Lab 3

Elm - Useful built-in data types and functions

Goals

In this lab you will learn to:

1. Use type variables to define generic types and functions
2. Use type constraints to restrict the types a type variable can take
3. Use the `Maybe` type to express nullability (the possibility that a value is missing)
4. Use the `Result` type to indicate the possibility of failure
5. Build, transform and process lists

Resources

Table 3.1: 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/`

3.1 Type variables and constraints

Do you notice any inconsistencies in the inferred function signatures?

Elm REPL

```
> isEq a b = a == b
<function> : a -> a -> Bool
> isGt a b = a > b
<function> : comparable -> comparable -> Bool
```

Some type names are lowercase letters (like `a`), some are lowercase names (like `comparable`) and some are CamelCase (like `Bool`). Lowercase letters are *type variables* and lowercase names are *type constraints*. These help us write code that is *generic* over multiple types.

3.1.1 Type variables

A type variable is a variable ranging over types. It can be used in function signatures, as you've seen above for `isEq`. A type variable names are lowercase and may contain more than just one character (`t1`, `elem`, `type1` are all valid names).

In function signatures type variables can appear more then once to indicate that these values must have the same type (but this type can be *any* type).

Listing 3.1.1 of `Generic.elm (reverseTuple)` Elm code

```elm
reverseTuple : (a, a) -> (a, a)
reverseTuple (a, b) = (b, a)
```

It can also be used in type declarations to create *generic containers*. The simplest such type is just a wrapper around another type:

Listing 3.1.2 of `Generic.elm (Box)` Elm code

```elm
type Box a = Box a
```

To use this type we can write the following (mostly useless) function:

Listing 3.1.3 of `Generic.elm (unboxInt)` Elm code

```elm
unboxInt : Box Int -> Int
unboxInt (Box i) = i
```

A common pattern with such containers is to use type aliases to create types for common "configurations":

Listing 3.1.4 of `Generic.elm (pairAlias)` Elm code

```elm
type Pair a b = Pair a b
type alias IntPair = Pair Int Int

addIntPair : IntPair -> Int
addIntPair (Pair a b) = a + b
```

Note 3.1.1

Type variables and function parameters can have the same name, but that doesn't mean that they are related in any way!
The following functions are semantically the same:

Listing 3.1.5: makeTuple — Elm code

```
makeTuple : a -> b -> (a, b)
makeTuple a b = (a, b)
--
makeTuple : fst -> snd -> (fst, snd)
makeTuple a b = (a, b)
--
makeTuple : b -> a -> (b, a)
makeTuple a b = (a, b)
--
makeTuple : x -> y -> (x, y)
makeTuple elem1 elem2 = (elem1, elem2)
```

Question 3.1.1 *

What Java concept is the equivalent to type variables? What about C++?

3.1.2 Type constraints

There is actually one restriction related to the names of type variables: they can't be[1] : "appendable", "number" or "comparable" because these are the names of *type constraints*. A type variable by itself means "any type", which sometimes can be too general to be useful.

Indeed consider functions like `compare` and operators, like `(<)`, `(>)`, which should be able to compare any two instances of the same type. Let's check their signature:

Elm REPL
```
> (<)
<function> : comparable -> comparable -> Bool
> compare
<function> : comparable -> comparable -> Order
```

Here `comparable` indicates that: the first and second parameters must have the same type **and** must be comparable.

As a final example consider the function `parAdd` which takes two pairs of numbers and adds the two elements of each tuple:

Elm REPL
```
> parAdd (a, b) (c, d) = (a + b, c + d)
<function> : ( number1, number1 ) -> ( number, number ) -> ( number1, number )
```

Notice how in the inferred type the elements of the first tuple and their result are all *related* since they have type `number1` (**with a** 1), while the elements of the second tuple and their result have type `number` (**without a** 1).

[1]Actually they can't be reserved names or the reserved names above with a number after them (`number1`).

The tree main type constraints in Elm are:

- `appendable`: types that can be appended: `String` and `List`
- `number`: types that are numbers: `Int` and `Float`
- `comparable`: types that can be compared: numbers, `Char`, `String`, `List` of comparables, and tuples of comparables.

Note 3.1.2

In Elm, there is (by design) no way to make new user defined types that satisfy the 3 built-in type constraints. You can implement custom functions, but they have to be used explicitly.

Question 3.1.2 **

Can we constrain type variables in Java or C++? If yes how?

3.2 Equality

Concept 3.2.1: Types of equality

Programming languages often implement two types of equality:
Reference equality: only the address to which the references (pointers) point is compared
Structural equality: the contents (fields) of the two objects are compared, using their equality functions, which, depending on the language, may perform different types of comparisons:
→ Shallow: only fields of the top-level object are compared, using reference equality
→ Deep: each field is recursively compared with deep equality, using its equality function (which, for example can compare the data byte-by-byte)
These equality functions might be customizable (e.g. overridable in Java) too.

After seeing the list above you're probably wondering: "is there no type constraint for equality?" or "in order to check if two values are equal do they also need to have an order relationship defined?".

By default, Elm *automatically* implements *deep structural equality* for all values through the `==` operator. This means that we can always compare two instances of the same type for equality.

Concept 3.2.2: Immutability and equality

When we have immutable data, with some clever optimizations, we can always check for structural equality using reference equality.
This is because we know that the data is never modified directly, but a new copy with the modifications is created. If we make sure not to have two copies (i.e. a series of bytes representing the same data, in two different regions of memory) of the same data, we can always use reference equality to check for full (deep) structural equality.

Note 3.2.1

The default implementation for `==` can't be modified. To compare only a subset of fields, you have to implement your own custom equality function and use it explicitly.

Question 3.2.1 ***

How is equality handled for float types? Try to evaluate in the REPL: `(0/0)` and then `(0/0) == (0/0)`. Does this cause a problem for reference equality?

3.3 Case study: types for error handling

Now that we understand type variables, let's finally understand some more useful and interesting types.

At this point you should know enough to navigate the documentation of the Elm standard library here: `https://package.elm-lang.org/packages/elm/core/1.0.5/Basics`.

3.3.1 Signaling the possibility of nullability: the `Maybe` type

Question 3.3.1 **

What is known as the "billion dollar mistake" in Computer Science?

Concept 3.3.1: Nullability

In programming languages like C, C++ and Java, we use the null value to indicate that a pointer or reference is invalid.

In chapter 1 we noted that functions must handle every possible input. But what happens when there is no well-defined output for a certain input?

Consider a the famous division by 0 case: what should happen when we try to divide by zero?

```
                                                    Elm REPL
> 1 / 0
Infinity : Float
> 1 // 0
0 : Int
> modBy 0 10
Error: Cannot perform mod 0. Division by zero error.
```

As you can see we get different results depending on the types (`Int` or `Float`) we're working with, including even a runtime error, something that should not happen in Elm.

Another example would be the `heron` function from chapter 2. The function is not 100% correct, because not only can we call it with arguments that represent three sides that can't form a triangle (e.g. `heron 1 1 10`), but we can also call it with negative arguments!
So what should the function return in this case?

A first thought might be to just return 0, signaling that something is probably not right. But this creates confusion in the case when we call the function with arguments that represent the sides of a degenerated triangle, formed from collinear point (i.e. `heron 2 2 4` should be 0 because the area is 0, not because the input is invalid).

Another solution to better differentiate between outputs for valid and invalid input is to return a result that should be impossible to get from valid inputs, like the classic -1. This does solve the previous problem of ambiguous outputs, but creates a new problem: this convention has to be clearly documented and taken into consideration when using the function. Bugs like forgetting to test the result and passing it to a function that expects a positive integer can easily creep in.

The solution in Functional Programming is to use *enumerated types* to represent the two possible outcomes:

- a well defined result
- no result

This concept is implemented in Elm with the `Maybe` type:

Listing 3.3.1: Maybe's type definition Elm code

```elm
type Maybe a
    = Just a
    | Nothing
```

The `Just` variant represents the case of the well defined result and the `Nothing` variant represents the "no result" case.

The main advantage of using this type is that we need to **explicitly check** the result of the function, otherwise we will get a compile error.

As an example, lets address the problems with the `heron` function by changing the return type to `Maybe Float` and returning `Nothing` if the arguments are negative or can't form a valid triangle:

Listing 3.3.2 of `Shape.elm (validTriangle, safeHeron)` Elm code

```elm
validTriangle a b c =
  ((a > 0) && (b > 0) && (c > 0)) &&
  ((a + b >= c) && (a + c >= b) && (b + c >= a))
safeHeron : Float -> Float -> Float -> Maybe Float
safeHeron a b c =
  if not (validTriangle a b c) then
    Nothing
  else
    Just (heron a b c)
```

Elm REPL

```
> import Shape exposing (..)
> safeHeron 1 1 1
Just 0.4330127018922193 : Maybe Float
> safeHeron 2 2 10
Nothing : Maybe Float
> safeHeron -2 3 4
Nothing : Maybe Float
```

Now when we try to use the function with invalid input values it returns `Nothing`, meaning that the result is undefined. When the inputs are valid the result is returned wrapped in the `Just` variant.

Question 3.3.2 **

Can we know at **compile time** if any given pointer is null or not in C?
Are C++ references different?

Question 3.3.3 ***

Do you know any language that has the solution to the nullability problem *built-in as language feature*?

3.3.2 Signaling the possibility of failure: the `Result` type

The previous approach (i.e., using `Maybe`) has a disadvantage: the same return variant is used to represent all errors. Therefore, when multiple errors can arise, we don't know precisely which one has caused the function to fail.

To solve this problem, we can use the `Result` type, which is can wrap a successful result in the `Ok` variant and an error type in the `Err` variant:

Listing 3.3.3: Result's type definition — elm code

```elm
type Result error value
    = Ok value
    | Err error
```

A good use case for this type would be the `area` function, which can fail if:

- the radius of the circle is negative
- any of the rectangle's width or height is negative
- the sides of the triangle are negative or can't form a valid triangle

There are two main approaches for returning errors:

Using a string to return an error message

Listing 3.3.4 of Shape.elm (safeArea) Elm code

```elm
safeArea : Shape -> Result String Float
safeArea shape =
  case shape of
    Circle radius ->
      if radius < 0 then
        Err "Negative circle radius"
      else
        Ok (pi * radius * radius)
    Rectangle width height ->
      if (width < 0) || (height < 0) then
        Err "Negative rectangle width or height"
      else
        Ok (width * height)
    Triangle a b c ->
      case safeHeron a b c of
        Just area -> Ok area
        Nothing -> Err "Sides can't form a triangle"
```

For each error case we either return the result wrapped in the `Ok` variant or return an error message stating the problem in the `Err` variant.

We'll check the result of `safeHeron` with a `case` expression to see if the function returned a value in the `Just` variant, in which case we rewrap it in the `Ok` variant of `Result`, or in the `Nothing` variant, when we'll know the function has failed because the sides can't form a valid triangle.

Elm REPL

```
> safeArea (Triangle 2 2 3)
Ok 1.984313483298443 : Result String Float
> safeArea (Triangle 2 2 10)
Err ("Sides can't form a triangle") : Result String Float
> safeArea (Rectangle 2 10)
Ok 20 : Result String Float
> safeArea (Rectangle 2 -10)
Err ("Negative width or height") : Result String Float
> safeArea (Circle 2)
Ok 12.566370614359172 : Result String Float
> safeArea (Circle -2)
Err ("Negative circle radius") : Result String Float
```

Defining an enum type that represents each possible error

The previous approach still has one problem: the caller of the function has to process the returned string in order to actually be able to handle the error.

To solve this problem, we can define an enumerated type with a variant for each error, that the caller can pattern match to handle the error programatically.

For example, to represent the possible errors that can occur when calculating the area of a triangle, we can define the following types to represent all possible errors:

Listing 3.3.5 of `Shape.elm (InvalidTriangleError, TriangleSide)` Elm code

```elm
type InvalidTriangleError
 = NegativeSide TriangleSide
 | ImpossibleTriangle
type TriangleSide = A | B | C
```

And then return the appropriate type for each error:

Listing 3.3.6 of `Shape.elm (safeHeronEnum)` Elm code

```elm
safeHeronEnum : Float -> Float -> Float -> Result InvalidTriangleError Float
safeHeronEnum a b c =
  if (a < 0) then
    Err (NegativeSide A)
  else if (b < 0) then
    Err (NegativeSide B)
  else if (c < 0) then
    Err (NegativeSide C)
  else if ((a + b < c) || (a + c < b) || (b + c < a)) then
    Err ImpossibleTriangle
  else Ok (heron a b c)
```

Elm REPL

```
> safeHeronEnum 1 1 1
Ok 0.4330127018922193 : Result InvalidTriangleError Float
> safeHeronEnum 1 2 1
Ok 0 : Result InvalidTriangleError Float
> safeHeronEnum 1 -1 3
Err (NegativeSide B) : Result InvalidTriangleError Float
> safeHeronEnum 1 3 1
Err ImpossibleTriangle : Result InvalidTriangleError Float
```

In both cases, functions that return errors can be easily composed:

- In the case of string errors, we can return the error without any changes, or “append” some extra context to the string.
- In the case of enumerated errors, we can wrap the error type of the called function in a variant of the error type of the caller function. (e.g. in the case of `safeHeronEnum` we wrap `InvalidTriangleError` with the `InvalidTriangle` variant)

Question 3.3.4 **

Discuss at least 2 advantages and disadvantages of each approach. In which cases would you use one over the other?

3.4 Lists - part 1

In Elm (as well as other modern Functional Programming languages) lists are *singly linked lists*, which are defined as:

Listing 3.4.1: List definition — Elm code

```elm
type List a
  = Cons a (List a)
  | Nil
```

With the definition above, we can create lists as follows:

Elm REPL

```
> Nil
Nil : List a
> Cons 1 Nil
Cons 1 Nil : List number
> Cons 1 (Cons 2 Nil)
Cons 1 (Cons 2 Nil) : List number
```

We can use the constructors to build lists:

Elm REPL

```
> countFromTo a b = if a >= b then Nil else Cons a (countFromTo (a+1) b)
<function> : number -> number -> List number
> countFromTo 1 5
Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```

and to process lists using pattern matching:

Elm REPL

```
> sumOfElements l =
|   case l of
|     Nil -> 0
|     Cons x xs -> x + sumOfElements xs
|
<function> : List number -> number
> sumOfElements (countFromTo 1 10)
45 : number
```

As you can see, both defining and printing lists in this form is quite verbose and hard to understand. Thus, Elm (as well as many (not just) Functional Programming languages) has *syntax sugar* for working with lists.

Concept 3.4.1: Syntax sugar

Many programming languages provide features that help in writing common patterns in a concise manner. One might say that the resulting syntax for achieving these tasks is “sweet and short”, hence the name syntax sugar.
The process by which the compiler processes these constructs is called *desugaring*, a transformation from the concise form into a form which uses only a few primitive constructs.
Examples include operator overloading in C++ and lambda expressions in Java.

First, we can define *list literals* between brackets:

```
                                                        Elm REPL
> [1, 2, 3]
[1,2,3] : List number
```

Second, we have the `(::)` operator (read as "cons") for constructing lists in a more readable fashion:

```
                                                        Elm REPL
> (::)
<function> : a -> List a -> List a
> 1 :: 2 :: 3 :: []
[1,2,3] : List number
```

Finally, you can see above that `Nil` can also be replaced by the empty list literal `[]`.

So the functions defined above would normally look like:

Listing 3.4.2 of `Lists.elm (countFromTo, sumOfElements)` Elm code

```elm
countFromTo : Int -> Int -> List Int
countFromTo from to =
  if from >= to then
    []
  else
    from :: countFromTo (from + 1) to
sumOfElements : List Int -> Int
sumOfElements l =
  case l of
    [] -> 0
    x::xs -> x + sumOfElements xs
```

Exercise 3.4.1 *

Write a function `len` that returns that length of a list (i.e., the number of elements in it).

3.4.1 Working with lists efficiently

Question 3.4.1 *

What is the time complexity of the following operations on a singly linked list:

1. Insert at the list beginning (head)
2. Insert at the list end (tail)
3. Get the i$^{\text{th}}$ element

One of the aspects discussed in 1.9 on page 13 is that recursive functions can easily run out of stack space if they are not written in a tail recursive style.

```
                                                        Elm REPL
> countFromTo 1 10000
RangeError: Maximum call stack size exceeded
```

Exercise 3.4.2 *

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

Before we look at the solution, it will be useful to take a look at how to reverse lists:

Listing 3.4.4 of `Lists.elm (reverse)` Elm code

```elm
reverse : List a -> List a
reverse l =
  let
    reverseAcc lx acc =
      case lx of
        [] -> acc
        x::xs -> reverseAcc xs (x::acc)
  in
    reverseAcc l []
```

Here we can notice that the accumulator parameter `acc` in the auxiliary function `reverseAcc` acts as a *stack*: each time the function calls itself recursively it adds one element on the top of it.

So, taking into consideration that, when we build a list in a tail recursive function with accumulators, that list will be reversed, we can count *down* in the auxiliary function to achieve the same output as the `countFromTo` function:

Listing 3.4.5 of `Lists.elm (countFromToTail)` Elm code

```elm
countFromToTail : Int -> Int -> List Int
countFromToTail from to =
  let
    cnt a b acc =
      if a >= b then
        acc
      else
        cnt a (b - 1) ((b - 1)::acc)
  in
    cnt from to []
```

Note 3.4.1

To work with singly linked lists efficiently, the preferred method of building lists is *appending elements in front of the list.*

3.4.2 Appending lists

If we want to perform more complex operations, we clearly need some more abstractions to work with lists.

The function that will be very useful is the `append` function:

Listing 3.4.6 of `Lists.elm (append)` Elm code

```elm
append : List a -> List a -> List a
append lx ly =
  case lx of
    [] -> ly
    x::xs -> x :: append xs ly
```

Here we need to traverse the first list in order to access the empty list marker (`[]`) and replace it with the second list.

We can also rewrite this in a tail recursive style, but then we need to reverse the first list before the appending each of its elements to the second list, because as we've noted list construction in tail recursion works in a similar fashion to a stack.

Listing 3.4.7 of `Lists.elm (appendTail)` Elm code

```elm
appendTail : List a -> List a -> List a
appendTail la lb =
  let
    appTail lx acc =
      case lx of
        [] -> acc
        x::xs -> appTail xs (x::acc)
  in
    appTail (reverse la) lb
```

Question 3.4.2 *

What is the algorithmic complexity of the `appendTail` function?

We also have an operator for appending lists: `(++)`:

Elm REPL

```
> ["Have", "a"] ++ ["nice", "day"]
["Have","a","nice","day"] : List String
```

Again to understand the importance of tail recursive functions, especially when operation on lists, consider the following task: append two lists that contain a range of integers and return its length (to avoid printing huge lists).

Elm REPL

```
> let
|   l1 = countFromTo 0 10000
|   l2 = countFromTo 20000 30000
| in
|   len (append l1 l2)
|
RangeError: Maximum call stack size exceeded
```

As you can see when we use the simple (non tail recursive) versions of the functions they will overflow the stack.

We can see regardless of how we combine these functions, if one of them is not tail recursive they will overflow the stack.

```
Elm REPL

> len (countFromTo 0 10000)
RangeError: Maximum call stack size exceeded
> lenTail (countFromTo 0 10000)
RangeError: Maximum call stack size exceeded
> lenTail (countFromToTail 0 10000)
10000 : number
```

```
Elm REPL

> let
|   l1 = countFromToTail 0 10000
|   l2 = countFromToTail 20000 30000
| in
|   lenTail (append l1 l2)
|
RangeError: Maximum call stack size exceeded
> let
|   l1 = countFromToTail 0 10000
|   l2 = countFromToTail 20000 30000
| in
|   lenTail (appendTail l1 l2)
|
20000 : number
```

3.4.3 The `head` and `tail` functions

Every decent linked list implementation provides a function that returns the first element of the list and a function that returns the list without the first element. In Elm these functions are called `head` and `tail`:

Listing 3.4.8 of `Lists.elm (head, tail)` Elm code

```elm
head : List a -> Maybe a
head l =
  case l of
    [] -> Nothing
    x::_ -> Just x
tail : List a -> Maybe (List a)
tail l =
  case l of
    [] -> Nothing
    _::xs -> Just xs
```

These are real examples of functions which use the `Maybe` type to signal that the function can't return a valid result for all inputs:

- What is the first element of an empty list?
- How can we skip the first element of an empty list?

In both cases, the easy way out is to just return `Nothing`.

3.5 Practice problems

Exercise 3.5.1 *

Write a function with the signature `safeDiv : Int -> Int -> Maybe Int` that returns `Nothing` when we try to divide by 0 and the result wrapped in `Just` otherwise.

Exercise 3.5.2 *

Rewrite the `len` function defined above in a tail-recursive style, with the name `lenTail`.

Exercise 3.5.3 *

Implement a function `last` that returns the last element of a list.

Exercise 3.5.4 *

Write a function `indexList i l` which returns the i^{th} element of the list `l`.

Exercise 3.5.5 **

Write a function `fibs start end`, which takes a two numbers, `start` and `end` and returns a list of the Fibonacci numbers such that:

$$fibs(start, end) = \{fib(i) | i \in \mathbb{N}, start \leq i < end\}$$

```
                                                     Elm REPL
> fibs 0 1
[1]: List number
> fibs 0 2
[1,1] : List number
> fibs 0 3
[1,1,2] : List number
> fibs 0 4
[1,1,2,3] : List number
> fibs 3 10
[3,5,8,13,21,34,55] : List number
```

Exercise 3.5.6 **

Modify the `fibs` function to return a list of tuples, where the first element in each tuple denotes the index of the Fibonacci number and the second the Fibonacci number itself.

```
                                                     Elm REPL
> fibs 0 2
[(0,1),(1,1)] : List ( number1, number )
> fibs 0 3
[(0,1),(1,1),(2,2)] : List ( number1, number )
> fibs 0 4
[(0,1),(1,1),(2,2),(3,3)] : List ( number1, number )
> fibs 3 10
[(3,3),(4,5),(5,8),(6,13),(7,21),(8,34),(9,55)] : List ( number1, number )
```

Exercise 3.5.7 ***

Write a function with the signature `cmpShapes : Shape -> Shape -> Result String Order` that uses the `safeArea` function to calculate the area of the 2 input shapes and returns the ordering between them wrapped in the `Ok` variant, if the area of both shapes can be calculated. Otherwise it should return an error message wrapped in the `Err` variant.

```
                                                                   Elm REPL
> cmpShapes (Circle 2) (Circle 3)
Ok LT : Result String Order
> cmpShapes (Triangle 2 2 2) (Circle 3)
Ok LT : Result String Order
> cmpShapes (Rectangle 4 4) (Circle 3)
Ok LT : Result String Order
> cmpShapes (Rectangle 6 6) (Circle 3)
Ok GT : Result String Order
> cmpShapes (Rectangle 2 3) (Triangle 3 4 5)
Ok EQ : Result String Order
> cmpShapes (Rectangle 2 3) (Triangle -3 4 5)
Err ("Invalid input for right shape: Sides can't form a triangle")
    : Result String Order
> cmpShapes (Rectangle -2 3) (Triangle -3 4 5)
Err ("Invalid input for left shape: Negative rectangle width or height")
    : Result String Order
```

Exercise 3.5.8 ***

Write a function with the signature: `totalArea : List Shape -> Result (Int, InvalidShapeError) Float` that uses the `safeAreaEnum` function to calculate the total area of the list input shapes. It should returns total area (the sum of all areas) in the `Ok` variant, if **all** the areas can be calculated. Otherwise it should return a tuple with the error returned by `safeAreaEnum` and the index of the shape that caused the error wrapped in the `Err` variant.

```
                                                                   Elm REPL
> totalArea [Circle 2, Circle 2]
Ok 25.132741228718345 : Result ( Int, InvalidShapeError ) Float
> totalArea [Circle 2, Circle 2, Rectangle 3 4, Triangle 3 4 5]
Ok 43.132741228718345 : Result ( Int, InvalidShapeError ) Float
> totalArea [Circle 2, Circle 2, Rectangle 3 4, Triangle 1 1 5]
Err (3,InvalidTriangle ImpossibleTriangle): Result ( Int, InvalidShapeError ) Float
> totalArea [Circle 2, Circle -2, Rectangle 3 4, Triangle 1 1 5]
Err (1,InvalidCircle): Result ( Int, InvalidShapeError ) Float
```