HANDS-ON

Haskell Introduction

Erik Hesselink
Sebastiaan Visser

October 24-25th, 2013 | Murray Edwards College, Cambridge UK

Silk

http://silk.co

Silk Is Hiring
Functional Programmers!

Preliminaties

Haskell platform:
compiler (GHC) and libraries.

GHCi: interactive environment (REPL).

Preliminaties

Start GHCi:

```
> ghci
GHCi, version 7.6.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

Preliminaties

Start GHCi:

```
> ghci
GHCi, version 7.6.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

Supports command line editing (readline).

Numbers

```
> 1
1
> 1 + 1
2
> 2 * (3.1 - 1)
4.2
```

Also: `(-)`, `div`, `mod`, `(/)`, ...

Numbers

```
> 1
1
> 1 + 1
2
> 2 * (3.1 - 1)
4.2
```

Also: `(-)`, `div`, `mod`, `(/)`, ...

Q: what is 56088 divided by 456?

Bindings

```
> let x = 1
> x + x
2
```

Bindings

```
> let x = 1
> x + x
2
```

GHCi only; In a file, just use `x = 1`.

Bindings

```
> let x = 1
> x + x
2
```

GHCi only; In a file, just use `x = 1`.

Non-mutable; Re-binding shadows.

Files

```
x = 1
y = x + 2
```

Save this with extension `.hs`.

Load in GHCi with `ghci <filename>`

Reload with `:r`

Booleans

```
> True
True
> not True
False
> True && (False || True)
True
```

Application

```
> not True
False
```

Application with space.

Application

```
> not True
False
```

Application with space.

```
> min 1 2
1
```

Multiple arguments with another space.

If-Then-Else

```
> if True then 1 else 2
1
```

This is an *expression*:
you cannot leave out a branch.

C.f. ternary operator.

Types

```
> :t True
True :: Bool
> :t not
not :: Bool -> Bool
```

Types

```
> :t True
True :: Bool
> :t not
not :: Bool -> Bool
```

Q: What is the type of `(&&)`?

Bonus: Look at the type of `(+)`.

Type Errors

```
> if not then True else False
```

```
<interactive>:1:4:
    Couldn't match expected type `Bool'
      with actual type `Bool -> Bool'
    In the expression: not
    In the expression:
      if not then True else False
```

Type Classes

```
> :t 1
1 :: Num a => a
```

Type Classes

```
> :t 1
1 :: Num a => a
```

`Num` is a *type class*.

Says: `1` can be any type that is numeric.

Type Classes

```
> :t 1
1 :: Num a => a
```

`Num` is a *type class*.

Says: `1` can be any type that is numeric.

Q: Explain the type of (+).

Type Class Errors

```
> 1 + True
```

```
<interactive>:25:3:
    No instance for (Num Bool) arising from
      a use of `+'
    Possible fix: add an instance declaration
      for (Num Bool)
    In the expression: 1 + True
    In an equation for `it': it =
```

Number Types

Integral:	`Integer, Int`
Floating:	`Double, Float`
Fractional:	`Rational`
Fixed size:	`Int8 ... Int64,` `Word8 ... Word64`

Characters And Strings

```
> :t 'c'
'c' :: Char
> :t "Hello world"
"Hello World" :: String
> "FP" ++ "days"
"FPdays"
```

Characters And Strings

```
> :t 'c'
'c' :: Char
> :t "Hello world"
"Hello World" :: String
> "FP" ++ "days"
"FPdays"
```

Q: What is the type of "Hello World" really?

Lists

```
> :t [True, True, False]
[True, True, False] :: [Bool]
> :t []
[] :: [a]
> 1 : [2,3]
[1,2,3]
> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

Lists

```
> :t [True, True, False]
[True, True, False] :: [Bool]
> :t []
[] :: [a]
> 1 : [2,3]
[1,2,3]
> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

Q: Try to make a list containing one, two and false.

List Functions

```
> length [1,4,7]
3
```

List Functions

```
> length [1,4,7]
3
```

```
> take 3 [1..10]
[1,2,3]
```

List Functions

```
> length [1,4,7]
3
```

```
> take 3 [1..10]
[1,2,3]
```

```
> drop 3 [10,9..1]
[7,6,5,4,3,2,1]
```

List Functions - 2

```
> head [4,5,6]
4
```

List Functions - 2

```
> head [4,5,6]
4
```

```
> tail [4,5,6]
[5,6]
```

List Functions - 2

```
> head [4,5,6]
4
```

```
> tail [4,5,6]
[5,6]
```

```
> elem 3 [1,2,4]
False
```

Infinite Lists

```
> head [1..]
1
```

Infinite Lists

```
> head [1..]
1
```

```
> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
```

Infinite Lists

```
> head [1..]
1
```

```
> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
```

```
> head (tail (repeat 2))
2
```

Tuples

```
> :t (True, 'c')
(True, 'c') :: (Bool, Char)
```

```
> :t (True, 'c', "Hello")
(True, 'c', "Hello") :: (Bool, Char, [Char])
```

Lists are homogeneous, variable length.

Tuples are heterogeneous, fixed length.

Tuple Functions

```
> fst (True, 'c')
True
```

```
> snd (True, 'c')
'c'
```

Equality

```
> 1 == 1
True
> [1,2,3] /= [1,2,4]
True
```

Equality

```
> 1 == 1
True
> [1,2,3] /= [1,2,4]
True
```

Works on most types.

Q: what is the type of `(==)`?

Comparison

```
> 1 > 2
False
> [1,2,3] < [1,2,4]
True
```

Also `(<=), (>=)`.

Comparison

```
> 1 > 2
False
> [1,2,3] < [1,2,4]
True
```

Also `(<=), (>=)`.

```
> :t (<)
(<) :: Ord a => a -> a -> Bool
```

Functions

Functions

```
upper c = (c >= 'A') && (c <= 'Z')
```

Functions

```haskell
upper c = (c >= 'A') && (c <= 'Z')
```

```
> :t upper
upper :: Char -> Bool
```

Character as input,
boolean as output.

Functions

```haskell
upper :: Char -> Bool
upper c = (c >= 'A') && (c <= 'Z')
```

Functions

```haskell
upper :: Char -> Bool
upper c = (c >= 'A') && (c <= 'Z')
```

Type signature is optional,
but recommended.

Function Application

```
> upper 'a'
False
```

Just use a space!

Application Precedence

Function application binds stronger than operators.

Application Precedence

so:

```
not (1 > 2)
```

and not:

~~`not 1 > 2`~~

Application Precedence

so:

```
not (1 > 2)
```

and not:

~~`not 1 > 2`~~

Q: What happens when you try `not 1 > 2`?

Multiple Arguments

```
both a b = upper a && upper b
```

Multiple Arguments

```
both a b = upper a && upper b
```

```
> :t both
both :: Char -> Char -> Bool
```

Multiple arguments,
multiple arrows in type.

Currying

```
> both 'a' 'X'
False
```

Currying

```
> both 'a' 'X'
False
```

```
> (both 'a') 'X'
False
```

Currying

```
> both 'a' 'X'
False
```

```
> (both 'a') 'X'
False
```

```
> :t both 'a'
both 'a' :: Char -> Bool
```

Currying

```
both :: Char ->  Char -> Bool
both :: Char -> (Char -> Bool)
```

Application associates to the left,
function arrows to the right.

Currying

Q: What would be the type of:

```
side :: ???
side c = if c
         then head
         else last
```

```
head, last :: [Char] -> Char
```

Currying

Q: What would be the type of:

```
side :: Bool -> ([Char] -> Char)
side c = if c
         then head
         else last
```

```
head, last :: [Char] -> Char
```

Currying

Q: What would be the type of:

```
side :: Bool -> [Char] -> Char
side c = if c
         then head
         else last
```

```
head, last :: [Char] -> Char
```

Currying

Q: What would be the type of:

```
side :: Bool -> [Char] -> Char
side c a = if c
             then head a
             else last a
```

```
head, last :: [Char] -> Char
```

Lambda Expressions

```
both a b = upper a && upper b
```

```
???
```

```
both = \a b -> upper a && upper b
```

Lambda Expressions

```
both a b = upper a && upper b
```

```
both a = \b -> upper a && upper b
```

```
both = \a b -> upper a && upper b
```

All equivalent.

Operators Are Functions

```
> True && False
False
```

Operators Are Functions

```
> True && False
False
```

```
> (&&) True False
False
```

Operators Are Functions

```
> True && False
False
```

```
> (&&) True False
False
```

```
> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

Just Functions

```
(&&) :: Bool -> Bool -> Bool
(&&) True  True  = True
(&&) False True  = False
(&&) True  False = False
(&&) False False = False
```

Just Functions

```
(&&) :: Bool -> Bool -> Bool
(&&) True  True  = True
(&&) _     _     = False
```

Implementation

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_    && _    = False
```

Operator Sections

```
> (True &&) False
False
```

```
> :t (True &&)
(True &&) :: Bool -> Bool
```

Operator Sections

```
> (&& False) True
False
```

```
> :t (&& False)
(&& False) :: Bool -> Bool
```

Infix Functions

```
> elem 20 [10, 20, 30]
True
```

Infix Functions

```
> elem 20 [10, 20, 30]
True
```

Similarly:

```
> 20 `elem` [10, 20, 30]
True
```

Functions Are Values

```
funs = [ upper
       , (== 'a')
       , (`elem` "xyz")
       ]
```

Q: What is the type?

Functions Are Values

```haskell
funs = [ upper
       , (== 'a')
       , (`elem` "xyz")
       ]
```

Q: What is the type?

```
> :t funs
funs :: [Char -> Bool]
```

Polymorphism

```haskell
swap t = (snd t, fst t)
```

```
> swap (1, 2)
(2,1)
```

Polymorphism

```haskell
swap t = (snd t, fst t)
```

```
> swap (1, 2)
(2,1)
```

```
> :t swap
swap :: (a, b) -> (b, a)
```

Polymorphism

```
> :t fst
fst :: (a, b) -> a
```

```
> :t snd
snd :: (a, b) -> b
```

Pattern Matching

```haskell
swap :: (a, b) -> (b, a)
swap (f, s) = (s, f)
```

Higher Order Functions

Functions that take
other functions as arguments.

Higher Order Functions

```
twice f a b = (f a, f b)
```

Higher Order Functions

```
twice f a b = (f a, f b)
```

```
> twice (*2) 2 3
(4, 6)
```

Higher Order Functions

```
twice f a b = (f a, f b)
```

```
> twice (*2) 2 3
(4, 6)
```

```
> :t twice
???
```

Higher Order Functions

```
twice f a b = (f a, f b)
```

```
> twice (*2) 2 3
(4, 6)
```

```
> :t twice
twice :: ? -> ? -> ? -> ?
```

Higher Order Functions

```
twice f a b = (f a, f b)
```

```
> twice (*2) 2 3
(4, 6)
```

```
> :t twice
twice :: (? -> ?) -> ? -> ? -> (?, ?)
```

Higher Order Functions

```
twice f a b = (f a, f b)
```

```
> twice (*2) 2 3
(4, 6)
```

```
> :t twice
twice :: (a -> b) -> ? -> ? -> (?, ?)
```

Higher Order Functions

```haskell
twice f a b = (f a, f b)
```

```haskell
> twice (*2) 2 3
(4, 6)
```

```haskell
> :t twice
twice :: (a -> b) -> a -> a -> (b, b)
```

Specialization

```haskell
upper :: Char -> Bool
twice :: (a -> b) -> a -> a -> (b, b)
```

```
> :t twice upper
twice upper :: ???
```

Q: What is the type?

Specialization

```haskell
upper :: Char -> Bool
twice :: (a -> b) -> a -> a -> (b, b)
```

```
> :t twice upper
twice upper :: Char -> Char -> (Bool, Bool)
```

Q: What is the type?

Basic Functions

```
id :: ???
id a = a
```

```
const ???
const a b = a
```

```
flip :: ???
flip f a b = f b a
```

Basic Functions

```
id :: a -> a
id a = a
```

```
const :: a -> b -> a
const a b = a
```

```
flip :: (a -> b -> c) -> b -> a -> c
flip f a b = f b a
```

Basic Functions

Application as operator:

```
($) :: ???
($) f a = f a
```

Composition:

```
(.) :: ???
(.) f g a = f (g a)
```

Basic Functions

Application as operator:

```haskell
($) :: (a -> b) -> a -> b
($) f a = f a
```

Composition:

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g a = f (g a)
```

Data Types

Data Types

```haskell
data Person = MkPerson String Int
  deriving Show
```

Data Types

```haskell
data Person = MkPerson String Int
  deriving Show
```

```
> :t MkPerson
MkPerson :: String -> Int -> Person
```

Data Types

```haskell
data Person = MkPerson String Int
  deriving Show
```

```
> :t MkPerson
MkPerson :: String -> Int -> Person
```

```
> MkPerson "Alice" 25
MkPerson "Alice" 25
```

Data Types

Types and constructors don't clash:

```haskell
data Person = Person String Int
```

Type Synonyms

```
type Name = String
```

Just aliases.

Record Types

```haskell
data Person = Person
  { name :: Name
  , age  :: Int
  } deriving Show
```

Record Types

```haskell
data Person = Person
   { name :: Name
   , age  :: Int
   } deriving Show
```

```
> :t name
name :: Person -> String
> :t age
age :: Person -> Int
```

Type Variables

```
data Pair a = MkPair a a
```

Type Variables

```
data Pair a = MkPair a a
```

```
> :t MkPair
MkPair :: a -> a -> Pair a
```

Type Variables

```
data Pair a = MkPair a a
```

```
> :t MkPair
MkPair :: a -> a -> Pair a
```

```
> let total (MkPair a b) = a + b
> total (MkPair 15 10)
25
```

Enumerations

```
data Direction
   = North
   | East
   | South
   | West
```

```
> :t North
North :: Direction
```

Sum Types

```haskell
data Maybe a = Nothing | Just a
```

Sum Types

```haskell
data Maybe a = Nothing | Just a
```

```
> :t Just
Just :: a -> Maybe a
```

```
> :t Nothing
Nothing :: Maybe a
```

Recursive Types

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

Recursive Types

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

```
let hi = Cons 'h' (Cons 'i' (Cons '!' Nil))
```

Pattern Matching

```
myHead :: List a -> Maybe a
myHead l =
  case l of
   Nil      -> Nothing
   Cons x _ -> Just x
```

Implementing `Last`

```
myLast :: List a -> Maybe a
```

```
???
```

Implementing `Last`

```
myLast :: List a -> Maybe a
myLast l =
  case l of
    Nil        -> ...
    Cons x Nil -> ...
    Cons _ xs  -> ...
```

Implementing `Last`

```
myLast :: List a -> Maybe a
myLast l =
  case l of
    Nil        -> Nothing
    Cons x Nil -> Just x
    Cons _ xs  -> myLast xs
```

Built-In Lists

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

```
data [a]    = []  | a : [a]
```

Pattern Matching On Lists

Pattern match on `[]` and `:`

```
and :: [Bool] -> Bool
and []     = ...
and (x:xs) = ...
```

```
or :: [Bool] -> Bool
...
```

Pattern Matching On Lists

Pattern match on `[]` and `:`

```
and :: [Bool] -> Bool
and []       = True
and (x:xs) = x && and xs
```

```
or :: [Bool] -> Bool
or []       = False
or (x:xs) = x || or xs
```

Generalizing List Functions

```
and []     = True
or  []     = False
```

```
and (x:xs) = x && and xs
or  (x:xs) = x || or  xs
```

Generalizing List Functions

```
and [] = True
or  [] = False
```

```
and (x:xs) = x && and xs
or  (x:xs) = x || or  xs
```

Combine function `f` + default case `d`:

```
gen _ d []     = d
gen f d (x:xs) = x `f` gen f d xs
```

Generalizing List Functions

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ d []     = d
foldr f d (x:xs) = f x (foldr f d xs)
```

Generalizing List Functions

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ d []     = d
foldr f d (x:xs) = f x (foldr f d xs)
```

```
and, or :: [Bool] -> Bool
and = foldr (&&) True
or  = foldr (||) False
```

More Folds

```
and      = foldr (&&) True
or       = foldr (||) False
sum      = foldr ...
product  = foldr ...
concat   = foldr ...
identity = foldr ...
reverse  = foldr ...
```

Q: Can you implement these as folds?

More Folds

```haskell
and      = foldr (&&) True
or       = foldr (||) False
sum      = foldr (+) 0
product  = foldr (*) 1
concat   = foldr (++) []
identity = foldr (:) []
reverse  = foldr (\a b -> b ++ [a]) []
```

And many more!

List Functions

```
map :: (a -> b) -> [a] -> [b]
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
???
```

Either use direct recursion,
or use `foldr`.

Map

```
map _ []     = []
map f (x:xs) = f x : map f xs
```

Map

```
map _ []     = []
map f (x:xs) = f x : map f xs
```

```
map f = foldr (\a b -> f a : b) []
```

Filter

```
filter _ []       = []
filter p (x:xs) =
  if p x
  then x : filter p xs
  else filter p xs
```

Filter

```haskell
filter _ []     = []
filter p (x:xs) =
  if p x
  then x : filter p xs
  else filter p xs
```

```haskell
filter p = foldr step []
  where step a b = if p a
                   then a : b
                   else b
```

10

IO?

Haskell function are *pure* and *lazy*.

Not ideal for IO: needs side effects and sequencing.

Do-Notation

```haskell
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello " ++ name ++ "!")
```

Do-Notation

```
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello " ++ name ++ "!")
```

Perform statements one after another.

Use `<-` to bind results.

IO Type

```haskell
putStrLn :: String -> IO ()
```

- Takes a `String`.
- Performs IO.
- Doesn't return result.

IO Type

```haskell
putStrLn :: String -> IO ()
```

- Takes a `String`.
- Performs IO.
- Doesn't return result.

```haskell
() :: ()
```

The unit type with only one value.

IO Operations

```
getLine :: IO String
```

IO Operations

```
getLine :: IO String
```

```
print :: Show a => a -> IO ()
```

IO Operations

```
getLine :: IO String
```

```
print :: Show a => a -> IO ()
```

```
readFile :: FilePath -> IO String
```

IO Operations

```
getLine :: IO String
```

```
print :: Show a => a -> IO ()
```

```
readFile :: FilePath -> IO String
```

```
writeFile :: FilePath -> String -> IO ()
```

Control Structures

IO actions are first class.

```
> when (even 2) (putStrLn "Two is even.")
Two is even.
```

Control Structures

IO actions are first class.

```
> when (even 2) (putStrLn "Two is even.")
Two is even.
```

```
when :: Bool -> IO () -> IO ()
when cond act =
  if cond
  then act
  else return ()
```

Return

Return isn't what you're used to.

```
return :: a -> IO a
```

Lifts a pure value into IO.

Doesn't jump out of the function.

Return

Try running the following:

```
f = do
  putStrLn "a"
  return ()
  putStrLn "b"
```

Control Structures - 2

Define a function that takes a list of IO actions, and performs all of them.

Control Structures - 2

Define a function that takes a list of IO actions, and performs all of them.

```
mySequence :: [IO ()] -> IO ()
mySequence [] = return ()
mySequence (act:acts) = do
  act
  mySequence acts
```

Other Control Functions

```
mapM :: (a -> IO b) -> [a] -> IO [b]
```

```
forM :: [a] -> (a -> IO b) -> IO [b]
```

Other Control Functions

```
mapM :: (a -> IO b) -> [a] -> IO [b]
```

```
forM :: [a] -> (a -> IO b) -> IO [b]
```

```
forever :: IO a -> IO b
```

Other Control Functions

```haskell
mapM :: (a -> IO b) -> [a] -> IO [b]
```

```haskell
forM :: [a] -> (a -> IO b) -> IO [b]
```

```haskell
forever :: IO a -> IO b
```

```haskell
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Classes

Classes

Classes allow *ad-hoc* overloading of functions.

```
(==) :: Eq a => a -> a -> Bool
```

Compare with (parametrically) polymorphic functions.

```
length :: [a] -> Bool
```

Defining Classes

```haskell
class Equal a where
  equal :: a -> a -> Bool
```

Defines an interface, no implementation yet.

Defining Classes

```
class Equal a where
  equal :: a -> a -> Bool
```

Defines an interface, no implementation yet.

```
instance Equal () where
  equal () () = True
```

Defining Classes

```haskell
class Equal a where
  equal :: a -> a -> Bool
```

Defines an interface, no implementation yet.

```haskell
instance Equal () where
  equal () () = True
```

Q: Give instances for `Bool` and `Maybe a`.

Common Type Classes

```
class Eq a where
  (==) :: a -> a -> Bool
```

```
class Ord a where
  (<=) :: a -> a -> Bool
```

Common Type Classes

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

```haskell
class Ord a where
  (<=) :: a -> a -> Bool
```

```haskell
class Show a where
  show :: a -> String
class Read a where
  read :: String -> a
```

Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Functor

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Generalizes list's `map`.

Instances for `Maybe, [], IO`, ...

Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Q: Define a binary tree with values in the branches.

Q: Give an instance Functor Tree.

Functor

```
data Tree a
  = Leaf
  | Branch (Tree a) a (Tree a)
```

Functor

```haskell
data Tree a
  = Leaf
  | Branch (Tree a) a (Tree a)
```

```haskell
instance Functor Tree where
  fmap f Leaf           = …
  fmap f (Branch l x r) = …
```

Functor

```
data Tree a
  = Leaf
  | Branch (Tree a) a (Tree a)
```

```
instance Functor Tree where
  fmap f Leaf                = Leaf
  fmap f (Branch l x r) = Branch …
```

Functor

```haskell
data Tree a
  = Leaf
  | Branch (Tree a) a (Tree a)
```

```haskell
instance Functor Tree where
  fmap f Leaf                 = Leaf
  fmap f (Branch l x r) = Branch ? (f x) ?
```

Functor

```haskell
data Tree a
  = Leaf
  | Branch (Tree a) a (Tree a)
```

```haskell
instance Functor Tree where
  fmap f Leaf           = Leaf
  fmap f (Branch l x r) =
    Branch (fmap f l) (f x) (fmap f r)
```

Nested Maybes

Imagine web application where users can have accounts and sites.

```
parseSessionId :: String    -> Maybe SessionId
lookupUser     :: SessionId -> Maybe User
getUserSite    :: User      -> Maybe Site
```

Nested Maybes

```
showSessionSite :: String -> Maybe String
showSessionSite str = …
```

Nested Maybes

```
showSessionSite :: String -> Maybe String
showSessionSite str =
  case parseSessionId str of
    Nothing  -> Nothing
    Just sid -> …
```

Nested Maybes

```haskell
showSessionSite :: String -> Maybe String
showSessionSite str =
  case parseSessionId str of
    Nothing  -> Nothing
    Just sid ->
      case lookupUser sid of
        Nothing  -> Nothing
        Just usr -> …
```

Nested Maybes

```
showSessionSite :: String -> Maybe String
showSessionSite str =
  case parseSessionId str of
    Nothing  -> Nothing
    Just sid ->
      case lookupUser sid of
        Nothing  -> Nothing
        Just usr ->
          case getUserSite of
            Nothing   -> Nothing
            Just site -> show site
```

Sequencing Maybes

Let's abstract the pattern:

```
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing  >>? _ = Nothing
(Just x) >>? f = f x
```

Sequencing Maybes

Now our example becomes:

```
showSessionSite :: String -> Maybe String
showSessionSite str =
  parseSessionId str >>? \sid  ->
  lookupUser sid     >>? \usr  ->
  getUserSite usr    >>? \site ->
  Just (show site)
```

Monad

This pattern occurs more often, and is captured in a class:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Monad

This pattern occurs more often, and is captured in a class:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

```
instance Monad Maybe where
  (>>=)  = (>>?)
  return = Just
```

Do Notation Revisited

Do notation isn't just for IO. It works for all monads.

```haskell
showSessionSite :: String -> Maybe String
showSessionSite str = do
  sid  <- parseSessionId str
  usr  <- lookupUser sid
  site <- getUserSite usr
  return (show site)
```

Many Monads

IO is a monad, as are many other things:

- Pure mutable state.
- Immutable state (configuration).
- Logging (writable state).
- Parsers.
- Randomness.
- Failure with error.
- ...

Modules

Importing Modules

```
import Data.List
```

Modules are hierarchical.

Explicit Imports

```
import Data.List (sort, group)
```

Hiding Imports

```
import Data.List hiding (intercalate)
```

Qualified Imports

```haskell
import qualified Data.List as Ls
```

```haskell
unlines :: [String] -> String
unlines = Ls.intercalate "\n"
```

Good Practice

Use explicit imports
and qualified imports
where possible.

Creating A Module

```
module MyApp.MyModule where
```

Make sure file name and module name match

Explicit Exports

```haskell
module MyApp.MyModule
( T (..)
, f
) where

data T = T Int

f = ...
g = ...
```

Packages

Multiple modules can be combined into a *Cabal package*.

Cabal packages can be uploaded to *Hackage*.

Final Exercise

Write a spell checker using all the tricks you learned today!

Write a Haskell module that

1. reads a word list from a file
2. parses the format into some dictionary type
3. starts an interactive loop that:

   - reads a word from standard input
   - spell checks the word
   - and prints if the word was found

Some Tips

- Split up program into small functions.
- Think about the types first.
- Use Hoogle (http://haskell.org/hoogle) to find functions.
- Team up if you want.
- Ask us anything!

Haskell Resources

Books

- Learn You a Haskell for Great Good! (http://learnyouahaskell.com)
- Real World Haskell (http://book.realworldhaskell.org)

Help and discussion

- Haskell reddit (http://reddit.com/r/haskell)
- Stack overflow (http://stackoverflow.com/questions/tagged/haskell)
- Haskell-cafe mailing list (http://haskell.org/mailman/listinfo/haskell-cafe)
- #haskell on Freenode.

Libraries

- Hackage (http://hackage.haskell.org) - package database
- Hoogle (http://haskell.org/hoogle) - library search

silk.co
@silkapp

github.com/silkapp

+Erik Hesselink

github.com/hesselink

fvisser.nl
@sfvisser

github.com/sebastiaanvisser