Introduction to Elm

About me

- Started as PHP dev
- Used JavaScript as primary language for 10 years (2007 - 2016)
    - Web and app development
    - Frameworks
    - Embedded systems
- At present working with Elm, Go, C

Other interests

- Privacy - Encryption, Tor
- Unix
- Hacking
- Coffee

The question is not “Why would you use Elm?” but
“Why aren’t you using it?”

JavaScript

- Current state of JavaScript app development is overly complex.
- Need to learn and integrate 3rd-party tools to try and control state and minimize bugs.
- Too many apps feel cobbled together and not well architected.
- No compiler support.
- OOP

Elm

+ No runtime exceptions!*
+ Elm is a pure functional language. It is easy to reason about the data and the application.
+ Refactoring is easy. By contrast, refactoring even seemingly simple code in JavaScript is error-prone and risky.
+ The compiler has your back!
+ Meaningful (and verbose) error messages.

* Well, mostly. There are some exceptions, i.e., division by zero and bad RegExps that the compiler isn't catching right now.

Thinking functionally will make you a better programmer!

This presentation will be covering...

1. Basic functional programming primer
2. Types
3. Extensible records
4. The Elm architecture

This presentation won't be covering...

1. Interoperability with JavaScript
2. Subscriptions
3. Passing in values at runtime
4. Tooling
5. Elm at scale

Let’s get started...

Why is functional programming a good thing?

- Pure functions
- Function composition
- Easy to reason about
- Easy to debug
- Easy to understand

Forward function application

```
query
     |> Dict.foldl fmtEquality ""
     |> String.dropRight 5
     |> Request.Consumer.query
     |> Http.send Fetch Consumers
```

Function composition

```
Http.send Fetch Consumers
     << Request.Consumer.query
     << String.dropRight 5
     << Dict.foldl fmtEquality ""
     <| query
```

Also...

You get immutable values, static types and currying for free.*

* Not native to JavaScript.

FP Primer

Terms

- Arity : The number of arguments that a function takes.
- Higher-order Function : A function that can take another function as an argument and can also return a function.
- Function Composition : Combining simple functions to build more complicated ones. Unix pipelines.
- Side Effects : Changes in state that do not depend on the input functions.

Terms, continued

- Pure Function : A function without side effects. A pure function has no free variables.
- Currying : Translating a function with multiple arguments into a sequence of function calls that take one argument.
- Partial Application : Fixing (binding) a number of arguments to a function to produce another function with a smaller arity.

```
programmers : List ( List String )
programmers =
      [ [ "Ken", "Thompson", "American", "B" ]
      , [ "Dennis", "Ritchie", "American", "C" ]
      , [ "Bjarne", "Stroustrup", "Danish", "C++" ]
      , [ "Evan", "Czaplicki", "American", "Elm" ]
      , [ "Rob", "Pike", "Canadian", "Go" ]
      , [ "Brendan", "Eich", "American", "JavaScript" ]
      , [ "Guido", "van Rossum", "Danish", "Python" ]
      ]
```

```
getLanguages =
      List.map
            ( Maybe.withDefault ""
                  << List.head
                  << List.reverse
            )
```

```
programmers |> getLanguages


["B","C","C++","Elm","Go","JavaScript","Python"]
```

```
makeList =
      List.map
            ( \r -> li [] [ r |> text ] )

main =
      ul []
      ( makeList
            << getLanguages
            <| programmers
      )
```

```
getNationality a =
     List.filter
          ( \r ->
               (==)
                    ( List.drop 2 r
                         |> List.head
                         >> Maybe.withDefault ""
                    )
                    a
          )
```

```
getAmericans =
     "American" |> getNationality


getDanes =
     "Danish" |> getNationality
```

```
main =
     div []
          [ text
               << toString
               << getLanguages
               << getDanes
               <| programmers
          ]
```

```
main =
     div []
          [ text
               << toString
               << List.reverse
               << getLanguages
               << ( \p ->
                    List.concat
                         [ p |> getAmericans
                         , p |> getDanes
                         ]
                    )
               <| programmers
          ]
```

Hello World

Hello, World!

```
module Hello exposing (..)
import Html exposing (text)

main =
     text "Hello, World!"
```

1. `elm reactor`
2. Open browser
3. Click on file

And here's a list...

```
module Main exposing (..)

import Html exposing (Html, div, h1, li, text, ul)
import Html.Attributes exposing (style)

main =
     div [] [
          h1 [] [ "Foundational programmers" |> text ]
          , ul [] [
               li [] [ "Ken Thompson" |> text ]
               , li [] [ "Brian Kernighan" |> text ]
               , li [
                   [ ( "background-color", "blue" ) , ( "font-weight", "bold" ) ] |> style ]
                   [ "Dennis Ritchie" |> text ]
               ]
     ]
```

Types

Union types

```
type Msg
    = Add
    | Delete User
    | Get User
    | Post User
    | Put User
```

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

Union types with type variable(s)

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

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

Type alias

```
type alias Name
    = String


type alias Age
    = Int
```

```
type alias Message a =
    { code : String
    , body : a
    }
```

```
type alias Message a =
     { code : String
     , body : a
     }

Message "1337" "foo" |> toString >> text

-----------------

m = Message "1337" [ 1, 2, 4]
n = ( Message "1337" ) [ "a", "b", "c"]

toString m |> text
n |> toString >> text

-----------------

text
     << toString
     << ( "1337" |> Message )
     <| ( "foo", "bar" )
```

Extensible Records

Extensible records are good to use when scaling your app, as it *narrows* your types.

That sounds great, but what does that mean?

```
type alias BarEmployee =
      { last : String
      , first : String
      }

type alias Model =
      { user : BarEmployee
      , city : String
      , state : String
      }

user : Model -> String
user model =
      model.user.first
      ++ " "
      ++ model.user.last

main =
      let
            model : Model
            model =
                  { user =
                  { last = "Kelly"
                  , first = "Charlie"
                        }
                  , city = "Philadelphia"
                  , state = "PA"
                  }
      in
      div [] [ model |> user |> text ]
```

```
user : BarEmployee -> String
user user =
      user.first
      ++ " "
      ++ user.last

…

div [] [ model.user |> user |> text ]
```

This is good, but we can do better!

```
type alias BarEmployee =
      { last : String
      , first : String
      }

type alias BarManager =
      { last : String
      , first : String
      , age : Int
      , location : String
      }

type alias Model =
      { user : BarManager
      , city : String
      , state : String
      }

user : BarEmployee -> String
user model =
      model.user.first
      ++ " "
      ++ model.user.last

main =
      let
            model = user | BarManager     ← pseudo-code!
            ...
      in
      div [] [ model.user |> user |> text ]
```

```
user :
      { n | last : String, first : String }
      -> String
user user =
      user.first
      ++ " "
      ++ user.last

…

div [] [ model.user |> user |> text ]
```

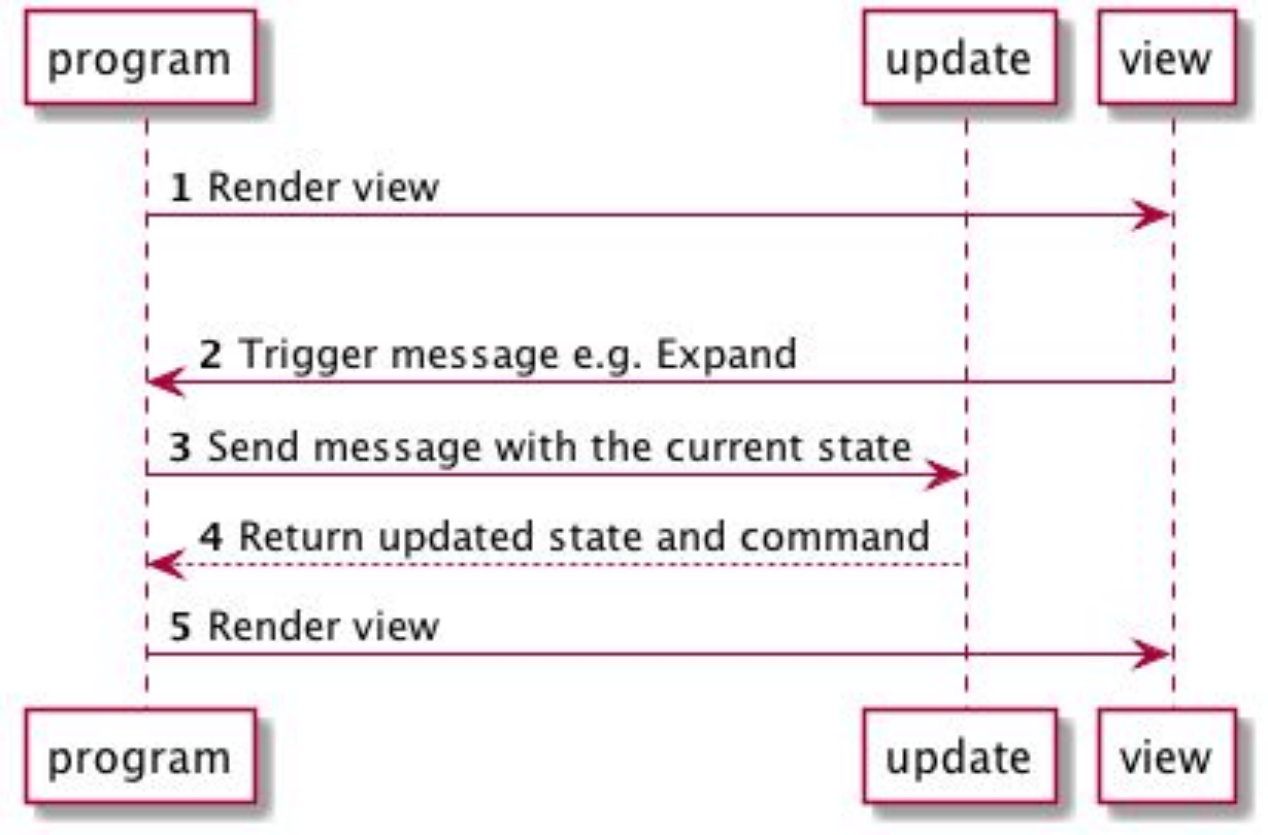
1. This is better yet. Now, any record type that contains a `last` and a `first` field can be used.

2. Also, this makes testing much easier, as the whole model doesn't need to be mocked in order to test a function that only pertains to a (user) name!

Let’s add some behavior!

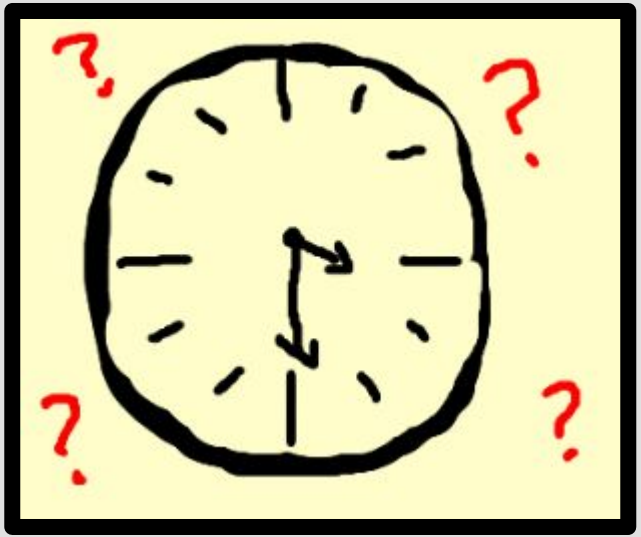
The Elm Architecture

```
-- MODEL

type alias Model
       = Int

-- UPDATE

type Msg
       = Decrease
       | Increase

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
       case msg of
              Decrease ->
                     ( model |> Bitwise.shiftRightBy 1 ) ! []

              Increase ->
                     ( model |> Bitwise.shiftLeftBy 1 ) ! []

-- VIEW

view : Model -> Html Msg
view model =
       div []
              [ h3 [] [ "Incremental bit shifting" |> text ]
              , button [ Decrease |> onClick ] [ "Decrease exponentially" |> text ]
              , button [ Increase |> onClick ] [ "Increase exponentially" |> text ]
              , span [] [ text << toString <| model ]
              ]

init : ( Model, Cmd Msg )
init =
       32 ! []

main =
       Html.program { init = init, update = update, view = view, subscriptions = always Sub.none }
```



https://www.elm-tutorial.org/en/02-elm-arch/04-flow.html

What time is it?




It’s Demo Time!!

My links

https://github.com/btoll

https://github.com/btoll/elm-remotepager-demo

http://www.benjamintoll.com

References and further reading

https://elm-lang.org/

https://www.elm-tutorial.org/en/ (the first part, not the when the app is built)

http://package.elm-lang.org

https://github.com/rtfeldman/elm-spa-example

Anything by Evan Czaplicki, Richard Feldman and the folks at NoRedInk

The End