Elm

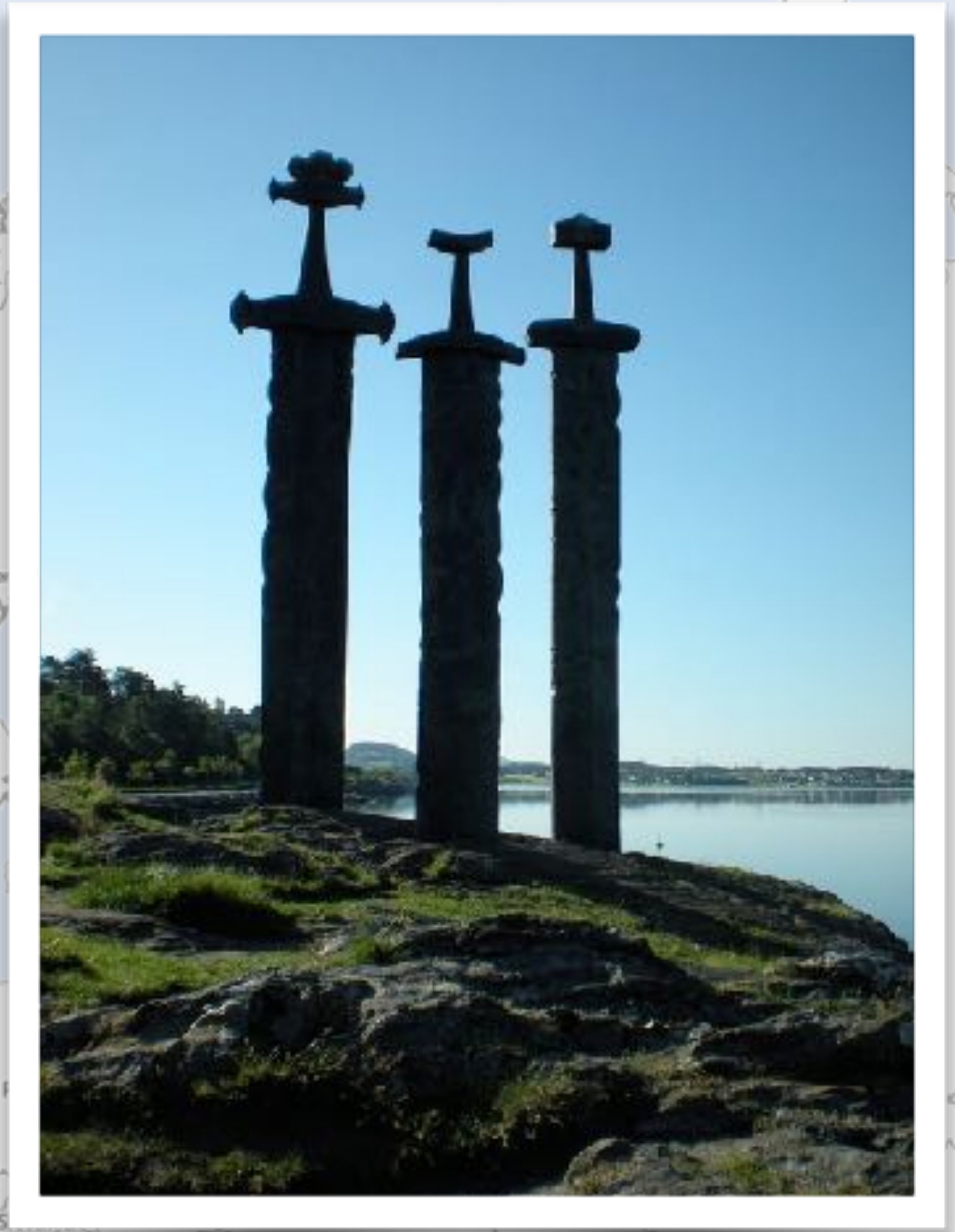
Functional Programming for the Web

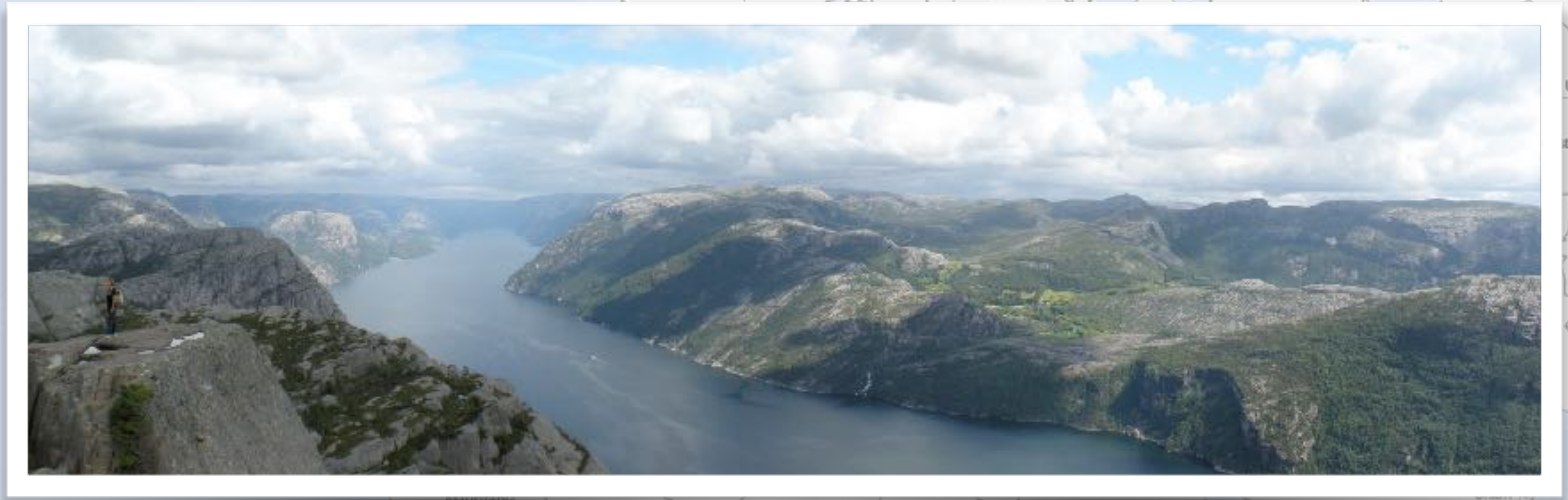
Austin Bingham

@austin_bingham

Canada
Iceland
Norway
Denmark
United Kingdom
Ireland
Netherlands
Germany
Belgium
Czech Rep.
France
Switzerland
Austria
Hungary
Moldova
Romania
Slovenia
Cuba
Dominican Rep.
Guatemala
Honduras
El Salvador
Nicaragua
Costa Rica
Panama
Venezuela
Colombia
Suriname
Ecuador
Mali
Niger
Chad
Sudan
Eritrea
Yemen
Senegal
Gambia
Guinea-Bissau
Guinea
Burkina Faso
Sierra Leone
Liberia
Côte d'Ivoire
Ghana
Benin
Nigeria
Cameroon
Central African Rep.
S. Sudan
Ethiopia
Somaliland
Somalia
Gabon
Congo
Dem. Rep. Congo
Uganda
Kenya
Tanzania
CARTODB
CartoDB attribution

SixtyNORTH



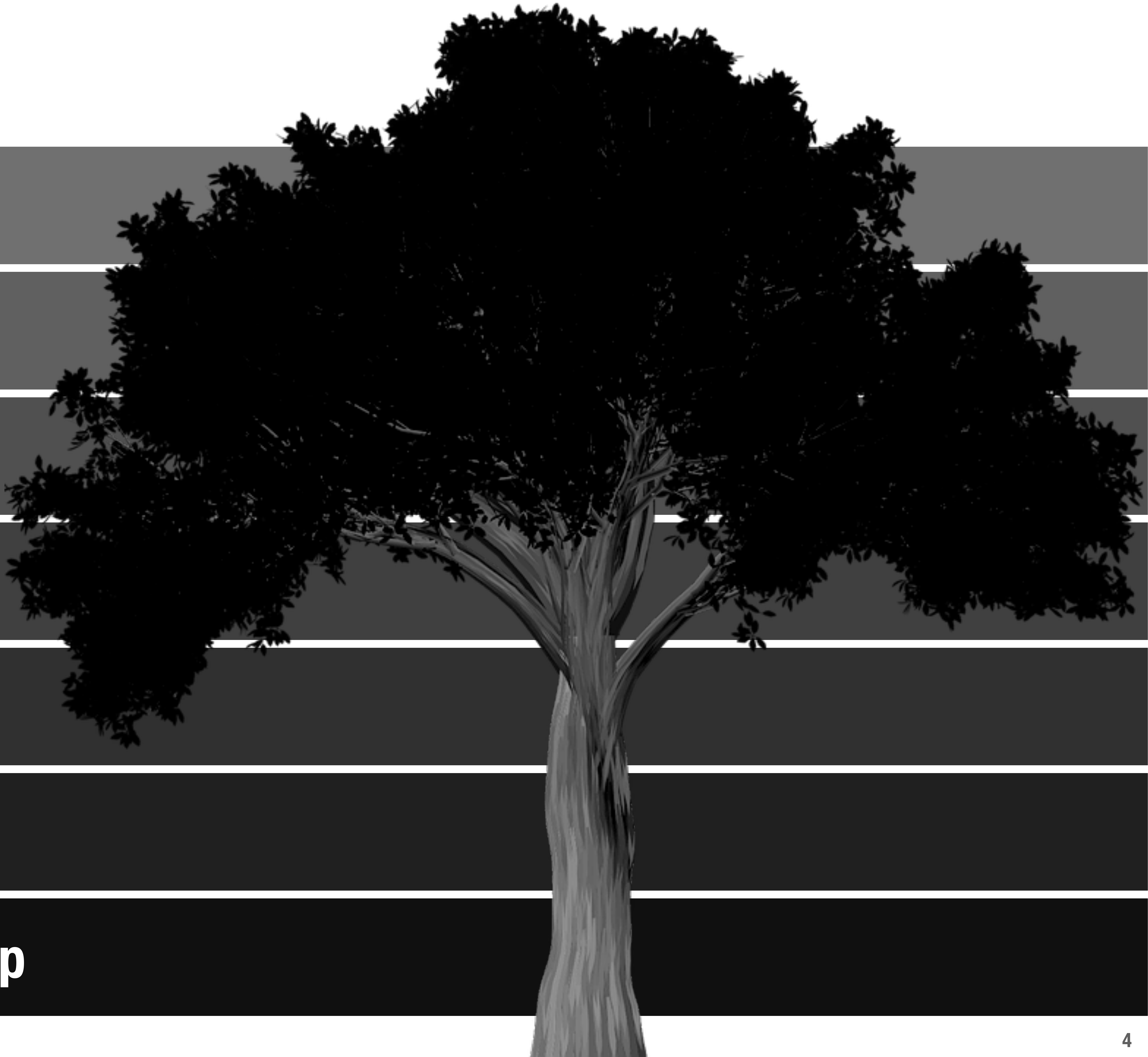


The conference app!

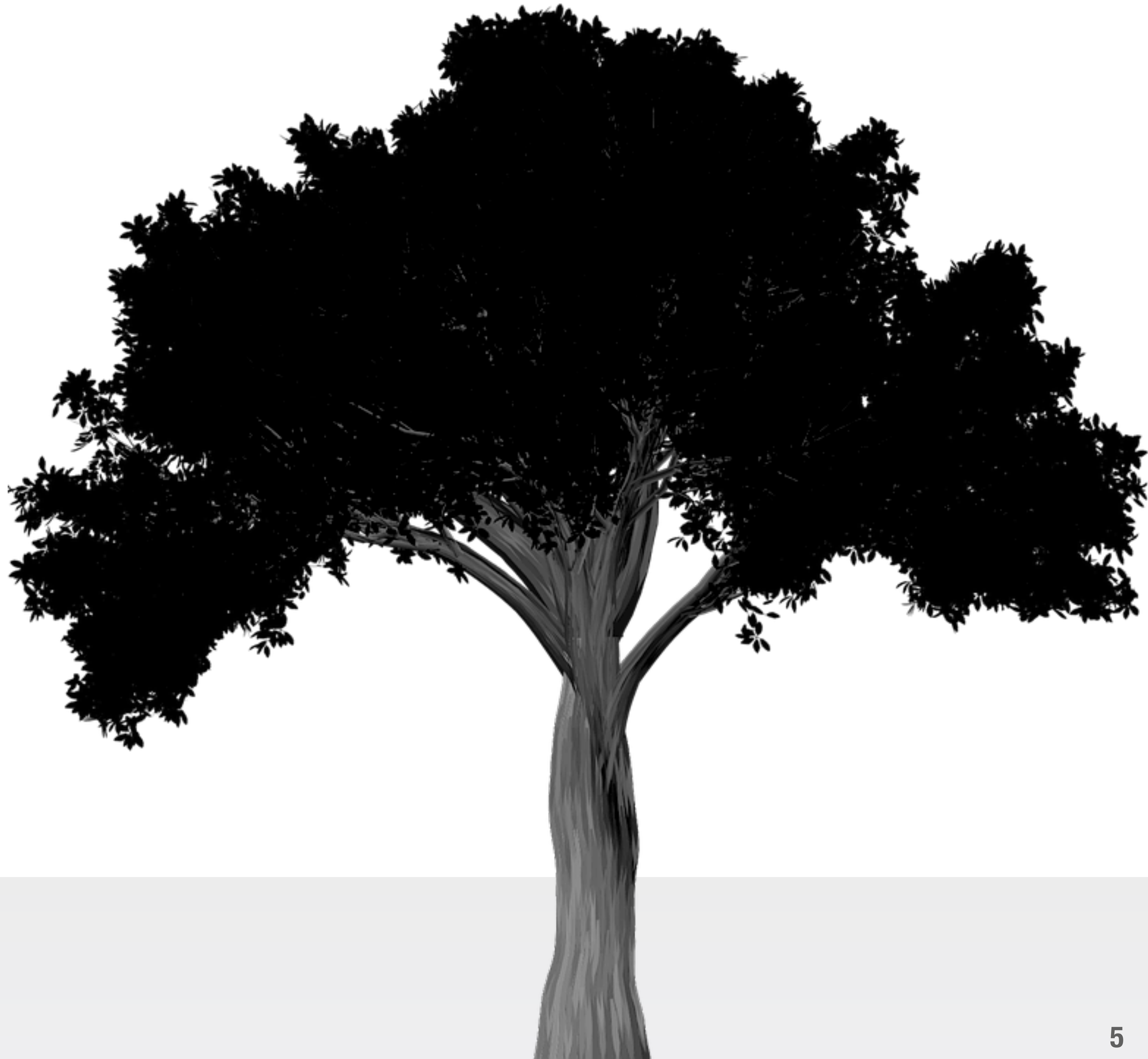
Live sixty-north.com/c/accu-2017/

Source github.com/abingham/accu-2017-elm-app

Agenda



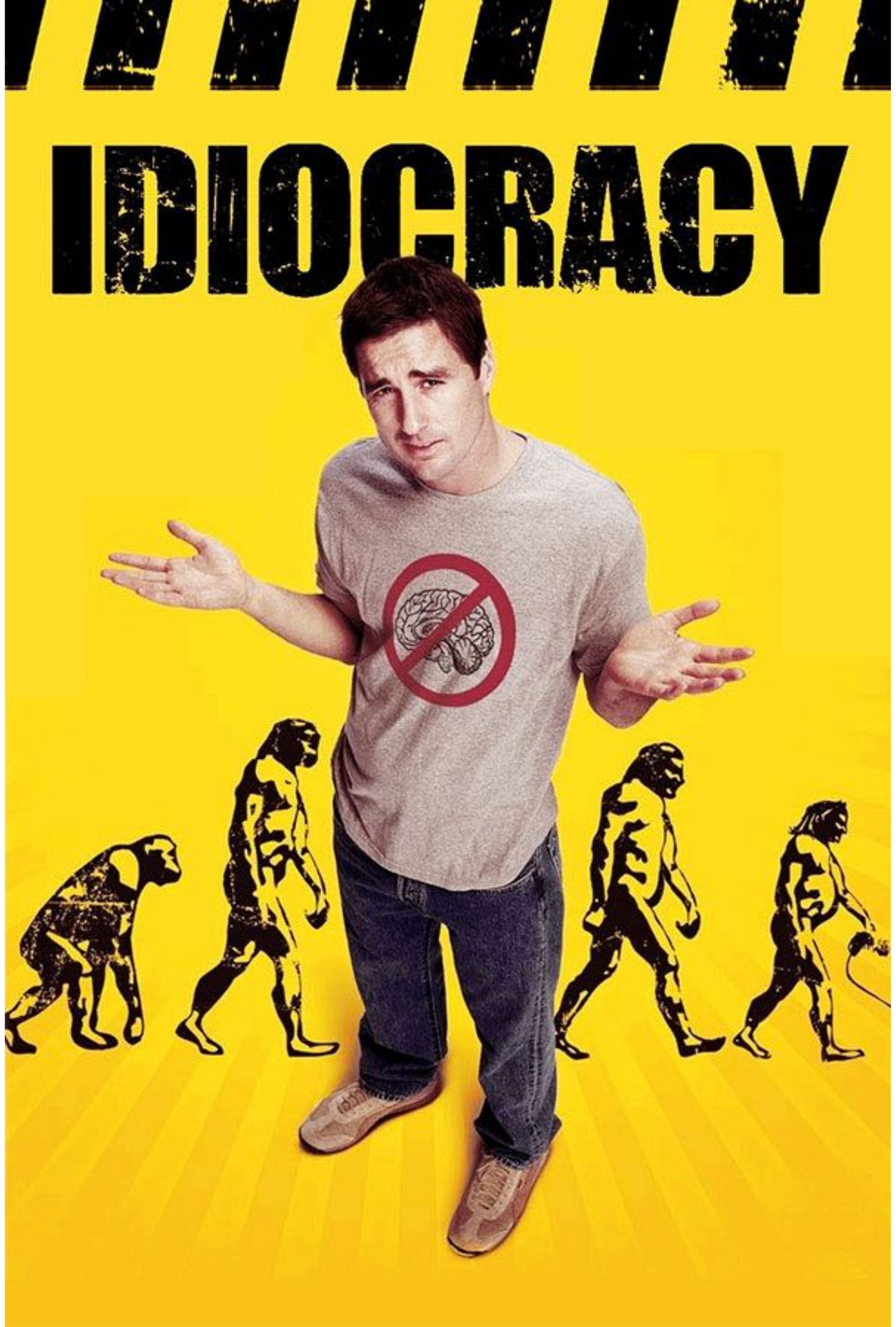
What is Elm?



What is wrong with JavaScript?

Common complaints

- Hopeless type system
- Bolt-on modularity
- “flavor of the week”
- Requires tests, but hard to test
- Unexpected runtime errors
- Not really one language



How does Elm help?

Elm directly addresses many JS deficiencies

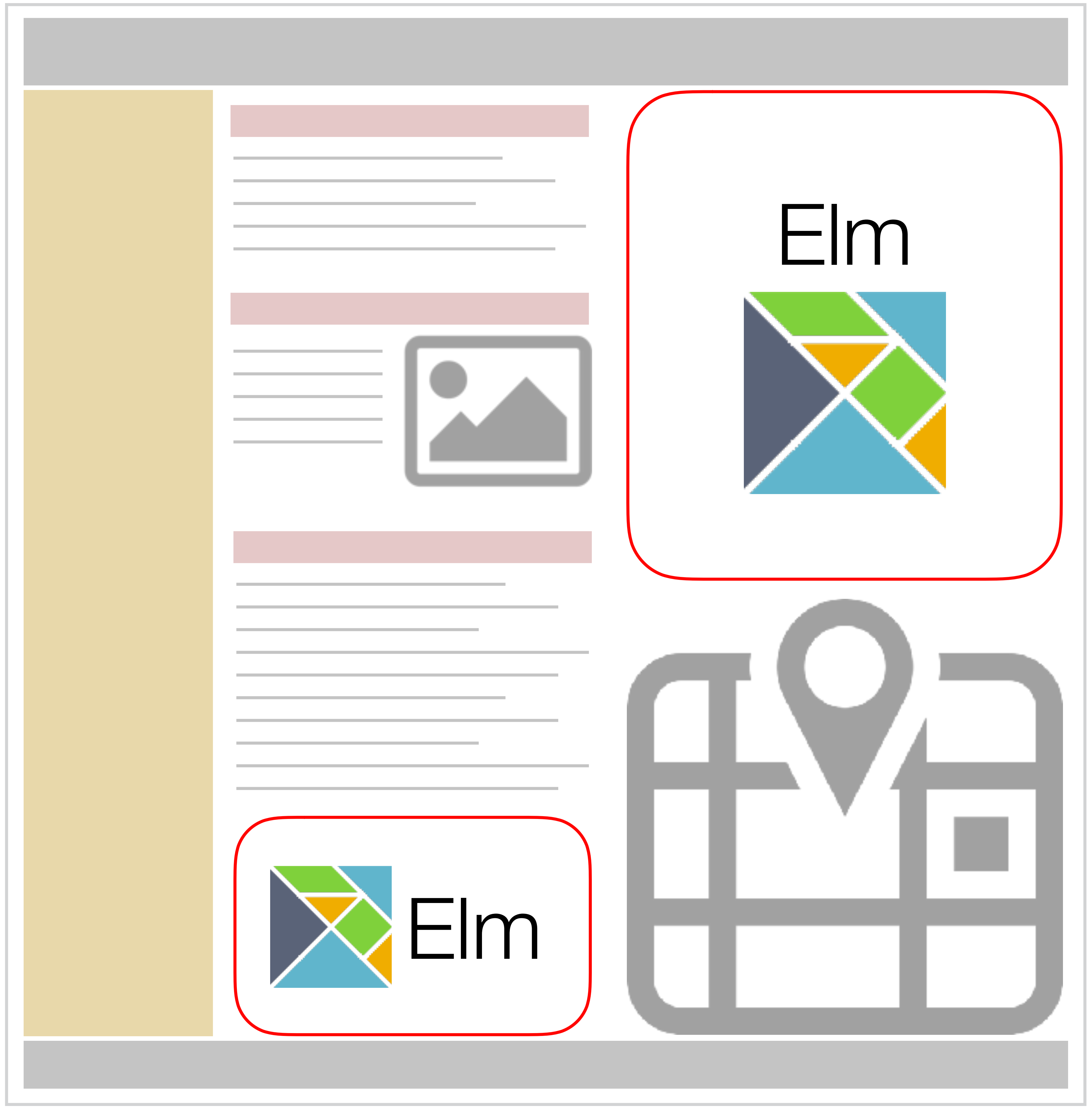
An exercise in intelligent tradeoffs

JavaScript		Elm
Hopeless type system	→	Haskell-inspired type system
Bolt-on modularity	→	Intuitive native module system
“Flavor of the week”	→	Prescribed patterns, “opinionated”
Requires tests, but hard to test	→	Far fewer tests needed
Unexpected runtime errors	→	“no runtime exceptions”
Not really one language	→	A single, BDFL-style language

Where does Elm fit in the world?

Elm is for developing web clients

The same places as React, Angular, Backbone, et al.



or

as part(s) of a larger page

controlling a full page

Who is behind Elm?

Elm is an open-source project

An active, fast-moving, and welcoming community

The result of Evan Czaplicki's senior thesis

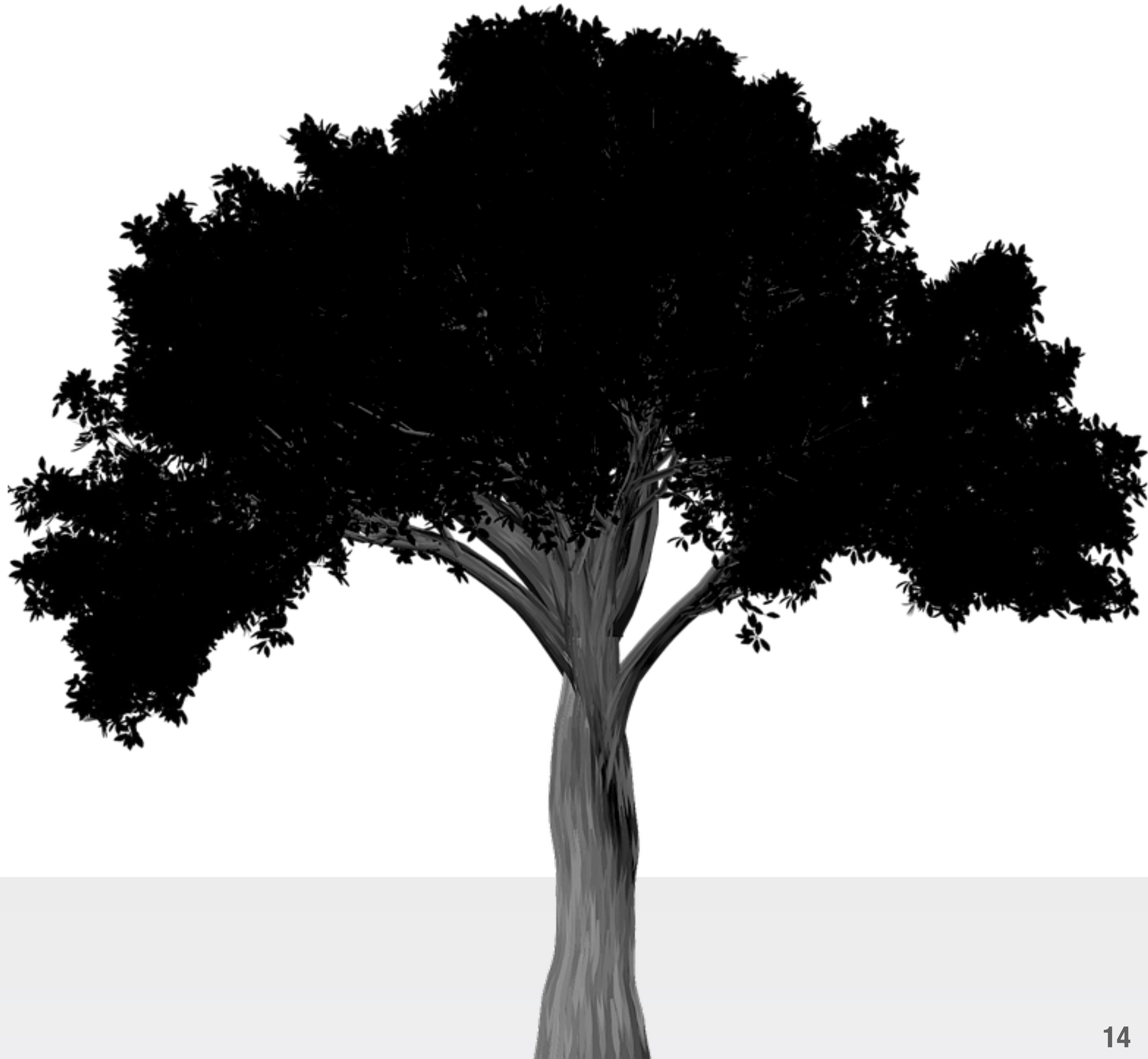
elm-lang.org

github.com/elm-lang/

groups.google.com/forum/#!forum/elm-discuss

Prezi

The Language



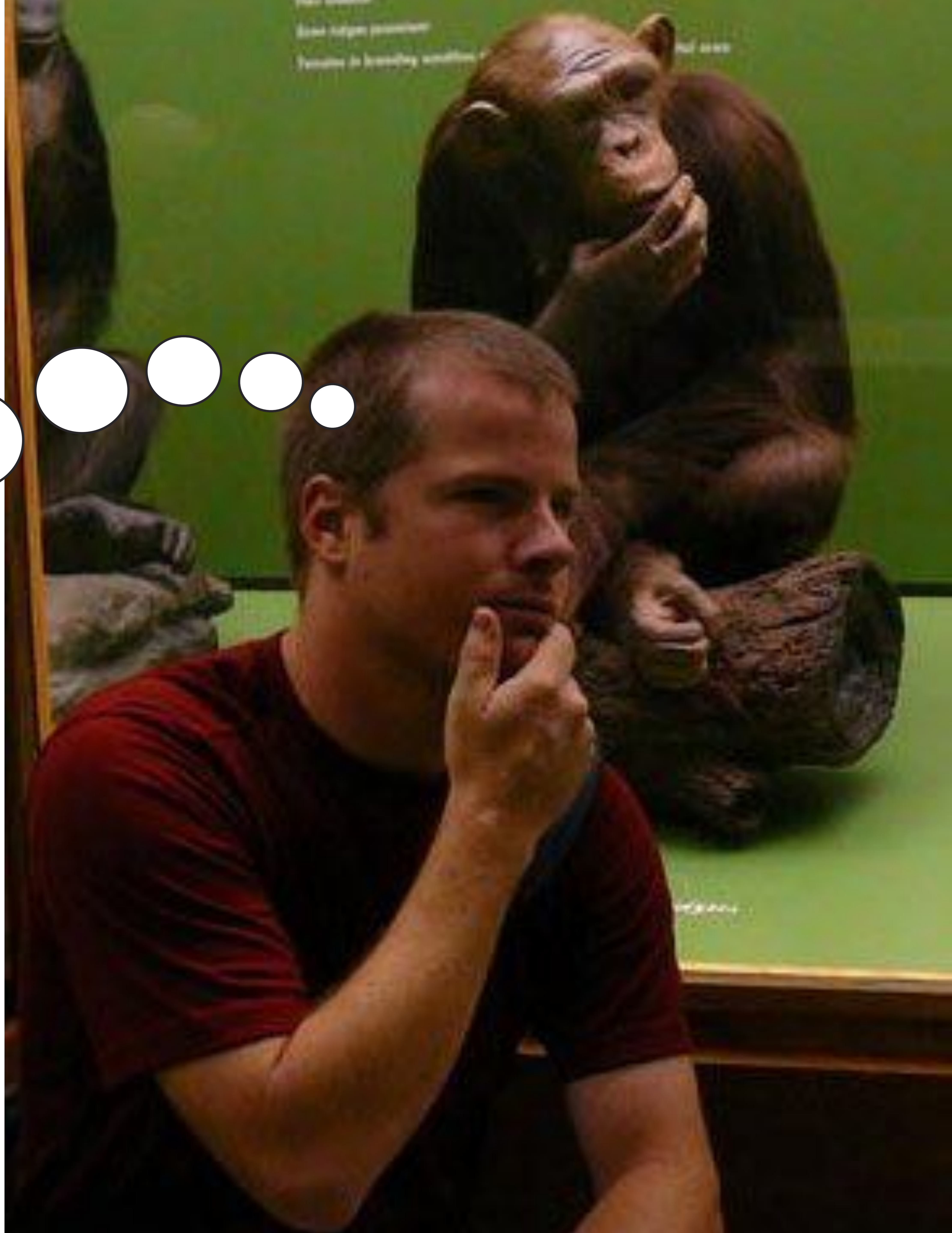
- Statically typed
- Compiled (to JavaScript)
- Immutable data structures
- Type inferencing
- Partial application, currying

HASKELL FOR WEB PAGES

Primitive types

Nothing surprising here

```
42 : Int

3.14 : Float

“Ulmaceae” : String

‘🌲’ : Char

True : Bool

False : Bool
```



Functions

Optional typing, partial application, composition…the works!

```
multiply : number -> number -> number
multiply x y =
    x * y

double : number -> number
double =
    multiply 2

quadruple : number -> number
quadruple =
    double >> double
```

Homogenous, iterable data types

Lists, arrays, set, and dictionaries

```
import Array exposing (..)
import Dict exposing (..)
import Set exposing (..)

list : List number
list = [1, 2, 3]

dict : Dict.Dict number String
dict = Dict.empty |> Dict.insert 42 "answer"

array : Array.Array number
array = Array.fromList [2, 3, 4]

set : Set.Set number
set = Set.fromList [1, 1, 2, 2, 3, 3]
```

key-value mapping

fast indexing

removes duplicates

Records and type aliases

The workhorse of Elm data modelling

```
type alias FullName =
    { firstName : String
    , lastName : String
    }


type alias User =
    { name : FullName
    , age : Int
    }

me : User
me = User (FullName "Austin" "Bingham") 42

getFirstName : User -> String
getFirstName = .name >> .firstName
```

use custom type

"constructors"

field accessors

Union types and pattern matching

A natural way to model weird (and not so weird) shaped data

```
type Shape
    -- A circle has a radius
    = Circle Float
      -- A square has an edge length
    | Square Float
      -- Regular polygons have a number
      -- of sides and an edge length
    | RegularPolygon Int Float
```

```
render : Shape -> String
render shape =
    case shape of
        Circle radius ->
            "circle"

        Square length ->
            "square"

        RegularPolygon sides length ->
            "regular polygon"
```

exhaustive

Modules and importing

Easily and intuitively split code into rational pieces

```elm
module Loader exposing (load)

load : String -> String
load filename =
   . . .
```

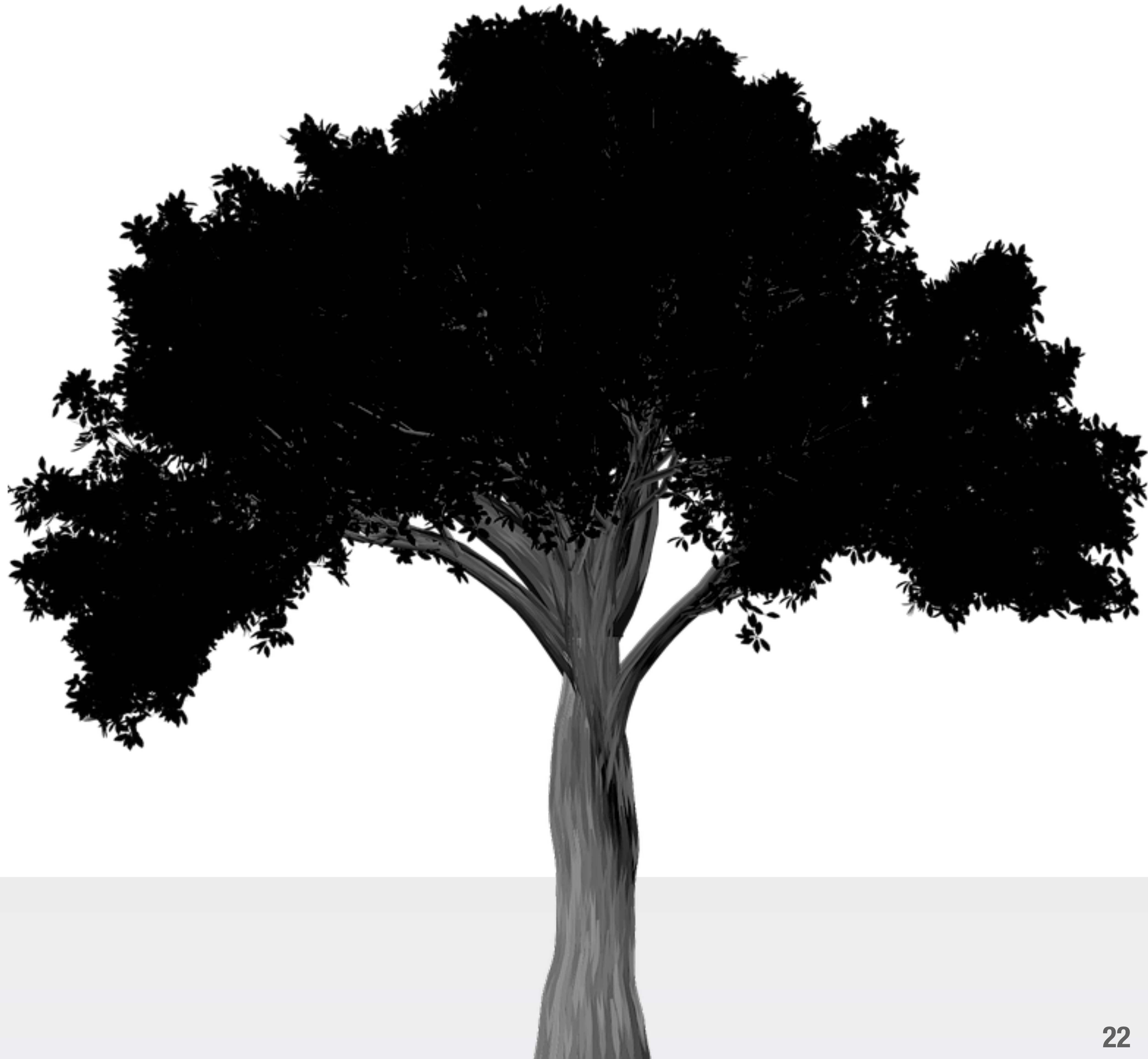
Loader.elm

```elm
import Loader

process = Loader.load "data.csv"
```

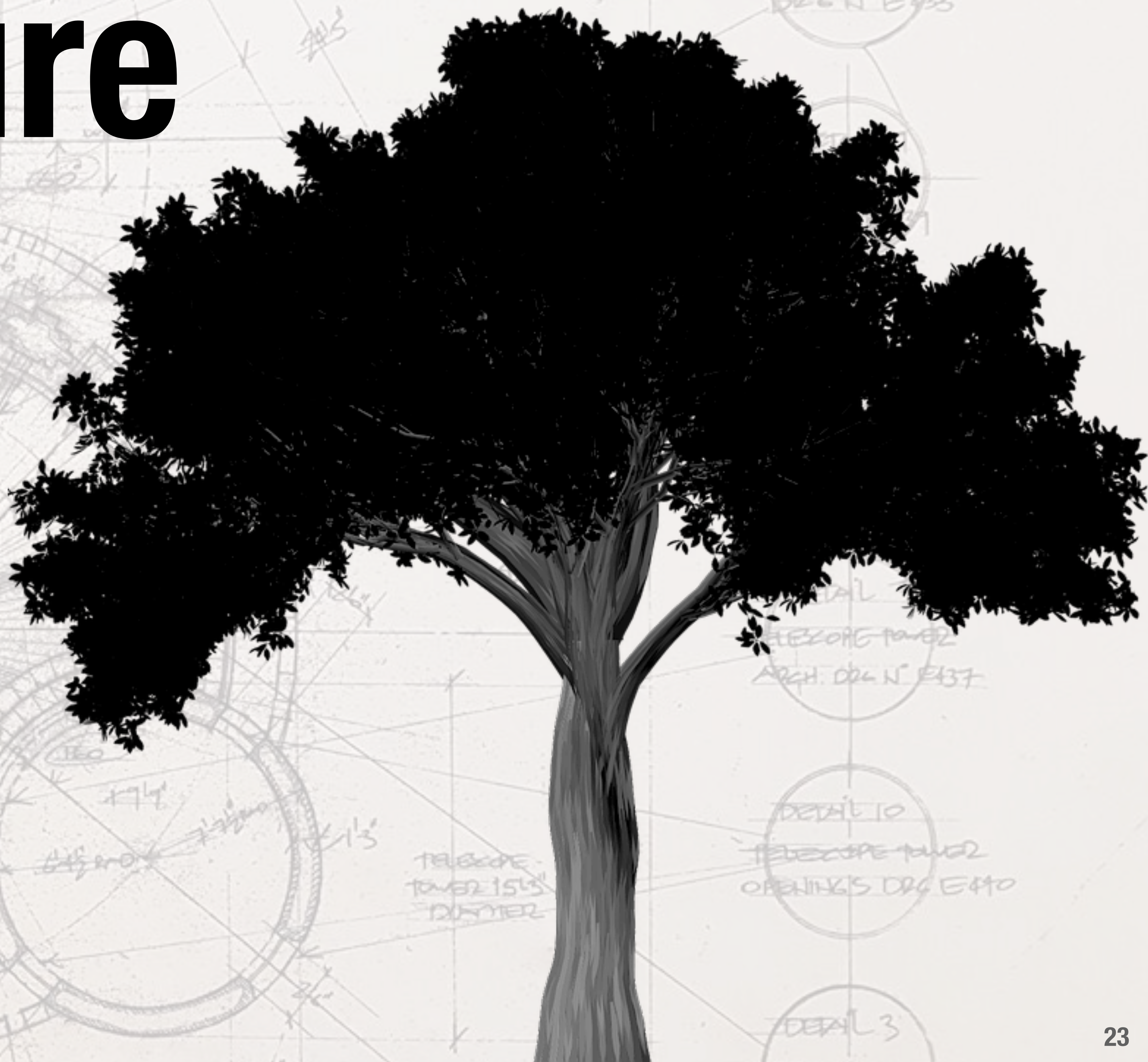
```elm
import Loader as L

process = L.load "data.csv"
```

```elm
import Loader exposing (load)

process = load "data.csv"
```

A Taste of Elm

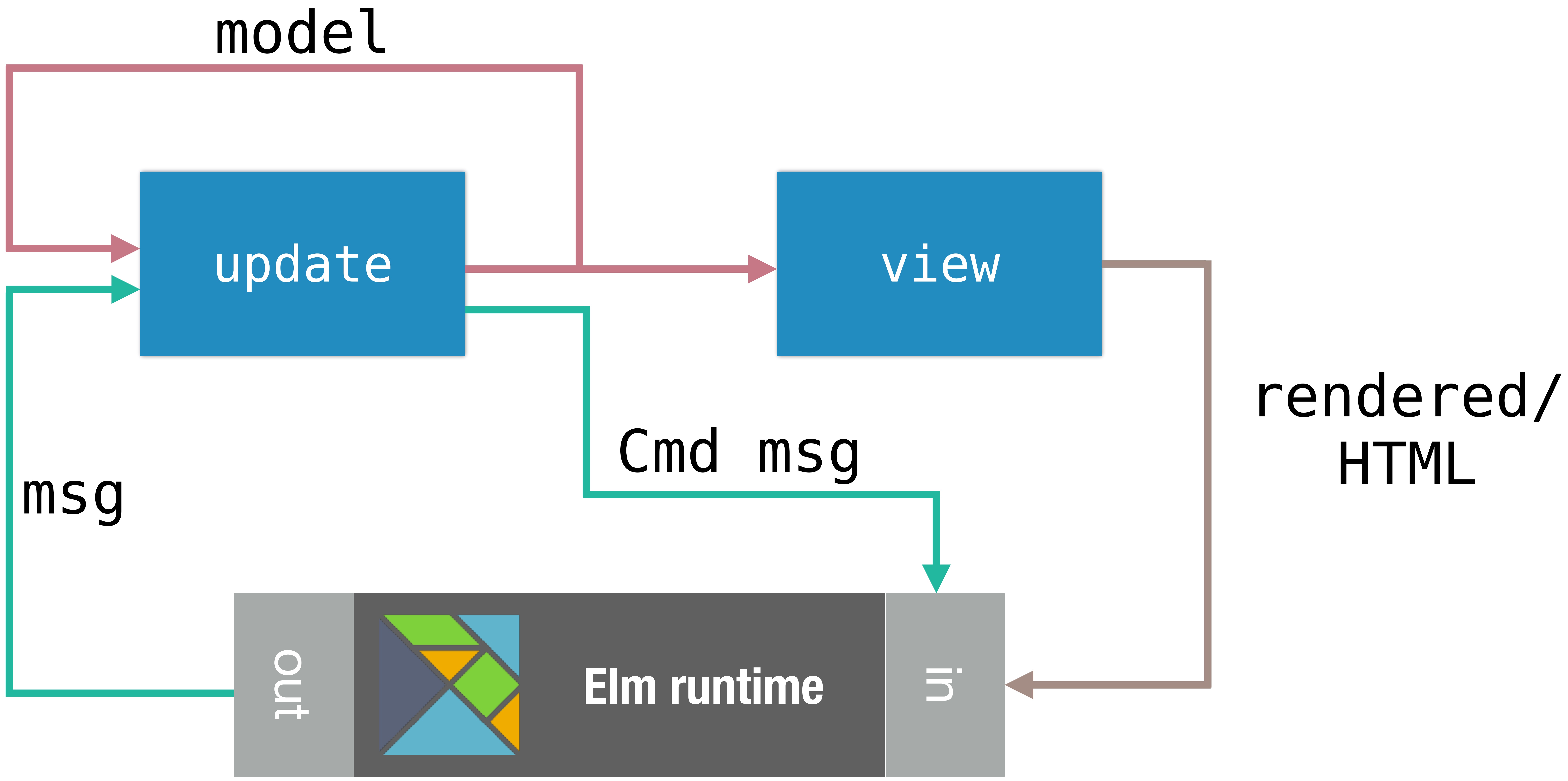


The Elm Architecture



The Elm Architecture

Model, messages, update, and view



The architecture in code...

...and the order in which I write them.

1. The shape of the data

```
-- MODEL
type alias Model = { id: Int, ... }
```

2. How it looks

```
-- VIEW
view : Model -> Html Msg
view =
  ...
```

```
-- MSG VOCABULARY
type Msg = Reset | ...
```

3. How it changes

```
-- UPDATE
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Reset -> ...
    ...
```

Some assembly required

Your `main` function is what ties these parts together

```
import Html
import Platform.Cmd
import Platform.Sub

main : Program Never Model Msg
main =
    Html.program
        { init = ( initialModel, Platform.Cmd.none )
        , view = view
        , update = update
        , subscriptions = \_ -> Platform.Sub.none
        }
```

standard type for main

flags

your model

your messages

starting model

initial "actions"

your view

your update

messages from JavaScript

What is “Program”?

Different factories for different applications

```
main : Program Never Model Msg
```

`Platform.program`	A headless program without a `view` function
`Html.program`	A program with an HTML `view`
`Navigation.program`	A program with an HTML `view` and which calls `update` when the route changes

Cmds vs. messages

Requesting work in JavaScript, and getting results

ask the runtime to "do stuff"
update
get notifications from the runtime
Msg
Html
Cmd
Elm Runtime

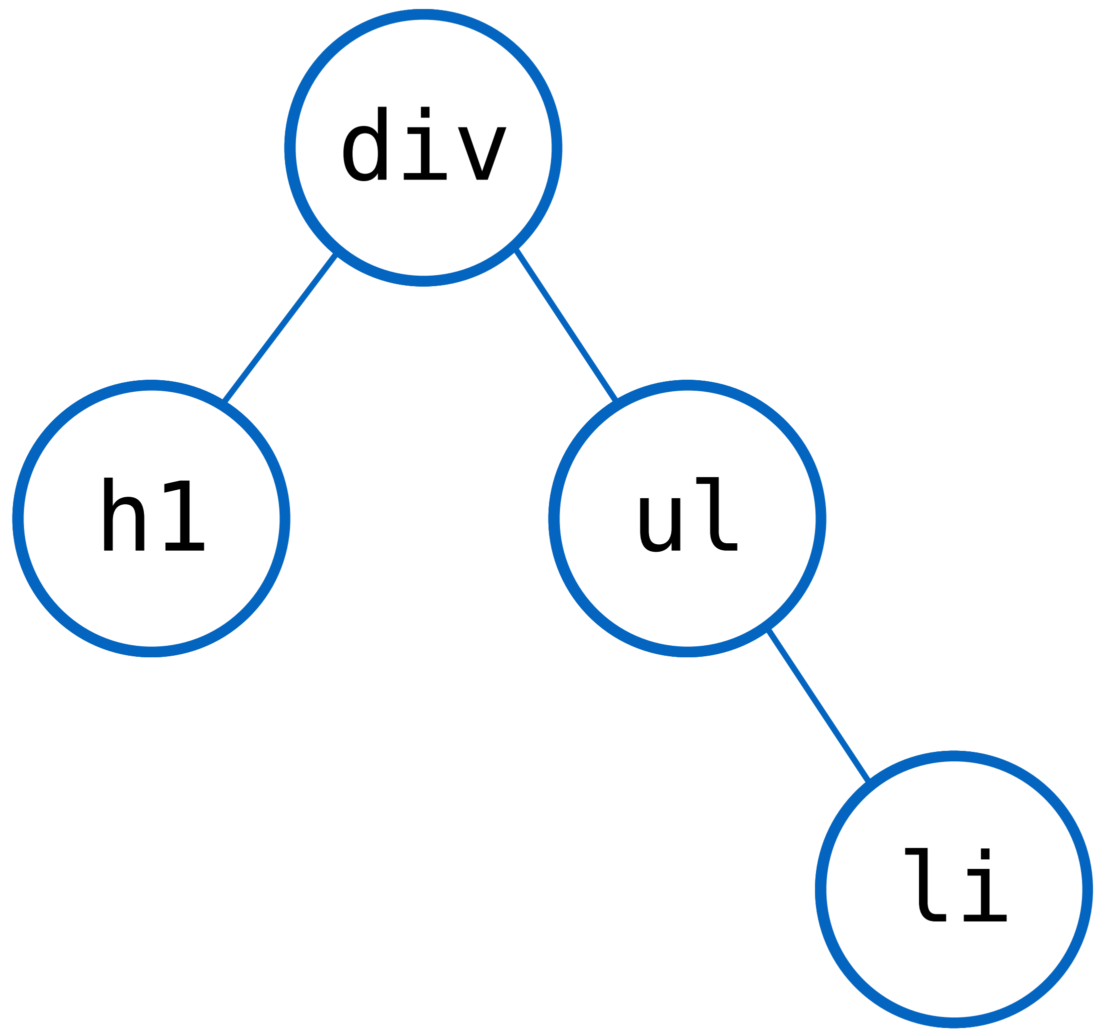
"We create data that *describes* what we want to do, and the Elm Runtime does the dirty work."
—elm-lang.org

Diagram from https://guide.elm-lang.org/architecture/effects/

Virtual DOM

The magic that makes all of this fast

Virtual	DOM

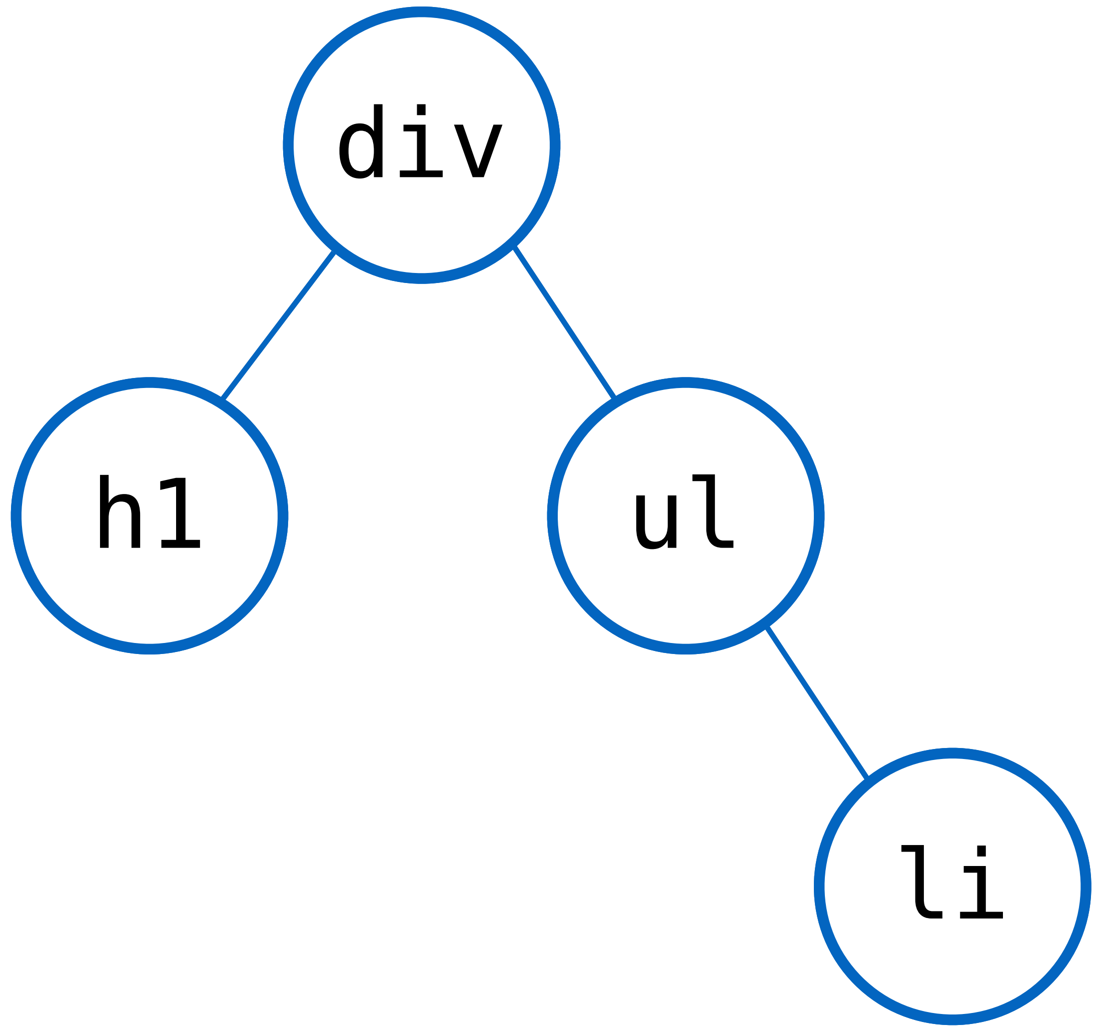


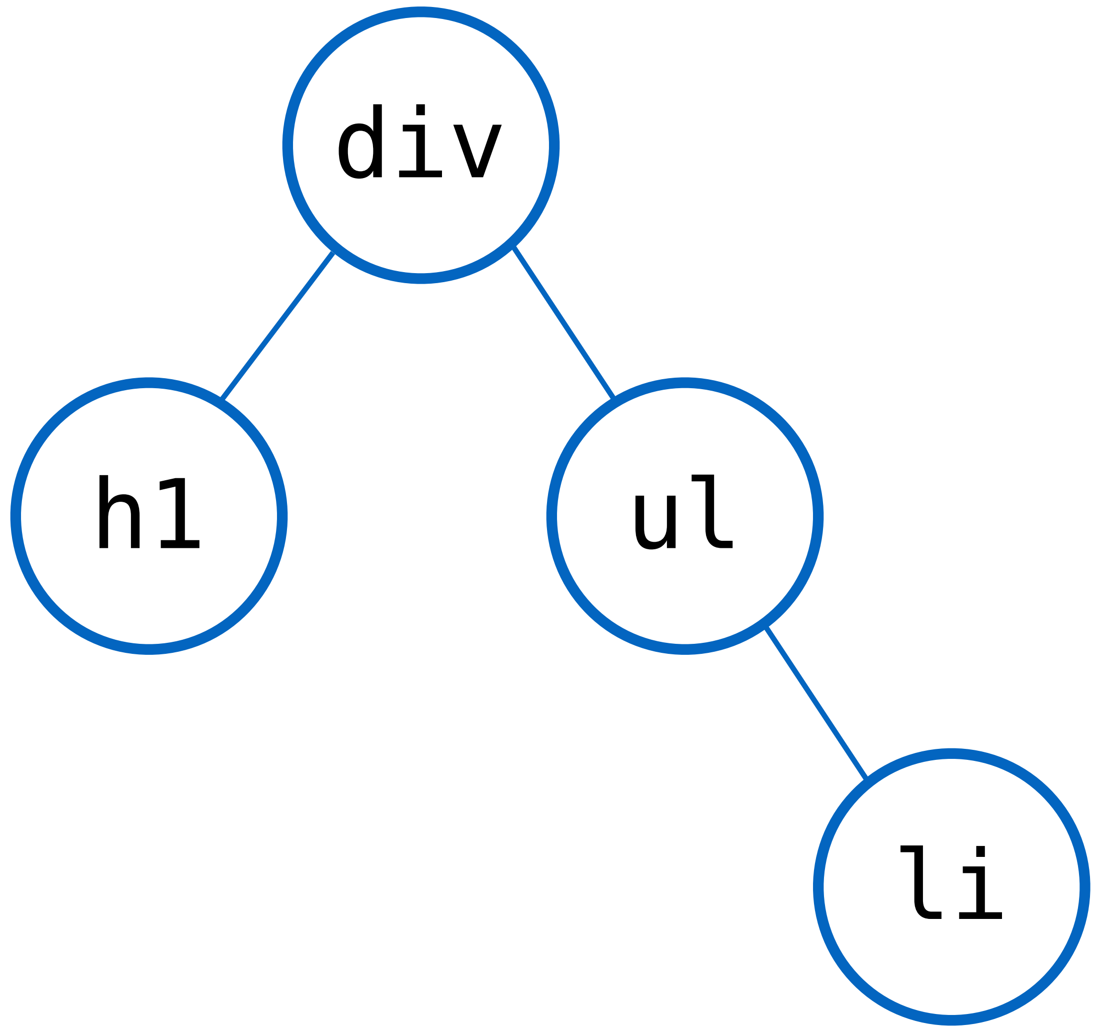
1 Build the virtual DOM

Virtual DOM

The magic that makes all of this fast

Virtual	DOM





2 Render the live DOM

Virtual DOM

The magic that makes all of this fast

Virtual
DOM
old
div
h1
ul
li
new
div
h1
ul
li
li
div
h1
ul
li

3 Render the new virtual DOM

Virtual DOM

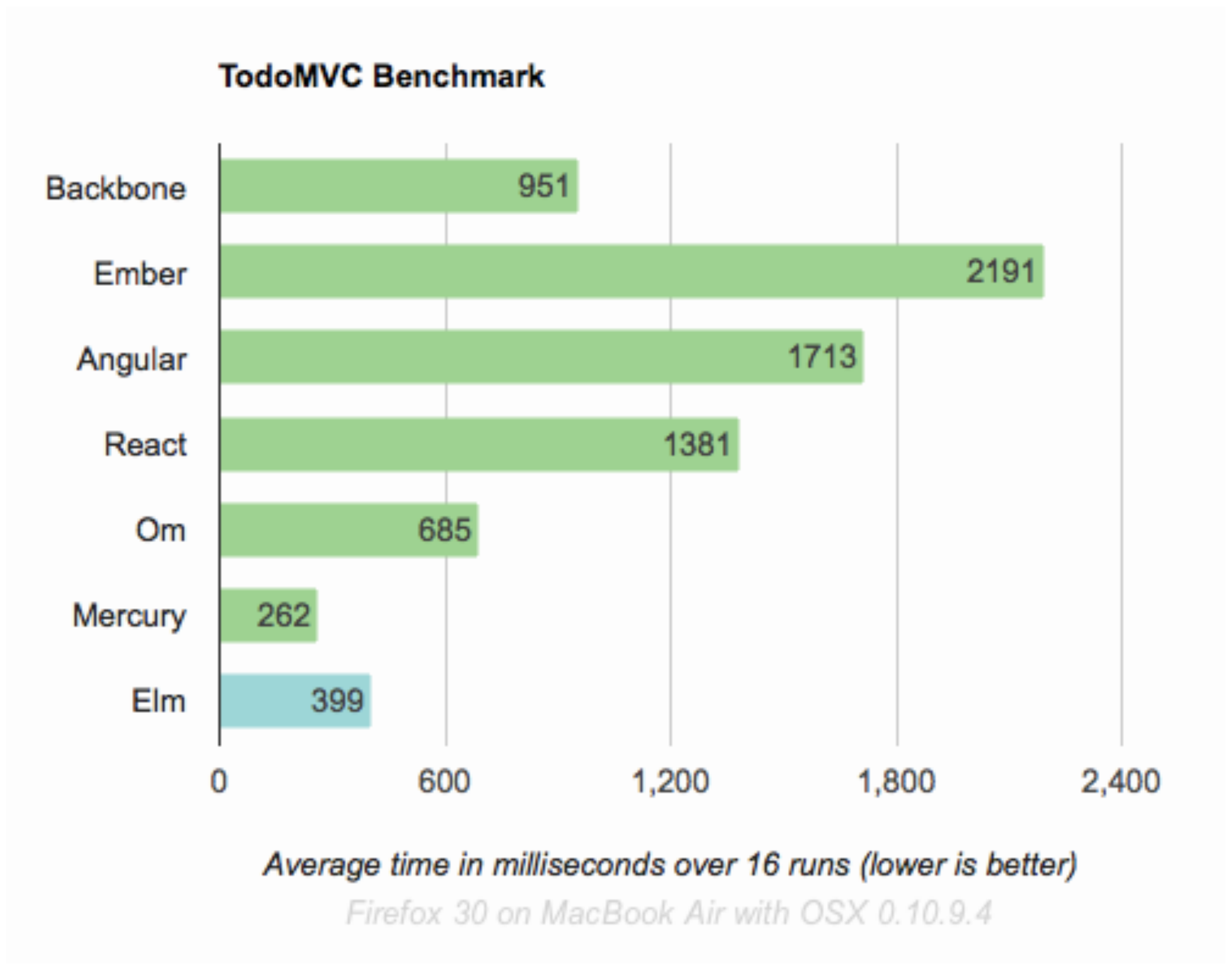
The magic that makes all of this fast

Virtual
DOM
old
div
h1
ul
li
new
div
h1
ul
li
li
div
h1
ul
li
li

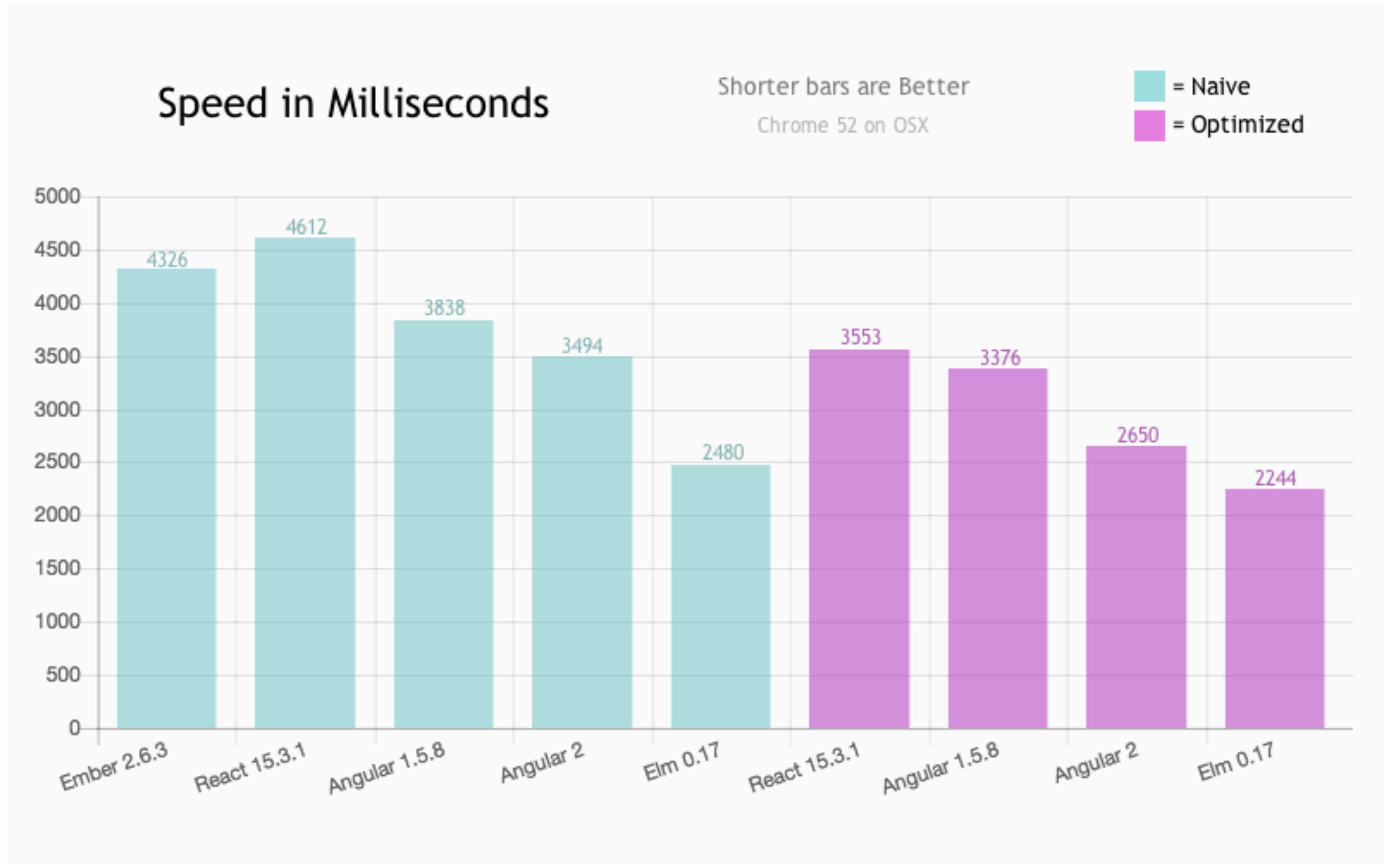
4 Update DOM based on the *diff*

Elm’s Virtual DOM is fast

And getting this speed is easier than with other frameworks

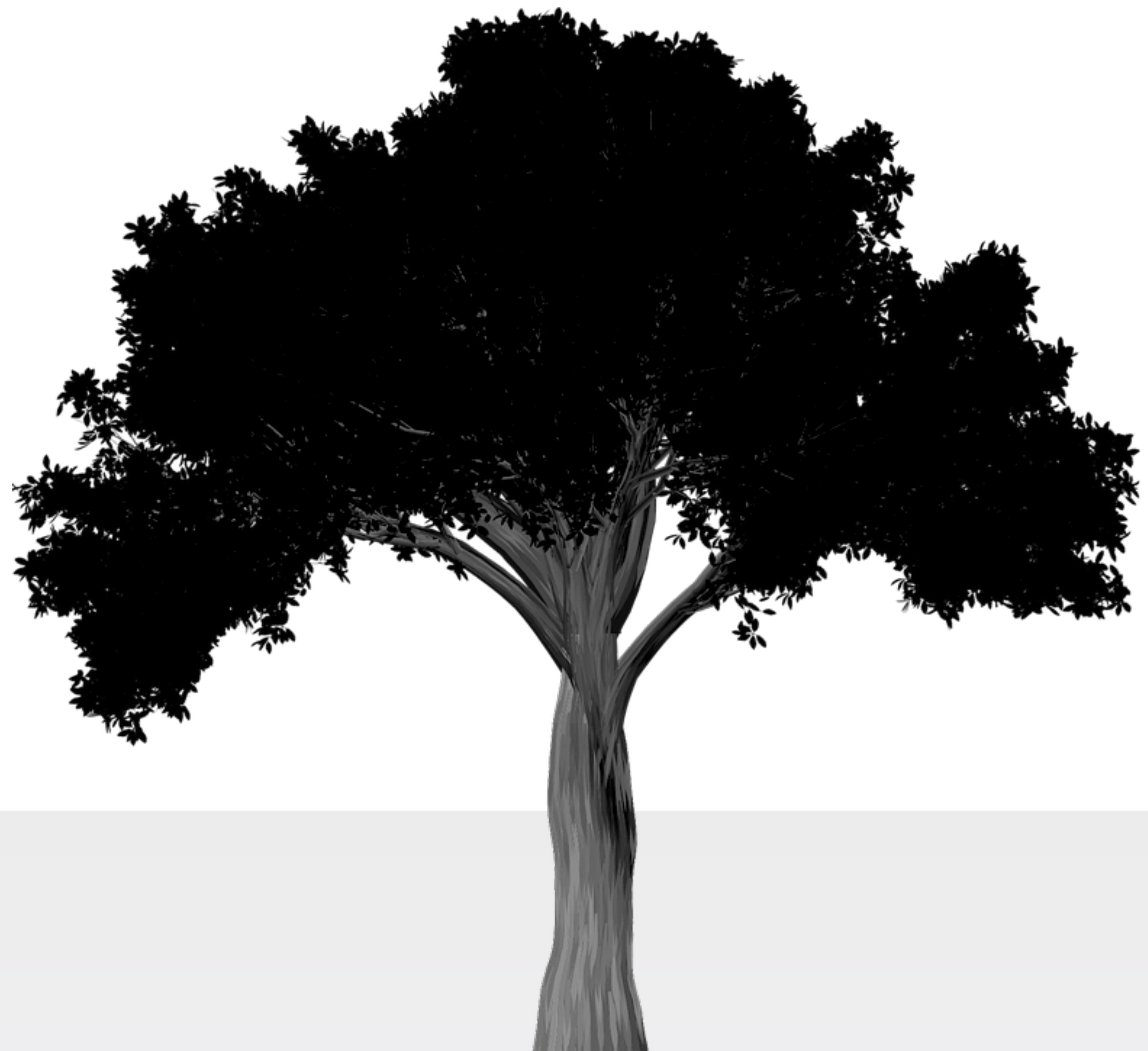


http://elm-lang.org/blog/blazing-fast-html

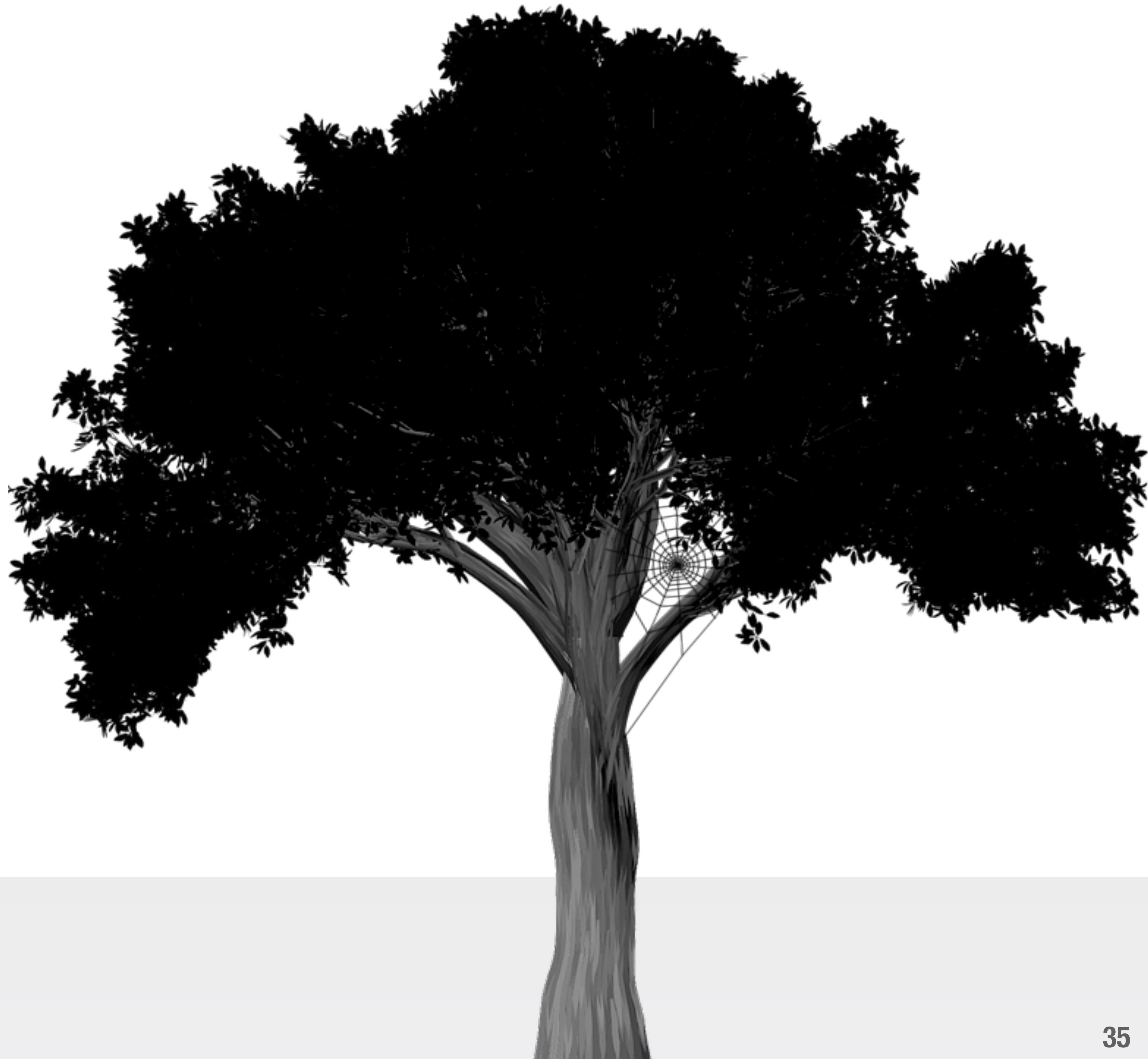


http://elm-lang.org/blog/blazing-fast-html-round-two

The Elm Architecture in Miniature



Talking to the web



The real world breaks sometimes

HTTP requests are an example of actions that might fail

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

everything worked

something went wrong

```
send : (Result Error a -> msg) -> Request a -> Cmd msg
```

translate a result to your vocabulary

Decoding JSON

This is a tricky spot for many new Elm programmers

Json.Decode.Pipeline

```
Json.Decode.Decoder a
```

1 Start with primitive decoders

```
int : Decoder Int
```

2 Combine them into more complex decoders

```
list int : Decoder (List Int)
```

3 Extract fields from JSON structures

```
field "prices" (list int) : Decoder (List Int)
```

4 Construct records from decoded values

```
map Stock (field "prices" (list int)) : Decoder Stock
```

Decoding JSON

This is a tricky spot for many new Elm programmers

Json.Decode.Pipeline

```
Json.Decode.Decoder a
```

1 Start with primitive decoders

```
int : Decoder Int
```

```
decodeString : Decoder a -> String -> Result String a
```

```
list int : Decoder (List Int)
```

3 Extract fields from JSON structures

```
field "prices" (list int) : Decoder (List Int)
```

4 Construct records from decoded values

```
map Stock (field "prices" (list int)) : Decoder Stock
```

Encoding JSON

Generally much simpler!

```
Json.Encode
```

1 Convert primitive objects to `Value` objects

```
int : Int -> Value
```

2 Convert aggregates of `Value`s into `Value`s

```
list : List Value -> Value
```

3 Encode `Value`s into strings

```
encode : Int -> Value -> String
```

Encoding JSON

Generally much simpler!

Json.Encode

1 Con

2 Con

3 Enc

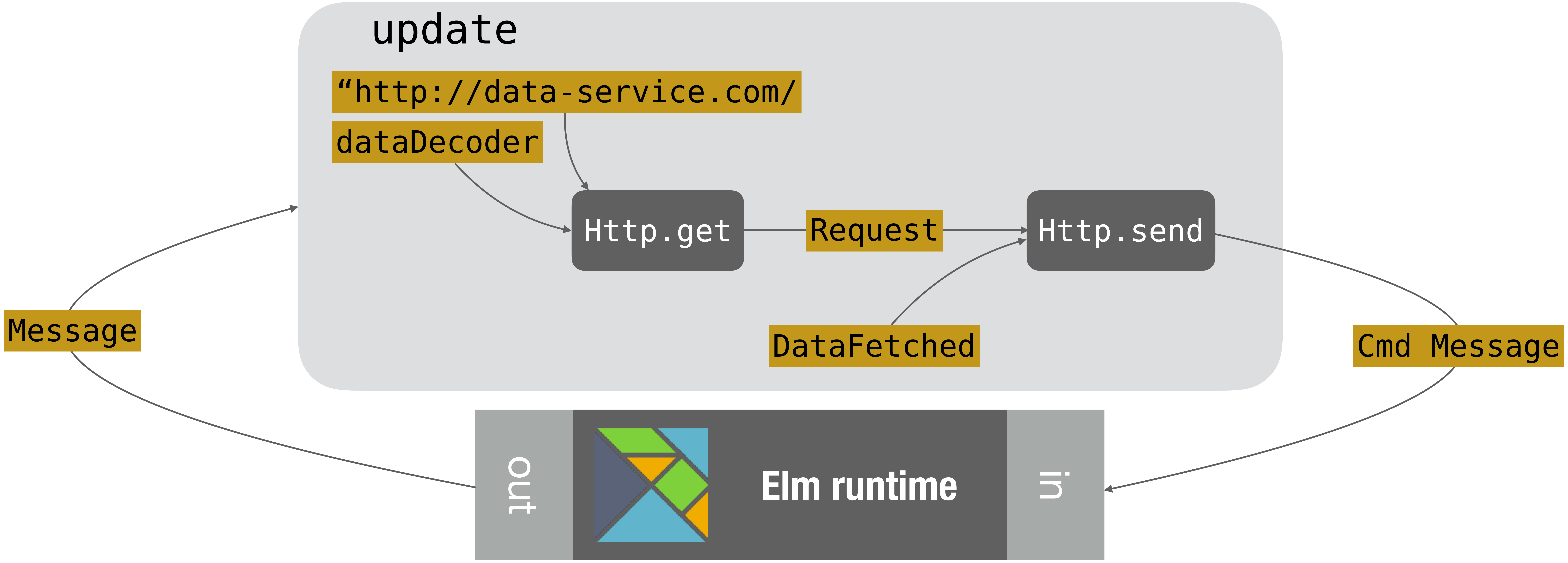
```
encodeInts : List Int -> String
encodeInts =
    List.map Json.Encode.int
        >> Json.Encode.list
        >> Json.Encode.encode 2
```

```
encode : Int -> Value -> String
```

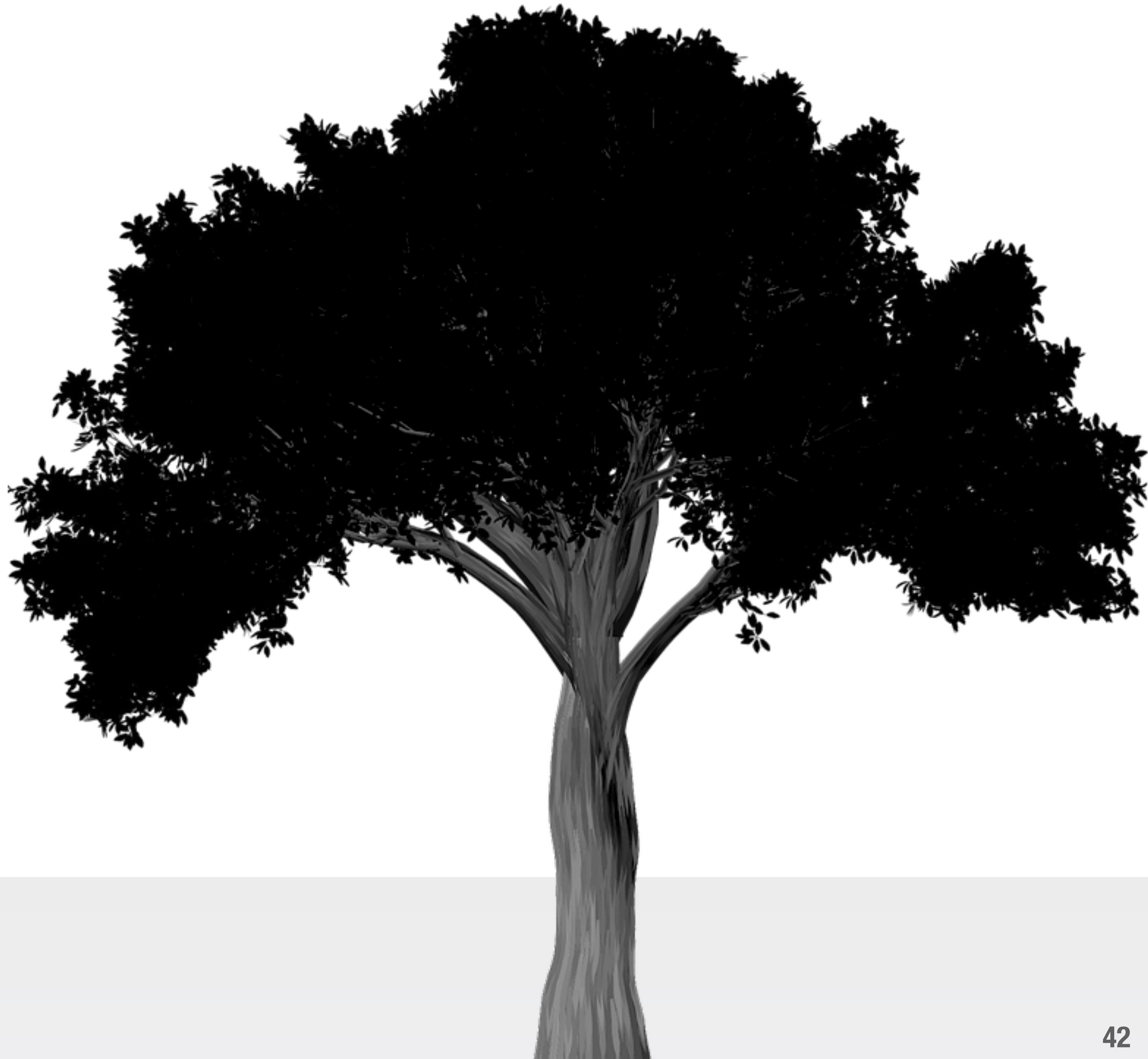
HTTP

Type-safe interaction with web servers and other services

```
type alias Data = { . . .}
type Message = DataFetched (Result Http.Error Data)
dataDecoder : Json.Decode.Decoder Data
```



JavaScript Interop



Flags

Passing data to your app at startup

Javascript

```javascript
var flags = {
    debug = True;
    connectionString =
        'sqlite://test.db';
};

var Elm = . . . ;
var mountNode =
    document.getElementById('main');

Elm.MyApp.embed(mountNode, flags);
```

Elm

```elm
type alias Flags =
    { debug : Bool
    , connectionString : String
    }


main : Html.Program Flags Model Msg
main =
    Html.programWithFlags
        { init =
            \flags -> . . .
        , view = view
        , update = update
        , subscriptions = . . .
        }
```

Subscriptions

Sources of messages that you can listen to

subscriptions must result in our message type

specific message constructor

```
type Msg = NewMessage String

subscriptions : Model -> Sub Msg
subscriptions model =
  WebSocket.listen "ws://echo.websocket.org" NewMessage

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

subscribe in main

Ports

Define “tunnels” for sending and receiving data to and from Javascript

Elm

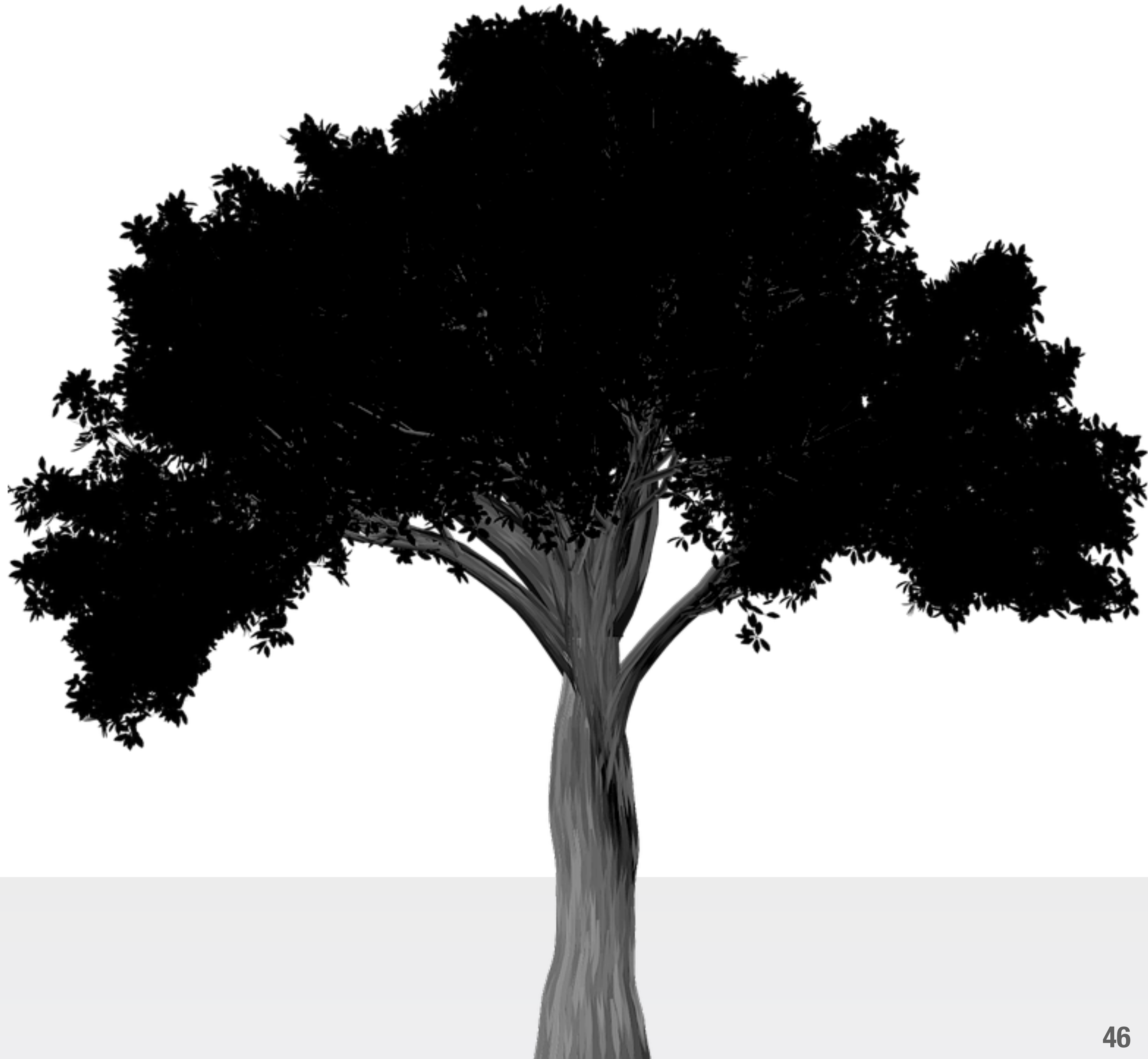
```elm
port module Spelling exposing (..)

-- port for sending strings out to JavaScript
port check : String -> Cmd msg

-- port for listening for suggestions from JavaScript
port suggestions : (List String -> msg) -> Sub msg
```

Javascript

```javascript
var app = Elm.Spelling.fullscreen();

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

Example taken from https://guide.elm-lang.org/interop/javascript.html

A Tour of the ACCU 2017 App



ACCU APP

ACCU APP

ACCU APP
Main.elm
Oak Spring

ACCU APP

Model.elm

ACCU APP
Msg.elm
Oak Spring

Update.elm

Comms.elm

ACCU APP
Oak Spring
Json.elm

ACCU APP

Routing.elm

Search.elm

ACCU APP
Storage.elm
Oak Spring

ACCU APP
Asciidoc.elm
Oak Spring

ACCU APP

View.elm

Caveat Elm-ptor

Elm has a lot of potential, but you need to be aware of its rough edges

- **Language and core are evolving**
- **Best practices are far from settled**
- **Bus factor and mind share**
- **Tooling has room for improvement**
- **It’s just different**

Links

Presentation code and examples

`github.com/abingham/elm-presentation-material`

Jupyter kernel for Elm

`github.com/abingham/jupyter-elm-kernel`

ACCU Schedule app

`github.com/abingham/accu-2017-elm-app`

Don't forget the put a (green) card in a box!

Thank you!

Austin Bingham
@austin_bingham

SixtyNORTH

@sixty_north