INTRODUCTION TO

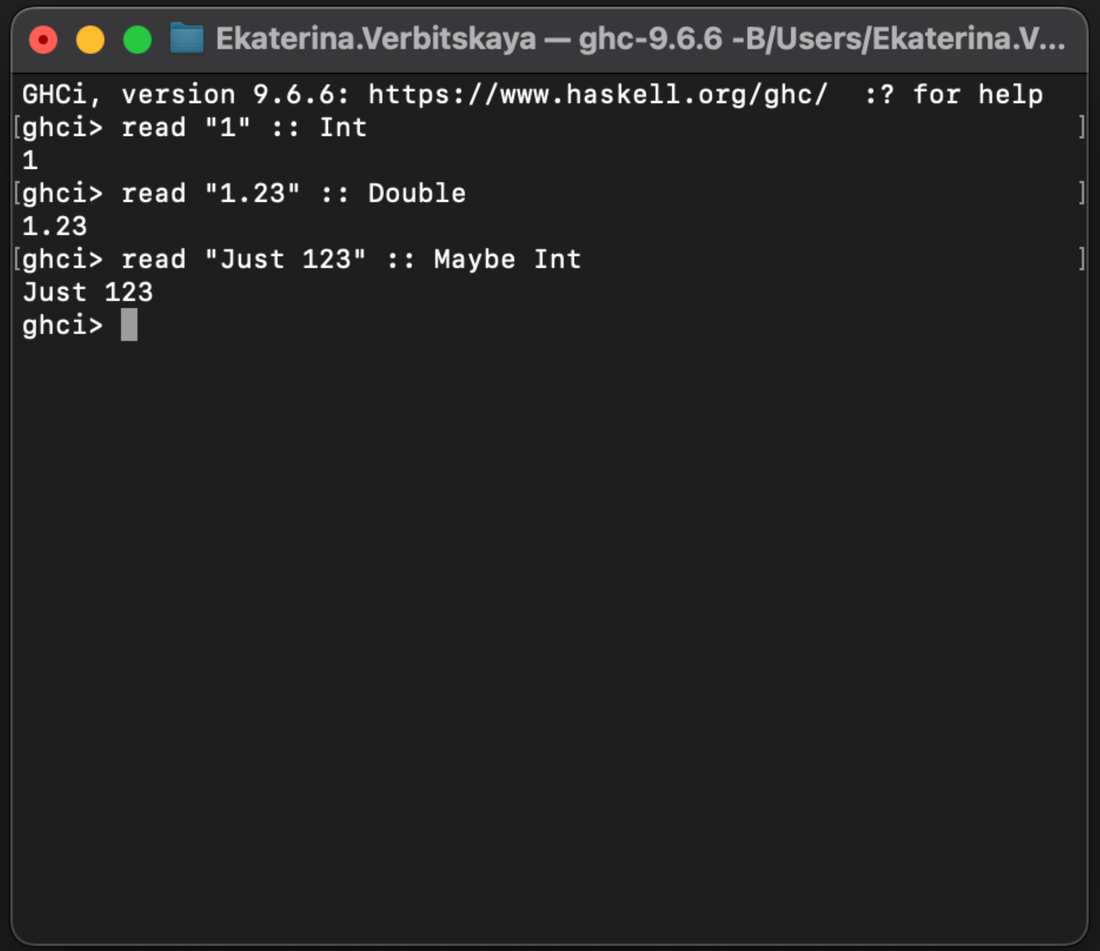
FUNCTIONAL PROGRAMMING

OUTLINE

- Read and Show

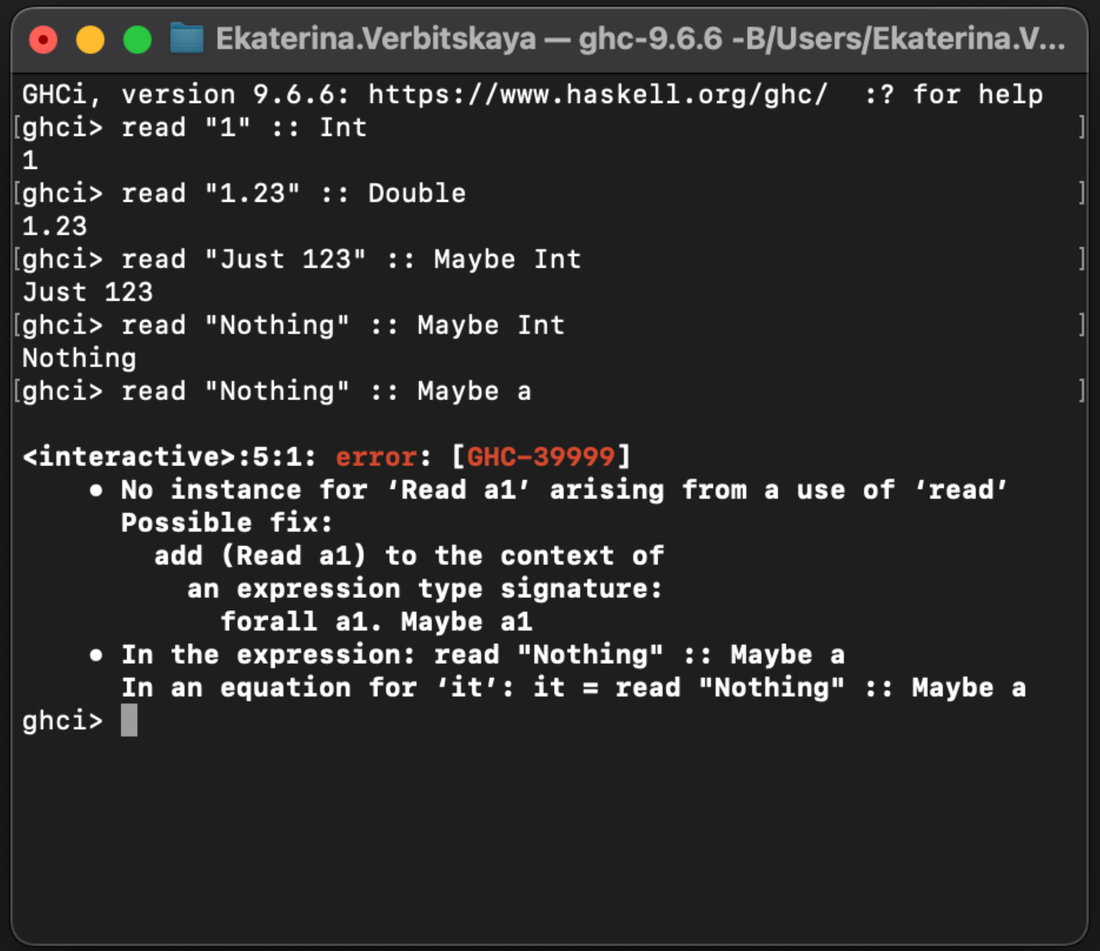
READ

- 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`



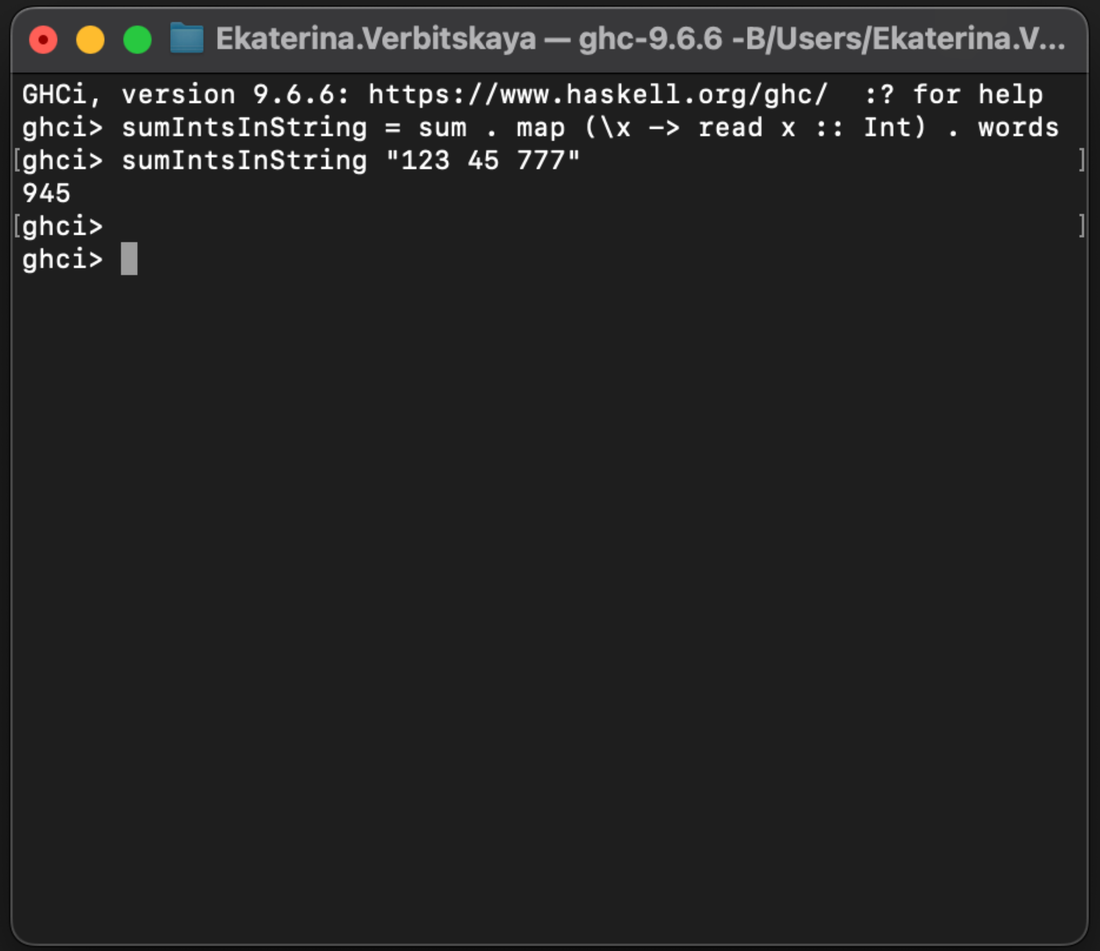
READ

- 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



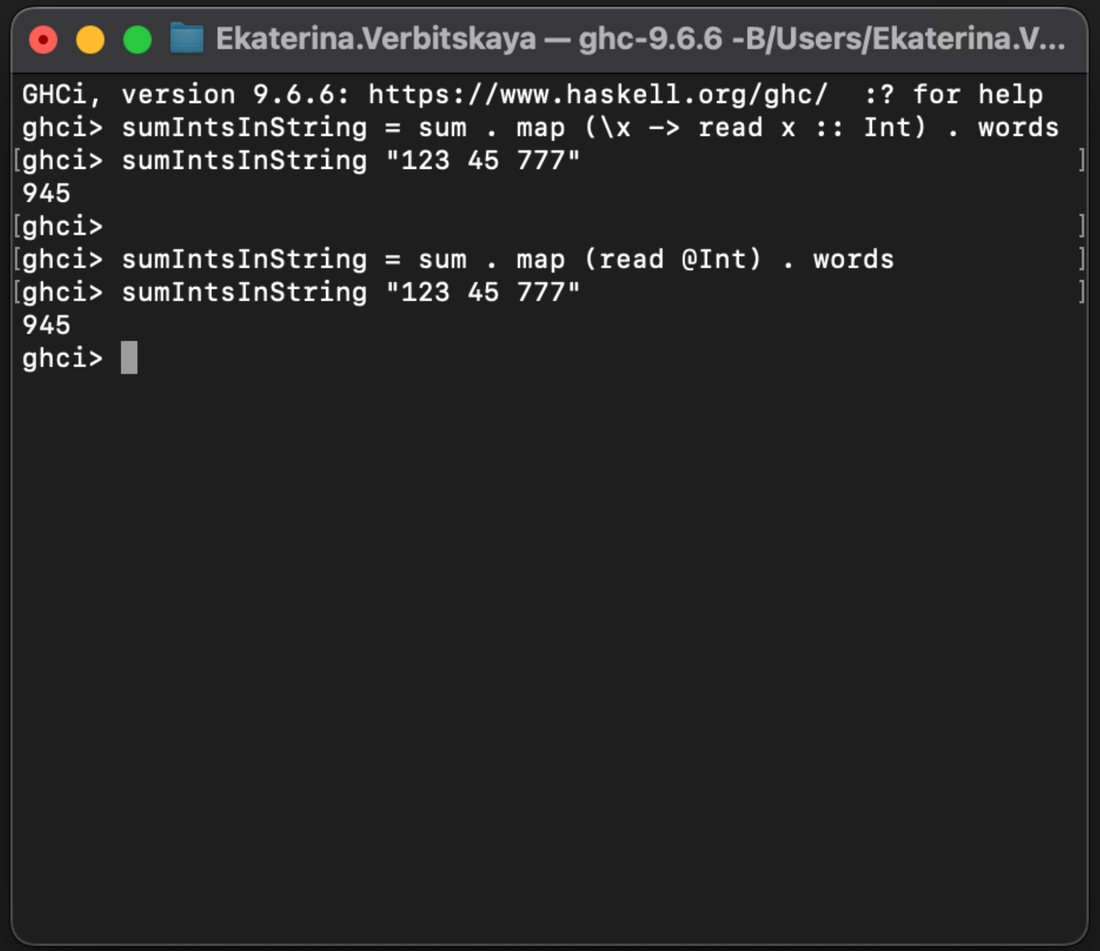
USING READ

- 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
-



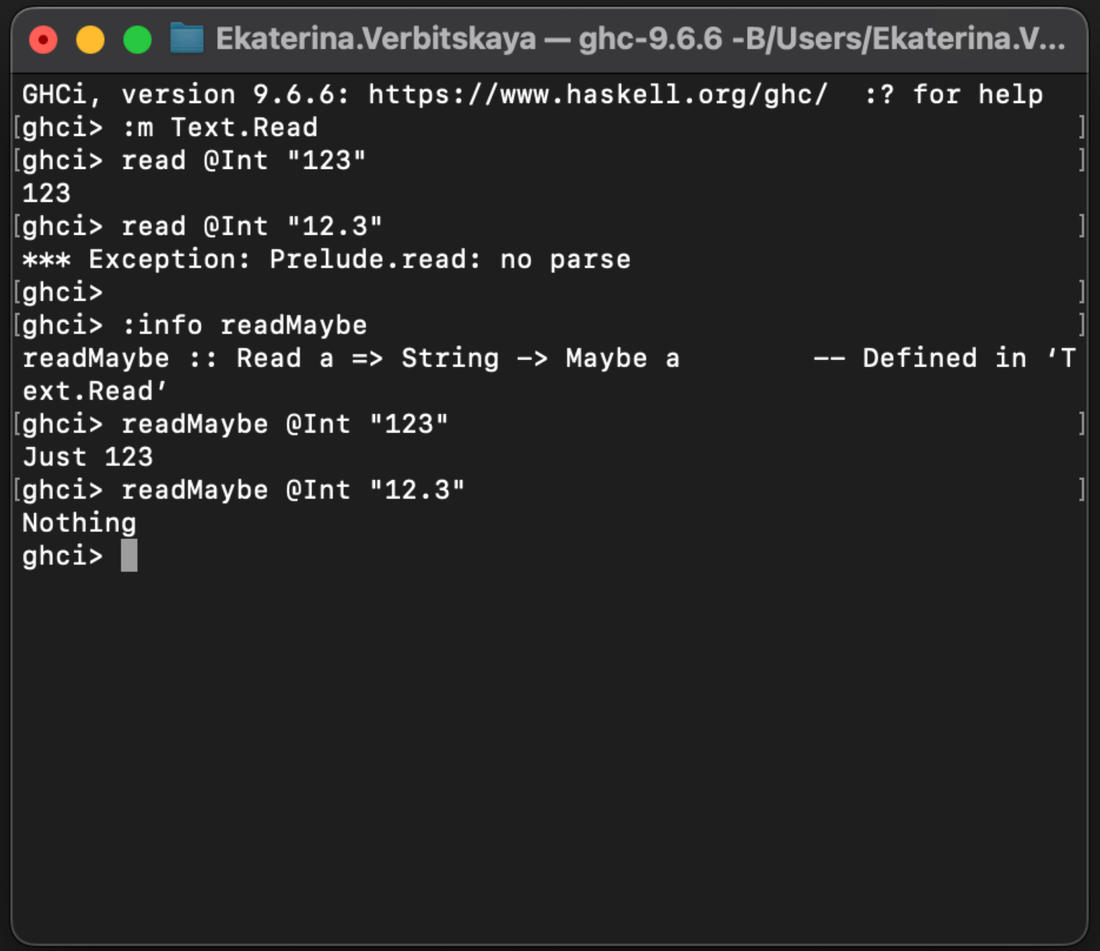
USING READ

- 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`



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`



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

```haskell
module Tree where

import Text.Read (readMaybe)

data Tree a
  = E
  | N (Tree a) a (Tree a)
  deriving (Show, Eq, Read)

insert :: Ord a => a -> Tree a -> Tree a
insert = undefined

data ParseResult
  = IntList [Int]
  | IntTree (Tree Int)

parse :: String -> Either String ParseResult
parse = undefined

makeTree :: ParseResult -> Tree Int
makeTree (IntTree x) = x
makeTree (IntList xs) = foldr insert E xs

main :: IO ()
main = do
  str <- getLine
  let input = parse str
  case input of
    Right pr -> print $ makeTree pr
    Left err -> putStrLn err
```