INTRODUCTION TO

FUNCTIONAL PROGRAMMING

OUTLINE

- Organization
- Introduction
- Syntax basics
- Simplest types

MEET YOUR INSTRUCTOR

- Ekaterina Verbitskaia
- Researcher @ JetBrains Research PLAN, Programming Languages and Program Analysis Lab
- Interested in:
  - Functional and Relational Programming
  - Partial Evaluation, Metacomputations
- @kajigor

ORGANIZATION

- 2 classes per week
- Written exam (50% of your mark)
- Work during the term (50% of your mark)
  - Homework assignments (40/100 points)
  - Project at the end of the course (40/100 points)
  - Active participation during classes (40/100 points)

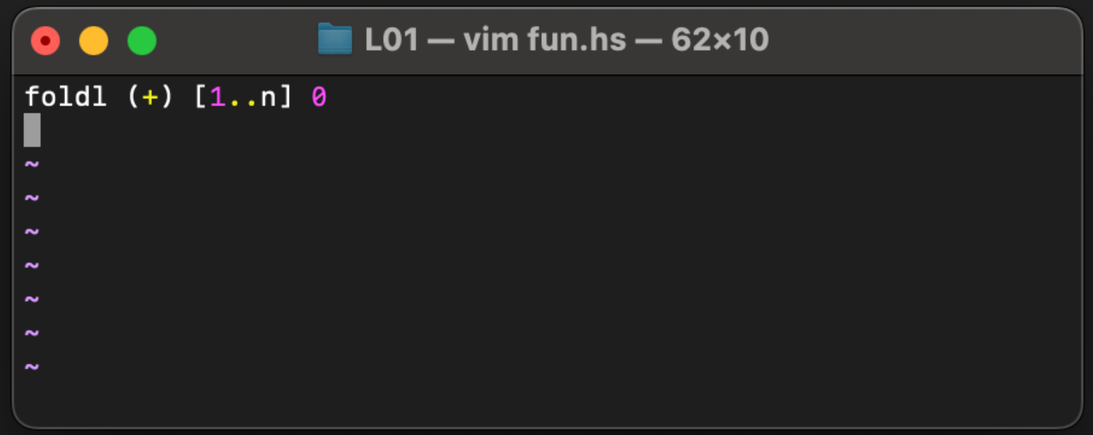
OUTLINE

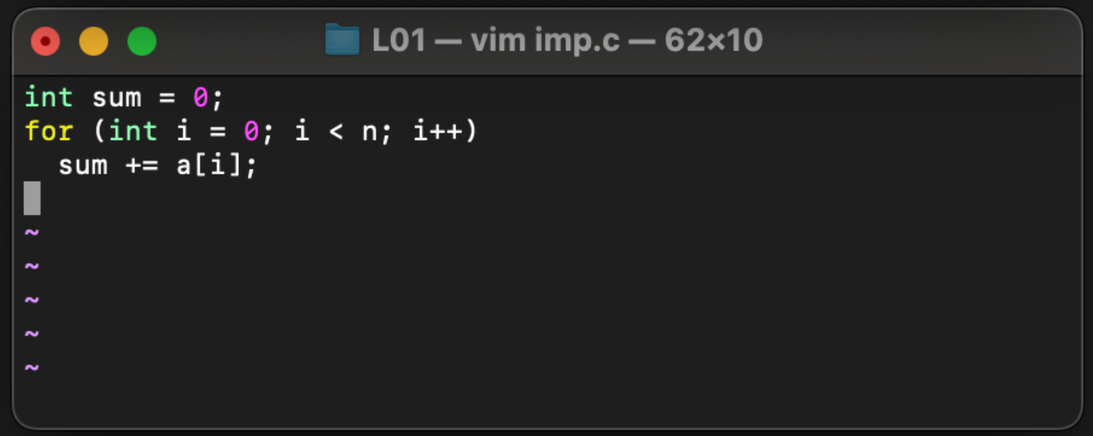
- Organization
- Introduction
- Syntax basics
- Simplest types

FUNCTIONAL PROGRAMMING

- Functional programming computes by means of evaluation of functions

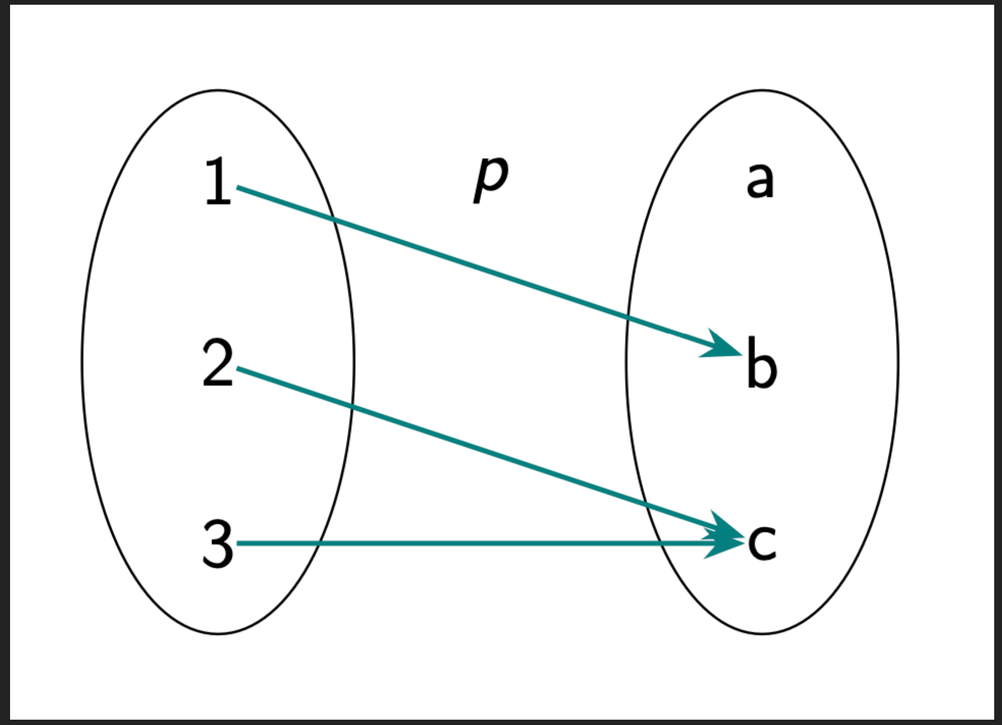
- Imperative programming uses statements to change the program's state





FUNCTION

- Fundamental building block of a program
- Pure
  - No side-effects such as IO or mutable state
  - **Always** the same result for the same input
- First-class: functions can be passed as arguments and returned as results
- Composition to create new functions



HASKELL

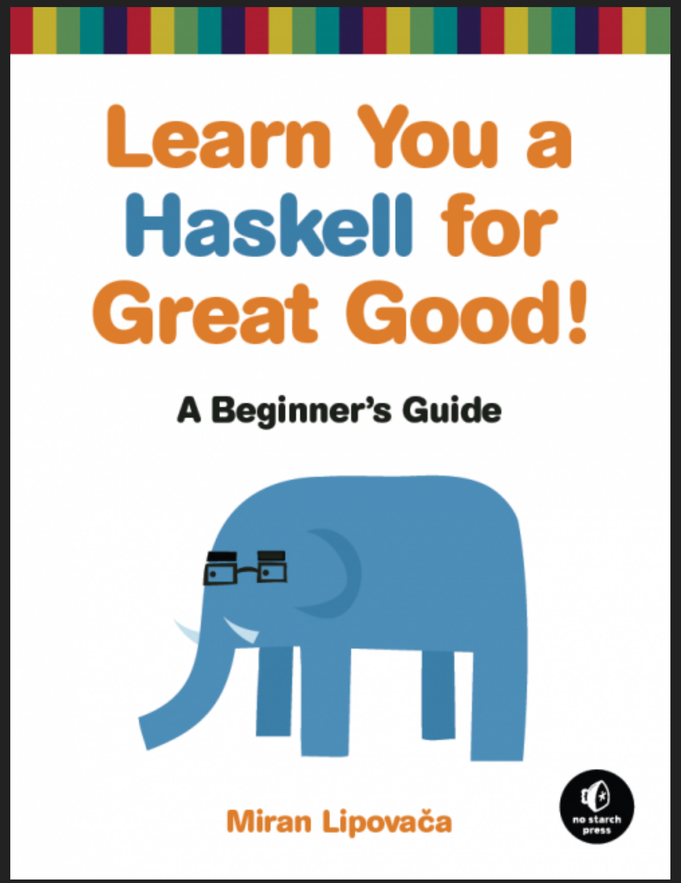
- Pure functional programming language
- Lazy
- Strong static type system
- Type inference

WHAT YOU NEED TO START

- Install with GHCup:
  - GHC 9.6.6 – compiler
  - Stack 3.1.1 – build system
  - HLS 2.9.0.1 – language server
- VSCode
  - Haskell extension

RECOMMENDED RESOURCES

- Books:
  - Learn You a Haskell for Great Good! (free)
  - Real World Haskell (free)
  - Haskell in Depth
- Cheat sheet
- Hoogle

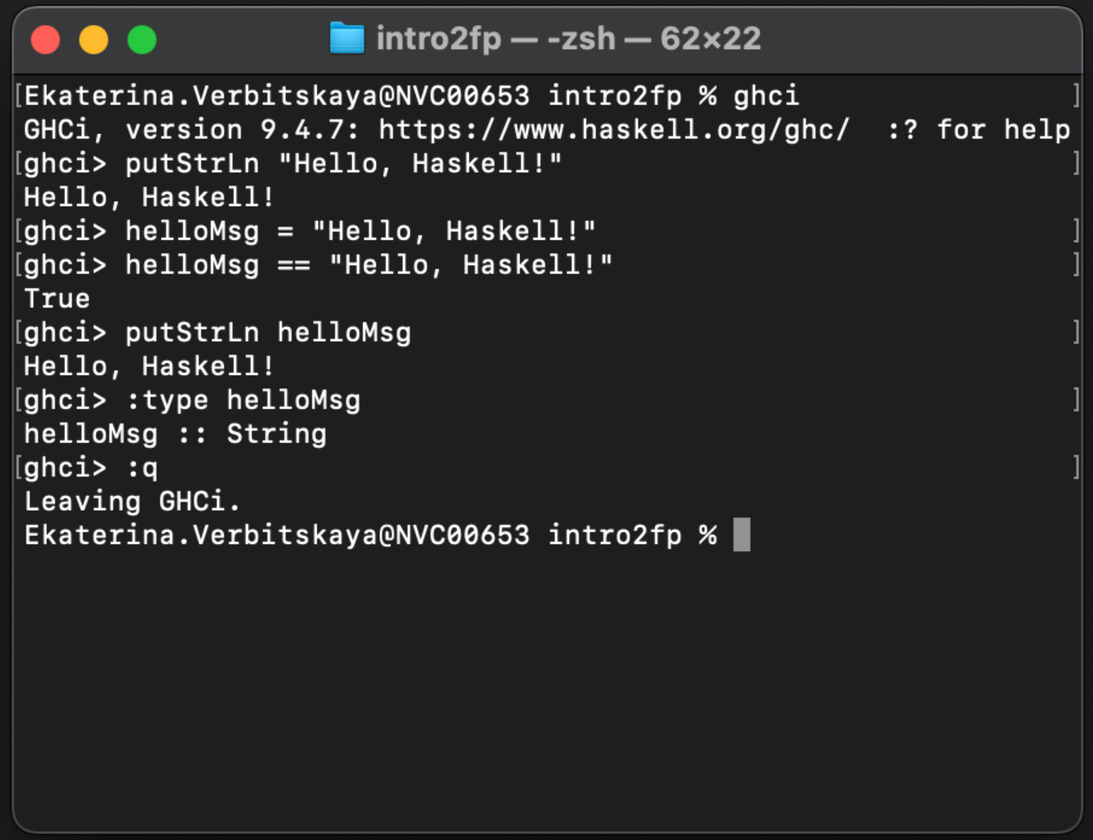


OUTLINE

- Organization
- Introduction
- Syntax basics
- Simplest types

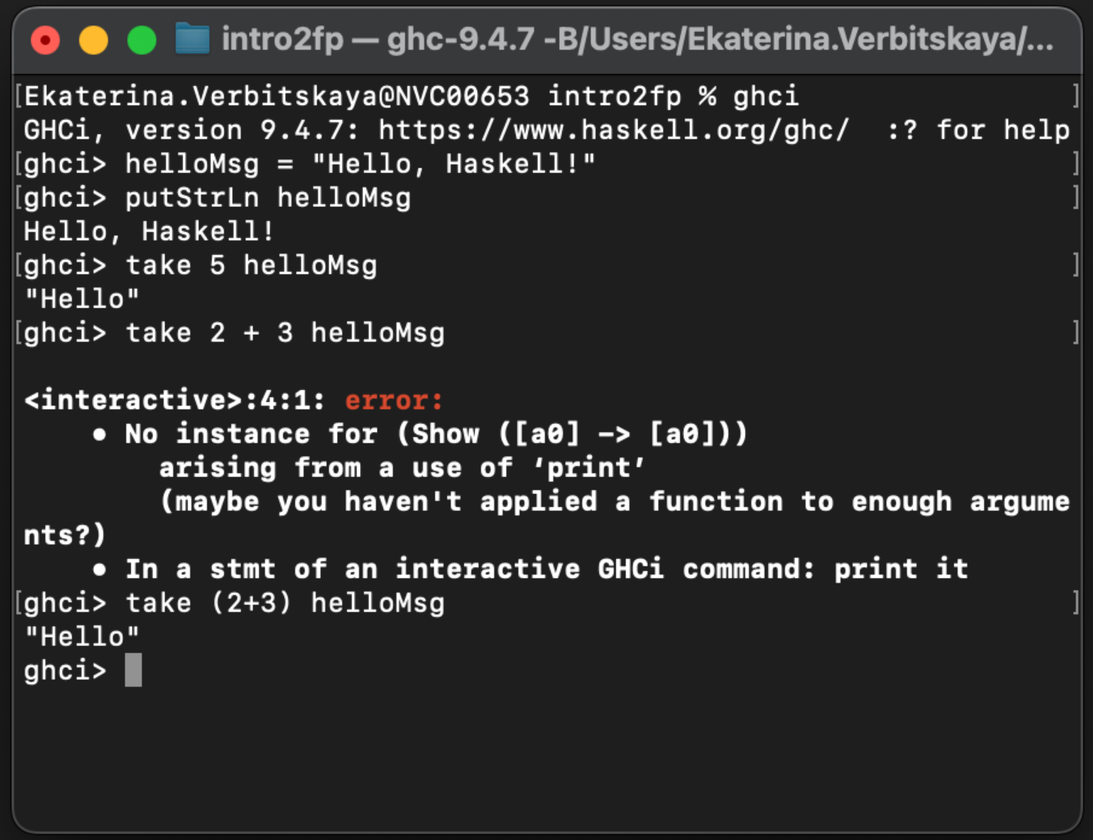
HELLO HASKELL

- Run ghci and type after the prompt `ghci>`
- `putStrLn` to output a string
- `=` for variable binding
- `==` for equality check
- `:type` to check the type of the expression
- `:q` to quite ghci



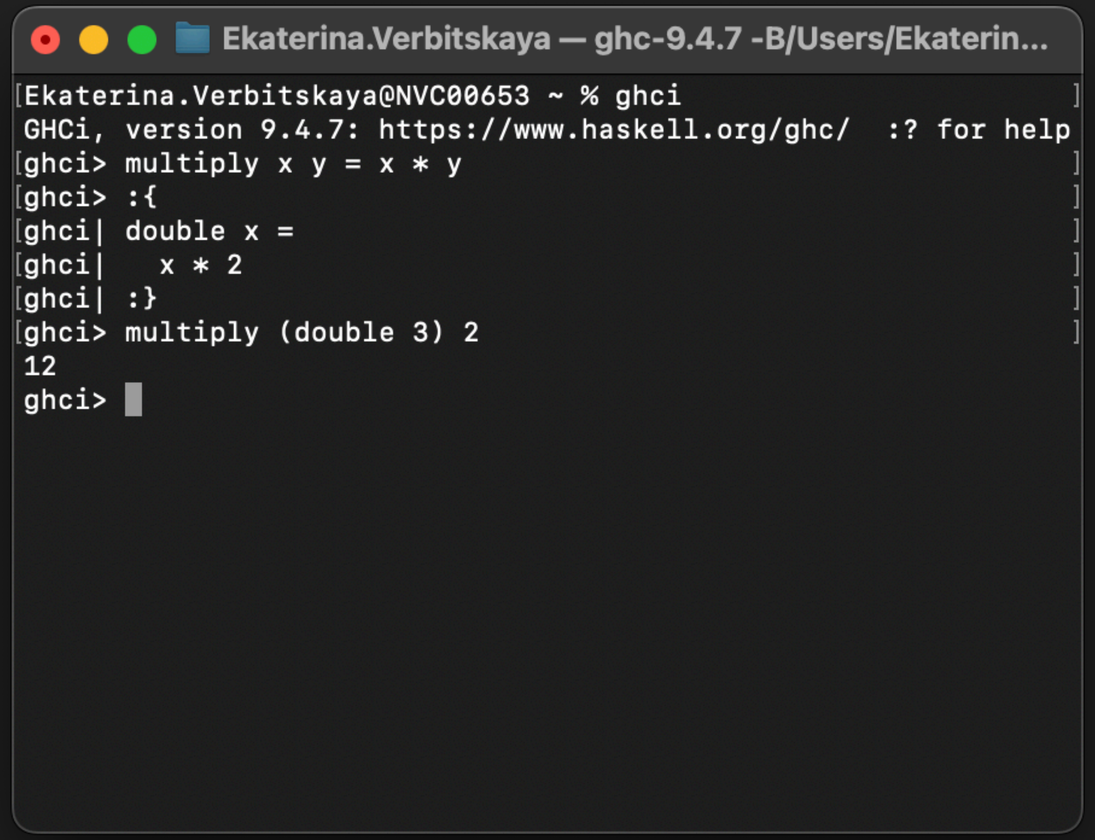
FUNCTION APPLICATION

- No parentheses
  - `putStrLn helloMsg`
  - `take 5 helloMsg`
- Function application has the highest precedence:
  - `take 2 + 3 helloMsg`
  - `(take 2) + (3 helloMsg)`
- Parentheses should instead be used:
  - `take (2 + 3) helloMsg`



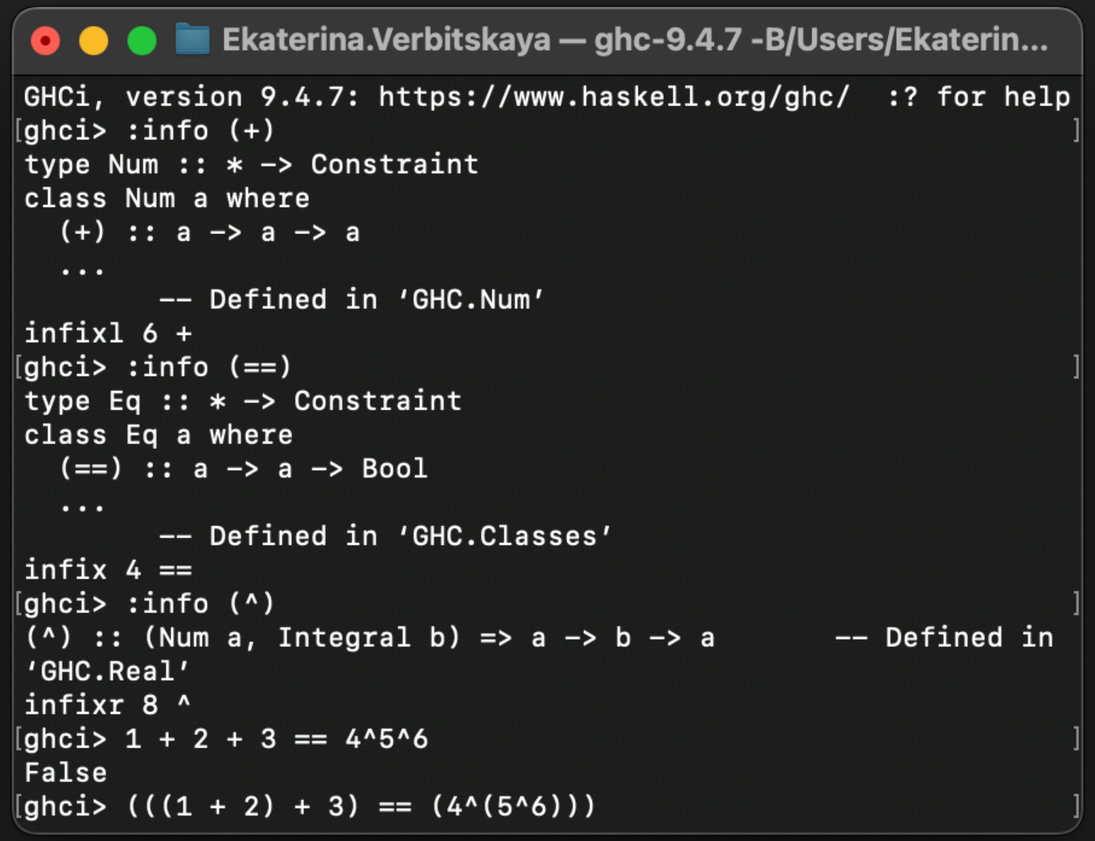
FUNCTION DEFINITION

- `name arg1 … argN = body`
  - No parentheses around args or body
  - Body must be on the same line or indented
- `name arg1 … argN =`
  `  body line 1`
  `  …`
  `  body line M`



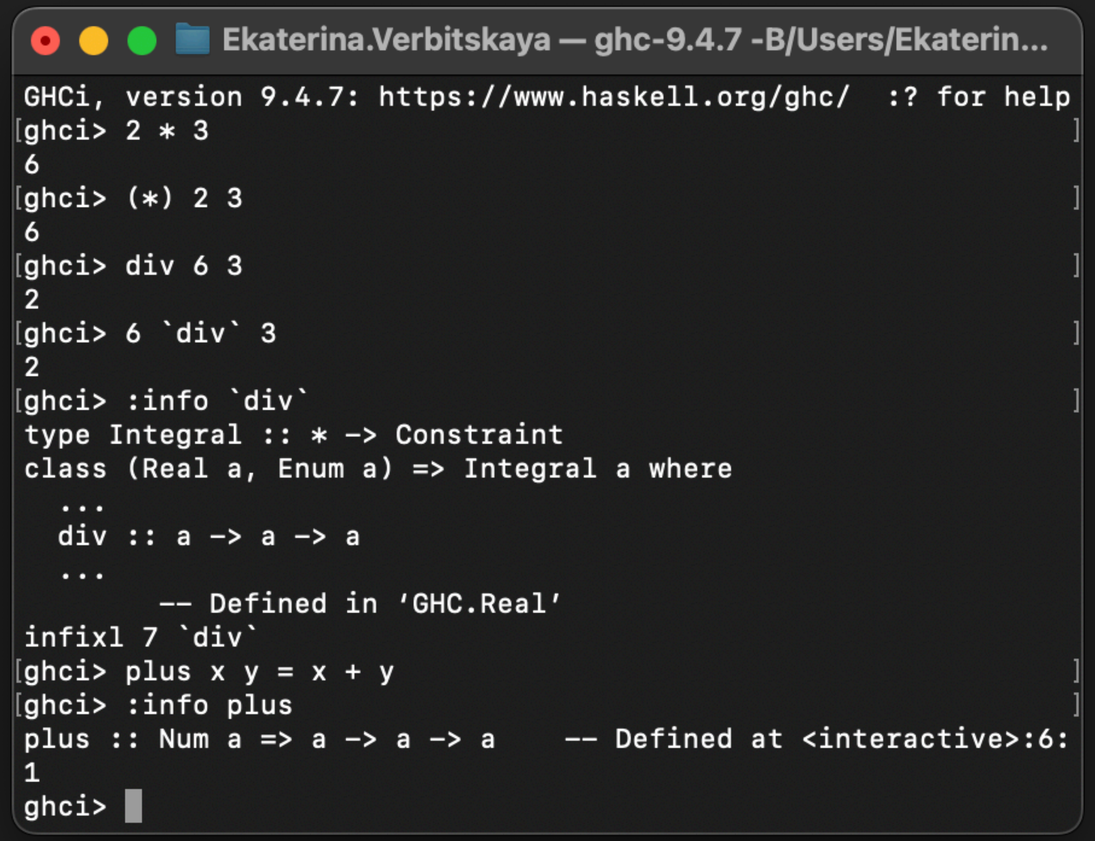
INFIX OPERATORS

- Binary operators are intended to be used between operands
- Can have different fixity, associativity and precedence: infixl 6 + means that (+) is
  - an infix operator
  - left associative
  - has precedence of 6 (can be from 0 to 9, higher precedence is applied first)



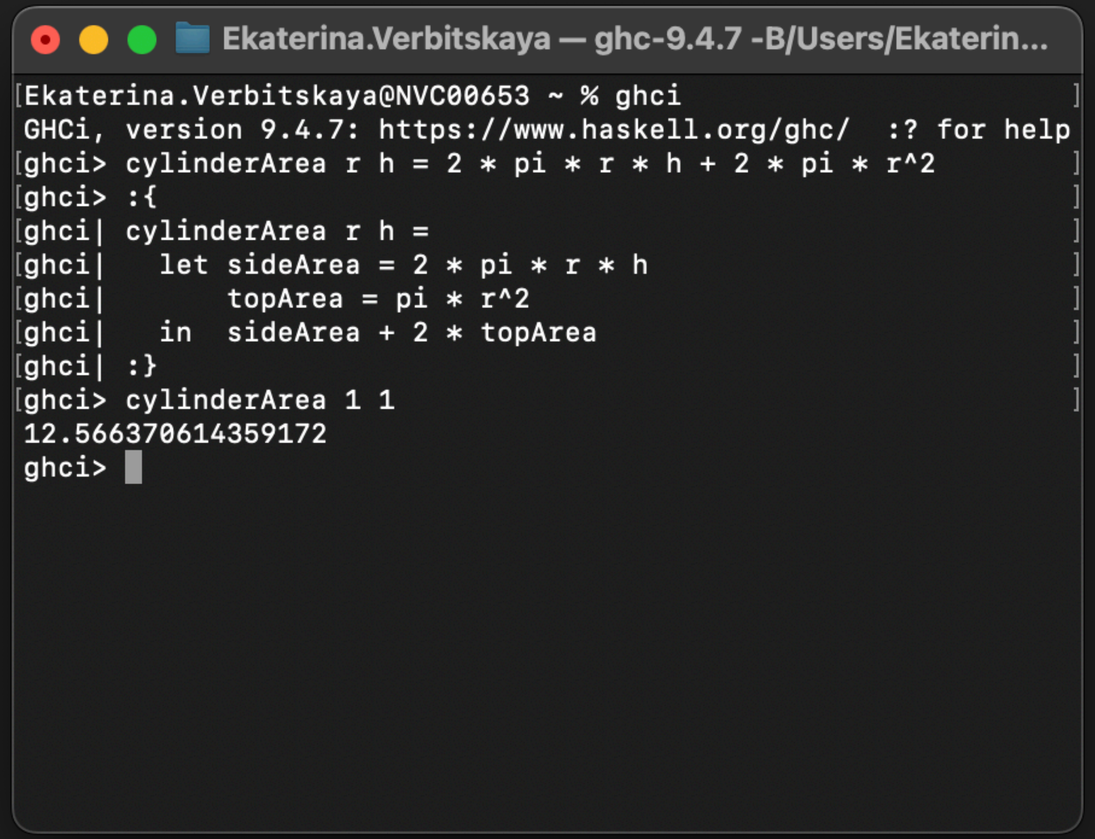
INFIX AND PREFIX NOTATION

- Infix operators can be used in prefix form
- Function names can be used in infix form
  - When used in infix form, may be supplemented with fixity specification
    - By default, no specification



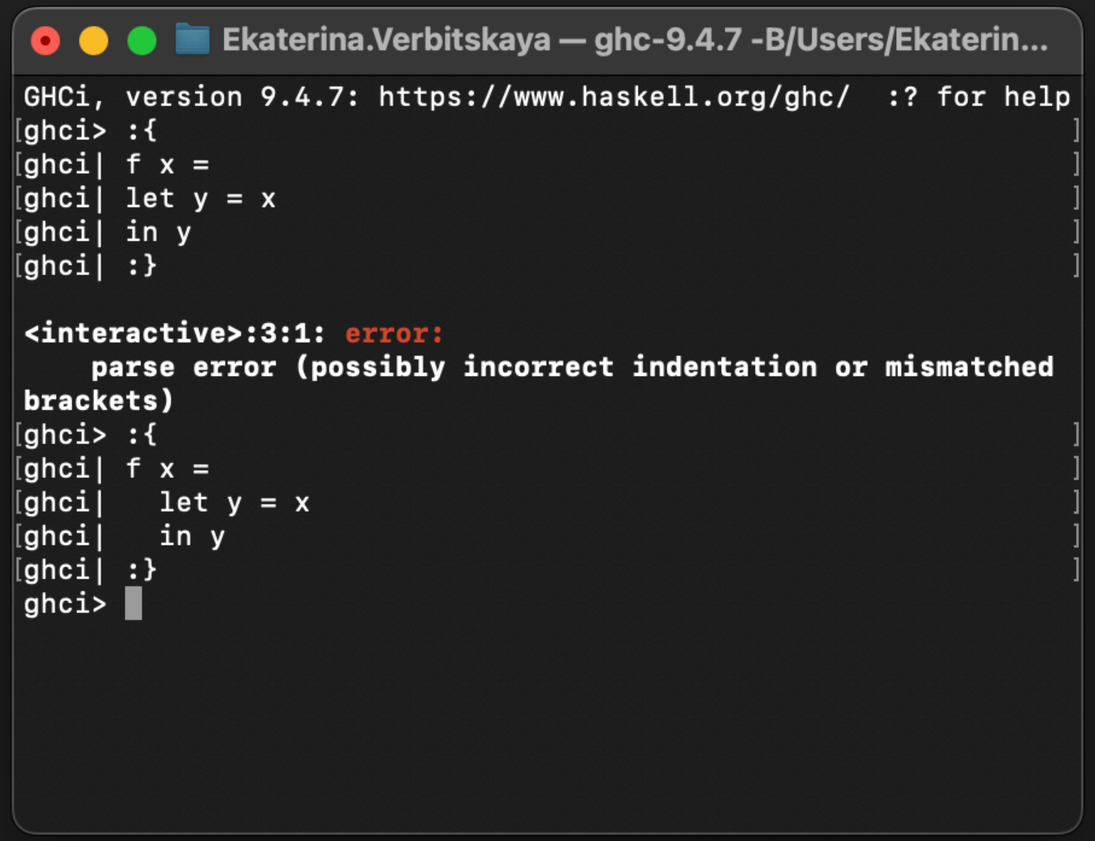
VARIABLE BINDING

- To make your code more readable, introduce names for subexpressions
- ```
  let varName1 = value1
      …
      varNameN = valueN
  in  body
  ```


  ```

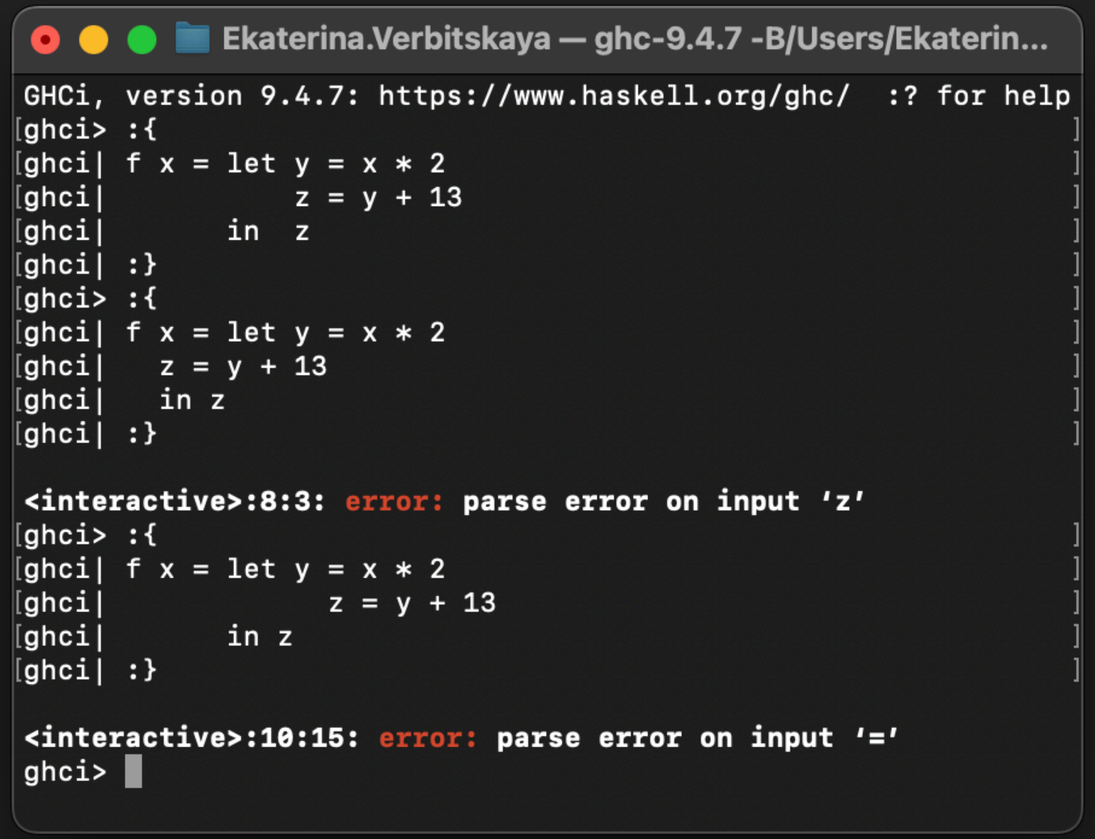
INDENTATION IS IMPORTANT

- Code which is part of some expression should be indented further in than the beginning of that expression



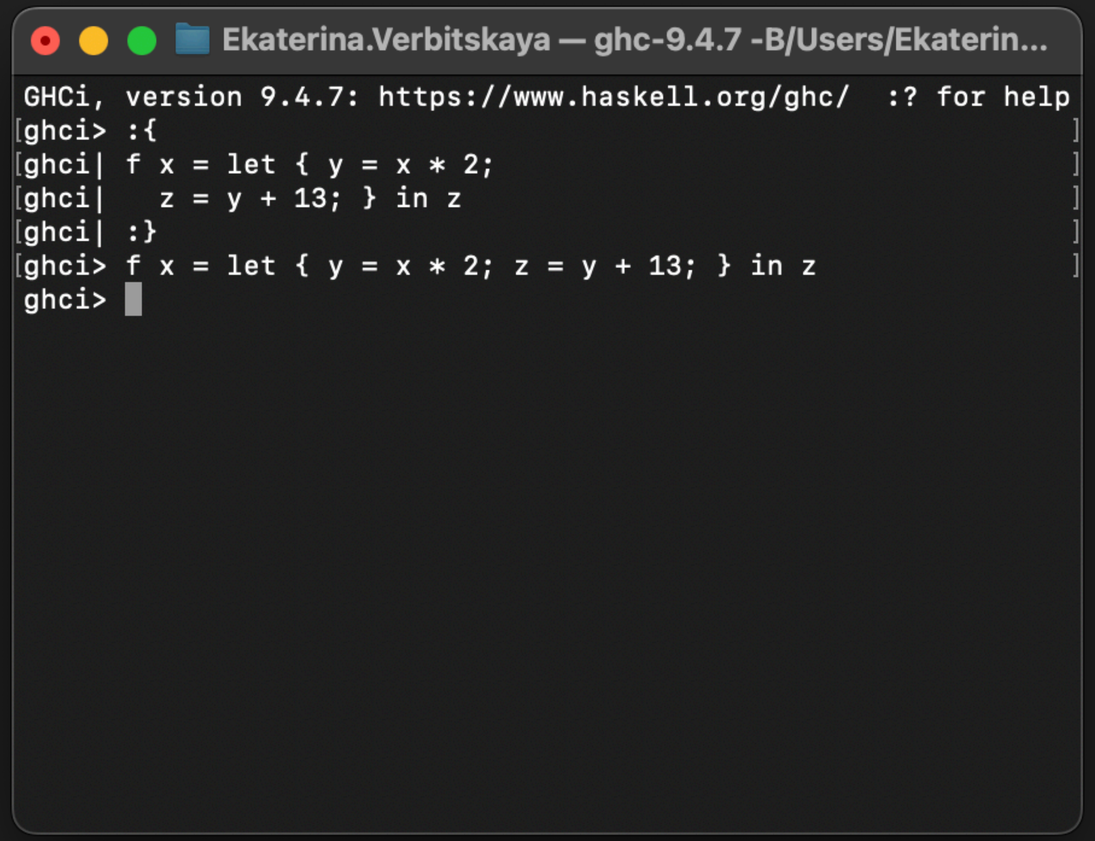
MORE INDENTATION EXAMPLES

- Align subsequent let-bindings



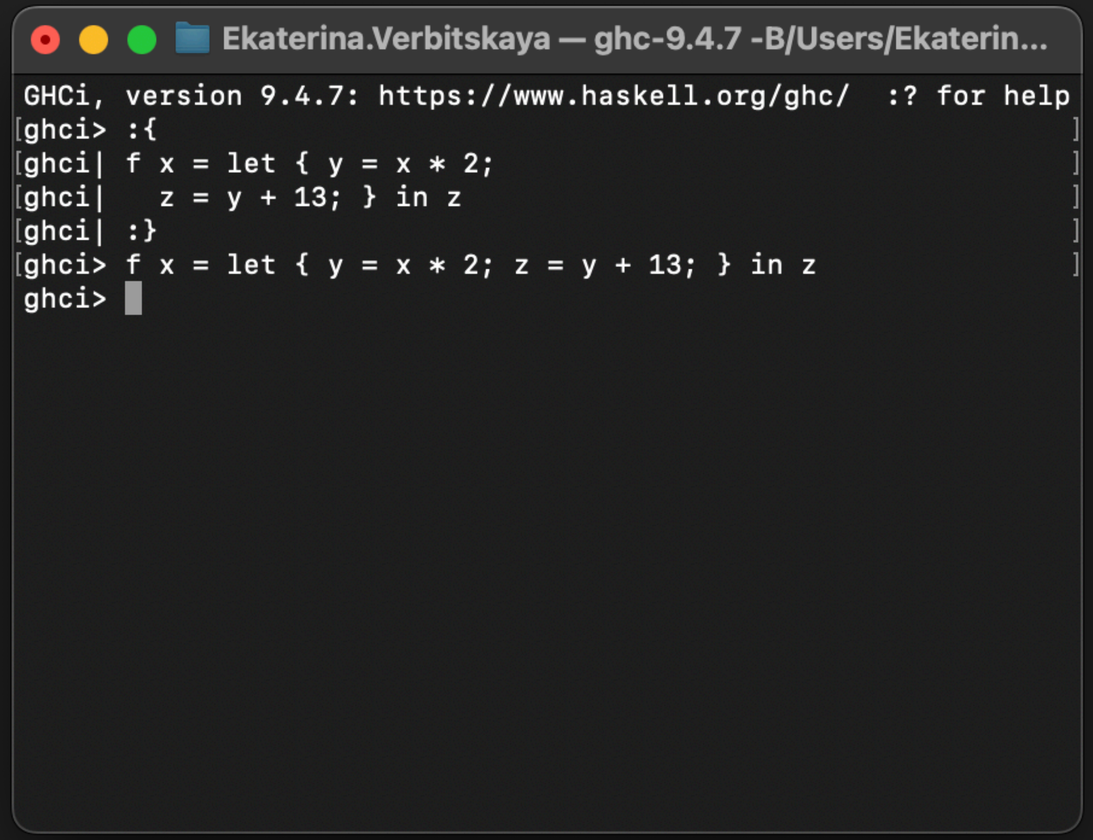
EXPLICIT ; AND {} IN PLACE OF INDENTATION

- When you want to write a one-liner
- When you generate a Haskell program from another program



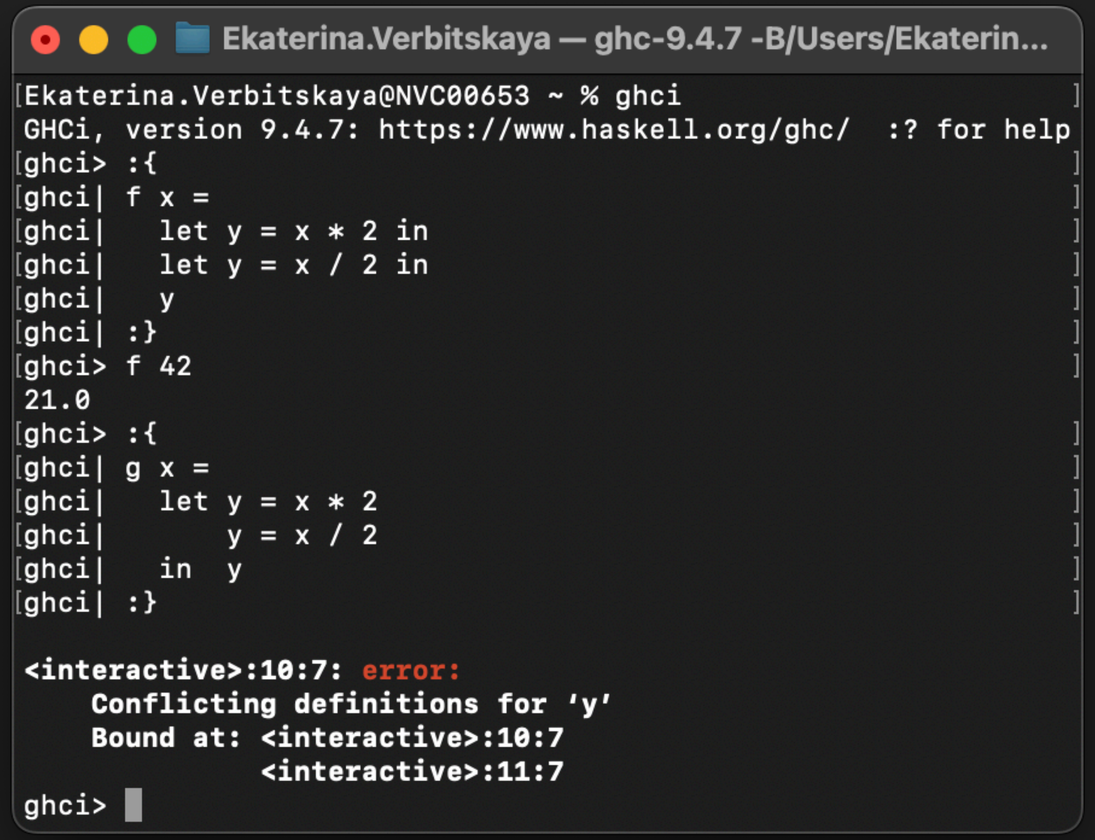
EXPLICIT ; AND {} IN PLACE OF INDENTATION

- When you want to write a one-liner
- When you generate a Haskell program from another program
- When you miss C and Java too much



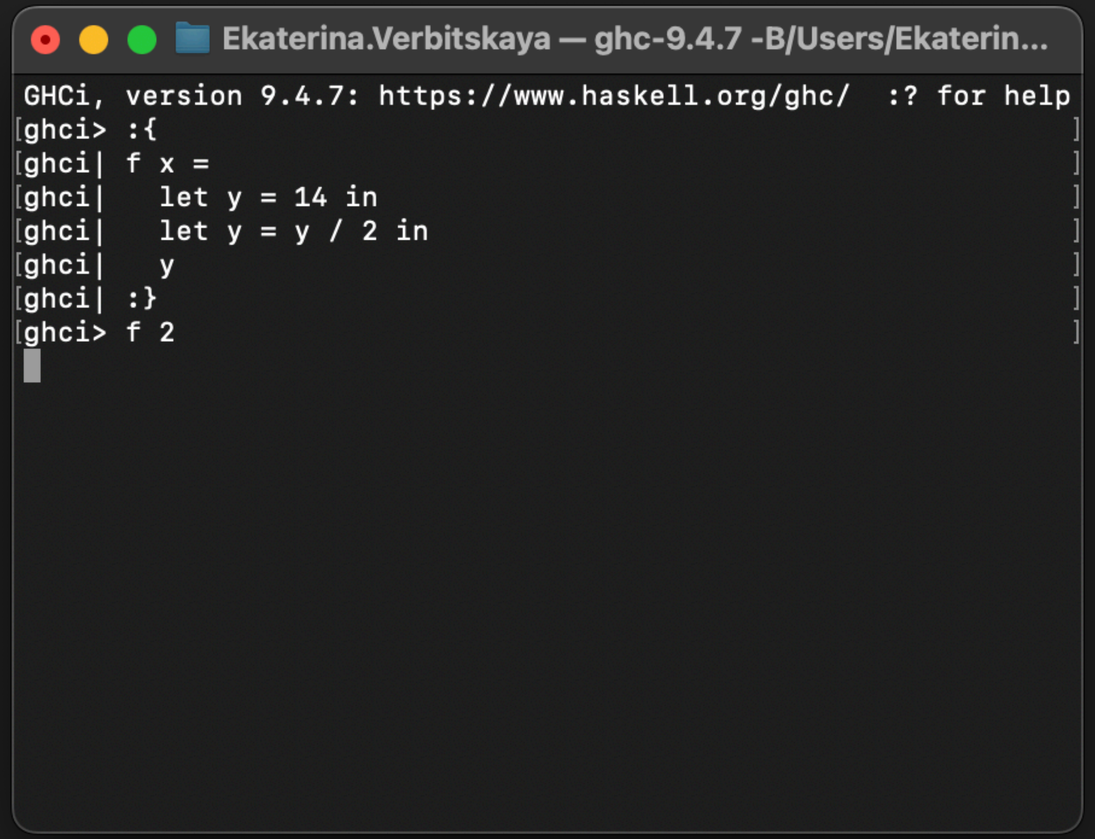
LET-BINDING IS NOT ASSIGNMENT

- Let-bindings only associate a variable name with the value
- Variable name is only available in the body
- Let-bindings can **shadow** each other
- Multiple let-bindings of the same variable in the same block are prohibited



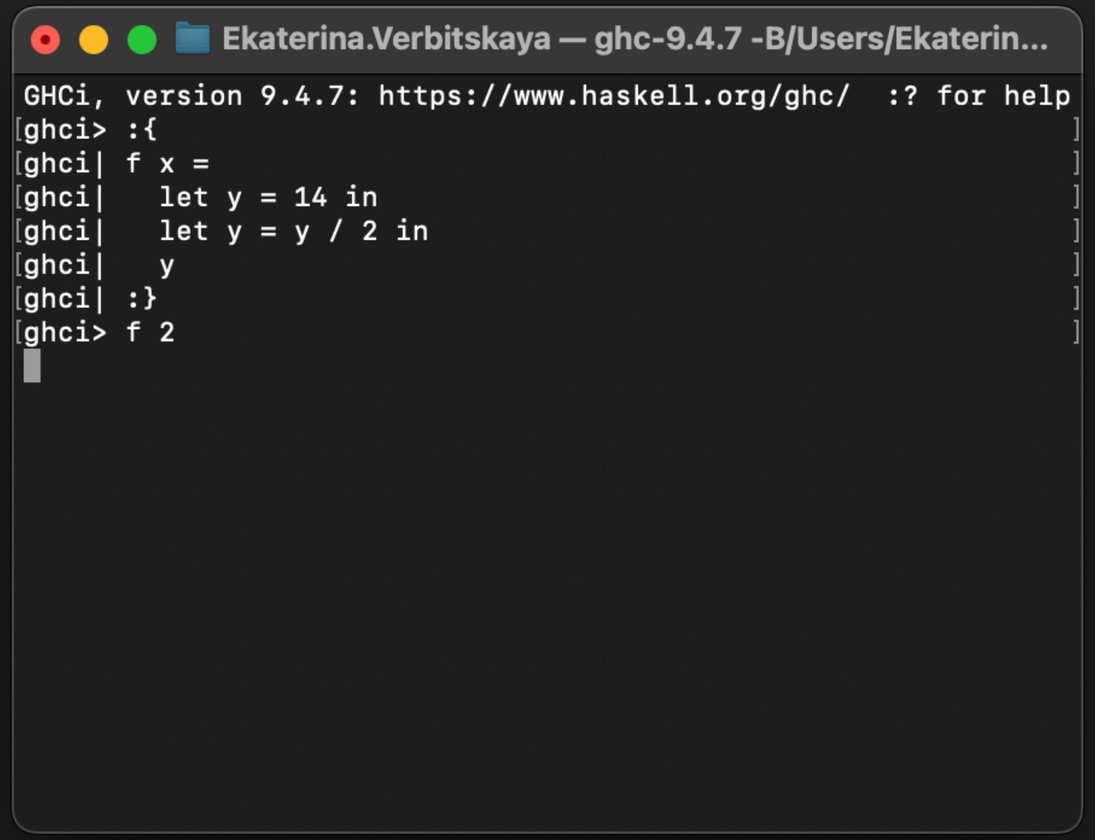
SHADOWING IN LET-BINDINGS

- What will happen if we execute the code on the screenshot?



SHADOWING IN LET-BINDINGS

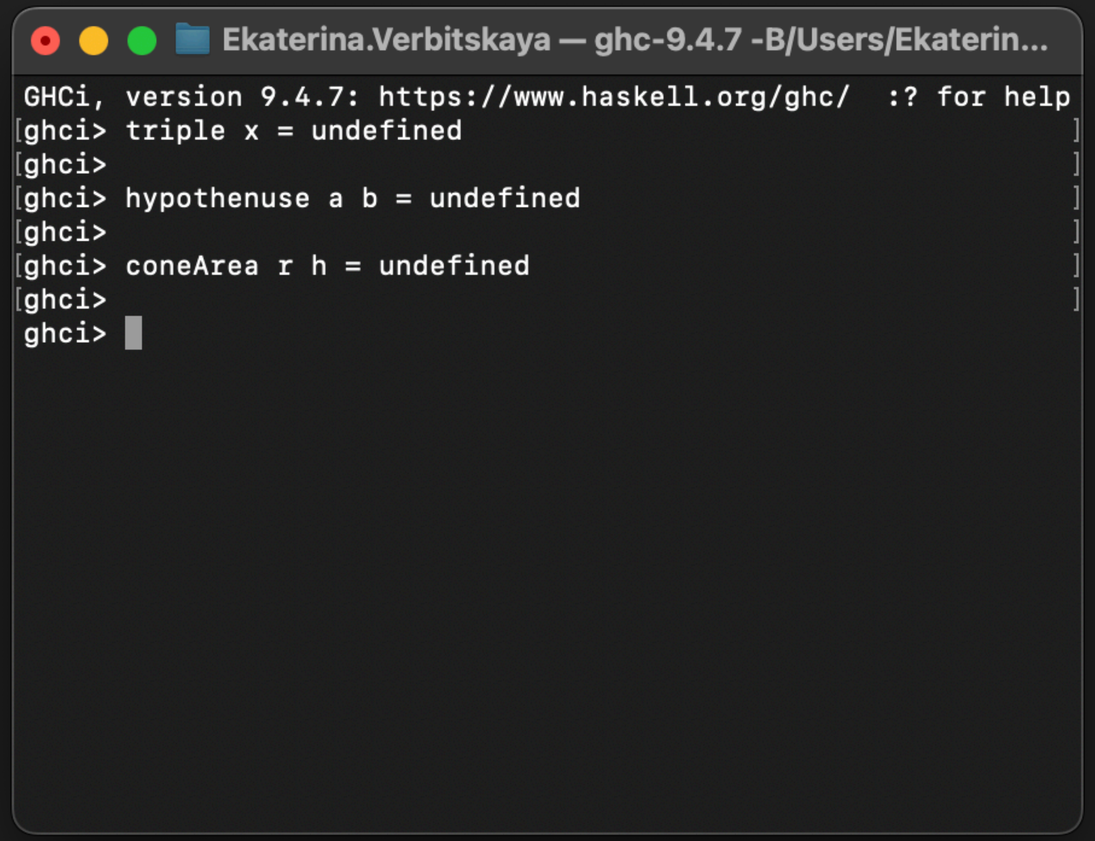
- What will happen if we execute the code on the screenshot?



- It diverges!
- The second let shadows the first binding
- In the body of let y = y / 2, y refers to itself!

EXERCISES

- Implement functions:
  - triple that triples the input
  - hypothenuse that computes the length of a hypothenuse
    - Use the function sqrt from Prelude
  - coneArea which computes the area of a cone
    - Use the constant pi from Prelude

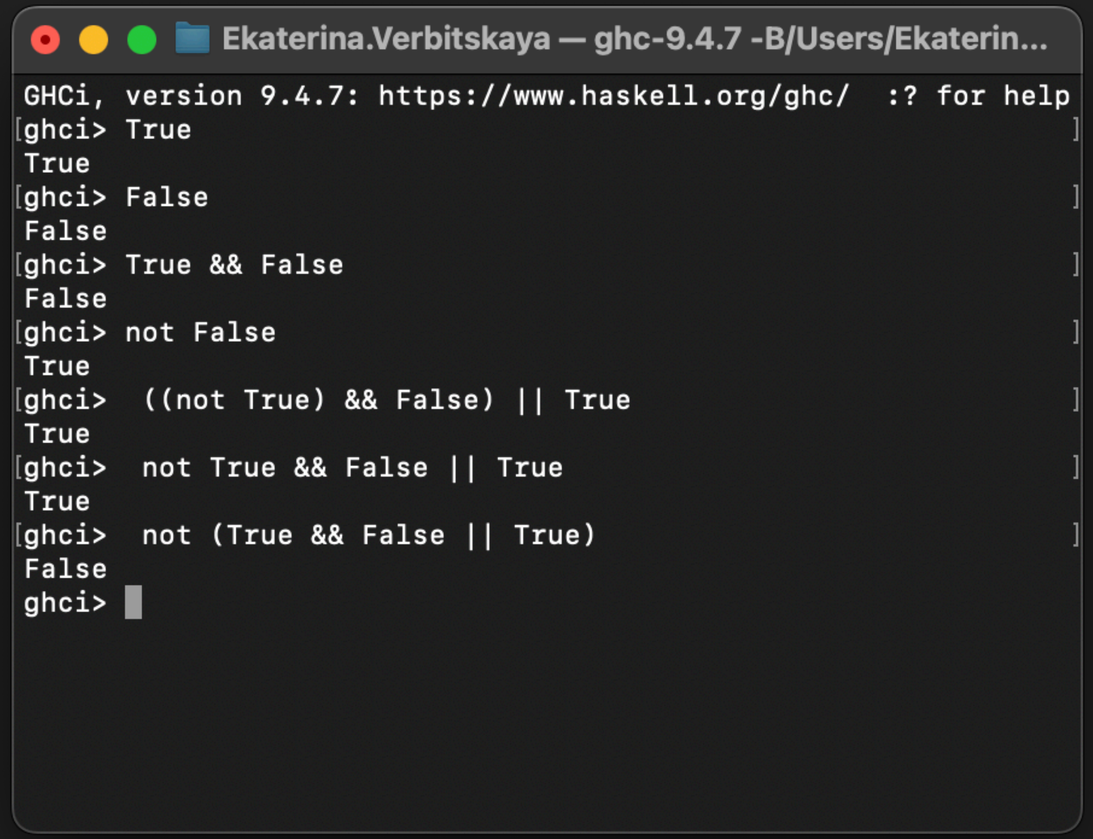


OUTLINE

- Organization
- Introduction
- Syntax basics
- Simplest types

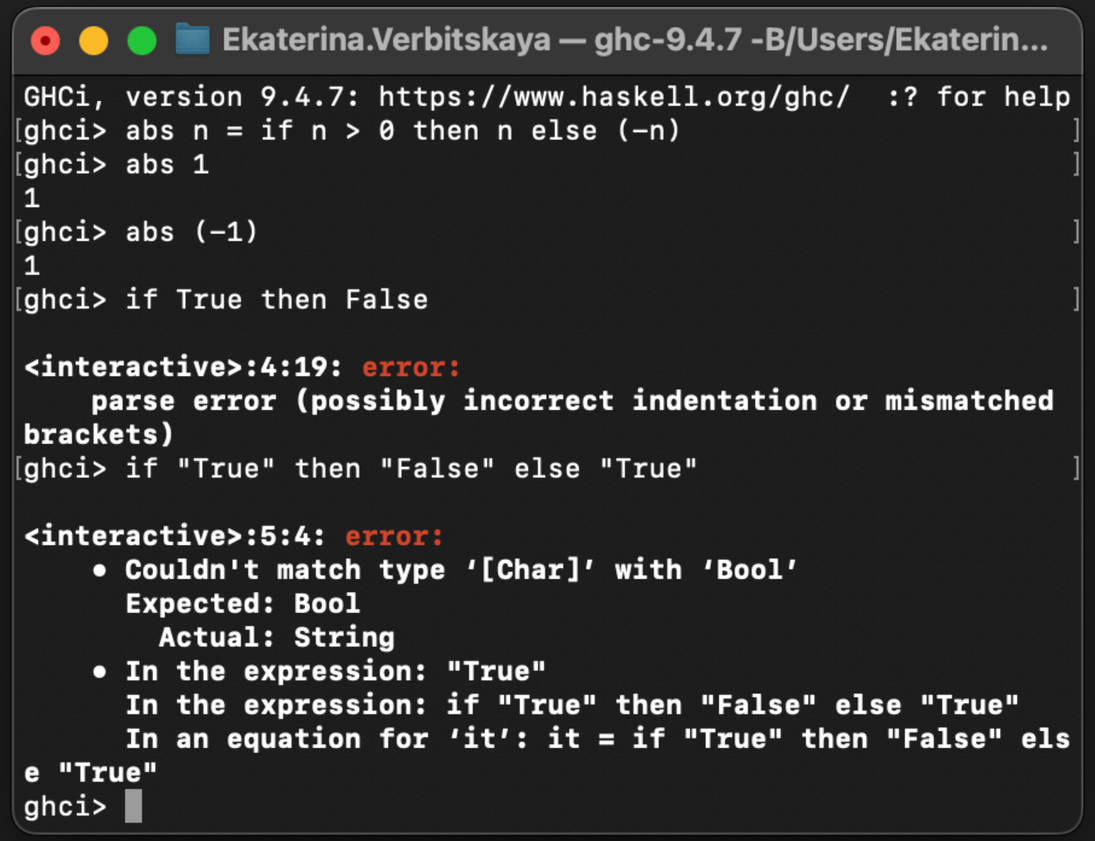
BOOLEAN

- Two values: True, False
- Usual operations over booleans



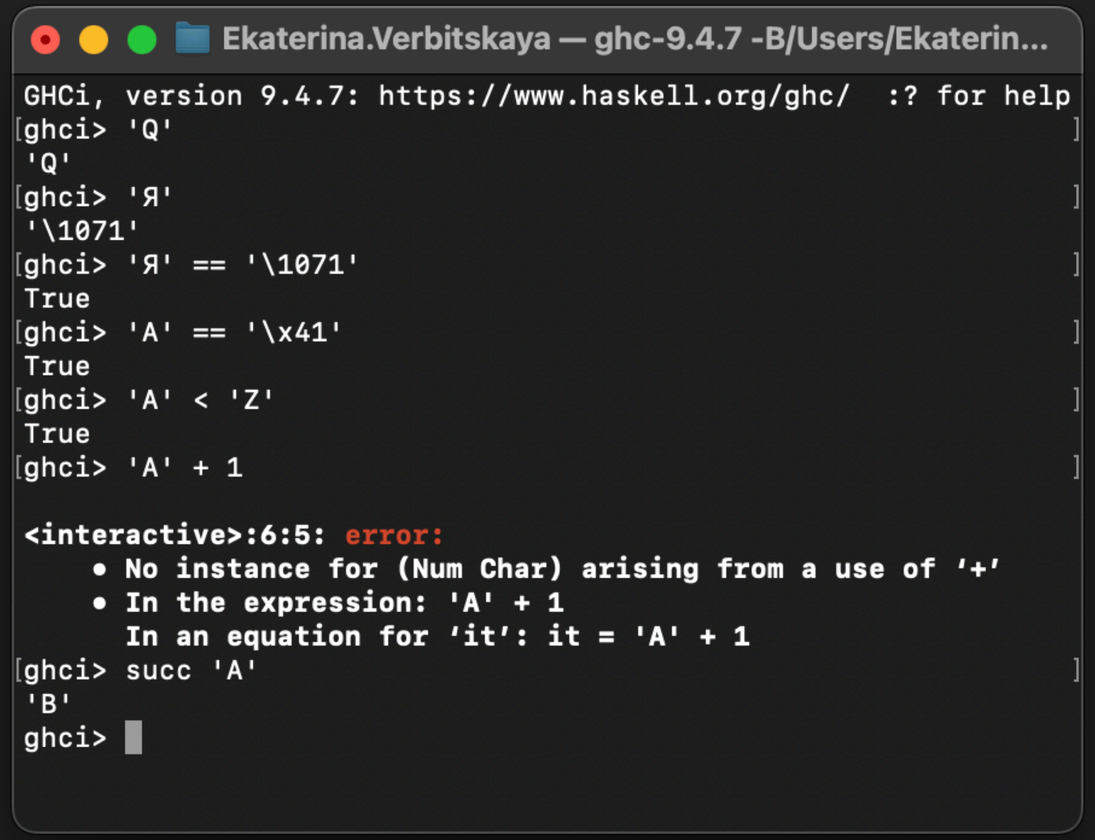
IF-EXPRESSION

- Condition must be a Boolean value
- There must be both branches
- Both branches have to have the same type



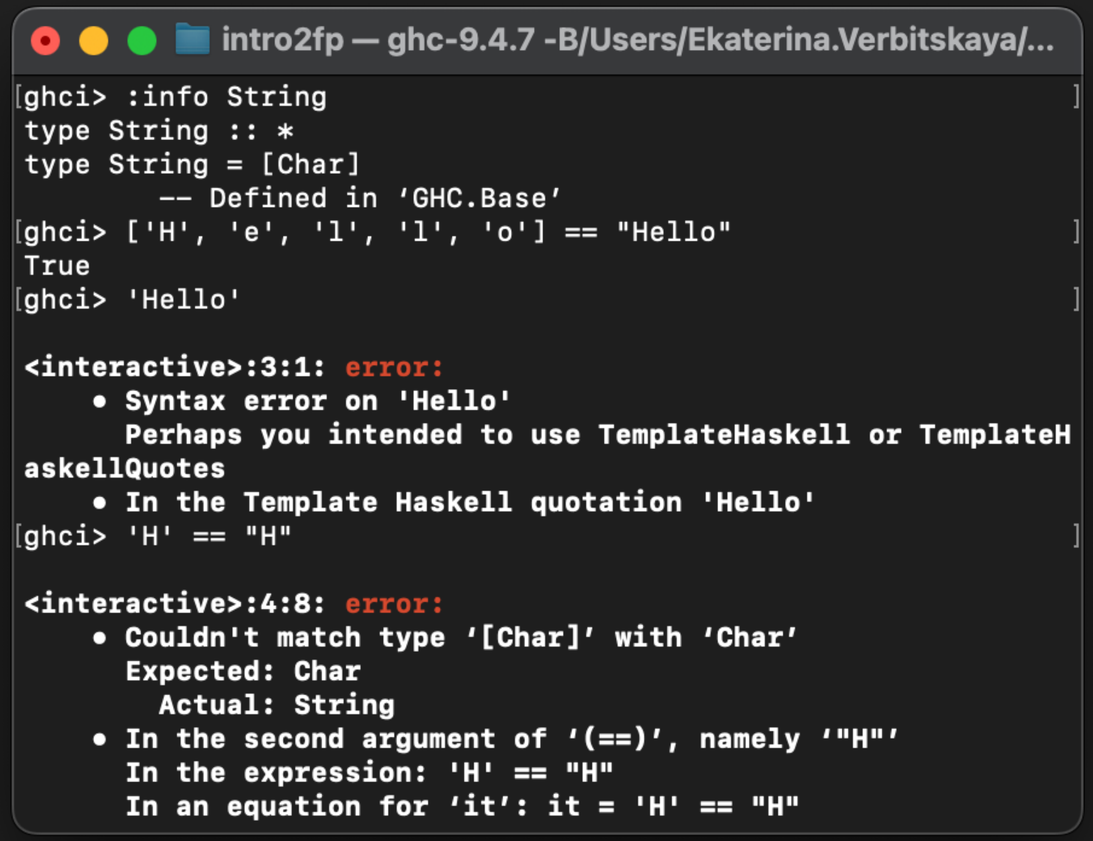
CHARACTERS

- Single quotes
- Unicode
- You can use decimal, hexadecimal or octal notation
- Characters aren't numbers!
- But they can be ordered and you can get the next/previous character
- '\\' is \; '\'' is '



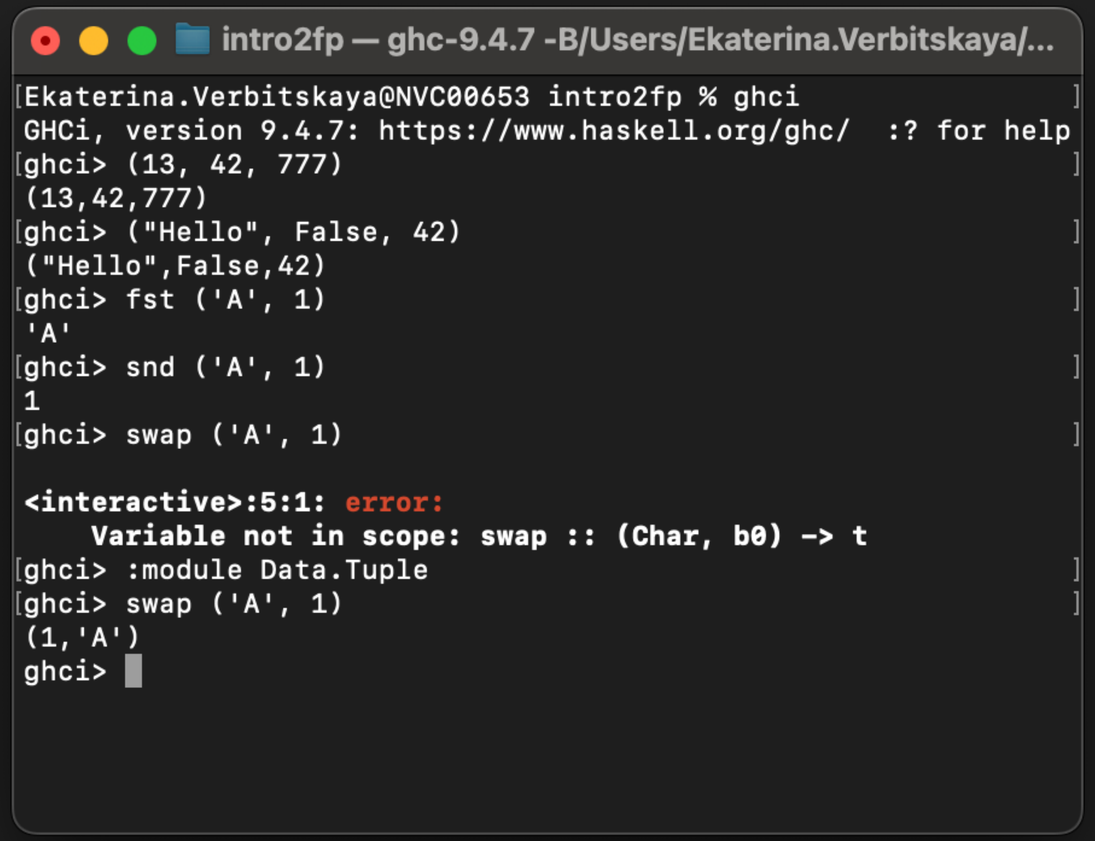
STRINGS

- Strings are lists of characters
- Strings are in double quotes
- A single-character string is not the same as the character



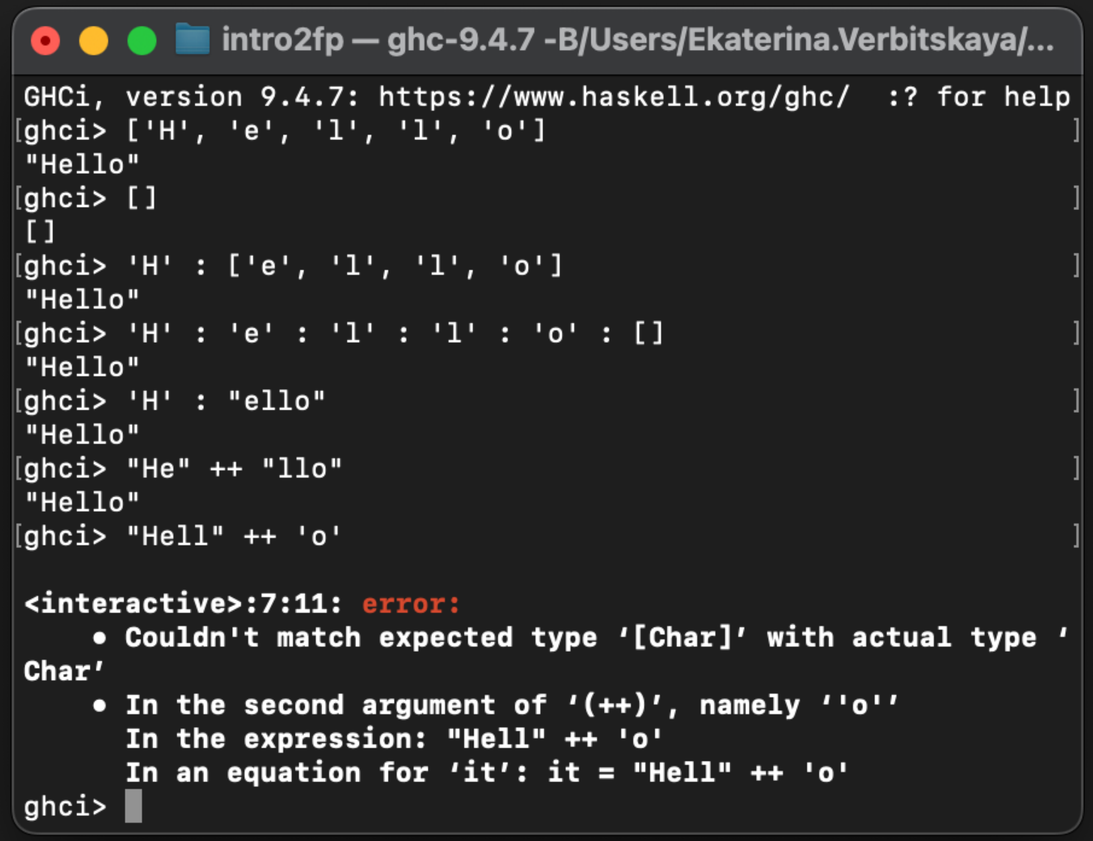
TUPLES

- Combine several values into one
  - From 2 to 62 elements (in GHC)
- Heterogeneous: elements are allowed to have different types
  - `("Hello", False, 42)`
- Special functions for pairs
  - `fst, snd`
  - `swap`



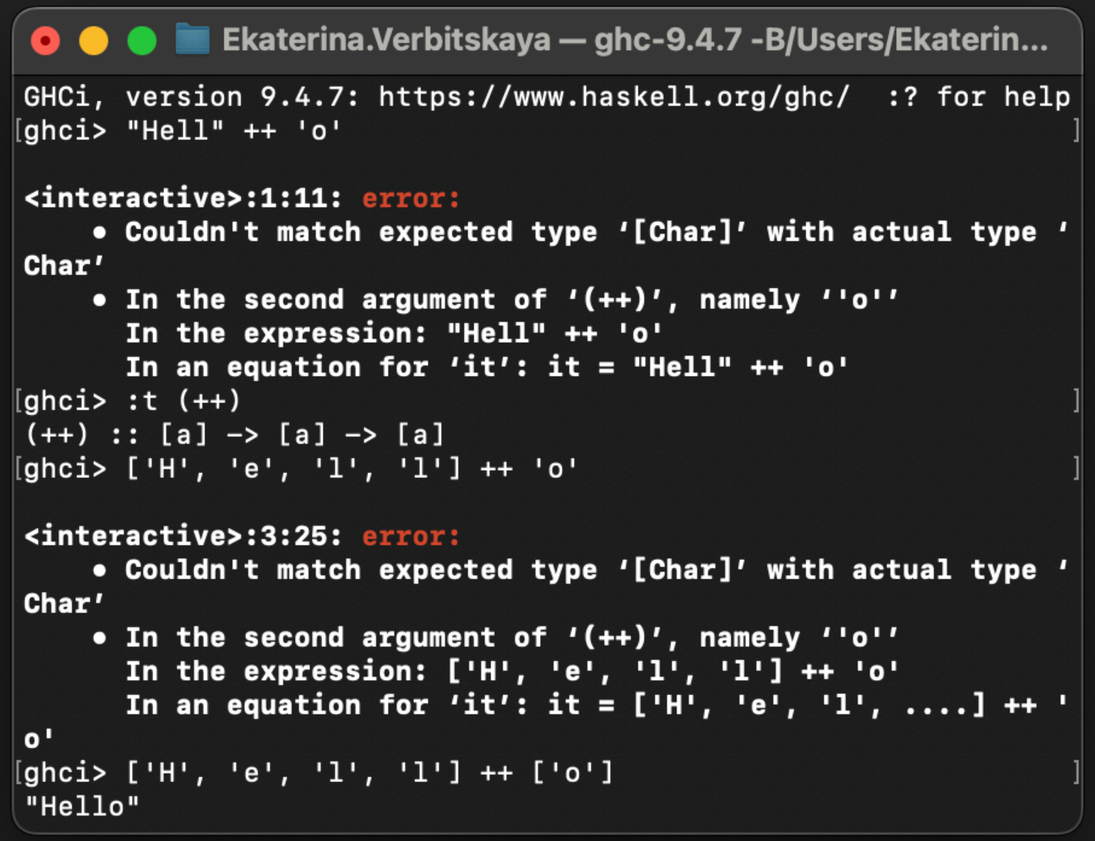
LISTS

- Homogeneous: all elements have the same type
- List literal: `['H', 'e', 'l', 'l', 'o']`
- Empty list: `[]`
- List constructor:
  - `'H' : ['e', 'l', 'l', 'o']`
  - `'H' : 'e' : 'l' : 'l' : 'o' : []`
  - `'H' : "ello"`
- List concatenation: `"He" ++ "llo"`



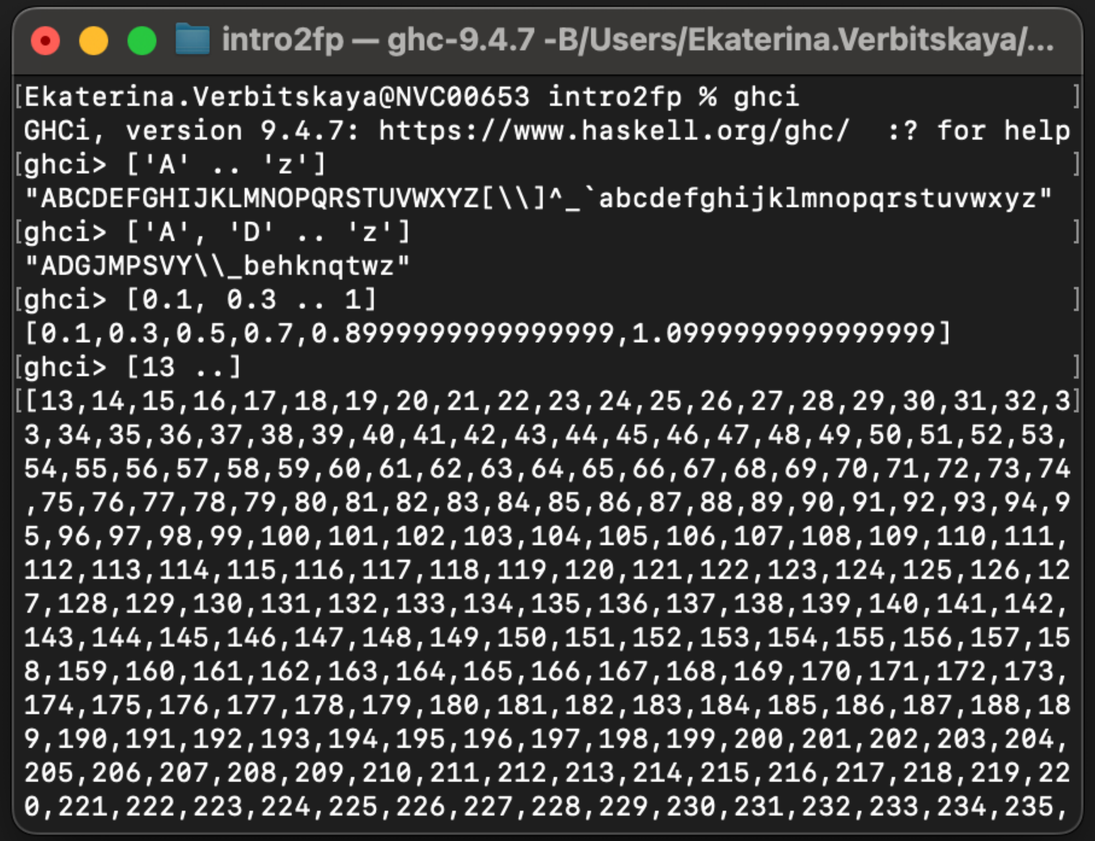
BABY'S FIRST TYPE ERRORS

- Static type system
  - The type of every expression is known and compile time
  - The type cannot change in run time.
- Type inference
  - Haskell can (almost often) infer types



RANGES

- Range from ‘A’ to ‘z’
  - `['A' .. 'z']`
- Range from ‘A’ to ‘z’, every third character
  - `['A', 'D' .. 'z']`
- Don’t use ranges with floating point numbers
  - `[0.1, 0.3 .. 1]`
- Range with no upper limit — infinite list
  - `[13 ..]`



EXERCISES

- Implement functions:
  - even that checks whether an Int is even
  - odd that checks whether an Int is odd
  - evens that returns an infinite list of even numbers
  - f that returns the first character of a string, along with its predecessor and successor
    - use head