Elm. Functional, Reactive, for the Web

Grzegorz Balcerek

Lambda Days 2015

Functional

- statically-typed

- statically-typed
- Haskell-like syntax

- statically-typed
- Haskell-like syntax
- strict

- statically-typed
- Haskell-like syntax
- strict
- functions as values

- statically-typed
- Haskell-like syntax
- strict
- functions as values
- immutable data structures

```
module Fib where

import List (head,reverse,tail,(::))

fib : Int -> List Int
fib n =
  let second = tail >> head
      nextNumber numbers =
        head numbers + second numbers
      fib’ n ns =
        if n <= 2
        then ns
        else fib’ (n-1) (nextNumber ns :: ns)
  in fib’ n [1,0] |> reverse
```

$

```
$ elm-repl
$ elm-repl
Elm REPL 0.4 <https://github.com/elm-lang/elm-repl#elm-repl>
Type :help for help, :exit to exit
>
```

```
$ elm-repl
Elm REPL 0.4 <https://github.com/elm-lang/elm-repl#elm-repl>
Type :help for help, :exit to exit
> import Fib (fib)
```

```
$ elm-repl
Elm REPL 0.4 <https://github.com/elm-lang/elm-repl#elm-repl>
Type :help for help, :exit to exit
> import Fib (fib)
>
```

```
$ elm-repl
Elm REPL 0.4 <https://github.com/elm-lang/elm-repl#elm-repl>
Type :help for help, :exit to exit
> import Fib (fib)
> fib 8
```

```
$ elm-repl
Elm REPL 0.4 <https://github.com/elm-lang/elm-repl#elm-repl>
Type :help for help, :exit to exit
> import Fib (fib)
> fib 8
[0,1,1,2,3,5,8,13] : List Int
>
```

```
$ elm-repl
Elm REPL 0.4 <https://github.com/elm-lang/elm-repl#elm-repl>
Type :help for help, :exit to exit
> import Fib (fib)
> fib 8
[0,1,1,2,3,5,8,13] : List Int
> fib 12
```

```
$ elm-repl
Elm REPL 0.4 <https://github.com/elm-lang/elm-repl#elm-repl>
Type :help for help, :exit to exit
> import Fib (fib)
> fib 8
[0,1,1,2,3,5,8,13] : List Int
> fib 12
[0,1,1,2,3,5,8,13,21,34,55,89] : List Int
>
```

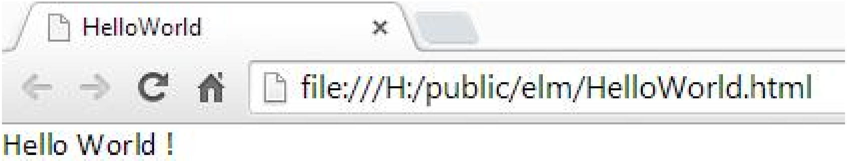
```
$ elm-repl
Elm REPL 0.4 <https://github.com/elm-lang/elm-repl#elm-repl>
Type :help for help, :exit to exit
> import Fib (fib)
> fib 8
[0,1,1,2,3,5,8,13] : List Int
> fib 12
[0,1,1,2,3,5,8,13,21,34,55,89] : List Int
> :exit
```

```
$ elm-repl
Elm REPL 0.4 <https://github.com/elm-lang/elm-repl#elm-repl>
Type :help for help, :exit to exit
> import Fib (fib)
> fib 8
[0,1,1,2,3,5,8,13] : List Int
> fib 12
[0,1,1,2,3,5,8,13,21,34,55,89] : List Int
> :exit

$
```

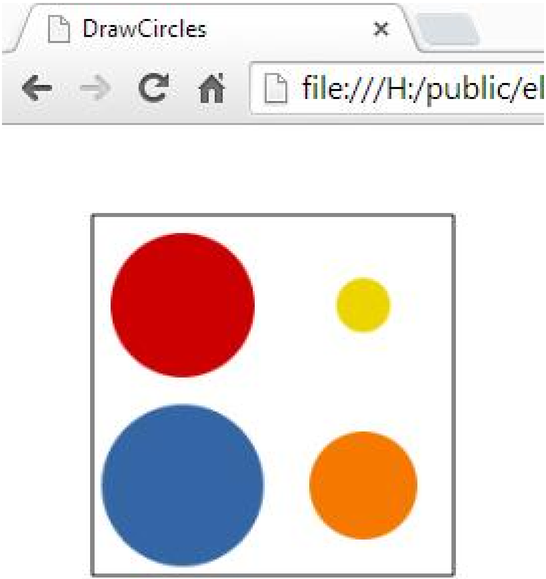
Web

```
module HelloWorld where

import Graphics.Element (Element)
import Text (plainText)

main : Element
main = plainText "Hello World !"
```

```
$ elm-make HelloWorld.elm --output HelloWorld.html
Successfully generated HelloWorld.html
```



Hello World !


DrawCircles
file:///H:/public/el

```
color : Int -> Color
color n =
  let colors = [ green, red, blue, yellow,
                 brown, purple, orange ]
  in
      head <| drop (n % length colors) colors
```

```
color : Int -> Color
color n =
  let colors = [ green, red, blue, yellow,
                 brown, purple, orange ]
  in
      head <| drop (n % length colors) colors

type alias CircleDescription =
  (Int, (Int, Int))

circleForm : CircleDescription -> Form
circleForm (r, (x, y)) =
    circle (toFloat r*5)
        |> filled (color r)
        |> move (toFloat x,toFloat y)
```

```
drawCircles : List CircleDescription
           -> (Int, Int)
           -> Element
drawCircles descriptions (w, h) =
  collage w h (map circleForm descriptions)
```

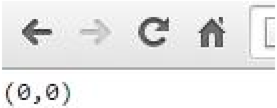
```
fourCircles : Element
fourCircles = drawCircles [
    (3, (50, 50)), (6, (50, -50)),
    (8, (-50, 50)), (9, (-50, -50))
  ] (300, 300)
```

```
fourCircles : Element
fourCircles = drawCircles [
    (3, (50, 50)), (6, (50, -50)),
    (8, (-50, 50)), (9, (-50, -50))
  ] (300, 300)

blackSquare : Element
blackSquare = collage 300 300 [
  outlined (solid black) (rect 200 200) ]
```

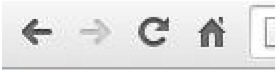
```
fourCircles : Element
fourCircles = drawCircles [
    (3, (50, 50)), (6, (50, -50)),
    (8, (-50, 50)), (9, (-50, -50))
  ] (300, 300)

blackSquare : Element
blackSquare = collage 300 300 [
  outlined (solid black) (rect 200 200) ]

main : Element
main = layers [ blackSquare, fourCircles ]
```

Reactive

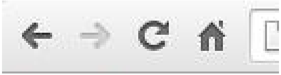
```
main : Signal Element
main = Signal.map asText Mouse.position
```



```
(0,0)
```



(0,0)



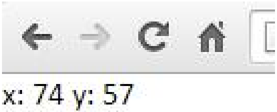
(206,73)

(0,0)

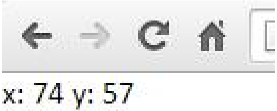
(206,73)

(1238,550)

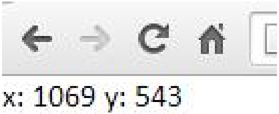
```elm
showXY : Int -> Int -> Element
showXY x y = plainText <|
  "x: " ++ toString x ++ " y: " ++ toString y

main : Signal Element
main = Signal.map2 showXY Mouse.x Mouse.y
```



x: 74 y: 57



x: 74 y: 57

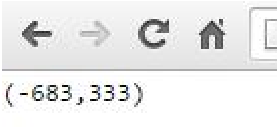


x: 1069 y: 543

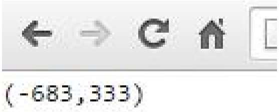
```
showXY : Int -> Int -> Element
showXY x y = plainText <|
  "x: " ++ toString x ++ " y: " ++ toString y

main : Signal Element
main = showXY <~ Mouse.x ~ Mouse.y
```

```
mousePosition : Signal (Int, Int)
mousePosition =
  let adjust (w, h) (x, y) = (x-w//2,h//2-y)
  in
     adjust <~ Window.dimensions
             ~ Mouse.position
```

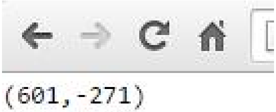
```
mousePosition : Signal (Int, Int)
mousePosition =
  let adjust (w, h) (x, y) = (x-w//2,h//2-y)
  in
     adjust <~ Window.dimensions
             ~ Mouse.position

main : Signal Element
main = asText <~ mousePosition
```



```
(-683,333)
```



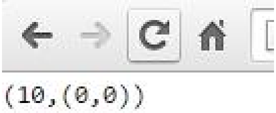
```
(-683,333)
```



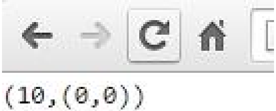
```
(601,-271)
```

```
delayedPosition : Int
               -> Signal (Int,Int)
               -> Signal (Int, (Int,Int))
delayedPosition time positionSignal =
  Signal.map (\pos -> (time,pos)) <|
    delay (toFloat time*100) positionSignal
```

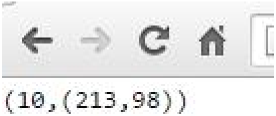
```
delayedPosition : Int
               -> Signal (Int,Int)
               -> Signal (Int, (Int,Int))
delayedPosition time positionSignal =
  Signal.map (\pos -> (time,pos)) <|
    delay (toFloat time*100) positionSignal

main : Signal Element
main =
  asText <~ delayedPosition 10 Mouse.position
```



```
(10,(0,0))
```



```
(10,(0,0))
```

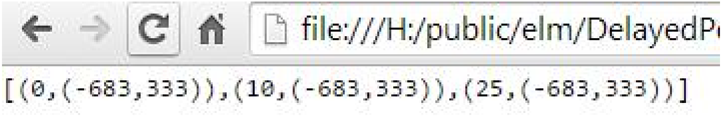


```
(10,(213,98))
```

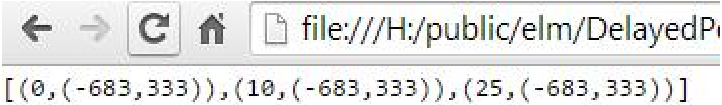
```
delayedPositionsList : List Int
            -> List (Signal (Int, (Int, Int)))
delayedPositionsList rs =
  List.map2 delayedPosition rs <|
    repeat (length rs) mousePosition
```

```
sequence : List (Signal a) -> Signal (List a)
sequence =
  foldr (Signal.map2 (::)) (constant [])
```

```
sequence : List (Signal a) -> Signal (List a)
sequence =
  foldr (Signal.map2 (::)) (constant [])

delayedPositions : List Int
           -> Signal (List (Int, (Int, Int)))
delayedPositions =
  sequence << delayedPositionsList
sequence : List (Signal a) -> Signal (List a)
sequence =
  foldr (Signal.map2 (::)) (constant [])

delayedPositions : List Int
           -> Signal (List (Int, (Int, Int)))
delayedPositions =
  sequence << delayedPositionsList

main : Signal Element
main = asText <~ delayedPositions [0,10,25]
```

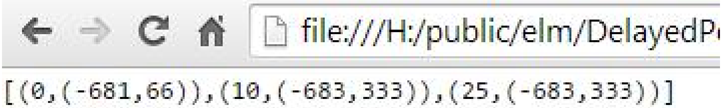


```
[(0,(-683,333)),(10,(-683,333)),(25,(-683,333))]
```



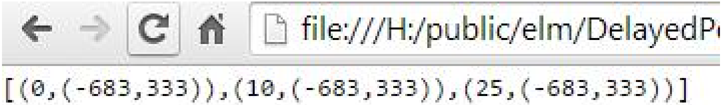
file:///H:/public/elm/DelayedP

```
[(0,(-683,333)),(10,(-683,333)),(25,(-683,333))]
```



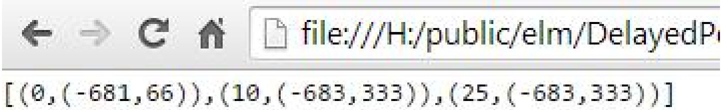
file:///H:/public/elm/DelayedP

```
[(0,(-681,66)),(10,(-683,333)),(25,(-683,333))]
```



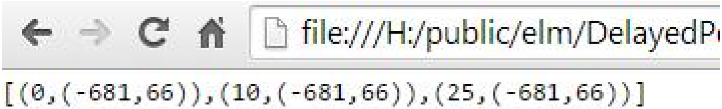
file:///H:/public/elm/DelayedP

```
[(0,(-683,333)),(10,(-683,333)),(25,(-683,333))]
```



file:///H:/public/elm/DelayedP

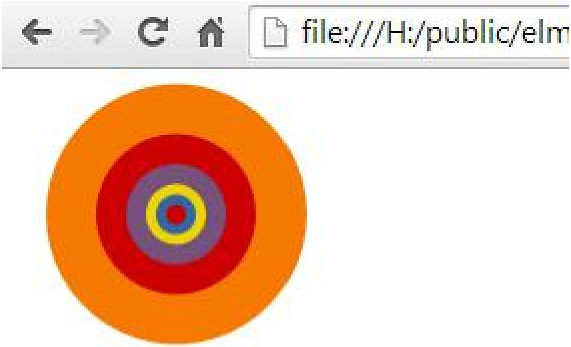
```
[(0,(-681,66)),(10,(-683,333)),(25,(-683,333))]
```

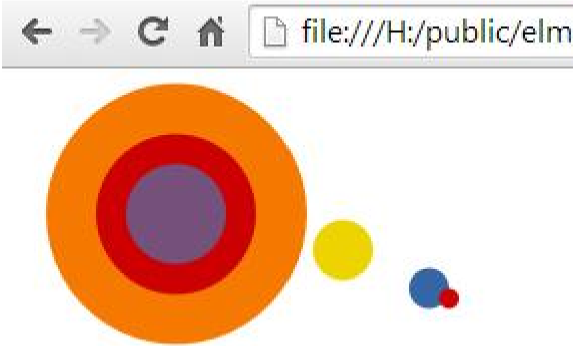


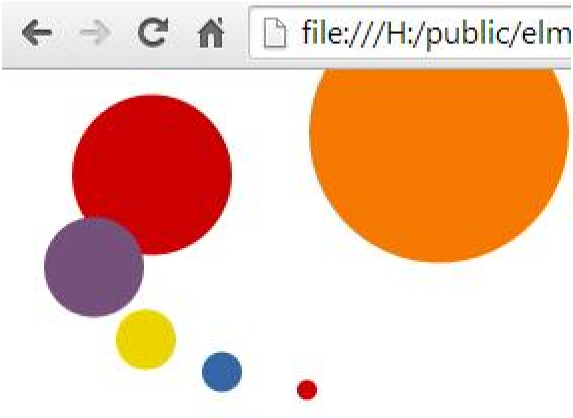
file:///H:/public/elm/DelayedP

```
[(0,(-681,66)),(10,(-681,66)),(25,(-681,66))]
```

```
main : Signal Element
main =
  drawCircles
    <~ delayedPositions (fib 8 |> drop 2)
     ~ Window.dimensions
```


file:///H:/public/elm


file:///H:/public/elm


file:///H:/public/elm

State

```
type alias State =
  { frozen: Bool
  , circles: List (Int, (Int, Int))
  }
```

```
type alias State =
  { frozen: Bool
  , circles: List (Int, (Int, Int))
  }

initialState : State
initialState =
  { frozen = False
  , circles = []
  }
```

```
type Event =
    Click
  | Circles (List CircleDescription)
```

```
type Event =
    Click
  | Circles (List CircleDescription)

mouseClicks : Signal Event
mouseClicks = always Click <~ Mouse.clicks
```

```
type Event =
    Click
  | Circles (List CircleDescription)

mouseClicks : Signal Event
mouseClicks = always Click <~ Mouse.clicks

circles : Signal Event
circles =
  Circles <~ delayedPositions
               (fib 8 |> drop 2)
```

```
type Event =
    Click
  | Circles (List CircleDescription)

mouseClicks : Signal Event
mouseClicks = always Click <~ Mouse.clicks

circles : Signal Event
circles =
  Circles <~ delayedPositions
               (fib 8 |> drop 2)

events : Signal Event
events = merge mouseClicks circles
```

```
step : Event -> State -> State
step event state =
  case (state.frozen, event) of
    (_, Click) ->
      { state | frozen <- not state.frozen }
    (False, Circles positions) ->
      { state | circles <- positions }
    (True, Circles _) ->
      state
```

```
step : Event -> State -> State
step event state =
  case (state.frozen, event) of
    (_, Click) ->
      { state | frozen <- not state.frozen }
    (False, Circles positions) ->
      { state | circles <- positions }
    (True, Circles _) ->
      state

stateSignal : Signal State
stateSignal =
  foldp step initialState events
```

```
statefulPositions :
      Signal (List CircleDescription)
statefulPositions =
  .circles <~ stateSignal
```

```
statefulPositions :
      Signal (List CircleDescription)
statefulPositions =
  .circles <~ stateSignal

main : Signal Element
main =
  drawCircles
    <~ statefulPositions
     ~ Window.dimensions
```

elm-lang.org

elm-by-example.org