

INTRODUCTION TO

---

# FUNCTIONAL PROGRAMMING



# READ

- ▶ Parsing of strings into Haskell data types
  - ▶ [doc](#)
- ▶ You need to provide a type into which the string is supposed to be parsed
- ▶ Derived instances of **Read** work nicely with derived instances of **Show**

A terminal window titled "Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.V..." showing the GHCi prompt. The user enters three commands: "read \"1\" :: Int", "read \"1.23\" :: Double", and "read \"Just 123\" :: Maybe Int". The corresponding outputs are "1", "1.23", and "Just 123". The prompt "ghci>" is shown at the end of the last line.

```
GHCI, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci> read "1" :: Int
1
[ghci> read "1.23" :: Double
1.23
[ghci> read "Just 123" :: Maybe Int
Just 123
ghci> █
```

## READ

- ▶ Parsing of strings into Haskell data types
  - ▶ [doc](#)
- ▶ You need to provide a type into which the string is supposed to be parsed
- ▶ Derived instances of `Read` work nicely with derived instances of `Show`
- ▶ Make sure the compiler knows exactly what type you want your string to be parsed into

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.V...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci> read "1" :: Int
1
[ghci> read "1.23" :: Double
1.23
[ghci> read "Just 123" :: Maybe Int
Just 123
[ghci> read "Nothing" :: Maybe Int
Nothing
[ghci> read "Nothing" :: Maybe a

<interactive>:5:1: error: [GHC-39999]
• No instance for 'Read a1' arising from a use of 'read'
  Possible fix:
    add (Read a1) to the context of
      an expression type signature:
        forall a1. Maybe a1
• In the expression: read "Nothing" :: Maybe a
  In an equation for 'it': it = read "Nothing" :: Maybe a
ghci> █
```

# USING READ

- ▶ Parse numbers from a string and sum them up
- ▶ `words :: String → [String]` splits a string into a list of strings based on whitespace characters
  - ▶ TG: `unwords`, `lines`, `unlines`
- ▶ `map (\x → read x :: Int)` parses numbers
- ▶ `sum` sums them up

▶

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.V...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
ghci> sumIntsInString = sum . map (\x -> read x :: Int) . words
[ghci> sumIntsInString "123 45 777"
945
[ghci>
ghci>
```

# USING READ

- ▶ Parse numbers from a string and sum them up
- ▶ `words :: String → [String]` splits a string into a list of strings based on whitespace characters
  - ▶ TG: `unwords`, `lines`, `unlines`
- ▶ `map (\x → read x :: Int)` parses numbers
- ▶ `sum` sums them up
- ▶ Notice: type application `@`
  - ▶ `read :: ∀a. Read a ⇒ String → a`

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.V...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
ghci> sumIntsInString = sum . map (\x → read x :: Int) . words
[ghci> sumIntsInString "123 45 777"
945
[ghci>
[ghci> sumIntsInString = sum . map (read @Int) . words
[ghci> sumIntsInString "123 45 777"
945
ghci> █
```

# SAFE(R) READ

- ▶ It's always better to ensure `read` doesn't fail at runtime
- ▶ If there is a chance of read failing, use `readMaybe`

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.V...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci> :m Text.Read
[ghci> read @Int "123"
123
[ghci> read @Int "12.3"
*** Exception: Prelude.read: no parse
[ghci>
[ghci> :info readMaybe
readMaybe :: Read a => String -> Maybe a           -- Defined in 'Text.Read'
[ghci> readMaybe @Int "123"
Just 123
[ghci> readMaybe @Int "12.3"
Nothing
ghci> █
```



# EXERCISE

- ▶ Read a string from the standard input
- ▶ If it's a string of integers, separated by spaces, create a binary tree out of them
- ▶ If it's a string representation of a binary tree, create a binary tree out of it
- ▶ If it's neither, report an error
- ▶ [gist](#)

```
1  module Tree where
2
3  import Text.Read (readMaybe)
4
5  data Tree a
6  | = E
7  | N (Tree a) a (Tree a)
8  deriving (Show, Eq, Read)
9
10 insert :: Ord a => a -> Tree a -> Tree a
11 insert = undefined
12
13 data ParseResult
14 | = IntList [Int]
15 | IntTree (Tree Int)
16
17 parse :: String -> Either String ParseResult
18 parse = undefined
19
20 makeTree :: ParseResult -> Tree Int
21 makeTree (IntTree x) = x
22 makeTree (IntList xs) = foldr insert E xs
23
24 main :: IO ()
25 main = do
26   str <- getLine
27   let input = parse str
28   case input of
29   | Right pr -> print $ makeTree pr
30   | Left err -> putStrLn err
```