Elm Architecture
Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

Spring 2017

Outline

Elm language

Elm Language

- Selections
- Iterations
- Sequences

Union types

HTTP in Elm

Architecture

- The Monad
- The Model
- The View
- And Updates

REST assignment

Values

- Strings
  - `"Hello"`
  - `"Hello"++" "++"World!"` is `"Hello World!"`
- Numbers
  - `7`
  - `22.67`
  - `2 + 3 * 4` is `14`
  - `9/2` is `4.5`
  - `9//2` is `4`

Functions

```
> isNegative n = n < 0
<function>

> isNegative 4
False

> isNegative -7
True

> isNegative (-3 * -4)
False
```

Selections

```
> if True then "hello" else "world"
"hello"

> if False then "hello" else "world"
"world"
```

Lists

```
> names = [ "Alice", "Bob", "Chuck" ]
["Alice","Bob","Chuck"]

> List.isEmpty names
False

> List.length names
3

> List.reverse names
["Chuck","Bob","Alice"]
```

Lists

```
> numbers = [1,4,3,2]
[1,4,3,2]

> List.sort numbers
[1,2,3,4]

> double n = n * 2
<function>

> List.map double numbers
[2,8,6,4]
```

Tuples

```
> import String

> goodName name = \
|   if String.length name <= 20 then \
|     (True, "name␣accepted!") \
|   else \
|     (False, "name␣was␣too␣long")

> goodName "Tom"
(True, "name␣accepted!")
```

Records

```
> point = { x = 3, y = 4 }
{ x = 3, y = 4 }

> point.x
3

> bill = { name = "Gates", age = 57 }
{ age = 57, name = "Gates" }

> bill.name
"Gates"
```

Records

```
> .name bill
"Gates"

> List.map .name [bill,bill,bill]
["Gates","Gates","Gates"]

> { bill | name = "Nye" }
{ age = 57, name = "Nye" }

> { bill | age = 22 }
{ age = 22, name = "Gates" }
```

Records

```
> under70 {age} = age < 70
<function>

> under70 bill
True

> under70 { species = "Triceratops", age = 68000000 }
False
```

Primitives and Lists

```
> "hello"
"hello" : String

> not True
False : Bool

> round 3.1415
3 : Int
```

```
> [ "Alice", "Bob" ]
[ "Alice", "Bob" ] : List String

> [ 1.0, 8.6, 42.1 ]
[ 1.0, 8.6, 42.1 ] : List Float

> []
[] : List a
```

Functions

```
> import String
> String.length
<function> : String -> Int

> String.length "Supercalifragilisticexpialidocious"
34 : Int

> String.length [1,2,3]
-- error!

> String.length True
-- error!
```

Lambdas - anonymous functions

```
> \n -> n / 2
<function> : Float -> Float

> (\n -> n / 2) 128
64 : Float

> oneHundredAndTwentyEight = 128.0
128 : Float

> half = \n -> n / 2
<function> : Float -> Float

> half oneHundredAndTwentyEight
64 : Float

> half n = n / 2
<function> : Float -> Float
```

Functions with more arguments

```
> divide x y = x / y
<function> : Float -> Float -> Float

> divide 3 2
1.5 : Float

> divide x = \y -> x / y
<function> : Float -> Float -> Float

> divide = \x -> (\y -> x / y)
<function> : Float -> Float -> Float
```

Functions - again

```
divide 3 2
```

Functions - again

```
divide 3 2
```

```
(divide 3) 2                   -- 1: Implicit parentheses
```

Functions - again

```
divide 3 2
```

```
(divide 3) 2                -- 1: Implicit parentheses
```

```
((\x -> (\y -> x / y)) 3) 2 -- 2: Expand ’divide’
```

Functions - again

```
divide 3 2
```

```
(divide 3) 2                -- 1: Implicit parentheses
```

```
((\x -> (\y -> x / y)) 3) 2 -- 2: Expand 'divide'
```

```
(\y -> 3 / y) 2             -- 3: Replace x with 3
```

Functions - again

```
divide 3 2
```

```
(divide 3) 2                            -- 1: Implicit parentheses
```

```
((\x -> (\y -> x / y)) 3) 2 -- 2: Expand ’divide’
```

```
(\y -> 3 / y) 2                         -- 3: Replace x with 3
```

```
3 / 2                                   -- 4: Replace y with 2
```

Functions - again

```
divide 3 2
```

```
(divide 3) 2                -- 1: Implicit parentheses
```

```
((\x -> (\y -> x / y)) 3) 2 -- 2: Expand ’divide’
```

```
(\y -> 3 / y) 2             -- 3: Replace x with 3
```

```
3 / 2                       -- 4: Replace y with 2
```

```
1.5                         -- 5: Do the math
```

Type annotations

```elm
half : Float -> Float
half n =
  n / 2

divide : Float -> Float -> Float
divide x y =
  x / y

askVegeta : Int -> String
askVegeta powerLevel =
  if powerLevel > 9000 then
    "It's␣over␣9000!!!"
  else
    "It␣is␣" ++ toString powerLevel ++ "."
```

If statements

```
 > if True then "hello" else "world"
```

```
if 10 == 10 then "hi"
```

If statements

```
> if True then "hello" else "world"
```

`if 10 == 10 then "hi"` **Won't compile!**

If statements

```
> if True then "hello" else "world"
```

if 10 == 10 then "hi" **Won't compile!** if 10 == 10 then "hi" else "ho"

If statements

```
> if True then "hello" else "world"
```

`if 10 == 10 then "hi"` **Won’t compile!** `if 10 == 10 then "hi" else "ho"`

Remember that **everything must have a return type**!

case matches

```
case n of
  0 -> "Zero"
  1 -> "One"
  _ -> "Moar"
```

case matches

```
case n of
  0 -> "Zero"
  1 -> "One"
  _ -> "Moar"
```

```
case n of
  0 -> "Zero"
  1 -> "One"
  2 -> "Two"
```

case matches

```
case n of
  0 -> "Zero"
  1 -> "One"
  _ -> "Moar"
```

```
case n of
  0 -> "Zero"
  1 -> "One"
  2 -> "Two"
```

Error!: this ‘case‘ does not have branches for all possibilities.

case matches

```
case n of
  0 -> "Zero"
  1 -> "One"
  _ -> "Moar"
```

```
case n of
  0 -> "Zero"
  1 -> "One"
  2 -> "Two"
```

Error!: `this ‘case‘ does not have branches for all possibilities.` Because everything has a fixed type, we *know* whether we will match everything!

This is seriously **cool**!

Debugging

How do you normally debug in Java?

Debugging

How do you normally debug in Java?
What is the problem with that in Elm?

Debugging

How do you normally debug in Java?
What is the problem with that in Elm?
Everything needs a return type. Also debugging/println/logging.

Debugging

How do you normally debug in Java?
What is the problem with that in Elm?
Everything needs a return type. Also debugging/println/logging.
Solution is to encapsulate the side-effect:

```
    import Debug

    if (Debug.log "A is" a) == 10 then "hi" else "ho" -- "A
```

Classic language building blocks

In LISP we still had the classic programming constructs. But Elm is pure functional, so what then?

Sequence
?

Selection (control-logic)
?

Iteration
?

Classic language building blocks

In LISP we still had the classic programming constructs. But Elm is pure functional, so what then?

Sequence
?

Selection (control-logic)
Easy: Expressions instead of statements

Iteration
?

Classic language building blocks

In LISP we still had the classic programming constructs. But Elm is pure functional, so what then?

Sequence	
	?
Selection	(control-logic)
	Easy: Expressions instead of statements
Iteration	
	Medium: Recursion!

Classic language building blocks

In LISP we still had the classic programming constructs. But Elm is pure functional, so what then?

Sequence

?

Selection (control-logic)

Easy: Expressions instead of statements

Iteration

Medium: Recursion!

Classic language building blocks

In LISP we still had the classic programming constructs. But Elm is pure functional, so what then?

Sequence

Hard: Monads

Selection (control-logic)

Easy: Expressions instead of statements

Iteration

Medium: Recursion!

Destructuring - Pattern matching

- In functions calls
- In `case`-`of` constructs
- In `let`-`in` constructs

Destructuring - Pattern matching

- □ In functions calls
- □ In case-of constructs
- □ In let-in constructs

```
p = (3.5, 4.2)
l = [7, 9, 13],
n = { name = "Kurt", age = 34 }:
```

Destructuring - Pattern matching

- □ In functions calls
- □ In `case`-`of` constructs
- □ In `let`-`in` constructs

```
p = (3.5, 4.2)
l = [7, 9, 13],
n = { name = "Kurt", age = 34 }:
```

```
manhattan point =
  let
    (x, y) = point
  in
    x + y
```

```
manhattan p -- 7.7
```

Destructuring - Pattern matching

- □ In functions calls
- □ In `case`-`of` constructs
- □ In `let`-`in` constructs

```
p = (3.5, 4.2)
l = [7, 9, 13],
n = { name = "Kurt", age = 34 }:
```

```
manhattan point =
  let
    (x, y) = point
  in
    x + y
```

```
manhattan p -- 7.7
```

```
describe {name, age} =
  name ++ "␣is␣" ++ (toString age)
```

```
describe n -- "Kurt is 34"
```

Destructuring - Pattern matching

- In functions calls
- In case-of constructs
- In let-in constructs

```
p = (3.5, 4.2)
l = [7, 9, 13],
n = { name = "Kurt", age = 34 }:
```

```
manhattan point =
  let
    (x, y) = point
  in
    x + y
```

```
manhattan p -- 7.7
```

```
describe {name, age} =
  name ++ "␣is␣" ++ (toString age)
```

```
describe n -- "Kurt is 34"
```

```
sum list =
  case list of
    []           -> 0
    head :: tail -> head + (sum tail)
```

```
sum l -- 29
```

Selection

You can use `if-then-else` and `case-of` constructs in Elm:

```
fact n =
  if n == 0 then 1
  else n*(fact (n - 1))

case x of
  Just a  -> a
  Nothing -> 0
```

Exercise 1 - Selections in Elm

Create an Elm function that

Calculates the third product $5 * 6 = 30$ of a list of points, if the list is empty the result should be 0, if the list has less than three elements the result should be 1:

```
points = [(1, 2), (3, 4), (5, 6), (7, 8)]
```

Exercise 1 solution

```elm
import Html exposing (text)

main =
  text (toString (exercise1 points))

points = [(1,2), (3,4), (5,6), (7,8)]
-- points = [(1,2), (3,4)]
-- points = []

exercise1 list =
  case list of
    [] -> 0
    _ :: _ :: (z1, z2) :: _ -> z1*z2
    _ -> 1
```

Iteration

Where did the `while` and `for` loops go?

Iteration

Where did the `while` and `for` loops go?

There aren't any, you have to use recursion!

Iteration

Where did the `while` and `for` loops go?

There aren't any, you have to use recursion! But you get help from the List core module:

- Create lists with:

```
List.repeat 3 (0,0) == [(0,0),(0,0),(0,0)]
List.range 3 6 == [3, 4, 5, 6]
1 :: [2,3] == [1,2,3]
1 :: [] == [1]
List.append [1,1,2] [3,5,8] == [1,1,2,3,5,8]
['a','b'] ++ ['c'] == ['a','b','c']
```

- Map and fold lists:

```
List.map sqrt [1,4,9] == [1,2,3]
List.map2 (+) [1,2,3] [1,2,3,4] == [2,4,6]
List.sum [1,2,3,4] == 10
List.product [1,2,3,4] == 24
```

Exercise 2 - List based factorial

Create a function `factorial` that calculates $n!$ for $n > 0$ using the `List` module, especially `range` and `product` are interesting.

Exercise 3 - Iterations in Elm

Create a function, that calculates:

$$4 \cdot \sum_{n=1}^{100} (-1)^{n+1} \cdot \frac{1}{2n-1}$$

Exercise 3 - Iterations in Elm

Create a function, that calculates:

$$4 \cdot \sum_{n=1}^{100} (-1)^{n+1} \cdot \frac{1}{2n-1}$$

Hint, that is the same as:

$$4 \cdot \left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots + \frac{1}{197} - \frac{1}{199} \right)$$

Sequence

In Elm, monads are hidden in the architecture, but we will return to monads in **Haskell**.

But surprisingly, we don't need much sequential processing creating web pages!

Maybe

Maybe

In Java

```java
Person getPerson(Long id) throws IOException {
  return database.getPersonById(id);
}
```

Maybe

In Java

```java
Person getPerson(Long id) throws IOException {
  return database.getPersonById(id);
}
```

If Elm does not have `null` values or exceptions, how do you represent a failure?

Maybe

In Java

```
Person getPerson(Long id) throws IOException {
  return database.getPersonById(id);
}
```

If Elm does not have `null` values or exceptions, how do you represent a failure?

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

Maybe

In Java

```java
Person getPerson(Long id) throws IOException {
  return database.getPersonById(id);
}
```

If Elm does not have `null` values or exceptions, how do you represent a failure?

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

```elm
getPerson : Int -> Maybe Person
getPerson id = ...
```

Union types 1/2

A union type is a piece of memory which can take the form of one or more values, but only one at the time.

Union types 1/2

A union type is a piece of memory which can take the form of one or more values, but only one at the time.

Is this a union type?

```
int amIUnion = 20;
```

Union types 1/2

A union type is a piece of memory which can take the form of one or more values, but only one at the time.

Is this a union type?

```
int amIUnion = 20;
```

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

Union types 1/2

A union type is a piece of memory which can take the form of one or more values, but only one at the time.

Is this a union type?

```
int amIUnion = 20;
```

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

```
maybeBaby : Maybe String
```

Union types 1/2

A union type is a piece of memory which can take the form of one or more values, but only one at the time.

Is this a union type?

```
int amIUnion = 20;
```

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

```
maybeBaby : Maybe String
```

`maybeBaby` can now either be

Union types 1/2

A union type is a piece of memory which can take the form of one or more values, but only one at the time.

Is this a union type?

```
int amIUnion = 20;
```

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

```
maybeBaby : Maybe String
```

`maybeBaby` can now either be `Just String`

Union types 1/2

A union type is a piece of memory which can take the form of one or more values, but only one at the time.

Is this a union type?

```
int amIUnion = 20;
```

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

```
maybeBaby : Maybe String
```

`maybeBaby` can now either be `Just String` or `Nothing`

Union types 1/2

A union type is a piece of memory which can take the form of one or more values, but only one at the time.

Is this a union type?

```
int amIUnion = 20;
```

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

```
maybeBaby : Maybe String
```

`maybeBaby` can now either be `Just String` or `Nothing`

It **cannot be anything else**.

Capturing errors

Capturing errors

In Java

```java
Person getPerson(Long id) throws IOException {
  return database.getPersonById(id);
}
```

Capturing errors

In Java

```java
Person getPerson(Long id) throws IOException {
  return database.getPersonById(id);
}
```

Did we forget something?

Capturing errors

In Java

```java
Person getPerson(Long id) throws IOException {
  return database.getPersonById(id);
}
```

Did we forget something? Yes! The exception!

Capturing errors

In Java

```
Person getPerson(Long id) throws IOException {
  return database.getPersonById(id);
}
```

Did we forget something? Yes! The exception!

```
type Either a b
  = Left a          -- The exception
  | Right b         -- The success
```

Capturing errors

In Java

```java
Person getPerson(Long id) throws IOException {
  return database.getPersonById(id);
}
```

Did we forget something? Yes! The exception!

```elm
type Either a b
  = Left a        -- The exception
  | Right b       -- The success
```

```elm
getPerson : Int -> Either Exception Person
getPerson id = ...
```

Union types 2/2

A union type is a piece of memory which can take the form of one or more values, but only one at the time.

```
type Either a b
  = Left a        -- The exception
  | Right b       -- The success
```

HTTP

For a HTTP call, what would you expect as input?

HTTP

For a HTTP call, what would you expect as input?
For a HTTP call, what would you expect as output?

HTTP

For a HTTP call, what would you expect as input?
For a HTTP call, what would you expect as output?

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

HTTP

For a HTTP call, what would you expect as input?
For a HTTP call, what would you expect as output?

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

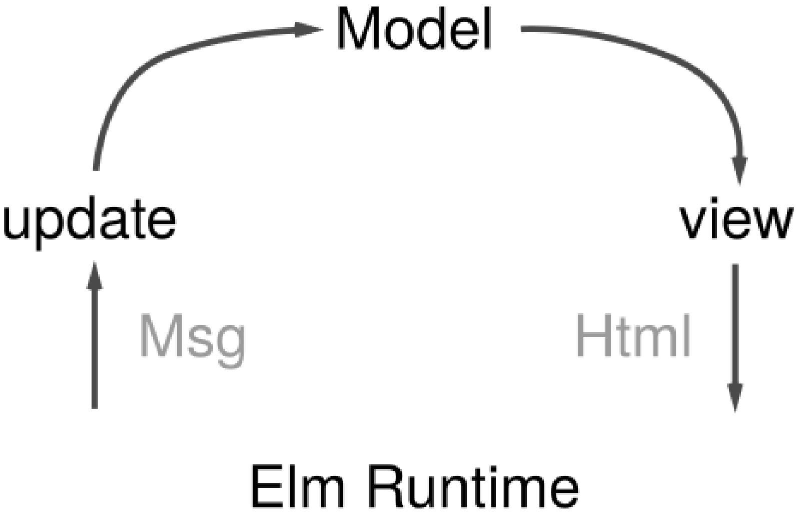
Union type

HTTP request

```
type alias Request a
  = Request a
```

```
getString : String -> Request String
```

HTML updates



HTTP to HTML

To insert the HTTP result, we have to put it into the HTML page with a `Cmd`

HTTP to HTML

To insert the HTTP result, we have to put it into the HTML page with a `Cmd`

```
type Msg
  = NewContent ?
```

HTTP to HTML

To insert the HTTP result, we have to put it into the HTML page with a `Cmd`

```
type Msg
  = NewContent ?
```

```
type Msg
  = NewContent (Result Http.Error String)
```

Getting the HTTP Result

Now we have a `HTTP Request` and a way to insert it into our view

But how do we get the `HTTP Result`?

Getting the HTTP Result

Now we have a HTTP Request and a way to insert it into our view

But how do we get the HTTP Result?

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

Getting the HTTP Result

Now we have a `HTTP Request` and a way to insert it into our view

But how do we get the `HTTP Result`?

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

Translated:

- `HTTP.send` takes two parameters

Getting the HTTP Result

Now we have a `HTTP Request` and a way to insert it into our view

But how do we get the `HTTP Result`?

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

Translated:

- `HTTP.send` takes two parameters
- 1: One function which takes a result and converts it into something else

Getting the HTTP Result

Now we have a HTTP Request and a way to insert it into our view

But how do we get the HTTP Result?

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

Translated:

- HTTP.send takes two parameters
- 1: One function which takes a result and converts it into something else
- 2: One request which performs the HTTP call

Getting the HTTP Result

Now we have a `HTTP Request` and a way to insert it into our view

But how do we get the `HTTP Result`?

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

Translated:

- `HTTP.send` takes two parameters
- 1: One function which takes a result and converts it into something else
- 2: One request which performs the HTTP call
- `HTTP.send` returns the message extracted from the first function

Putting it together

```
import Http

type Msg = Click | NewBook (Result Http.Error String)

update : Msg -> Model -> Model
update msg model =
  case msg of
    Click -> ( model, getWarAndPeace )

    NewBook (Ok book) -> ...

    NewBook (Err _) -> ...

getWarAndPeace : Cmd Msg
getWarAndPeace =
  Http.send NewBook <|
    Http.getString "https://example.com/some_book.md"
```

Putting it together

Live coding!

Architecture

The Monad - the program

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (onInput)

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

Architecture

Model

```elm
type alias Model =
  { name : String
  , password : String
  , passwordAgain : String
  }

model : Model
model =
  Model "" "" ""
```

Architecture

View

```
view : Model -> Html Msg
view model =
  div []
    [ input
      [ type_ "text"
      , placeholder "Name"
      , onInput Name ] []
    , input
      [ type_ "password"
      , placeholder "Password"
      , onInput Password ] []
    , input
      [ type_ "password"
      , placeholder "Re-enter␣Password"
      , onInput PasswordAgain ] []
    , viewValidation model
    ]
```

Architecture

View

```
viewValidation : Model -> Html msg
viewValidation model =
  let
    (color, message) =
      if model.password == model.passwordAgain then
        ("green", "OK")
      else
        ("red", "Passwords␣do␣not␣match!")
  in
    div [ style [("color", color)] ] [ text message ]
```

Architecture

Update

```elm
type Msg
    = Name String
    | Password String
    | PasswordAgain String

update : Msg -> Model -> Model
update msg model =
  case msg of
    Name name ->
      { model | name = name }
    Password password ->
      { model | password = password }
    PasswordAgain password ->
      { model | passwordAgain = password }
```

Exercise 5 - Hello anybody

Create a hello world web site with one input field and a text field that shows `"Hello "` and the content of the input field when a button is pushed.

```
$ mkdir hello
$ cd hello
```

Copy the following into `Main.elm`

```
import Html exposing (text)

main =
  text "Hello,␣World!"
```

And:

```
$ elm-package install -y
$ elm-reactor
```

The code from today can be found here: `https://github.com/cphbus-functional-programming/elm-exercises`

REST assignment

Write a server in a language of your choice with two HTTP REST methods:

1. `GET /counter`: increments and returns an integer counter
2. `PUT /counter/{value}`: sets the counter to `value`

REST assignment

Write a server in a language of your choice with two HTTP REST methods:

1. `GET /counter`: increments and returns an integer counter
2. `PUT /counter/{value}`: sets the counter to `value`

Write an Elm client using the model - view - update architecture. Your client must have:

1. A model containing one counter `Model { counter:  Int }`
2. A view with two HTML buttons (get and set) as well as the counter in an HTML `H2` element
3. An update part which can 1) get the counter value from your REST service and 2) set the counter to a fixed value of 1

REST assignment

Write a server in a language of your choice with two HTTP REST methods:

1. `GET /counter`: increments and returns an integer counter
2. `PUT /counter/{value}`: sets the counter to `value`

Write an Elm client using the model - view - update architecture. Your client must have:

1. A model containing one counter `Model { counter:  Int }`
2. A view with two HTML buttons (get and set) as well as the counter in an HTML `H2` element
3. An update part which can 1) get the counter value from your REST service and 2) set the counter to a fixed value of 1

Hand in `before` **17th March 12:00**
Review `before` **18th March 23:59**