Lazy Functional Programming in Haskell

David Raymond Christiansen

25 November, 2013
What is Haskell?

- Pure functional language: no side effects
- Lazy evaluation: function arguments are evaluated when needed
- Rich type system: higher-order types
What is Haskell?

- Pure functional language: no side effects
What is Haskell?

- Pure functional language: no side effects
- Lazy evaluation: function arguments are evaluated when needed
What is Haskell?

- Pure functional language: no side effects
- Lazy evaluation: function arguments are evaluated when needed
- Rich type system: higher-order types
Just the latest hipster fad?
Why Should I Care?

- Get better at functional programming
Why Should I Care?

- Get better at functional programming
- Laboratory for new language features — use the future, today!
Why Should I Care?

- Get better at functional programming
- Laboratory for new language features — use the future, today!
- Very mature compiler and tools
Overview

Hello, World

Basic Syntax

Lazy Evaluation
  Laziness
  Space Leaks

Type Classes
  Overloading
  Examples

Getting Things Done

Computing with State
  Monads in Haskell

Learn More

What’s Next
Overview

HELLO, WORLD

BASIC SYNTAX

LAZY EVALUATION
  Laziness
  Space Leaks

TYPE CLASSES
  Overloading
  Examples

GETTING THINGS DONE

COMPUTING WITH STATE
  Monads in Haskell

LEARN MORE

WHAT’S NEXT
module Main

main :: IO ()
main = do putStrLn "Who are you?"
      name <- getLine
      putStrLn ("Hello, " ++ name)
Fibonacci

\[
\text{fib} :: \text{Int} \rightarrow \text{Int} \\
\text{fib} \ 0 \ = \ 0 \\
\text{fib} \ 1 \ = \ 1 \\
\text{fib} \ n \ = \ \text{fib} \ (n \ - \ 1) \ + \ \text{fib} \ (n \ - \ 2)
\]
Overview

Hello, World

Basic Syntax

Lazy Evaluation
  Laziness
  Space Leaks

Type Classes
  Overloading
  Examples

Getting Things Done

Computing with State
  Monads in Haskell

Learn More

What’s Next
Expressions

- Function application: \( f \ x \ y \ z \)
Expressions

- Function application: \[ f \ x \ y \ z \]
- Anonymous functions: \[ \lambda x \ y \rightarrow x + y \]
Expressions

- Function application: \( f \ x \ y \ z \)
- Anonymous functions: \( \lambda x \ y \to x + y \)
- Infix operators: \( f \ x + 3 \)
Expressions

- Function application: \( f \ x \ y \ z \)
- Anonymous functions: \( \lambda x \ y \rightarrow x + y \)
- Infix operators: \( f \ x + 3 \)
- Infix function application: \( 2 \ ‘elem‘ \ [1, 2, 3] \)
Expressions

- Function application: \( f \ x \ y \ z \)
- Anonymous functions: \( \backslash x \ y \rightarrow x + y \)
- Infix operators: \( f \ x + 3 \)
- Infix function application: \( 2 \text{ 'elem' } [1, 2, 3] \)
- Prefix operator application: \( (+) \ 1 \ 2 \)
Expressions

- Function application: \( f \ x \ y \ z \)
- Anonymous functions: \( \{x \ y \} \rightarrow x + y \)
- Infix operators: \( f \ x + 3 \)
- Infix function application: \( 2 \ ‘elem‘ \ [1, 2, 3] \)
- Prefix operator application: \( (+) \ 1 \ 2 \)
- Operator slices: \( \text{map} \ (+ \ 1) \ [1, 2, 3] \)
Expressions

- Function application: $f \ x \ y \ z$
- Anonymous functions: $\lambda x \ y \ . \ x + y$
- Infix operators: $f \ x + 3$
- Infix function application: $2 \ ‘elem’ \ [1, 2, 3]$
- Prefix operator application: $(+) \ 1 \ 2$
- Operator slices: $\text{map} \ (+ \ 1) \ [1, 2, 3]$
- Function composition: $f \ . \ g = \lambda x \ . \ f \ (g \ x)$
F#:

let rec reverse (lst : 'a list) : 'a list =
  match lst with
  | [] => []
  | (x::xs) => reverse xs @ [x]
F#:

```fsharp
let rec reverse (lst : 'a list) : 'a list =
    match lst with
    | [] => []
    | (x::xs) => reverse xs @ [x]
```

Haskell:

```haskell
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```
reverse' :: [a] -> [a]
reverse' xs = helper [] xs
  where helper acc [] = acc
       helper acc (y:ys) = helper (y:acc) ys

Put the interesting part first!
data Maybe a = Just a
    | Nothing
data Maybe a = Just a
  | Nothing

data List a = Cons a (List a)
  | Nil
data Maybe a = Just a
    | Nothing

data List a = Cons a (List a)
    | Nil

data Tree a = Empty
    | Node (Tree a) a (Tree a)
data Maybe a = Just a
     | Nothing

data List a = Cons a (List a)
     | Nil

data Tree a = Empty
     | Node (Tree a) a (Tree a)

data Either a b = Left a
     | Right b
data Maybe a = Just a  
    | Nothing  

data List a = Cons a (List a)  
    | Nil  

data Tree a = Empty  
    | Node (Tree a) a (Tree a)  

data Either a b = Left a  
    | Right b  

data Labeled a = Labeled String a
Case Sensitivity

UPPERCASE

- Types
- Constructors
- Type constructors
- Modules
UPPERCASE

- Types
- Constructors
- Type constructors
- Modules

LOWERCASE

- Variables
- Type variables
- Defined operators
Haskell is indentation-sensitive (like F#):

```haskell
reverse' :: [a] -> [a]
reverse' xs = helper [] xs
where helper acc [] = acc
       helper acc (y:ys) = helper (y:acc) ys
```
Overview

HELLO, WORLD

BASIC SYNTAX

LAZY EVALUATION
  Laziness
  Space Leaks

TYPE CLASSES
  Overloading
  Examples

GETTING THINGS DONE

COMPUTING WITH STATE
  Monads in Haskell

LEARN MORE

WHAT’S NEXT
Overview

HELLO, WORLD

BASIC SYNTAX

LAZY EVALUATION
  Laziness
  Space Leaks

TYPE CLASSES
  Overloading
  Examples

GETTING THINGS DONE

COMPUTING WITH STATE
  Monads in Haskell

LEARN MORE

WHAT’S NEXT
**Lazy Evaluation**

**Strict Evaluation** evaluates arguments before calling functions.

**Lazy Evaluation** evaluates arguments as they are used.
Strict vs Lazy Evaluation

Strict:

\[
\text{map show . map (+1) . filter (>2)} [1,2,3,4,5]
\]
Strict vs Lazy Evaluation

Strict:

$$(\text{map show} \ . \ \text{map (+1)} \ . \ \text{filter (>2)}) \ [1,2,3,4,5]$$

$$\Rightarrow (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [1,2,3,4,5])))$$
Strict vs Lazy Evaluation

Strict:

\[
(map \ show \ . \ map (+1) \ . \ filter (>2)) \ [1,2,3,4,5]
\]

\[
==> (map \ show \ (map (+1) \ (filter (>2) \ [1,2,3,4,5])))
\]

\[
==> (map \ show \ (map (+1) \ [3,4,5]))
\]

Lazy:

\[
(map \ show \ . \ map (+1) \ . \ filter (>2)) \ [1,2,3,4,5]
\]

\[
==> (map \ show \ (map (+1) \ (filter (>2) \ [1,2,3,4,5])))
\]

\[
==> (map \ show \ (map (+1) \ (filter (>2) \ [2,3,4,5])))
\]

\[
==> (map \ show \ (map (+1) \ (filter (>2) \ [3,4,5])))
\]

\[
==> "4" : (map \ show \ (map (+1) \ (filter (>2) \ [4,5])))
\]

\[
==> "4" : "5" : (map \ show \ (map (+1) \ (filter (>2) \ [5])))
\]

\[
==> "4" : "5" : "6" : (map \ show \ (map (+1) \ (filter (>2) \ [])))
\]

\[
==> ["4","5","6"]
\]
Strict vs Lazy Evaluation

Strict:

\[
\begin{align*}
\text{(map show . map (+1) . filter (>2)) [1,2,3,4,5]} \\
\Rightarrow \text{(map show (map (+1) (filter (>2) [1,2,3,4,5])))} \\
\Rightarrow \text{(map show (map (+1) [3,4,5])}) \\
\Rightarrow \text{(map show [4,5,6])}
\end{align*}
\]
Strict vs Lazy Evaluation

Strict:

(map show . map (+1) . filter (>2)) [1,2,3,4,5]
==> (map show (map (+1) (filter (>2) [1,2,3,4,5])))
==> (map show (map (+1) (filter (>2) [2,3,4,5])))
==> (map show (map (+1) [3,4,5]))
==> (map show [4,5,6])
==> ["4","5","6"]
Strict vs Lazy Evaluation

Strict:

$$(\text{map show} \ . \ \text{map} \ (+1) \ . \ \text{filter} \ (>2)) \ [1,2,3,4,5]$$
$==> (\text{map show} \ (\text{map} \ (+1) \ (\text{filter} \ (>2) \ [1,2,3,4,5])))$$
$==> (\text{map show} \ (\text{map} \ (+1) \ [3,4,5]))$$
$==> (\text{map show} \ [4,5,6])$$
$==> ["4","5","6"]$$

Lazy:

$$(\text{map show} \ . \ \text{map} \ (+1) \ . \ \text{filter} \ (>2)) \ [1,2,3,4,5]$$
Strict vs Lazy Evaluation

Strict:

\[
\text{(map show \ . \ map (+1) \ . \ filter (>2))} \ [1,2,3,4,5]
\]
\[
=> \text{(map show (map (+1) (filter (>2) [1,2,3,4,5])))}
\]
\[
=> \text{(map show (map (+1) [3,4,5])}
\]
\[
=> \text{(map show [4,5,6])}
\]
\[
=> \text{["4","5","6"]}
\]

Lazy:

\[
\text{(map show \ . \ map (+1) \ . \ filter (>2))} \ [1,2,3,4,5]
\]
\[
=> \text{(map show (map (+1) (filter (>2) [1,2,3,4,5])))}
\]
Strict vs Lazy Evaluation

**Strict:**

\[
\text{(map show \ . \ map (+1) \ . \ filter (>2)) \ [1,2,3,4,5]}
\]

\[
\Rightarrow \text{(map show (map (+1) (filter (>2) \ [1,2,3,4,5])))}
\]

\[
\Rightarrow \text{(map show (map (+1) \ [3,4,5])})
\]

\[
\Rightarrow \text{(map show \ [4,5,6])}
\]

\[
\Rightarrow \text{["4","5","6"]}
\]

**Lazy:**

\[
\text{(map show \ . \ map (+1) \ . \ filter (>2)) \ [1,2,3,4,5]}
\]

\[
\Rightarrow \text{(map show (map (+1) (filter (>2) \ [1,2,3,4,5])))}
\]

\[
\Rightarrow \text{(map show (map (+1) (filter (>2) \ [2,3,4,5])))}
\]
Strict vs Lazy Evaluation

Strict:

\[(\text{map show } . \text{ map } (+1) . \text{ filter } (>2)) \ [1,2,3,4,5]\]
\[\Rightarrow (\text{map show } (\text{map } (+1) (\text{filter } (>2) \ [1,2,3,4,5])))\]
\[\Rightarrow (\text{map show } (\text{map } (+1) \ [3,4,5]))\]
\[\Rightarrow (\text{map show } \ [4,5,6])\]
\[\Rightarrow ["4","5","6"]\]

Lazy:

\[(\text{map show } . \text{ map } (+1) . \text{ filter } (>2)) \ [1,2,3,4,5]\]
\[\Rightarrow (\text{map show } (\text{map } (+1) (\text{filter } (>2) \ [1,2,3,4,5])))\]
\[\Rightarrow (\text{map show } (\text{map } (+1) (\text{filter } (>2) \ [2,3,4,5])))\]
\[\Rightarrow (\text{map show } (\text{map } (+1) (\text{filter } (>2) \ [3,4,5])))\]

```
Strict vs Lazy Evaluation

Strict:

(map show . map (+1) . filter (>2)) [1,2,3,4,5]
==> (map show (map (+1) (filter (>2) [1,2,3,4,5])
==> (map show (map (+1) [3,4,5]))
==> (map show [4,5,6])
==> ["4","5","6"]

Lazy:

(map show . map (+1) . filter (>2)) [1,2,3,4,5]
==> (map show (map (+1) (filter (>2) [1,2,3,4,5])))
==> (map show (map (+1) (filter (>2) [2,3,4,5])))
==> (map show (map (+1) (filter (>2) [3,4,5])))
==> "4" : (map show (map (+1) (filter (>2) [4,5])))
Strict vs Lazy Evaluation

Strict:

\[(\text{map show} \ . \ \text{map (+1)} \ . \ \text{filter (>2)}) \ [1,2,3,4,5]\]
\[\Rightarrow (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [1,2,3,4,5])))\]
\[\Rightarrow (\text{map show} \ (\text{map (+1)} \ [3,4,5]))\]
\[\Rightarrow (\text{map show} \ [4,5,6])\]
\[\Rightarrow ["4","5","6"]\]

Lazy:

\[(\text{map show} \ . \ \text{map (+1)} \ . \ \text{filter (>2)}) \ [1,2,3,4,5]\]
\[\Rightarrow (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [1,2,3,4,5])))\]
\[\Rightarrow (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [2,3,4,5])))\]
\[\Rightarrow (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [3,4,5])))\]
\[\Rightarrow "4" : (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [4,5])))\]
\[\Rightarrow "4" : "5" : (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [5])))\]
Strict vs Lazy Evaluation

Strict:

\[(\text{map show} \ . \ \text{map} (+1) . \ \text{filter} (>2)) \ [1,2,3,4,5]\]

\[\Rightarrow (\text{map show} (\text{map} (+1) (\text{filter} (>2) [1,2,3,4,5])))\]

\[\Rightarrow (\text{map show} (\text{map} (+1) [3,4,5]))\]

\[\Rightarrow (\text{map show} [4,5,6])\]

\[\Rightarrow ["4","5","6"]\]

Lazy:

\[(\text{map show} \ . \ \text{map} (+1) . \ \text{filter} (>2)) \ [1,2,3,4,5]\]

\[\Rightarrow (\text{map show} (\text{map} (+1) (\text{filter} (>2) [1,2,3,4,5])))\]

\[\Rightarrow (\text{map show} (\text{map} (+1) (\text{filter} (>2) [2,3,4,5])))\]

\[\Rightarrow (\text{map show} (\text{map} (+1) (\text{filter} (>2) [3,4,5])))\]

\[\Rightarrow "4" : (\text{map show} (\text{map} (+1) (\text{filter} (>2) [4,5])))\]

\[\Rightarrow "4" : "5" : (\text{map show} (\text{map} (+1) (\text{filter} (>2) [5])))\]

\[\Rightarrow "4" : "5" : "6" : (\text{map show} (\text{map} (+1) (\text{filter} (>2) [])))\]
Strict vs Lazy Evaluation

Strict:

\[(\text{map show} \ . \ \text{map (+1)} \ . \ \text{filter (>2)}) \ [1,2,3,4,5]\]
\[\Rightarrow (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [1,2,3,4,5])))\]
\[\Rightarrow (\text{map show} \ (\text{map (+1)} \ [3,4,5]))\]
\[\Rightarrow (\text{map show} \ [4,5,6])\]
\[\Rightarrow ["4","5","6"]\]

Lazy:

\[(\text{map show} \ . \ \text{map (+1)} \ . \ \text{filter (>2)}) \ [1,2,3,4,5]\]
\[\Rightarrow (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [1,2,3,4,5])))\]
\[\Rightarrow (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [2,3,4,5])))\]
\[\Rightarrow (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [3,4,5])))\]
\[\Rightarrow "4" : (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [4,5])))\]
\[\Rightarrow "4" : "5" : (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} \ [5])))\]
\[\Rightarrow "4" : "5" : "6" : (\text{map show} \ (\text{map (+1)} \ (\text{filter (>2)} []))\]
\[\Rightarrow ["4","5","6"]\]
Infinite Data Structures

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

```
zipWith f [] _ = []
zipWith f _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```
Infinite Data Structures

\[
\text{zipWith} :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c] \\
\text{zipWith } f \ [\] \ _ \ = \ [] \\
\text{zipWith } f \ _ \ [\] \ = \ [] \\
\text{zipWith } f \ (x:xs) \ (y:ys) \ = \ f \ x \ y \ : \ \text{zipWith } f \ xs \ ys
\]
Infinite Data Structures

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f [] _ = []
zipWith f _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
Infinite Data Structures

\[\text{fibs} = 0 : 1 : \text{zipWith (+) fibs (tail fibs)}\]
Infinite Data Structures

\[ \text{fibs} = 0 : 1 : \text{zipWith (+) fibs (tail fibs)} \]

\[ \text{take 3 fibs} \]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

take 3 fibs

==> take 3 THUNK
Infinite Data Structures

fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

take 3 fibs
===> take 3 THUNK
===> take 3 (0 : THUNK)
Infinite Data Structures

\[ \text{fibs} = 0 : 1 : \text{zipWith (+) fibs (tail fibs)} \]

take 3 fibs

\[ \Rightarrow \text{take 3 THUNK} \]

\[ \Rightarrow \text{take 3 (0 : THUNK)} \]

\[ \Rightarrow 0 : (\text{take 2 THUNK}) \]
fib\textsc{s} = 0 : 1 : \text{zipWith (+) fibs (tail fibs)}

take 3 fibs

==\rightarrow take 3 \text{ THUNK}

==\rightarrow take 3 (0 : \text{ THUNK})

==\rightarrow 0 : (take 2 \text{ THUNK})

==\rightarrow 0 : 1 : (take 1 \text{ THUNK})
Infinite Data Structures

fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

take 3 fibs
==> take 3 THUNK
==> take 3 (0 : THUNK)
==> 0 : (take 2 THUNK)
==> 0 : 1 : (take 1 THUNK)
==> 0 : 1 : 1 : (take 0 THUNK)
Infinite Data Structures

fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

take 3 fibs

===> take 3 THUNK
===> take 3 (0 : THUNK)
===> 0 : (take 2 THUNK)
===> 0 : 1 : (take 1 THUNK)
===> 0 : 1 : 1 : (take 0 THUNK)
===> 0 : 1 : 1 : []
Infinite Data Structures

fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

take 3 fibs
==> take 3 THUNK
==> take 3 (0 : THUNK)
==> 0 : (take 2 THUNK)
==> 0 : 1 : (take 1 THUNK)
==> 0 : 1 : 1 : (take 0 THUNK)
==> 0 : 1 : 1 : []

After execution,

fibs = 0 : 1 : 1 : THUNK
Simon Peyton Jones: Laziness kept Haskell honest\textsuperscript{1}

- Laziness makes side effects completely unpredictable

\textsuperscript{1}http://research.microsoft.com/en-us/um/people/simonpj/papers/haskell-retrospective/HaskellRetrospective.pdf
Simon Peyton Jones: Laziness kept Haskell honest\textsuperscript{1}

- Laziness makes side effects completely unpredictable
- Thus, Haskell is still pure

\textsuperscript{1}http://research.microsoft.com/en-us/um/people/simonpj/papers/haskell-retrospective/HaskellRetrospective.pdf
Lazy Lists are Control Structures

```scala
var output = ""
for (i <- 1 to 10) {
    output = output + i.toString
}

vs

(concat . take 10 . map show) [1..]
```
Overview

HELLO, WORLD

BASIC SYNTAX

LAZY EVALUATION
  Laziness
  Space Leaks

TYPE CLASSES
  Overloading
  Examples

GETTING THINGS DONE

COMPUTING WITH STATE
  Monads in Haskell

LEARN MORE

WHAT’S NEXT
Dude - where’s my RAM?
Recall \texttt{foldl}:

\begin{verbatim}
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
\end{verbatim}
Space Leaks and Laziness

Recall foldl:

\[
\text{foldl} :: (a \to b \to a) \to a \to \text{[}b\text{]} \to a
\]

\[
\text{foldl}\ f\ z\ [] = z
\]

\[
\text{foldl}\ f\ z\ (x:x:s) = \text{foldl}\ f\ (f\ z\ x)\ x:s
\]

Summing a list:

\[
\text{foldl}\ (+)\ 0\ [1..10]
\]
Recall $\text{foldl}$:

$$\text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$$

$$\text{foldl} f z [] = z$$

$$\text{foldl} f z (x:xs) = \text{foldl} f (f z x) xs$$

Summing a list:

$$\text{foldl} (+) 0 [1..10]$$

$$\Rightarrow \text{foldl} (+) (0 + 1) [2..10]$$
Recall \texttt{foldl}:

\begin{verbatim}
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
\end{verbatim}

Summing a list:

\begin{verbatim}
foldl (+) 0 [1..10]
==&gt; foldl (+) (0 + 1) [2..10]
==&gt; foldl (+) ((0 + 1) + 2) [3..10]
\end{verbatim}
Space Leaks and Laziness

Recall foldl:

\[
\text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a
\]

\[
\text{foldl } f \ z \ [] \ = \ z
\]

\[
\text{foldl } f \ z \ (x:x:\text{xs}) \ = \ \text{foldl } f \ (f \ z \ x) \ \text{xs}
\]

Summing a list:

\[
\text{foldl } (+) \ 0 \ [1..10]
\]

\[
\Rightarrow \ \text{foldl } (+) \ ((0 + 1) \ [2..10])
\]

\[
\Rightarrow \ \text{foldl } (+) \ (((0 + 1) + 2) \ [3..10])
\]

\[
\Rightarrow \ \text{foldl } (+) \ (((((0 + 1) + 2) + 3) \ [4..10])
\]
Space Leaks and Laziness

Recall \texttt{foldl}: 

\[
\text{foldl} \ : \ (a \to b \to a) \to a \to [b] \to a
\]

\[
\text{foldl} \ f \ z \ [] = z
\]

\[
\text{foldl} \ f \ z \ (x:xs) = \text{foldl} \ f \ (f \ z \ x) \ xs
\]

Summing a list:

\[
\text{foldl} \ (+) \ 0 \ [1..10]
\]

\[
\Rightarrow \text{foldl} \ (+) \ (0 + 1) \ [2..10]
\]

\[
\Rightarrow \text{foldl} \ (+) \ (((0 + 1) + 2) \ [3..10]
\]

\[
\Rightarrow \text{foldl} \ (+) \ (((((0 + 1) + 2) + 3) \ [4..10]
\]

\[
\ldots
\]

\[
\Rightarrow \text{foldl} \ (+) \ (((((0 + 1) + 2) + 3) + 4) + \ldots \ 10) \ []
\]
Recall \texttt{foldl}:

\begin{verbatim}
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []    = z
foldl f z (x:xs) = foldl f (f z x) xs
\end{verbatim}

Summing a list:

\begin{verbatim}
foldl (+) 0 [1..10]
==> foldl (+) (0 + 1) [2..10]
==> foldl (+) ((0 + 1) + 2) [3..10]
==> foldl (+) (((0 + 1) + 2) + 3) [4..10]
...  
==> foldl (+) ((((0 + 1) + 2) + 3) + 4) + ... 10) []
\end{verbatim}

Code to perform repeated additions takes much more space than the answer.
▶ Strictness annotations

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
➤ Strictness annotations
➤ Don’t be too tail-recursive: consider \texttt{foldr}:

\[
\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]
\[
\text{foldr } f \ z \ [] = z
\]
\[
\text{foldr } f \ z \ (x:xs) = f \ x \ (\text{foldr } f \ z \ xs)
\]
Strictness annotations

Don’t be too tail-recursive: consider \( \text{foldr} \):

\[
\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]

\[
\text{foldr} f z [] = z
\]

\[
\text{foldr} f z (x:xs) = f x (\text{foldr} f z xs)
\]

Use lazy containers with strict contents
Laziness Summary

**ADVANTAGES**

**CODE RE-USE** by not committing to a strategy too early

**PERFORMANCE** by not wasting work

**DISADVANTAGES**

**PURITY** is required

**SPACE USAGE** is difficult to reason about

**DEBUGGERS** are more difficult to implement
Overview

HELLO, WORLD

BASIC SYNTAX

LAZY EVALUATION
  Laziness
  Space Leaks

TYPE CLASSES
  Overloading
  Examples

GETTING THINGS DONE

COMPUTING WITH STATE
  Monads in Haskell

LEARN MORE

WHAT’S NEXT

IT UNIVERSITY OF COPENHAGEN
Overview

HELLO, WORLD

BASIC SYNTAX

LAZY EVALUATION
  Laziness
  Space Leaks

TYPE CLASSES
  Overloading
  Examples

GETTING THINGS DONE

COMPUTING WITH STATE
  Monads in Haskell

LEARN MORE

WHAT’S NEXT
data Tree a = Empty | Node (Tree a) a (Tree a)
data Tree a = Empty | Node (Tree a) a (Tree a)

insert :: Tree a -> a -> Tree a
insert Empty x = Node Empty x Empty
insert (Node l y r) x = if x <= y
  then Node (insert x l) y r
  else Node l y (insert x r)
data Tree a = Empty | Node (Tree a) a (Tree a)

insert :: Tree a -> a -> Tree a
insert Empty x = Node Empty x Empty
insert (Node l y r) x = if x <= y
    then Node (insert x l) y r
    else Node l y (insert x r)

What does this mean?
abstract class Tree[A]
case class Empty[A]() extends Tree[A]
case class Node[A](
    left: Tree[A],
    x : A,
    right: Tree[A]
) extends Tree[A]
abstract class Tree[A]
case class Empty[A]() extends Tree[A]
case class Node[A](
    left: Tree[A],
    x : A,
    right: Tree[A]
) extends Tree[A]

def insert[A <: Ordered[A]](t : Tree[A], x : A): Tree[A] =
t match {
    case Empty() => Node(Empty(), x, Empty()
    case Node(l, y, r) => if (x <= y)
        Node(insert(l, x), y, r)
    else
        Node(l, y, insert(r, x))
}

Type Classes

For a given type, provide:

- Operations on the type
- Default implementations of some operations

```haskell
data Ordering = LT | EQ | GT

class Ord a where
    compare :: a -> a -> Ordering
    (>):= a -> a -> Bool
    a > b = case compare a b of
             GT -> True
             _  -> False
    ...
```
data Tree a = Empty | Node (Tree a) a (Tree a)

insert :: (Ord a) => Tree a -> a -> Tree a
insert Empty x = Node Empty x Empty
insert (Node l y r) x = if x <= y
    then Node (insert x l) y r
    else Node l y (insert x r)
Overview

HELLO, WORLD

BASIC SYNTAX

LAZY EVALUATION
- Laziness
- Space Leaks

TYPE CLASSES
- Overloading
- Examples

GETTING THINGS DONE

COMPUTING WITH STATE
- Monads in Haskell

LEARN MORE

WHAT’S NEXT
class Eq a where
  (==) : a -> a -> Bool
  x == y = not (x /= y)
  (/=) : a -> a -> Bool
  x /= y = not (x == y)
class Eq a where
  (==) : a -> a -> Bool
  x == y = not (x /= y)
  (/=) : a -> a -> Bool
  x /= y = not (x == y)

Default implementations: clients need only define enough!
class (Eq a) => Ord a where
    compare :: a -> a -> Ordering
    (<=), (>=), (>), (<) :: a -> a -> Bool
    max, min :: a -> a -> a

    -- plus default methods...
Ord

class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a

  -- plus default methods...

Superclass constraint: all Ord must have Eq
class Num a where  
    (+), (*) :: a -> a -> a  
    neg :: a -> a  

instance Num Int where  
    (+) = addInt  
    (*) = mulInt  
    neg = negInt

instance Num Float where  
    (+) = addFloat  
    (*) = mulFloat  
    neg = negFloat

sumSquares :: (Num a) => a -> a -> a  
sumSquares x y = x * x + y * y
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a)

plus (NumDict p t n) = p

times (NumDict p t n) = t

neg (NumDict p t n) = n

instance Num Int where
  (+) = addInt
  (*) = mulInt
  neg = negInt

instance Num Float where
  (+) = addFloat
  (*) = mulFloat
  neg = negFloat

sumSquares :: (Num a) => a -> a -> a
sumSquares x y = x * x + y * y
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a)  
plus (NumDict p t n) = p  
times (NumDict p t n) = t  
neg (NumDict p t n) = n

numInt = NumDict addInt mulInt negInt

instance Num Float where  
  (+) = addFloat  
  (*) = mulFloat  
  neg = negFloat

sumSquares :: (Num a) => a -> a -> a  
sumSquares x y = x * x + y * y
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a) 
plus (NumDict p t n) = p 
times (NumDict p t n) = t 
neg (NumDict p t n) = n

numInt = NumDict addInt mulInt negInt

numFloat = NumDict addFloat mulFloat negFloat

sumSquares :: (Num a) => a -> a -> a 
sumSquares x y = x * x + y * y
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a)
  plus (NumDict p t n) = p
  times (NumDict p t n) = t
  neg (NumDict p t n) = n

numInt = NumDict addInt mulInt negInt

numFloat = NumDict addFloat mulFloat negFloat

sumSquares :: (Num a) -> a -> a -> a
  sumSquares x y = x * x + y * y
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a) 
  plus (NumDict p t n) = p 
  times (NumDict p t n) = t 
  neg (NumDict p t n) = n 

numInt = NumDict addInt mulInt negInt 

numFloat = NumDict addFloat mulFloat negFloat 

sumSquares :: (Num a) -> a -> a -> a 
  sumSquares dict x y = plus dict (times dict x y) 
                          (times dict x y)
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a)  
  plus (NumDict p t n) = p  
  times (NumDict p t n) = t  
  neg (NumDict p t n) = n

numInt = NumDict addInt mulInt negInt

numFloat = NumDict addFloat mulFloat negFloat

sumSquares :: (Num a) -> a -> a -> a
sumSquares dict x y = plus dict (times dict x y)  
  (times dict x y)
Overview

HELLO, WORLD

BASIC SYNTAX

LAZY EVALUATION
  Laziness
  Space Leaks

TYPE CLASSES
  Overloading
  Examples

GETTING THINGS DONE

COMPUTING WITH STATE
  Monads in Haskell

LEARN MORE

WHAT’S NEXT
Overview

HELLO, WORLD

BASIC SYNTAX

LAZY EVALUATION
  Laziness
  SpaceLeaks

TYPE CLASSES
  Overloading
  Examples

GETTING THINGS DONE

COMPUTING WITH STATE
  Monads in Haskell

LEARN MORE

WHAT’S NEXT
Generating labels:

let lastlab = ref (-1)
newLabel () =
    lastlab := !lastlab + 1
    "label" + (!lastlab.ToString())
Return a new state after updating:

\[
\text{startLabel} = -1 \\
\text{newLabel old} = \\
\quad \text{let next} = \text{old} + 1 \\
\quad ("label" ++ \text{show next, next})
\]
Return a new state after updating:

```haskell
startLabel = -1
newLabel old =
    let next = old + 1
    ("label" ++ show next, next)
```

- Easy to forget
Return a new state after updating:

```
startLabel = -1
newLabel old =
  let next = old + 1
  ("label" ++ show next, next)
```

- Easy to forget
- Difficult to read
Return a new state after updating:

```haskell
startLabel = -1
newLabel old =
  let next = old + 1
  ("label" ++ show next, next)
```

- Easy to forget
- Difficult to read
- Risk of “time travel”
data WithLabels a = Labeled (Int -> (a, Int))
data WithLabels a = Labeled (Int -> (a, Int))

only :: a -> WithLabels a
only x = Labeled (\i -> (x, i))
data WithLabels a = Labeled (Int -> (a, Int))

only :: a -> WithLabels a
only x = Labeled (\i -> (x, i))

get :: WithLabels Int
get = Labeled (\i -> (i, i))
data WithLabels a = Labeled (Int -> (a, Int))

only :: a -> WithLabels a
only x = Labeled (\i -> (x, i))

get :: WithLabels Int
get = Labeled (\i -> (i, i))

set :: Int -> WithLabels ()
set i = Labeled (\j -> (() , i))
data WithLabels a = Labeled (Int -> (a, Int))

only :: a -> WithLabels a
only x = Labeled (\i -> (x, i))

define get :: WithLabels Int
define get = Labeled (\i -> (i, i))

define set :: Int -> WithLabels ()
define set i = Labeled (\j -> ((), i))

define run :: Int -> WithLabels a -> a
run i (Labeled comp) = fst (comp i)
data WithLabels a = Labeled (Int -> (a, Int))

only :: a -> WithLabels a
only x = Labeled (\i -> (x, i))

get :: WithLabels Int
get = Labeled (\i -> (i, i))

set :: Int -> WithLabels ()
set i = Labeled (\j -> ((), i))

run :: Int -> WithLabels a -> a
run i (Labeled comp) = fst (comp i)
Threading State

\[
\text{step} :: \text{WithLabels} \ a \\
\quad \rightarrow (a \rightarrow \text{WithLabels} \ b) \\
\quad \rightarrow \text{WithLabels} \ b
\]
step :: WithLabels a
    -> (a -> WithLabels b)
    -> WithLabels b
step (Labeled comp) next =
    Labeled (\i -> let (x, j) = comp i
        (Labeled cont) = next x
        in cont j)
Generating Labels, purely

getLabel :: WithLabels String
getLabel = get

-- At REPL
> run 0 (getLabel 'step' (\-> getLabel))
"label1"
Generating Labels, purely

getLabel :: WithLabels String
getLabel = get ‘step‘ (\i ->
    let lab = "label" ++ show i in
    set (i + 1)
Generating Labels, purely

```haskell
getLabel :: WithLabels String
getLabel = get "step" (\i ->
    let lab = "label" ++ show i in
    set (i + 1) "step" (\() ->
        only lab))
```

```haskell
> run 0 (getLabel "step" (\ -> getLabel))
"label1"
```
Generating Labels, purely

getLabel :: WithLabels String
getLabel = get 'step' (i ->
    let lab = "label" ++ show i in
    set (i + 1) 'step' (() ->
        only lab))

-- At REPL
> run 0 (getLabel 'step' (_ -> getLabel))
"label1"
The code

getLabel :: WithLabels String
getLabel = get 'step' (
  \i ->
    let lab = "label" ++ show i in
    set (i + 1) 'step' (\() ->
      only lab)

is much better when written:

getLabel :: WithLabels String
getLabel = do i <- get
  let lab = "label" ++ show i
  set (i + 1)
  return lab
Overview

HELLO, WORLD

BASIC SYNTAX

LAZY EVALUATION
  Laziness
  Space Leaks

TYPE CLASSES
  Overloading
  Examples

GETTING THINGS DONE

COMPUTING WITH STATE
  Monads in Haskell

LEARN MORE

WHAT’S NEXT
class Monad m where
  return : a -> m a
  (>>=) : m a -> (a -> m b) -> m b
  (>>) : m a -> m b -> m b
  x >> y = x >>= (\_ -> y)

m has kind * -> *

(>>=) looks like the Scala signature
def flatMap[B](f: A => C[B]): C[B]

Laws:
- Left identity: return x >>= f ≡ f x
- Right identity: x >>= return ≡ x
- Associativity: (x >>= f) >>= g ≡ x >>= (\i -> f i >>= g)
class Monad m where
  return : a -> m a
  (>>=) : m a -> (a -> m b) -> m b
  (>>) : m a -> m b -> m b
  x >> y = x >>= (_ -> y)

  m has kind * -> *

(>>=) looks like the Scala signature
  def flatMap[B](f: A => C[B]): C[B]

Laws:
- Left identity:
  return x >>= f ≡ f x
- Right identity:
  x >>= return ≡ x
- Associativity:
  (x >>= f) >>= g ≡ x >>= (\i -> f i >>= g)
class Monad m where
    return : a -> m a
    (>>=) : m a -> (a -> m b) -> m b
    (>>)  : m a -> m b -> m b
    x >> y = x >>= (\_ -> y)

- m has kind * -> *
- (>>=) looks like the Scala signature
  def flatMap[B](f: A => C[B]): C[B]
class Monad m where
    return : a -> m a
    (>>=) : m a -> (a -> m b) -> m b
    (>>)  : m a -> m b -> m b
    x >> y = x >>= (\_ -> y)

- m has kind * -> *
- (>>=) looks like the Scala signature
  def flatMap[B](f: A => C[B]): C[B]
- Laws:
  - Left identity: return x >>= f ≡ f x
class Monad m where
  return : a -> m a
  (>>=) : m a -> (a -> m b) -> m b
  (>>) : m a -> m b -> m b
  x >> y = x >>= (_ -> y)

- m has kind * -> *
- (>>=) looks like the Scala signature
  
  def flatMap[B](f: A => C[B]): C[B]

- Laws:
  - Left identity: return x >>= f ≡ f x
  - Right identity: x >>= return ≡ x
class Monad m where
    return : a -> m a
    (>>=) : m a -> (a -> m b) -> m b
    (>>) : m a -> m b -> m b
    x >> y = x >>= (
       _ -> y)

m has kind * -> *

(>>=) looks like the Scala signature
def flatMap[B](f: A => C[B]): C[B]

Laws:
  - Left identity: return x >>= f ≡ f x
  - Right identity: x >>= return ≡ x
  - Associativity:
    (x >>= f) >>= g ≡ x >>= (\i -> f i >>= g)
Desugaring "do"

\[
\text{do } x \implies x
\]
Desugaring "do"

\[
\begin{align*}
\text{do } x & \quad \Rightarrow \quad x \\
\text{do } x \\
\text{rest} & \quad \Rightarrow \quad x \gg \text{do } y
\end{align*}
\]
Desugaring "do"

\[
\begin{align*}
\text{do } x & \quad \Rightarrow \quad x \\
\text{do } x \\
\quad \text{rest} & \quad \Rightarrow \quad x \gg do \ y \\
\text{do } i <- x \\
\quad \text{rest} & \quad \Rightarrow \quad x \ggg (\lambda i -> do \ rest)
\end{align*}
\]
Desugaring "do"

\[
\text{do } x \quad \Rightarrow \quad x
\]

\[
\text{do } x \\
\quad \text{rest} \quad \Rightarrow \quad x \gg \text{do } y
\]

\[
\text{do } i \leftarrow x \\
\quad \text{rest} \quad \Rightarrow \quad x \ggg (\lambda i \rightarrow \text{do } \text{rest})
\]

\[
\text{do } \text{let } i = x \\
\quad \text{rest} \quad \Rightarrow \quad \text{let } i = x \text{ in do rest}
\]
WithLabels is a Monad!

instance Monad WithLabels where
  return = only
  (>>=) = step
Desugaring getLabel

getLabel :: WithLabels String
getLabel = do i <- get
    let lab = "label" ++ show i
    set (i + 1)
    return lab

getLabel :: WithLabels String
getLabel = get >>= (\i ->
    let lab = "label" ++ show i in
    set (i + 1) >>= return lab)
Example Monads

```
instance Monad [] where
    return x = [x]
    [] >>= f = []
    (x:xs) >>= f = f x ++ (xs >>= f)
```

Example Monads

instance Monad [] where
    return x     = [x]
    []           >>= f = []
    (x:xs) >>= f = f x ++ (xs >>= f)

instance Monad Maybe where
    return x     = Just x
    (Just x) >>= f = f x
    Nothing      >>= f = Nothing
instance Monad [] where
  return x = [x]
  [] >>= f = []
  (x:xs) >>= f = f x ++ (xs >>= f)

instance Monad Maybe where
  return x = Just x
  (Just x) >>= f = f x
  Nothing >>= f = Nothing

instance Monad (Either a) where
  return x = Right x
  (Left err) >>= f = Left err
  (Right x) >>= f = f x
I/O takes the world as a parameter and constructs a new world!
Use a monad to thread the world through computation:

data IO a = MkIO (World -> (a, World))

instance Monad IO where
    return x = MkIO (\w -> (x, w))
    (MkIO x) >>= y =
        MkIO (\w -> let (res, newWorld) = x w
                  (MkIO cont) = y res
                  in cont newWorld)

User code can’t pattern-match MkIO — it only has return and ( >>= ) — so no shenanigans with the World.
main :: IO ()
main = do putStrLn "Who are you?"
        name <- getLine
        putStrLn ("Hello, " ++ name)
main :: IO ()
main = do putStrLn "Who are you?"
        name <- getLine
        putStrLn ("Hello, " ++ name)

main :: IO ()
main = putStrLn >>= (\_ ->
    getline >>= (\name ->
        putStrLn ("Hello, " ++ name)))

Not a world in sight...
Monad Libraries

\[
\text{mapM} \::\: \text{Monad } m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ [b] \\
\text{mapM\_} \::\: \text{Monad } m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ ()
\]
Monad Libraries

mapM :: Monad m => (a -> m b) -> [a] -> m [b]

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()

sequence :: Monad m => [m a] -> m [a]

sequence_ :: Monad m => [m a] -> m ()
mapM :: Monad m => (a -> m b) -> [a] -> m [b]

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()

sequence :: Monad m => [m a] -> m [a]

sequence_ :: Monad m => [m a] -> m ()

when :: Monad m => Bool -> m () -> m ()
LINQ is inspired by Haskell monads:

```csharp
public static IEnumerable<B> SelectMany<A, B>(
    this IEnumerable<A> source,
    Func<A, IEnumerable<B>> selector
)
```

vs

```haskell
(>>=) :: m a -> (a -> m b) -> m b
```
Higher-Order Types at Work

- .NET loses type information — Select and SelectMany return IEnumerable
- Monad preserves type information
- Being a monad is a property of *type constructors*, while being IEnumerable is a merely property of *types*. 
Overview

Hello, world

Basic Syntax

Lazy Evaluation
  Laziness
  Space Leaks

Type Classes
  Overloading
  Examples

Getting Things Done

Computing with State
  Monads in Haskell

Learn More

What’s Next
RESOURCES

- *Learn You a Haskell for Great Good*
- https://www.fpcomplete.com/school
- Hoogle lets you search libraries by type

PROJECT POSSIBILITIES

Idris is an advanced open-source functional language with a compiler written in Haskell. We’re a good source of projects!
Overview

Hello, World

Basic Syntax

Lazy Evaluation
  Laziness
  Space Leaks

Type Classes
  Overloading
  Examples

Getting Things Done

Computing with State
  Monads in Haskell

Learn More

What’s Next
What’s Next

- Today, 13:00-13:50: Demonstration in Aud. 3 of starting project with fsyacc and fslex, from scratch (Ahmad is in 2A54)
- Spørgetime on Friday, December 20 from 10 AM to noon in Aud. 3
- Exam: may need to be turned in on paper — news to follow.
- Assignment deadline is Wednesday, December 4