# Functional Programming with Elm

Jos van Bakel
Media2B
PyGrunn 2017

# Introducing: Elm

Elm is purely functional

Elm is statically typed

Elm is declarative

Elm compiles to Javascript.

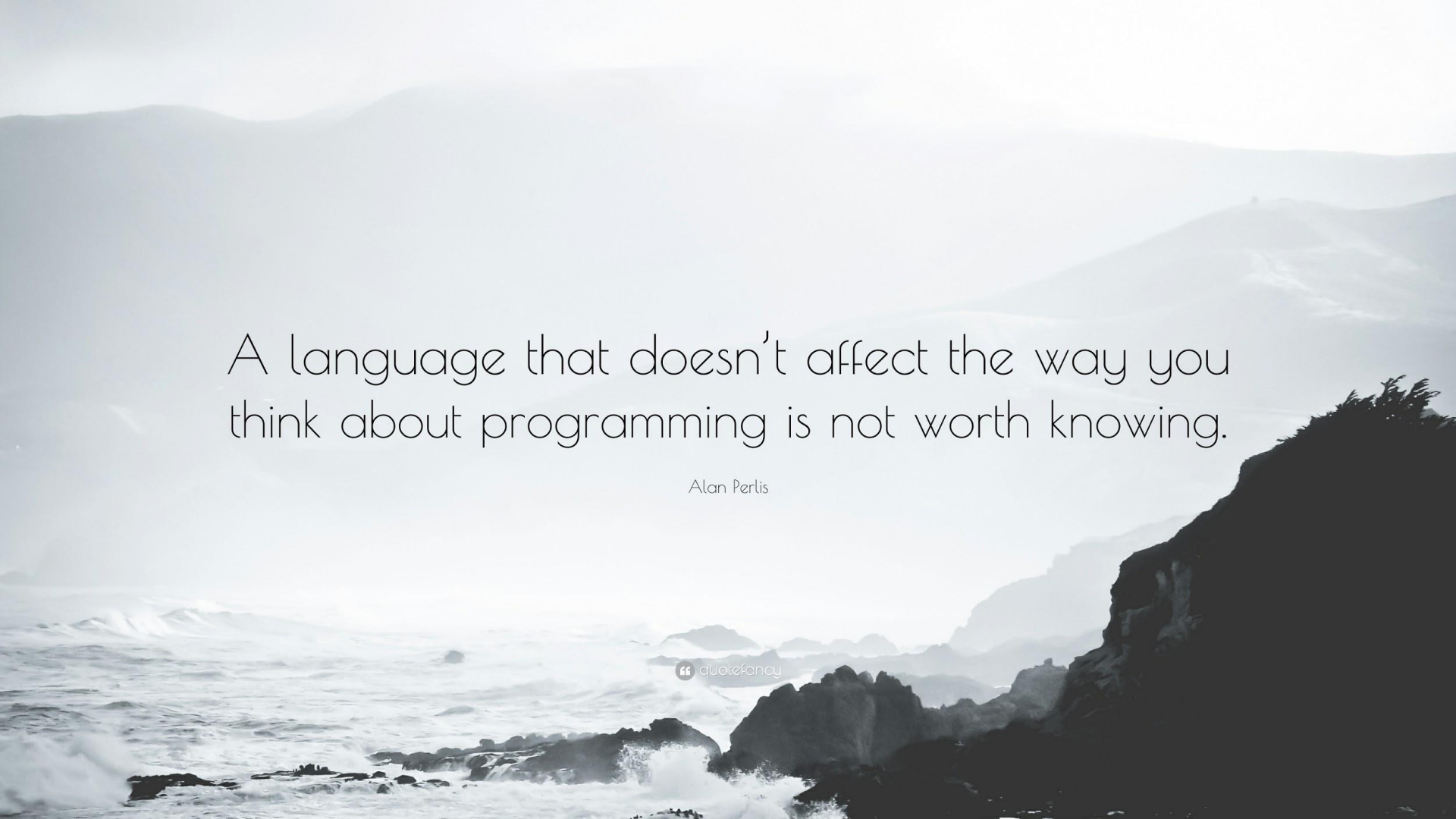
Elm makes it fun again to make web GUI's.

Created in 2012 by Evan Czaplicki. BDFL --->

# Why yet another language?

TIOBE Index May 2017

| Number | Language | Ratings | Number | Language | Ratings |
|---|---|---|---|---|---|
| 1 | Java | 14.6% | 6 | Visual Basic .NET | 3.3% |
| 2 | C | 7.0% | 7 | JavaScript | 3.0% |
| 3 | C++ | 4.7% | 8 | Assembly language | 2.8% |
| 4 | Python | 3.5% | 9 | PHP | 2.6% |
| 5 | C# | 3.4% | 10 | Perl | 2.6% |


A language that doesn't affect the way you think about programming is not worth knowing.
Alan Perlis
quotefancy

# Agenda

- What is Functional Programming
- Elm by example
- The Elm Architecture
- Demo
- Superpowers
- Conclusion

# What is Functional Programming

In computer science, **functional programming** is a **programming** paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.


Functional programming - Wikipedia
https://en.wikipedia.org/wiki/Functional_programming

About this result  Feedback

# What is Functional Programming

In **functional code**, the output value of a function depends only on the arguments that are passed to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f*(*x*) each time.


Functional programming - Wikipedia
https://en.wikipedia.org/wiki/Functional_programming

About this result  Feedback

# … and Elm

“If typed functional programming is so great, how come nobody ever uses it?”

*Evan Czaplicki*

# Python

```python
def imperative_plus_one(lst):
    new_list = []
    for i in range(0, len(lst)):
        new_list.append(lst[i] + 1)
    return new_list
```

# Python vs Python

```python
def imperative_plus_one(lst):
    new_list = []
    for i in range(0, len(lst)):
        new_list.append(lst[i] + 1)
    return new_list
```

```python
def functional_plus_one(lst):
    return list(map(lambda x: x + 1, lst))
```

# Python vs Python vs Elm

```python
def imperative_plus_one(lst):
    new_list = []
    for i in range(0, len(lst)):
        new_list.append(lst[i] + 1)
    return new_list
```

```elm
plusOne lst = List.map (\x -> x + 1) lst
```

```python
def functional_plus_one(lst):
    return list(map(lambda x: x + 1, lst))
```

# Python vs Python vs Elm vs Elm

```python
def imperative_plus_one(lst):
    new_list = []
    for i in range(0, len(lst)):
        new_list.append(lst[i] + 1)
    return new_list
```

```elm
plusOne lst = List.map (\x -> x + 1) lst
```

```elm
inc = (+) 1
plusOne = List.map inc
```

```python
def functional_plus_one(lst):
    return list(map(lambda x: x + 1, lst))
```

# Python vs Elm: lists

```python
def head(lst):
    return lst[0]

head([1, 2, 3])
# 1
head([])
# IndexError: list index out of range

def headOrZero(lst):
    return lst[0] if lst else 0
```

```elm
List.head [1, 2, 3]
-- Just 1

List.head []
-- Nothing

headOrZero lst =
    case List.head lst of
        Just x  -> x
        Nothing -> 0
```

# Python vs Elm: currying

```python
def append(a, b):
    return a + " " + b

def append_curry(a):
    return lambda b: a + " " + b

say_hello = append_curry("Hello")
say_hello("PyGrunn")
# "Hello PyGrunn"

list(map(say_hello,
    ["World", "PyGrunn", "Groningen"]))
# ...
```

```elm
append a b = a ++ " " ++ b
-- <function>: String -> String -> String

sayHello = append "Hello"
-- <function>: String -> String

sayHelloToAll = List.map sayHello
-- <function>: List String -> List String

sayHelloToAll ["World", "PyGrunn", "Groningen"]
```

# Python vs Elm: Recursion

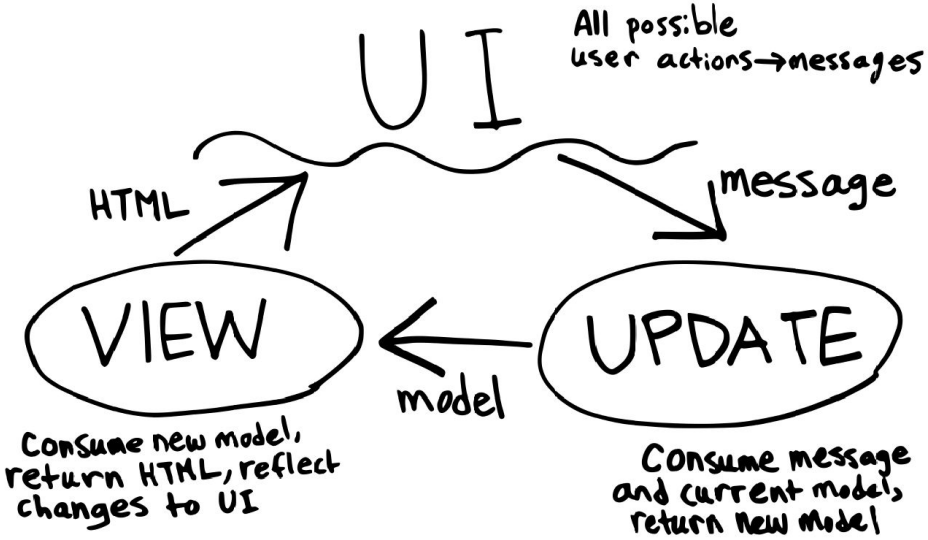
```python
def map(fn, lst):
    new_list = []
    for elem in lst:
        new_list.append(fn(elem))
    return new_list

inc = lambda x: x + 1
map(inc, range(1, 10))
```

```elm
map fn lst =
    case lst of
        []       -> []
        hd :: tl -> (fn hd) :: (map fn tl)

inc x = x + 1
map inc (List.range 1 10)
```

# The Elm Architecture



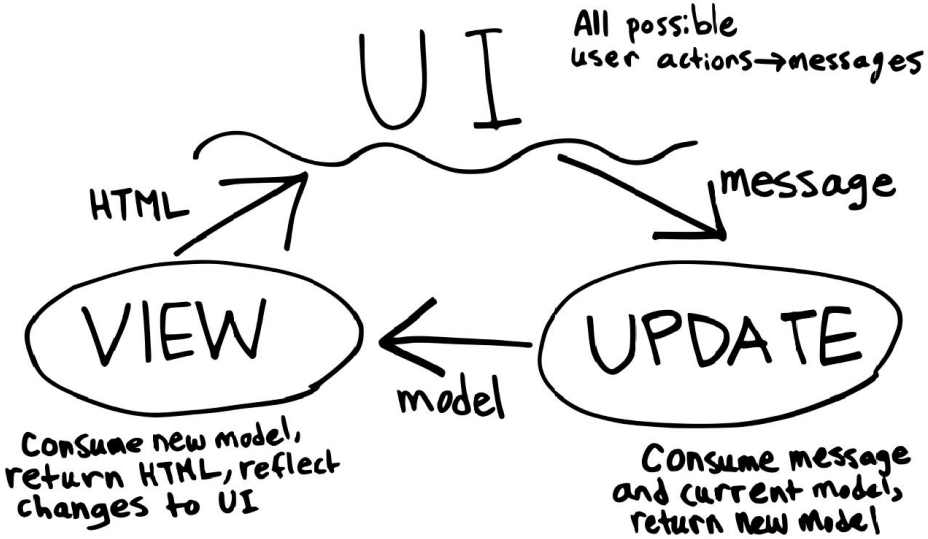
# The Elm Architecture

```
type Model = ...

type Msg = …

update : Msg -> Model -> Model
update msg model = ...

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

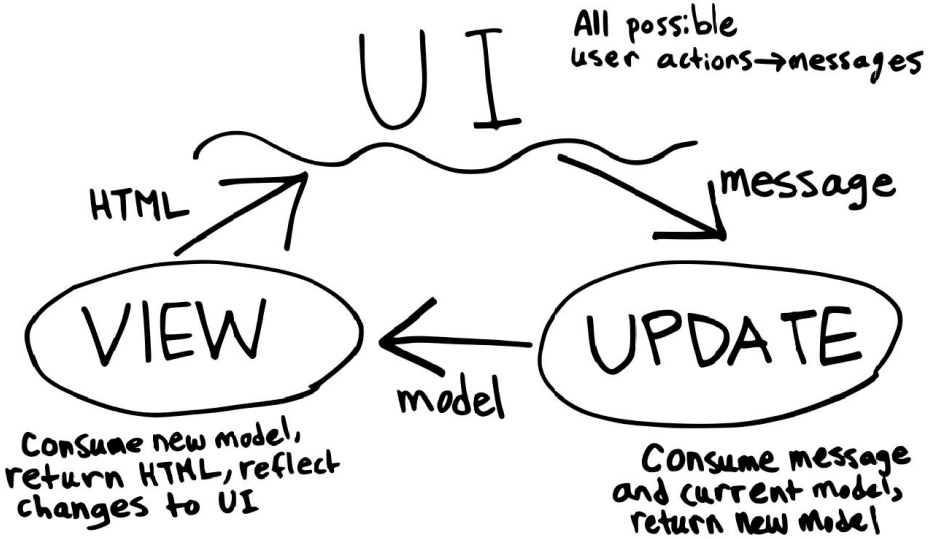


# The Elm Architecture

```
type alias Model = Int
type Msg = Increment | Decrement

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

view model = div []
   [ h1 [] [ text "A simple counter" ]
   , button [ onClick Increment ] [ text "+" ]
   , div [] [ text (toString model) ]
   , button [ onClick Decrement ] [ text "-" ]
   ]
```



# The Elm Architecture: I/O

Output: commands

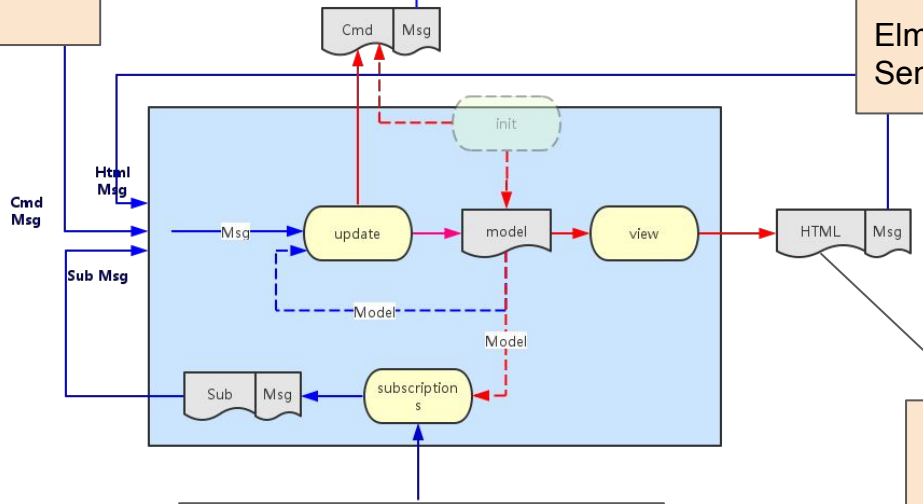
- HTTP request
- Websockets
- Local storage
- Random
- Time

Input: subscriptions

- Keyboard
- Mouse
- Websocket
- Navigation

# The Elm Architecture with I/O

Elm runtime:
Execute commands.




Elm runtime:
Send Msg on DOM events.

Elm runtime:
VDOM diff and update DOM

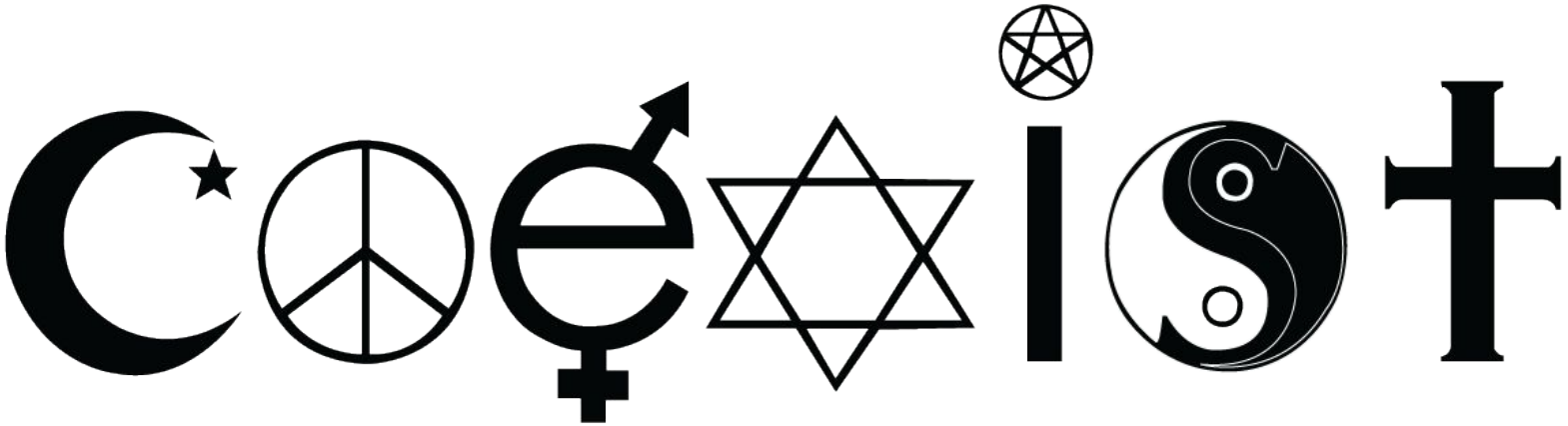
Elm runtime:
Listen to subscriptions.


DEMO

# Superpowers

| Feature | Power |
| --- | --- |
| Statically typed | Simple to refactor, be confident about it |
| Controlled side effects | Simple to reason about |
| No runtime errors | Reliable |
| No monads, no type classes | Quickly get up to speed |
| Enforced formatting (elm-format) | Readable |
| Friendly error messages | Developer friendly |
| Pure functions | Faster than Javascript |

# It’s not a religion



# Where do I start?

http://elm-lang.org/examples

https://guide.elm-lang.org/

https://www.elm-tutorial.org/

# We’re hiring!

# Python vs Elm: Pipelines

```python
input = "a string with not too many words"

' '.join(
   map(str,
       sorted(
           map(len,
               input.split(' '))))))
```

```elm
"a string with not too many words"
   |> String.words
   |> List.map String.length
   |> List.sort
   |> List.map toString
   |> String.join " "
```

# Python vs Elm: Recursion (2)

```python
def filter(predicate, lst):
    new_list = []
    for i in range(0, len(lst)):
        if predicate(lst[i]):
            new_list.append(lst[i])
    return new_list

isEven = lambda x: x % 2 == 0
filter(isEven, range(1, 10))
# [2, 4, 6, 8]
```

```elm
filter : (a -> Bool) -> List a -> List a
filter predicate lst =
    case lst of
        [] ->
            []

        hd :: tl ->
            if predicate hd then
                hd :: filter predicate tl
            else
                filter predicate tl

isEven x = x % 2 == 0
filter isEven (List.range 1 10)
```

# The Elm Architecture: controlled (side) effects

```
type Msg
   = RequestTime
   | NewTime Time


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
   case msg of
       RequestTime ->
           ( model, Task.perform NewTime Time.now )

       NewTime time ->
           ( time, Cmd.none )
```

# The Elm Architecture: subscriptions

```
type Msg
   = Tick Time


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


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