INTRODUCTION TO

FUNCTIONAL PROGRAMMING

OUTLINE

- Algebraic Data Types
- Type Classes
- Recursive Data Types

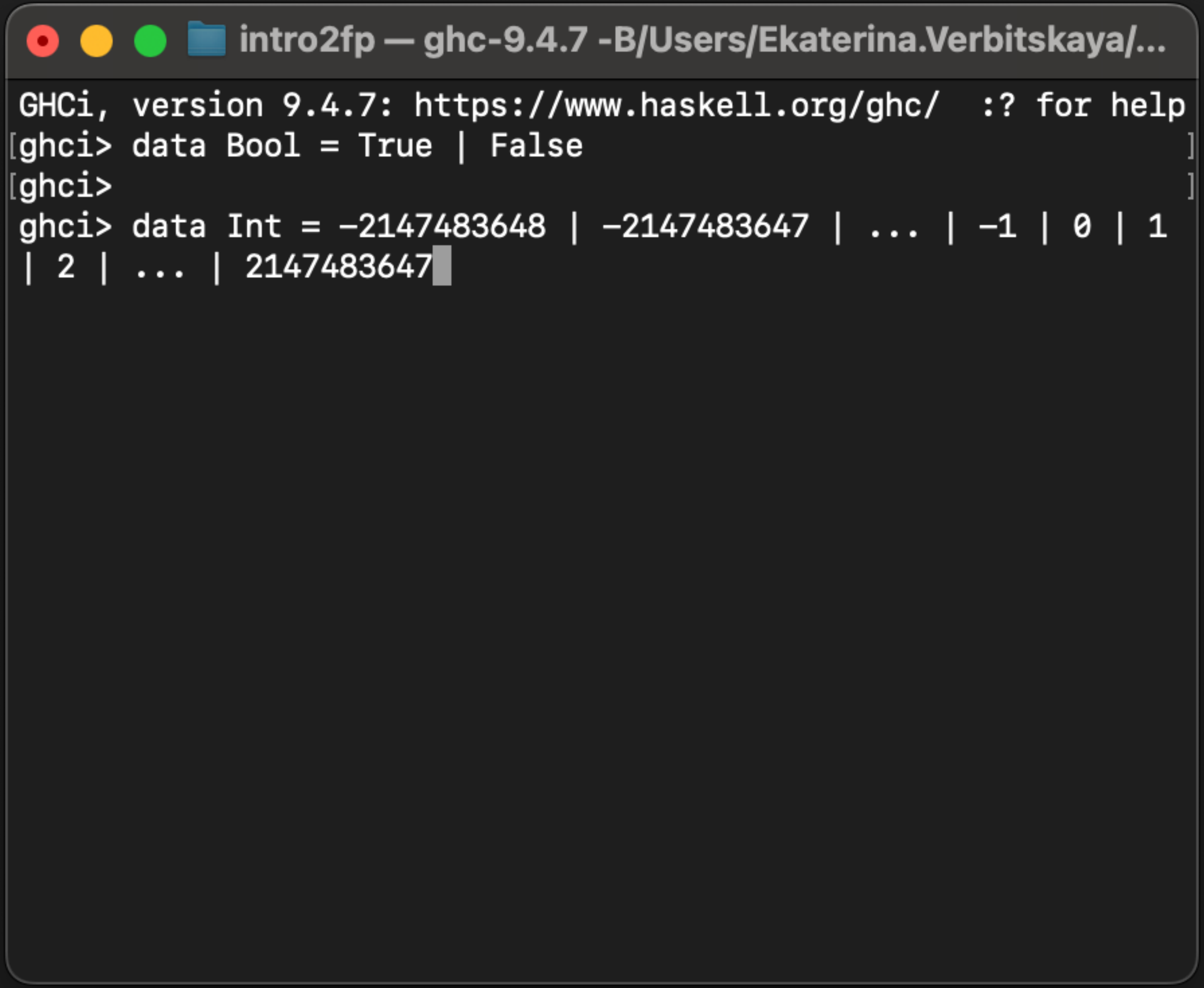
WHAT IS A TYPE?

- A collection of its values

- Bool === { True, False }
- Int === { -2147483648, -2147483647, …, 0, 1, …, 2147483647 }
- [Bool] === { [], [True], [False], [True, True], [True, False], [False, True], …}
- a -> a === { \x -> x }

STANDARD TYPES AS ADT

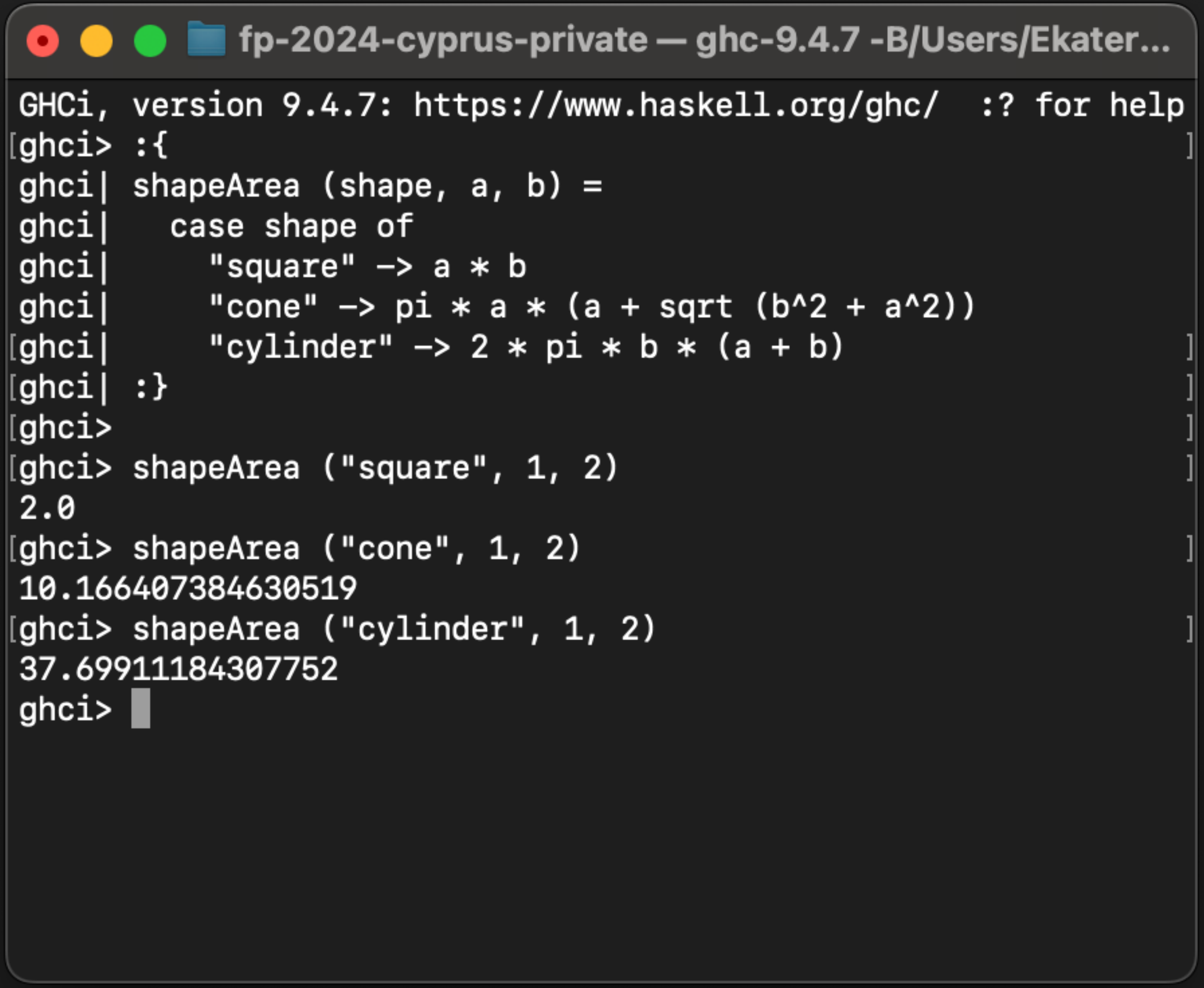
- Bool, Int – type name
- True, False, 2147483647 – values constructors
- | – “or”
- (Int is not defined like this in reality)



```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> data Bool = True | False
ghci>
ghci> data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1
 | 2 | ... | 2147483647
```

LET'S REVISIT SHAPEAREA

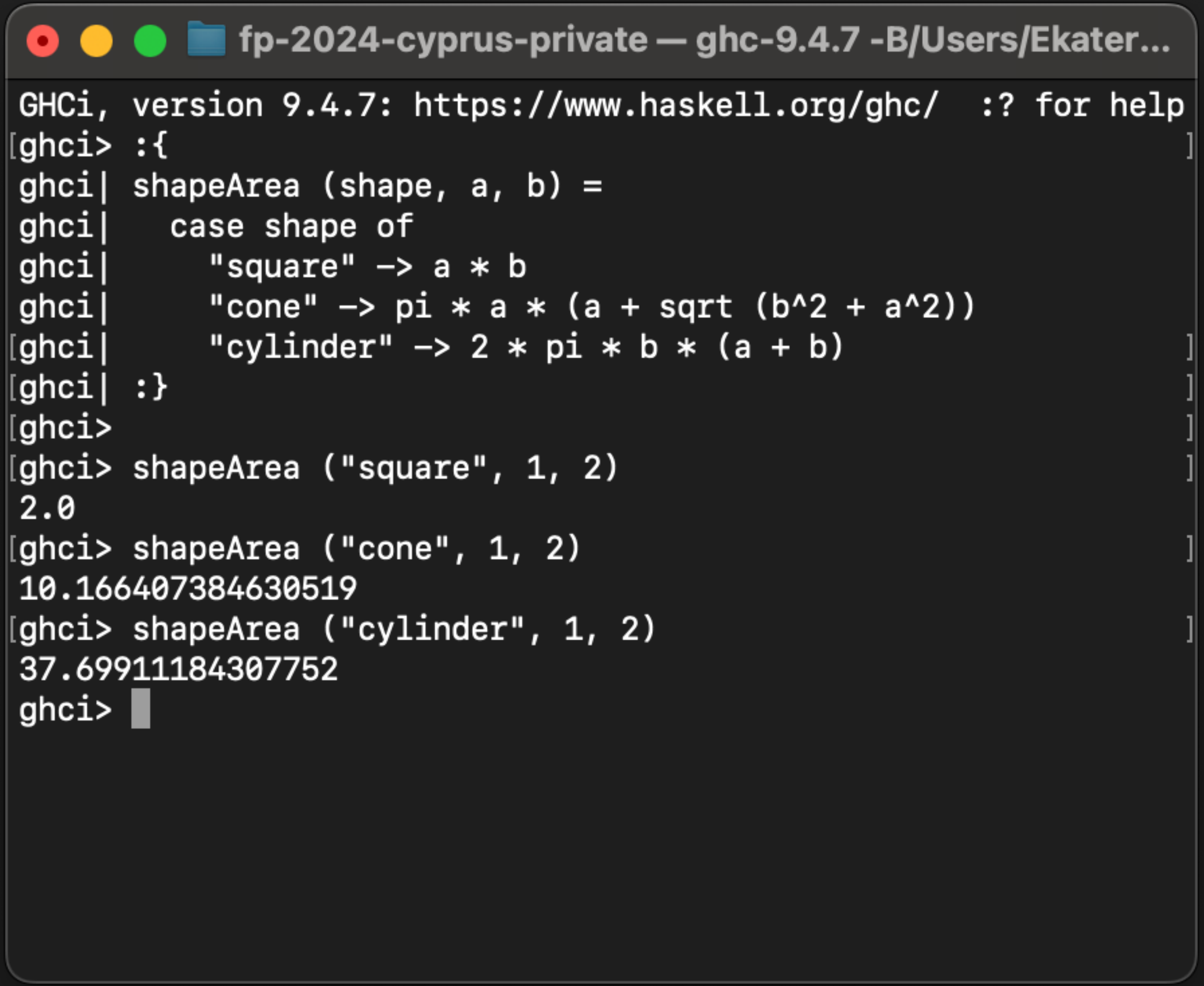
- How many issues can you spot in this code?



```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
[ghci> :{
ghci| shapeArea (shape, a, b) =
ghci|   case shape of
ghci|     "square" -> a * b
ghci|     "cone" -> pi * a * (a + sqrt (b^2 + a^2))
[ghci|     "cylinder" -> 2 * pi * b * (a + b)
[ghci| :}
[ghci>
[ghci> shapeArea ("square", 1, 2)
2.0
[ghci> shapeArea ("cone", 1, 2)
10.166407384630519
[ghci> shapeArea ("cylinder", 1, 2)
37.69911184307752
ghci> 
```

LET'S REVISIT SHAPEAREA

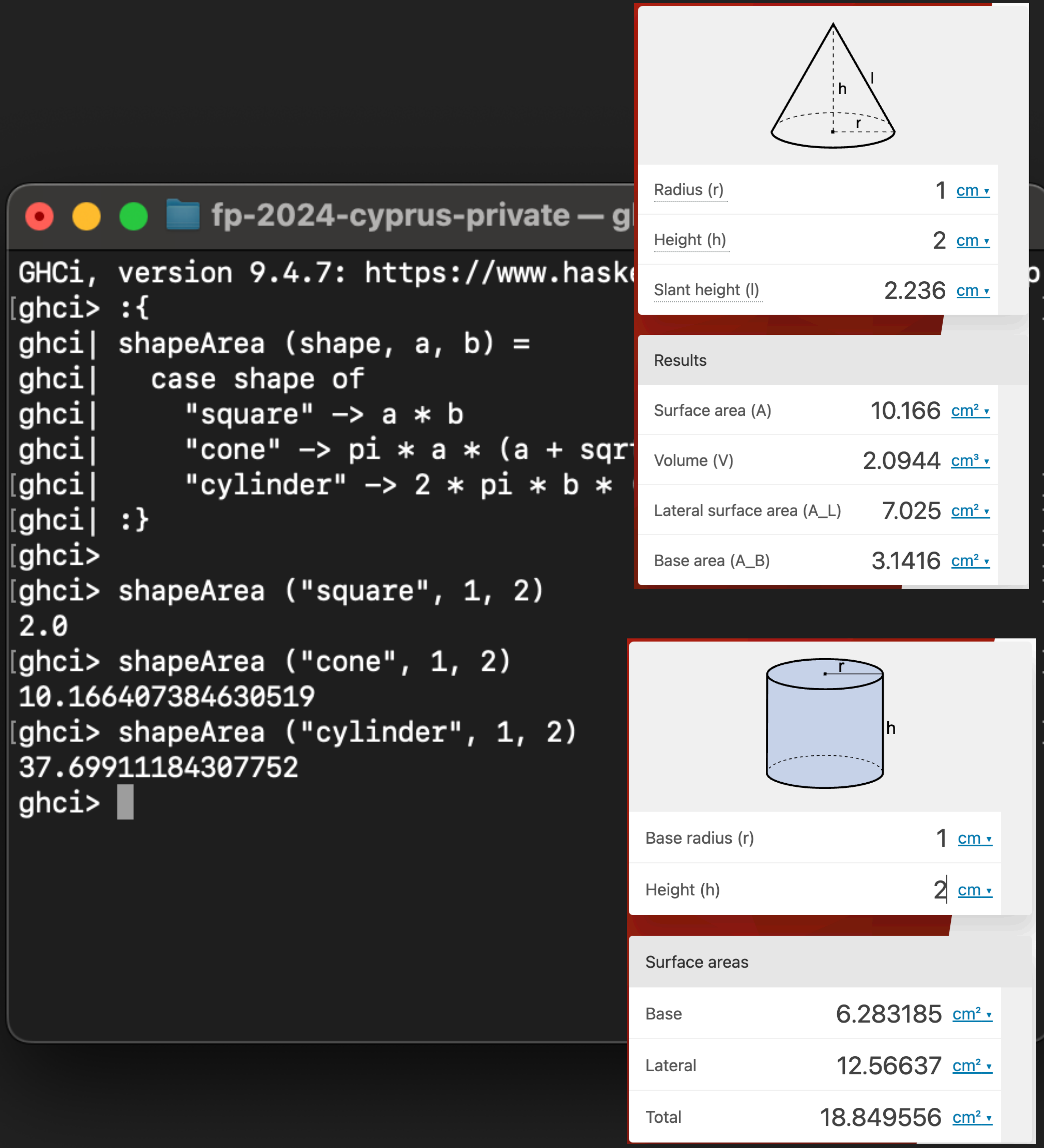
- How many issues can you spot in this code?
- Why square has two sides?
- What are a and b?
- What about case sensitivity?
- What if we get a "rectangle" or other string?



```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :{
ghci| shapeArea (shape, a, b) =
ghci|   case shape of
ghci|     "square" -> a * b
ghci|     "cone" -> pi * a * (a + sqrt (b^2 + a^2))
ghci|     "cylinder" -> 2 * pi * b * (a + b)
ghci| :}
ghci>
ghci> shapeArea ("square", 1, 2)
2.0
ghci> shapeArea ("cone", 1, 2)
10.166407384630519
ghci> shapeArea ("cylinder", 1, 2)
37.69911184307752
ghci> 
```

LET'S REVISIT SHAPEAREA

- How many issues can you spot in this code?
- Why square has two sides?
- What are a and b?
- What about case sensitivity?
- What if we get a "rectangle" or other string?
- Stop, it computes wrong values



```
GHCi, version 9.4.7: https://www.hask
[ghci> :{
ghci| shapeArea (shape, a, b) =
ghci|   case shape of
ghci|     "square" -> a * b
ghci|     "cone" -> pi * a * (a + sqr
[ghci|     "cylinder" -> 2 * pi * b *
[ghci| :}
[ghci>
[ghci> shapeArea ("square", 1, 2)
2.0
[ghci> shapeArea ("cone", 1, 2)
10.166407384630519
[ghci> shapeArea ("cylinder", 1, 2)
37.69911184307752
ghci>
```

Radius (r) 1 cm
Height (h) 2 cm
Slant height (l) 2.236 cm

Results

Surface area (A) 10.166 cm²
Volume (V) 2.0944 cm³
Lateral surface area (A_L) 7.025 cm²
Base area (A_B) 3.1416 cm²

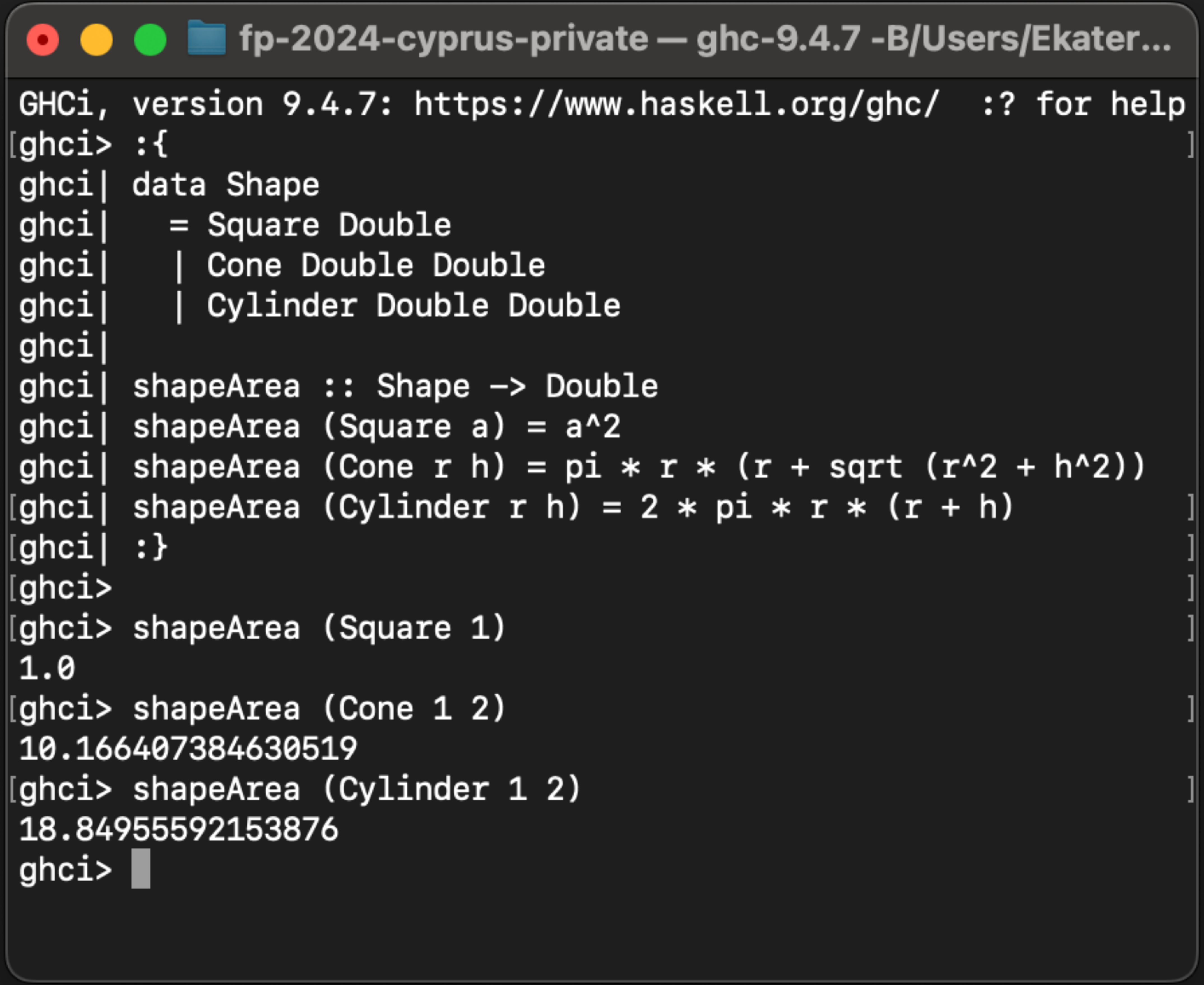
Base radius (r) 1 cm
Height (h) 2 cm

Surface areas

Base 6.283185 cm²
Lateral 12.56637 cm²
Total 18.849556 cm²

BETTER WAY TO DESIGN SHAPE

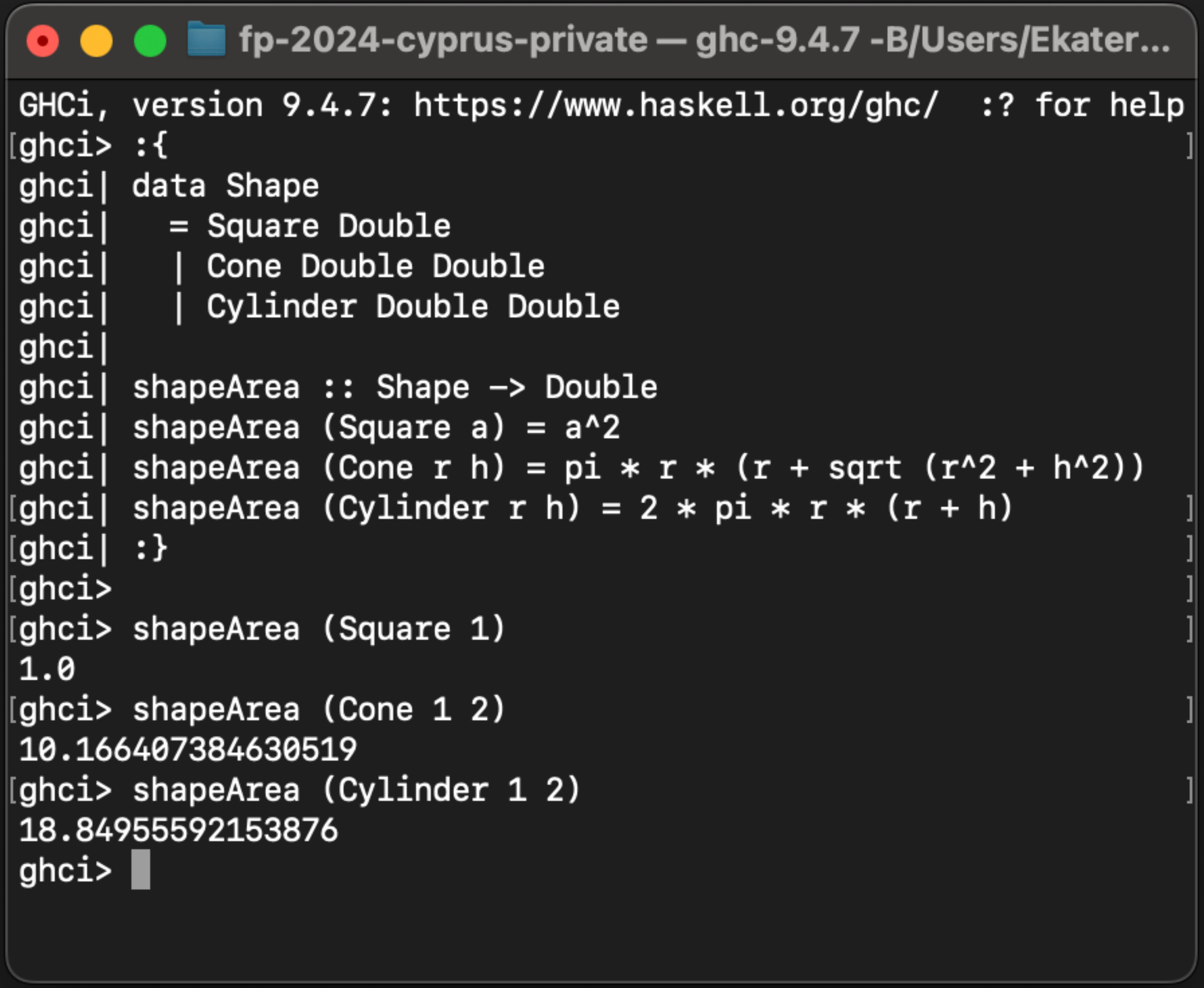
- A square is square at last
- No other shape can be created and passed to the function
- It's still hard to distinguish between r and h



```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
[ghci> :{
ghci| data Shape
ghci|   = Square Double
ghci|   | Cone Double Double
ghci|   | Cylinder Double Double
ghci|
ghci| shapeArea :: Shape -> Double
ghci| shapeArea (Square a) = a^2
ghci| shapeArea (Cone r h) = pi * r * (r + sqrt (r^2 + h^2))
[ghci| shapeArea (Cylinder r h) = 2 * pi * r * (r + h)
[ghci| :}
[ghci>
[ghci> shapeArea (Square 1)
1.0
[ghci> shapeArea (Cone 1 2)
10.166407384630519
[ghci> shapeArea (Cylinder 1 2)
18.84955592153876
ghci> 
```

PROPERTIES OF ADTS

- Distinctness
  - $\forall j \neq i \,.\, C_i^n(x) \neq C_j^n(y)$
- Injectivity
  - $C_i^n(x_1, \ldots, x_n) = C_j^n(y_1, \ldots, y_n) \Rightarrow \forall k \,.\, x_k = y_k$
- Exhaustiveness
  - x of ADT $\Rightarrow \exists i \,.\, x = C_i^n(y_1, \ldots, y_n)$
- Selection
  - $\exists s_i^k \,.\, s_i^k(C_k^n(x_1, \ldots, x_n)) = x_i$



```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :{
ghci| data Shape
ghci|   = Square Double
ghci|   | Cone Double Double
ghci|   | Cylinder Double Double
ghci|
ghci| shapeArea :: Shape -> Double
ghci| shapeArea (Square a) = a^2
ghci| shapeArea (Cone r h) = pi * r * (r + sqrt (r^2 + h^2))
ghci| shapeArea (Cylinder r h) = 2 * pi * r * (r + h)
ghci| :}
ghci>
ghci> shapeArea (Square 1)
1.0
ghci> shapeArea (Cone 1 2)
10.166407384630519
ghci> shapeArea (Cylinder 1 2)
18.84955592153876
ghci>
```

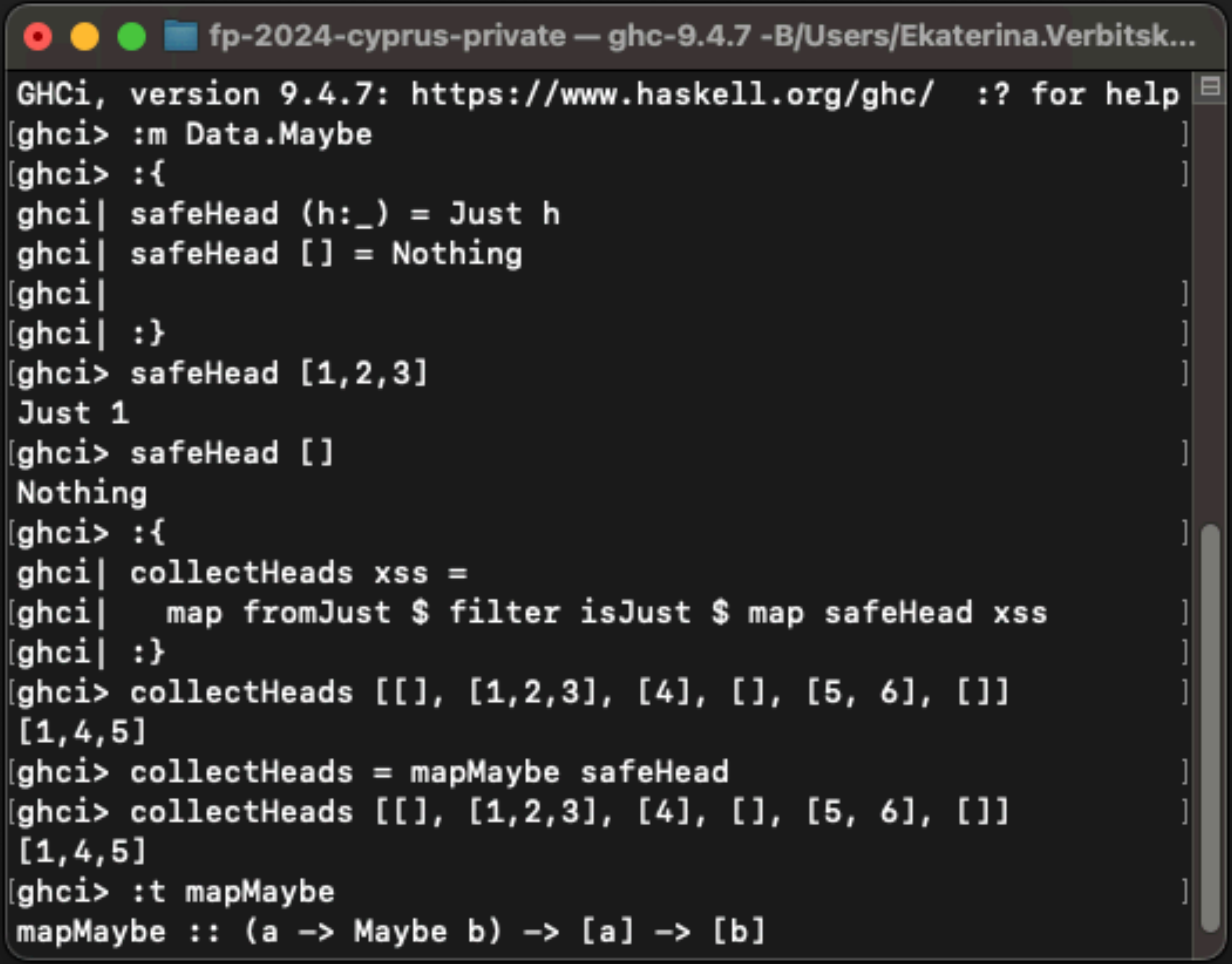
FAILING COMPUTATIONS: MAYBE

- data Maybe a = Just a | Nothing
  - from Data.Maybe

- A way to fix partiality of a function

- Only use it when there is a single way for a function to fail

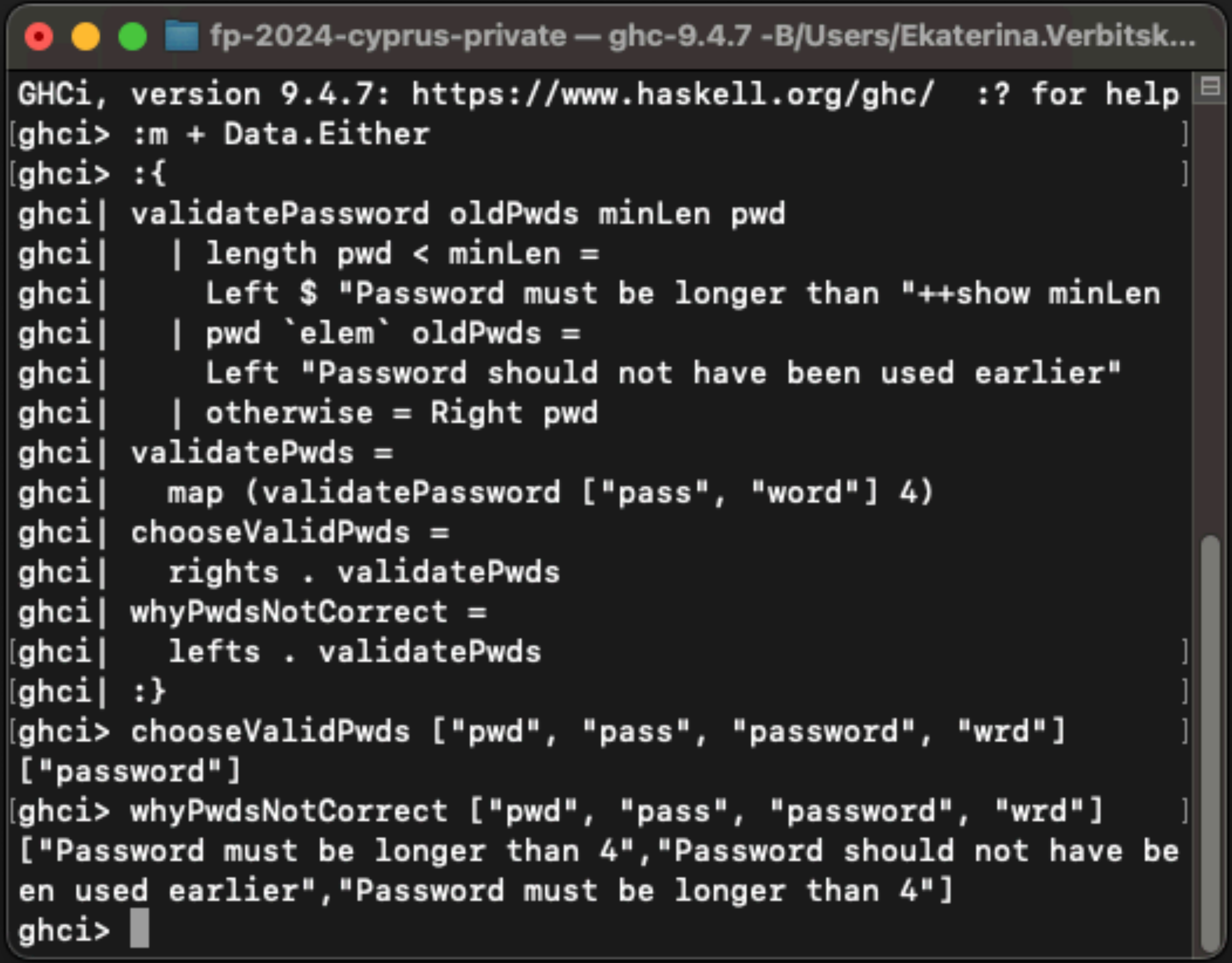
- isJust, isNothing, fromJust, fromMaybe, listToMaybe, catMaybes, mapMaybes, maybe


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
[ghci> :m Data.Maybe
[ghci> :{
ghci| safeHead (h:_) = Just h
ghci| safeHead [] = Nothing
[ghci|
[ghci| :}
[ghci> safeHead [1,2,3]
Just 1
[ghci> safeHead []
Nothing
[ghci> :{
ghci| collectHeads xss =
[ghci|   map fromJust $ filter isJust $ map safeHead xss
[ghci| :}
[ghci> collectHeads [[], [1,2,3], [4], [], [5, 6], []]
[1,4,5]
[ghci> collectHeads = mapMaybe safeHead
[ghci> collectHeads [[], [1,2,3], [4], [], [5, 6], []]
[1,4,5]
[ghci> :t mapMaybe
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```

FAILING COMPUTATIONS: EITHER

- data Either a b = Left a | Right b
  - from Data.Either
- Right for the right answer
- Left for fail
- lefts, rights, isLeft, isRight, fromLeft, fromRight, partitionEithers


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :m + Data.Either
ghci> :{
ghci| validatePassword oldPwds minLen pwd
ghci|   | length pwd < minLen =
ghci|     Left $ "Password must be longer than "++show minLen
ghci|   | pwd `elem` oldPwds =
ghci|     Left "Password should not have been used earlier"
ghci|   | otherwise = Right pwd
ghci| validatePwds =
ghci|   map (validatePassword ["pass", "word"] 4)
ghci| chooseValidPwds =
ghci|   rights . validatePwds
ghci| whyPwdsNotCorrect =
ghci|   lefts . validatePwds
ghci| :}
ghci> chooseValidPwds ["pwd", "pass", "password", "wrd"]
["password"]
ghci> whyPwdsNotCorrect ["pwd", "pass", "password", "wrd"]
["Password must be longer than 4","Password should not have be
en used earlier","Password must be longer than 4"]
ghci> 
```

EXERCISES

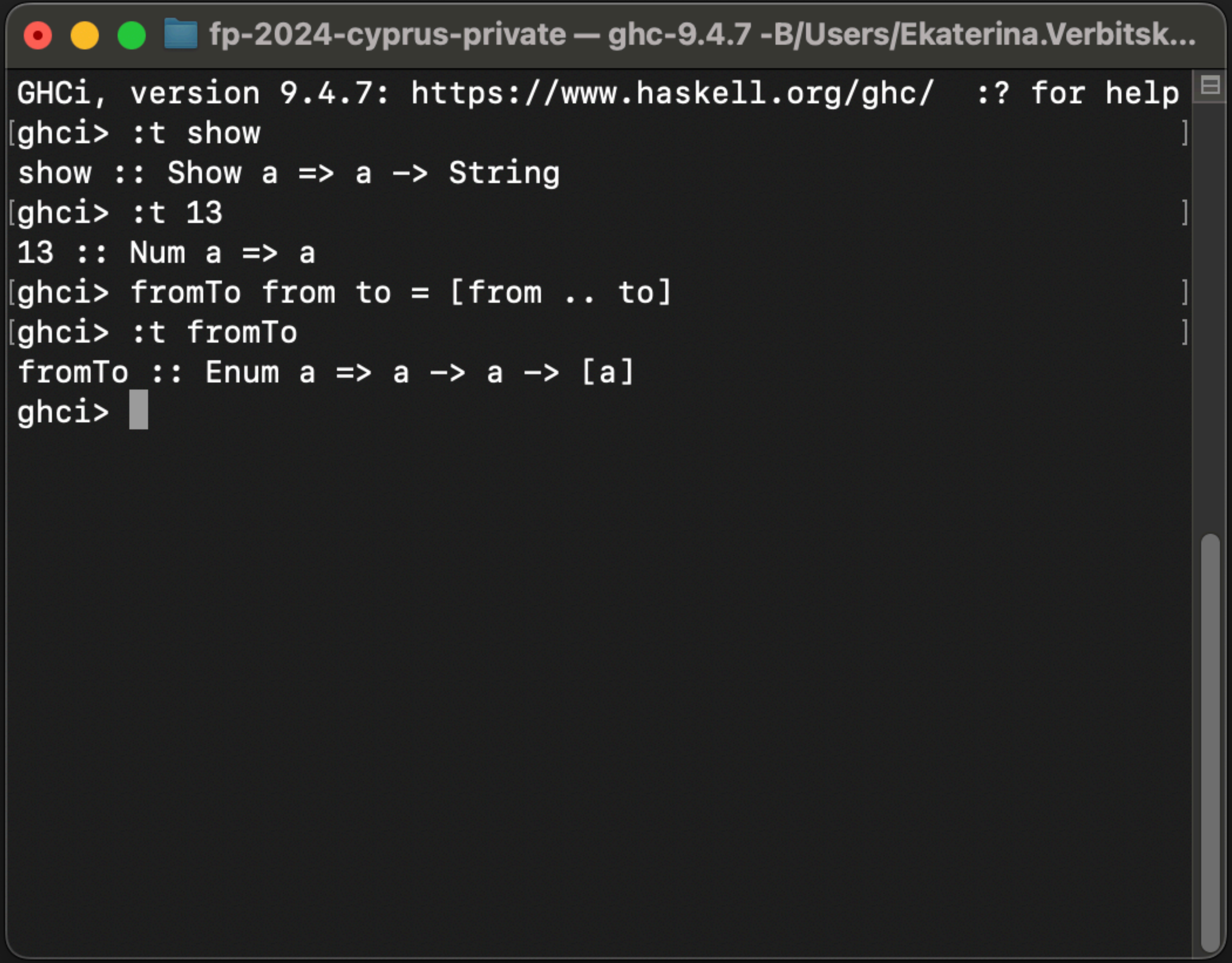
- Fix partially applied functions in HW01: pick the best suiting way to represent failing computations

OUTLINE

- Algebraic Data Types
- Type Classes
- Recursive Data Types

WE'VE SEEN THEM BEFORE

- Everything before => is a constraint
- Describes behaviour through available functions

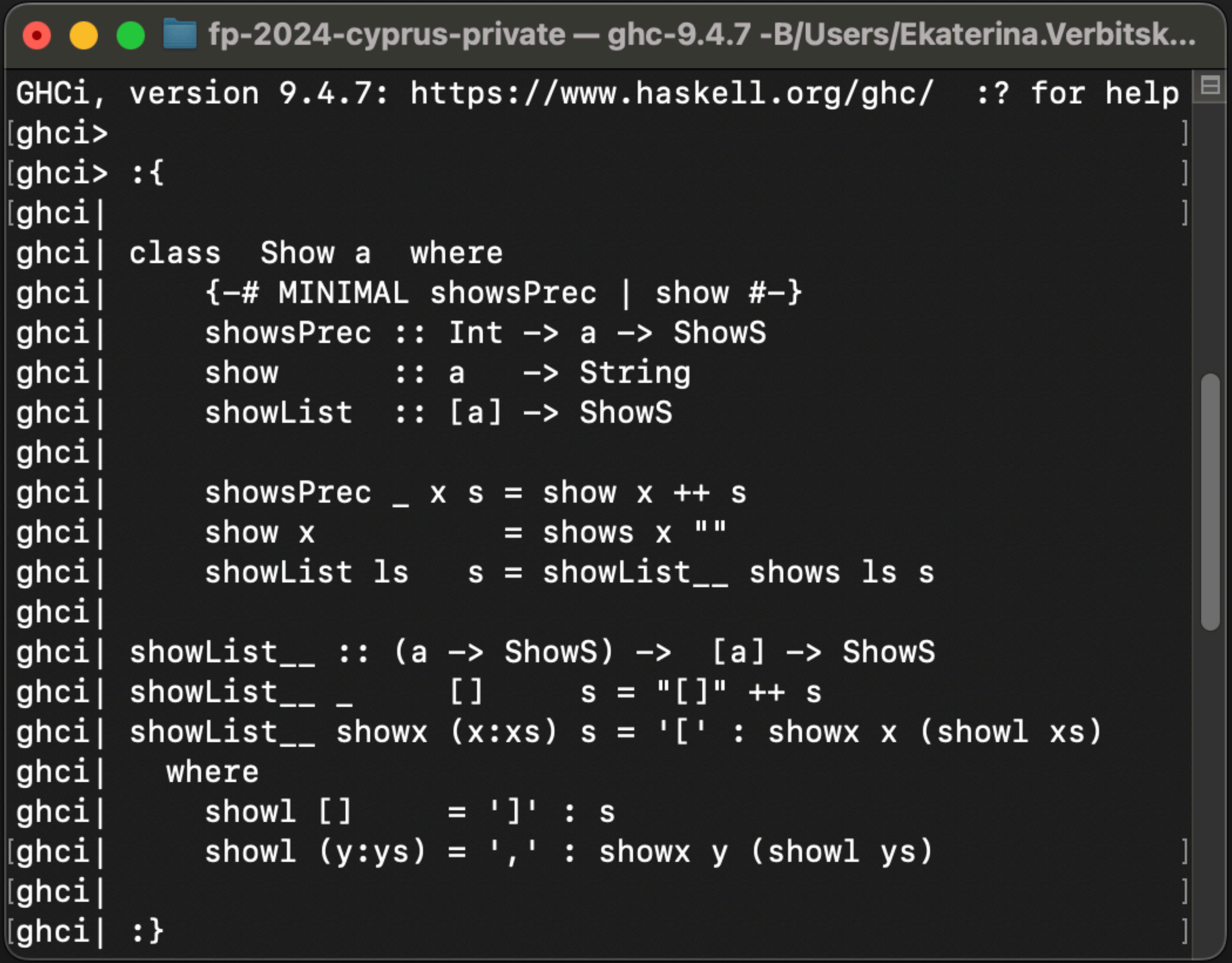


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :t show
show :: Show a => a -> String
ghci> :t 13
13 :: Num a => a
ghci> fromTo from to = [from .. to]
ghci> :t fromTo
fromTo :: Enum a => a -> a -> [a]
ghci>
```

CLASS DEFINITION SYNTAX

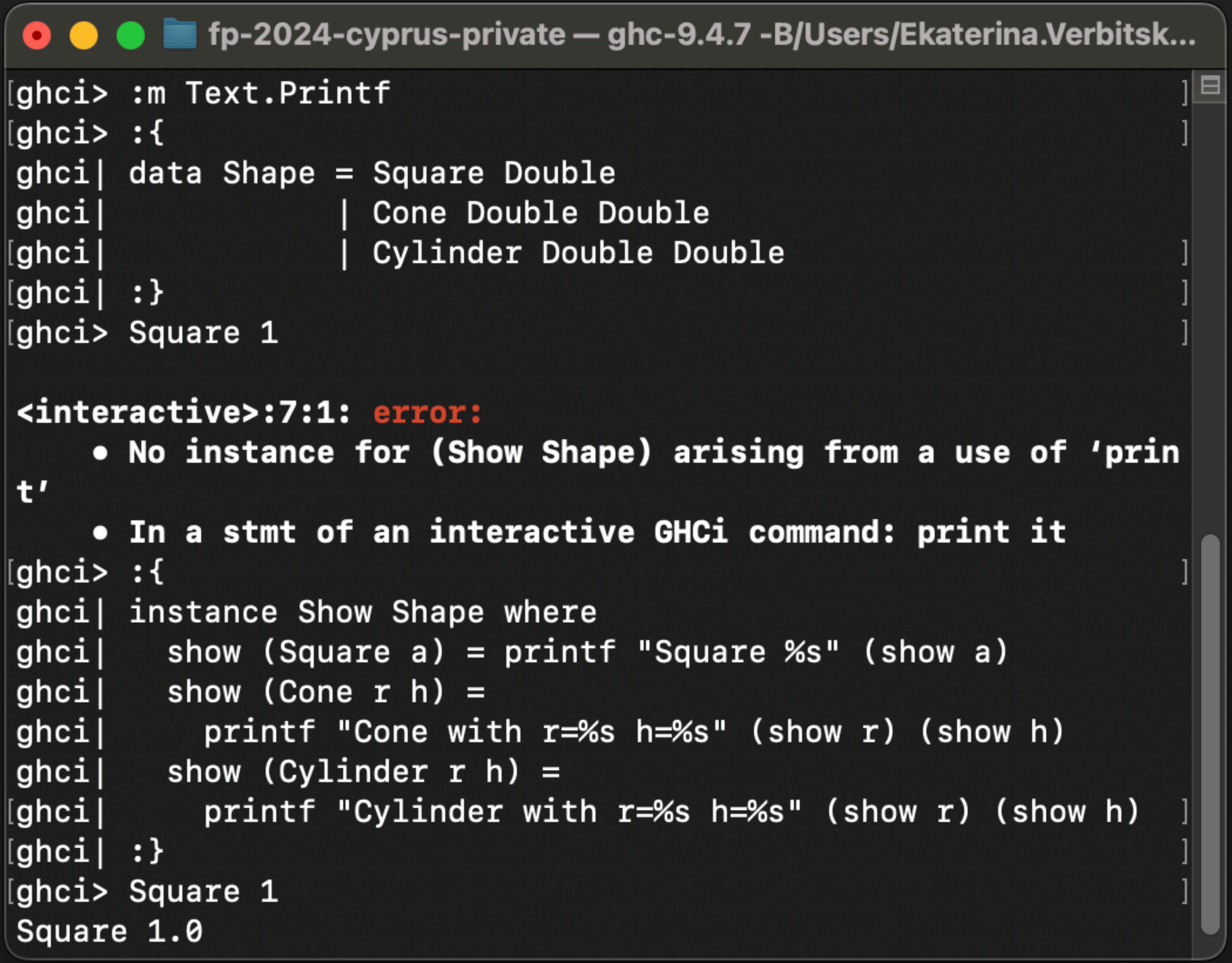
- {-# MINIMAL … #-} describes which functions should be implemented in any instance
- Then goes a list of functions of a type class
- Some (or all) functions can have default implementations


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci>
ghci> :{
ghci|
ghci| class  Show a  where
ghci|     {-# MINIMAL showsPrec | show #-}
ghci|     showsPrec :: Int -> a -> ShowS
ghci|     show      :: a    -> String
ghci|     showList  :: [a] -> ShowS
ghci|
ghci|     showsPrec _ x s = show x ++ s
ghci|     show x          = shows x ""
ghci|     showList ls   s = showList__ shows ls s
ghci|
ghci| showList__ :: (a -> ShowS) ->  [a] -> ShowS
ghci| showList__ _     []     s = "[]" ++ s
ghci| showList__ showx (x:xs) s = '[' : showx x (showl xs)
ghci|   where
ghci|     showl []     = ']' : s
ghci|     showl (y:ys) = ',' : showx y (showl ys)
ghci|
ghci| :}
```

INSTANCES

- When you type an expression into ghci, it calls show on the expression, so it assumes Show
- You need to provide an instance – the implementation of Show
- You need to implement at least minimal functions, but you can implement all functions of a type class

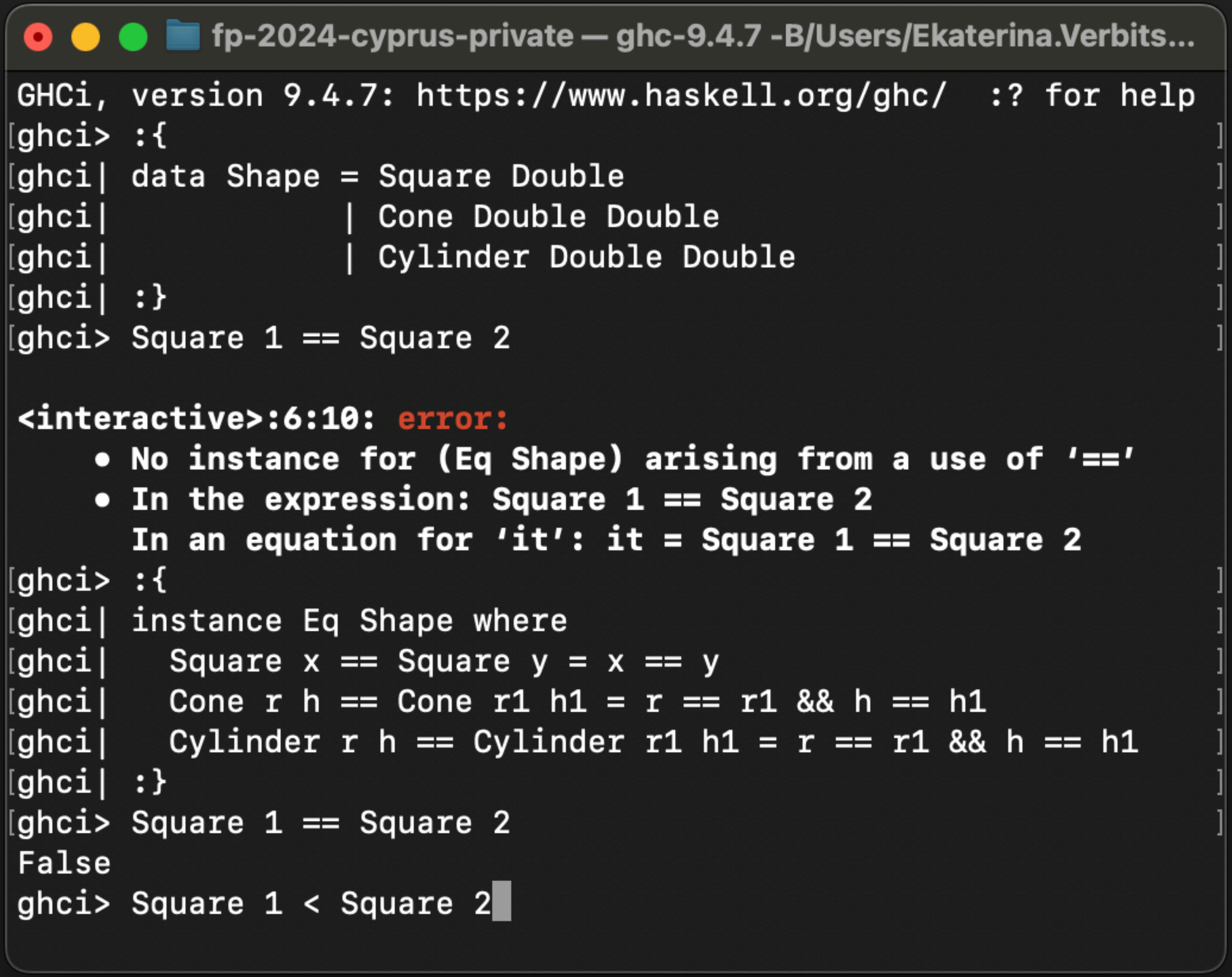


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
ghci> :m Text.Printf
ghci> :{
ghci| data Shape = Square Double
ghci|            | Cone Double Double
ghci|            | Cylinder Double Double
ghci| :}
ghci> Square 1

<interactive>:7:1: error:
    • No instance for (Show Shape) arising from a use of ‘prin
t’
    • In a stmt of an interactive GHCi command: print it
ghci> :{
ghci| instance Show Shape where
ghci|   show (Square a) = printf "Square %s" (show a)
ghci|   show (Cone r h) =
ghci|     printf "Cone with r=%s h=%s" (show r) (show h)
ghci|   show (Cylinder r h) =
ghci|     printf "Cylinder with r=%s h=%s" (show r) (show h)
ghci| :}
ghci> Square 1
Square 1.0
```

EQ TYPE CLASS

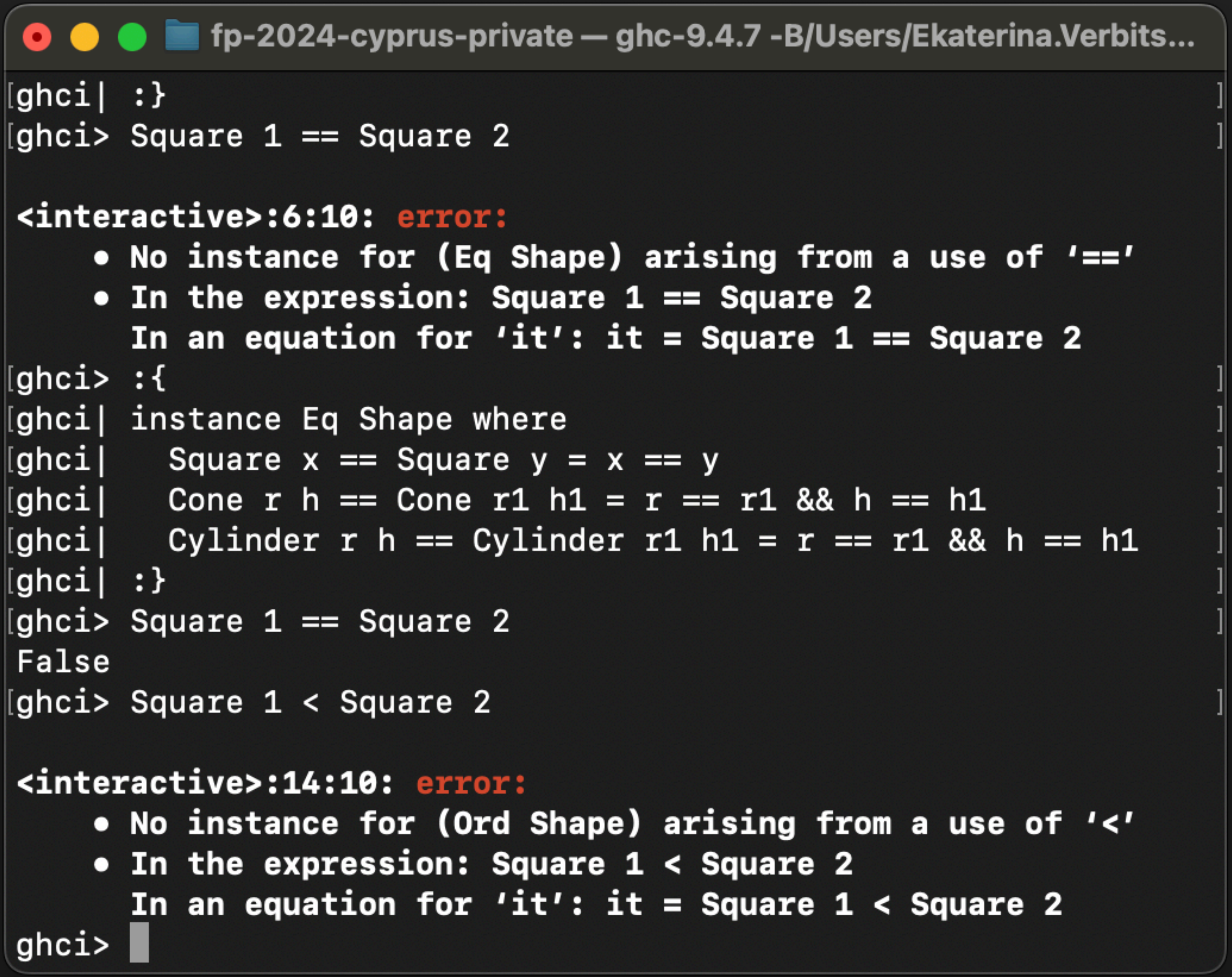
- Not everything can be checked for equality, only instances of Eq
- See documentation
- What will happen when I enter the expression Square 1 < Square 2?


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbits...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
[ghci> :{
[ghci| data Shape = Square Double
[ghci|            | Cone Double Double
[ghci|            | Cylinder Double Double
[ghci| :}
[ghci> Square 1 == Square 2

<interactive>:6:10: error:
    • No instance for (Eq Shape) arising from a use of ‘==’
    • In the expression: Square 1 == Square 2
      In an equation for ‘it’: it = Square 1 == Square 2
[ghci> :{
[ghci| instance Eq Shape where
[ghci|   Square x == Square y = x == y
[ghci|   Cone r h == Cone r1 h1 = r == r1 && h == h1
[ghci|   Cylinder r h == Cylinder r1 h1 = r == r1 && h == h1
[ghci| :}
[ghci> Square 1 == Square 2
False
ghci> Square 1 < Square 2
```

EQ TYPE CLASS

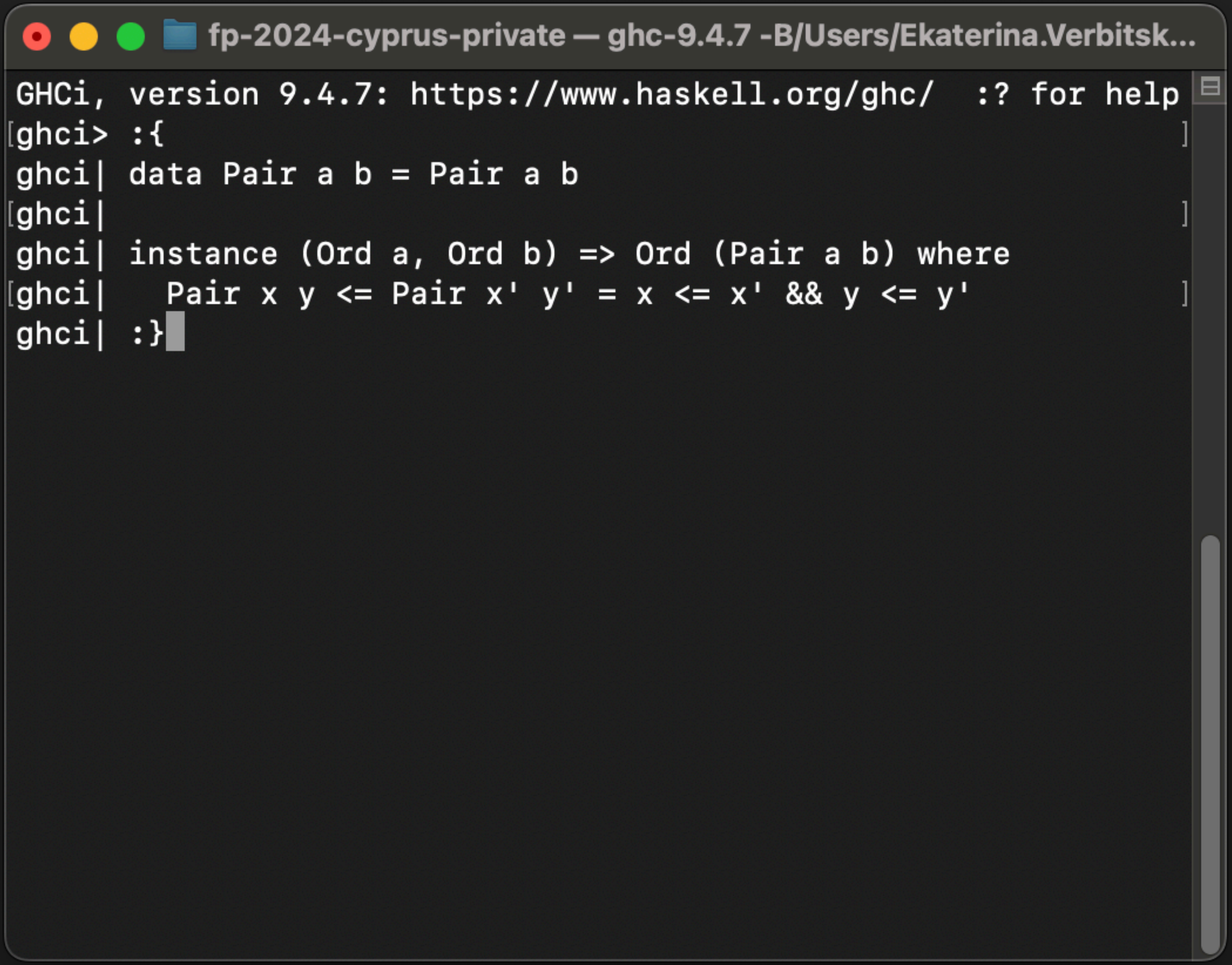
- Not everything can be checked for equality, only instances of Eq
- See documentation
- What will happen when I enter the expression Square 1 < Square 2?
- Error, because Eq has nothing to do with comparisons


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbits...

```
[ghci| :}
[ghci> Square 1 == Square 2

<interactive>:6:10: error:
    • No instance for (Eq Shape) arising from a use of '=='
    • In the expression: Square 1 == Square 2
      In an equation for 'it': it = Square 1 == Square 2
[ghci> :{
[ghci| instance Eq Shape where
[ghci|   Square x == Square y = x == y
[ghci|   Cone r h == Cone r1 h1 = r == r1 && h == h1
[ghci|   Cylinder r h == Cylinder r1 h1 = r == r1 && h == h1
[ghci| :}
[ghci> Square 1 == Square 2
False
[ghci> Square 1 < Square 2

<interactive>:14:10: error:
    • No instance for (Ord Shape) arising from a use of '<'
    • In the expression: Square 1 < Square 2
      In an equation for 'it': it = Square 1 < Square 2
ghci> 
```

ORD TYPE CLASS

- Ord: when you want to compare stuff
- See documentation
- Let’s make a custom Pair an instance of Ord



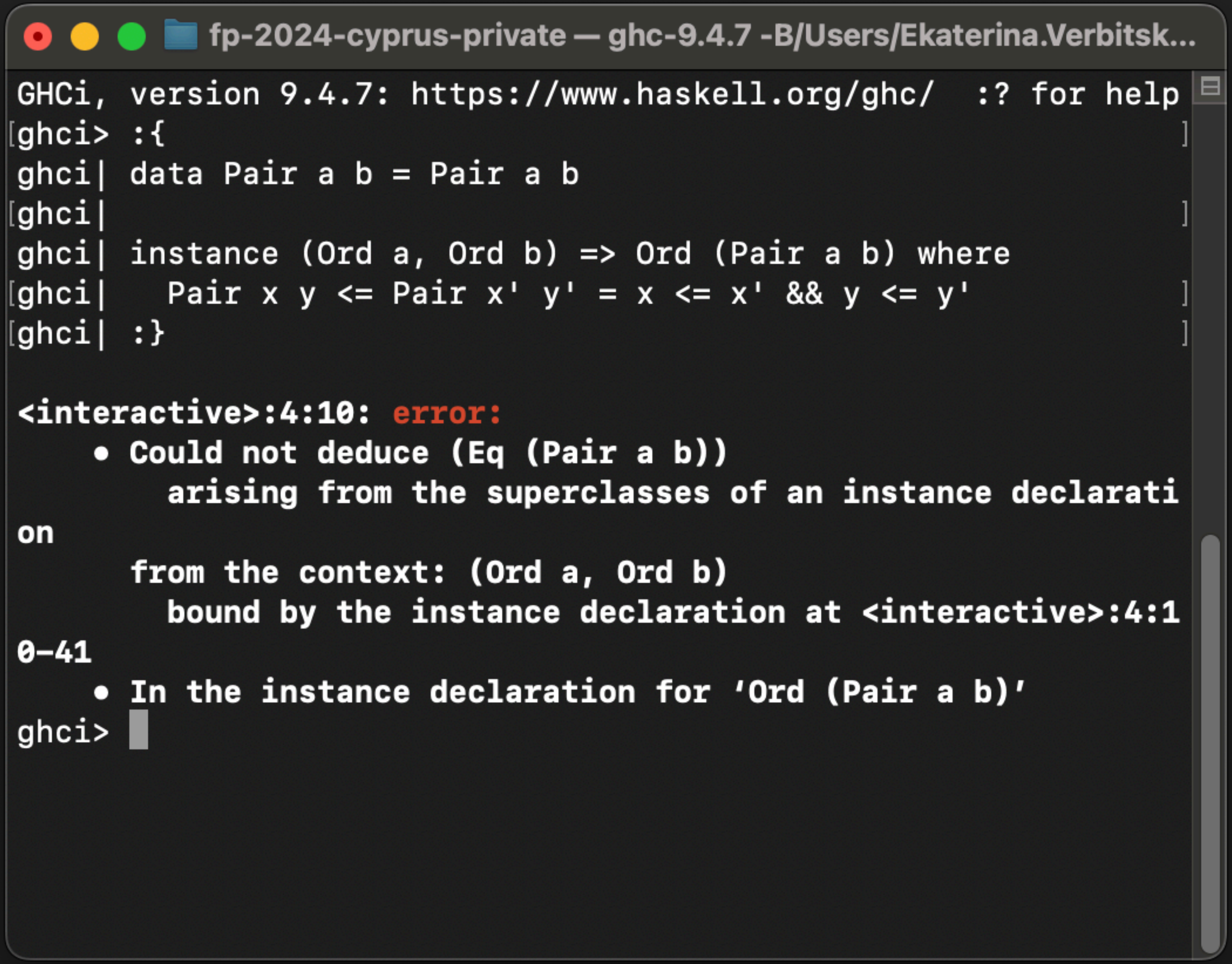
fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :{
ghci| data Pair a b = Pair a b
ghci|
ghci| instance (Ord a, Ord b) => Ord (Pair a b) where
ghci|   Pair x y <= Pair x' y' = x <= x' && y <= y'
ghci| :}
```

ORD TYPE CLASS

- Ord: when you want to compare stuff
- See documentation

- Let's make a custom Pair an instance of Ord
- Oops: we need to make it an instance of Eq

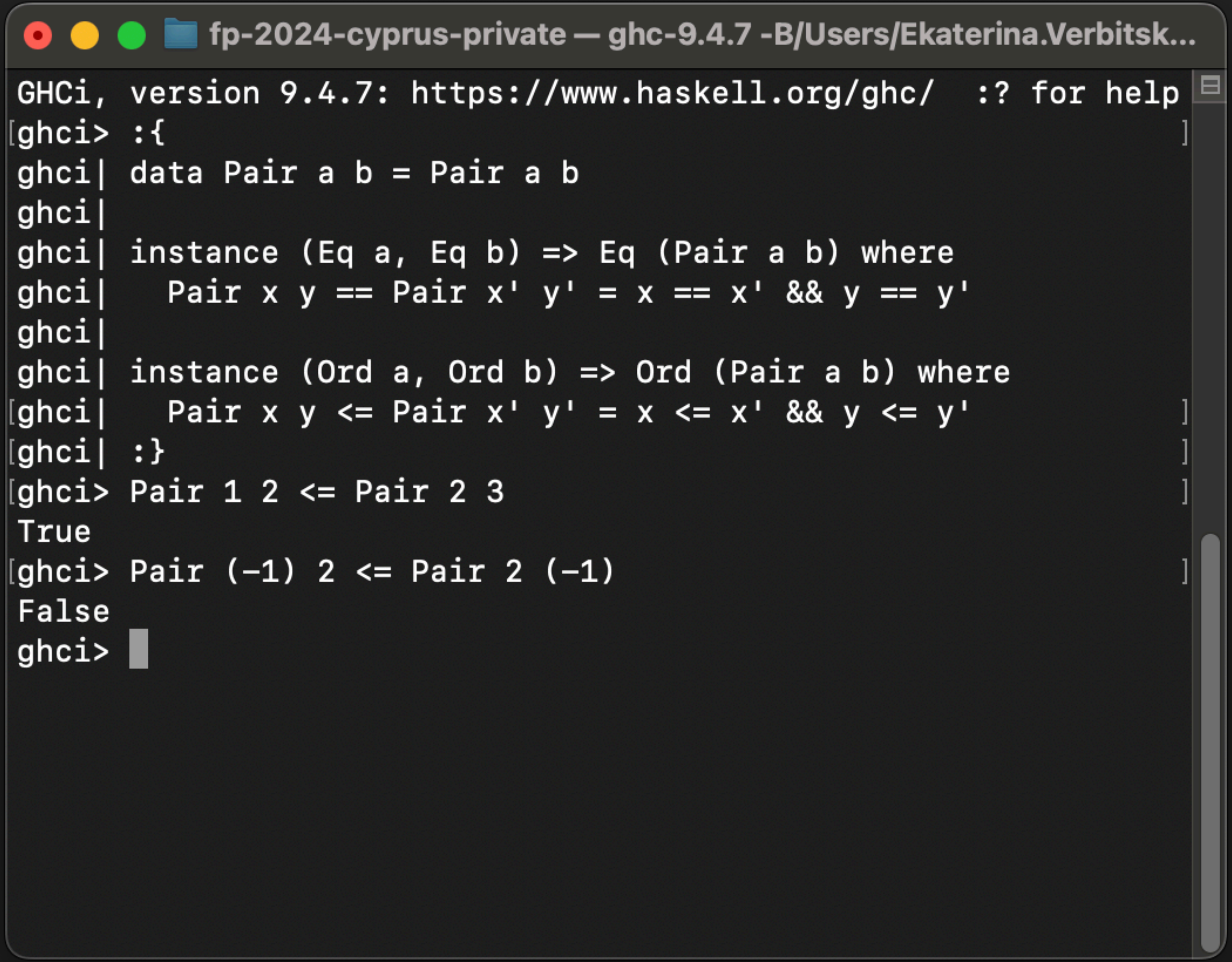

fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :{
ghci| data Pair a b = Pair a b
ghci|
ghci| instance (Ord a, Ord b) => Ord (Pair a b) where
ghci|   Pair x y <= Pair x' y' = x <= x' && y <= y'
ghci| :}

<interactive>:4:10: error:
    • Could not deduce (Eq (Pair a b))
        arising from the superclasses of an instance declarati
on
      from the context: (Ord a, Ord b)
        bound by the instance declaration at <interactive>:4:1
0-41
    • In the instance declaration for ‘Ord (Pair a b)’
ghci> 
```

ORD TYPE CLASS

- Ord: when you want to compare stuff
- See documentation

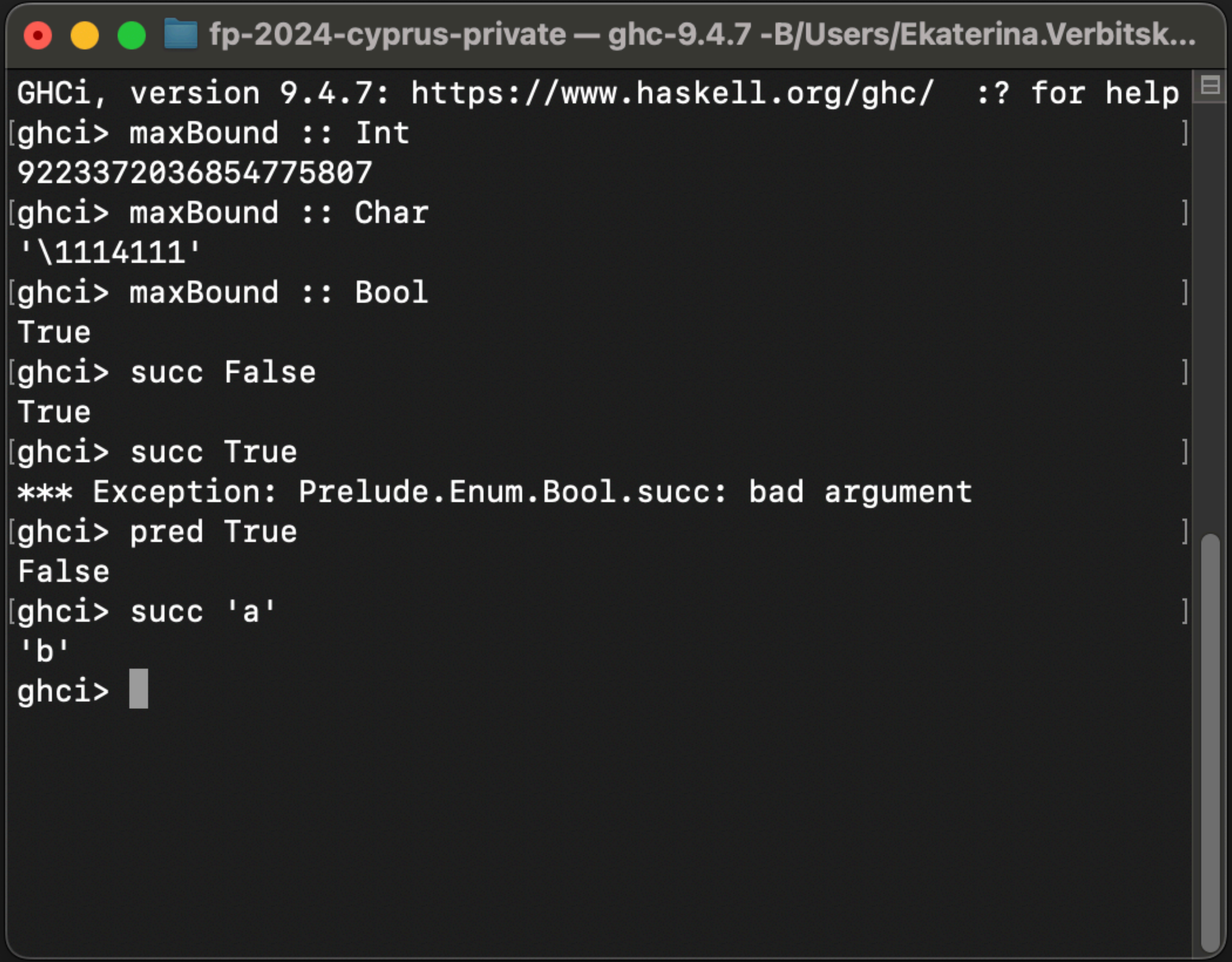
- Let’s make a custom Pair an instance of Ord
- Oops: we need to make it an instance of Eq


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :{
ghci| data Pair a b = Pair a b
ghci|
ghci| instance (Eq a, Eq b) => Eq (Pair a b) where
ghci|   Pair x y == Pair x' y' = x == x' && y == y'
ghci|
ghci| instance (Ord a, Ord b) => Ord (Pair a b) where
ghci|   Pair x y <= Pair x' y' = x <= x' && y <= y'
ghci| :}
ghci> Pair 1 2 <= Pair 2 3
True
ghci> Pair (-1) 2 <= Pair 2 (-1)
False
ghci> 
```

BOUNDED, ENUM

- Bounded
- Enum

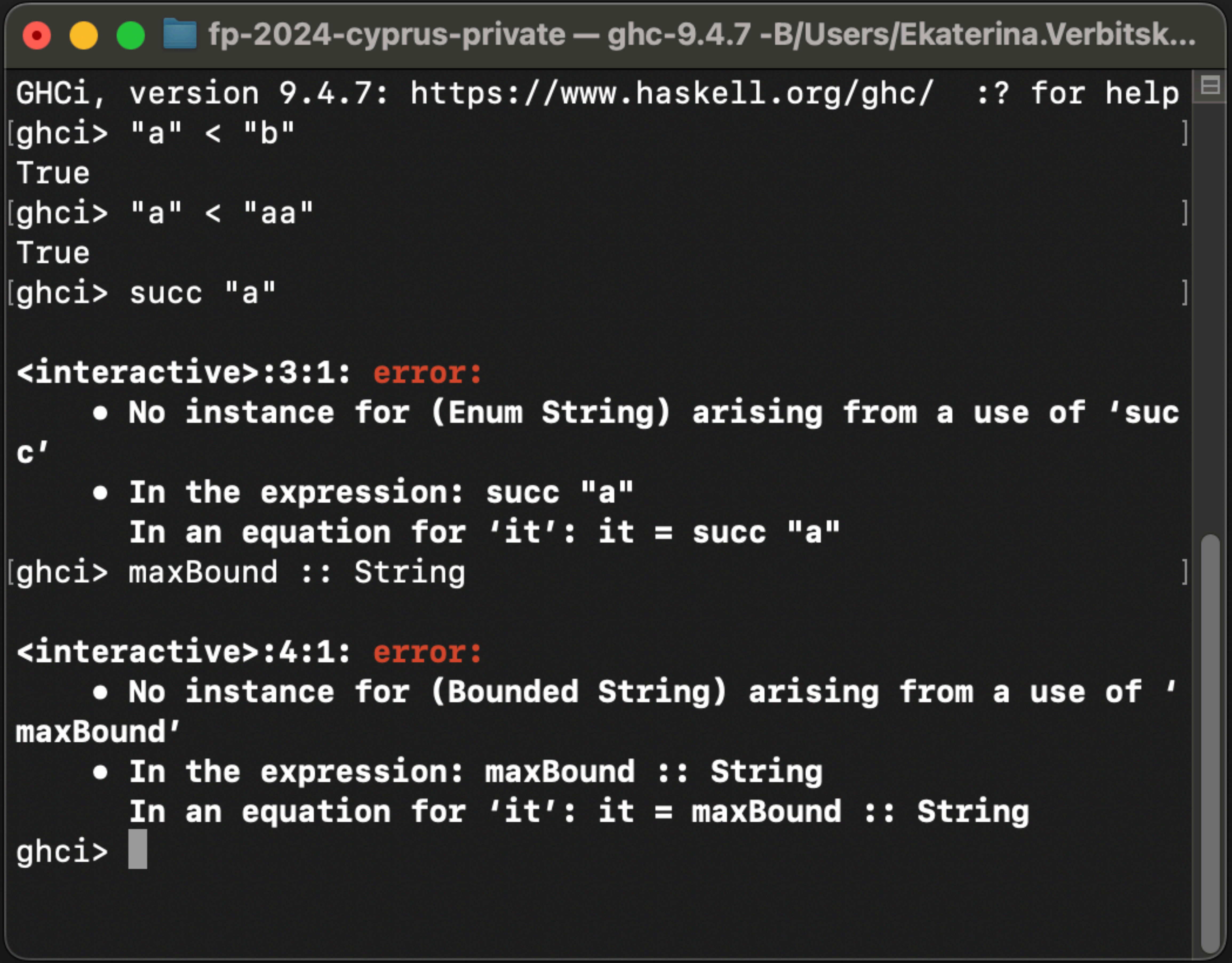


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> maxBound :: Int
9223372036854775807
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> succ False
True
ghci> succ True
*** Exception: Prelude.Enum.Bool.succ: bad argument
ghci> pred True
False
ghci> succ 'a'
'b'
ghci> 
```

SOME TYPES ARE NOT INSTANCES OF SOME TYPE CLASSES

- String (in fact, any List) is not an instance of either Enum or Bounded
- Strings can have any length, so they are not bounded
- What is the 'next' string?
- When it doesn't make sense, don't force the instance


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

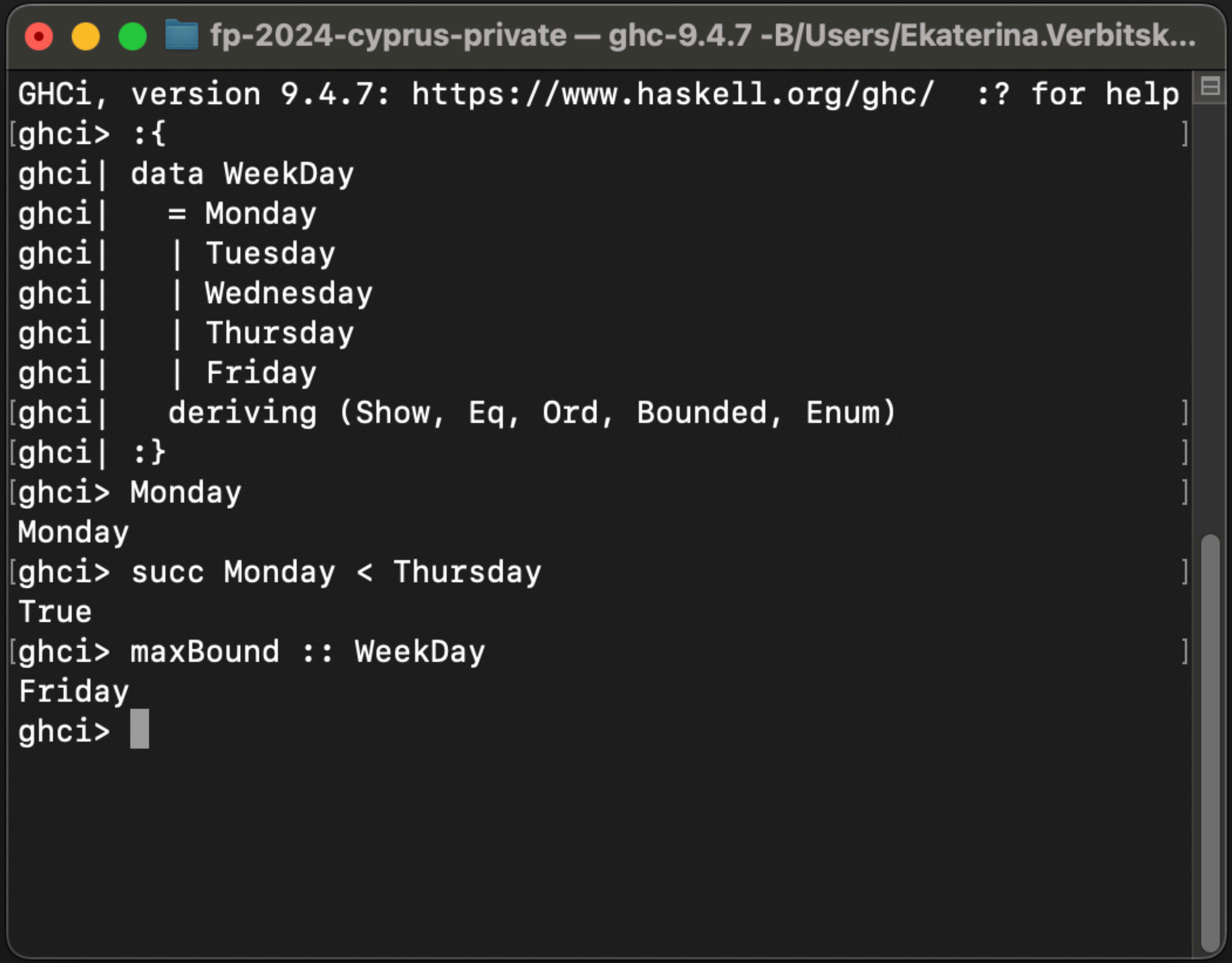
```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> "a" < "b"
True
ghci> "a" < "aa"
True
ghci> succ "a"

<interactive>:3:1: error:
    • No instance for (Enum String) arising from a use of ‘suc
c’
    • In the expression: succ "a"
      In an equation for ‘it’: it = succ "a"
ghci> maxBound :: String

<interactive>:4:1: error:
    • No instance for (Bounded String) arising from a use of ‘
maxBound’
    • In the expression: maxBound :: String
      In an equation for ‘it’: it = maxBound :: String
ghci> 
```

EXERCISES

- Make custom error data types for functions from the previous exercise, make them an instance of Show

DERIVING

- Some instances are boilerplate, for which an automatic deriving is possible
- Show, Eq, Ord, Bounded, Enum can be derived



fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

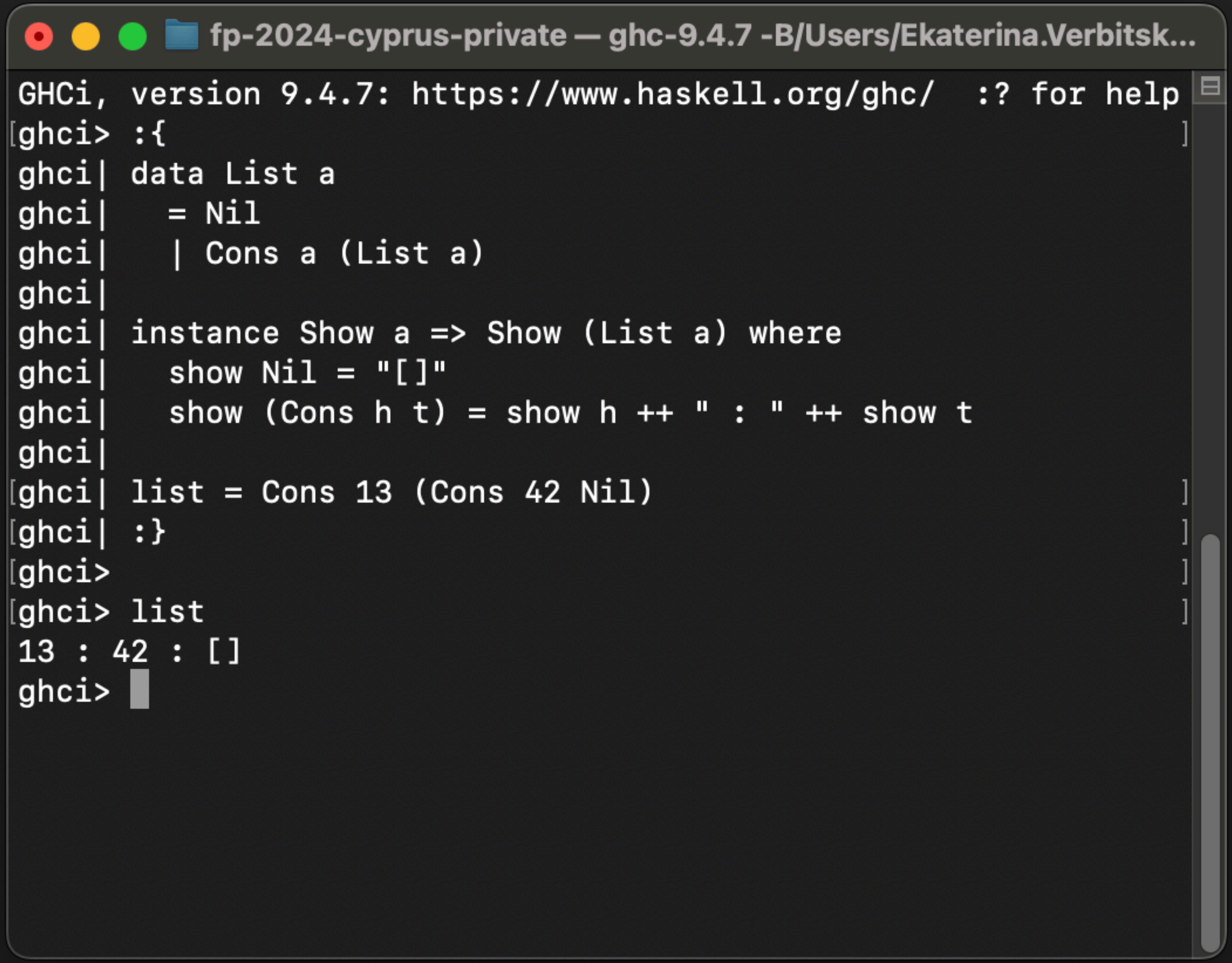
```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :{
ghci| data WeekDay
ghci|   = Monday
ghci|   | Tuesday
ghci|   | Wednesday
ghci|   | Thursday
ghci|   | Friday
ghci|   deriving (Show, Eq, Ord, Bounded, Enum)
ghci| :}
ghci> Monday
Monday
ghci> succ Monday < Thursday
True
ghci> maxBound :: WeekDay
Friday
ghci> 
```

OUTLINE

- Algebraic Data Types
- Type Classes
- Recursive Data Types

LIST

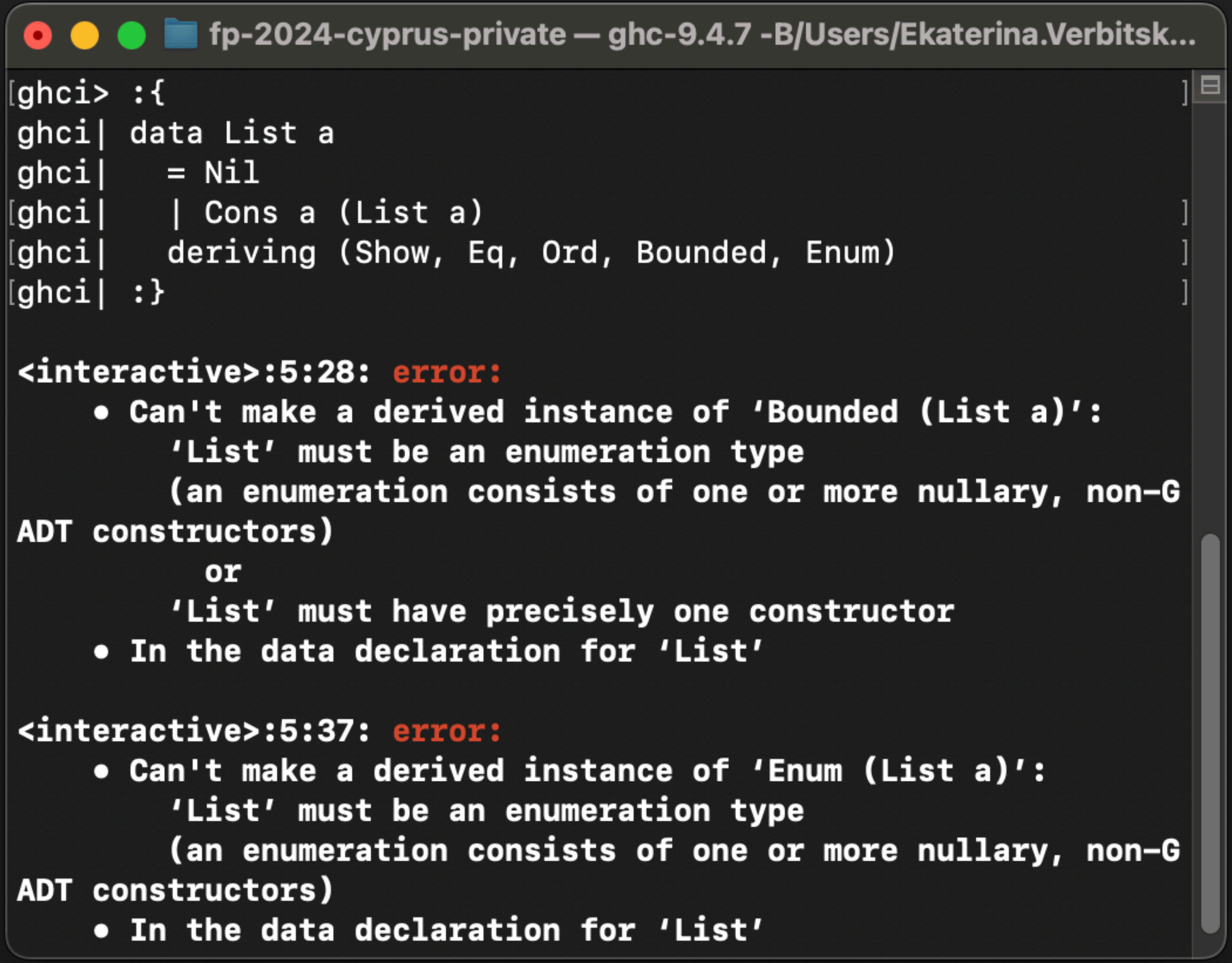
- We can use type constructor in the definition of the type
- Nothing is that different, compared to non-recursive data structures


fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :{
ghci| data List a
ghci|   = Nil
ghci|   | Cons a (List a)
ghci|
ghci| instance Show a => Show (List a) where
ghci|   show Nil = "[]"
ghci|   show (Cons h t) = show h ++ " : " ++ show t
ghci|
ghci| list = Cons 13 (Cons 42 Nil)
ghci| :}
ghci>
ghci> list
13 : 42 : []
ghci> 
```

LIST

- We can use type constructor in the definition of the type
- Nothing is that different, compared to non-recursive data structures
- Well, you cannot derive some instances



fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

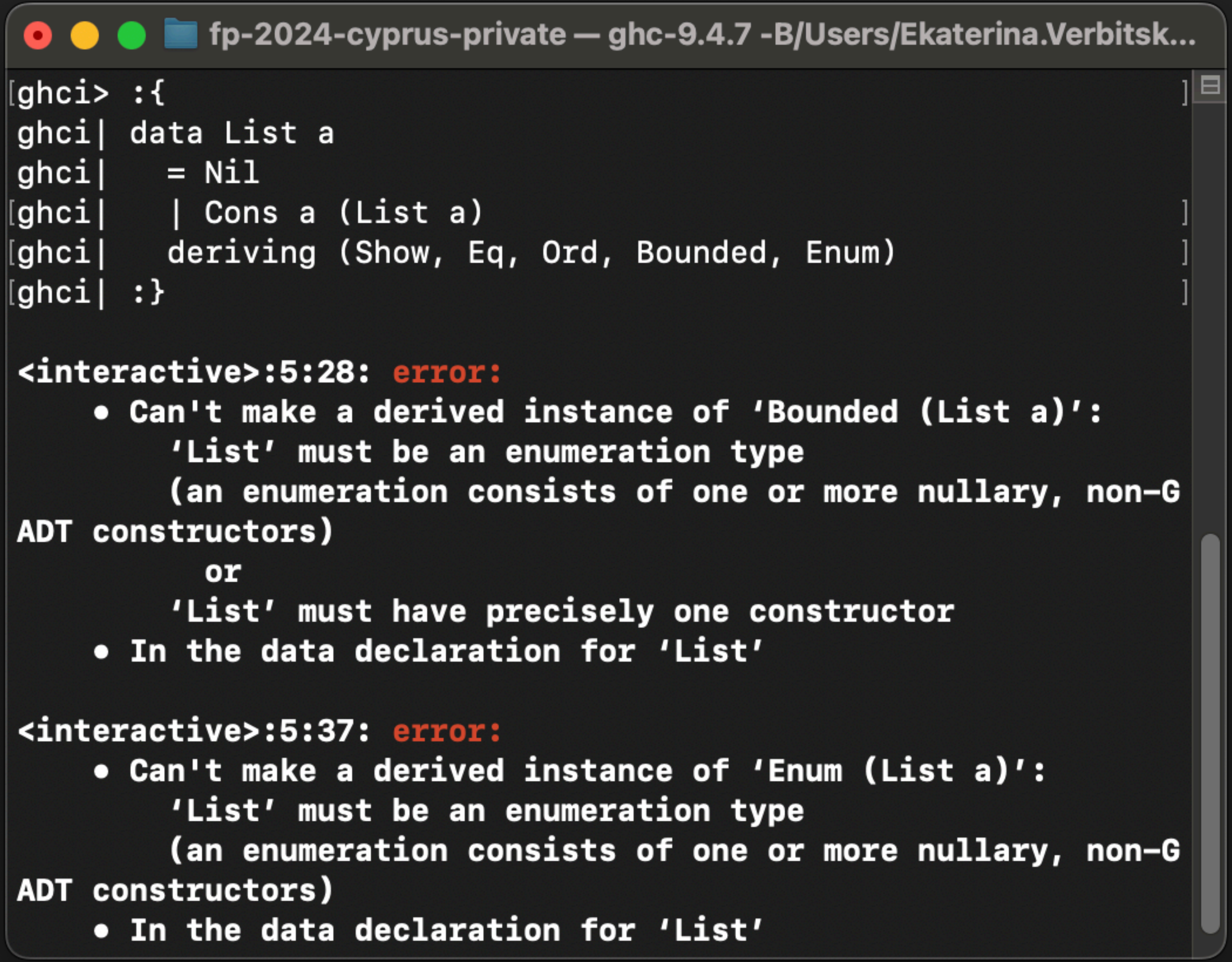
```
ghci> :{
ghci| data List a
ghci|   = Nil
ghci|   | Cons a (List a)
ghci|   deriving (Show, Eq, Ord, Bounded, Enum)
ghci| :}

<interactive>:5:28: error:
    • Can't make a derived instance of ‘Bounded (List a)’:
        ‘List’ must be an enumeration type
        (an enumeration consists of one or more nullary, non-G
ADT constructors)
          or
        ‘List’ must have precisely one constructor
    • In the data declaration for ‘List’

<interactive>:5:37: error:
    • Can't make a derived instance of ‘Enum (List a)’:
        ‘List’ must be an enumeration type
        (an enumeration consists of one or more nullary, non-G
ADT constructors)
    • In the data declaration for ‘List’
```

EXERCISES

- Implement 5 of any list functions from Prelude for our List
- Create any 2 instances of any type classes for our List

LIST

- We can use type constructor in the definition of the type
- Nothing is that different, compared to non-recursive data structures
- Well, you cannot derive some instances



fp-2024-cyprus-private — ghc-9.4.7 -B/Users/Ekaterina.Verbitsk...

```
ghci> :{
ghci| data List a
ghci|   = Nil
ghci|   | Cons a (List a)
ghci|   deriving (Show, Eq, Ord, Bounded, Enum)
ghci| :}

<interactive>:5:28: error:
    • Can't make a derived instance of ‘Bounded (List a)’:
        ‘List’ must be an enumeration type
        (an enumeration consists of one or more nullary, non-G
ADT constructors)
          or
        ‘List’ must have precisely one constructor
    • In the data declaration for ‘List’

<interactive>:5:37: error:
    • Can't make a derived instance of ‘Enum (List a)’:
        ‘List’ must be an enumeration type
        (an enumeration consists of one or more nullary, non-G
ADT constructors)
    • In the data declaration for ‘List’
```