values temporarily
- Records
    - Collection of named fields
    - ex. { firstName : String, lastName : String }
    - Call function

```
> fullTitle person = (if person.idDr then "Dr. " else "") ++
|   person.firstName ++ " " ++ person.lstName
|
<function>
    : { a | firstName : String, idDr : Bool, lstName : String } -> String
> fullTitle {firstName = "Haskell", idDr = True, lstName = "Curry"}
"Dr. Haskell Curry" : String
>
```

- Type aliases
  - Used to give a new name to existing types (in addition to the existing name)
  - Most common used to give name to records ex. type alias User = {firstName: String, lastName: String}
  - Create instances
    - v1

```
> type alias User = {firstName: String, lastName: String}
> fullName : User -> String
| fullName person = person.firstName ++ " " ++ person.lastName
|
<function> : User -> String
> fullName {firstName = "Cristina", lastName = "Nilvan"}
"Cristina Nilvan" : String
> fullName User "Cristina" "Nilvan"
```

    - v2

```
> cristina = User "Cristina" "Nilvan"
{ firstName = "Cristina", lastName = "Nilvan" }
    : User
> fullName cristina
"Cristina Nilvan" : String
```

- Type definitions
  - Allow us to create **new types**
  - Used to create enumerated types
  - ex.
    - type Color = Red | Green | Blue
    - type Point = Point Int Int
      - first Point = type name
      - second Point = constructor name
- Sum types and Product types
  - For the sum types, the cardinality is equal to the number of variants of the given type
    - Color type has cardinality = 3
    - Int ^ a cardinality = 2 ^ 32

   - For product types, the cardinality is equal to the product of the cardinality of each field
      - Point type has two Int fields => cardinality = 2 ^ 32 * 2 ^ 32

The variants can also contain data, which is similar to how one might use **unions** in C:

Listing 2.3.1 of **Shapes.elm (Shape)** Elm code

```elm
type Shape
  = Circle Float
  | Rectangle Float Float
  | Triangle Float Float Float
```

It can be **very beneficial** to use records in variants for clarity in the names:

Listing 2.3.2 of **Shapes.elm (ShapeRec)** Elm code

```elm
type ShapeRec
  = CircleRec { radius : Float }
  | RectangleRec { width : Float, height : Float }
  | TriangleRec { sideA : Float, sideB : Float, sideC : Float }
```

- Let … in
   - With let … in we can declare bindings and use them in a local scope
- Destructuring
   - Listing 2.5.1 of **Types.elm (Person, fullName)** Elm code

     ```elm
     type Person = Person String String
     fullName : Person -> String
     fullName (Person firstName lastName) = firstName ++ " " ++ lastName
     ```

   - Listing 2.5.3 of **Types.elm (PersonRec, fullNameRec)** Elm code

     ```elm
     type alias PersonRec = {firstName: String, lastName: String}
     fullNameRec : PersonRec -> String
     fullNameRec {firstName, lastName} = firstName ++ " " ++ lastName
     ```

   - _ : for the variables we don't want to destructure

- Case
    - _ ⇔ others
    - Patterns are checked from top to bottom until one matches and that branch is chosen
    - Multiple conditions case
        - case (windSpeed < 61, cloudLayer < 1400) of

Lab 3

- Type variables
    - Variable ranging over types
    - Names are in lowercase and may contain more than just one character
    - Can appear more than once to indicate that these values must have the same type, but this type cand be any type
- Equality
    - By default, Elm automatically implements **deep structural equality** for all values through the == operator
- Maybe type
    - For nullability
    - type Maybe a = Just a | Nothing
- Result type
    - For failure
    - type Result = Ok value | Err error
    - Signaling the possibility of failure
        - Using a string to return an error message
        - Defining an enum type that represents each possible error
- Lists part 1
    - Define lists
        - [1, 2, 3]
        - 1 :: 2 :: 3 :: []

Lab 4

- Controlling exported items
  - module NumeFisier exposing (functionName, TypeName(..) …)
- Open imports
  - import NumeFisier
  - NumeFisier.functionName
- Qualified imports
  - import NumeFisier as n
  - n.functionName
  - import NumeFisier as n exposing(etc)
- Higher order functions
  - A function which manipulates other functions: it either takes other functions as parameters or returns a function
- Partial application and Currying
  - Every function, when applied to fewer arguments than the number of parameters returns a function
  - Currying
    - A curried function can take its arguments, one at the time
    - Each time we provide one or more (but not all arguments), the function will return a new function which expects the remaining arguments

```
> mul3 a b c = a * b * c
<function> : number -> number -> number -> number
> mul3 1
<function> : number -> number -> number
> mul3 1 2
<function> : number -> number
> mul3 1 2 3
6 : number
```

  - Point free
    - the main goal is to hide the parameters (points) the function is applied to

- Lambdas and closures
  - Lambda expressions
    - Syntax: \arguments -> returnedValue
    - Ex. with multiple arguments

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

  - Closures
    - A function that captures its environment when it is created
    - 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
- Combinator functions
  - Functions that only refer to their arguments
  - Const function
    - Takes one argument and returns a function which always returns this argument
  - Flip function
    - Takes a function as argument and returns a function which takes the arguments of the first function in reverse order
  - Uncurry function
    - Takes a curried function, which takes 2 arguments and returns a function which takes a 2-tuple
- Lists part 2
  - Take
    - Take the first *n* members of a list
  - Drop
    - Drop the first *n* members of a list
  - Take while
    - Take the members of a list while a predicate function
  - Drop while
    - Drop the members of a list while a predicate function

- Zip and unzip

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)
```

- Map

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
```

- Filter

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
```

- Foldl and Foldr

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

```
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
```

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
```

- Strings
  - String.toList

- All and Any

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
```