INTRODUCTION TO

FUNCTIONAL PROGRAMMING

OUTLINE

- Algebraic Data Types
- Type Classes
- Recursive Data Types

WHAT IS A TYPE?

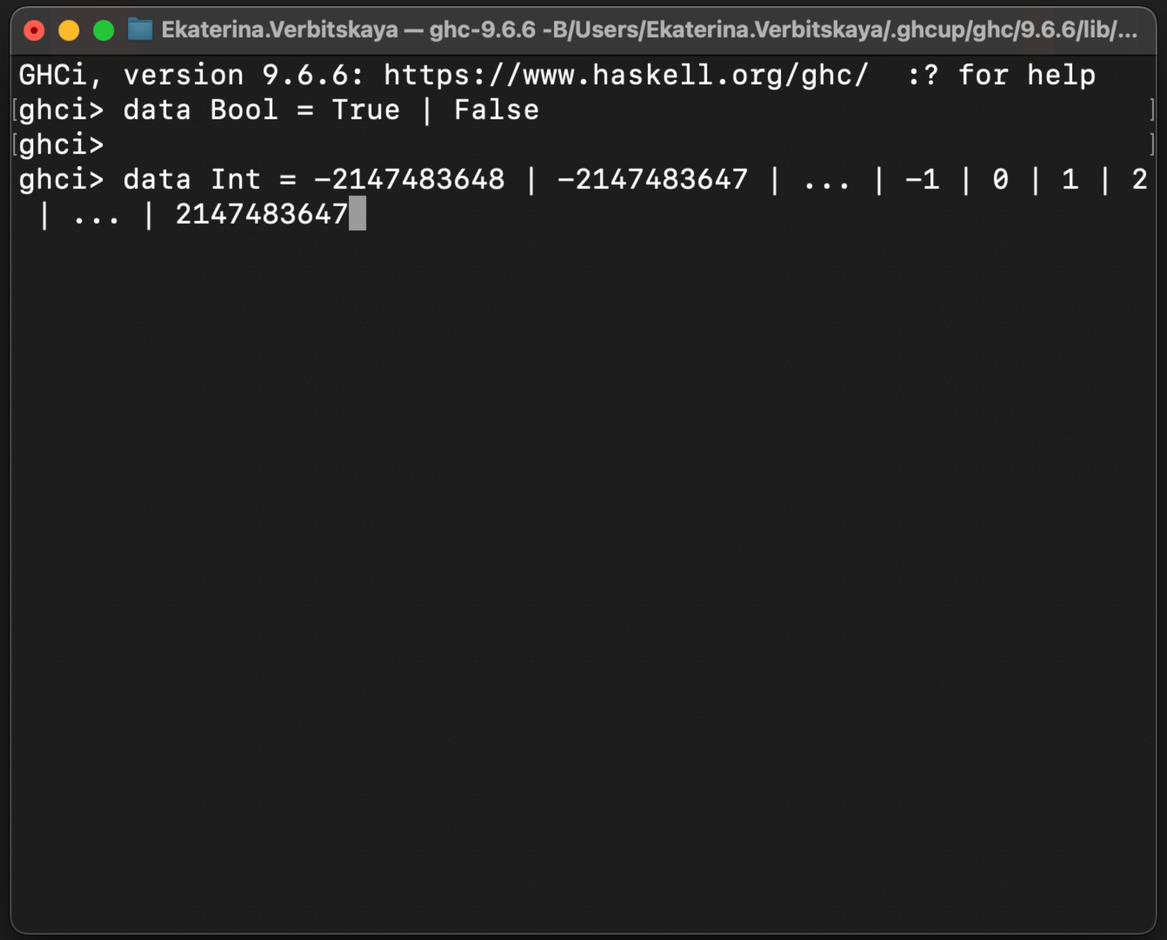
- A collection of its values

- 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

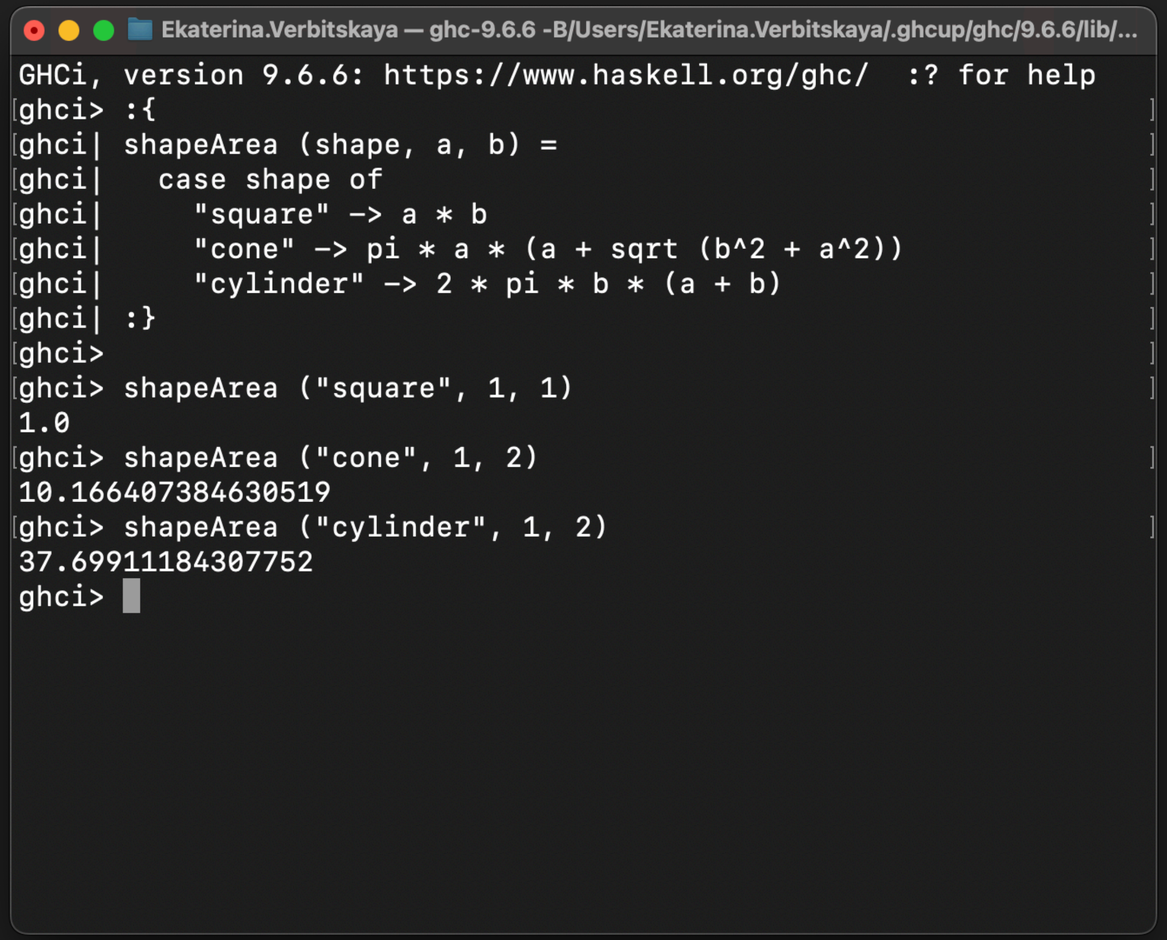
- Bool, Int – type name
- True, False, 2147483647 – values constructors
- | – “or”

- (Int is not defined like this in reality)



LET'S REVISIT SHAPEAREA

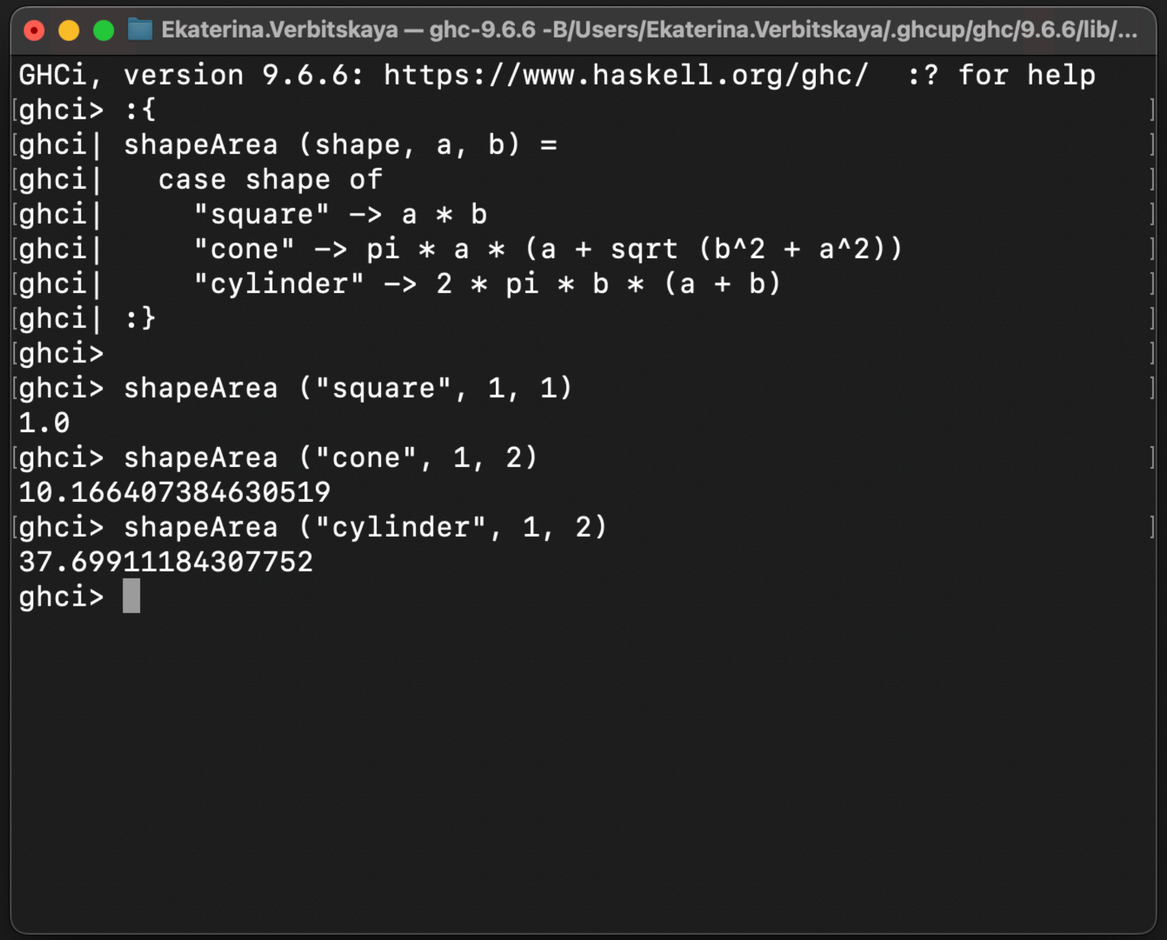
- How many issues can you spot in this code?



LET'S REVISIT SHAPEAREA

- How many issues can you spot in this code?

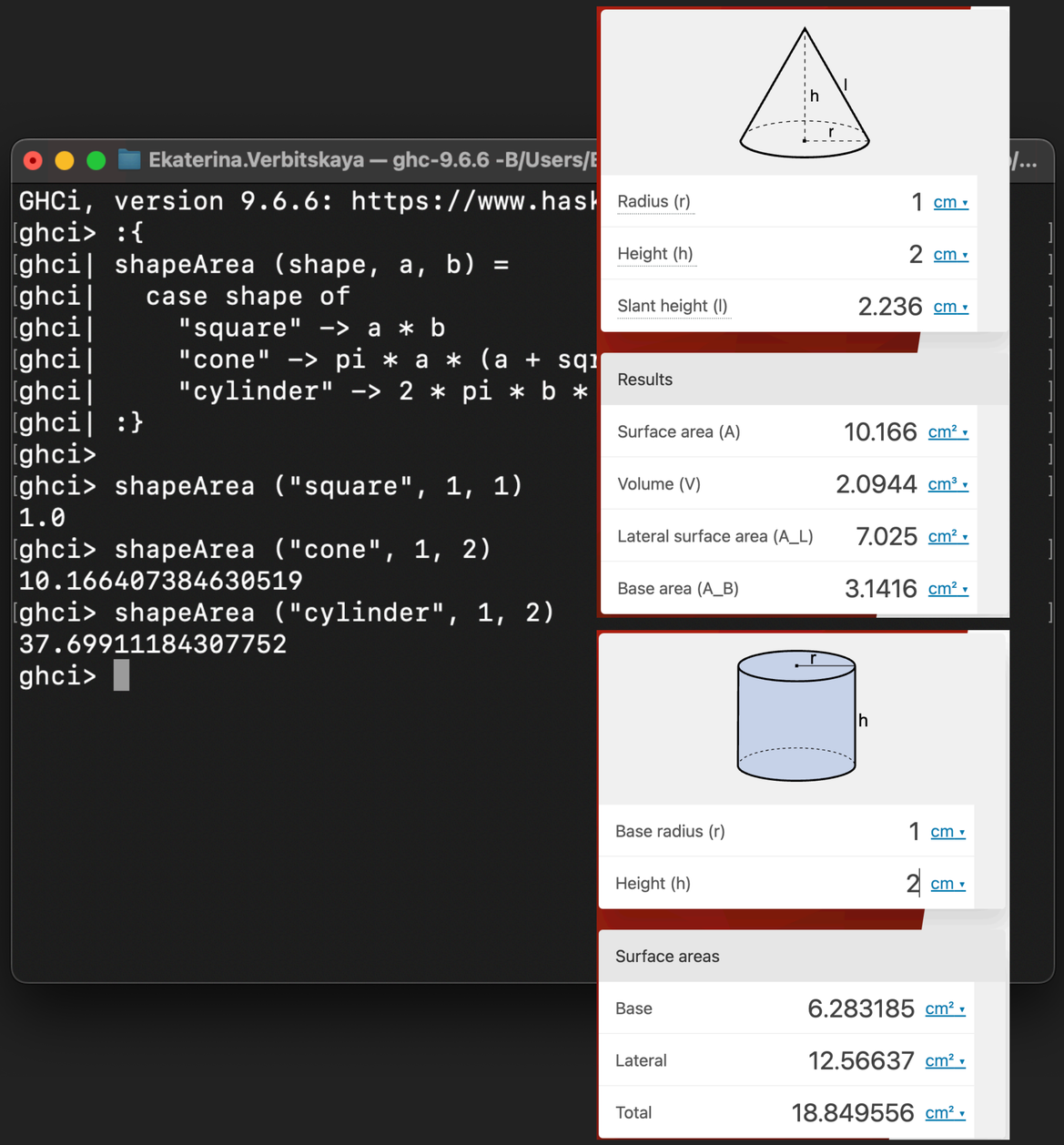
- Why has a square two sides?
- What are a and b?
- What about case sensitivity?
- What if we get a "rectangle" or other string?



LET'S REVISIT SHAPEAREA

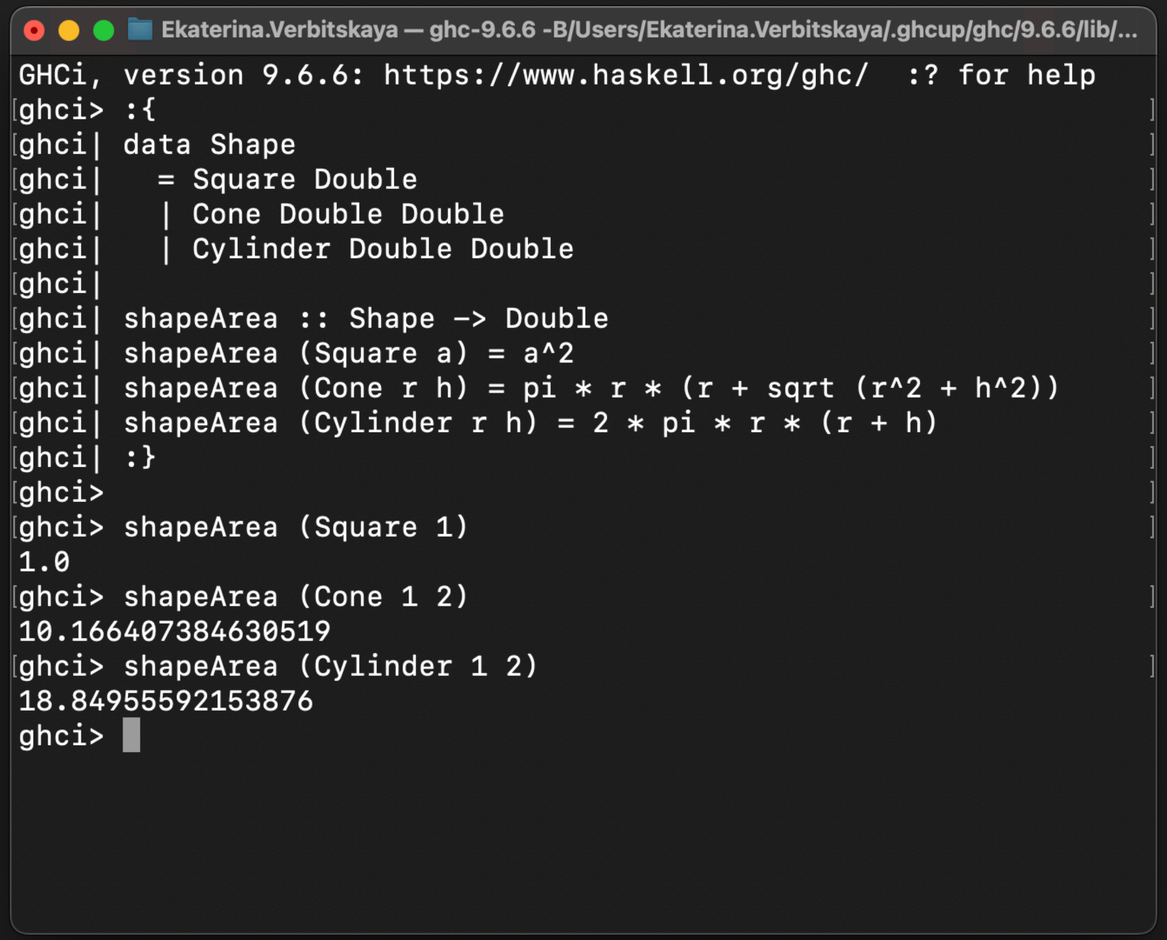
- How many issues can you spot in this code?

- Why has a square 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



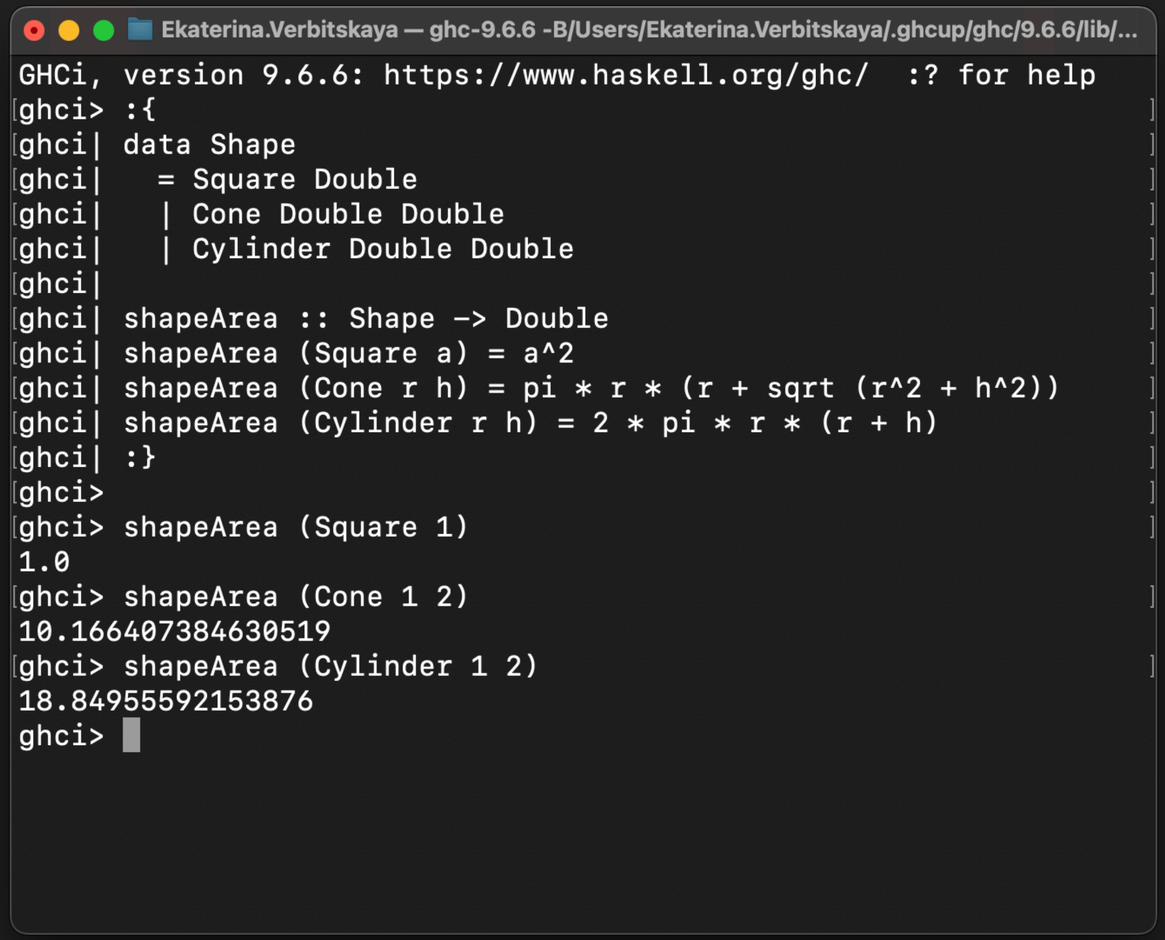
BETTER WAY TO DESIGN SHAPE

- 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



PROPERTIES OF ADTS

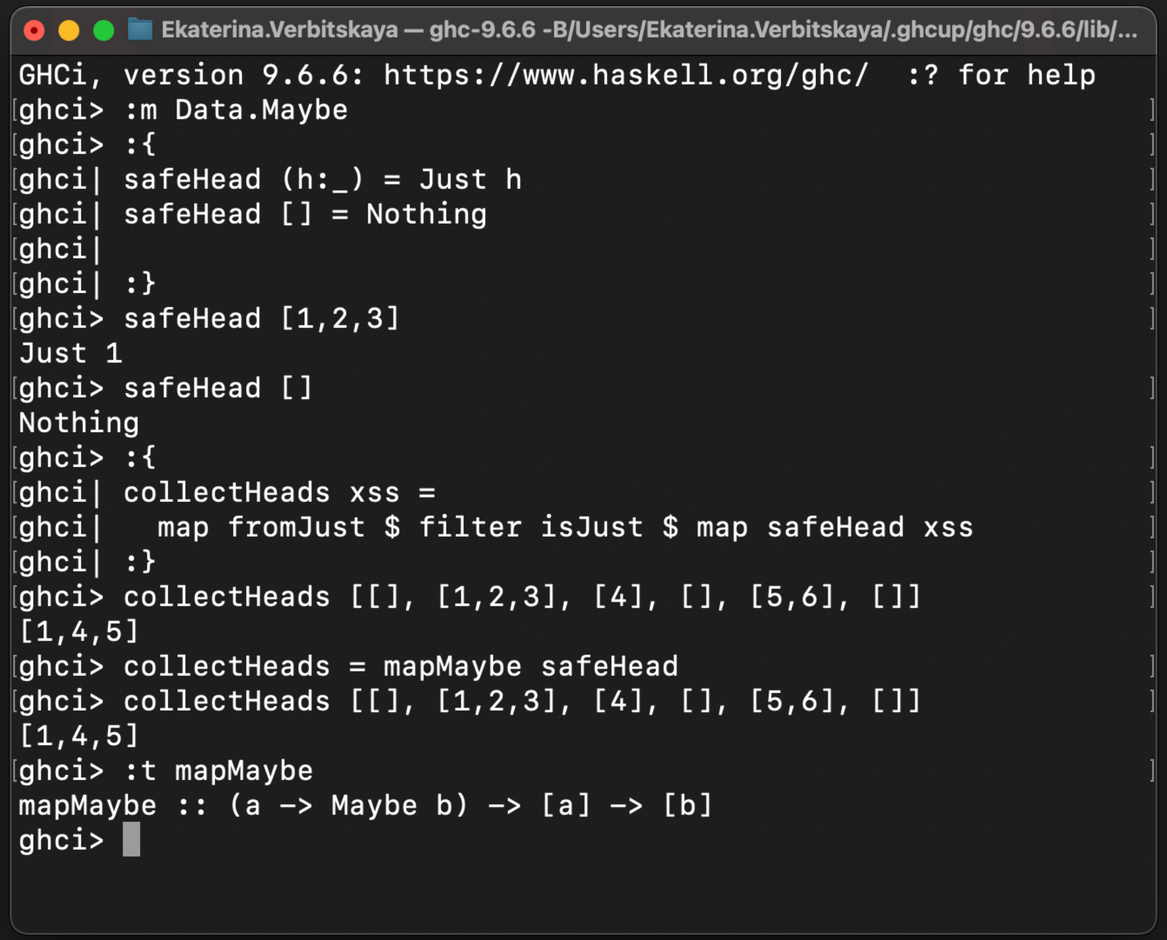
- Distinctness
  - $\forall j \neq i \,.\, C_i^n(x) \neq C_j^n(y)$
- Injectivity
  - $C_i^n(x_1, \ldots, x_n) = C_j^n(y_1, \ldots, y_n) \Rightarrow \forall k \,.\, x_k = y_k$
- Exhaustiveness
  - x of ADT $\Rightarrow \exists i \,.\, x = C_i^n(y_1, \ldots, y_n)$
- Selection
  - $\exists s_i^k \,.\, s_i^n(C_k^n(x_1,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



FAILING COMPUTATIONS: MAYBE

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

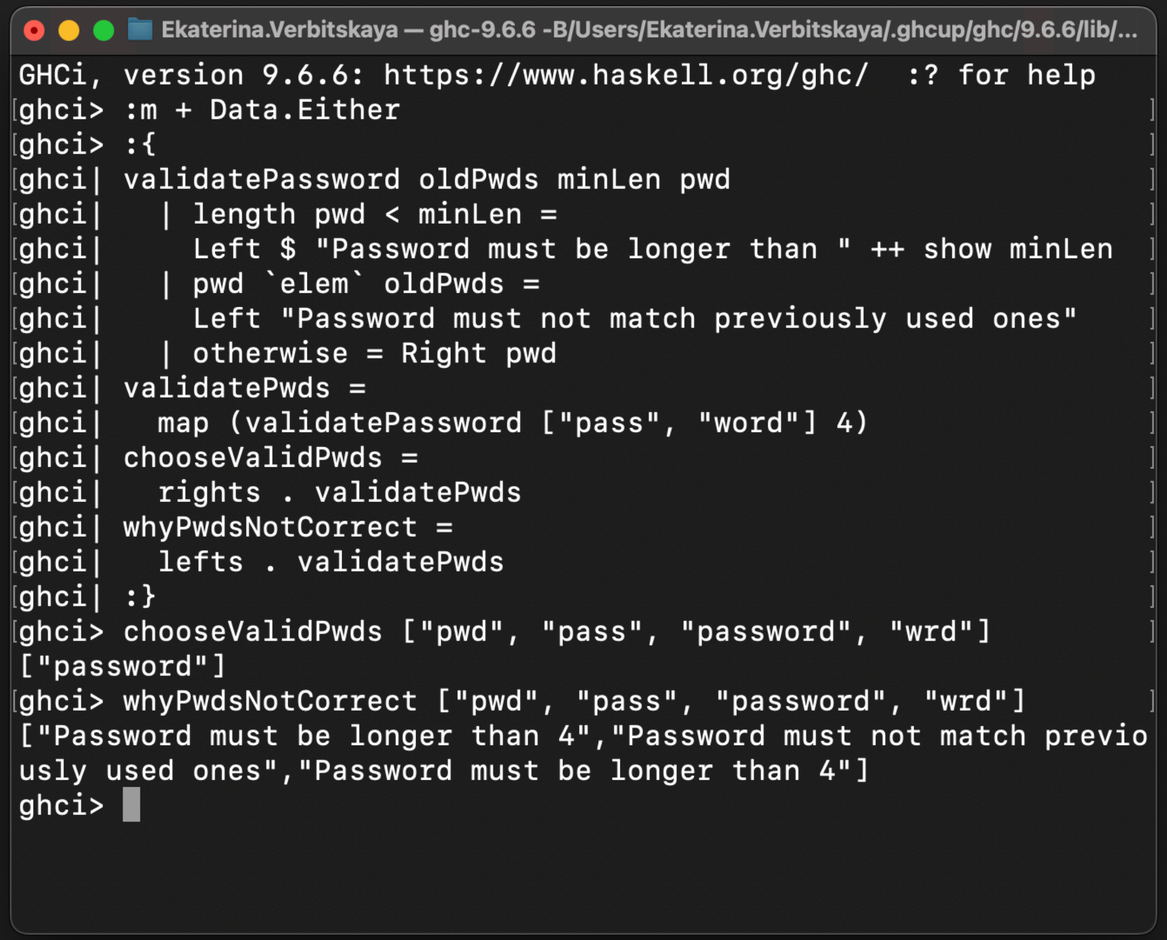
- A way to fix partiality of a function
- Only use it when there is a single way for a function to fail
- isJust, isNothing, fromJust, fromMaybe, listToMaybe, catMaybes, mapMaybes, maybe



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



EXERCISES

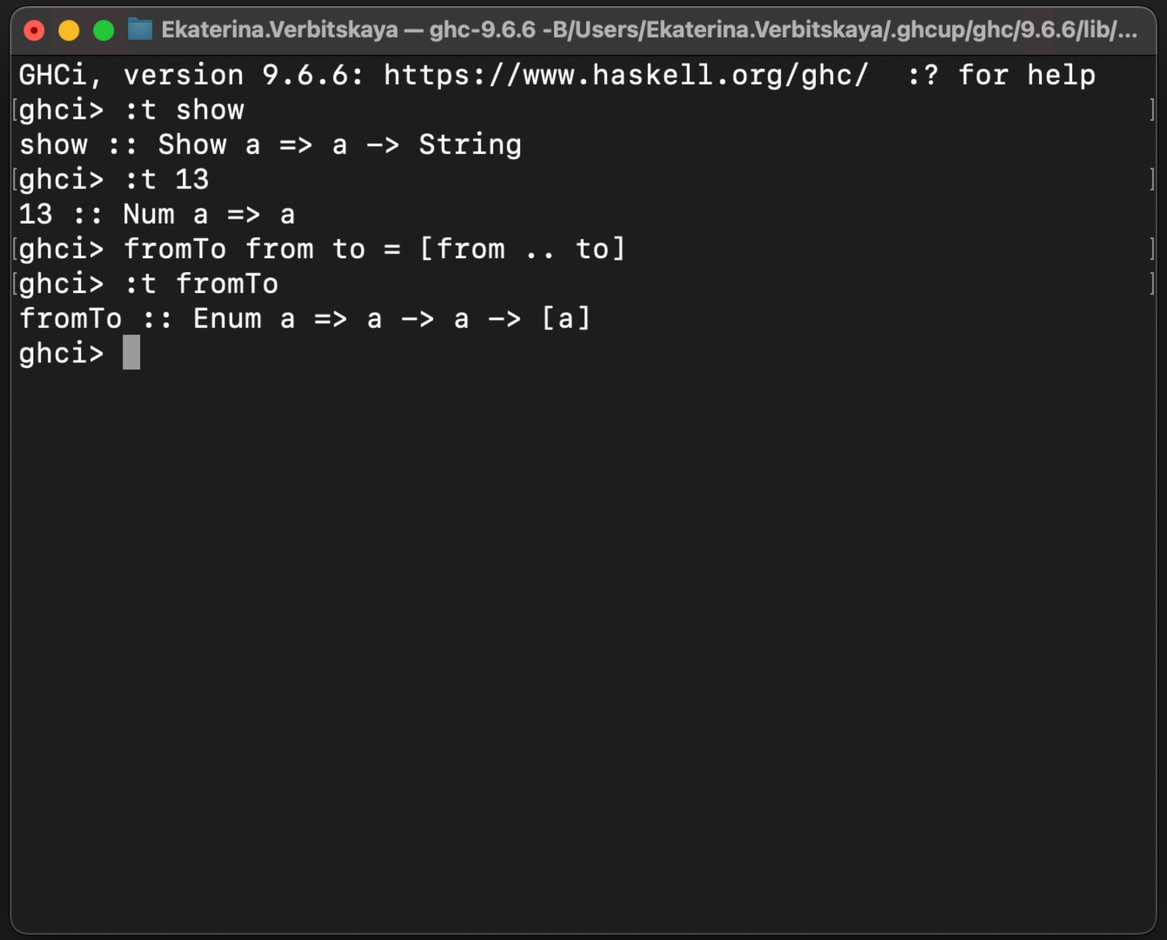
- Fix partially applied functions in HW01: pick the best suiting way to represent failing computations

OUTLINE

- Algebraic Data Types
- Type Classes
- Recursive Data Types

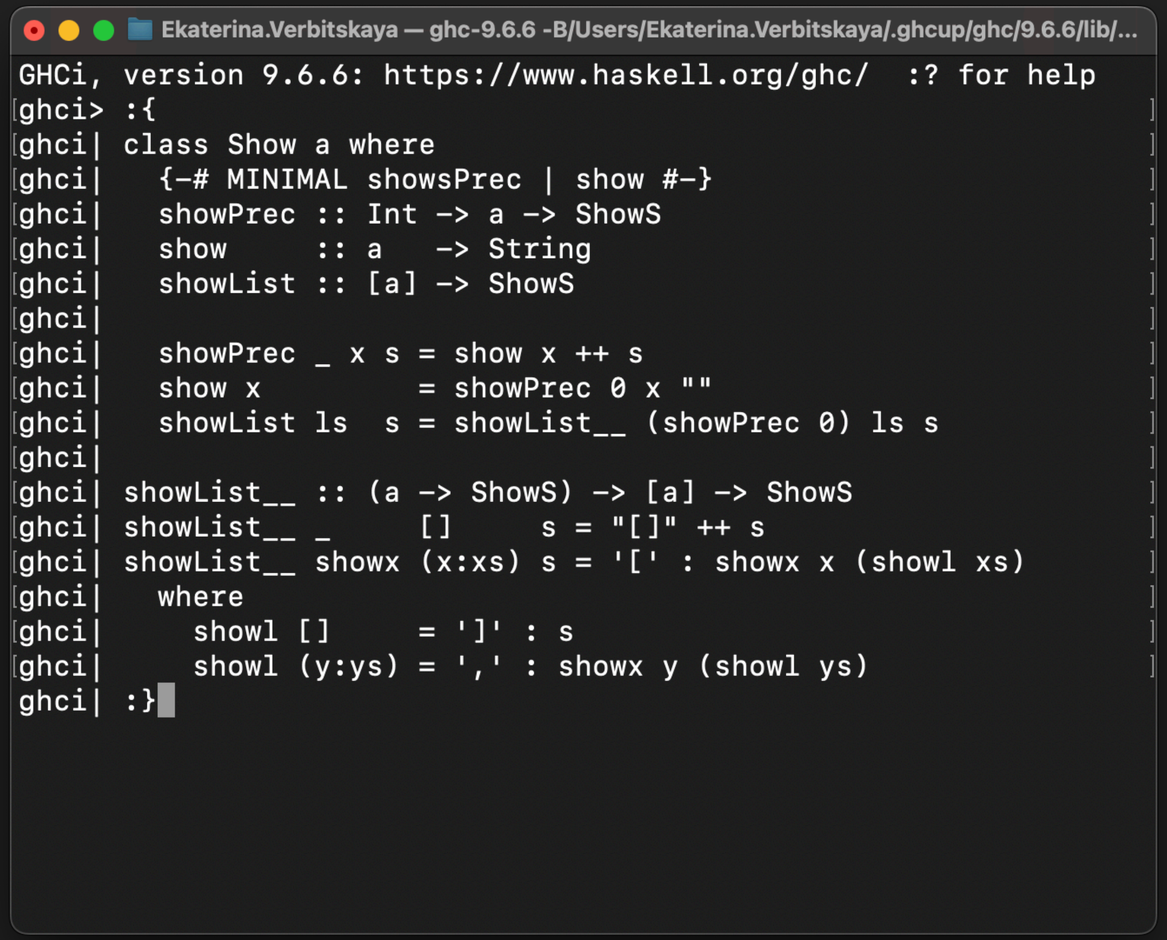
WE'VE SEEN THEM BEFORE

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



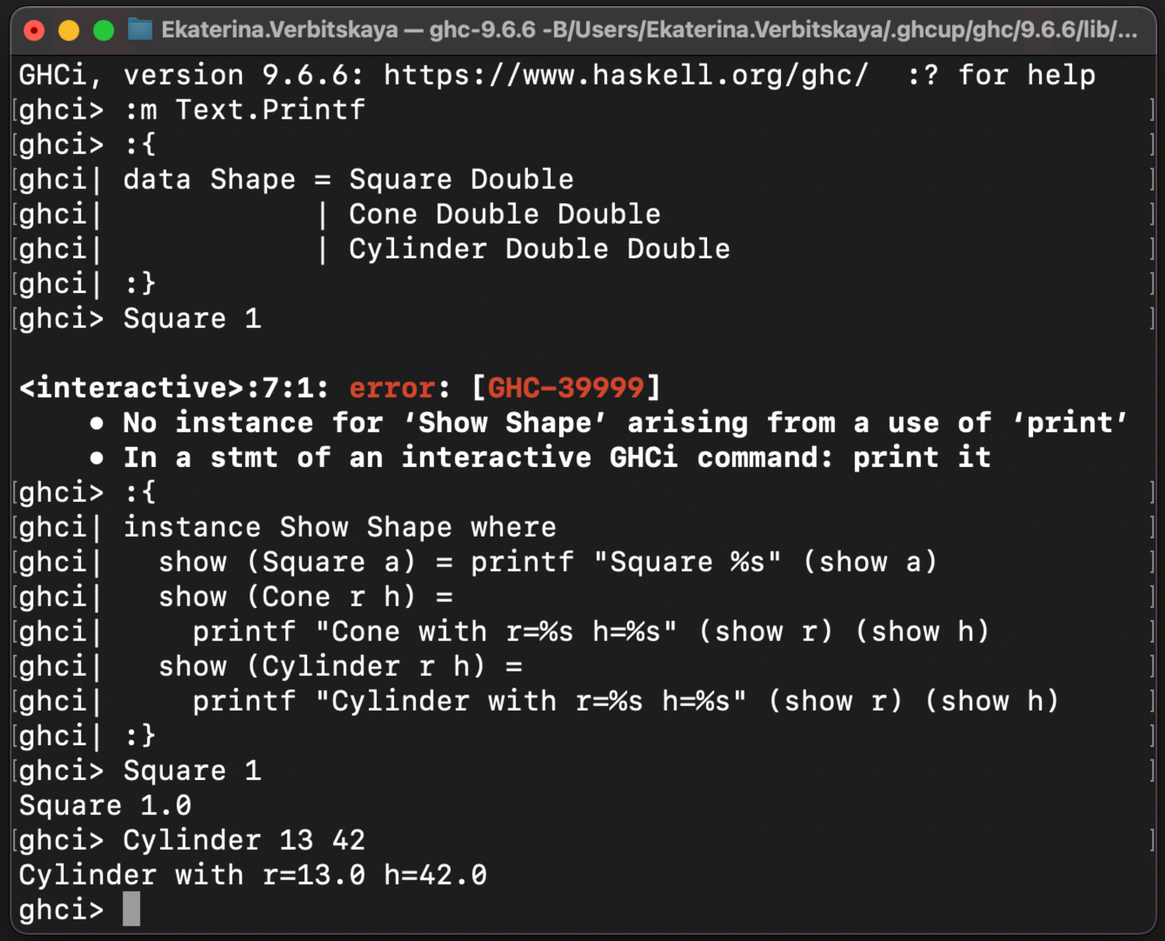
CLASS DEFINITION SYNTAX

- {-# 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



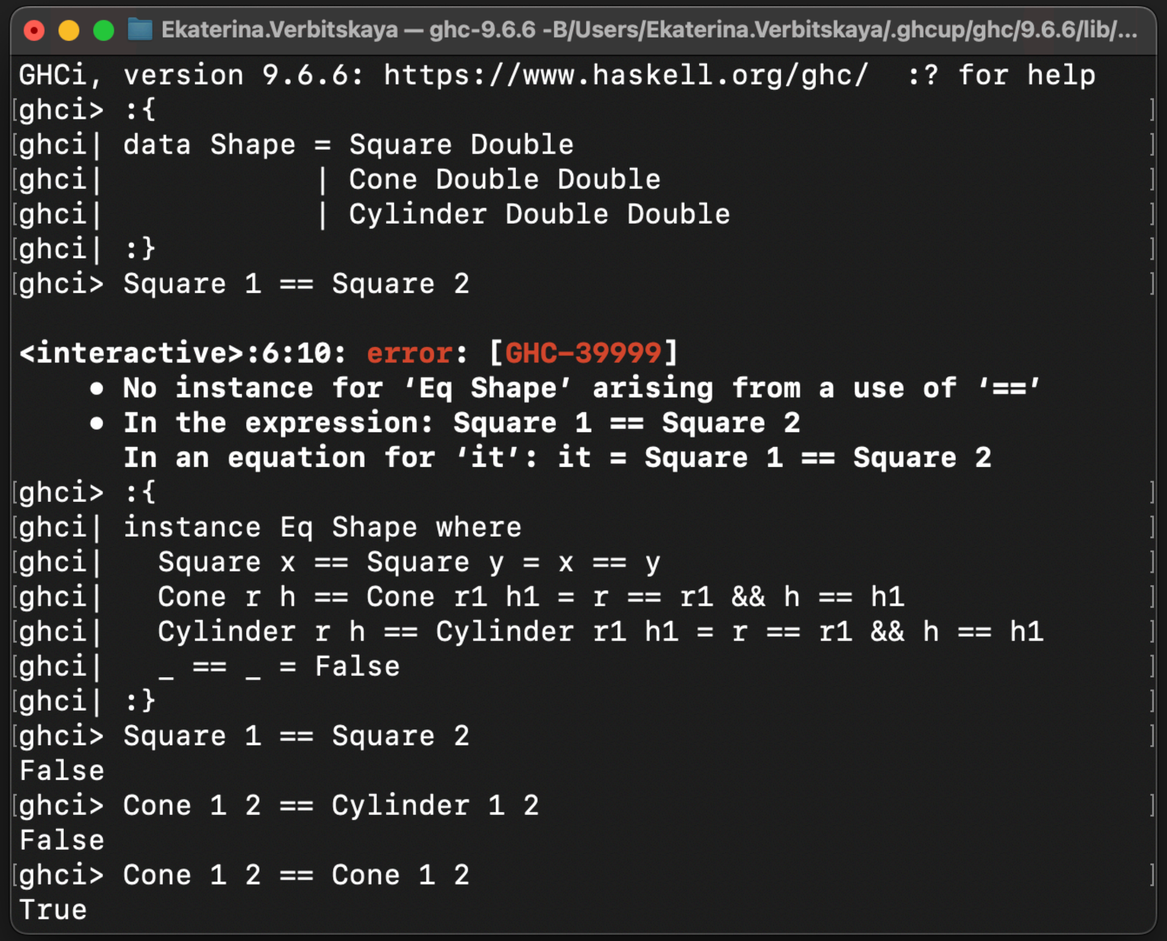
INSTANCES

- 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



EQ TYPE CLASS

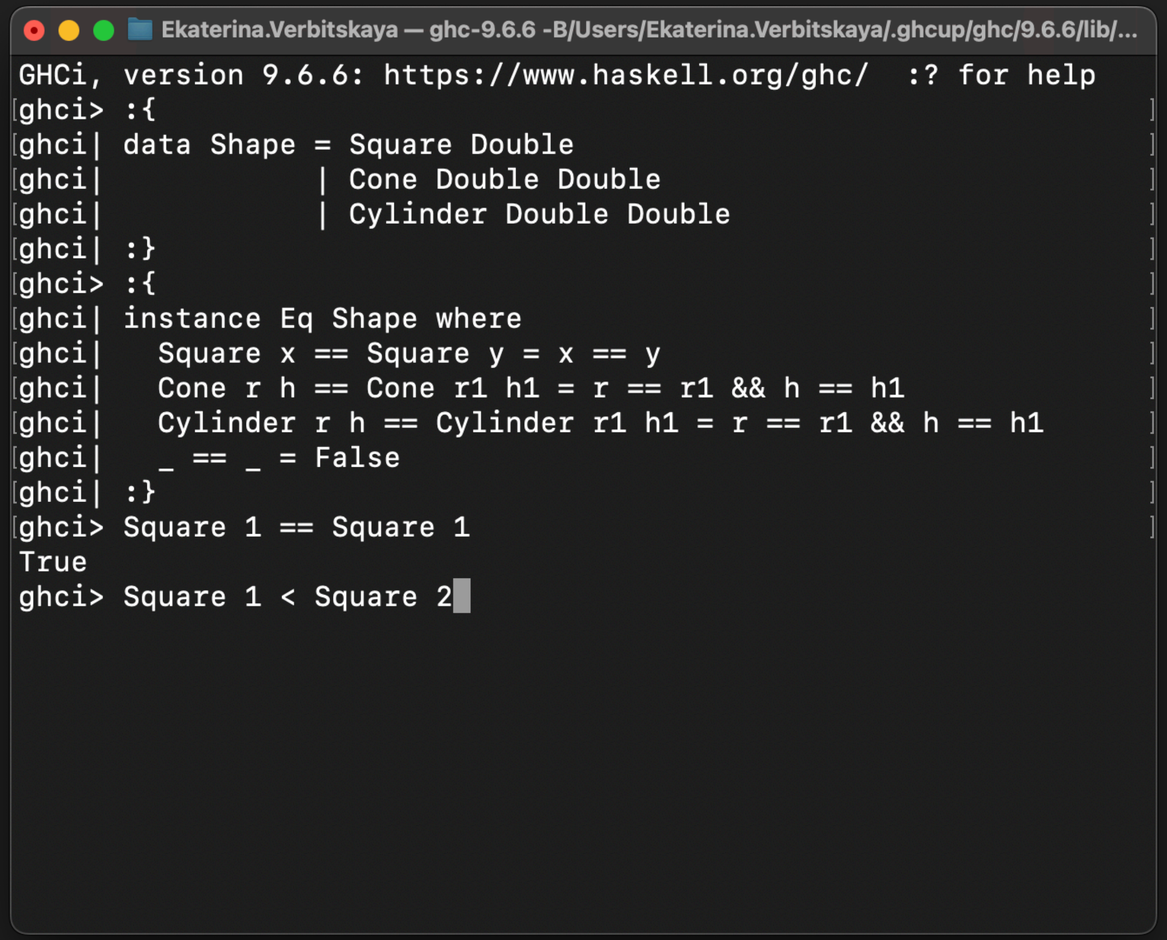
- Not everything can be checked for equality, only instances of Eq
- See documentation



EQ TYPE CLASS

- Not everything can be checked for equality, only instances of Eq
- See documentation

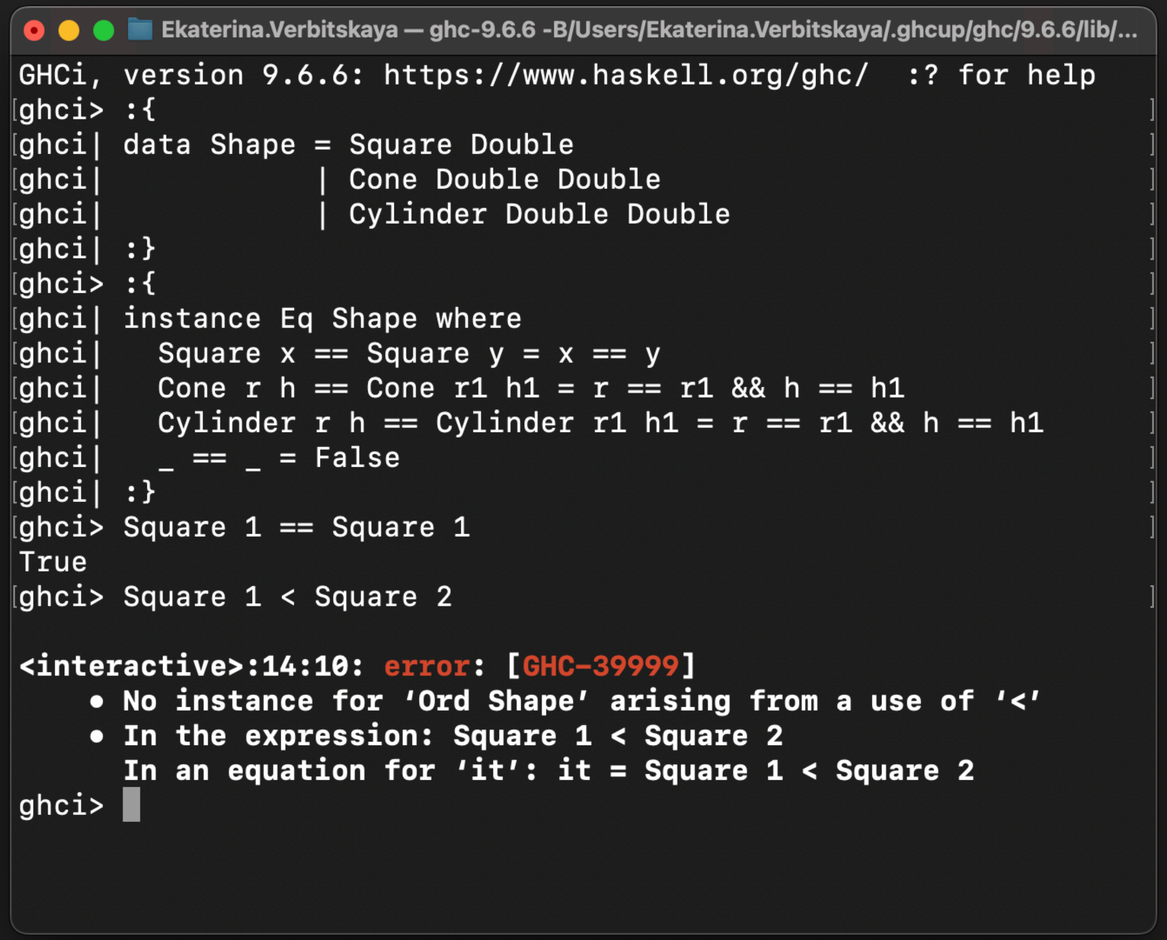
- What will happen when I enter the expression Square 1 < Square 2?



EQ TYPE CLASS

- Not everything can be checked for equality, only instances of Eq
- See documentation

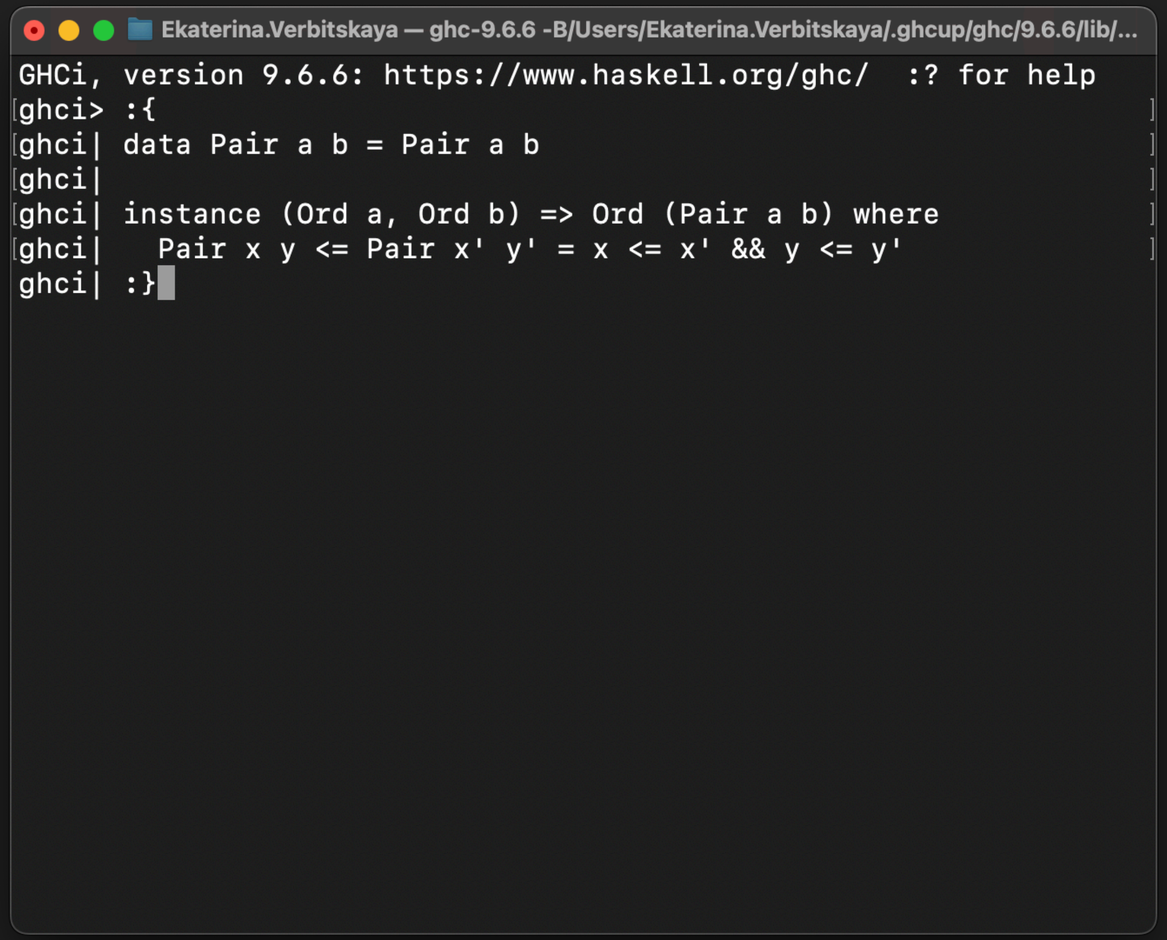
- What will happen when I enter the expression Square 1 < Square 2?
- Error, because Eq has nothing to do with comparisons



ORD TYPE CLASS

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

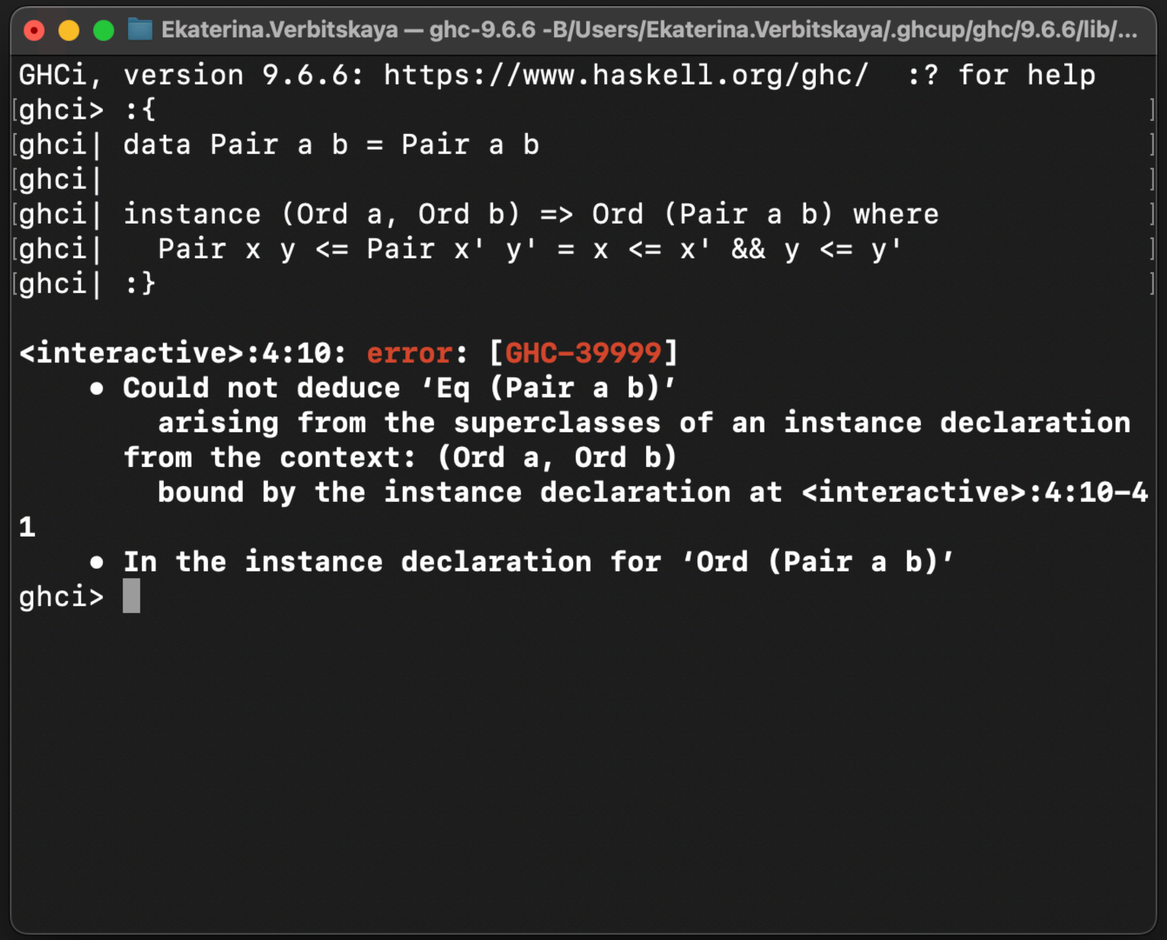
- Let's make a custom Pair an instance of Ord



ORD TYPE CLASS

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

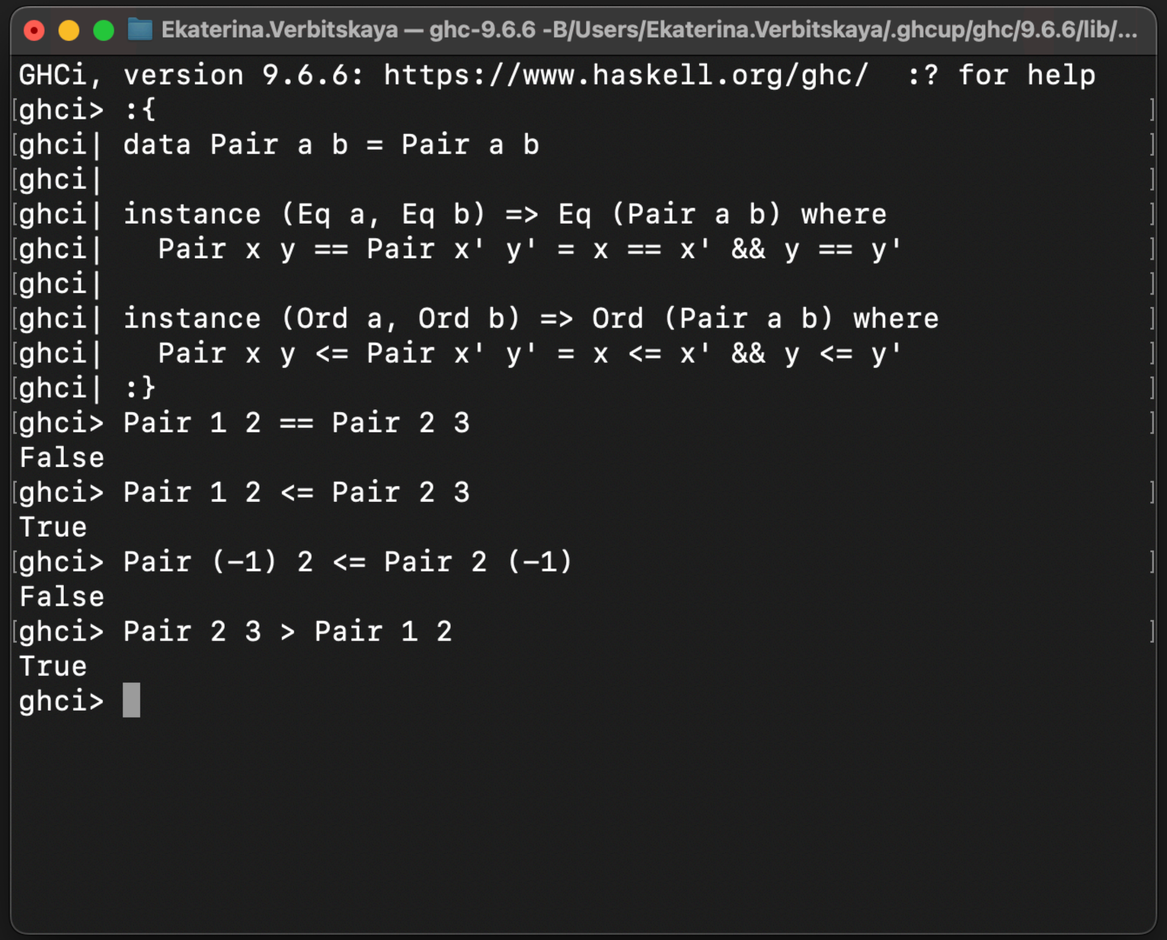
- Let's make a custom Pair an instance of Ord
- Oops: we need to make it an instance of Eq



ORD TYPE CLASS

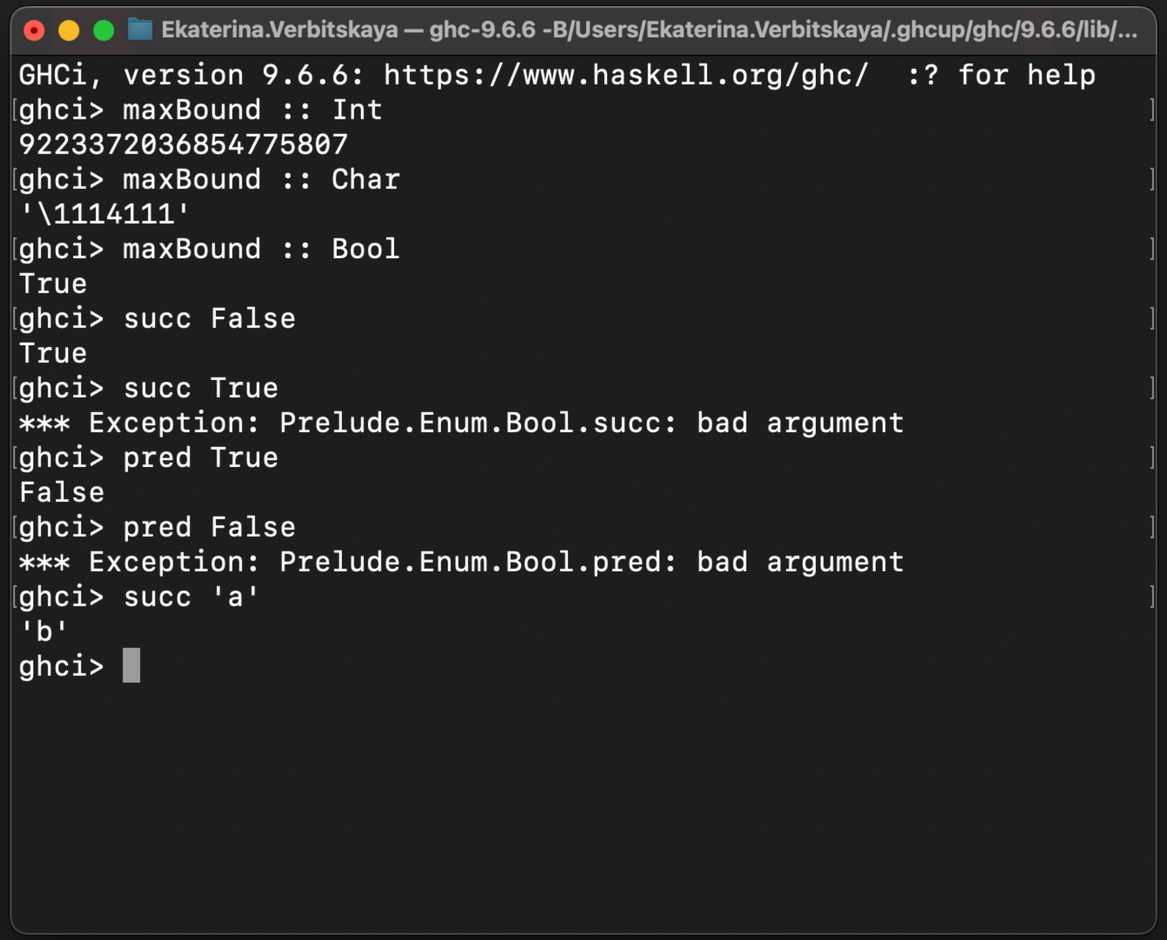
- Ord: when you want to compare stuff
- See documentation

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



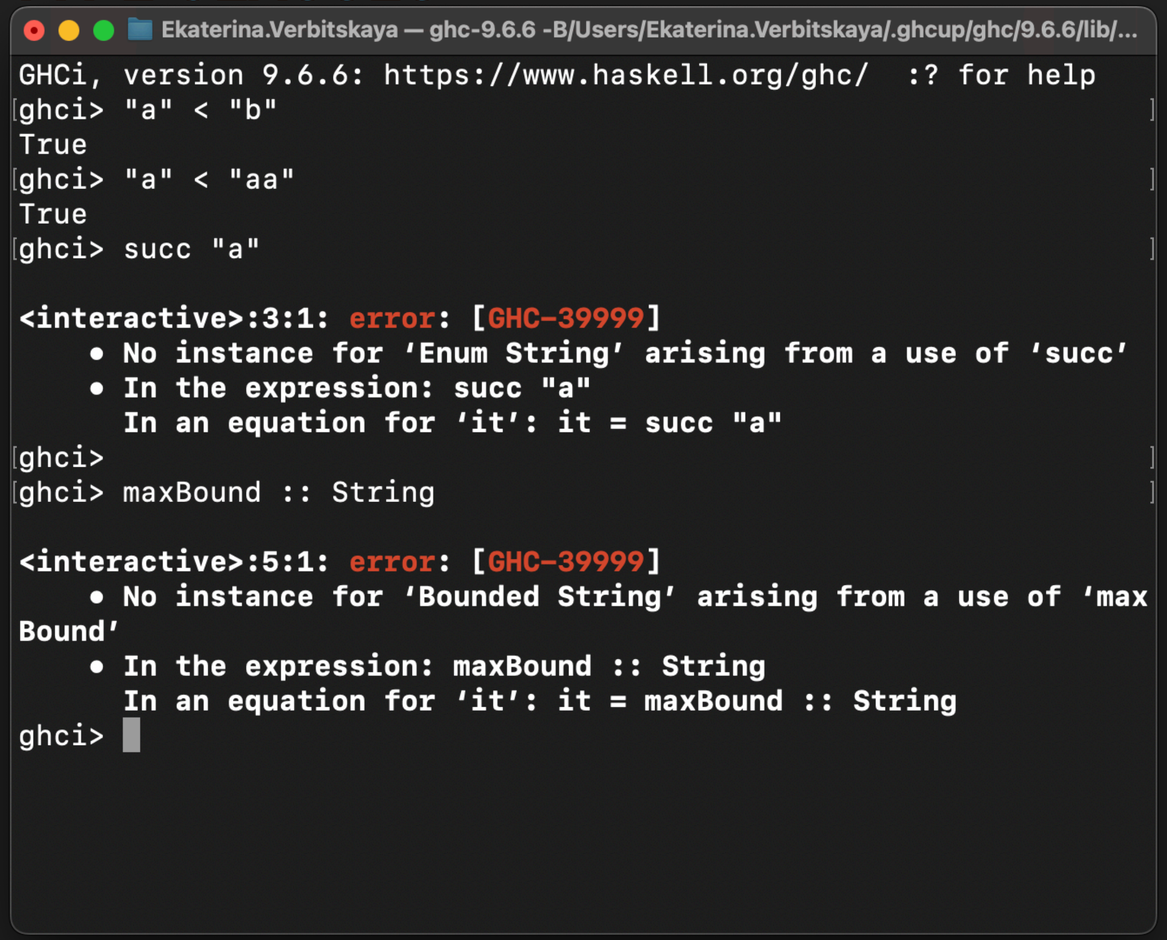
BOUNDED, ENUM

- Bounded
- Enum



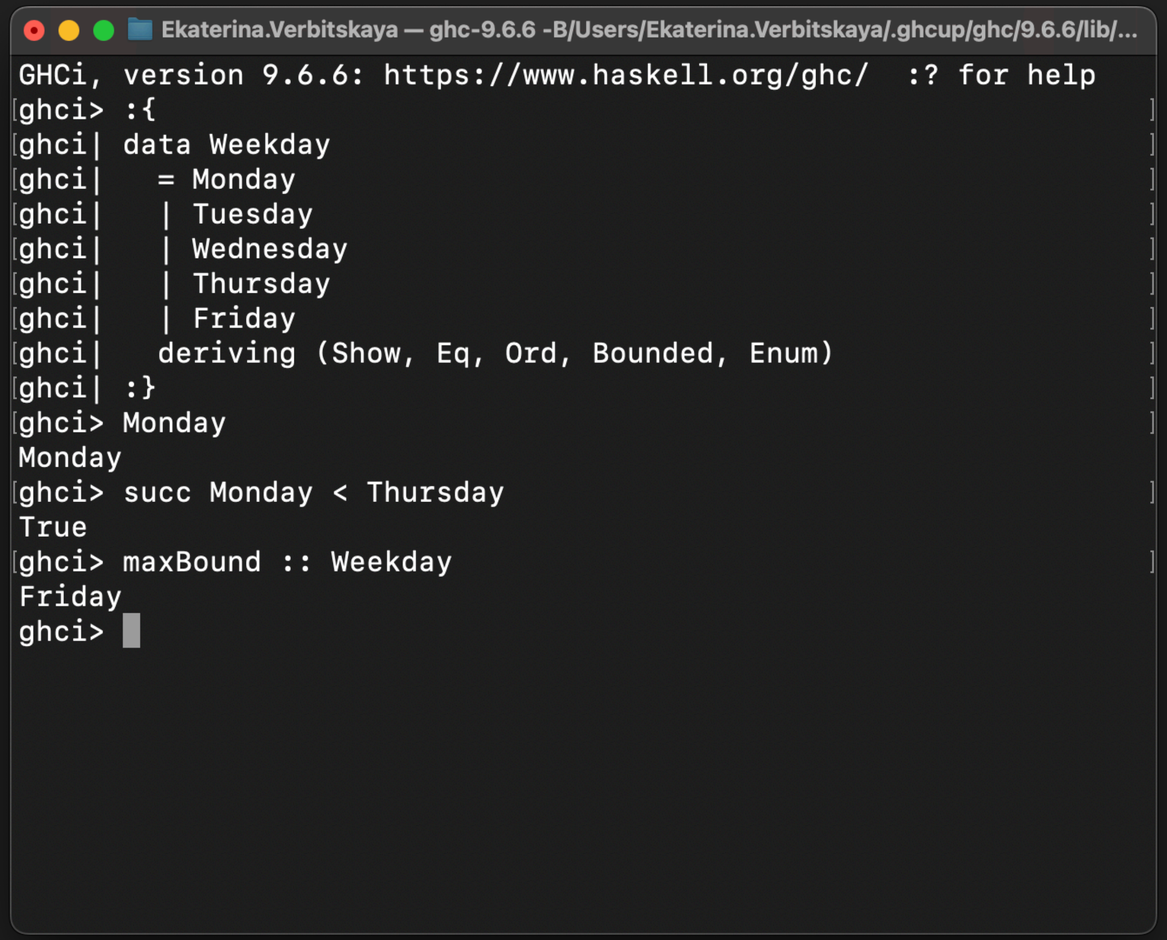
SOME TYPES ARE NOT INSTANCES OF SOME TYPE CLASSES

- 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



DERIVING

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



EXERCISES

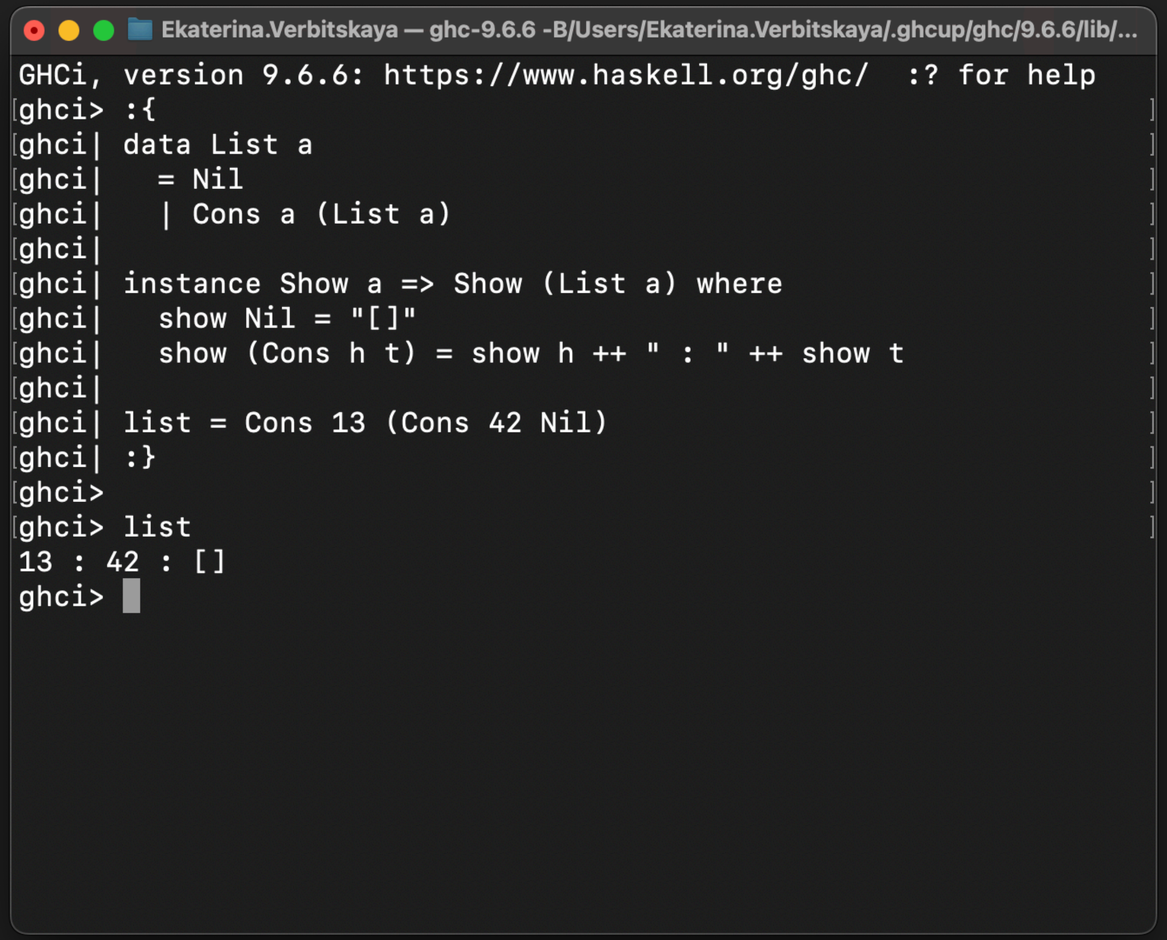
- Make custom error data types for functions from the previous exercise, make them an instance of Show
- Make expression data type an instance of Eq and Ord

OUTLINE

- Algebraic Data Types
- Type Classes
- Recursive Data Types

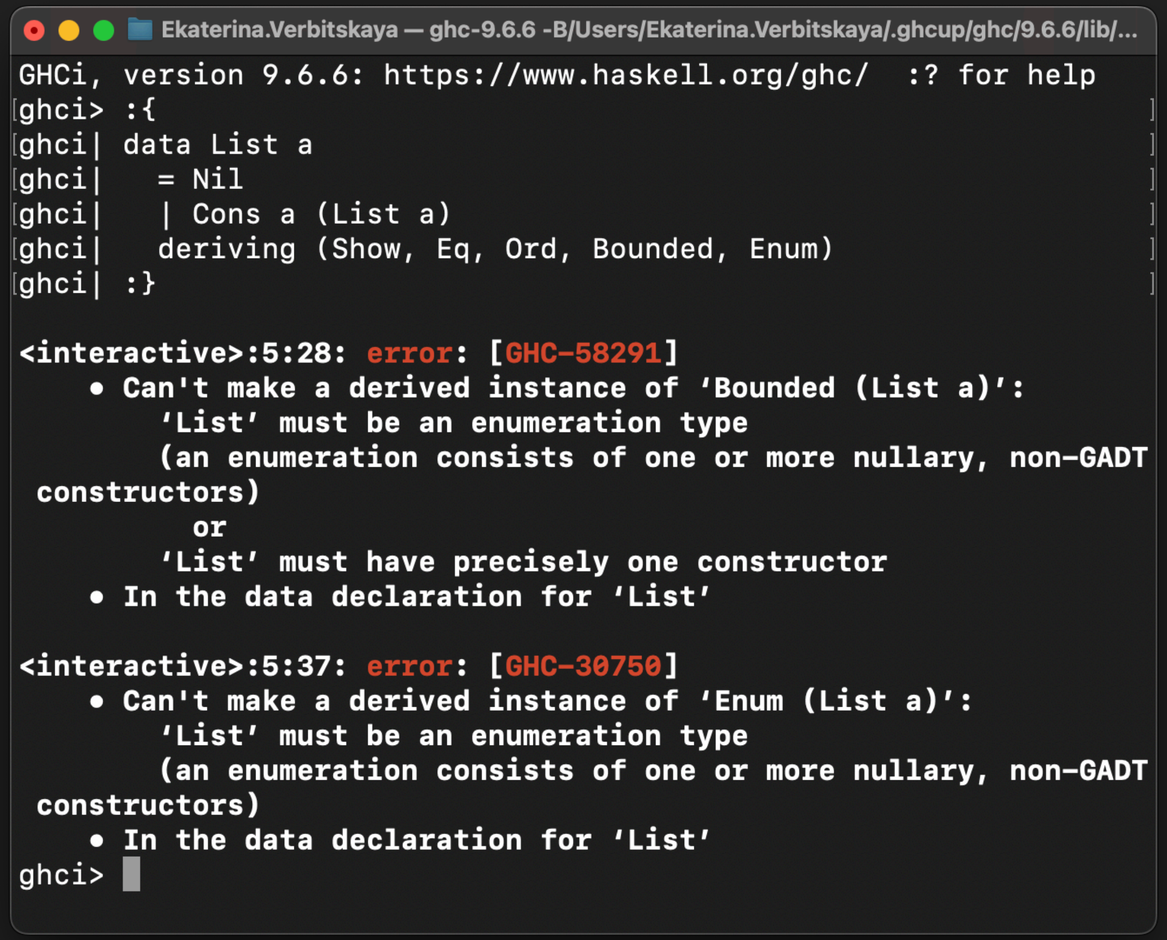
LIST

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



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



EXERCISES

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