Lab 2

Elm - Basic type definitions

Goals

In this lab you will learn to:

1. Define and load modules in the REPL
2. Use tuples and records
3. Define type aliases to give new names to existing types
4. Define new types, including types that have multiple variants
5. Avoid repeating yourself using `let ... in`
6. Extract data from instances using destructuring
7. Use pattern matching with `case` expression

Resources

Table 2.1: 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/`

2.1 Elm modules

So far we typed all the code in the REPL. Here you will learn how to load source files in the REPL, which will be useful when defining longer types and functions.

Again since Elm is opinionated on some aspects, we'll have to do as the designer of Elm intended. First, we'll have to create a new project with the following commands:

```
Shell session
> mkdir L2
> cd L2
L2> elm init
```

Type `Y` when prompted.

You should see 3 new items: 2 folders, `src` and `elm-stuff` and 1 one file, `elm.json`. For now we'll work in the `src` folder and we'll discuss more details about the `elm.json` file when we learn about web development with Elm.

Note 2.1.1

It is recommended to create a new folder for each Elm lab and run `elm init` in the in each folder.

Creating modules

To create a module, create a file with a `CapitalCase` name and the `.elm` extension and type on the first line: `module` ⟨*FileName*⟩ `exposing (..)`.

Note 2.1.2

Modules must contain at least one definition in order to be considered valid by Elm.
The definition can be as simple as `x = 42`.

So if you wanted to create a module named `Records`, first you would create a file named `Records.elm` in the `src` folder which would contain:

Listing 2.1.1: src/Records.elm — Elm code

```elm
module Records exposing (..)

x = 42
```

Loading modules

Once these steps are done you can run `elm repl` in the folder where you previously ran `elm init`. To load a file, place it in the `src` folder. For example if we have a file named `Records.elm` in the `src` folder we can load it by typing:

```
Elm REPL
> import Records exposing (..)
```

For now, ignore the `exposing` part (but don't forget it, or you will get errors).

Note 2.1.3

The REPL will automatically reload modules every time you call a function from the module.

2.2 Tuples and records

So far we've seen how to define recursive functions and make basic decisions using ifs. To write more useful and interesting programs we should also be able to create data structures and define operations on them.

First we'll see how to create very simple data structures using the types already built into Elm: tuples and records.

2.2.1 Tuples

Concept 2.2.1: Tuples

A tuple is a finite sequence of n values of possibility different types.

```
                                                              Elm REPL
> (1, 2)
(1,2) : ( number, number1 )
> (1, 2, 3)
(1,2,3) : ( number, number1, number2 )
```

Tuples are *heterogenous*, which means they can contain *different* types of data:

```
                                                              Elm REPL
> (1, "Hello")
(1,"Hello") : ( number, String )
> (1, "Hello", 'a')
(1,"Hello",'a') : ( number, String, Char )
> (1, ("Hello", 2), (3, 4))
(1,("Hello",2),(3,4)) : ( number, ( String, number1 ), ( number2, number3 ) )
```

Tuples help us keep related data close together, or pair up related values temporarily.

In Elm tuples are *limited by design* to contain at most 3 items:

```
                                                              Elm REPL
> (1, 2, 3, 4)
-- BAD TUPLE --------------------------------------------------------------- REPL

I only accept tuples with two or three items. This has too many:

5|   (1, 2, 3, 4)
     ^^^^^^^^^^^^
I recommend switching to records. Each item will be named, and you can use the
‘point.x‘ syntax to access them.

Note: Read <https://elm-lang.org/0.19.1/tuples> for more comprehensive advice on
working with large chunks of data in Elm.
```

2.2.2 Records

A record is a collection of named fields, similar to `struct` declaration in C or C++:

Elm REPL
```
> {firstName = "Haskell", lastName = "Curry"}
{ firstName = "Haskell", lastName = "Curry" } : { firstName : String, lastName : String }
```

How do we use records? The first way is to just let Elm infer the structure of the record:

Elm REPL
```
> fullName person = person.firstName ++ " " ++ person.lastName
<function> : { a | firstName : String, lastName : String } -> String
> fullName {firstName = "Haskell", lastName = "Curry"}
"Haskell Curry" : String
```

The second way is to specify the record in the function signature:

Elm REPL
```
> fullName : {firstName : String, lastName : String} -> String
| fullName person = person.firstName ++ " " ++ person.lastName
|
<function> : { firstName : String, lastName : String } -> String
```

Exercise 2.2.1 *

Copy and paste the following code in the REPL:

Listing 2.2.1: fullTitle — Elm code
```
fullTitle person = (if person.idDr then "Dr. " else "") ++
person.firstName ++ " " ++ person.lstName
```

Try to call the function with an argument such that `"Dr. Haskell Curry"` is displayed.

Note 2.2.1

Whole-program type inference is very useful and makes the code concise, but specifying the signature of functions will prevent a **lot** of errors like the typos in the exercise above!

2.3 Defining custom data types

In Elm we can define new data types in 2 ways: give a new name to an existing type using *type aliases* or introduce an entirely new type using *type definitions*.

2.3.1 Type aliases

Type aliases can be used to *give a new name to existing types (in addition to the existing name)*. Essentially we say that the new name can be substituted for the existing name. This works for any existing type.

The most common use case is to give name to records, because writing out the full record each time we write a type signature is slow and error-prone. For example we can name the `{firstName: String, lastName: String}` record `User` as below:

```
Elm REPL

> type alias User = {firstName: String, lastName: String}
```

Now we can rewrite the function `fullName` as:

```
Elm REPL

> fullName : User -> String
| fullName person = person.firstName ++ " " ++ person.lastName
|
<function> : User -> String
```

How do we call this function, or in other words, *how do create instances of the* `User` *type*?

The first way is to simply call it as before:

```
Elm REPL

> fullName {firstName = "Haskell", lastName = "Curry"}
"Haskell Curry" : String
```

The second way is to use the *type constructor* generated by Elm. We can check this by simply typing the name of the type in the REPL:

```
Elm REPL

> User
<function> : String -> String -> User
> User "Haskell" "Curry"
{ firstName = "Haskell", lastName = "Curry" } : User
```

Exercise 2.3.1 *

Call the `fullName` function using the `User` type constructor. Did you encounter any errors?

The other use of type aliases is to make a function signature more expressive by defining aliases for standard types. Consider the following function, which compares two temperatures, one in degrees celsius and the other one in kelvin:

```
Elm REPL

> sameTemp c k = c + 273.15 == k
<function> : Float -> Float -> Bool
> sameTemp 10 283.15
True : Bool
> sameTemp 10 283.16
False : Bool
```

If this function was defined in a library and we wanted to use it, we might try to check it's signature to try and understand what it does:

```
Elm REPL

> sameTemp
<function> : Float -> Float -> Bool
```

This is not terribly useful, because neither the documentation, nor the signature told us anything about which parameter corresponds to which temperature.

One simple solution is to define two type aliases for Celsius and Kelvin and use them to indicate the types of the parameters:

```
Elm REPL

> type alias Celsius = Float
> type alias Kelvin = Float
> sameTemp : Celsius -> Kelvin -> Bool
| sameTemp c k = c + 273.15 == k
|
<function> : Celsius -> Kelvin -> Bool
```

Now we get a much more useful type signature!

Note 2.3.1

Don't rely on type aliases for *type safety*!
The compiler still "sees" the previous signature (`sameTemp: Float -> Float -> Bool`)!
The following function will compile and work just fine:

```
Elm REPL

> addCwithK : Celsius -> Kelvin -> Float
| addCwithK c k = c + k
|
<function> : Celsius -> Kelvin -> Float
```

Use type aliases to define shorter names for records, tuples and other types.

Question 2.3.1 *

Does the way `type alias` works remind you of any keyword in C and C++?

Exercise 2.3.2 *

Define a type alias `Address`, which includes 4 fields: `street`, `number`, `city` and `country`.

Exercise 2.3.3 *

Write a function `formatAddress`, which takes an instance of an `Address` and displays it as *street number, city, country*.

```
Elm REPL

> formatAddress (Address "Baritiu street" 26 "Cluj-Napoca" "Romania")
"Baritiu street 26, Cluj-Napoca, Romania" : String
```

2.3.2 Type definitions

Type aliases are useful, but type definitions are a much more powerful mechanism that allow us to create **new types**, not just aliases.

A common use for type definitions is to create *union types* (also known as *sum types* or *enu-*

merated types), which express the possibility of a type having *multiple variants*. The simplest such definition is similar to enumerations (enum) in C and Java:

```
Elm REPL

> type Color = Red | Green | Blue
> Red
Red : Color
> Green
Green : Color
```

Here `Color` is the *type name* and `Red`, `Green` and `Blue` are *type constructors*.

We can also define types with a single variant, that contains some fields:

```
Elm REPL

> type Point = Point Int Int
> Point
<function> : Int -> Int -> Point
> Point 2 3
Point 2 3 : Point
```

Here we can see that the type and the constructor can have the same name. The `Point` to the left of the = sign is the type name and the `Point` to the right of the = sign is the constructor name.

Concept 2.3.1: Sum types and Product types

The terms "sum" and "product" come from the *cardinality* (number of elements in a set) of the types.

For sum types, the cardinality is equal to the number of variants of the given type.

→ The Color type has 3 variants, Red, Green and Blue, so its cardinality is 3.

→ The Int[a] type can represent integers from -2^{31} to $2^{31}-1$, so its cardinality is 2^{32}.

For product types, the cardinality is equal to the product of the cardinality of each field.

→ The Point type has two Int fields, so its cardinality is $2^{32} * 2^{32}$.

[a]https://package.elm-lang.org/packages/elm/core/1.0.5/Basics#Int

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 }
```

2.4 Local declarations (`let ... in` expressions)

Before we tackle how to actually use the data types we declared, we'll take a short detour to see how to avoid repeating ourselves.

Declaring constants

Remember the `howBig` function from section 1.8, repeated here for your convenience:

Listing 2.4.1 of LetIn.elm (howBig) Elm code

```elm
howBig n =
  if n < 10 then
    "Small"
  else if n < 100 then
    "Medium"
  else
    "Large"
```

We can avoid hardcoding the numbers in the if expression by using the `let ... in` expressions:

Listing 2.4.2 of LetIn.elm (howBigLetIn) Elm code

```elm
howBigLetIn n =
  let
    smallNumber = 10
    mediumNumber = 100
  in
    if n < smallNumber then
      "Small"
    else if n < mediumNumber then
      "Medium"
    else
      "Large"
```

As you can see, with `let ... in` we can declare *bindings* and use them in a *local scope*.

Heron's formula

A more convincing example is Heron's formula without using `let ... in` :

Listing 2.4.3 of Shapes.elm (heron) Elm code

```elm
heron a b c =
  sqrt
    (((a + b + c) / 2)
      * (((a + b + c) / 2) - a)
      * (((a + b + c) / 2) - b)
      * (((a + b + c) / 2) - c)
    )
```

and with using `let ... in` :

Listing 2.4.4 of Shapes.elm (heronShort) Elm code

```elm
heronShort a b c =
  let
    s = (a + b + c) / 2
  in
    sqrt (s * (s - a) * (s - b) * (s - c))
```

Avoiding shadowing in the REPL

As we've seen in Note 1.5.1 on page 11, we need to be careful with the way we define constants and functions to avoid shadowing errors in Elm. A trick you can use is to write a `let ... in` expression in the REPL, defining all the arguments that will be passed to the function locally:

Elm REPL

```
> double n = n * 2
<function> : number -> number
> let n = 10 in double n
20 : number
```

Keeping helper functions local

Remember the recommendation in Note 1.9.2 on page 15, about defining auxiliary functions to pass initial parameter values to tail recursive functions with accumulators. With our new knowledge we can hide these helper functions by defining them locally, using `let ... in` :

Elm REPL

```
> factAcc n =
|  let
|    factAccHelper i acc = if i == 0 then acc else factAccHelper (i-1) (acc * i)
|  in
|    factAccHelper n 1
|
<function> : number -> number
```

2.5 Pattern matching and destructuring

So far we've seen how to declare data types, but not how to actually use them.

2.5.1 Destructuring

Given the `Person` type, where the first `String` is the first name and second `String` is the last name of the person, we would like to write the `fullName` function, in the same way we did previously.

In Elm (and most Functional Programming languages) *type constructors* (also known as *data constructors*) *can work both ways*, meaning that besides obtaining instances of the type you can use them to *deconstruct* existing instances to extract data from them:

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
```

`Elm REPL`

```
> import Types exposing (..)
> let
|   john = Person "John" "Doe"
| in
|   fullName john
|
"John Doe" : String
```

Above we deconstruct the instance of `Person` passed to `fullName` in the parameter list, by *binding* each field to a variable name (i.e. `firstName` and `lastName`).

We can also deconstruct instances by using `let ... in` expressions:

Listing 2.5.2 of `Types.elm (fullNameLetIn)` — `Elm code`

```elm
fullNameLetIn : Person -> String
fullNameLetIn person =
  let
    (Person firstName lastName) = person
  in
    firstName ++ " " ++ lastName
```

Note 2.5.1

The main advantage of destructuring using `let ... in` expressions is that you'll also have to access to the instance as a whole (`person`). If we wanted to have the whole data in `fullName` we would have to "piece it back together" using the `Person` constructor.

This also works for records, by writing the name of each field in the record:

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
```

```
Elm REPL
> let
|   person = PersonRec "John" "Doe"
| in
|   fullNameRec person
|
"John Doe" : String
```

Note 2.5.2

The fields of new data types can be bound to any variable names, while record fields must be explicitly enumerated.

Retrieving only a subset of the fields

Sometimes we are not interested in all the fields of a type when we deconstruct it. For example we might have a function `greet : Person -> String` where we are interested in the person's first name:

Listing 2.5.4: Discarding some fields — Elm code

```elm
type Person = Person String String Int
greet : Person -> String
greet ( Person firstName _ _ ) = "Hello, " ++ firstName
```

In the example above, we wrote instead of a variable name _ for the fields that we didn't need. Note that this "*don't care*" pattern can be used as many times as needed.

We can also use only a subset of the fields for the record types too, by only writing the names of the fields that will be used:

Listing 2.5.5: Discarding some fields for records — Elm code

```elm
type alias PersonRec = {firstName: String, lastName: String, age: Int}
greetRec : Person -> String
greetRec {firstName} = "Hello, " ++ firstName
```

2.5.2 Pattern matching (case expressions)

So far we've seen how to extract the data from types that have *only one variant*.

First let's see how to handle simple enumerated types, like `Color`:

Listing 2.5.6 of Types.elm (Color, colorToHexString) — Elm code

```elm
type Color = Red | Green | Blue
colorToHexString : Color -> String
colorToHexString color =
  case color of
    Red -> "FF0000"
    Green -> "00FF00"
    Blue -> "0000FF"
```

```
Elm REPL
> import Types exposing (..)
> colorToHexString Red
"FF0000" : String
```

We can also use destructuring in `case` expressions to extract data from enumerated data types, like `Shape`:

Listing 2.5.7 of `Shapes.elm (area)` Elm code

```elm
area : Shape -> Float
area shape =
  case shape of
    Circle radius -> pi * radius * radius
    Rectangle width height -> width * height
    Triangle a b c -> heron a b c
```

Finally, we can also match literals, like numbers and strings:

Listing 2.5.8 of `Types.elm (numberToMedal)` Elm code

```elm
numberToMedal : Int -> String
numberToMedal n =
  case n of
    1 -> "Gold"
    2 -> "Silved"
    3 -> "Broze"
    _ -> "Better luck next time"
```

In the last line we have a *wildcard* (or *catch-all*) pattern. This is an *irrefutable* pattern that matches any value.

Exercise 2.5.1

Try to remove the last line (`_ -> "Better luck next time"`) and check if the code could be compiled.

Exercise 2.5.2

Try to swap the `1 -> "Gold"` and `_ -> "Better luck next time"` lines. Evaluate the following expressions in the REPL `(numberToMedal 1)`, `(numberToMedal 2)`, `(numberToMedal 10)`

Note 2.5.3: Three rules for pattern matching and destructuring

Variable names must be unique in pattern bindings!
The order of patterns in `case` expression is important!

→ The patterns are checked from top to bottom, until one pattern matches and that branch is chosen.

→ If a pattern never matches because a pattern before it will match all values that this pattern could match, the second pattern will be considered *redundant* and the code will fail to compile.

At least one pattern must always match! If the patterns in are not *exhaustive*, the code will not compile.

A useful pattern with `case` expression is to check multiple conditions and report which one failed, without using nested `if`s. The function `launchCommit` checks if the weather conditions are optimal for a rocket launch. If the wind speed and the thickness of the cloud layer are below certain values, the launch can proceed, otherwise the the reason for cancellation is printed:

Listing 2.5.9 of `Types.elm (rocketLaunch)` `Elm code`

```elm
type WeatherConditions = WeatherConditions {windSpeed: Int, cloudLayer: Int}

launchCommit : WeatherConditions -> (String, Bool)
launchCommit (WeatherConditions {windSpeed, cloudLayer}) =
  case (windSpeed < 61, cloudLayer < 1400) of
    (True, True) -> ("Launch can proceed", True)
    (False, True) -> ("Wind speeds are too high", False)
    (True, False) -> ("Cloud layer is too thick", False)
    (_, _) -> ("Suboptimal conditions", False)
```

2.6 Case study: The `Bool` and `Order` types

2.6.1 The `Bool` type

It might seem surprising, but in Elm there are no special types (except strings and number) built into the language[1], like `bool` in C++ and Java. All values can be defined with sum and product types:

Listing 2.6.1: Bool `Elm code`

```elm
type Bool = True | False
```

2.6.2 The `Order` type

Listing 2.6.2: Order `Elm code`

```elm
type Order = LT | EQ | GT
```

We can elegantly handle the case when we want to handle each result of a comparison:

[1]Since we can write `if True then ...`, `Bool` does get special treatment, but only in this case.

Listing 2.6.3: comparison Elm code

```
relation a b =
  case compare a b of
    LT -> "Less"
    EQ -> "Equal"
    GT -> "Greater"
```

2.7 Review questions

Question 2.7.1 *

What is the cardinality of the `Bool` type?

Question 2.7.2 **

How would you define `Int` as a sum type? Is the definition valid Elm syntax?

Question 2.7.3 ***

What are the built-in types that have cardinality 1 and 0, respectively? Can you define such types (i.e. will the compiler allow it)? What is the use case for such types?

2.8 Practice problems

Exercise 2.8.1 *

Define a type for a dice which has six sides.

Exercise 2.8.2 *

Define a type `DicePair`, which contains 2 `Dice`, in two ways, one using type aliases and one using type definitions.

Exercise 2.8.3 **

Write a function `luckyRoll` which takes a `DicePair` and returns a `String`. It should return “Very lucky” if the roll contains 2 sixes, “Lucky” it contains one six and “Meh” otherwise.

Exercise 2.8.4 **

Write the function `areaRec` for `ShapeRec`

```
Elm REPL
> areaRec
<function> : ShapeRec -> Float
> areaRec (CircleRec {radius = 2})
12.566370614359172 : Float
> areaRec (TriangleRec {sideA = 3, sideB = 4, sideC = 5})
6 : Float
```

Exercise 2.8.5 ***

Using the declarations for `Point` and `Shape2D` write a function `pointInShape`, which determines if a given point is inside a given shape.

Listing 2.8.5 of `PointInShape.elm (Shape2D)` Elm code

```
type alias Point = {x: Float, y: Float}
type Shape2D
  = Circle {center: Point, radius: Float}
  | Rectangle {topLeftCorner: Point, bottomRightCorner: Point}
  | Triangle {pointA: Point, pointB: Point, pointC: Point}
```

- A point (P_x, P_y) is inside a circle with center $C = (C_x, C_y)$ and radius r, if the distance between the point and the center of the circle is less than the radius of the circle (i.e. $\sqrt{(P_x - C_x)^2 + (P_y - C_y)^2} < r$).
- A point (P_x, P_y) is inside a rectangle, described by the coordinates of its top left corner (A_x, A_y) and its bottom right corner (B_x, B_y), if $A_x < P_x < B_x$ and $B_y < P_y < A_y$.
- A point $P = (P_x, P_y)$ is inside a triangle described by its 3 vertices $A = (A_x, A_y)$, $B = (B_x, B_y)$ and $C = (C_x, C_y)$ if $Area_{ABC} = Area_{PAB} + Area_{PAC} + Area_{PBC}$[a].

Hints:

1. Make sure to use local definitions (`let ... in`) for local variables and helper functions.
2. Defining a function to calculate the distance between two points will be very helpful.
3. An alternative, more reliable and faster solution for checking if a point is inside a triangle can be found here: `https://www.youtube.com/watch?v=HYAgJN3x4GA`.

[a]Instead of checking for equality, you might want to check if the difference less than some small value ϵ

Exercise 2.8.6 ***

Write a function `validateCard : Date -> CreditCard -> Bool` which checks if a credit card is valid.

1. Define the `Date` type, which stores the month and year until a card is valid.
2. Define the `CardNumber` type, which stores the 16 digits of the card as 2 `Int`s of 8 digits each. This is necessary because a 16 digit, positive integer can't be stored in a 32 bit `Int` type.
3. Define the `CreditCard` type for a credit card which has:
   - an issuer (Visa or Mastercard)
   - a card number, which is of type `CardNumber`
   - an expiration date, which is of type `Date`

4. Write a function `isDateAfter` to check if the second date is after the first date.
5. Write a function `isCardNumberValid` to check if the credit card number is valid:
    (a) To check that the whole number is valid, use the Luhn algorithm[a].
    (b) If the INN (**I**ssuer **I**dentification **N**umber) matches the card issuer:
    - Visa cards start with the digit 4
    - Mastercard cards have the first 4 digits between 2221 and 2720 or have the first 2 digits between 51 and 55

[a] `https://en.wikipedia.org/wiki/Luhn_algorithm`