INTRODUCTION TO

FUNCTIONAL PROGRAMMING

OUTLINE

- Function evaluation
- Operator sections
- Functions as arguments
- Higher-order functions on Lists

EVALUATION IN ARITHMETICS

- Evaluate the innermost subexpression
- Replace it with the result
- Repeat until the expression is a number

- 1 + 2 * 3 - 4
- (1 + (2 * 3)) - 4
- (1 + 6) - 4
- 7 - 4
- 3

EVALUATION IN ARITHMETICS: VARIABLES

- Evaluate the innermost subexpression
- If it contains variables, find their definition and evaluate it
- Replace the subexpression with the result
- Repeat until the expression is a number

- let x = 4 - 2 in let y = 4 in 1 + x * 3 - y
- let x = 2 in let y = 4 in 1 + x * 3 - y
- let y = 4 in 1 + 2 * 3 - y
- let y = 4 in 1 + 6 - y
- let y = 4 in 7 - y
- 7 - 4
- 3

EVALUATION OF A FUNCTION

- When a function is applied to a value, replace its argument in the body with the value
- Evaluate the body like an expression
- Repeat until the body becomes a value

- inc i = i + 1
- inc 2 => 2 + 1 => 3

- double x = x + x
- double 3 => 3 + 3 => 6

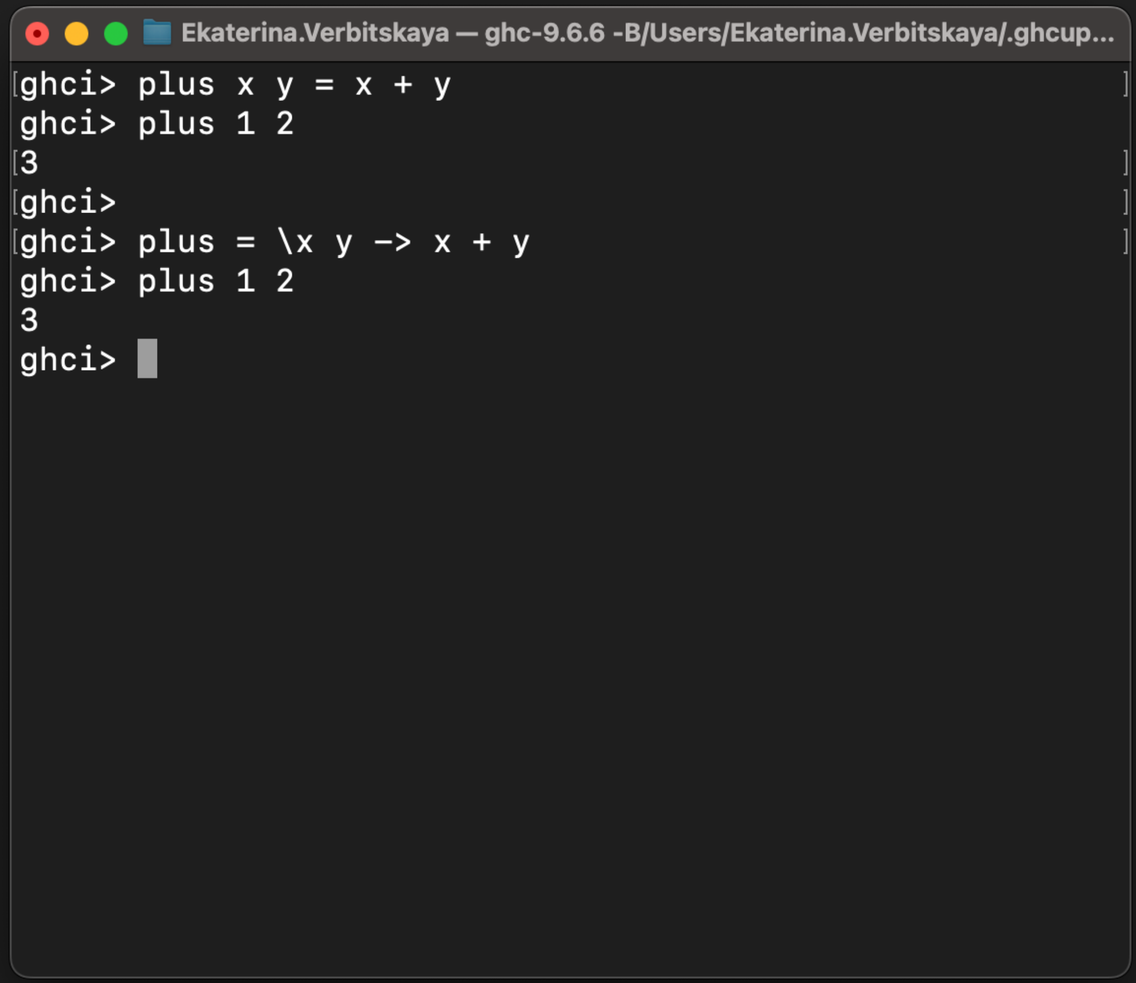
WHAT IF THERE ARE 2 ARGUMENTS?

- Pretty much the same thing happens:
  - Arguments are replaced with the values provided
  - Body is evaluated

- plus x y = x + y
- plus 1 2 => 1 + 2 => 3

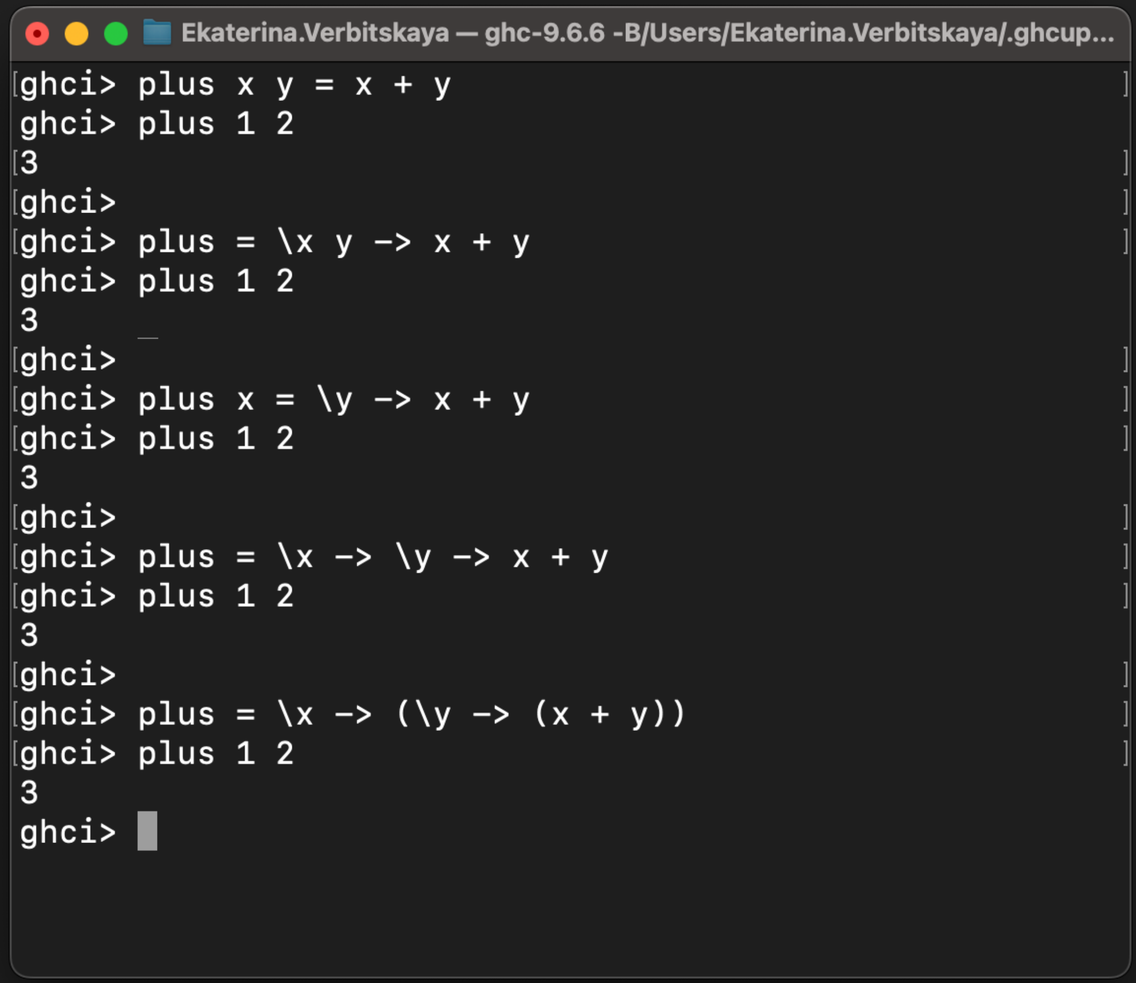
ANONYMOUS FUNCTIONS

- An anonymous function doesn't have a name
  - a.k.a. lambda-function
- It can be helpful when you only need the function once in a small context
- `\varName1 … varNameN → body`
- `body` lasts until the end of the line



CARRYING

- `f x y .. z = body` is a shorthand for
  `f = \x → \y → .. \z → body`
- Functions of multiple arguments don’t exist!



EVALUATION OF FUNCTIONS

- `f x y .. z = body` ≡≡≡
  `f = \x → \y → .. \z → body`
- Evaluation of a function with multiple arguments is in fact evaluation of several functions with a single arguments

- `plus = \x → \y → x + y`
- `plus 1 2 ⇒`
  - `(plus 1) 2 ⇒`
  - `((\x → \y → x + y) 1) 2 ⇒`
  - `(\y → 1 + y) 2 ⇒`
  - `1 + 2 ⇒`
  - `3`

PARTIAL APPLICATION

- What happens if we don’t provide enough values for all arguments of a function?

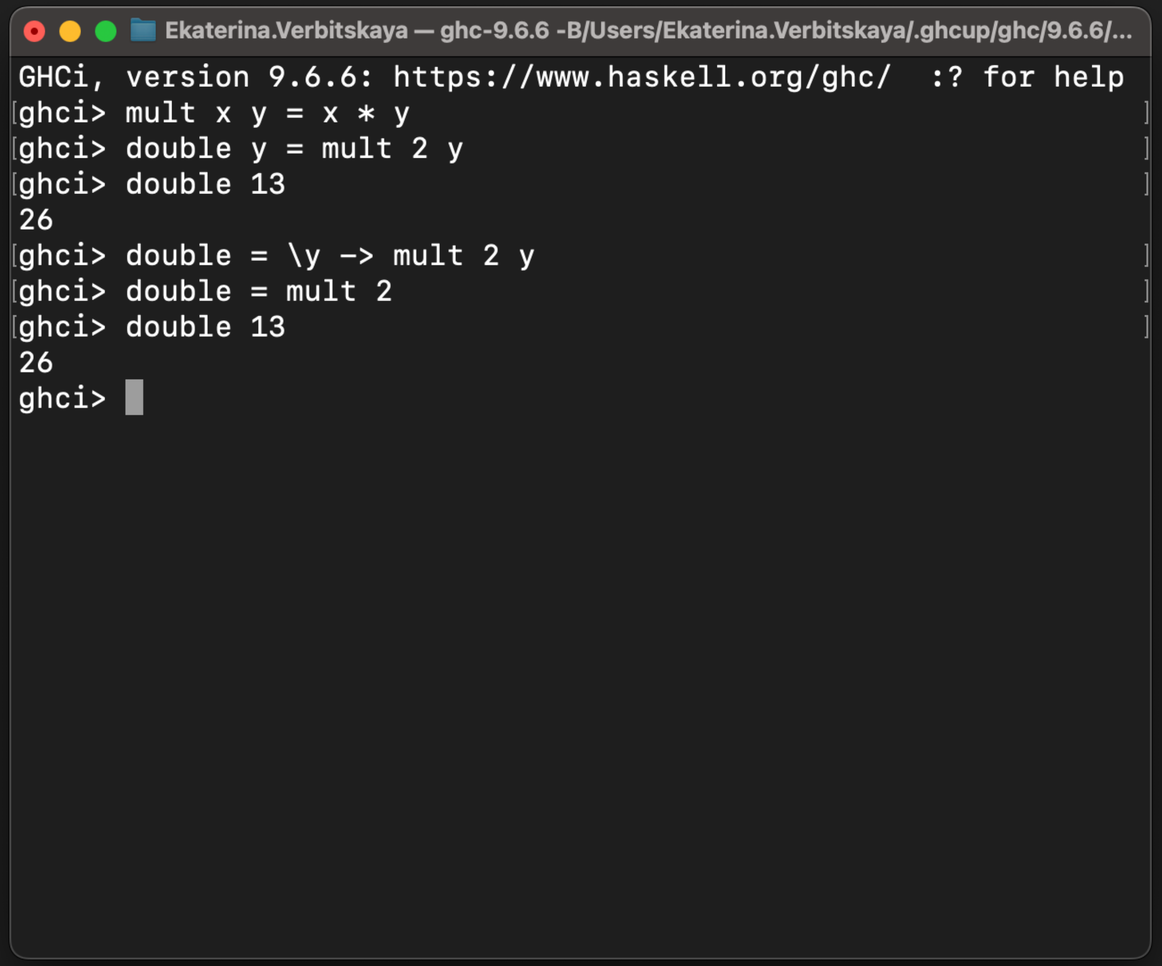
PARTIAL APPLICATION

- What happens if we don't provide enough values for all arguments of a function?
- **Partial application** happens
  - A new function is created
  - Only arguments provided are replaced with the values
  - The function is said to be partially applied

- `plus = \x → \y → x + y`
- `plus 1 ⇒`
  - `(\x → \y → x + y) 1 ⇒`
  - `\y → 1 + y`

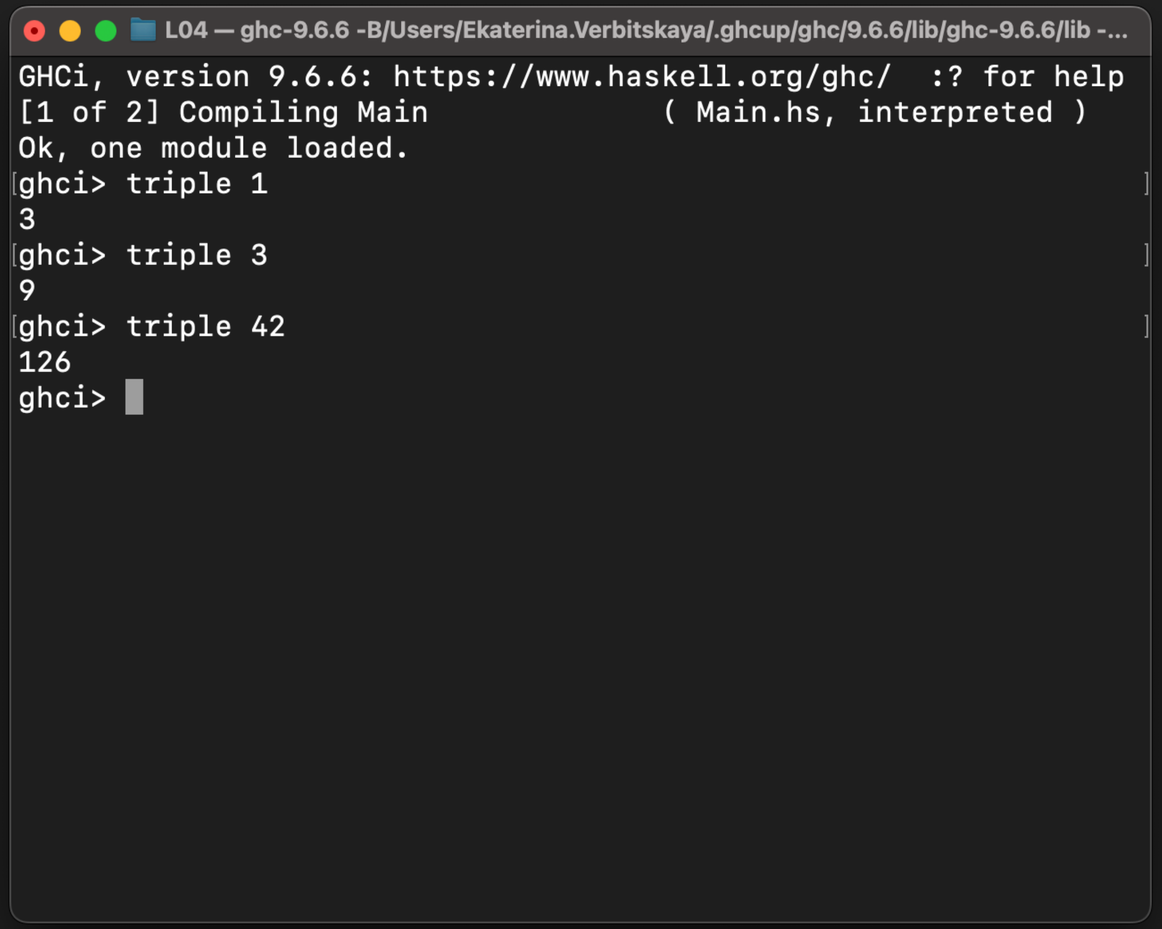
ETA-REDUCTION

- We can return functions as results from other function
- If the last argument of the function is only passed as the last argument of the function in the body, remove it
  - `\x → f x ≡ f`



EXERCISES

- Partially apply the function `mult` to implement function `triple` that multiplies a number by 3
- Show how the expression is evaluated:
  - ```
    let y = 13 in
    let x = triple (triple (1+y)) in
    mult x x
    ```

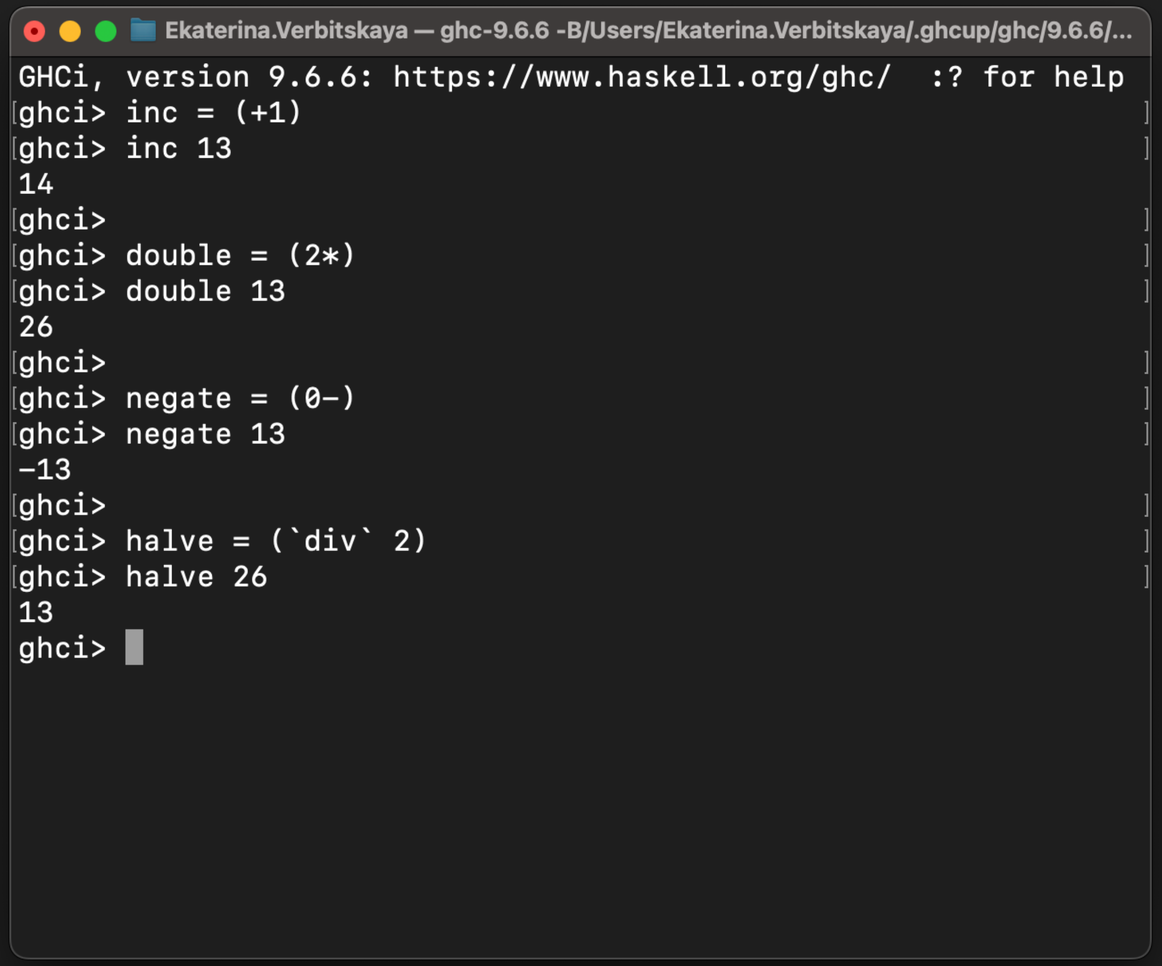

    ```

OUTLINE

- Function evaluation
- Operator sections
- Functions as arguments
- Higher-order functions on Lists

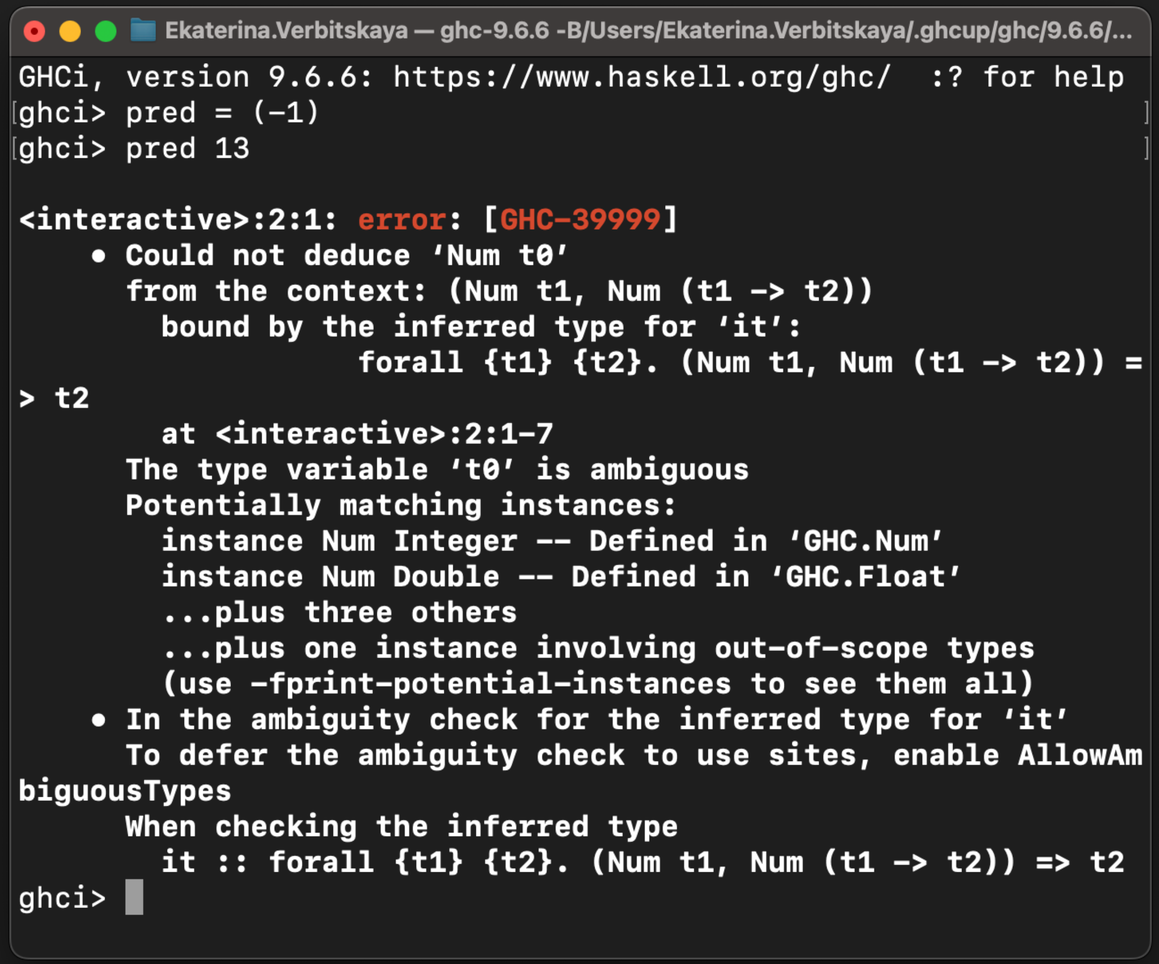
OPERATOR SECTIONS

- `(+1) ≡ \x → x + 1`
- `(2 *) ≡ \y → 2 * y`
- `(0 -) ≡ \y → 0 - y`
- `(`div` 2) ≡ \x → x `div` 2`
  `≡ \x → div x 2`



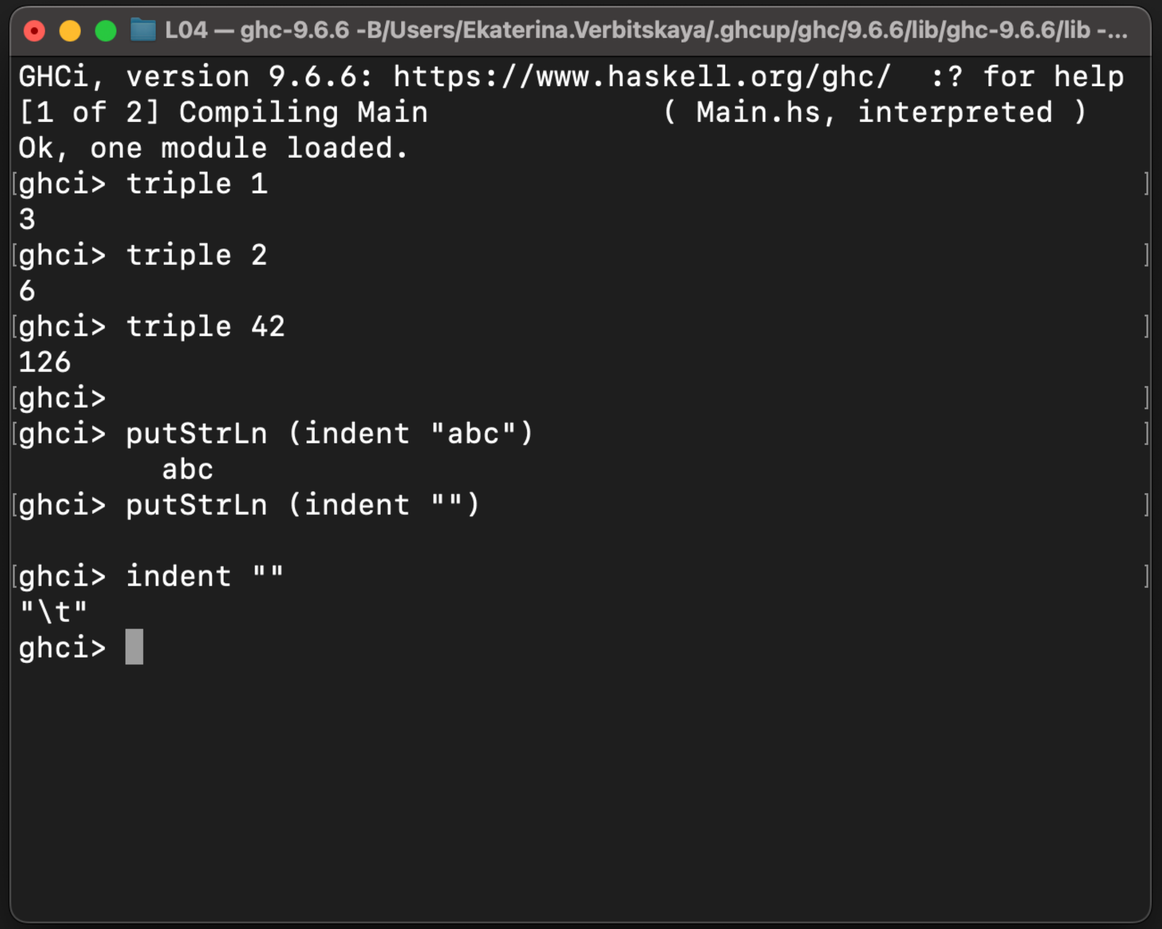
BEWARE OF (-1)

- Unary `-` messes with sections
- `(-n)` is a negation of `n`, not a section
- Use anonymous functions instead
  - `\x → x - 1`
  - `f x = x - 1`



EXERCISE

- Reimplement `triple` using operator sections
- Using operator sections, implement indenting function, that adds a `'\t'` in the beginning of a string

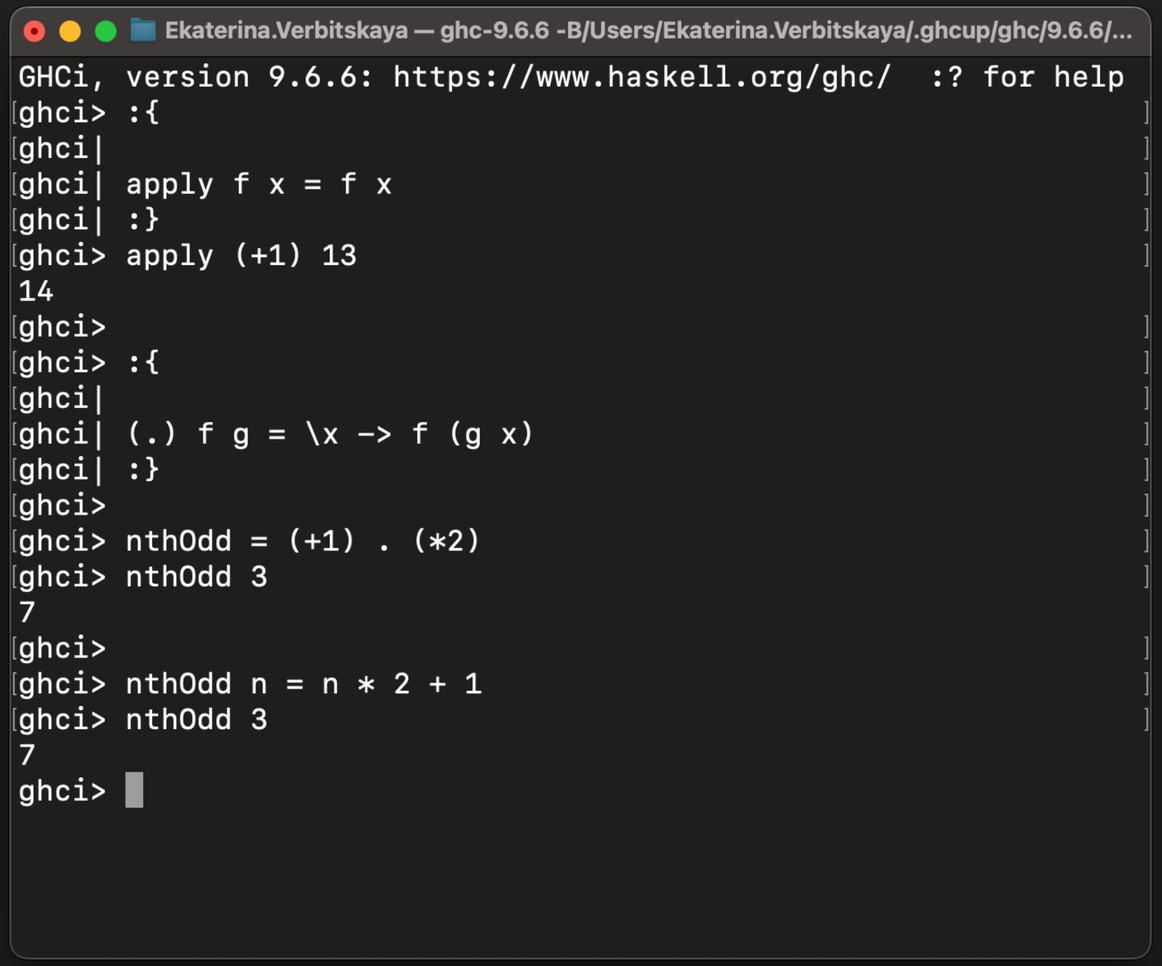


OUTLINE

- Function evaluation
- Operator sections
- Functions as arguments
- Higher-order functions on Lists

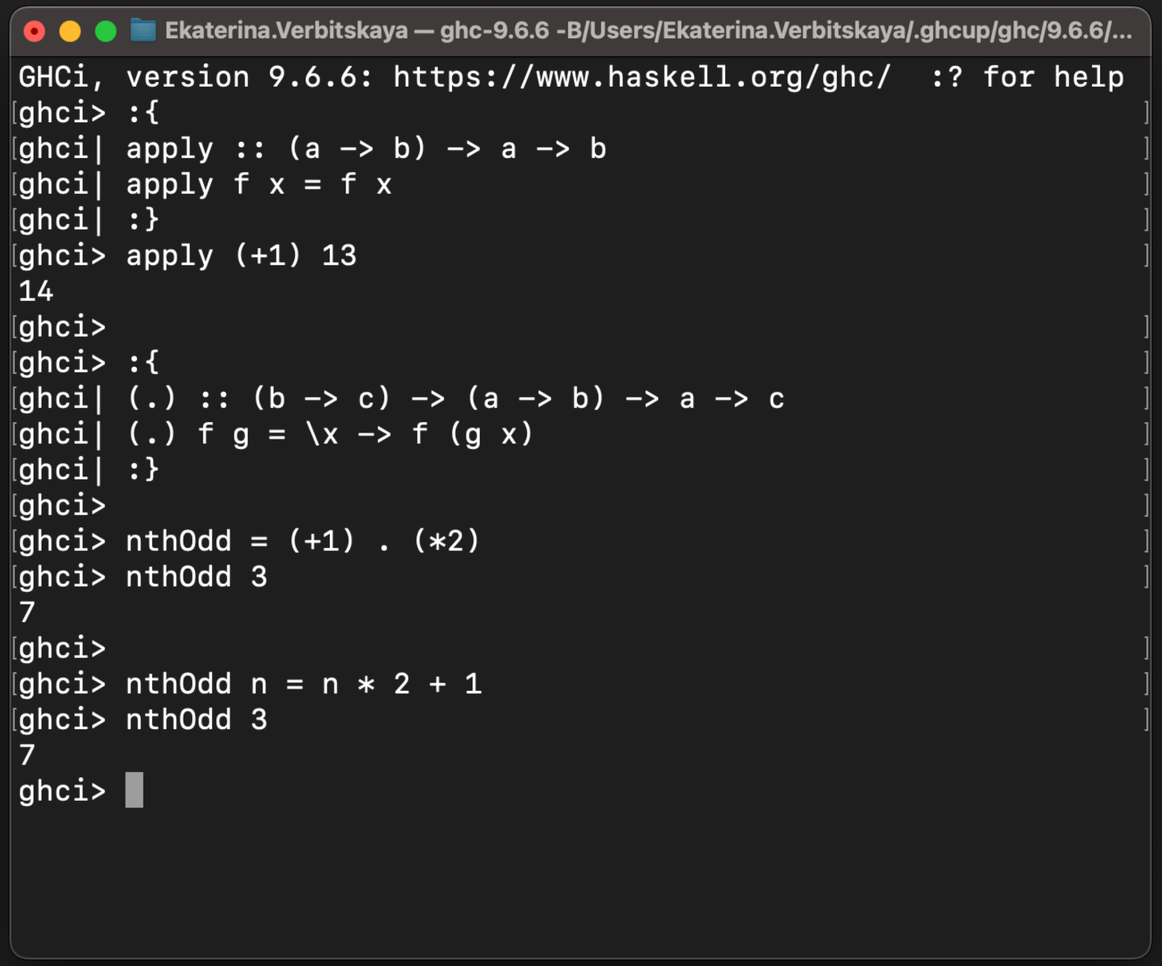
FUNCTIONS AS ARGUMENTS

- We can return functions, why not pass them as arguments?
- apply applies a function to its argument
  - a.k.a. `infixr 0 $`
- `infixr 9 (.)` is a function composition
  - pointfree style



FUNCTIONS AS ARGUMENTS

- We can return functions, why not pass them as arguments?
- apply applies a function to its argument
  - a.k.a. `infixr 0 $`
- `infixr 9 (.)` is a function composition
  - pointfree style

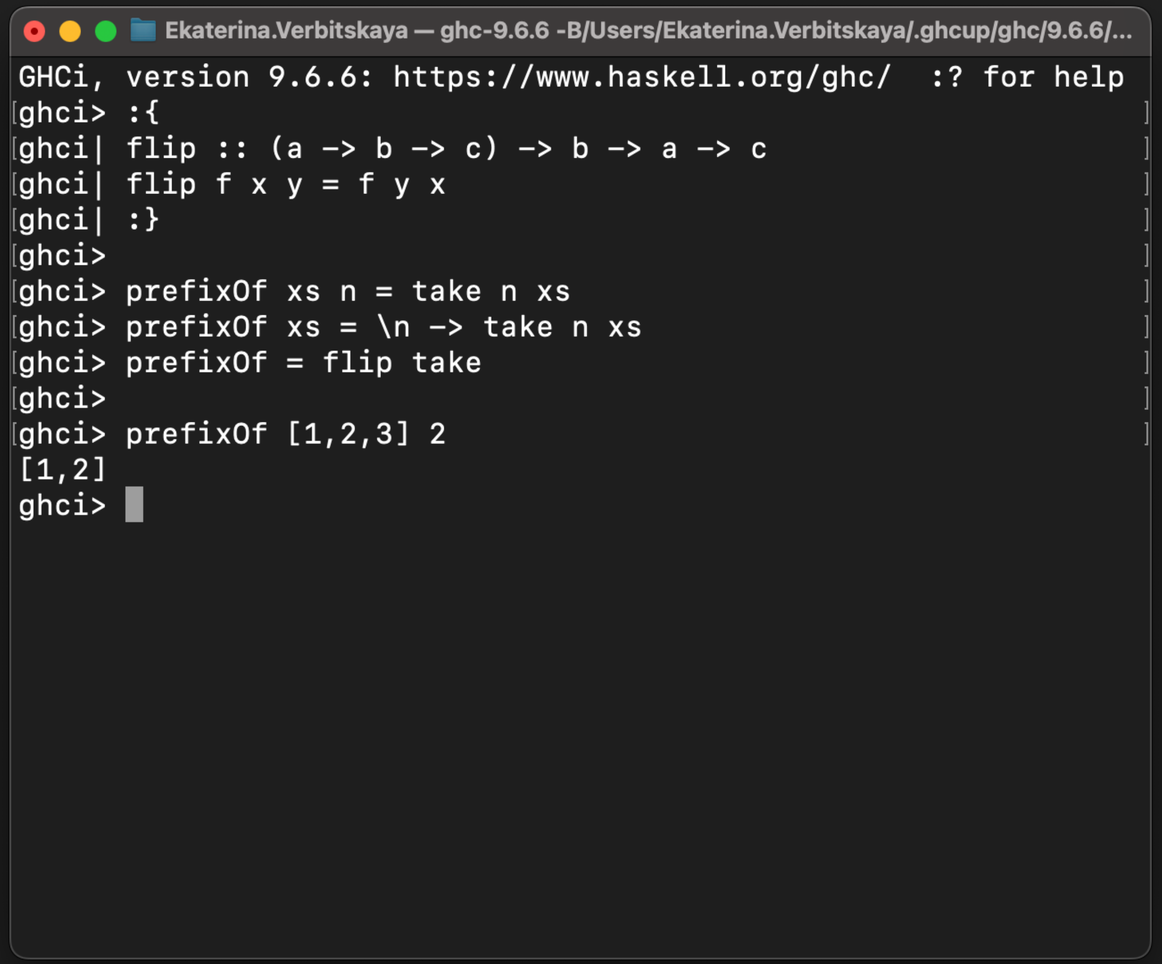


FLIP

- `flip` flips the order of the two arguments of a function

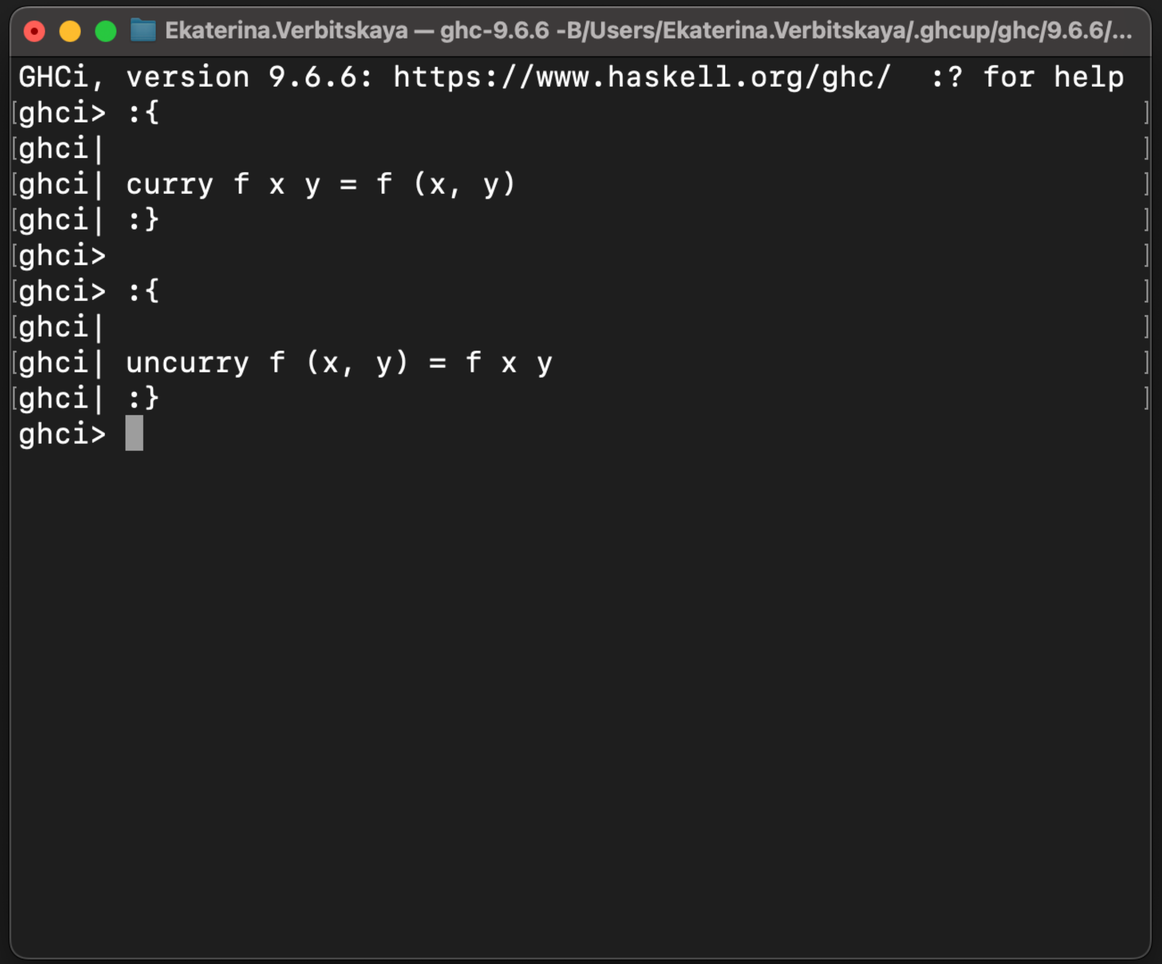
FLIP

- `flip` flips the order of the two arguments of a function
  - Helps avoid unnecessary lambda-functions



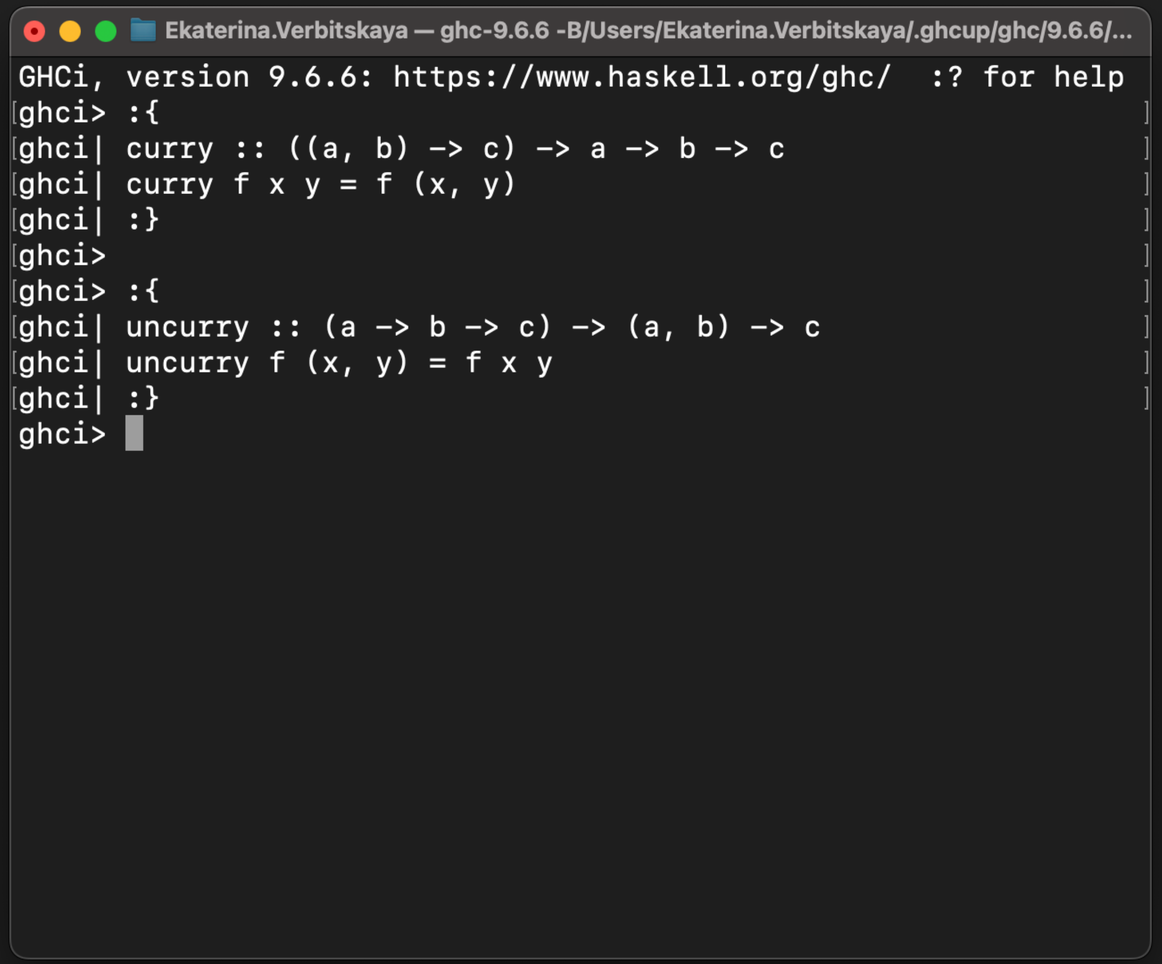
CURRY AND UNCURRY

- `curry` makes a function which takes a tuple into a function which may be partially applied
- `uncurry` makes a normal function with two arguments work on a tuple



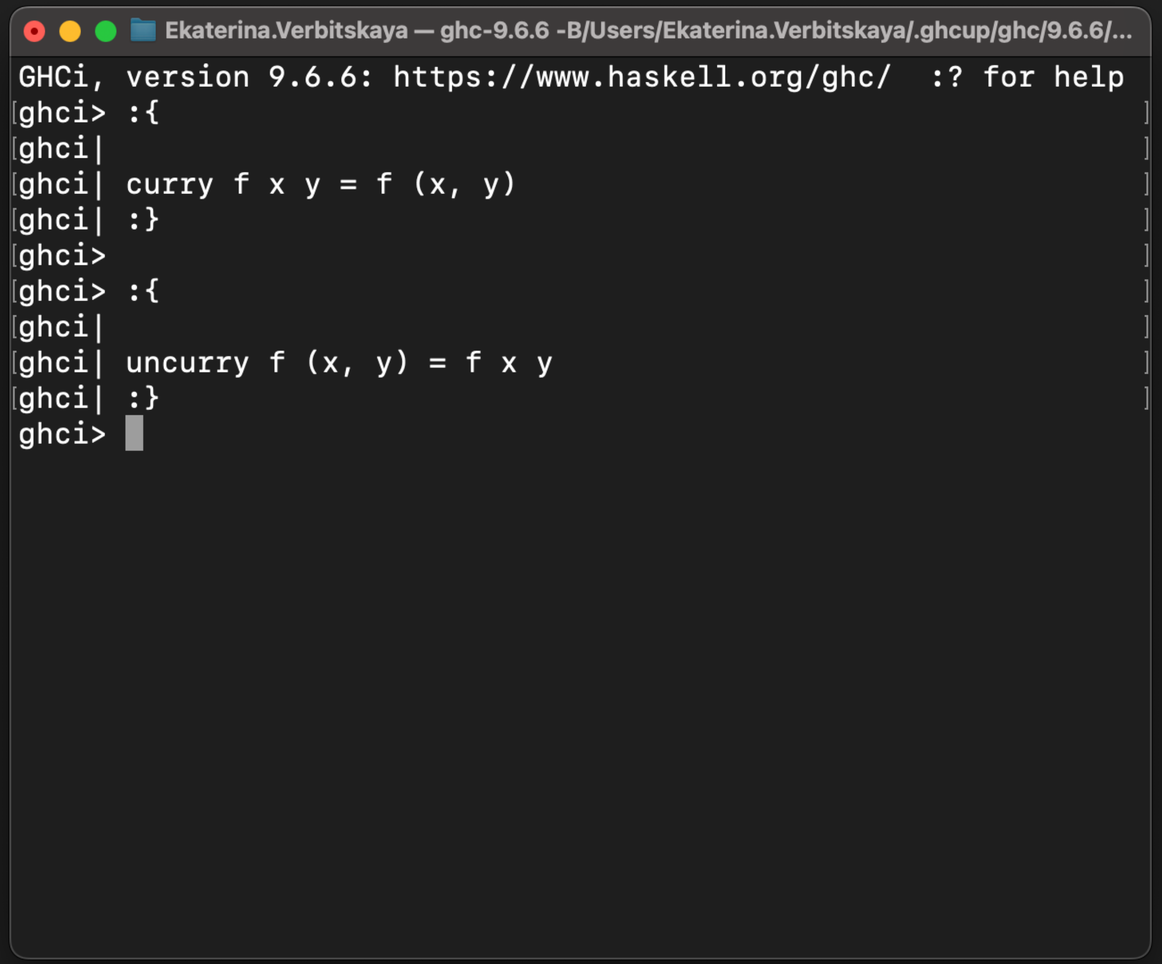
CURRY AND UNCURRY

- `curry` makes a function which takes a tuple into a function which may be partially applied
- `uncurry` makes a normal function with two arguments work on a tuple



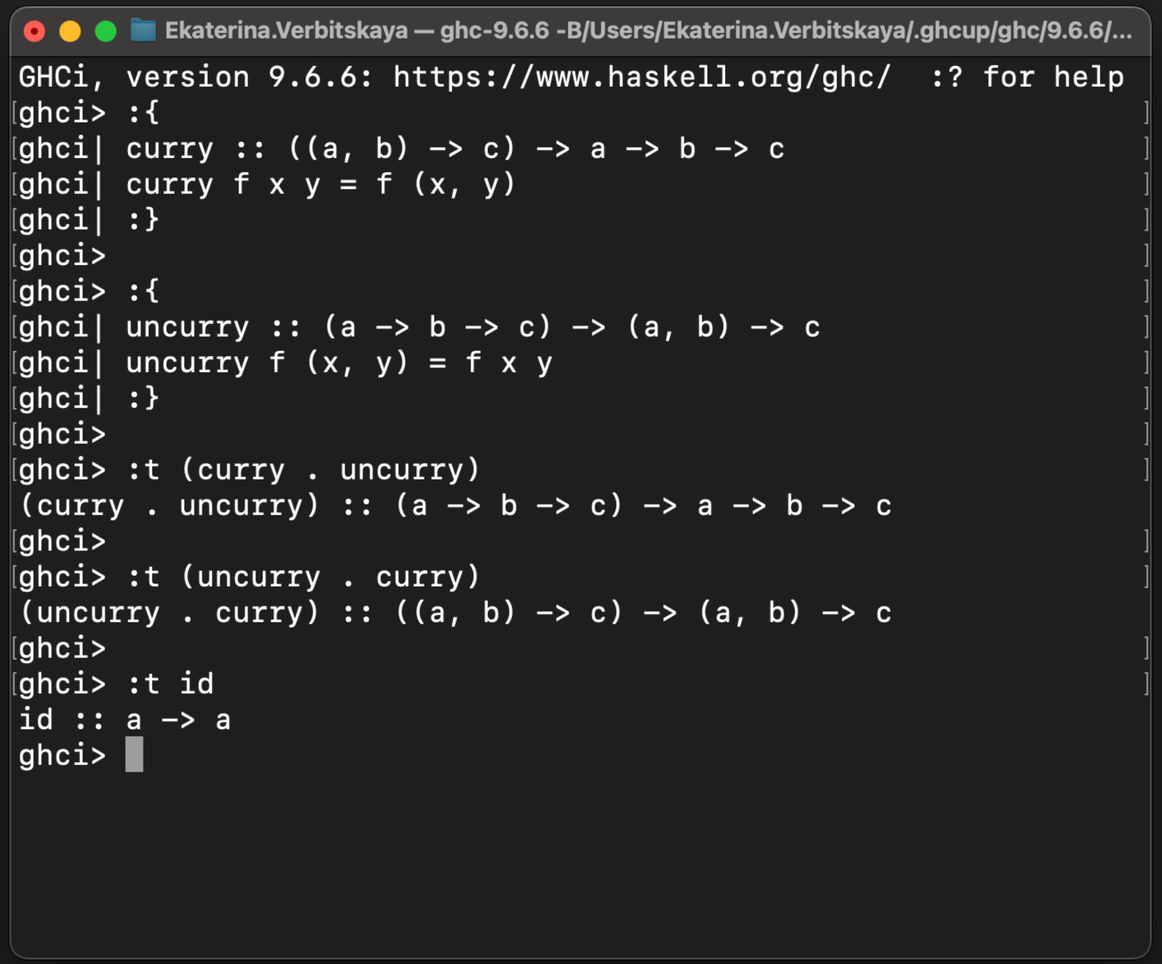
CURRY AND UNCURRY

- `curry` makes a function which takes a tuple into a function which may be partially applied
- `uncurry` makes a normal function with two arguments work on a tuple
- What do the following functions do?
  - `curry . uncurry`
  - `uncurry . curry`



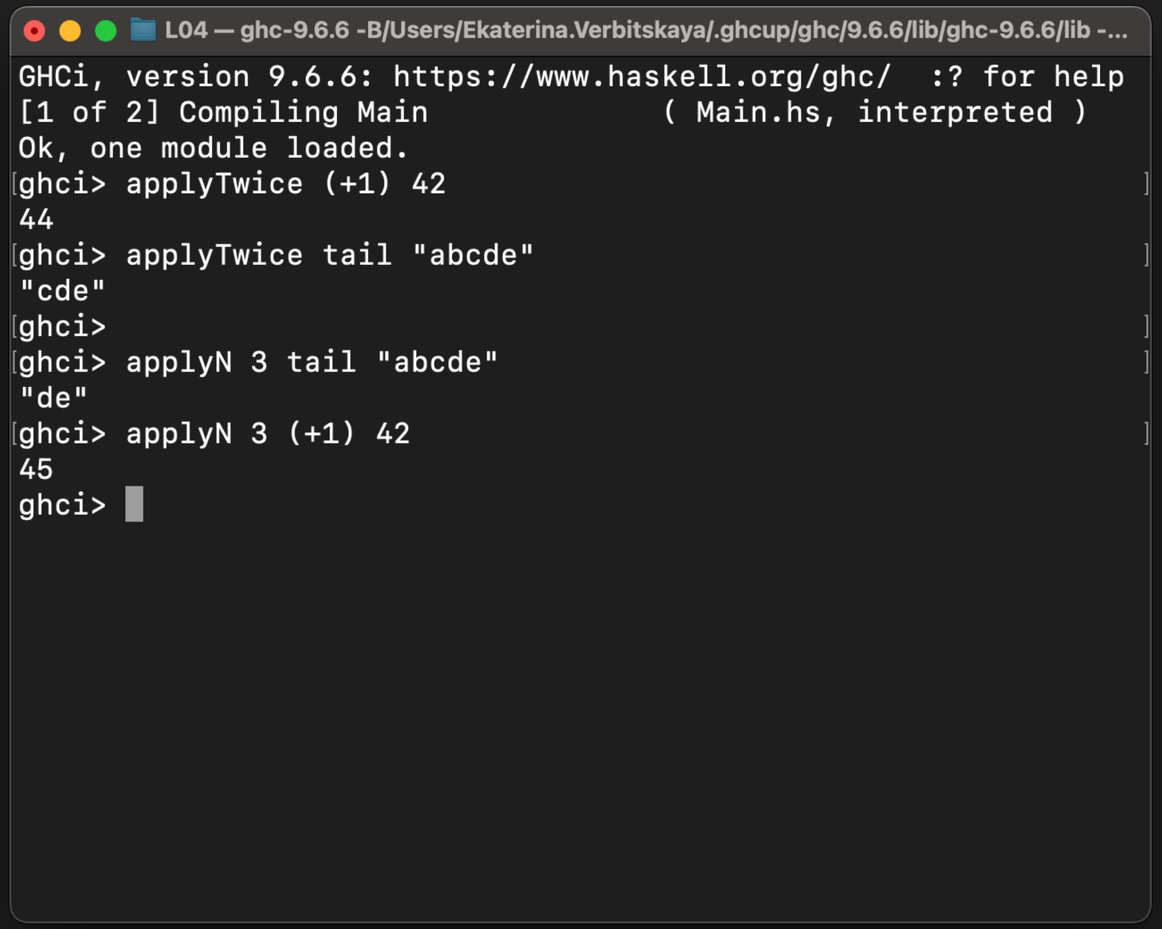
CURRY AND UNCURRY

- `curry` makes a function which takes a tuple into a function which may be partially applied
- `uncurry` makes a normal function with two arguments work on a tuple
- What do the following functions do?
  - `curry . uncurry`
  - `uncurry . curry`



EXERCISES

- Implement the function `applyTwice` which applies a function to the result of applying the function to its argument
  - What's its type?
- Implement the function `applyN` which applies a function to its argument n times
  - What's its type?

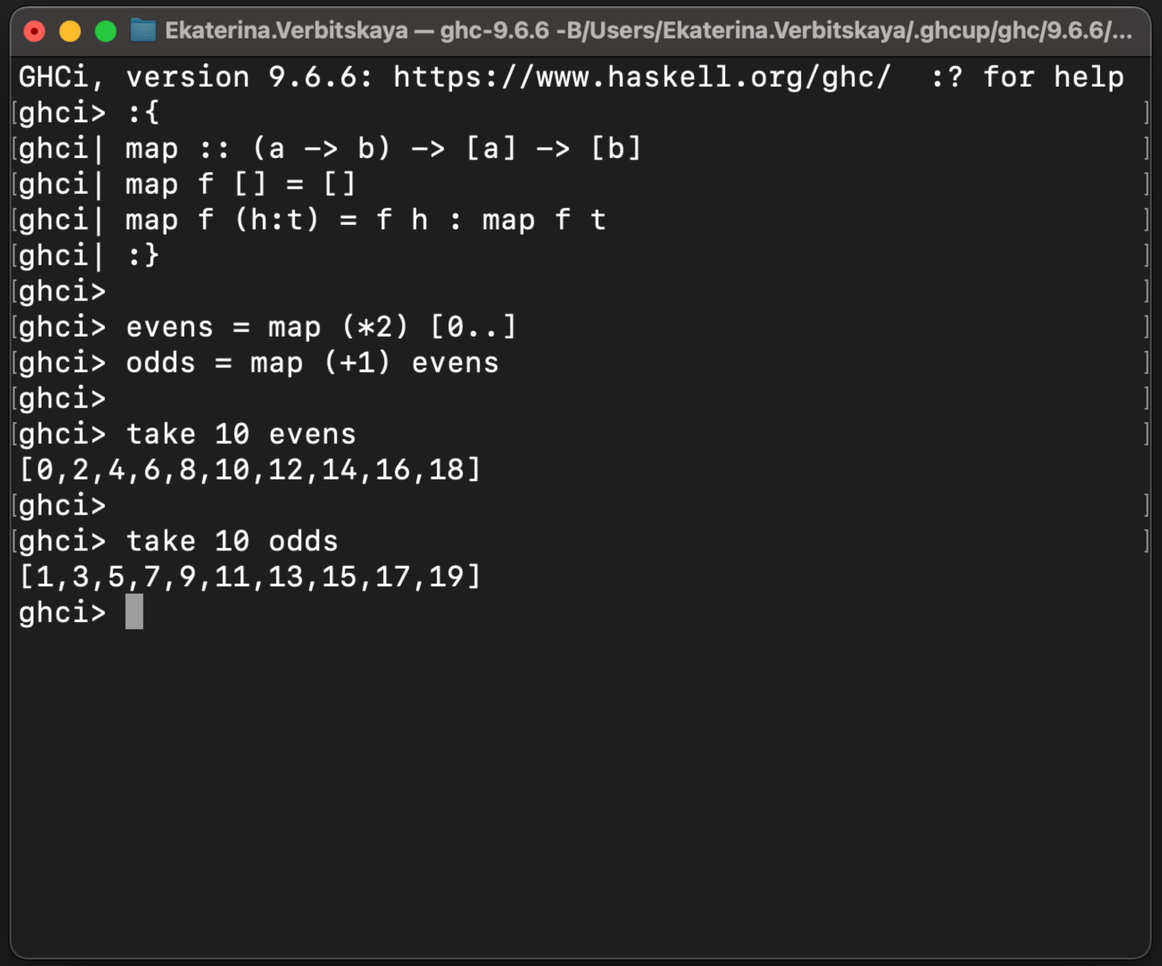


OUTLINE

- Function evaluation
- Operator sections
- Functions as arguments
- Higher-order functions on Lists

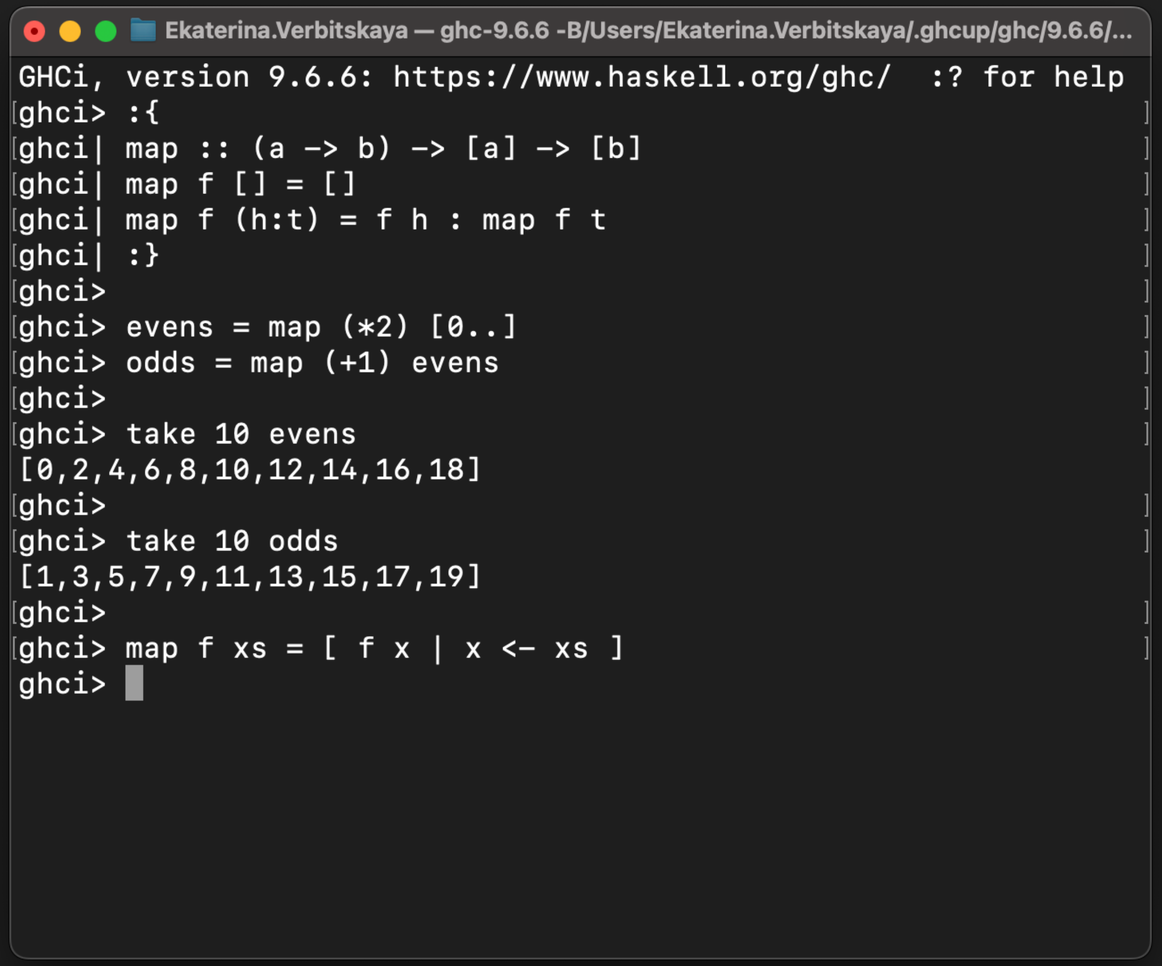
MAP

- `map` applies a function to every element of a list
- The length stays the same
- What does it remind you of?



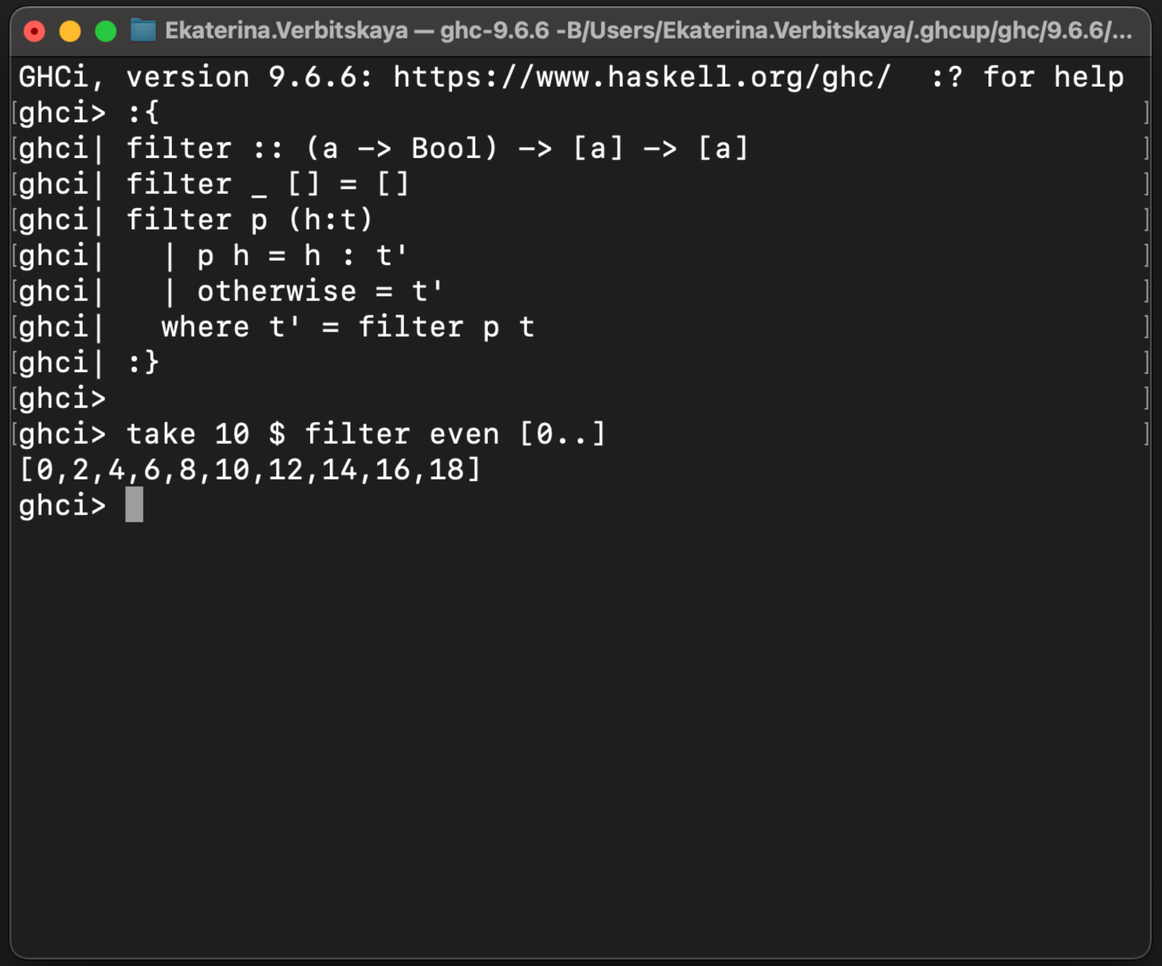
MAP

- `map` applies a function to every element of a list
- The length stays the same
- What does it remind you of?
  - Post-processing in list comprehensions



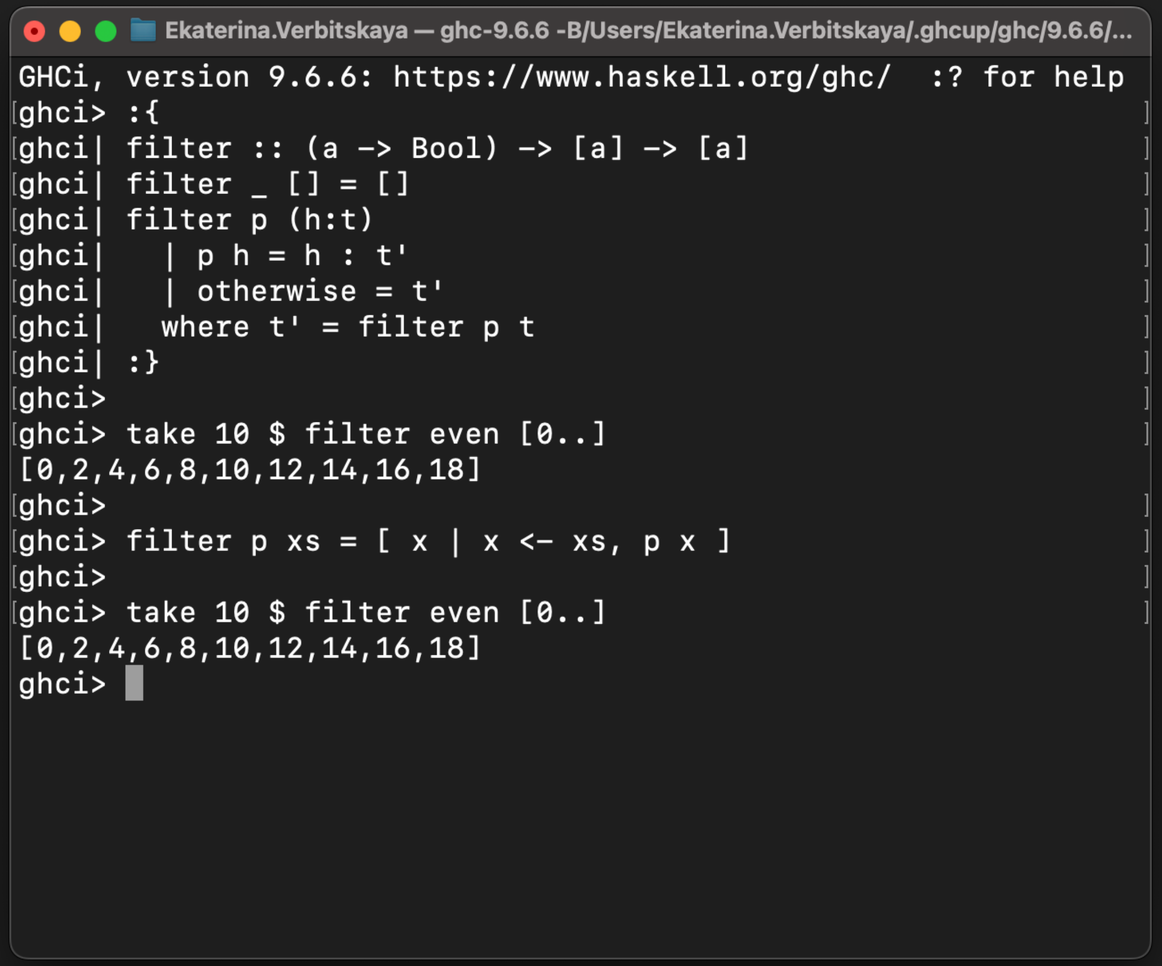
FILTER

- `filter p xs` leaves only the elements of the list `xs` for which the predicate `p` holds
- Changes the length
- What does it remind you of?



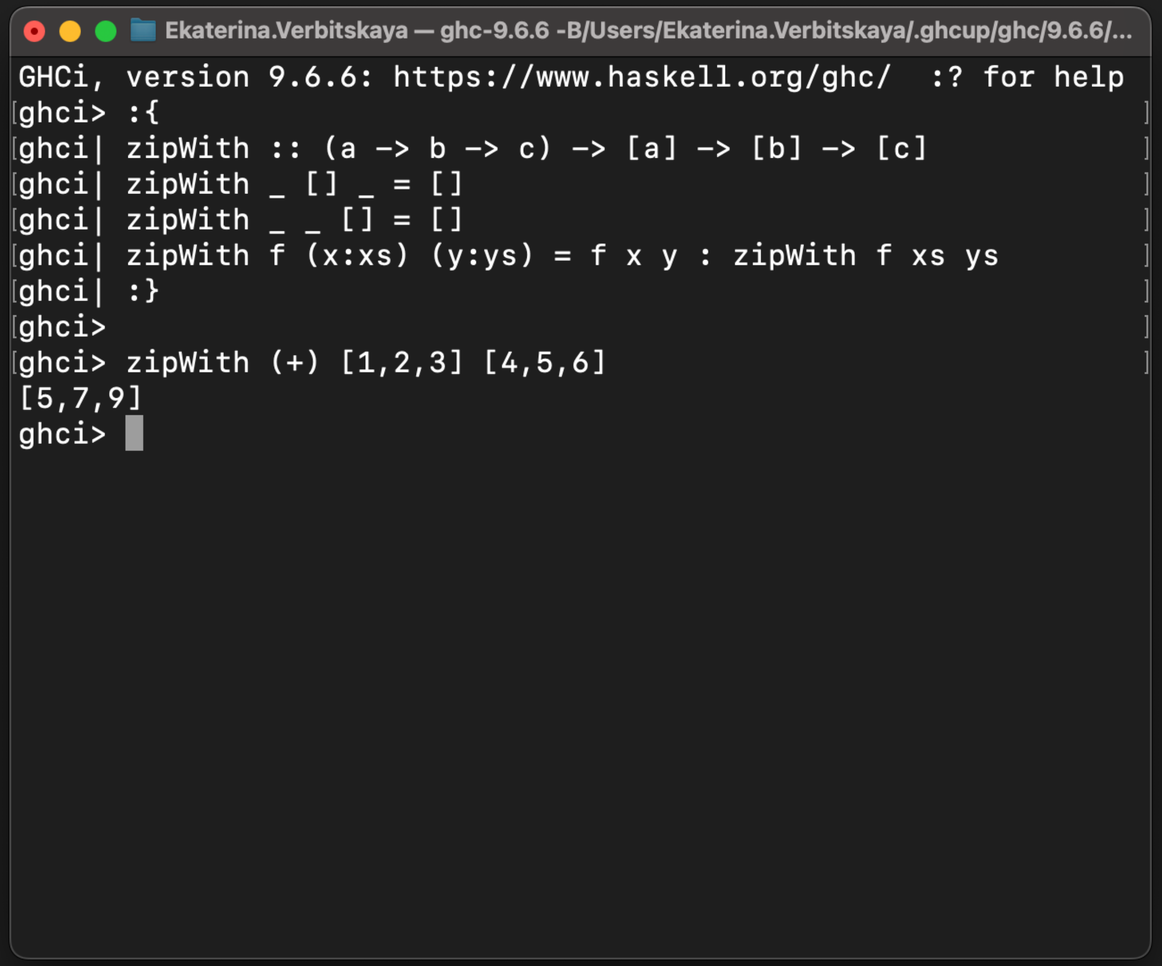
FILTER

- `filter p xs` leaves only the elements of the list `xs` for which the predicate `p` holds
- Changes the length
- What does it remind you of?
  - List comprehensions, again



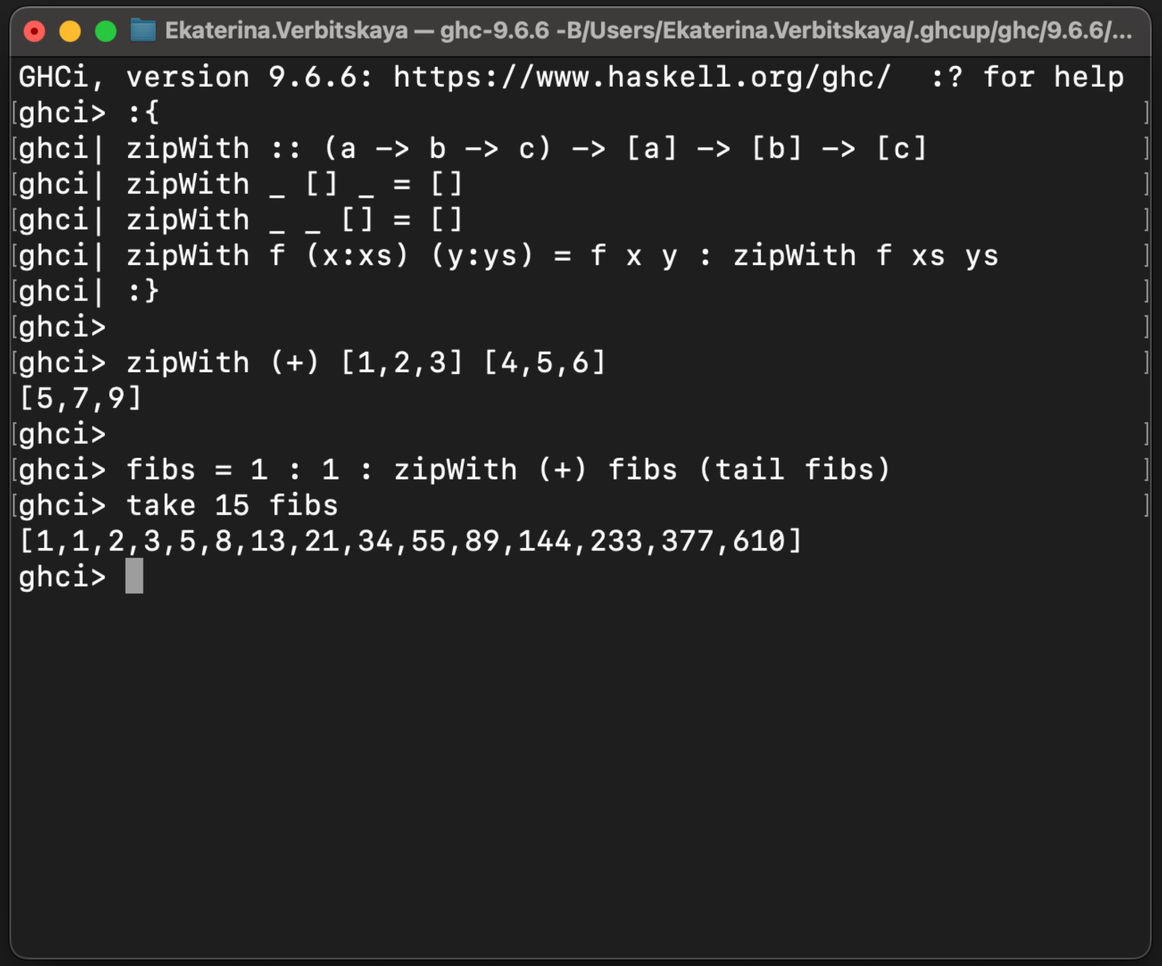
ZIPWITH

- `zipWith` works as a combination of `zip` and `map`
- Produces the list of the least length
- How to implement `fibs` with `zipWith`?



ZIPWITH

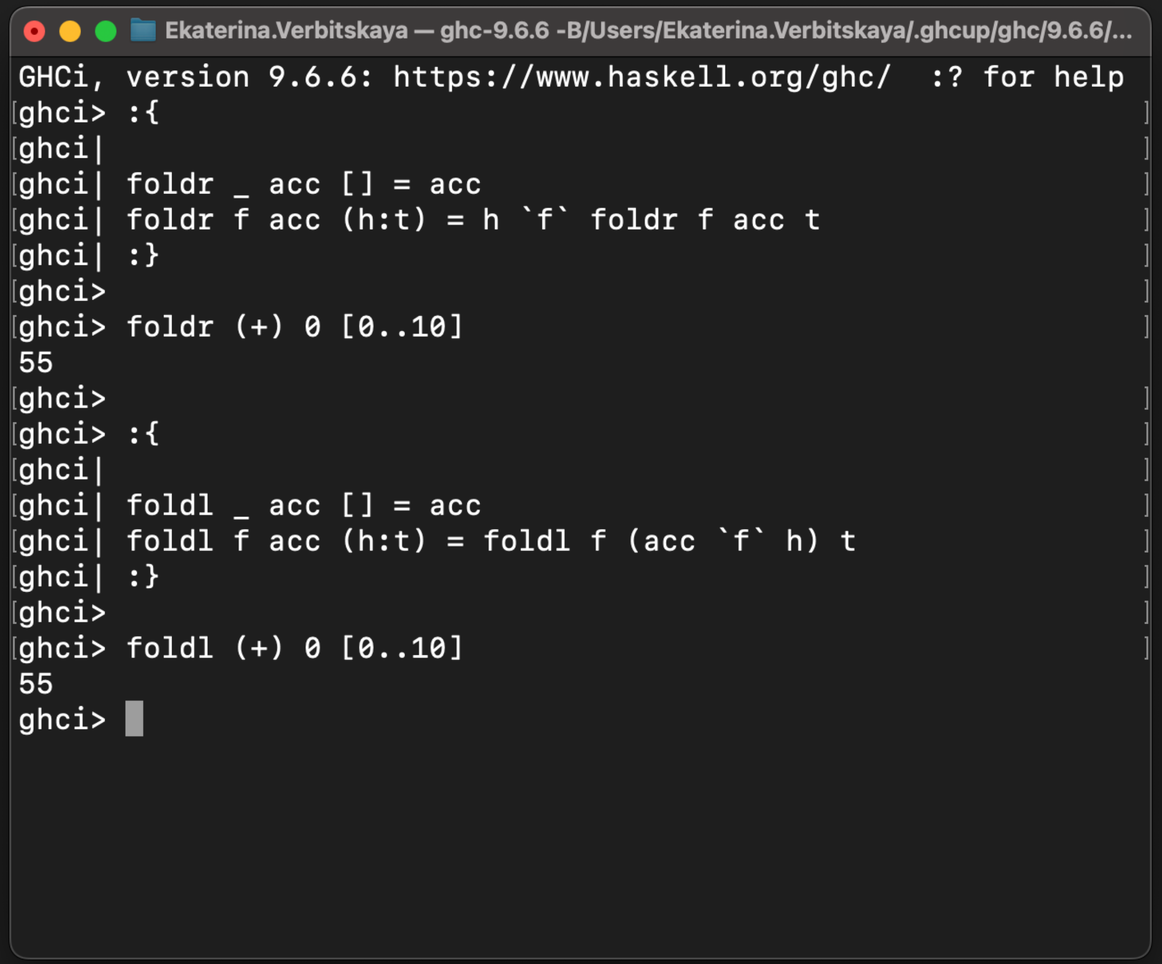
- `zipWith` works as a combination of `zip` and `map`
- Produces the list of the least length
- How to implement `fibs` with `zipWith`?



FOLD

- `foldr f acc [x, y, …, z]` ⟹
  `x `f` (y `f` … (z `f` acc)…)`

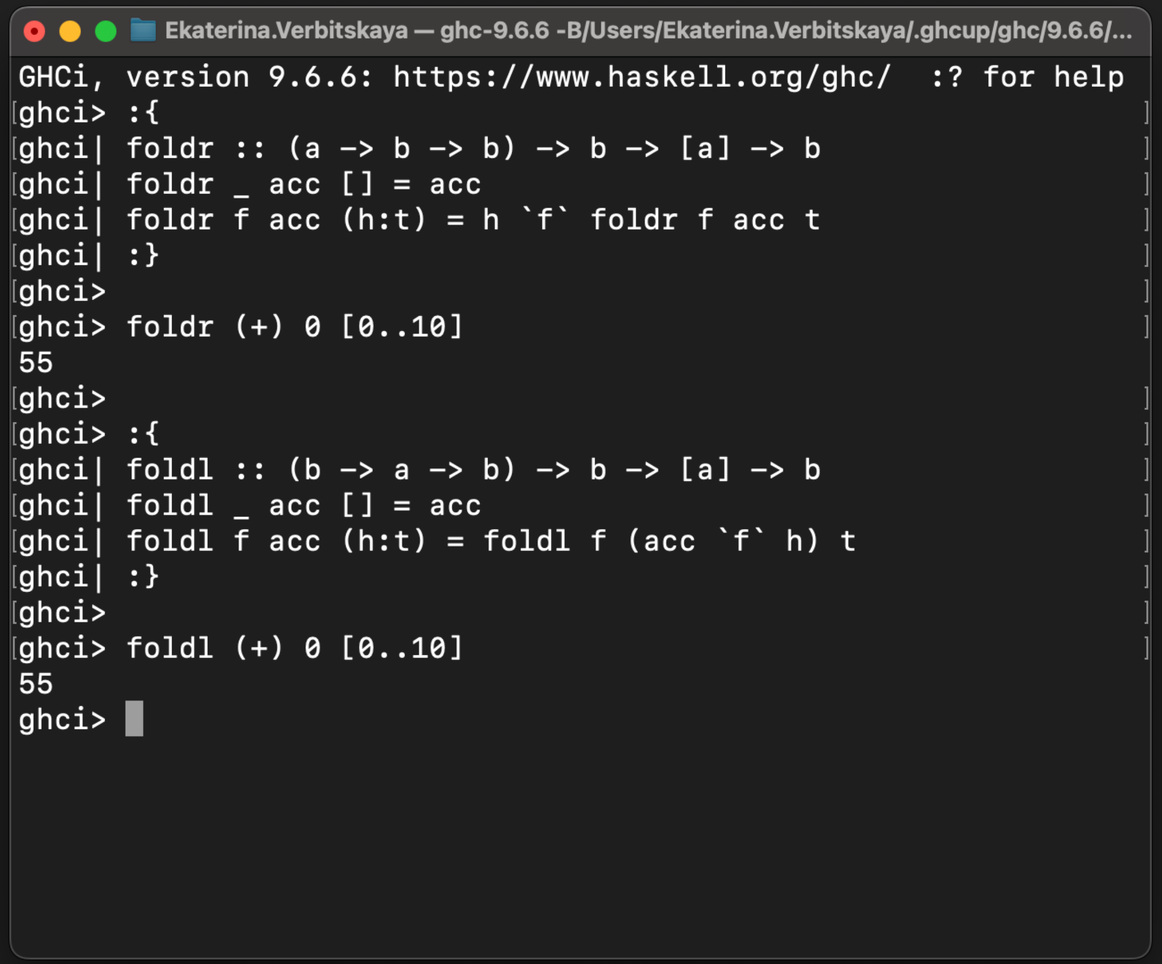
- `foldl f acc [x, y, …, z]` ⟹
  `(…((acc `f` x) `f` …) `f` z`



FOLD

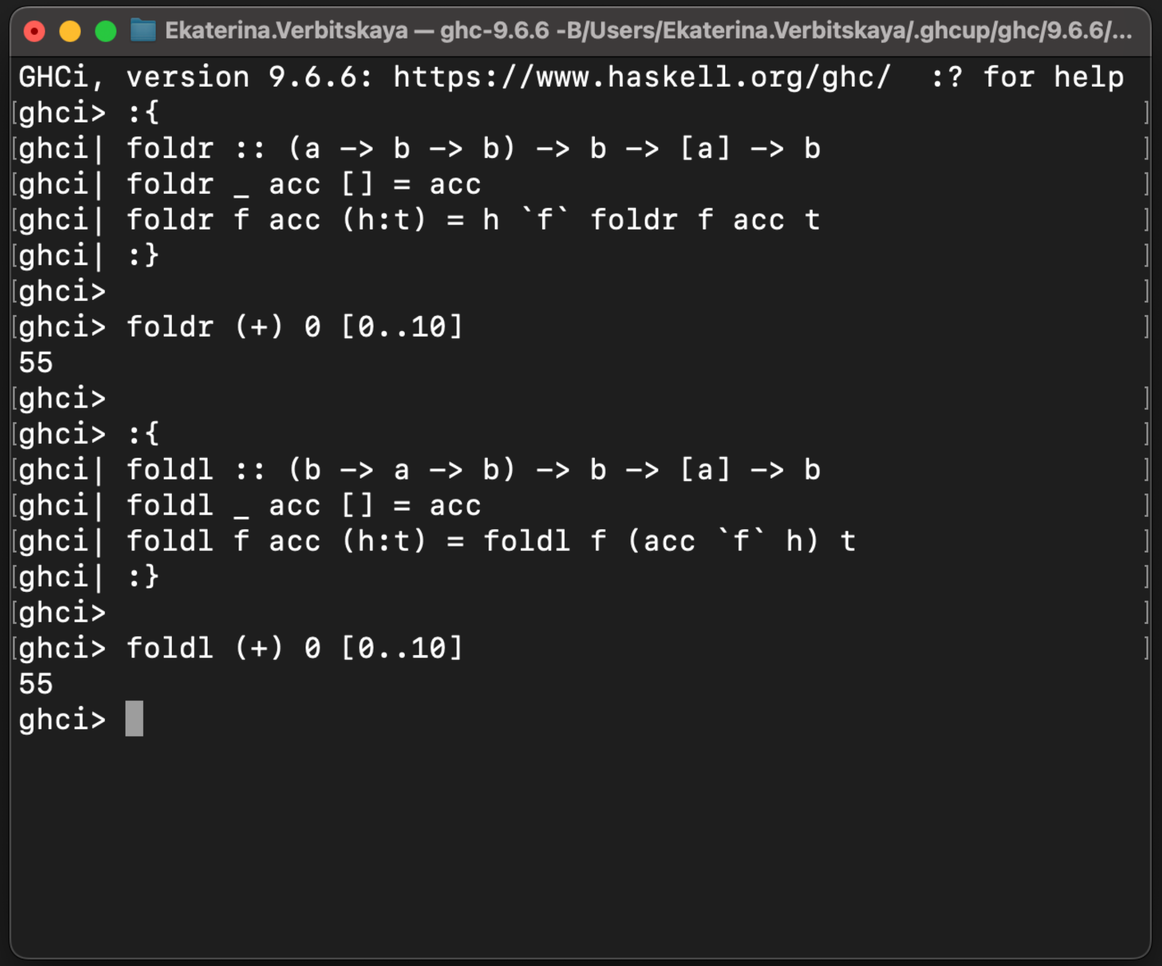
- `foldr f acc [x, y, …, z] ==`
  `x `f` (y `f` … (z `f` acc)…)`

- `foldl f acc [x, y, …, z] ==`
  `(…((acc `f` x) `f` …) `f` z`



FOLD

- `foldr f acc [x, y, …, z]` ≡≡≡ `x `f` (y `f` … (z `f` acc)…)`
- `foldl f acc [x, y, …, z]` ≡≡≡ `(…((acc `f` x) `f` …) `f` z`
- Which of the functions is safe to be applied to an infinite list?



EXERCISES

- Implement functions `sumList` and `prodList` using higher order functions