Don’t Worry About Monads

Just build it already

Beau Lyddon

Managing Partner at Real Kinetic

We mentor engineering teams to unleash your full potential.

@lyddonb
linkedin.com/in/beau-lyddon
github.com/lyddonb

We’re going to focus on Strongly-Typed Functional Languages in the ML family

Haskell, Purescript, Elm, OCaml, Idris, F#

Although many of the concepts apply to other functional languages: Scala, lisp family, Rust, etc

So why am I doing this?

Because the world (especially our industry) will be better if we get more to embrace functional programming principles

And for that we need to start shipping code based around functional programming

This is not a direct "teach you the language" presentation

It’s more of a “get comfortable with syntax, structure and terms” presentation

It's also a kill the FUD (Fear, Uncertainty, Doubt) Presentation

I want to remove the barriers that may be keeping you from diving in

This comes from my own experience attempting to learn these languages

I have/had imposter syndrome from a lacking mathematics background

And I struggle to learn programming from books

Much of the code and documentation seemed quite advanced

REALKINETIC
YOUR POTENTIAL REALIZED

Difficult to separate struggles from lack of familiarity of syntax and complex/ advanced constructs

But …

One thing I want to make clear ...

This is not going to be an anti-terminology presentation

I was that person,
I was wrong

The terminology can be quite helpful as you learn it

Instead I hope to show

…

You don’t need to understand all of the terminology

(While still appreciating it’s value)

You don’t need to be great at math

Or even go through books

To be productive in functional languages today

And still get many of the advantages these languages provide

This is why we will mostly focus on Elm

Elm is a compile to Javascript mostly, front-end (in the browser) focused language

(Yes, it can be used in other environments)

Elm is meant to be the more accessible strongly-typed (ML family) language

It is designed for you to get work done and shipped to production

If you leave this presentation excited you can go spend a couple of hours doing the getting started with Elm

And build something real by the time your done

Elm sacrifices some power to allow more access

They purposely avoid non-common mathematical terms, etc

But hopefully once you've built some stuff in Elm you'll be excited to learn about this magic that's available in these other languages

And then you can dig as deep into terminology, formalism, etc as you would prefer

And you'll get even more benefits appearing in your programs

Fair warning:
Once you start, you may not be able to stop :)

Now, let’s get started

The first hurdle is the syntax

We're going to go very fast through the syntax

I want to get to the fun stuff at the end

This is all taken from the Elm documentation and starter material (tutorials, etc)

Let’s make a quick counter

In Python

```python
from jinja2 import Template

def main():
    return SomeProcessor(model, view, update)

class Msg:
    Increment = "INCREMENT"
    Decrement = "DECREMENT"

class Model(object):

    def __init__(self, count):
        self.count = count


def model():
    return Model(0)

def update(msg, model):
    if msg == Msg.Increment:
        model.count += 1
    elif msg == Msg.Decrement:
        model.count -= 1

    return model

TEMPLATE = """
<div>
  <button onClick="decrement()">-</button>
  <div>{{ count }}
  <button onClick="increment()">+</button>
</div>
"""

def view(model):
    template = Template(TEMPLATE)
    return template.render(count=model.count)
```

Now, Elm

http://elm-lang.org/examples/buttons

```elm
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Html.beginnerProgram { model = model, view = view, update = update }

model = 0

type Msg = Increment | Decrement

update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

```python
from jinja2 import Template

def main():
    return SomeProcessor(model, view, update)

class Msg:
    Increment = "INCREMENT"
    Decrement = "DECREMENT"

class Model(object):

    def __init__(self, count):
        self.count = count


def model():
    return Model(0)

def update(msg, model):
    if msg == Msg.Increment:
        model.count += 1
    elif msg == Msg.Decrement:
        model.count -= 1

    return model

TEMPLATE = """
<div>
  <button onClick="decrement()">-</button>
  <div>{{ count }}
  <button onClick="increment()">+</button>
</div>
"""

def view(model):
    template = Template(TEMPLATE)
    return template.render(count=model.count)
```

Where are the types?

They are not required as they are inferred.

But it is recommend to annotate your code

Let's start with our Python example

```python
class Model(object):

    def __init__(self, count):
        """Initialize a counter.

        args: count (Int)
        """
        self.count = count

def model():
    """Initializes a new model.

    returns: Model
    """
    return Model(0)

def update(msg, model):
    """Updates a model based off the message and existing model state.

    args: msg (Msg)
          model (Model)

    returns: Model
    """
    if msg == Msg.Increment:
        model.count += 1
    elif msg == Msg.Decrement:
        model.count -= 1

    return model

def view(model):
    """Renders our view with our template and the passed in model state.

    args: model (Model)

    returns: String
    """
    template = Template(TEMPLATE)
    return template.render(count=model.count)
```

And now Elm

```elm
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

Program Never Model Msg
main =
  Html.beginnerProgram { model = model, view = view, update = update }

type alias Model = Int

model : Model
model = 0

type Msg = Increment | Decrement

update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

```elm
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Html.beginnerProgram { model = model, view = view, update = update }

model = 0

type Msg = Increment | Decrement

update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

Let’s dive in and step through so we can get more comfortable with the syntax

With type definitions the last type is the return type

```elm
-- A single type means it takes 0 arguments
thing : Int
thing = 0


-- This takes 2 arguments each of int and
-- returns and int
add : Int -> Int -> Int
add x y = x + y
```

```elm
-- You can alias a type
type alias Things = Int


-- This allows us to make more readable types.
thing : Things
thing = 0
```

```
-- This is a sum or union type
     (specifically a boolean type in this case).
-- You may also see: tagged or disjoint
-- unions or variant types
type Msg = Increment | Decrement
```

```
-- Sum Types can have as many options as you'd like
type Cars
    = Mustang
    | Camero
    | Taurus
    | Fit
    | Focus


-- With Boolean there are 2 possible options
type Bool
    = False
    | True
```

Why called Sum Type?

You can find the total number of possible values by adding them

Bool has 2 possible options:

(False + True)

Cars has 5 possible options:

(Mustang + Camero + Taurus + Fit + Focus)

Sum Types are a bit like Enums in other languages

Enums in Python

```python
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3



>>> print(Color.RED)
Color.RED

>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
True
>>> print(Color.RED.name)
RED
```

Enums in Javascript

Kinda

```javascript
// Enum
var DaysEnum = Object.freeze({"monday":1,
                              "tuesday":2,
                              "wednesday":3
                            });

DaysEnum.monday = 33;
// Throws an error in strict mode

console.log(DaysEnum.tuesday);
// expected output: 2
```

But not quite. It usually takes quite a bit of code to make a proper Sum type in other languages

Full Sum Type in Javascript

https://medium.com/fullstack-academy/better-js-cases-with-sum-types-92876e48fd9f

```
const PointTag = Symbol('Point')
const Point = (x, y) => {
   if (typeof x !== 'number') throw TypeError('x must be a Number')
   if (typeof y !== 'number') throw TypeError('y must be a Number')
   return { x, y, tag: PointTag }
}

const CircleTag = Symbol('Circle')
const RectangleTag = Symbol('Rectangle')
const Shape = {
   Circle: (center, radius) => {
      if (center.tag !== PointTag) throw TypeError('center must be a Point')
      if (typeof radius !== 'number') throw TypeError('radius must be a Number')
      return { center, radius, tag: CircleTag }
   },
   Rectangle: (corner1, corner2) => {
      if (corner1.tag !== PointTag) throw TypeError('corner1 must be a Point')
      if (corner2.tag !== PointTag) throw TypeError('corner2 must be a Point')
      return { corner1, corner2, tag: RectangleTag }
   }
}
```

```
const circ1 = Shape.Circle(Point(2, 3), 6.5)
const circ2 = Shape.Circle(Point(5, 1), 3)

const rect1 = Shape.Rectangle(Point(1.5, 9), Point(7, 7))
const rect2 = Shape.Rectangle(Point(0,   3), Point(3, 0))

console.log('Is circ1 a circle?', circ1.tag === CircleTag) // true
console.log('Is circ2 a circle?', circ2.tag === CircleTag) // true
console.log('Is rect1 a rectangle?', rect1.tag === RectangleTag) // true
console.log('Is rect2 a rectangle?', rect2.tag === RectangleTag) // true

const rect3 = Shape.Rectangle(Point(1, 2), 9) // ERROR: corner2 must be a Point
```

Pattern Matching

```
type MyBool
    = MyFalse
    | MyTrue

handleBool : MyBool -> String
handleBool myBool  =
  case myBool of
    MyTrue ->
      "It's my true"

    MyFalse ->
      "It's my false”

handleBool MyTrue
> "It's My True"

handleBool MyFalse
> "It's My False"
```

```
class MyBoo(Enum):
    MyTrue = 1
    MyFalse = 0

def handle_bool(my_bool):
    if my_bool == MyBoo.MyTrue:
        return "It's my true"
    elif my_bool == MyBoo.MyFalse:
        return "It's my false”

handle_bool(MyBoo.MyTrue)
> "It's my true"

handle_bool(MyBoo.MyFalse)
> "It's my false"
```

But what about …

```
handle_bool("THIS CAN BE ANYTHING")
> None
```

```
def handle_bool_all(my_bool):
    if my_bool == MyBoo.MyTrue:
        return "It's my true"
    elif my_bool == MyBoo.MyFalse:
        return "It's my false”
    else:
        return “Oops”


handle_bool_all(“THIS CAN BE ANYTHING!")
> "Oops"
```

```
var MyBool = Object.freeze({"myTrue": 1, "myFalse": 0});

const handleMyBool = (myBool) => {
    switch (myBool) {
        case MyBool.myTrue: return "It's my true"
        case MyBool.myFalse: return "It's my false"
        default: "Oops!"
    }
}
```

Btw, there’s nothing stopping us from doing …

```python
def handle_bool_missing(my_bool):
    if my_bool == MyBoo.MyTrue:
        return "It's my true”
```

```
handleBool : MyBool -> String
handleBool myBool  =
  case myBool of
    MyTrue ->
      "It's my true"
```

Missing Patterns

This **`case`** does not have branches for all possibilities.

You need to account for the following values:

    Main.MyFalse

Add a branch to cover this pattern!

If you are seeing this error for the first time, check out these hints:
<https://github.com/elm-lang/elm-compiler/blob/0.18.0/hints/missing-patterns.md>
The recommendations about wildcard patterns and `Debug.crash` are important!

```
handleBool : MyBool -> String
handleBool myBool  =
  case myBool of
    MyTrue ->
      "It's my true"

    MyFalse ->
      "It's my false”

    Default ->
      "The default"
```

Naming Error

Cannot find pattern **`Default`**

```
handleBool : MyBool -> String
handleBool myBool  =
  case myBool of
    MyTrue ->
      "It's my true"

    _ ->
      "The default”

-- WORKS!
```

```
handleBool : MyBool -> String
handleBool myBool  =
  case myBool of
    MyTrue ->
      "It's my true"

    MyFalse ->
      "It's my false"

    _ ->
      "The default"
```

Redundant Pattern

The following pattern is redundant. Remove it.

Any value with this shape will be handled by a previous pattern.

Functions, Arrows, Partial Application, and Currying

Sounds harder than it is

Javascript, Python, Ruby support partial application and currying

```
-- This takes 2 integers and returns and Int
add2Things : Int -> Int -> Int
add2Things x y = x + y
```

```
-- But it doesn't have to.
-- You can pass only one argument
add2Things 1
```

This is called: partial application

(You are partially applying arguments to a function)

```
-- This means when you don't pass all of the
-- arguments in you will get back a function.

-- So:
add2Things 1

-- Returns a function
(Int -> Int)
```

```
-- You can read that like:
add2ThingsPartial : Int -> (Int -> Int)

-- This is now a function that takes a single:
Int

-- And returns a function of:
(Int -> Int)
```

```
-- Let's set that to a variable
let addTo1 = add2Things 1
```

```
-- Now addTo1 is a function that takes a single
argument. It looks like
addTo1 : Int -> Int
addTo1 x = add2Things 1

-- or
addTo1 x = 1 + x

-- So if we call it
addTo1 2 == 3
```

Python

```
def add2Things(x):
    return lambda y: x + y

addTo1 = add2Things(1)

>>> addTo1(2)
3
```

Comes with built-in for partial application

```python
from functools import partial


def add2Things(x, y):
    return x + y

addTo1 = partial(add2Things, 1)

>>> addTo1(2)
3
```

Javascript

```
// As mentioned this can be done in Javascript
var add2Things = function (x, y){
    return function (y){
      x + y;
    }
}

> add2Things(1);

function (y){
  x + y;
}

> var addTo1 = add2Things(1);
> addTo1(2);
> 3
```

You now have seen partial application, currying and what are called:

Higher Order Functions

A function that does at least one of the following:

Takes one or more functions as arguments

Or returns a function as its result

That's all higher order functions are

Functions that take and/or return functions themselves

We will come back to that later and use it often.

It’s important in functional languages to get comfortable with treating functions like data that can be passed around.

Let’s go through more to keep getting familiar with syntax

```elm
type Msg = Increment | Decrement


-- This takes 2 arguments,
-- a Msg and a Model and then returns a Model
update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1
```

```
const helloWorldReducer = (state=0, action) => {
  switch(action.type){

    case PLUS:
      return Object.assign({}, state, state + 1)

    case MINUS:
      return Object.assign({}, state, state - 1)

    default:
      return state
  }
}
```

```elm
type Msg = Increment | Decrement


-- This takes a single type of Model
-- and returns a single type of Html Msg
view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

```
-- Html is another type alias for:
type alias Html msg = VirtualDom.Node msg
-- This type as a "generic" type.


-- The msg could be anything:
type alias Html a = VirtualDom.Node a



// Java Generics:
public interface Html<A> {}
Html<Msg> myHtml = new MyHtml(myMSG)
```

```
import React from 'react'

const Hello = ( {onClickPlus, onClickMinus, message} ) => {
  return(
    <div>
      <h2>{message}</h2>
        <button onClick={onClickPlus}>+</button>
        <button onClick={onClickMinus}>-</button>
    </div>
  )
}

export default Hello
```

What if we want to store multiple values

```elm
-- Records!
type alias Counter =
    { value : Int
    , count : Int
    }
```

Records are often called Product Types or Tuples

Why?

It's a compound type that is formed by a sequence of types and is commonly denoted:

(T1, T2, ..., Tn)

or

T1 x T2 x ... x Tn

They correspond to cartesian products thus products types

By allowing you to be named they become records

Or potentially in other languages ... structs, classes, etc

So you find the total number of options by multiplying the maximum value of each option.

https://www.stephanboyer.com/post/18/algebraic-data-types

```elm
type alias Counter =
    { value : Int
    , count : Int
    }

counter : Counter
counter =
    { value = 0
    , count = 0
    }
```

```
-- Accessing properties
getValue : Counter -> Int
getValue counter = counter.value


getValue counter
> 0

getValue { value = 2, count = 1}
> 2
```

```
-- Shorthand accessors
getValueShort : Counter -> Int
getValueShort counter = .value counter


getValueShort counter
> 0

getValueShort { value = 2, count = 1}
> 2
```

```
-- Updating Single Property
updateValue : Int -> Counter -> Counter
updateValue newValue existingCounter =
    { existingCounter | value = newValue }



updateValue 10 counter
> { value = 10, count 0 }
```

```
-- Updating Multiple Properties
updateValueWithCount : Int -> Counter -> Counter
updateValueWithCount newValue existingCounter =
    { existingCounter
    | value = newValue
    , count = existingCounter.count + 1
    }


updateValueWithCount 10 counter
> { value = 10, count 1 }
```

Python

```python
dict_counter = {"value": 1, "count": 1}

>>> dict_counter["value"]
1
>>> dict_counter["value"] = 2
>>> dict_counter["value"]
2
>>> dict_counter["count"]
1

def update(val, rec):
    cnt["value"] = val
    cnt["count"] = int(cnt("count", 0)) + 1
    return cut

>>> update(33, dict_counter)
>>> dict_counter["value"]
33
>>> dict_counter["count"]
2
```

```python
class Counter(object):

    def __init__(self, val):
        self._value = val
        self._count = 0

    @property
    def count(self):
        return self._count

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, value):
        self._value = value
        self._count += 1
```

```python
>>> cnt = Counter(1)
>>> cnt.value
1
>>> cnt.count
1
>>> cnt.value = 22
>>> cnt.value
22
>>> cnt.count
2
```

Javascript

```
var counter = { _count: 0, _value: 0};

Object.defineProperty(counter, "count", {
    get: function() { return this._count; }
})

Object.defineProperty(counter, "value", {
    get: function() { return this._value; },
    set: function(v) { this._count++; return this._value = val; }
})

>>> counter.value;
0
>>> counter.count;
0
>>> counter.value = 44;
>>> counter.value;
44
>>> counter.count;
1
```

Extensible Records in Elm

```elm
type alias Record2 =
    { value : Int
    , count : Int
    , foo : String
    }

type alias Record3 =
    { value : Int
    , count : Int
    , bar : String
    }

type alias Record4 =
    { value : Int
    , foobar : String
    }
```

```elm
-- Extensible record alias
type alias ValueRecord a =
    { a | value : Int }


getValue2 : ValueRecord a -> Int
getValue2 valRec = valRec.value

-- COMPILES: getValue2 record2
-- COMPILES: getValue2 record3
-- COMPILES: getValue2 record4
-- COMPILES: getValue2 { value = 0 }
-- FAILURE: getValue2 { foo = 0 }
-- FAILURE: getValue2 {}
```

We can make our update function more generic and readable

```
updateRecord
    : Int
    -> { a | value : Int, count : Int }
    -> { a | value : Int, count : Int }
updateRecord rec newValue =
    { rec
    | value = newValue
    , count = count + 1
    }

-- COMPILES: updateRecord 1 record2
-- COMPILES: updateRecord 1 record3
-- DOESN'T COMPILE: updateRecord 1 record4
```

```
type alias ValueRecord a b =
    { a | value : b, count : Int }

updateRecord
    : Int
    -> ValueRecord a b
    -> ValueRecord a b
updateRecord rec newValue =
    { rec
    | value = newValue
    , count = count + 1
    }

-- COMPILES: updateRecord 1 record2
-- COMPILES: updateRecord 1 record3
-- COMPILES:
  updateRecord “a” { value = “b”, count = 0 }
```

Nulls?

Do NOT exist

Yay!

If we can't have a null we need something to help us

What does a NULL mean?

It means we have “nothing” when we might have been expecting “something”

So what about something called "either"?

```
type Either a b
    = Left a
    | Right b
```

This is something we could use (and do use often) but our need is less generic.

Each side isn't some general structure. One means something very specific.

So what about?

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

This is closer but our result isn't really an error it's "Nothing"

So maybe we could go with:

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

Now we're on to something.

Maybe we Just have a value (of type a) or we have Nothing

```
giveMeIfGreatherThan0 : Int -> Maybe Int
giveMeIfGreatherThan0 val =
    if val > 0 then
        Just val
    else
        Nothing
```

```
giveMeIfGreatherThan0 10
> Just 10

giveMeIfGreatherThan0 -23
> Nothing
```

But now we have this structure that we have to deal with

And we don’t want our code to look like Go’s error handling code

Where you have this same code everywhere …

```go
f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
```

Thankfully there are functions in the maybe package to help us out

```
withDefault : a -> Maybe a -> a
withDefault default maybe =
    case maybe of
      Just value -> value
      Nothing -> default
```

```
withDefault 10 (Just 15) -- 15
withDefault 10 (Nothing) -- 10
withDefault "foo" (Just "bar") -- "bar"
withDefault "foo" (Nothing) -- “foo”
```

```elm
-- Map
map : (a -> b) -> Maybe a -> Maybe b
map f maybe =
    case maybe of
      Just value -> Just (f value)
      Nothing -> Nothing
```

So we take in a function of (a -> b), Maybe a and return Maybe b

Let’s break this down

We may have an “a” and if do we want to get back a “b”

So the function we give needs to take an "a" and give back a "b"

The map will take care of the actual application

If you give it a “Just a” it will take the “a” out of the “Just” and apply your function

It will take the result of that function and put that inside a “Just”

If you give it “Nothing” it will skip applying the function and will just give you “Nothing”

Let's create an function of (a -> b) and walk through

```elm
add1 : Int -> Int
add1 val = val + 1
-- In this case we use the same type.
-- It doesn't have to be 2 different types.



-- but we can do that
positiveMessage : Int -> String
positiveMessage val =
    if val > 0 then
        "I'm a positive message!"
    else
        "I'm not so postivie :("
```

And now when we “run” it:

```
map add1 (Just 10)
> Just 11

map add1 Nothing
> Nothing

map positiveMessage (Just 10)
> Just "I'm a positive message!”

map positiveMessage (Just -23)
> Just "I'm not so positive :(“

map positiveMessage Nothing
> Nothing
```

Javascript

We're going to cheat and use a javascript maybe library

https://github.com/alexanderjarvis/maybe

```
import { maybe } from 'maybes'
import { maybe, just, nothing } from 'maybes'

const value = maybe(1) // Just(1)
value.isJust() // true
value.isNothing() // false
value.just() // 1 (or could error since JS is not safe)
value.map(v => v + 1) // Just(2)

const empty = maybe(null)
empty.isJust() // false
empty.isNothing() // true
empty.just() // throws error
empty.map(v => v + 1) // noop (No Operation)
empty.map(v => v.toUpperCase()).orJust('hello') // 'hello'
```

Lists

```elm
simpleList : List Int
simpleList = [1, 2, 3, 4]


insertIntoSimpleList : Int -> List Int
insertIntoSimpleList num = num :: simpleList


insertIntoSimpleList 10
> [10, 1, 2, 3, 4]

insertIntoSimpleList 333
> [333, 1, 2, 3, 4]
```

Loops

Elm doesn’t have them

We instead use functions

(Remember that passing functions around like data thing)

We’ll start with a “fold”

(AKA: Reduce)

```
-- This is a fold left of Ints
-- The left means we reduce (or traverse) from the left
foldlInt : (Int -> List Int -> List Int) -> Int -> List Int -> Int
foldlInt func aggVal list =
    case list of
        [] ->
            aggVal

        x :: xs ->
            foldl func (func x aggVal) xs
```

Ooh, we've got that "higher order functions" thing again

(A function that takes a function in this case)

It’s first argument is a function of two values that returns one

(Int -> List Int -> List Int)

It then takes some value to accumulate into. Something that can accumulate like a list, a string, a number.

It also takes a starting value

```
foldlInt (::) [] [1,2,3]


-- (::) is called `cons`
(::) : a -> List a -> List a
```

So this takes a function of “cons” or “insert at 0 index”

(insert into the 0 index of a list)

As well as an empty list to accumulate into

And of course our starting list

Let's step through our fold

Thus starting from the left of [1,2,3] we prepend 1 to our starting empty list [].

We now have [1]

And then inserting 2 into the 0 index of [1]

We now have [2, 1]

And then we finish by prepending the 3 to [2,1]:

[3,2,1]

```
-- We also can fold from the right
foldrInt : (Int -> List Int -> List Int) -> Int -> List Int -> Int



foldrInt (::) [] [1,2,3] == [1,2,3]
-- Using the same arguments as before we end
--   up with the same list we started
-- Since we started from the right we prepend 3
--   we have [3]
-- And the inserting 2 into the 0 index we have [2, 3]
-- And then we finish by prepending the 1: [1,2,3]
```

Python

```python
from functools import reduce

def insert(arr, val):
    arr.insert(0, val)
    return arr

arr = [1, 2, 3]
>>> reduce(insert, [], arr)
[3, 2, 1]
```

Javascript

```
const arr = [1, 2, 3];
const reducer = (accumulator, currentValue) =>
{ accumulator.push(currentValue); return accumulator };

> arr.reduce(reducer, []);
[ 1, 2, 3]


const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) =>
accumulator + currentValue;

// 1 + 2 + 3 + 4
> array1.reduce(reducer);
10

// 5 + 1 + 2 + 3 + 4
> array1.reduce(reducer, 5);
15
```

Back to Elm

We of course can pass in different types of functions

What if we use the max function?

```
foldlInt max 0 [1,2,3] == 3


-- We give it the builtin max function
-- that is a more generic version of:
max : Int -> Int -> Int
max x y = if x > y then x else y
```

We can then make a nice function to get the maximum integer

```
maximumInt : List Int -> Int
maximumInt list =
  case list of

    [] ->
        0

    [x] ->
        x

    x :: xs ->
        foldl max x xs
```

But this kinda sucks. If we give an empty list we get back a 0.

```
maximumInt [1,2] == 2
maximumInt [1] == 1
maximumInt [] == 0 -- Blech!
```

This is standard in other types of languages but we're using strong-type FP for a reason

What if we introduce a "Maybe"?

```
maximumInt : List Int -> Maybe Int
maximumInt list =
  case list of
    x :: xs ->
      Just (foldl max x xs)

    _ ->
      Nothing
```

Now it’s obvious to us when our result has no maximum value as we gave it no values

```
maximumInt [1,2] == Just 2
maximumInt [1] == Just 1
maximumInt [] == Nothing
```

And if we want to get a 0 value when the list is empty we just use what we already have:

```
withDefault 0 (maximumInt [1,2]) == 2
withDefault 0 (maximumInt [1]) == 1
withDefault 0 (maximumInt []) == 0
```

While we're here, we do this function chaining quite a bit.

And we're not a lisp so we'd like to avoid all of the parentheses.

(foo (bar 1 (another “a” “b”)) 3)

Thankfully elm gives us some nice symbols to use

(These symbols are actually functions known as infix operators)

```
withDefault 0 <| maximumInt [1,2]
> 2

withDefault 0 <| maximumInt [1]
> 1

withDefault 0 <| maximumInt []
> 0
```

This allows us to do nice, readable chaining when we have many functions

```
withDefault 0
    <| maximumInt
    <| range 0 10

> 10



-- other direction:
[1,2,3]
    |> maximumInt
    |> withDefault 0

> 3
```

Let’s go back and tweak our fold functions to make them more generic

```
foldl : (a -> b -> b) -> b -> List a -> b
foldl func acc list =
    case list of
        [] ->
            acc

        x :: xs ->
            foldl func (func x acc) xs
```

Now our fold function can work on many different data types.

With generic fold functions what can we do?

It turns out, quite a lot.

```
map : (a -> b) -> List a -> List b
map f xs =
  foldr (\x acc -> f x :: acc) [] xs

map (\x -> x + 1) [1,2,3]
> [2,3,4]

add1 : Int -> Int
add1 x = x + 1

map add1 [1,2,3]
> [2,3,4]
```

Javascript & Python

```
// Javascript
var arr = [1, 2, 3]
> arr.map(x => x + 1)
[2, 3, 4]
```

```
# Python
arr = [1, 2, 3]
>>> map(lambda x: x + 1, arr)
[2, 3, 4]
```

There are all kinds of functions for working with lists in the List.elm module in the standard library

Many of these leverage fold (and each other)

We end up with a library of functions that will look very similar to what you find in Python, Javascript, Ruby, etc isEmpty, length, reverse, member, head, tail, filter, take, drop, sum, all, etc

Immutability

Elm is functional and immutable … so can we store and mutate state?

Yes, but we need to leverage “let” expressions

```
forty : Int
forty =
    let
        twentyFour =
            3 * 8

        sixteen =
            4 ^ 2
    in
        twentyFour + sixteen
```

You just can't reassign the same variable

```
bad : Int
bad =
    let
        twentyFour =
            3 * 8

        twentyFour =
            20 + 4
    in
        twentyFour
```

This will not compile. You will get this message:

There are multiple values named `twentyFour` in this let-expression.

Search through this let-expression, find all the values named `twentyFour`, and give each of them a unique name.

```
good : Int
good =
    let
        twentyFour =
            3 * 8

        newTwentyFour =
            20 + 4
    in
        newTwentyFour
-- This will compile.
```

2 Other Notes on Let expressions

You can assign functions

And you can provide type annotations

```elm
letFunctions : Int
letFunctions =
    let
        hypotenuse a b =
            sqrt (a^2 + b^2)

        name : String
        name =
            "Hermann"

        increment : Int -> Int
        increment n =
            n + 1
    in
        increment 10
```

Those are the basics you need to be successful in Elm

But if you're still nervous don't worry.

You have another tool available to you

The compiler

Don’t fight the compiler.
Let it help you.

You don’t even have to provide the types

The compiler will give you the types!!!

```
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1
```

```
============= WARNINGS ================

-- missing type annotation - /.../elm-counter/src/elm/Main.elm

Top-level value `update` does not have a type annotation.

29| update msg model =
    ^^^^^^

I inferred the type annotation so you can copy it into your code:

update
  : Msg
  -> Model
  -> Model
```

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1
```

And it will even let you know when you might be overly strict

```
============= WARNINGS ================

-- missing type annotation - /.../elm-counter/src/elm/Main.elm

Top-level value `update` does not have a type annotation.

29| update msg model =
    ^^^^^^

I inferred the type annotation so you can copy it into your code:

update
  : Msg
  -> { a | value : Int }
  -> ( { a | value : Int }, Cmd Msg )
```

The Elm compiler is a really helpful tool.

Don’t fight it.

Don’t attempt to out smart it and write the perfect code if you’re unsure.

Just get something in and compile.

One last syntax thing before we move into architecture and building applications

Debugging

In the standard library (core) we are provided some very helpful debugging functions

Debug.log

```
log : String -> a -> a
```

Log a tagged value on the developer console, and then return the value.

```
1 + log "number" 1
-- equals 2, logs "number: 1"
```

```
length (log "start" [])
-- equals 0, logs "start: []"
```

Notice that log is not a pure function! It should only be used for investigating bugs or performance problems.

```
myFunc : Action -> String
myFunc action  =
  case action of
    act1 ->
      "It's act 1"

    act2 ->
      "It's act 2"
```

```elm
myFunc : Action -> String
myFunc action  =
    let
        _ = Debug.log "Action: " action
    in
        case action of
            act1 ->
                "It's act 1"

            act2 ->
                "It's act 2"
```

```
myFunc : Action -> String
myFunc action  =
    case (Debug.log "Action: " action) of
        act1 ->
            "It's act 1"

        act2 ->
            "It's act 2"
```

Debug.crash

```
crash : String -> a
```

Crash the program with an error message. This is an uncatchable error, intended for code that is soon-to-be-implemented.

USE THIS
if you want to do some testing while you are partway through writing a function.

DO NOT USE THIS IF
you want to do some typical try-catch exception handling. Use the Maybe or Result libraries instead.

The Elm Architecture

Model
Update
View

Model

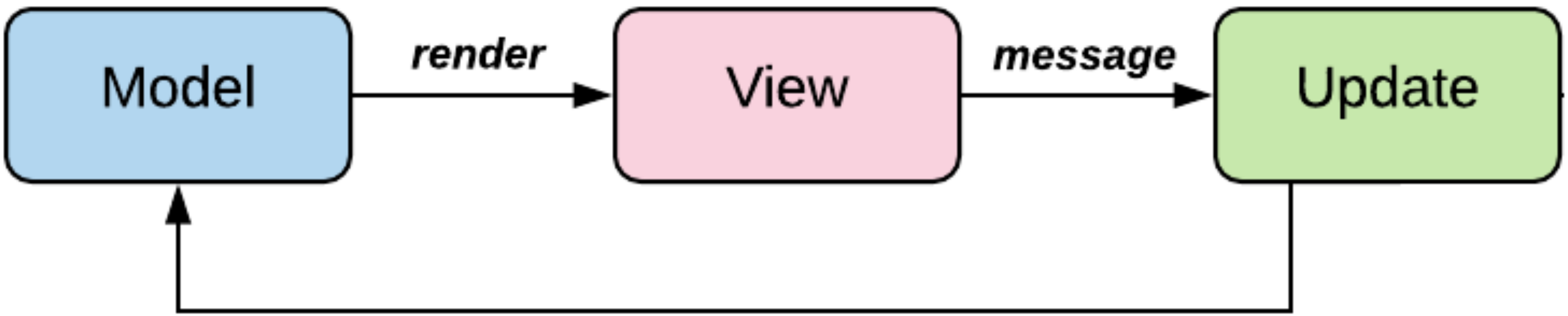
The state of your application

Update

A way to update your state

View

A way to view
your state as HTML



http://elmprogramming.com/subscriptions.html

This architecture is very important as it helps us with …

Side Effects

Historical FUD:

"You can't build anything in real in functional programming because it's 'pure'"

Then came along Philip Wadler

Monads for Functional Programming

Wadler, 1992

http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf

We finally mentioned Monads

When people are freaked out by these languages it’s often around side effects and those specific MONADS

IO ()

```haskell
getMeetupNameAndVenues :: GroupId -> IO [(Text, Text)]
getMeetupNameAndVenues groupId =
  getWith (eventsOptions groupId) meetupEventsUrl
  >>= asValue
  >>= ((^.. responseBody
       . key "results"
       . _Array . traverse)
        >>> map ((^. key "name" . _String)
                  &&& (^. key "venue"
                       . key "name" . _String)
                 )
        >>> return
```

Purescript

Eff ()
Aff ()

```
forall eff. Eff (console :: CONSOLE, random :: RANDOM | eff) Unit
```

```
eval :: Query ~> H.ComponentDSL State Query Void (Aff (ajax :: AX.AJAX | eff))
eval = case _ of
    SetUsername username next -> do
      H.modify (_ { username = username, result = Nothing :: Maybe String })
      pure next
    MakeRequest next -> do
      username <- H.gets _.username
      H.modify (_ { loading = true })
      response <- H.liftAff $ AX.get ("https://api.github.com/users/" <> username)
      H.modify (_ { loading = false, result = Just response.response })
      pure next
```

Elm Doesn’t Support This

All Side Effects are handled by the architecture via tasks and "effect managers"

You likely will not need to write an effect manager (and only occasionally will you write tasks)

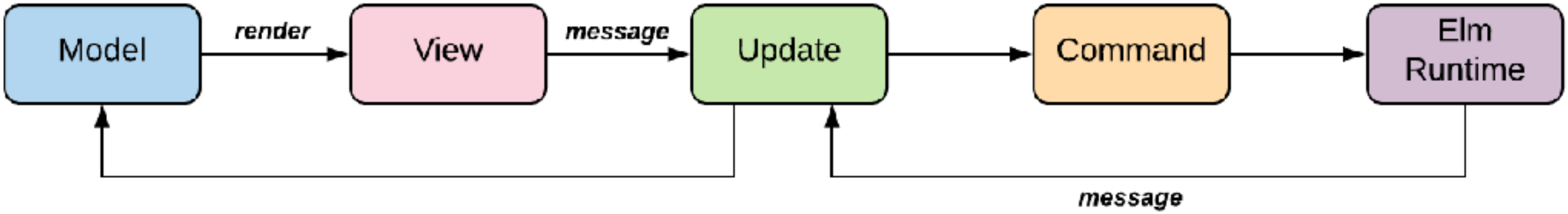
Elm's goal is to provide all necessary effect managers as part of the standard library or as provided libraries

But if you need to write them you can

Side Effects in Elm

Elm uses commands for side effects

You trigger a command which will then trigger actions as part of it's effects



http://elmprogramming.com/subscriptions.html

Http

http://elm-lang.org/examples/http

Example:
Making HTTP Requests

(Loading gifs)

```
type alias Model =
  { topic : String
  , gifUrl : String
  }


init : (Model, Cmd Msg)
init =
  (Model "cats" "waiting.gif", Cmd.none)
```

```
view : Model -> Html Msg
view model =
  div []
    [ h2 [] [text model.topic]
    , img [src model.gifUrl] []
    , button
        [ onClick MorePlease ]
        [ text "More Please!" ]
    ]
```

```elm
type Msg
    = MorePlease
    | NewGif (Result Http.Error String)



update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    MorePlease ->
      (model, getRandomGif model.topic)

    NewGif (Ok newUrl) ->
      ( { model | gifUrl = newUrl }, Cmd.none)

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

```elm
getRandomGif : String -> Cmd Msg
getRandomGif topic =
  let
    url =
      "https://api.giphy.com/v1/gifs/random?"
      ++ "api_key=dc6zaTOxFJmzC&tag=" ++ topic

    request =
      Http.get url decodeGifUrl
  in
    Http.send NewGif request


decodeGifUrl : Decode.Decoder String
decodeGifUrl =
  Decode.at ["data", "image_url"] Decode.string
```

```
Http.get
    : String
    -> Decode.Decoder value
    -> Http.Request value

Http.send
    : (Result Error value -> msg)
    -> Http.Request value
    -> Cmd msg

-- MSG:
-- NewGif (Result Http.Error String)

-- UPDATE:
-- NewGif (Ok newUrl) ->
--        ( { model | gifUrl = newUrl }, Cmd.none)
```

Time

http://elm-lang.org/examples/time

Example:
"Subscribing" to Time

(Making a Clock)

```
main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }
```

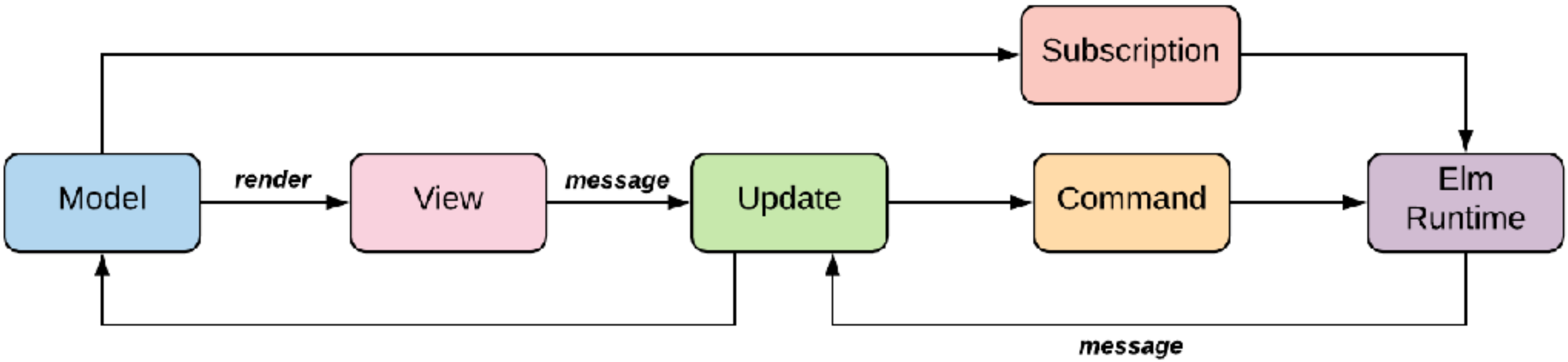
```
type alias Model = Time


init : (Model, Cmd Msg)
init =
  (0, Cmd.none)
```

```elm
view : Model -> Html Msg
view model =
  let
    angle =
      turns (Time.inMinutes model)

    handX =
      toString (50 + 40 * cos angle)

    handY =
      toString (50 + 40 * sin angle)
  in
    svg [ viewBox "0 0 100 100", width "300px" ]
      [ circle [ cx "50", cy "50", r "45", fill "#0B79CE" ] []
      , line [ x1 "50", y1 "50", x2 handX, y2 handY, stroke "#023963" ] []
      ]
```

```elm
type Msg
  = Tick Time



update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Tick newTime ->
      (newTime, Cmd.none)
```

```
main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }
```

```elm
import Time exposing (Time, second)

subscriptions : Model -> Sub Msg
subscriptions model =
  Time.every second Tick




-- ‘Every' Type Signature
every : Time -> (Time -> msg) -> Sub msg

-- ‘second' Type Signature
second : Time

-- ‘Time’ MSG
type Msg = Tick Time
```



http://elmprogramming.com/subscriptions.html

Javascript Interop

Example:
Calling into Javascript from Elm

(A Spellchecker)

This is how you inject your Elm app into your browser via Javascript …

```
<div id="spelling"></div>
<script src="spelling.js"></script>
<script>
    var app = Elm.Spelling.fullscreen();
</script>
```

```
### NOTE:
spelling.js is the Javascript generated by the
Elm compiler
```

Let’s add our javascript functions

```html
<div id="spelling"></div>
<script src="spelling.js"></script>
<script>
    var app = Elm.Spelling.fullscreen();

    app.ports.check.subscribe(function(word) {
      var suggestions = spellCheck(word);
      app.ports.suggestions.send(suggestions);
    });

    fruits = []

    function spellCheck(word) {
      // You can check on the js console if fruits
      // was updated by elm typing fruits
      fruits.push(word);
      return fruits;
    }
</script>
```

```elm
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }



type alias Model =
    { word : String
    , suggestions : List String
    }


init : ( Model, Cmd Msg )
init =
    ( Model "" [], Cmd.none )
```

```
view : Model -> Html Msg
view model =
    div []
        [ input [ onInput Change ] []
        , button
            [ onClick Check ]
            [ text "Check" ]
        , div
            []
            [ text (String.join ", " model.suggestions)
            ]
        ]
```

```elm
type Msg
    = Change String
    | Check
    | Suggest (List String)


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Change newWord ->
            ( Model newWord [], Cmd.none )

        Check ->
            ( model, check model.word )

        Suggest newSuggestions ->
            ( Model model.word newSuggestions, Cmd.none )


port check : String -> Cmd msg
```

```javascript
// Javascript Function
app.ports.check.subscribe(function(word) {
    var suggestions = spellCheck(word);
    app.ports.suggestions.send(suggestions);
});
```

```elm
-- Elm Wrapper
port check : String -> Cmd msg
```

```elm
port suggestions : (List String -> msg) -> Sub msg


subscriptions : Model -> Sub Msg
subscriptions model =
    suggestions Suggest
```

```
app.ports.check.subscribe(function(word) {
    var suggestions = spellCheck(word);
     // Javascript Function
     app.ports.suggestions.send(suggestions);
});
```

```
-- Elm Wrapper
port suggestions : (List String -> msg) -> Sub msg
```

And now you can call javascript functions as well as subscribe to javascript functions!

Terminology

We've seen functions like `map` and the concept of applying functions to data

(Instead of looping through data)

We often call data structures that can be mapped over: mappable

(Look at us making up words)

But there is precise term from mathematics to describe this concept:

(Wait for it)

Functor

In mathematics, a functor is a type of mapping (a homomorphism) between categories arising in category theory. In the category of small categories, functors can be thought of more generally as morphisms.

(https://en.wikipedia.org/wiki/Functor)

In non category theory …

A functor is simply something that can be mapped over.

In Haskell and Purescript we have a “typeclass”

(This is something like an “interface” or “abstract class” in other languages)

These allows us to define rules for data structures

```haskell
class Functor f where
      fmap :: (a -> b) -> f a -> f b
```

```haskell
-- List Functor Instance
instance Functor [] where
      fmap = map


-- Maybe Functor Instance
instance Functor Maybe where
      fmap _ Nothing          = Nothing
      fmap f (Just a)         = Just (f a)


-- Either Functor Instance
instance Functor (Either a) where
      fmap _ (Left x) = Left x
      fmap f (Right y) = Right (f y)
```

With those classes defined you can now fmap over those structures.

So in Haskell and Purescript when you define new structures you can implement classes for those methods as well

Elm does not support this concept so we accomplish it a different way

We just create the functions in the modules and use them directly

So instead of having one `map` function that works over all structures that implement the class

We create a function per structure:

List.map
Maybe.map
Result.map

Btw the next step up beyond Functor is Applicative Functor which then leads to Monads

(Oh no!)

We’re not going to dive into those today

:)

But we have been using them throughout the presentation!

(Wut!)

Let’s look at another common pattern

We’ll start with multiplication

When we multiply something by 1 we always get the that value back

$$x * 1 == x$$
$$1 * x == x$$

We have similar with addition

$$x + 0 == x$$
$$0 + x == x$$

A similar when combining lists

Let x = [1,2]
x ++ [] == x
[] ++ x == x
append x [] == x
append [] x == x

These all share the same properties

The function takes 2 parameters

The parameters and the returned value are the same type

There exists a value that doesn't change other values when used with the binary function

And another property that we haven’t shown yet

They are associative

$$(2 + 7) + 4 == 2 + (7 + 4)$$

Notice this kills things like subtraction and division

$$(2 - 7) - 4 \;\; != \;\; 2 - (7 - 4)$$

$$(2 / 7) / 4 \;\; != \;\; 2 / (7 / 4)$$

Structures that adhere to these properties (laws) are known as:

Monoids

In abstract algebra, a branch of mathematics, a monoid is an algebraic structure with a single associative binary operation and an identity element.

(https://en.wikipedia.org/wiki/Monoid)

```haskell
class Monoid m where
      mempty :: m
      mappend :: m -> m -> m
      mconcat :: [m] -> m
      mconcat = foldr mappend mempty
```

Why do we care about Monoids?

There are tons of reasons.

But here is one example …

Our hint was in the last line of the Monoid type class and we showed it earlier

Folds

The fold type definition:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

For it’s arguments it requires:

A function that takes 2 arguments

An initial value that when applied to the binary function doesn't change

(This gives us our base case or starting point)

And of course the item(s) we'll apply the function to and start with the initial value

```
foldr (*) 1 [1,2,3]
> 6

foldr (+) 0 [2,3,4]
> 9

foldr (++) [] [[1,2,3], [20, 30, 40]]
> [1,2,3,20,30,40]
```

This means that if we have a monoid or if we can make a structure become a monoid …

Then we get all of this free code

Free code that follow mathematical laws

Which means that free code his highly unlikely to change (especially the API) and thus …

It's free code without many of the pains that come with dependent code and dealing with changes, versions, etc

(The best and really, only kind of free code)

This is part of the reason folks in these worlds get so excited about precise terms and laws

These things aren’t random or context specific.

They are precise.

When you say monoid it means that it has those specific characteristics

It can have more than those characteristics of course but at minimum it must have those to be a monoid

From functor we get to applicative to monad and from there the ability to implement real programs

And more keeps coming out like things like “Free” which allow us to think about the entire structure/architecture of our program

This is why people around you this week will be super excited.

You should dive in.

Expose yourself to this world.

Open your mind and expect to feel “dumb”

It’s ok.

We all do.

Moving Forward

I highly recommend you give Elm a shot

And then give OCaml, F-Sharp, Haskell, Idris and Purescript a look.

Especially coming from Elm I'd give Purescript a try.

There is a great set of libraries from `rgempel` on GitHub that is a bunch of Elm's modules implement in Purescript

This will give you a good mapping of how the Elm concepts and terminology map into Purescript

```
-- | Equivalent to Purescript's `show`.
toString :: ∀ a. (Show a) => a -> String
toString = show

-- | Equivalent to Purescript's `<<<`.
compose :: ∀ a b c. (b -> c) -> (a -> b) -> (a -> c)
compose = (<<<)

-- | Equivalent to Purescript's `$`.
applyFn :: ∀ a b. (a -> b) -> a -> b
applyFn = ($)

-- | The Purescript equivalent is `id`.
identity :: ∀ a. a -> a
identity = id

-- | The Purescript equivalent is `const`.
always :: ∀ a b. a -> b -> a
always = const

-- | The Purescript equivalent is `Void`.
type Never = Void
```

And if you have or you're already wanting more then here a few resources.

First, if you're going to buy a single book then buy "Haskell Programming from first principles" aka "The Haskell Book" even if you're wanting to learn Elm

And it’s not yet finished but Richard Feldman’s “Elm in Action” will almost certainly be one of the best resources specific to Elm

Resources

- Elm Website :: Your best resource for Elm
  - http://elm-lang.org/
- Elm to Purescript
  - https://github.com/pselm
- Elm in Action
  - https://www.manning.com/books/elm-in-action
- Haskell Programming from first principles (Haskell Book)
  - http://haskellbook.com/
- Type Driven Development Book
  - https://www.manning.com/books/type-driven-development-with-idris
- PureScript Conf 2018
  - June 6th … here!
  - https://github.com/lambdaconf/lambdaconf-2018/wiki/PureScript-Conf-2018#schedule
- Elm-Conf 2018
  - September 26, 2018 in St. Louis, MO as a Strange Loop pre-conference event
  - https://www.elm-conf.us/

Thank you!

(And please come find me if you have questions.)

Beau Lyddon

Managing Partner at Real Kinetic

We mentor engineering teams to unleash your full potential.

@lyddonb
linkedin.com/in/beau-lyddon
github.com/lyddonb

REALKINETIC
YOUR POTENTIAL REALIZED