Lab 5

Elm - Common patterns and package management

Goals

In this lab you will learn to:

1. Create pipelines with the `|>` and `<|` operators
2. Compose functions with the `>>` and `<<` operators
3. Inspect partial results with `Debug.log` and `Debug.toString`
4. Understand record accessors
5. Use the record update syntax to modify record instances
6. Use `Maybe` and `Result` for more complex error handling
7. Install and use Elm packages
8. Test your code with elm-test

Resources

Table 5.1: Lab Resources

Resource	Link
Elm core library	`https://package.elm-lang.org/packages/elm/core/1.0.5/`
Elm `Debug` module	`https://package.elm-lang.org/packages/elm/core/1.0.5/Debug`
Elm package repository	`https://package.elm-lang.org/`
Elm `Test` package	`https://package.elm-lang.org/packages/elm-explorations/test/latest/`
Elm test runner	`https://github.com/rtfeldman/node-test-runner`

5.1 Pipelines: the |> and <| operators

Lets implement a function which returns the sum of the last digit of all odd elements in a list.

Listing 5.1.1 of `AdvancedLists.elm (sumOfOddLastDigits)` Elm code

```
sumOfOddLastDigits : List Int -> Int
sumOfOddLastDigits l =
  foldl (+) 0 (map (modBy 10) (filter (\x -> modBy 2 x == 1) l))
```

As you can see, this functions isn't something one would call "easily readable", because we had to write the operations in reverse order of their logical order.

To solve this problem, we have the pipeline operators `(|>)` and `(<|)`, which are simply defined as[1]:

Listing 5.1.2: The pipe operators Elm code

```
x |> f = f x
f <| x = f x
```

We can read this as: "`x` piped into `f`".

Elm REPL

```
import Lists as L
> [1, 2, 3] |> L.take 1
[1] : List number
> [1, 2, 3] |> L.drop 1 |> L.take 1
[2] : List number
> List.drop 1 <| List.take 1 [1, 2, 3]
[2] : List number
> 1 |> (\x -> x + 1)
2 : number
```

Of course, (ab)using pipe operators doesn't guarantee better readability.

The backward pipe `<|` is useful for providing the last argument of a function without parentheses:

Elm REPL

```
List.take 2 <| List.drop 2 [1, 2, 3, 4]
[3,4] : List number
```

The forward pipe `|>` is useful when we have a longer sequence of operations where we want to pass a value through a sequence of functions, and the `sumOfOddLastDigits` fits this description very well:

[1]Note that in Elm you can't define infix operators, so you cannot simply compile the code in Listing 5.1.2.

Listing 5.1.3 of `AdvancedLists.elm (sumOfOddLastDigitsPipe)` Elm code

```elm
sumOfOddLastDigitsPipe : List Int -> Int
sumOfOddLastDigitsPipe l =
  l
    |> filter (\x -> modBy 2 x == 1)
    |> map (modBy 10)
    |> foldl (+) 0
```

5.2 Function composition: the >> and << operators

You may remember from Math class that instead of writing $f(g(x))$ to denote nested functions, you could you the notation

$$(f \circ g)(x)$$

which is called *function composition*.

As we just discussed, Elm treats functions as first-class citizens, so it makes sense to have some way to easily compose functions. This functionality is provided by the function composition operators, `>>` and `<<`, which compose their arguments from left to right and right to left, respectively.

Listing 5.2.1: The function composition operators Elm code

```elm
f >> g = \x -> g (f x)
f << g = \x -> f (g x)
```

Elm REPL

```
> inc = \x -> x + 1
<function> : number -> number
> triple = \x -> x * 3
<function> : number -> number
> (inc >> triple) 1
6 : number
> (inc << triple) 1
4 : number
> (inc >> inc >> triple) 1
9 : number
```

Their main use case is to make calls like `\x -> h (g (f x))` more concise and easier to read `f >> g >> h`. For example, we can rewrite `applyTwice` using to the `>>` operator to make it shorter:

Elm REPL

```
> applyTwice f x = (f >> f) x
<function> : (a -> b) -> a -> b
> applyTwice (\x -> x + 1) 1
3 : number
```

But we can also use it to pass two composed functions to it:

```
Elm REPL
> applyTwice (inc >> inc) 1
5 : number
> applyTwice (inc >> triple) 1
21 : number
```

Exercise 5.2.1 *

Trace the evaluation of `applyTwice (inc >> triple) 1`, showing each evaluation step.

Exercise 5.2.2 *

Implement the `all` and `any` functions by using a pipeline with `map` then `foldl`.

Question 5.2.1 *

Which is the core difference between function composition and pipelines?

Hint:
What type does each return?

Question 5.2.2 **

In which cases is it more elegant to use:

- function composition (`>>` and `<<`) over pipelines (`|>` and `<|`)
- pipelines (`|>` and `<|`) over function composition (`>>` and `<<`)

Question 5.2.3 ***

How can we rewrite a function that uses:

- function composition (`>>` and `<<`) to a function that uses only pipelines (`|>` and `<|`)
- functions that use pipelines (`|>` and `<|`) to a function that uses only function composition (`>>` and `<<`)

5.3 Debugging with `Debug.log` and `Debug.toString`

To inspect a partial result in a function, you can use the `Debug.log` function, which takes a `String` and any value as arguments, logs the string and value to the console and returns the received value.

As an example we'll try to log the values from each stage of the pipeline in the "sum of the last digits of odd numbers" from the last lab:

Listing 5.3.1 of Debugging.elm (sumOfOddLastDigitsPipe) Elm code

```elm
sumOfOddLastDigitsPipe : List Int -> Int
sumOfOddLastDigitsPipe l =
  l
    |> Debug.log "Original list: "
    |> L.filter (\x -> modBy 2 x == 1)
    |> Debug.log "Odd elements: "
    |> L.map (modBy 10)
    |> Debug.log "Last digits: "
    |> L.foldl (+) 0
    |> Debug.log "Final sum: "
```

Elm REPL

```
> import Debugging exposing (..)
> sumOfOddLastDigitsPipe [21, 2, 13, 4, 25, 6]
Original list: : [21,2,13,4,25,6]
Odd elements: : [21,13,25]
Last digits: : [1,3,5]
Final sum: : 9
9 : Int
```

5.4 Advanced records

5.4.1 Accessors and structural typing

Concept 5.4.1: Nominal and Structural typing

Nominal: only types with the same name or in the same inheritance hierarchy are assignable. (Java, C, C++)
Structural: types with same field names (and types) are assignable (Python, JavaScript, TypeScript).

The most important aspect that you should understand about records accessors is that the are also functions that take a record which has a field with the same name as the accessor function and return the value of the field:

Elm REPL

```
> .name
<function> : { b | name : a } -> a
> .name { name = "John" }
"John" : String
> { name = "John" }.name
"John" : String
```

This means that we can use the “same” accessor function to access fields of different types from records of different types:

```
Elm REPL
> type PetType = Cat | Dog
> type alias Pet = { name: String, petType: PetType }
> type alias User = { name: String, emailAddress: String}
> let fluffy = { name = "Fluffy", petType = Cat } in .name fluffy
"Fluffy" : String
> let john = { name = "John", emailAddress = "john@email.com" } in .name john
"John" : String
```

And that you can pass accessor functions to list processing functions like `map` and `filter`:

```
Elm REPL
> List.map .x [{x=1, y=2}, {x=3,y=3}, {x=4, y=2}, {x=0, y=2}]
[1,3,4,0] : List number
> List.map .name [{name="John", age=32}, {name="Alice", age=23}, {name="Bob", age=35}]
["John","Alice","Bob"] : List String
> List.filter (.center >> .x >> (\x -> x > 0)) [{center={x=1, y=2}}, {center={x=-3, y=4}}]
[{center={x=1, y=2}}] : List {center: {x: number, y: number1}}
```

The last example shows to to concisely access data from nested records using function composition.

5.4.2 Record updates

The last topic that pertains to the syntax of Elm is record updates.

We have a circle, which has a center point, represented by two coordinates `x` and `y`, a color and a radius.

Listing 5.4.1 of `Records.elm (Color, ColoredCircle)` — Elm code

```elm
type Color = Red | Green | Blue
type alias ColoredCircle = { x: Int, y: Int, color: Color, radius: Int}
```

We would like to write a function that moves this circle with a given value on the `x` and `y` axis, *without* changing its radius and color.

With our current knowledge we can do it in 3 ways:

1. **Using the generated constructor:**

   Listing 5.4.2 of `Records.elm (moveConstructor)` — Elm code

   ```elm
   moveConstructor : ColoredCircle -> Int -> Int -> ColoredCircle
   moveConstructor circle dx dy =
     ColoredCircle (circle.x + dx) (circle.y + dy) circle.color circle.radius
   ```

2. **Creating a new record in-place:**

Listing 5.4.3 of Records.elm (moveRec) Elm code

```
moveRec : ColoredCircle -> Int -> Int -> ColoredCircle
moveRec circle dx dy =
  { x = circle.x + dx
  , y = circle.y + dy
  , color = circle.color
  , radius = circle.radius
  }
```

3. **Using destructuring:**

Listing 5.4.4 of Records.elm (moveDestructure) Elm code

```
moveDestructure : ColoredCircle -> Int -> Int -> ColoredCircle
moveDestructure circle dx dy =
  let
    { x, y, color, radius } = circle
  in
    { x = x + dx, y = y + dy, color = color, radius = radius }
```

It should be noted that in every method described above, we needed to explicitly set even the fields that we wanted to leave unchanged.

With the record update syntax, we can take a record instance and modify only a subset of its fields, leaving the rest of the fields unchanged:

Listing 5.4.5 of Records.elm (moveUpdate) Elm code

```
moveUpdate : ColoredCircle -> Int -> Int -> ColoredCircle
moveUpdate circle dx dy =
  { circle | x = circle.x + dx, y = circle.y + dy }
```

Exercise 5.4.1 ***

Given the definition of `ColoredSphere`:

Listing 5.4.6 of ColoredSphere.elm (ColoredSphere) Elm code

```
type alias Point = {x: Int, y: Int, z: Int}
type Color = Red | Green | Blue

type alias ColoredSphere = {center: Point, color: Color, radius: Int}
```

write a function `moveUpdate : ColoredShpere -> Int -> Int -> ColoredShpere` to move the sphere on the x and y axes.

Hint:
You will need to research how to do record updates for nested records.

5.5 Elegant error handling with `Maybe` and `Result`

5.5.1 Transforming success values: the `map` function

Listing 5.5.1: Definition of the Maybe.map function — Elm code

```
map : (a -> b) -> Maybe a -> Maybe b
map f ma =
    case ma of
        Just a -> Just (f a)
        Nothing -> Nothing
```

Listing 5.5.2: Definition of the Result.map function — Elm code

```
map : (a -> b) -> Result err a -> Result err b
map f res =
    case res of
        Ok ok -> Ok (f ok)
        Err err -> Err err
```

Similar to the `map` function on lists, we can also transform the element inside of the `Maybe` and `Result` types. If the instance is the `Just` or `Ok` variant, the function will be applied to the wrapped value and the `Just` or `Ok` variant will be returned with the updated value. If the instance is the `Nothing` or `Err` variant, it will remain unchanged.

Elm REPL

```
> Just 3 |> Maybe.map (\x -> x + 1)
Just 4 : Maybe number
> Nothing |> Maybe.map (\x -> x + 1)
Nothing : Maybe number
> Ok 2 |> Result.map (\x -> x + 1)
Ok 3 : Result error number
> Err "Invalid number" |> Result.map (\x -> x + 1)
Err ("Invalid number") : Result String number
```

Using the `mapN` (i.e. `map2`, `map3`, ...) functions we can also handle the case when a constructor or function needs more than one parameter and the parameters are obtained from functions that can fail:

Elm REPL

```
> type alias User = {name: String, age: Int}
> Maybe.map2 User (Just "John") (Just 30)
Just { age = 30, name = "John" }: Maybe User
> Maybe.map2 User Nothing (Just 30)
Nothing : Maybe User
> Result.map2 User (Ok "John") (Ok 30)
Ok { age = 30, name = "John" } : Result x User
> Result.map2 User (Err "No name") (Ok 30)
Err ("No name") : Result String User
```

Exercise 5.5.1 **

Write the implementation of the `map2: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c` function for `Maybe`.

5.5.2 Extracting values without pattern matching: the `withDefault` function

Listing 5.5.4: Definition of the Maybe.withDefault function — Elm code

```
withDefault : a -> Maybe a -> a
withDefault def a =
    case a of
        Just x -> x
        Nothing -> def
```

Listing 5.5.5: Definition of the Result.withDefault function — Elm code

```
withDefault : ok -> Result err ok -> ok
withDefault def res =
    case res of
        Ok ok -> ok
        Err _ -> def
```

The `withDefault` function helps us "unwrap" a `Maybe a` or `Result err ok` instance to the `a` or `ok` type, without using `case` expressions, but as with `case` expressions, both variants (`Just` and `Nothing` or `Ok` and `Err`, respectively) must be handled. We do this by providing a *fallback* value for then case when we have the `Nothing` or `Err` variant:

Elm REPL

```
> greet name = "Hello, " ++ name
<function> : String -> String
> Nothing |> Maybe.withDefault "stranger" |> greet
"Hello, stranger" : String
> type NameError = NoNameProvided | NameTooShort
> Err NoNameProvided |> Result.withDefault "stranger" |> greet
"Hello, stranger" : String
```

5.5.3 Chaining functions that can fail: the `andThen` function

Listing 5.5.6: Definition of the Maybe.andThen function — Elm code

```
andThen : (a -> Maybe b) -> Maybe a -> Maybe b
andThen f ma =
    case ma of
        Just a -> f a
        Nothing -> Nothing
```

Listing 5.5.7: Definition of the Result.andThen function — Elm code

```
andThen : (a -> Result err b) -> Result err a -> Result err b
andThen f resA =
    case resA of
        Ok a -> f a
        Err err -> Err err
```

Sometimes we wan to call a chain of functions that might fail, passing the result from the previous function to the next one in the case of success and ending the chain in case of an error. The `andThen` function is just for this case, it takes as parameter a function that returns `Maybe` or `Result`, a value of type `Maybe` or `Result` and returns a value of type `Maybe` or `Result`.

```
Elm REPL

> List.tail [1, 2, 3]
Just [2,3] : Maybe (List number)
> List.tail [1, 2, 3] |> Maybe.andThen List.tail
Just [3] : Maybe (List number)
> List.tail [1, 2, 3] |> Maybe.andThen List.tail |> Maybe.andThen List.head
Just 3 : Maybe number
> List.tail [1] |> Maybe.andThen List.tail |> Maybe.andThen List.head
Nothing : Maybe number
> List.tail [1, 2] |> Maybe.andThen List.tail |> Maybe.andThen List.head
Nothing : Maybe number
```

5.5.4 Transforming errors: the `mapError` function

Listing 5.5.8: Definition of the mapErr function — Elm code

```
mapErr : (a -> b) -> Result a ok -> Result b ok
mapErr f res =
    case res of
        Ok ok -> Ok ok
        Err e -> Err (f e)
```

In section 3.3 on page 39 we learned that error handling with `Result` is *composable*: If we have two functions that return `Result`, we can easily compose them regardless of their type parameters. We achieved this by using `case` expressions to pattern match the result of the called function and changing returned value in each case (`Ok` or `Err`) to match the signature of the caller function.

We can refactor the `safeAreaEnum` function to use `Result.mapError` instead of manually using `case` expressions to "rewrap" the error returned by `safeHeronEnum` to match the signature of `safeAreaEnum`:

Listing 5.5.9 of `Shape.elm (safeAreaEnum)` — Elm code

```
safeAreaEnum : Shape -> Result InvalidShapeError Float
safeAreaEnum shape =
  case shape of
    Circle radius ->
      if radius < 0 then
        Err InvalidCircle
      else
        Ok (pi * radius * radius)
    Rectangle width height ->
      safeRectangleAreaEnum width height |> Result.mapError InvalidRectangle
    Triangle a b c ->
      safeHeronEnum a b c |> Result.mapError InvalidTriangle
```

Using pipelines makes the logic much easier to follow:

1. First, we call the function that can fail
2. Then, if it failed, we transform the error type to match the return type of the caller function

5.5.5 Error handling patterns

Applying a series of transformations to a result and supplying a default value at the end

These functions are often combined, because we often want to apply some transformations to the results of functions that return `Maybe` or `Result`, but not to the fallback value.

Consider the example where the user can type in the name of the theme they would like to use for the page.

Listing 5.5.10 of `Theme.elm (ThemeConfig, parseTheme)` `Elm code`

```elm
type ThemeConfig = Light | Dark
parseTheme : String -> Maybe ThemeConfig
parseTheme s =
    case s of
        "dark" -> Just Dark
        "light" -> Just Light
        _ -> Nothing
```

The theme can be light or dark, and for each theme the page will be rendered with a white or black background, respectively.

Listing 5.5.11 of `Theme.elm (Color, themeToColor)` `Elm code`

```elm
type Color = White | Black
themeToColor : ThemeConfig -> Color
themeToColor th =
  case th of
    Light -> White
    Dark -> Black
```

First, we try to parse the theme from the string provided by the user. If we managed to parse a valid theme, we will choose the background color based on the theme. If the user typed in an invalid theme name or left the field blank, the page will be rendered with a white background.

Listing 5.5.12 of `Theme.elm (pageBackground)` `Elm code`

```elm
pageBackground : String -> Color
pageBackground s =
  s
    |> parseTheme
    |> Maybe.map themeToColor
    |> Maybe.withDefault White
```

`Elm REPL`

```
> import Theme exposing (..)
> pageBackground "dark"
Black : Color
> pageBackground ""
White : Color
> pageBackground "green"
White : Color
```

5.6 Package management: Using libraries from the elm package repository

Elm has a package repository found at `https://package.elm-lang.org/`, which allows you to easily use libraries developed by others in your projects.

To add a package to your project, you should use the `elm install` command, specifying the package's author and name.

For example, to install the `elm-validate` package by `rtfeldman`, you would run the command:

```
powershell session
PS > elm install rtfeldman/elm-validate
```

This modifies the `elm.json` file, by adding the package's name and version under the `direct` key of `dependencies` and add its transitive dependencies under the `indirect` key of `dependencies`.

For `elm-validate`, this will add `rtfeldman/elm-validate` under `direct` and `elm/regex` under `indirect`.

For testing, there is separate key `test-dependencies`, that has the same keys as `dependencies` (i.e. `direct` and `indirect`).

> **Note 5.6.1**
>
> **During the lab and project, you should not use any packages other than the ones that are installed by default, unless explicitly specified in the instructions.**
> The goal is learn the basics, even if it sometimes means manually implementing functions from the standard library or writing boilerplate when you could use a library.

5.7 Testing with elm-test

5.7.1 Setting up elm-test

For working with `elm-test` (the Elm package), there is an npm package with the same name, that you can install globally with the following command:

```
powershell session
PS > npm install --global elm-test
```

To add the necessary dependencies and create the tests folder, run:

```
powershell session
PS > npx elm-test init
```

This will also create a `Example.elm` file in the `tests` directory that contains some basic starting code.

5.7.2 The anatomy of a test

We can define tests by creating Elm files in the `tests` folder. Each file is defined in the same way as any other file in the `src` folder, and it can import any module from the `src` folder.

First, import the modules needed for testing, `Expect` and `Test`:

Listing 5.7.1: Imported modules — Elm code

```elm
import Expect exposing (Expectation)
import Test exposing (..)
```

To define a test, we use the `test` function, which has the signature:

Listing 5.7.2: Signature of test — Elm code

```elm
test : String -> (() -> Expectation) -> Test
```

The first parameter is the description of the test, which must be unique. The second parameter is *function* that takes the unit value and returns an `Expectation`. For this parameter we should provide a lambda that calls the function we want to test with some input, and uses the function provided by the `Expect` module (the most commonly used being `equal`) to compare the returned result with the expected result.

For our first test, we will test the `Lists.take` function:

Listing 5.7.3 of `FirstTest.elm (emptyListTakeTest)` — Elm code

```elm
emptyListTakeTest : Test
emptyListTakeTest = test "Take for an empty list returns the empty list" <|
    \_ -> Expect.equal [] (Lists.take 1 [])
```

We use the left pipe operator to `<|` avoid putting parentheses around the lambda expression. Also note that `Expect.equal` takes the expected value first and then the actual value.

Now to run the test, use:

powershell session

```powershell
PS > npx elm-test
```

`elm-test` (the npm package) will automatically find and run any function with the signature `Test` in the files in the `tests` directory.

5.7.3 Organizing tests

To group tests, we can use the `describe` function exposed by the `Test` module, which has the signature:

Listing 5.7.4: Signature of describe — Elm code

```elm
describe : String -> List Test -> Test
```

It takes a string which is used to describe a list of tests and returns a new test. This means that we can nest tests as deeply as we want!

Listing 5.7.5 of `OrganizedTests.elm (listTests)` Elm code

```elm
listTests : Test
listTests =
  describe "Lists module"
    [ describe "Lists.take"
      [ test "Take for an empty list returns the empty list" <|
        \_ -> Expect.equal [] (Lists.take 1 [])
      , test "Take 0 returns the empty list for any list" <|
        \_ -> Expect.equal [] (Lists.take 0 [ 1, 2 ])
      ]
    , describe "List.drop"
      [ test "Drop for an empty list returns the empty list" <|
        \_ -> Expect.equal [] (Lists.drop 1 [])
      , test "Drop 0 returns the original list for any list" <|
        \_ -> Expect.equal [ 1, 2 ] (Lists.drop 0 [ 1, 2 ])
      ]
    ]
```

5.7.4 Choosing which tests to run

You can use the `skip` function to skip certain tests. This is useful when working on the exercises to skip the tests for the exercises that you haven't solved yet.

5.8 Practice problems

Exercise 5.8.1 *

Reimplement the `countVowels` function from the last lab, using pipelines and function composition. This implementation should handle lowercase and uppercase vowels too. Write at least two tests to check your implementation.

Hint:

- What are the logical steps that the function could be broken up into?
- What functions can you compose to make your code shorter?

Exercise 5.8.2 *

Given the following type definitions:

Listing 5.8.1 of `Exercises.elm (AccountConfiguration)` Elm code

```elm
type alias AccountConfiguration =
  { preferredTheme: ThemeConfig
  , subscribedToNewsletter: Bool
  , twoFactorAuthOn: Bool
  }
```

Write a function `changePreferenceToDarkTheme : List AccountConfiguration -> List AccountConfiguration` which receives a list of accounts and returns a list of accounts where the `preferredTheme` field is set to the `Dark` value.

Requirements:

1. Use pipelines and record updates in your implementation.
2. Write at least one more test case in the `ExerciseTests` file for this function by replacing the call to `todo`.
3. Don't forget to remove the call to `skip`!

Exercise 5.8.3 **

Given the following type definitions:

Listing 5.8.3 of `Exercises.elm (User)` Elm code

```elm
type alias UserDetails =
  { firstName: String
  , lastName: String
  , phoneNumber: Maybe String
  }
type alias User = {id: String, email: String, details: UserDetails}
```

Write a function `usersWithPhoneNumbers : List User -> List String` which receives a list of users and returns a list containing the email addresses the of users who have provided a phone number.

Requirements:

1. Use function composition and pipelines in your implementation.
2. Write at least one more test case in the `ExerciseTests` file for this function by replacing the call to `todo`.
3. Don't forget to remove the call to `skip`!

Exercise 5.8.4 **

Write a test suite for the `chunks` function implemented in the last lab. The test suite should include:

1. Both examples (repeated here for your convenience)
2. A test for the empty list case
3. Two more tests of your choice (try to be creative)

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 5.8.5 ***

Implement a function `splitLast : List a -> Maybe (List a, a)` which takes a list as argument and when it is non-empty it returns a tuple with the list without the last element and the last element wrapped in `Just`, otherwise it returns `Nothing`.

```
                                                              Elm REPL
> splitLast [1, 2, 3]
Just ([1, 2], 3) : Maybe (List number, number)
> splitLast [1]
Just ([], 1) : Maybe (List number, number)
> splitLast []
Nothing : Maybe (List number, number)
```

Read about fuzz tests, then write a fuzz test for this function.

Hints:
Read the documentation of the `fuzz` function to get started: https://package.elm-lang.org/packages/elm-explorations/test/latest/Test#fuzz
Think about what property should always hold for this function.