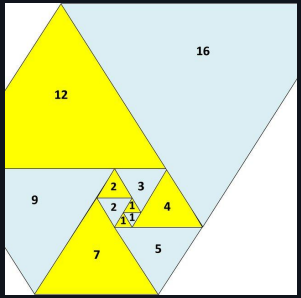
Lists

Mitsiu Alejandro
Carreño Sarabia



Agenda

- Recap
- Lists
- Loops
- Map

Recap

Code recap

What can we infer from the following type annotation?

```
something : Bool -> String
```

What does this function do?

```
getIdByName : String -> Int
getIdByName name =
    case name of
        "Fernanda" ->1
        "Miguel" ->2
        "Juan" ->3
        _ ->0
```

```
getGradeById : Int -> Float
getGradeById id =
    case id of
        1 ->9.4
        2 ->8.7
        3 ->9.3
        _ ->0
```

```
getFinalGrade : String -> Float
getFinalGrade name =
    getGradeById (getIdByName name)
```

Commands recap

Create an elm project?

- elm init

Start an elm repl session?

- elm repl

Track changes in git?

- git add <file>

Commit changes in git?

- git commit -m'description'

Push to remote repo in git?

- git push origin main

Verify elm code compilation?

- elm make src/*

Lists

Lists

To write more interesting functions, we have to introduce the primitive List

> For any type alpha, there's a type List alpha, which describes an ordered collection of 0 or more elements of type alpha

$$[\,] : \text{List}\ \ \alpha$$

So we have:

- List Int
- List String
- List List Int

For example:

- [1,2,3]
- ["Fernanda", "Juan", "Luis"]
- [[1,2,3]]

Lists

A list is two possible variants (two possible things):

- Nil: [] an empty list of type alpha
- Cons: (::) An operator such that:

$$x :: xs : \text{List } \alpha$$
$$if$$
$$x : \alpha$$
$$and$$
$$xs : \text{List } \alpha$$

So we can say:

```
2 :: []
```

Take take 2 and put it in this list as the first element

$$2 :: [\,] \cong [2]$$

Lists

In elm repl if we type (::) we get:

```
<function> : a -> List a -> List a
```

This is our first function with two parameters.

1. An element of type alpha
2. A list of type alpha

The output is of type List alpha (last arrow)

```
"Hi" :: [] : List String
```

Lists

When we write a list:

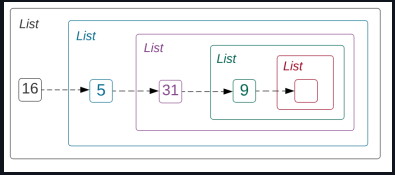
```
[16, 5, 31, 9] : List Int
```

We are actually doing this:

```
16 :: 5 :: 31 :: 9:: [] : List Int
```

Which will be right-associative:

```
16 :: (5 :: (31 :: (9 ::[]))) : List Int
```



Lists & Cases

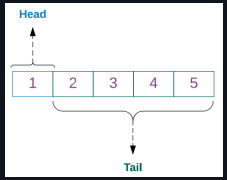
We just saw how to construct Lists, now lets learn how to deconstruct them, with cases.

Remember that cases have the notation:

$$
\begin{aligned}
&\text{case}\ \ e1 : \alpha\ \text{of} \\
&\quad pat1 : \alpha \text{ ->} \\
&\qquad e2 : \beta \\
&\quad pat2 : \alpha \text{ ->} \\
&\qquad e3 : \beta
\end{aligned}
$$

The pat stands for pattern which means we can case on structures rather than only on values:

```
case [1,2,3,4,5] of
    [] ->
        []
    x :: xs ->
        [x]
```



List

Let's make a quick function that uses the cons operator:

```
headAdder : a -> List a -> List a
headAdder newElement list =
    newElement :: list

headAdder 1.1 [1.2, 1.3]
```

List exercises

1. Make a function "isEmpty" that returns if a given String List is empty or not (Bool).
2. Make a function "head" that returns the first Int in a Int list of -100 if it's an empty list.

Loop

Our old ways

Remember how in imperative contexts we have:

```
1 int x = 0;
2 for (int i = 0; i < 5; i++){
3     x = x + 1;
4 }
```

- We agreed that x = x + 1 on line 3 is no bueno because we modify the value
- By the same principle i++ (i = i + 1) on line 2 must be gone

- Actually the whole concept of a for loop or while loop needs to be gone, so how is this done in functional programming?

Length

Let's use our new knowledge on lists to calculate the length of a given list
What's our type annotation?

```
length : List Int -> Int
```

We mentioned our list can be either one of two things...

```
length : List Int -> Int
length l =
    case l of
        [] ->
        x :: xs ->
```

Length

Current progress:

```
length : List Int ->
Int
length l =
    case l of
        [] ->
        x :: xs ->
```

- If our list is empty (nil) what's the length?

```
length : List Int -> Int
length l =
    case l of
        [] -> 0
        x :: xs ->
```

Length

Current progress:

```
length : List Int ->
Int
length l =
    case l of
        [] -> 0
        x :: xs ->
```

- If our list is not empty (x::xs) what should we do? How does it affect our length calculation?

```
length : List Int -> Int
length l =
    case l of
        [] -> 0
        x :: xs -> 1 + ??
```

At ?? do we care about x or xs?

We dont care about x because we already encoded it's value as 1 + something

Length

If we know that the list is not-empty, we add one and try to get the length of the rest of the list.

```
length : List Int -> Int
length l =
    case l of
        [] -> 0
        x :: xs -> 1 + length xs
```


Map

Map

Map is our first higher order function, it's one powerfull function that allows us to make a transformation to each element in our list. Let's start by analizing it's type:

```
List.map
<function> : (a -> b) -> List a -> List b
```

How many parameters does it have?

- Anwser 2:
  1. (a -> b)
  2. List a

Which type is the output?

- List b

What does the parameter (a -> b) mean?

▒▒▒ Map

Let's review our understanding with (a -> b): If I say

```
mystery : a -> b
```

What does it mean?

It means a function with input a (alpha) and output b (beta) Previously we used both alpha and beta as placeholder for a lot of types lets consider:

```
always2 : a -> Int
always2 x =
    2
```

How do we apply always2?

```
always2 1
always2 "1"
always2 'a'
always2 False
```

▒▒▒▒ Map

Let's track back into map.

```
List.map
<function> : (a -> b) -> List a -> List b
```

List.map is a function that receives a function and a list and returns other list.

This is a powerfull concept, functions are values, and as such we can pass them as parameters!

This notion has been adopted by a lot of programming paradigms & languages because is quite usefull.

Map

Let's make a simple function a->b with two different types.

```
cBool in =
    case in of
        0 ->
            False
        _ ->
            True
```

What's the type annotation of this function?

```
cBool : Int -> Bool
```

Now we have a type for a -> b let's replace on List.map:

- a => Int
- b => Bool

```
List.map
<function> : (a -> b) -> List a -> List b
```

```
List.map
<function> : (Int -> Bool) -> List Int ->
List Bool
```

Any idea on what List.map may do?

Map

1. (a -> b) = Performs a transformation on input a's and produces b's
2. List a = The transformation will be applied to each element in a list

```
cBool : Int -> Bool
cBool inp =
    case inp of
        0 -> False
        _ -> True

listA : List Int
listA = [-2,-1,0,1,2]

List.map cBool listA -- => [True, True, False, True, True]
```

Map Exercises

1. Create a function "canBuyAlcohol" that given an array of ages return True or False if is able to buy alcohol (+17)
2. Create a function "allUpperCase" that given an array of names return the same names in uppercase (String.toUpper)
3. Create a function "approveCourse" that given an array of grades (Float) return if it approve or not the course.