PRACTICAL ELM

FOR A BUSY DEVELOPER

Alex S. Korban

Practical Elm

For a Busy Developer

Alex S. Korban

Contents

Introduction

I wrote this book because I'm excited about Elm's potential to change how we approach web development. In some ways, it's already done that by inspiring people to create JavaScript libraries like Redux and Vuex. Elm reminds me of the early days of Ruby on Rails with its focus on developer happiness, and its key features – functional programming, a static type system, immutable data, and the Elm architecture – allow me to write more expressive and more reliable code.

If you are reading this book, I'm going to assume that you are comfortable with HTML, CSS and JavaScript, and you are already somewhat familiar with Elm and excited about its potential to redefine how we develop web applications. I assume that you've already learned that Elm is a functional language with immutable data and a mechanism for dealing with side effects, and you've gone through the Elm guide and experimented with some Elm code. I also assume that you're familiar with the Elm architecture based on the data model, the view, and the update function.

This book isn't going to walk you through the Elm installation instructions, basic language syntax, and other preliminaries, because they are adequately covered by other sources. My goal is to help you get comfortable in the intermediate territory – the vast and always under-documented area where you need to write code to solve real world problems but don't yet have enough experience to figure everything out by yourself.

As Elm is growing in popularity, books, courses and a profusion of blog posts are becoming available, which is good to see. However, the vast majority of resources are still focused on the basics, and I found that working on real world projects very often leads me down a rabbit hole of mailing list threads and GitHub issue discussions.

In this text, I want to focus on the practical aspects of Elm development like tools and

dealing with bugs, and I'll spend a large portion of the book walking you through writing a small but realistic application. My goal is to go beyond the basics and to provide you with solutions – or at least pointers for further research – for problems you might encounter when doing real work with Elm.

Since this isn't a beginner book, it requires some familiarity with Elm and its core libraries, and consequently the basics of functional programming.

Conversely, I should mention that some more advanced topics are beyond the scope of this book. For example, we are not going to look at code profiling and performance optimisation, or server side rendering, or the considerations for writing Elm packages.

Resources for Newcomers to Elm

If you would like to read *Practical Elm* but all you know about Elm is that it's a kind of tree that grows in the Northern Hemisphere, don't worry: it will not take a lot of effort to get to the point where this book becomes useful to you.

To become familiar with Elm, I recommend reviewing the Elm syntax and the Elm guide, as well as spending a bit of time writing basic programs based on the guide before reading this book.

Once you go through these resources and experiment a little bit with writing Elm code, you can get back to this book to go beyond the basics.

Capabilities of Elm: Beyond Basic Syntax

Overview

Before we get into building an application, we need to build a bit of a foundation. In this chapter, we will look at some of the capabilities provided by Elm, and the functional programming techniques that it affords us.

We will talk about functions: higher-order functions, combinators, point-free style and closures. We will also look at the various ways of combining multiple functions calls in a single expression. We will consider a couple of interesting features of record types. We will take a look at operators, and at some facets of using types. Lastly, you will learn about the pattern matching and destructuring options available in Elm.

A Few Useful Bits of Functional Programming Theory

I'd like to highlight several functional programming concepts, both because they are ubiquitous and because being cognisant of them allows you to write better code. These are:

- Higher-order functions
- Combinators
- Point-free style
- Closures.

Higher-order functions and combinators

Functions which take one or more other functions as some of their arguments, or functions that return a function are called *higher-order functions*.

If a higher-order function doesn't have any free variables – that is, its definition consists only of a combination of its arguments, then it's called a *combinator*. Note that it means it cannot rely on any functions other than whatever functions are passed in as arguments.

An example of a combinator is the `compose` function:

```
compose f g x = f (g x)

compose text Debug.toString [Json, Xml, PlainText]
-- [Json,Xml,PlainText] : String
```

The reason I bring this up is that thinking about your code in terms of combinators can help you come up with better abstractions which help you construct the solution from small generic pieces.

When working with higher-order functions, you may sometimes find it useful to define type aliases for function types. For example, this signature is not very easy to decipher:

```
generatePlanTree : (Plan -> msg) -> (Plan -> msg)
    -> (String -> Decoder PlanJson -> PlanJson)
    -> Html msg
```

We can define a couple of aliases to make it more palatable:

```
type alias MouseEventHandler msg = Plan -> msg
type alias PlanParser = String -> Decoder PlanJson -> PlanJson

generatePlanTree : MouseEventHandler msg -> MouseEventHandler msg
    -> PlanParser
    -> Html msg
```

Point-free style

Related to this is the so called point-free style, which means writing a function definition in a way that omits one or more argument names. “Point” in “point-free” actually stands for “argument” for historical reasons.

It is enabled by the fact that all functions are curried (meaning that they can be partially applied). Suppose we have a function to get a list of user IDs:

```
getIds users = List.map (\u -> u.userId) users
```

As we can partially apply `map`, we can just omit the `users` argument instead, and we can also pass the field accessor function directly to `map`:

```
getIds = List.map .userId
```

This version of `getIds` is still a function that takes a list of users and returns a list of user IDs, except now there are no arguments being passed explicitly.

Similarly to combinators, point-free style can help you think in terms of reusable pieces. However, it’s best not to get carried away. While many functions can be rewritten to produce a point-free version, it’s not necessarily going to be more readable, especially if you combine lots of function calls, so please keep readability in mind as well.

Closures

Once higher-order functions are in play, it becomes possible to create closures. A closure is a function together with a mapping of all its free variables (in Elm, it means the variables which are not arguments) to their values at the time of creating this function.

It can seem a bit counterintuitive in a pure language where all variables are actually constants, but it's still possible to create closures in Elm:

```
isSelectedUser : Int -> User -> Bool
isSelectedUser selectedUserId user = .userId user == selectedUserId

displayUsers currUser users =
    let
        isCurrentUser : User -> Bool
        isCurrentUser = isSelectedUser currUser.userId
    in
        ...
```

`isCurrentUser` in the `let` block is a function that takes a user and returns `True` if the user is the current user. The ID of the current user isn't passed as an argument: its value is enclosed from the function's environment – in this case, the partially applied `isSelectedUser` function.

Another way to create a closure would be to return a function:

```
stringRepeater n =
   \s -> String.repeat n s
```

The function returned by `stringRepeater` is a closure which carries the value of `n` from its surrounding context.

Using Functions

Now I'd like to show you a few tricks to do with functions.

Pipe operators

One of the potential readability issues with FP code is that because it consists entirely of function calls, you can end up with a huge number of parentheses:

```
text (toString (List.filter canAddUsers (getActiveUsers projectId users)))
```

Elm provides two pipe operators, more properly called function application operators, to help you avoid too much resemblance to Lisp in your code:

```
(|>) : a -> (a -> b) -> b
(<|) : (a -> b) -> a -> b
```

Both of the pipe operators are equivalent to plain function application:

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

However, because they have different precedence and associativity, they allow us to write expressions without parentheses for the most part. The backward function application operator passes an argument to a function on its left hand side:

```
text
    <| toString
    <| List.filter canAddUsers
    <| getActiveUsers projectId users
```

Here, we're producing a text element with the string returned by `toString`, which is in turn given the filtered list of active users.

The forward function application operator passes an argument to the function on its right hand side:

```
users
    |> getActiveUsers projectId
    |> List.filter canAddUsers
    |> toString
    |> text
```

We can read it like this: we take the active users, then filter them, then convert the resulting list to a string, and finally pass it to the `text` function.

Depending on the particular expression and perhaps what you are used to, either backward or forward operator will produce more readable code. These operators are used practically everywhere in Elm so you will see a lot of them in this book.

Function composition operators

Similarly to function application, there are two function composition operators:

```
(<<) : (b -> c) -> (a -> b) -> a -> c
(>>) : (a -> b) -> (b -> c) -> a -> c
```

Again, the idea is to choose one that results in more readable code. For example, we can filter a list of users to find those that aren't allowed to create new users in two ways:

```
List.filter (not << canAddUsers) [Engineer, Driver, Foreman, OfficeManager]

List.filter (canAddUsers >> not) [Engineer, Driver, Foreman, OfficeManager]
```

The first line with `not` on the left reads more naturally in this case.

Argument order

If you're coming to Elm from JavaScript or another object oriented language, it's worth pointing out that programming with curried functions requires a different order of arguments from what you may be used to.

The main or the most "varying" argument, so to speak, should be placed *last* in the function definition. To give an example, `List.map` takes a function first and the list it's applied to second. `Maybe.withDefault` takes the default value first, and the actual `Maybe` value second.

In imperative and object-oriented programming, you would normally do the opposite. For example, in Underscore.js or Lodash (both JavaScript libraries), the `map` function takes the array as its first argument and the function to apply follows:

```
_.map([1, 2, 3], (num) => num * 3)
```

However, that is rather inflexible. Placing the main argument last facilitates building reusable blocks out of partially applied functions, and it also makes them easier to write in a point-free style. For example:

```
isOverThreshold threshold n = n > threshold
getOutliers = List.filter <| isOverThreshold 100

getOutliers [1, 2, 200, 3, 400, 4]
```

The argument order in both `isOverThreshold` and `List.filter` makes it easier for me to write `getOutliers` in a point-free style by partially applying both of those functions. If, say, `List.filter` took the list as its first argument, it wouldn't work out quite so conveniently.

Flipping arguments

At times, we really do need to apply arguments in a different order. Consider the issue of checking a user's set of permissions to verify that they are allowed to access a particular page or piece of functionality. For instance, we can have multiple user types, and we might want to list the subset of users who are allowed to add new users:

```
type User
    = Driver
    | Engineer
    | Foreman
    | OfficeManager
    | ProjectManager
    | Surveyor

-- ...

List.filter canAddUsers [Engineer, Driver, Foreman, OfficeManager]
```

In `canAddUsers`, we need to check the user type against the list of types which are allowed to add users:

```
canAddUsers u = List.member u [Foreman, OfficeManager, ProjectManager]
```

We can rewrite `canAddUsers` in point-free style using the `flip` combinator:

```
canAddUsers = flip List.member [Foreman, OfficeManager, ProjectManager]
```

`flip` converts the function into a version with swapped arguments:

```
flip : (a -> b -> c) -> b -> a -> c
flip f b a =
    f a b
```

This combinator was provided by the Elm core library prior to Elm 0.19, but now you have to implement it yourself, or get it from some package.

Using Records

There are a couple of interesting things about record types beyond the basic syntax.

Constructors

The name of a record type is referenced in two contexts. One is the name of a *type alias*, and the other one is a *constructor function* used to create values of this type:

```
type alias User = { userId: Int, name: String }

User 1 "John"  -- User is a constructor function
```

For each record type alias, the compiler automatically generates a constructor function of the same name. For our `User` alias, it's a function that takes an integer and a string, and returns a record:

```
User : Int -> String -> User
```

The constructor is a regular function so it can be partially applied:

```
User 1
-- <function> : String -> User
```

Using extensible records to restrict arguments

Elm has the concept of extensible records:

```
type alias GenericNode a =
    { a | actualLoops : Int
    , actualRows : Int
    , actualStartupTime : Float
    , actualTotalTime : Float
    ...
    }

type alias CteNode
    = GenericNode
    { nameAlias : String
    }
```

In this example, a `CteNode` is a record with all the fields of `GenericNode` plus an additional alias field.

At first glance, this might look sort of like inheritance where you construct more specific types by extending generic ancestors. However, it's not really meant to be used in this fashion, as hinted by the fact that type aliases for extensible records don't get a constructor generated for them. The preferred solution in Elm is to use a nested record field for shared fields.

The main purpose of this syntax is actually to restrict arguments taken by functions. The reason this is useful is that it allows you to write simple functions focused on a specific task, which makes it easier to comprehend code and to debug it.

For example, I could have a function which appends an alias name to the node title. Using extensible records, I can make it clear that this function works specifically with a node title and an alias:

```
type alias NodeTitle = String
type alias Aliased n = { n | nameAlias : String }

appendAlias : NodeTitle -> Aliased n -> NodeTitle
appendAlias title {nameAlias} =
    title ++ " (" ++ nameAlias ++ ")"
```

This function will only work with records which have the `nameAlias` field, but it's also clear from the type annotation that it will not touch anything *other* than the alias when it's passed a record containing other fields in addition to the alias field. This results in improved readability of the code.

Operators as Functions

Operators are in fact functions in disguise, and the infix notation is just syntactic sugar. Any operator can be used as a function if it's wrapped in parentheses:

```
(*) 10 3
-- Produces 30

(::) "a" ["b", "c"]
-- Produces ["a", "b", "c"]
```

Since they are functions, we can partially apply them, which is handy for using them with higher-order functions:

```
List.map ((*) 2) [1, 2, 3]
-- Produces [2, 4, 6]
```

Using Types

With regards to types, there are several things worth understanding beyond the basics:

- Recursive types
- Special type variables
- Phantom types
- Opaque types.

I will show the first three in this section, but defer the discussion of opaque types till later in the book in order to be able to introduce them in the context of the sample application.

Recursive types

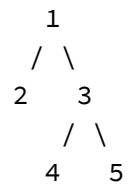
It's possible to define recursive types. For example, we could define a generic `Tree` type:

```
type Tree a
    = Empty
    | Node a (List (Tree a))
```

Each node has a value and a list of child nodes. The type of child nodes is recursively defined as `Tree a`. We could create a value of this type like this:

```
tree =
    Node 1
        [ Node 2 []
        , Node 3
            [ Node 4 []
            , Node 5 []
            ]
        ]
```

This value represents the following binary tree:



Special type variables

Normally when we define a generic type, we use some lowercase name like `a` or `msg` as a type placeholder:

```
map : (a -> msg) -> Cmd a -> Cmd msg
```

Elm has three *special* placeholder names which denote constrained sets of types and can be used instead of a regular placeholder:

- `number`
- `appendable`
- `comparable`

For example, we could constrain the tree type we defined before to only work with appendable types:

```
type Tree appendable
    = Empty
    | Node appendable (List (Tree appendable))
```

If I try to use this type with `Int` which is not appendable, the compiler will complain:

```
tree : Tree Int
tree = Node 1 []
-- Compile error
```

The only appendable types are strings and lists. The only number types are `Int` and `Float`. The set of comparable types consists of `Int`, `Float`, `Char`, `String`, lists, and tuples.

It's possible to use the same kind of placeholder more than once in a type signature. Consider `Set.map` from the Elm core library:

```
-- Set
map : (comparable -> comparable2) -> Set comparable -> Set comparable2
map func set =
    fromList (foldl (\x xs -> func x :: xs) [] set)
```

This function produces a new set from the input set by applying `func` to elements of `set`. Since set elements have to be comparable, `func` takes a value of a `comparable` type as its argument, and it also has to return a value of a `comparable` type. However, the input and output types are not necessarily the same, so it doesn't make sense to write `(comparable -> comparable)`. In order to allow these sorts of functions, `number`, `appendable` and `comparable` can be suffixed with an alphanumeric sequence without changing their meaning.

Phantom types

A phantom type is a generic type with one or more type variables which are not used in *any* of the constructors. For example:

```
type Unit tag value = Unit value
```

While `value` is used in the constructor, `tag` isn't, which makes `Unit` a phantom type. The interesting thing is that there is a way to use `tag` in order to get the compiler to detect incompatibility between values we want to distinguish.

Let's define a couple of types for distance units which we will use as tags:

```
type KmTag = KmTag
type MileTag = MileTag
```

So how do we use the `Unit` constructor? For that, we need to define corresponding functions:

```
km : number -> Unit KmTag number
km = Unit

mile : number -> Unit MileTag number
mile = Unit
```

Note that both of them are simply aliasing the `Unit` constructor, but the key is their return types which are using different tags (`KmTag` and `MileTag`). Now, if we pass a value created with `mile` to a function that expects a value tagged with `KmTag`, we will get a compilation error:

```
showKm : Unit KmTag Float -> String
showKm (Unit d) = String.fromFloat d

distance = mile 12.34

s = showKm distance  -- Compilation error
```

```
Type Mismatch

The 1st argument to `showKm` is not what I expect:

53| s = showKm distance  -- Compilation error
               ^^^^^^^^
This `distance` value is a:

    Unit MileTag Float

But `showKm` needs the 1st argument to be:

    Unit KmTag Float
```

So the compiler is able to distinguish between these values, even though at runtime values created with either `km` or `mile` are the same boxed integers or floats.

At first blush, this might seem like a really cool and useful concept, however in practice the situations where you might need phantom types are few and far between. In most cases, all you need is distinct types and perhaps also opaque types if you're trying to constrain how values are created.

For example, you might have account IDs and user IDs which are both integers but can't be used interchangeably. In this situation, you don't need phantom types, it's enough to define two distinct types:

```
type AccountId = AccountId Int
type UserId = UserId Int
```

However, if you want to distinguish between values which have the same runtime representation, *and* you have a shared set of operations you want to perform on all the different types of these values, then it's a use case for phantom types. Consider what we'd have to do to perform calculations on our `km` values, such as addition:

```
type alias Km = Unit KmTag Float

addKm : Km -> Km -> Km
addKm (Unit d1) (Unit d2) = km (d1 + d2)
```

For every operation we'd like to do (addition, subtraction etc.), we need to write a function to unwrap the arguments, perform the operation on raw numbers and wrap the result back up. Then we would have to write all the same functions for distances expressed in miles. Of course, this would be tedious and error-prone, so in this case there's a benefit in using phantom types. Even so, you may not need to roll your own implementation, and instead use the elm-tagged package which implements tagged types along with a number of convenience functions.

Another use case for phantom types is when you want to use a particular runtime representation for reasons of performance or memory use, but still want compile-time enforcement of the semantics of different types. This approach is in development in the `elm-css` library, whereby various values are represented as boxed strings at runtime (using the phantom type `type Value a = Value String`) but are tagged with different record types at compile time in order to make sure that the values are only generated using the appropriate functions. As a result, you will get a compiler error if you try to pass a size value (generated with `px 100` or similar) to a colour style, for instance.

Wildcard Match, Destructuring, and Pattern Matching

Elm provides a number of language level facilities for working with data values.

These are:

- Wildcard match
- Destructuring
- Pattern matching

Wildcard match

The underscore (_) means a wildcard match in Elm, and it can appear both in function definitions and in `case` expressions.

For instance, the function called `always` from the Elm core library can be defined like this:

```
always : a -> b -> a
always x _ = x
```

Using the underscore instead of a named argument signifies that the argument is ignored. `always` can then be used with `map` like this:

```
List.map (always 0) [1, 2, 3]

-- [0, 0, 0] : List Int
```

The underscore is also used in the default branch of `case` expressions:

```
describeNum n =
    case n of
        0 ->
            "none"
        1 ->
            "one"
        _ ->
            "many"
```

The difference between a regular variable and underscore is that underscore doesn't create a binding between the value and the name, which means that you can have multiple underscores in a single expression:

```
type Node = Table SchemaName TableName | Cte CteName

nodeType : Node -> String
nodeType node =
    case node of
        Table _ _ ->
            "Table"
        Cte _ ->
            "CTE"
```

Destructuring

The distinction between destructuring and pattern matching is somewhat blurred in Elm, because these things frequently go together. When I talk about destructuring, I mean binding parts of a composite value to separate identifiers:

```
let
    (firstName, lastName) = ("John", "Doe")
in
lastName
-- Doe : String
```

Destructuring works in function definitions:

```
second (_, snd) = snd

second (0, 1)
-- 1 : number
```

Records can be destructured too, but in this case you have to use matching field names rather than arbitrary variable identifiers. For example, this `length` function will work on any records which have fields called `x` and `y` of suitable types:

```
length { x, y } = sqrt <| toFloat <| x^2 + y^2
```

It can be applied to 2D vectors which just have `x` and `y` as well as 3D vectors with `x`, `y` and `z` fields. However, if you provide a type annotation for this function with a specific type alias, then the function will only work with that particular record type:

```
type alias Vector = { x : Int, y : Int }

length : Vector -> Float
length { x, y } = sqrt <| toFloat <| x^2 + y^2

length { x = 1, y = 2, z = 3 }  -- compile error
```

If you want to have a type annotation that lets it work with different record types, you need to write it using the extensible record syntax:

```
length : { r | x : Int, y : Int } -> Float
length { x, y } = sqrt <| toFloat <| x^2 + y^2

length { x = 1, y = 2, z = 3 }  -- OK
```

Alternatively, you can use a parametrised alias to achieve the same result:

```
type alias Vector r = { r | x : Int, y : Int }

length : Vector r -> Float
length { x, y } = sqrt <| toFloat <| x^2 + y^2
```

Additionally, it's possible to name the whole record while destructuring some of the fields:

```
displayNode ({ nodeType, relationName } as node) =
    case ( nodeType, relationName ) of
        ( "Seq Scan", "projects" ) ->
            highlightNode Color.blue node
        ( _, "projects" ) ->
            highlightNode Color.green node
        _ ->
            displayPlainNode node
```

In the above example, we can refer both to `nodeType` and `relationName` fields as well as the whole record (as `node`), which is useful because we need to pass the whole record to other functions. Note that the parentheses around the argument are required, otherwise the compiler will complain. I'll talk about how this `case` statement works in a minute, when we get to pattern matching.

Beyond tuples and records, you can even do nested destructuring for tuples within tuples or records within tuples:

```
isWithinBounds ( maxX, maxY, { x, y } ) =
    x < maxX && y < maxY

isWithinBounds ( 10, 10, { x = 9, y = 9 } )
```

One more use of destructuring is to extract the carried value from a type:

```
type User = User String

userId : User -> String
userId (User u) = u

userId (User "A")
-- A : String
```

This also works in a `let` expression:

```
userName : User -> String
userName user =
    let
        (User name) = user
    in
        String.toLower name
```

The last thing I'd like to point out here is that anonymous functions work just like regular functions with regards to destructuring:

```
( "John", "Doe" )
    |> \( a, b ) -> a ++ " " ++ b
-- John Doe : String

{ first = "John", last = "Doe" }
    |> \{ first, last } -> first ++ " " ++ last
-- John Doe : String

(\(User u) -> u) (User "A")
-- A : String
```

Pattern matching

Pattern matching is a mechanism for choosing the branch of code to execute based on the type or value of a given expression. Elm provides only one construct for pattern matching: the `case` expression. It is nonetheless a fairly versatile tool because it can be used to match custom types (also known as union types), values and lists.

Let's look at a few examples to see how this works. The most frequent situation is where you need to take different actions based on different values of a custom type, like handling different messages in the `update` function:

```
case msg of
    ChangePlanText s ->
        ( { model | currPlanText = s }, Cmd.none )

    SubmitPlan ->
        ( { model | currPage = DisplayPage }, Cmd.none )

    _ ->
        ( model, Cmd.none )
```

While matching a particular value, you can also name the data it contains. In this example, `ChangePlanText` carries a string which we name `s` and then use it to update the model. The last branch is a catch-all: it will match any value not matched by the preceding branches.

We are also able to match values of built-in types:

```
case userCount of
    0 ->
        "nobody"
    1 ->
        "1 person"
    _ ->
        "2 or more people"
```

We can even combine custom type and contained value checks:

```
case userCount of
    Just 0 ->
        "No users at the moment"
    Just 1 ->
        "Only 1 person"
    Just _ ->
        "2 or more people"
    Nothing ->
        "No users ever"
```

By employing tuples, we can also handle more complex logic. Here is the case statement for highlighting nodes based on their properties which I showed before:

```
displayNode ({nodeType, relationName} as node) =
    case (nodeType, relationName) of
        ("Seq Scan", "projects") ->
            highlightNode Color.blue node
        (_, "projects") ->
            highlightNode Color.green node
        _ ->
            displayPlainNode node
```

By constructing a tuple out of the destructured record fields, we can choose to do different things based on the combination of the two field values. If the node type is “Seq Scan” and the relation name is “projects”, then we highlight in blue. If it’s any other node type for the “projects” relation, then we highlight in green, while any other combination of values doesn’t get highlighted. This is a compact way to express conditional logic.

Finally, we can use the `case` expression to pattern match lists:

```
describeNode node =
    case node.children of
        [] ->
            "Leaf node"
        [ _ ] ->
            "1 child node"
        [ _, _ ] ->
            "2 child nodes"
        _ ->
            "Multiple child nodes"
```

In this example, the pattern is the number of elements in the list: zero, one or two.

Alternatively, lists can be pattern matched using the *cons* operator:

```
describeNode node =
    case node.children of
        [] ->
            "Leaf node"
        [ a ] ->
            "1 child node: " ++ a
        a::b::_ ->
            "2 or more child nodes"
```

The last branch matches lists with two or more elements.

Summary

In this chapter, we looked into the capabilities of the Elm language and a few functional programming concepts.

We talked about some of the functional programming concepts and techniques made available by Elm: higher-order functions, combinators, point-free style and closures. You learned some of the specifics of working with functions and records, as well as operators and types. You got introduced to three of the key functions employed in functional

programming: `map`, `filter` and `fold`. Finally, we talked about the tools available for working with data at the language level and handling conditional logic: wildcard matches, destructuring, and pattern matching.