CENG 3549 – Functional Programming

User-Defined Types & Trees & Input and Output

Burak Ekici

October 13, 2022

Outline

Functions You Need to Know

- infix operators and special syntax

  `(<=), (<), (==), (>=), (>), (||), (-), (,), (:), (/=), (.), (*), (&&), (+), (++), [], [m..n]`

- other `Prelude` functions

  `abs`, `compare`, `concat`, `const`, `div`, `drop`, `error`, `even`, `filter`, `foldr`, `foldr1`, `fromInteger`, `fst`, `head`, `init`, `last`, `length`, `lines`, `map`, `max`, `min`, `mod`, `negate`, `not`, `null`, `product`, `putStr`, `putStrLn`, `read`, `replicate`, `reverse`, `show`, `showList`, `showsPrec`, `signum`, `snd`, `splitAt`, `sum`, `tail`, `take`, `unlines`, `unwords`, `words`, `zip`, `zipWith`

- other `Prelude` constants

  `False`, `otherwise`, `True`

- other functions

  `Data.Char.isDigit`, `System.Environment.getArgs`

Syntax You Need to Recognize

- anonymous functions / functions without names

```
(\x -> 2 * x) -- an anonymous function for doubling
```

Syntax You Need to Recognize

- anonymous functions / functions without names

  `(\x -> 2 * x) -- an anonymous function for doubling`

- infix operators and sections

`(+)`	`= (\x y -> x + y)`	infix to prefix
``x `f` y``	`= f x y`	prefix to infix
`(a >)`	`= (\x -> a > x)`	argument smaller than a?
`(> b)`	`= (\x -> x > b)`	argument greater than b?

Syntax You Need to Recognize

- anonymous functions / functions without names

```
(\x -> 2 * x) -- an anonymous function for doubling
```

- infix operators and sections

`(+)`	`= (\x y -> x + y)`	infix to prefix
``x `f` y``	`= f x y`	prefix to infix
`(a >)`	`= (\x -> a > x)`	argument smaller than a?
`(> b)`	`= (\x -> x > b)`	argument greater than b?

- patterns and guards

```
headIfPositive xs = case xs of
                        x:_ | x > 0 -> x
```

Syntax You Need to Recognize

- anonymous functions / functions without names

```
(\x -> 2 * x) -- an anonymous function for doubling
```

- infix operators and sections

`(+)`	`= (\x y -> x + y)`	infix to prefix
``x `f` y``	`= f x y`	prefix to infix
`(a >)`	`= (\x -> a > x)`	argument smaller than a?
`(> b)`	`= (\x -> x > b)`	argument greater than b?

- patterns and guards

```
headIfPositive xs = case xs of
                      x:_ | x > 0 -> x
```

- list comprehensions

```
filter p xs         ==  [x   | x <- xs, p x]
map f xs            ==  [f x | x <- xs]
concat (map f xs)   ==  [y   | x <- xs, y <- f x]
concat $ map (\x -> map ((,)x) ys) xs == [(x, y) |x <- xs, y <- ys]
```

Types and Type Classes

- type signatures – annotate functions by types

```
range :: Int -> Int -> [Int]
range m n | m > n     = []
          | otherwise = m : range (m + 1) n
```

Types and Type Classes

- type signatures – annotate functions by types

```
range :: Int -> Int -> [Int]
range m n | m > n     = []
          | otherwise = m : range (m + 1) n
```

- type synonyms – mnemonic names for types

```
type Height = Int
type Width  = Int
```

Types and Type Classes

- type signatures – annotate functions by types

```
range :: Int -> Int -> [Int]
range m n | m > n     = []
          | otherwise = m : range (m + 1) n
```

- type synonyms – mnemonic names for types

```
type Height = Int
type Width  = Int
```

- type classes and class constraints – for every function `f`, specific to class `C`, type inference adds a `C`-constraint to type

Types and Type Classes

- type signatures – annotate functions by types

```
range :: Int -> Int -> [Int]
range m n | m > n     = []
          | otherwise = m : range (m + 1) n
```

- type synonyms – mnemonic names for types

```
type Height = Int
type Width  = Int
```

- type classes and class constraints – for every function `f`, specific to class `C`, type inference adds a `C`-constraint to type

Example

- without type signature, we get

```
ghci> :t range
range :: (Ord a, Num a) => a -> a -> [a]
```

- `m > n`, hence `m` and `n` of class `Ord` and `m` and `n` of same type
- `m + 1`, hence `m` of class `Num`
- `m` and `n` of same type, hence `n` of class `Num`

Equational Reasoning

- a function definition in Haskell is a (set of conditional) equation(s)
- if conditions are met, we may "replace equals by equals"
- in this way we may evaluate function calls by applying equations stepwise, until we reach final result

Equational Reasoning

- a function definition in Haskell is a (set of conditional) equation(s)
- if conditions are met, we may "replace equals by equals"
- in this way we may evaluate function calls by applying equations stepwise, until we reach final result

Kinds of Conditions

- "**if** b **then** t **else** e" is t, when b is true; and e, otherwise
- "**case** e **of** { p_1 -> e_1; ...; p_n -> e_n }" is e_i, if e first matches p_i

Equational Reasoning

- a function definition in Haskell is a (set of conditional) equation(s)
- if conditions are met, we may "replace equals by equals"
- in this way we may evaluate function calls by applying equations stepwise, until we reach final result

Kinds of Conditions

- "**if** b **then** t **else** e" is t, when b is true; and e, otherwise
- "**case** e **of** { p_1 -> e_1; ... ;p_n -> e_n }" is e_i, if e first matches p_i

Primitive Operations

- for primitive operations (like (+), (∗), ...), we assume predefined equations
- e.g., `1 + 2 = 3`, `0 * 10 = 0`, ...

Examples (equational reasoning)

- definition

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```

Examples (equational reasoning)

- definition

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```

- evaluate `zip [1,2,3] ['a','b']`

Examples (equational reasoning)

- definition

```haskell
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```

- evaluate `zip [1,2,3] ['a','b']`

- definition

```haskell
factorial n | n <= 1    = 1
            | otherwise = n * factorial (n - 1)
```

Examples (equational reasoning)

- definition
```haskell
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```
- evaluate `zip [1,2,3] ['a','b']`
- definition
```haskell
factorial n | n <= 1    = 1
            | otherwise = n * factorial (n - 1)
```
- evaluate `factorial 3`

Examples (equational reasoning)

- definition
```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```
- evaluate `zip [1,2,3] ['a','b']`
- definition
```
factorial n | n <= 1    = 1
            | otherwise = n * factorial (n - 1)
```
- evaluate `factorial 3`
- definition `head xs = case xs of x:_ -> x`

Examples (equational reasoning)

- definition
```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```
- evaluate `zip [1,2,3] ['a','b']`
- definition
```
factorial n | n <= 1    = 1
            | otherwise = n * factorial (n - 1)
```
- evaluate `factorial 3`
- definition `head xs = case xs of x:_ -> x`
- evaluate `head "ab"`

Examples (equational reasoning)

- definition

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```

- evaluate `zip [1,2,3] ['a','b']`
- definition

```
factorial n | n <= 1    = 1
            | otherwise = n * factorial (n - 1)
```

- evaluate `factorial 3`
- definition `head xs = case xs of x:_ -> x`
- evaluate `head "ab"`
- definitions

```
null xs = case xs of { [] -> True; _ -> False }
tail xs = case xs of _:ys -> ys
prod xs = if null xs then 1
          else head xs * prod (tail xs)
```

Examples (equational reasoning)

- definition
```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```
- evaluate `zip [1,2,3] ['a','b']`
- definition
```
factorial n | n <= 1    = 1
            | otherwise = n * factorial (n - 1)
```
- evaluate `factorial 3`
- definition `head xs = case xs of x:_ -> x`
- evaluate `head "ab"`
- definitions
```
null xs = case xs of { [] -> True; _ -> False }
tail xs = case xs of _:ys -> ys
prod xs = if null xs then 1
          else head xs * prod (tail xs)
```
- evaluate `prod [5,6]`

Outline

1. Intermediate Wrap-Up
2. User-Defined Types
3. Trees
4. Input and Output

Data Declarations – Algebraic Data Types

- new types are introduced by

$$
\begin{array}{rcl}
\textbf{data}\; T\; \alpha_1 \cdots \alpha_n & = & C_1\; \tau_{11} \cdots \tau_{1m_1} \\
& | & \vdots \\
& | & C_k\; \tau_{k1} \cdots \tau_{km_k}
\end{array}
$$

Data Declarations – Algebraic Data Types

- new types are introduced by

$$\begin{array}{rcl} \textbf{data}\ T\ \alpha_1\ \cdots\ \alpha_n & = & C_1\ \tau_{11}\ \cdots\ \tau_{1m_1} \\ & | & \vdots \\ & | & C_k\ \tau_{k1}\ \cdots\ \tau_{km_k} \end{array}$$

- where T is name of new type (constructor) –starting with capital letter– taking n type parameters α_1 to α_n

Data Declarations – Algebraic Data Types

- new types are introduced by

$$
\begin{array}{rcl}
\textbf{data}\ T\ \alpha_1\ \cdots\ \alpha_n & = & C_1\ \tau_{11}\ \cdots\ \tau_{1m_1} \\
& | & \vdots \\
& | & C_k\ \tau_{k1}\ \cdots\ \tau_{km_k}
\end{array}
$$

- where T is name of new type (constructor) –starting with capital letter– taking n type parameters α_1 to α_n
- and C_i is name of ith (data) constructor, taking m_i arguments of types τ_{i1} to τ_{im_i} (with type variables among α_1 to α_n)

Data Declarations – Algebraic Data Types

- new types are introduced by

$$\begin{array}{rcl} \textbf{data}\ T\ \alpha_1\ \cdots\ \alpha_n & = & C_1\ \tau_{11}\ \cdots\ \tau_{1m_1} \\ & | & \vdots \\ & | & C_k\ \tau_{k1}\ \cdots\ \tau_{km_k} \end{array}$$

- where T is name of new type (constructor) –starting with capital letter– taking n type parameters α_1 to α_n
- and C_i is name of ith (data) constructor, taking m_i arguments of types τ_{i1} to τ_{im_i} (with type variables among α_1 to α_n)

Examples

- `data Bool = False | True`
- `data List a = Nil | Cons a (List a)`
- `data Pair a b = Pair a b`

Data Declarations – Algebraic Data Types

- new types are introduced by

$$\begin{array}{rcl} \textbf{data}\ T\ \alpha_1\ \cdots\ \alpha_n & = & C_1\ \tau_{11}\ \cdots\ \tau_{1m_1} \\ & | & \vdots \\ & | & C_k\ \tau_{k1}\ \cdots\ \tau_{km_k} \end{array}$$

- where T is name of new type (constructor) –starting with capital letter– taking n type parameters α_1 to α_n
- and C_i is name of ith (data) constructor, taking m_i arguments of types τ_{i1} to τ_{im_i} (with type variables among α_1 to α_n)

Examples

- `data Bool = False | True`
- `data List a = Nil | Cons a (List a)`
- `data Pair a b = Pair a b`

constructors and type names live in different name spaces

Automatically Deriving Type Class Instances

- for some type classes it is possible to automatically derive instances for algebraic data types
- e.g.,

```
data List a = Nil | Cons a (List a)
  deriving (Eq, Show, Read)
```

- now, we are able to use (==), show, and read for Lists

Automatically Deriving Type Class Instances

- for some type classes it is possible to automatically derive instances for algebraic data types
- e.g.,

```haskell
data List a = Nil | Cons a (List a)
  deriving (Eq, Show, Read)
```

- now, we are able to use (==), show, and read for Lists

Examples

```
ghci> Nil == Cons 1 Nil
False
ghci> show (Cons 1 (Cons 2 Nil))
"Cons␣1␣(Cons␣2␣Nil)"
ghci> read it :: List Int
Cons 1 (Cons 2 Nil)
```

Outline

Definition (tree)

- (rooted) tree $T = (N, E)$

Definition (tree)

- (rooted) tree $T = (N, E)$
- with set of nodes/vertices N

Definition (tree)

- (rooted) tree $T = (N, E)$
- with set of nodes/vertices N
- and set of edges $E \subseteq N \times N$

Definition (tree)

- (rooted) tree $T = (N, E)$
- with set of nodes/vertices N
- and set of edges $E \subseteq N \times N$
- unique root of T ($root(T) \in N$) without predecessor

Definition (tree)

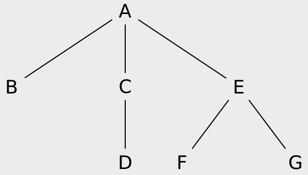
- (rooted) tree $T = (N, E)$
- with set of nodes/vertices N
- and set of edges $E \subseteq N \times N$
- unique root of T ($root(T) \in N$) without predecessor
- all other nodes have exactly one predecessor

Definition (tree)

- (rooted) tree $T = (N, E)$
- with set of nodes/vertices N
- and set of edges $E \subseteq N \times N$
- unique root of T ($root(T) \in N$) without predecessor
- all other nodes have exactly one predecessor

Example

- $N = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}\}$
- $E = \{(\mathrm{A}, \mathrm{B}), (\mathrm{A}, \mathrm{C}), (\mathrm{A}, \mathrm{E}), (\mathrm{C}, \mathrm{D}), (\mathrm{E}, \mathrm{F}), (\mathrm{E}, \mathrm{G})\}$
- $root(T) = \mathrm{A}$
- $T =$



Trees in Haskell

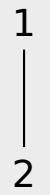
- possible type for trees with arbitrary nodes

```haskell
data Tree a = Empty | Node a [Tree a]
```

- a tree is either empty (0 nodes) or there is at least one node with content of type a and an arbitrary number of successor trees

Examples

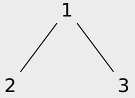
`Empty`



`Node 1 [Node 2 []]`

1

`Node 1 []`



`Node 1 [Node 2 [],Node 3 []]`

Binary Trees

- restrict number of successors (maximum 2)
- type

```
data BTree a = Empty | Node a (BTree a) (BTree a)
  deriving (Eq, Show, Read)
```

Binary Trees

- restrict number of successors (maximum 2)
- type

```haskell
data BTree a = Empty | Node a (BTree a) (BTree a)
  deriving (Eq, Show, Read)
```

Functions on Binary Trees

- size – number of nodes

```haskell
size :: BTree a -> Integer
size Empty        = 0
size (Node _ l r) = size l + size r + 1
```

Binary Trees

- restrict number of successors (maximum 2)
- type

```haskell
data BTree a = Empty | Node a (BTree a) (BTree a)
  deriving (Eq, Show, Read)
```

Functions on Binary Trees

- size – number of nodes

```haskell
size :: BTree a -> Integer
size Empty        = 0
size (Node _ l r) = size l + size r + 1
```

- height – length of longest path from root to some leaf

```haskell
height :: BTree a -> Integer
height Empty        = 0
height (Node _ l r) = max (height l) (height r) + 1
```

Creating Trees from Lists

- the easy way

```
fromList []     = Empty
fromList (x:xs) = Node x Empty (fromList xs)
```

Creating Trees from Lists

- the easy way

```
fromList []     = Empty
fromList (x:xs) = Node x Empty (fromList xs)
```

- the balanced way

```
make [] = Empty
make xs = Node z (make ys) (make zs)
  where
    m          = length xs `div` 2
    (ys, z:zs) = splitAt m xs
```

Creating Trees from Lists

- the easy way

```
fromList []     = Empty
fromList (x:xs) = Node x Empty (fromList xs)
```

- the balanced way

```
make [] = Empty
make xs = Node z (make ys) (make zs)
  where
    m          = length xs `div` 2
    (ys, z:zs) = splitAt m xs
```

- the orderly way

```
searchTree = foldr insert Empty
  where
    insert x Empty = Node x Empty Empty
    insert x (Node y l r)
      | x < y     = Node y (insert x l) r
      | otherwise = Node y l (insert x r)
```

Transforming Trees into Lists

```
flatten Empty        = []
flatten (Node x l r) = flatten l ++ [x] ++ flatten r
```

Example (a sorting algorithm for lists)

```
sort = flatten . searchTree
```

Outline

Example

- write the file `welcomeIO.hs`

```haskell
main = do
  putStrLn "Greetings! What's your name?"
  name <- getLine
  putStrLn (
    "Welcome to Haskell's IO, " ++ name ++ "!")
```

- compile it with GHC via
- and run it

Example

- write the file `welcomeIO.hs`

```haskell
main = do
  putStrLn "Greetings! What's your name?"
  name <- getLine
  putStrLn (
    "Welcome to Haskell's IO, " ++ name ++ "!")
```

- compile it with GHC via

- and run it

Remark

- `putStrLn` – prints string followed by newline
- `getLine` – reads line from standard input
- new syntax: **do** and <-

IO and the Type System

- consider

```
ghci> :load welcomeIO.hs
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t getLine
getLine :: IO String
ghci> :t main
main :: IO ()
```

IO and the Type System

- consider

```
ghci> :load welcomeIO.hs
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t getLine
getLine :: IO String
ghci> :t main
main :: IO ()
```

- `IO a` is type of IO actions delivering results of type `a` (in addition to their IO operations)

IO and the Type System

- consider

```
ghci> :load welcomeIO.hs
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t getLine
getLine :: IO String
ghci> :t main
main :: IO ()
```

- `IO a` is type of IO actions delivering results of type `a` (in addition to their IO operations)

Examples

- `String -> IO ()` – after supplying a string, we obtain an IO action (in case of `putStrLn`, “printing”)
- `IO ()` – just IO (in case of `main`, run our program)
- `IO String` – do some IO and deliver a string (in case of `getLine`, user-input)

Further Notes

- IO actions (everything of type `IO a`) are just descriptions of what should be done; nothing is actually done at time of specification

Further Notes

- IO actions (everything of type `IO a`) are just descriptions of what should be done; nothing is actually done at time of specification
- inside IO actions, order is important; IO actions are executed in order of appearance (once execution starts); result of sequence of IO actions is result of last action

Further Notes

- IO actions (everything of type `IO a`) are just descriptions of what should be done; nothing is actually done at time of specification
- inside IO actions, order is important; IO actions are executed in order of appearance (once execution starts); result of sequence of IO actions is result of last action
- inside IO actions, `x <- action` (where `action :: IO a`) may be used to bind result of `action` (which has type `a`) to name `x` (but seriously, this is actually only done, once execution starts)

Further Notes

- IO actions (everything of type `IO a`) are just descriptions of what should be done; nothing is actually done at time of specification
- inside IO actions, order is important; IO actions are executed in order of appearance (once execution starts); result of sequence of IO actions is result of last action
- inside IO actions, `x <- action` (where `action :: IO a`) may be used to bind result of `action` (which has type `a`) to name `x` (but seriously, this is actually only done, once execution starts)
- `x <- a` is not available outside IO actions

Further Notes

- IO actions (everything of type `IO a`) are just descriptions of what should be done; nothing is actually done at time of specification
- inside IO actions, order is important; IO actions are executed in order of appearance (once execution starts); result of sequence of IO actions is result of last action
- inside IO actions, `x <- action` (where `action :: IO a`) may be used to bind result of `action` (which has type `a`) to name `x` (but seriously, this is actually only done, once execution starts)
- `x <- a` is not available outside IO actions

Implications

- once we are inside an IO action, we cannot escape
- strict separation between purely functional code and IO
- when `IO a` does not appear inside type signature, we can be absolutely sure that no IO ("side-effect") is performed

Using Pure Code Inside IO Actions

- consider program `reply.hs`

```haskell
reply :: String -> String
reply name =
  "Pleased to meet you, " ++ name ++ ".\n" ++
  "Your name contains " ++ n ++ " characters."
  where
    n = show $ length name

main :: IO ()
main = do
  putStrLn "Greetings again. What's your name?"
  name <- getLine
  let niceReply = reply name
  putStrLn niceReply
```

Using Pure Code Inside IO Actions

- consider program `reply.hs`

```haskell
reply :: String -> String
reply name =
  "Pleased to meet you, " ++ name ++ ".\n" ++
  "Your name contains " ++ n ++ " characters."
  where
    n = show $ length name

main :: IO ()
main = do
  putStrLn "Greetings again. What's your name?"
  name <- getLine
  let niceReply = reply name
  putStrLn niceReply
```

- that is, we may use `let x = e` (there is no `in` here!) to bind result of pure expression `e` to name `x`

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin
- `getLine :: IO String` – read line (excluding newline)

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin
- `getLine :: IO String` – read line (excluding newline)
- `interact :: (String -> String) -> IO ()` – use function that gets input as string and produces output as string

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin
- `getLine :: IO String` – read line (excluding newline)
- `interact :: (String -> String) -> IO ()` – use function that gets input as string and produces output as string
- `type FilePath = String`

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin
- `getLine :: IO String` – read line (excluding newline)
- `interact :: (String -> String) -> IO ()` – use function that gets input as string and produces output as string
- `type FilePath = String`
- `readFile :: FilePath -> IO String` – read file content

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin
- `getLine :: IO String` – read line (excluding newline)
- `interact :: (String -> String) -> IO ()` – use function that gets input as string and produces output as string
- `type FilePath = String`
- `readFile :: FilePath -> IO String` – read file content
- `writeFile :: FilePath -> String -> IO ()`

Some Simple IO Functions

- `return :: a -> IO a` – turn anything into an IO action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin
- `getLine :: IO String` – read line (excluding newline)
- `interact :: (String -> String) -> IO ()` – use function that gets input as string and produces output as string
- `type FilePath = String`
- `readFile :: FilePath -> IO String` – read file content
- `writeFile :: FilePath -> String -> IO ()`
- `appendFile :: FilePath -> String -> IO ()`

Examples (imitating some GNU commands)

- `cat.hs` – print file contents

```
main = do
  [file] <- getArgs
  s <- readFile file
  putStr s
```

Examples (imitating some GNU commands)

- `cat.hs` – print file contents

```haskell
main = do
  [file] <- getArgs
  s <- readFile file
  putStr s
```

- `wc.hs` – count newlines/words/characters in input

```haskell
count s = ns ++ "  " ++ ws ++ "  " ++ bs ++ "\n"
  where ns = show $ length $ lines s
        ws = show $ length $ words s
        bs = show $ length s

main = interact count
```

Examples (imitating some GNU commands)

- `cat.hs` – print file contents

```haskell
main = do
  [file] <- getArgs
  s <- readFile file
  putStr s
```

- `wc.hs` – count newlines/words/characters in input

```haskell
count s = ns ++ "  " ++ ws ++ "  " ++ bs ++ "\n"
  where ns = show $ length $ lines s
        ws = show $ length $ words s
        bs = show $ length s

main = interact count
```

- `uniq.hs` – omit repeated lines of input

```haskell
main = interact (nub)
```

Examples (imitating some GNU commands)

- `cat.hs` – print file contents

```haskell
main = do
  [file] <- getArgs
  s <- readFile file
  putStr s
```

- `wc.hs` – count newlines/words/characters in input

```haskell
count s = ns ++ "  " ++ ws ++ "  " ++ bs ++ "\n"
  where ns = show $ length $ lines s
        ws = show $ length $ words s
        bs = show $ length s

main = interact count
```

- `uniq.hs` – omit repeated lines of input

```haskell
main = interact (nub)
```

- `sort.hs` – sort input lines

```haskell
main = interact (sort)
```

Remark

- `getArgs :: IO [String]` is in `System.Environment`
- `nub :: Eq a => [a] -> [a]` is in `Data.List`; eliminates duplicates
- `sort :: Ord a => [a] -> [a]` is in `Data.List`; sorts a list

Remark

- `getArgs :: IO [String]` is in `System.Environment`
- `nub :: Eq a => [a] -> [a]` is in `Data.List`; eliminates duplicates
- `sort :: Ord a => [a] -> [a]` is in `Data.List`; sorts a list

Remark

- `getArgs :: IO [String]` is in `System.Environment`
- `nub :: Eq a => [a] -> [a]` is in `Data.List`; eliminates duplicates
- `sort :: Ord a => [a] -> [a]` is in `Data.List`; sorts a list

Examples (do some IO action for each argument)

- ```
  foreach :: [a] -> (a -> IO ()) -> IO ()
  foreach []     io = return ()
  foreach (a:as) io = do { io a; foreach as io }
  ```
  ```

Remark

- `getArgs :: IO [String]` is in `System.Environment`
- `nub :: Eq a => [a] -> [a]` is in `Data.List`; eliminates duplicates
- `sort :: Ord a => [a] -> [a]` is in `Data.List`; sorts a list

Examples (do some IO action for each argument)

-
```
foreach :: [a] -> (a -> IO ()) -> IO ()
foreach []     io = return ()
foreach (a:as) io = do { io a; foreach as io }
```

- better `cat.hs`

```
main = do
  files <- getArgs
  if null files then interact id else do
    foreach files readAndPrint
    where readAndPrint file = do
            s <- readFile file
            putStr s
```

Thanks! & Questions?