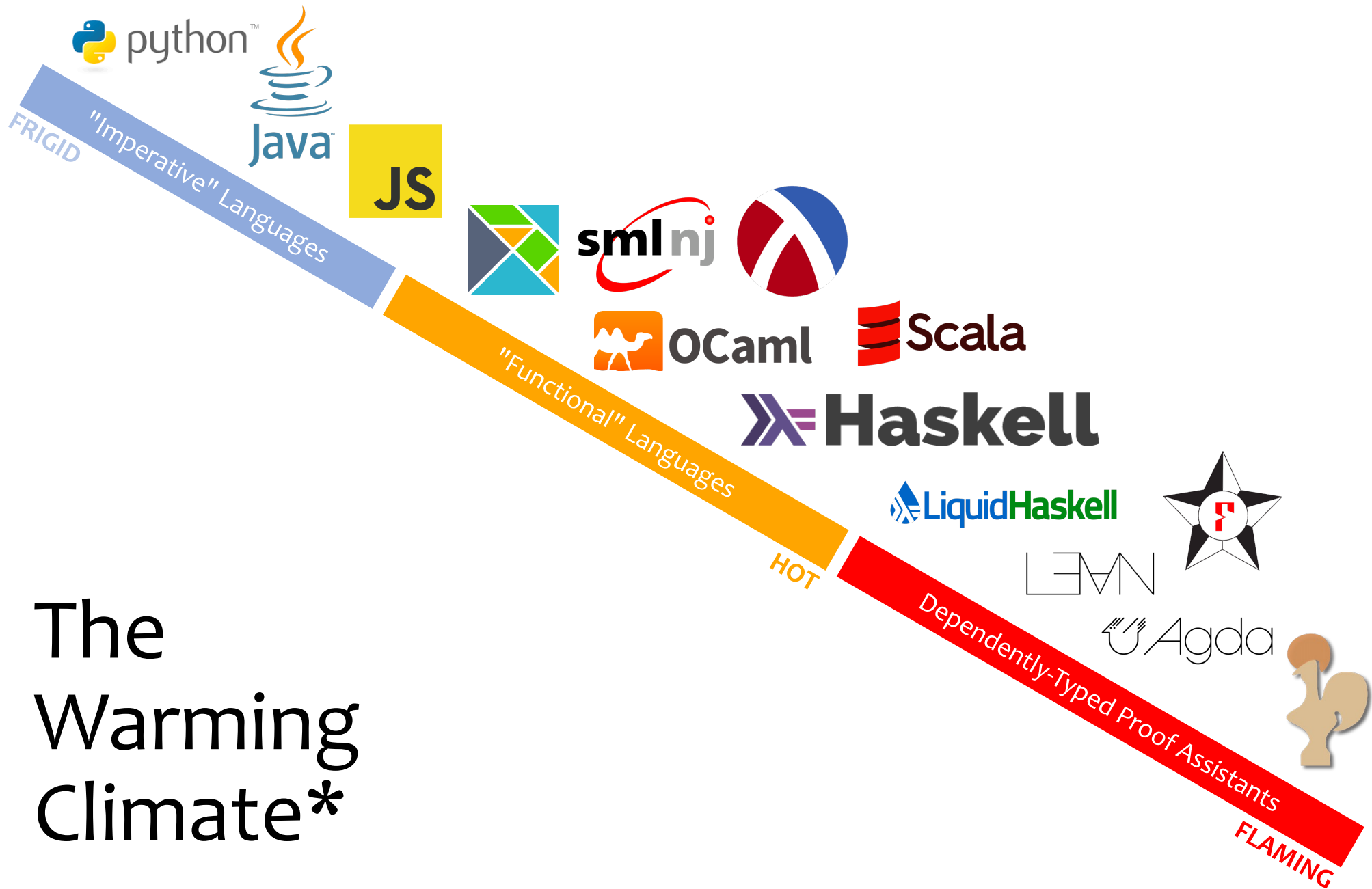


Functional Programming

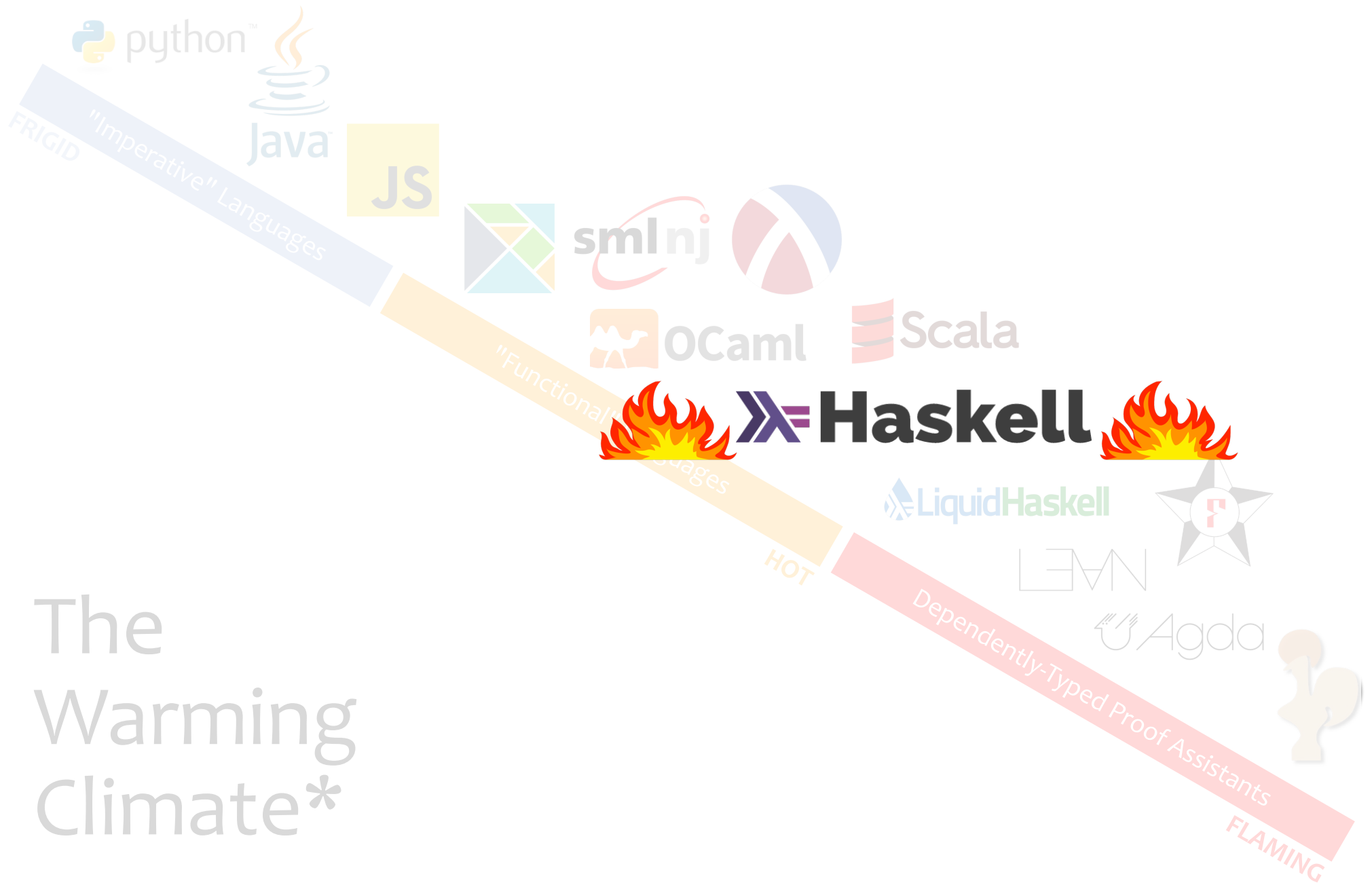
Ravi Chugh

UChicago CS 223
Winter 2023



The Warming Climate*

*An incomplete and unscientific account



The Warming Climate*

*An incomplete and unscientific account



Programming in Haskell

Algebraic Datatypes

Higher-Order Functions

Separation of Church and State

Syntactic Concision

Lazy Evaluation



Alonzo Church
 λ -calculus (1930s)



TYPE SYSTEM

A Silly Little I/O Loop

stdout

stdin

```
Tell me a nice number: Haskell, woohoo!!!  
Hmm, that doesn't seem like a number.  
Tell me a nice number: CMSC 22300  
Hmm, that doesn't seem like a number.  
Tell me a nice number: cs223  
Hmm, that doesn't seem like a number.  
Tell me a nice number: 223  
Yes, 223 is a nice number.  
Tell me a nice number: -223  
Yes, -223 is a nice number.  
Tell me a nice number:
```

```
main :: IO ()
main =
  do
    putStr "Tell me a nice number: "
    s <- getLine
    let i = read s :: Int
    putStrLn ("Yes, " ++ show i ++ " is a nice number.")
    main
```

```
main :: IO ()
main =
  do
    putStr "Tell me a nice number: "
    s <- getLine
    let i = read s :: Int
    putStrLn ("Yes, " ++ show i ++ " is a nice number.")
  main
```

```
main :: IO ()
main =
  do
    putStr "Tell me a nice number: "
    s <- getLine
    if all isDigit s then
      let i = read s :: Int in
      putStrLn ("Yes, " ++ show i ++ " is a nice number.")
    else
      putStrLn "Hmm, that doesn't seem like a number."
main
```

```
main :: IO ()
main =
  do
    putStr "Tell me a nice number: "
    s <- getLine
    if all isDigit s then
      let i = read s :: Int in
      putStrLn ("Yes, " ++ show i ++ " is a nice number.")
    else
      putStrLn "Hmm, that doesn't seem like a number."
  main
```

```
main :: IO ()
main =
  do
    putStr "Tell me a nice number: "
    s <- getLine
    let i = readInt s
    if i /= -999999999999999
    then putStrLn ("Yes, " ++ show i ++ " is a nice number.")
    else putStrLn "Hmm, that doesn't seem like a number."
main

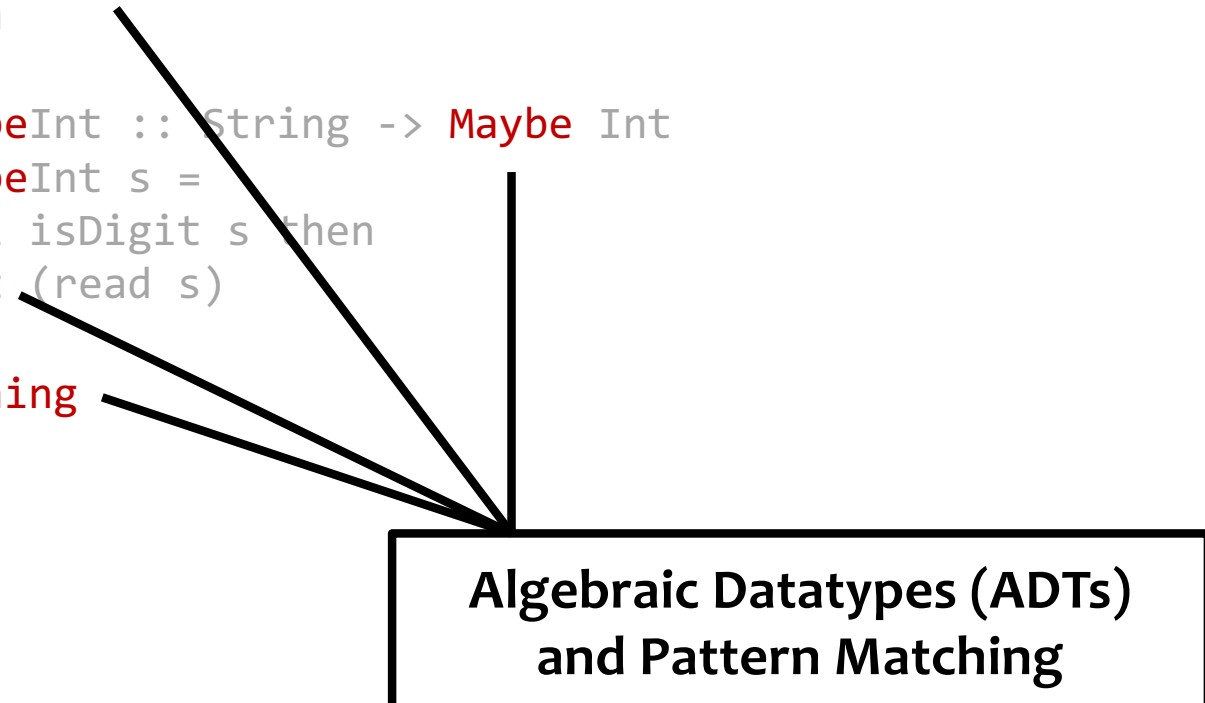
readInt :: String -> Int
readInt s =
  if all isDigit s then
    read s
  else
    -999999999999999
```

```
main :: IO ()
main =
  do
    putStr "Tell me a nice number: "
    s <- getLine
    let i = readInt s
    if i /= -999999999999999
      then putStrLn ("Yes, " ++ show i ++ " is a nice number.")
      else putStrLn "Hmm, that doesn't seem like a number."
  main

readInt :: String -> Int
readInt s =
  if all isDigit s then
    read s
  else
    -999999999999999
```

```
main :: IO ()
main =
  do
    putStr "Tell me a nice number: "
    s <- getLine
    case readMaybeInt s of
      Just i  -> putStrLn ("Yes, " ++ show i ++ " is a nice number.")
      Nothing -> putStrLn "Hmm, that doesn't seem like a number."
main

readMaybeInt :: String -> Maybe Int
readMaybeInt s =
  if all isDigit s then
    Just (read s)
  else
    Nothing
```



The diagram consists of four black arrows originating from the right side of the code block. Two arrows point to the `Just` constructor in the `case` expression of the `main` function. Two other arrows point to the `Just` and `Nothing` constructors in the `readMaybeInt` function definition. All four arrows converge towards a rectangular box at the bottom right of the slide.

**Algebraic Datatypes (ADTs)
and Pattern Matching**

```
main :: IO ()
main =
  do
    putStr "Tell me a nice number: "
    s <- getLine
    case readMaybeInt s of
      Just i  -> putStrLn ("Yes, " ++ show i ++ " is a nice number.")
      Nothing -> putStrLn "Hmm, that doesn't seem like a number."
  main
```

```
readMaybeInt :: String -> Maybe Int
readMaybeInt s =
  if all isDigit s then
    Just (read s)
  else
    Nothing
```

```
main :: IO ()
main =
  do
    putStr "Tell me a nice number: "
    s <- getLine
    putStrLn (response s)
    main
```

```
response :: String -> String
response s =
  case readMaybeInt s of
    Just i  -> "Yes, " ++ show i ++ " is a nice number."
    Nothing -> "Hmm, that doesn't seem like a number."
```

```
readMaybeInt :: String -> Maybe Int
readMaybeInt s =
  if all isDigit s then
    Just (read s)
  else
    Nothing
```

```
main :: IO ()
main =
  do
    putStr "Tell me a nice number: "
    s <- getLine
    putStrLn (response s)
    main

response :: String -> String
response s =
  case readMaybeInt s of
    Just i  -> "Yes, " ++ show i ++ " is a nice number."
    Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt s =
  if all isDigit s then
    Just (read s)
  else
    Nothing
```

```
main :: IO ()
main =
    loop "Tell me a nice number: " response
```

```
loop :: String -> (String -> String) -> IO ()
```

```
loop prompt f =
```

```
do
```

```
    putStr prompt
```

```
    s <- getLine
```

```
    putStrLn (f s)
```

```
    loop prompt f
```

```
response :: String -> String
```

```
response s =
```

```
    case readMaybeInt s of
```

```
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
```

```
        Nothing -> "Hmm, that doesn't seem like a number."
```

```
readMaybeInt :: String -> Maybe Int
```

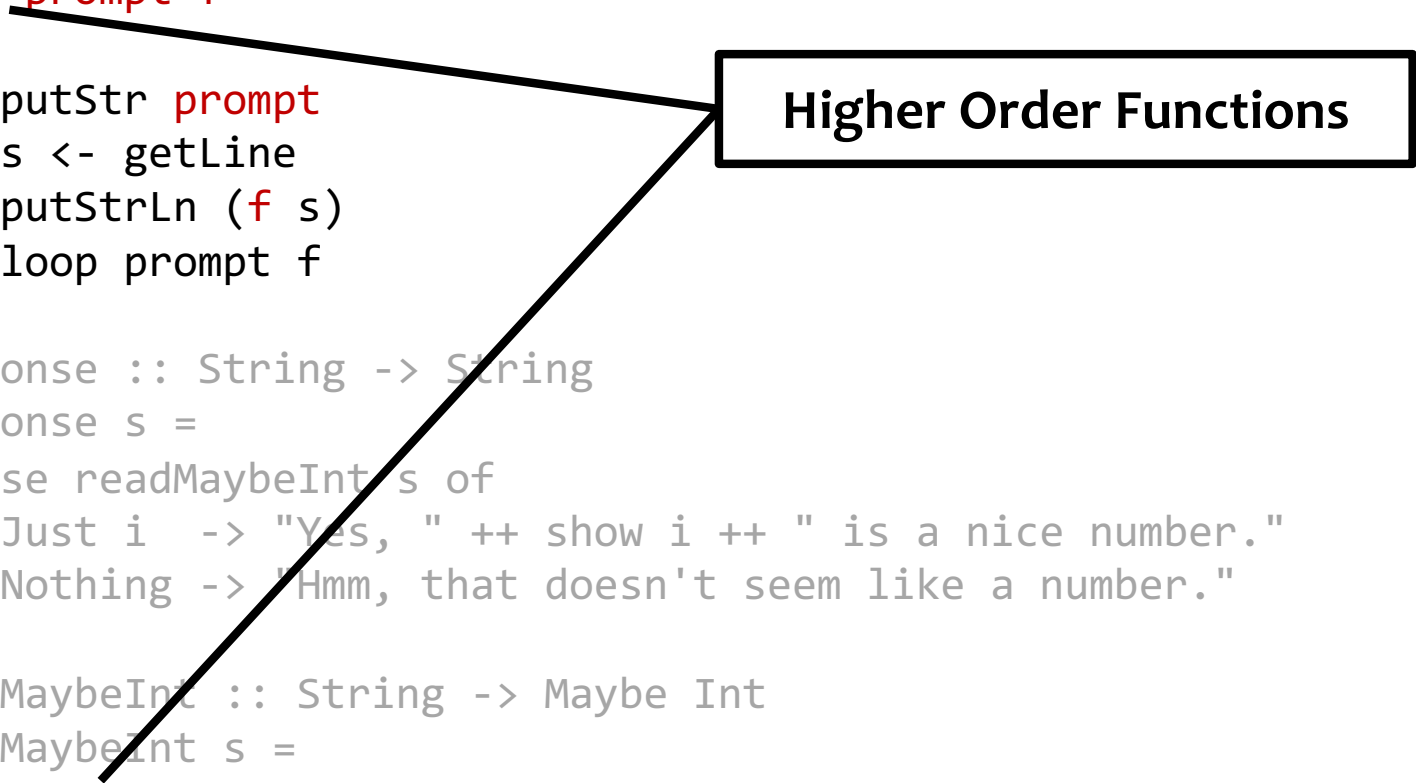
```
readMaybeInt s =
```

```
    if all isDigit s then
```

```
        Just (read s)
```

```
    else
```

```
        Nothing
```



Higher Order Functions

Effectful Code

```
main :: IO ()
main =
  loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
  do
    putStr prompt
    s <- getLine
    putStrLn (f s)
    loop prompt f
```

Pure Functions

```
response :: String -> String
response s =
  case readMaybeInt s of
    Just i  -> "Yes, " ++ show i ++ " is a nice number."
    Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt s =
  if all isDigit s then
    Just (read s)
  else
    Nothing
```

```
main :: IO ()
main =
    loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
    do
        putStr prompt
        s <- getLine
        putStrLn (f s)
        loop prompt f

response :: String -> String
response s =
    case readMaybeInt s of
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
        Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt s =
    if all isDigit s then
        Just (read s)
    else
        Nothing
```

```

main :: IO ()
main =
    loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
    do
        putStr prompt
        s <- getLine
        putStrLn (f s)
        loop prompt f

response :: String -> String
response s =
    case readMaybeInt s of
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
        Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt ""      = Nothing
readMaybeInt ('-':s) = case readMaybeInt s of
    Just i  -> Just (-1 * i)
    Nothing -> Nothing
readMaybeInt s      = if all isDigit s
    then Just (read s)
    else Nothing

```

```
main :: IO ()
main =
    loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
    do
        putStr prompt
        s <- getLine
        putStrLn (f s)
        loop prompt f

response :: String -> String
response s =
    case readMaybeInt s of
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
        Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt "" = Nothing
readMaybeInt ('-':s) = case readMaybeInt s of
    Just i  -> Just (-1 * i)
    Nothing -> Nothing
readMaybeInt s = if all isDigit s
    then Just (read s)
    else Nothing
```

```

main :: IO ()
main =
    loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
    do
        putStr prompt ;
        s <- getLine ;
        putStrLn (f s) ;
        loop prompt f ;

response :: String -> String
response s =
    case readMaybeInt s of
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
        Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt "" = Nothing
readMaybeInt ('-':s) = do
    i <- readMaybeInt s ;
    return (-1 * i) ;

readMaybeInt s = do
    guard (all isDigit s) ;
    return (read s) ;

```

**“Programmable
Semicolons”**

```

main :: IO ()
main =
    loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
    do
        putStr prompt
        s <- getLine
        putStrLn (f s)
        loop prompt f

response :: String -> String
response s =
    case readMaybeInt s of
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
        Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt "" = Nothing
readMaybeInt ('-':s) = do
    i <- readMaybeInt s
    return (-1 * i)
readMaybeInt s = do
    guard (all isDigit s)
    return (read s)

```

```

main :: IO ()
main =
    loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
    do
        putStr prompt
        s <- getLine
        putStrLn (f s)
        loop prompt f

response :: String -> String
response s =
    case readMaybeInt s of
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
        Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt "" = Nothing
readMaybeInt ('-':s) = (\i -> -1 * i) <$> readMaybeInt s
readMaybeInt s = guard (all isDigit s) >> return (read s)

```

```
main :: IO ()
main =
  loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
  do
    putStr prompt
    s <- getLine
    putStrLn (f s)
    loop prompt f

response :: String -> String
response s =
  case readMaybeInt s of
    Just i  -> "Yes, " ++ show i ++ " is a nice number."
    Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt "" = Nothing
readMaybeInt ('-':s) = ((-1)*) <$> readMaybeInt s
readMaybeInt s = guard (all isDigit s) >> return (read s)
```



Operator Overloading++

```
main :: IO ()
main =
    loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
    do
        putStr prompt
        s <- getLine
        putStrLn (f s)
        loop prompt f

response :: String -> String
response s =
    case readMaybeInt s of
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
        Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt "" = Nothing
readMaybeInt ('-':s) = ((-1)*) <$> readMaybeInt s
readMaybeInt s = guard (all isDigit s) >> return (read s)
```

```
main :: IO ()
main =
    loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
    do
        putStr prompt
        putStrLn =<< f <$> getLine
        loop prompt f

response :: String -> String
response s =
    case readMaybeInt s of
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
        Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt "" = Nothing
readMaybeInt ('-':s) = ((-1)*) <$> readMaybeInt s
readMaybeInt s = guard (all isDigit s) >> return (read s)
```

```
main :: IO ()
main =
    loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
    do
        putStr prompt
        putStrLn =<< f <$> getLine
        loop prompt f

response :: String -> String
response s =
    case readMaybeInt s of
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
        Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt ""          = Nothing
readMaybeInt ('-':s)    = ((-1)*) <$> readMaybeInt s
readMaybeInt s          = guard (all isDigit s) >> return (read s)
```

```

import Data.Char
import Control.Monad

main :: IO ()
main =
    loop "Tell me a nice number: " response

loop :: String -> (String -> String) -> IO ()
loop prompt f =
    do
        putStr prompt
        putStrLn =<< f <$> getLine
        loop prompt f

response :: String -> String
response s =
    case readMaybeInt s of
        Just i  -> "Yes, " ++ show i ++ " is a nice number."
        Nothing -> "Hmm, that doesn't seem like a number."

readMaybeInt :: String -> Maybe Int
readMaybeInt ""          = Nothing
readMaybeInt ('-':s) = ((-1)*) <$> readMaybeInt s
readMaybeInt s          = guard (all isDigit s) >> return (read s)

```



Programming in Haskell

Primary Big Ideas

Algebraic Datatypes

Higher-Order Functions

Separation of Church and State



Secondary

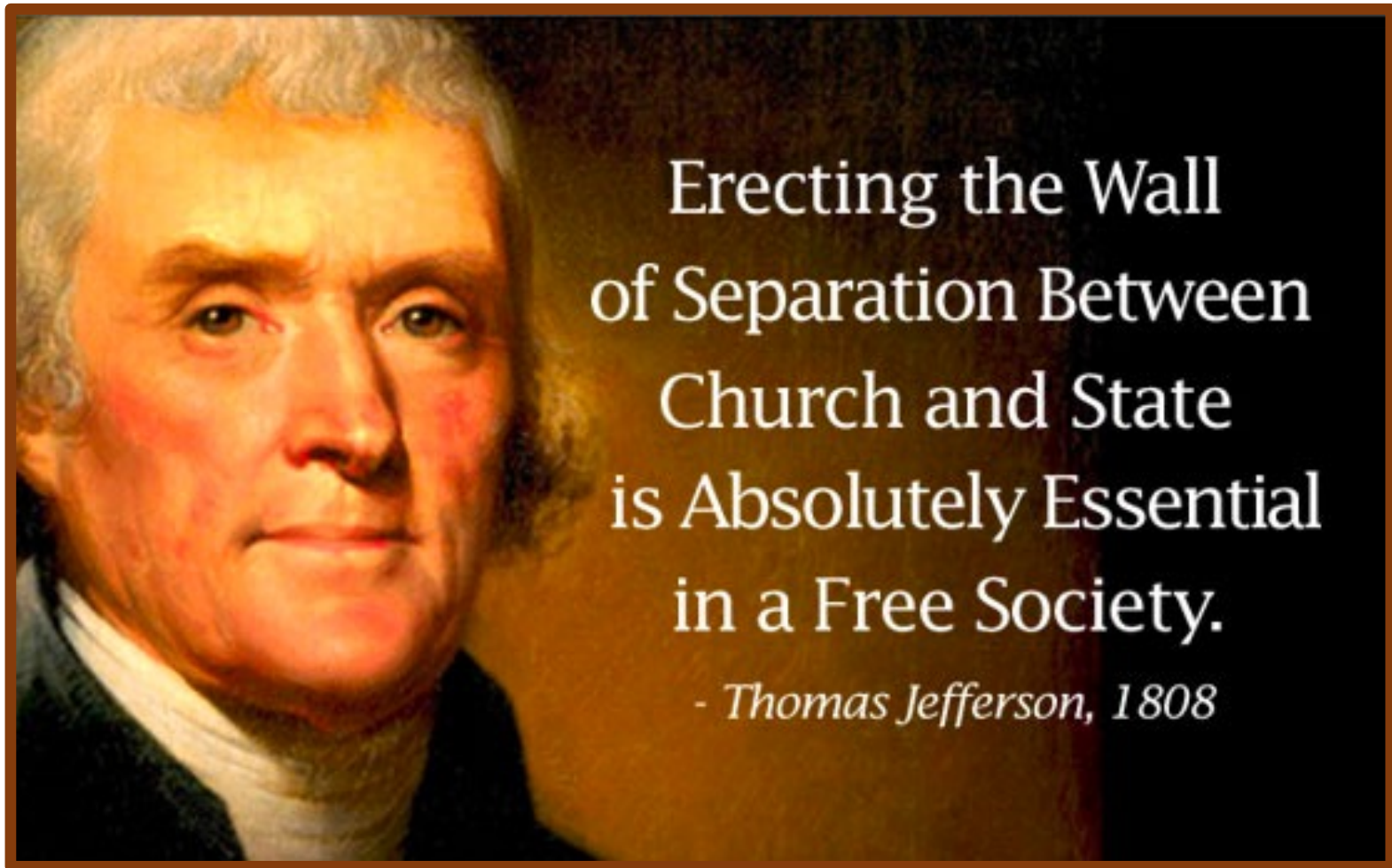
Syntactic Concision

(double-edged sword)

Lazy Evaluation

(ditto)

Separation of Church and State



Separation of Church and State



Erecting the Wall
of Separation Between
Church and State
is Absolutely Essential
in a Functional Program.

- Every Functional Programmer, Always

Disclaimer: This is not an authentic quote from Phil Wadler
<https://www.google.com/search?q=phil+wadler+lambda&tbm=isch>
<https://twitter.com/jeanqasaur/status/1201412242119356416>



https://www.keepcalmandposters.com/poster/1359123_keep_calm_and_do_haskell
https://www.keepcalmandposters.com/poster/5812159_keep_calm_and_learn_haskell
https://www.zazzle.com/keep_calm_and_curry_on_poster-228123322001929170

