

INTRODUCTION TO

FUNCTIONAL PROGRAMMING

OUTLINE

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

EVALUATION IN ARITHMETICS

- ▶ Evaluate the innermost subexpression
 - ▶ $1 + 2 * 3 - 4$
- ▶ Replace it with the result
 - ▶ $(1 + (2 * 3)) - 4$
- ▶ Repeat until the expression is a number
 - ▶ $(1 + 6) - 4$
 - ▶ $7 - 4$
 - ▶ 3

EVALUATION IN ARITHMETICS: VARIABLES

- ▶ Evaluate the innermost subexpression
 - ▶ let $x = 4 - 2$ in let $y = 4$ in $1 + x * 3 - y$
- ▶ If it contains variables, find their definition and evaluate it
 - ▶ 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
- ▶ Replace the subexpression with the result
- ▶ Repeat until the expression is a number

EVALUATION OF A FUNCTION

- ▶ When a function is applied to a value, replace its argument in the body with the value
 - ▶ inc $i = i + 1$
 - ▶ inc 2 $\Rightarrow 2 + 1 \Rightarrow 3$
- ▶ Evaluate the body like an expression
- ▶ Repeat until the body becomes a value
 - ▶ double $x = x + x$
 - ▶ double 3 $\Rightarrow 3 + 3 \Rightarrow 6$

WHAT IF THERE ARE 2 ARGUMENTS?

- ▶ Pretty much the same thing happens:
- ▶ Arguments are replaced with the values provided
- ▶ Body is evaluated
- ▶ $\text{plus } x \ y = x + y$
- ▶ $\text{plus } 1 \ 2 => 1 + 2 => 3$

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup...  
ghci> plus x y = x + y  
ghci> plus 1 2  
3  
ghci>  
ghci> plus = \x y -> x + y  
ghci> plus 1 2  
3  
ghci> █
```

CARRYING

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup... ]  
ghci> plus x y = x + y  
ghci> plus 1 2  
3  
ghci>  
ghci> plus = \x y -> x + y  
ghci> plus 1 2  
3  
ghci>  
ghci> plus x = \y -> x + y  
ghci> plus 1 2  
3  
ghci>  
ghci> plus = \x -> \y -> x + y  
ghci> plus 1 2  
3  
ghci>  
ghci> plus = \x -> (\y -> (x + y))  
ghci> plus 1 2  
3  
ghci> █
```

EVALUATION OF FUNCTIONS

- ▶ $f\ x\ y\ ..\ z = \text{body} \equiv$
 $f = \lambda x \rightarrow \lambda y \rightarrow .. \lambda z \rightarrow \text{body}$
- ▶ Evaluation of a function with multiple arguments is in fact evaluation of several functions with a single argument
- ▶ $\text{plus} = \lambda x \rightarrow \lambda y \rightarrow x + y$
- ▶ $\text{plus}\ 1\ 2 \Rightarrow$
- ▶ $(\text{plus}\ 1)\ 2 \Rightarrow$
- ▶ $((\lambda x \rightarrow \lambda y \rightarrow x + y)\ 1)\ 2 \Rightarrow$
- ▶ $(\lambda y \rightarrow 1 + y)\ 2 \Rightarrow$
- ▶ $1 + 2 \Rightarrow$
- ▶ 3

PARTIAL APPLICATION

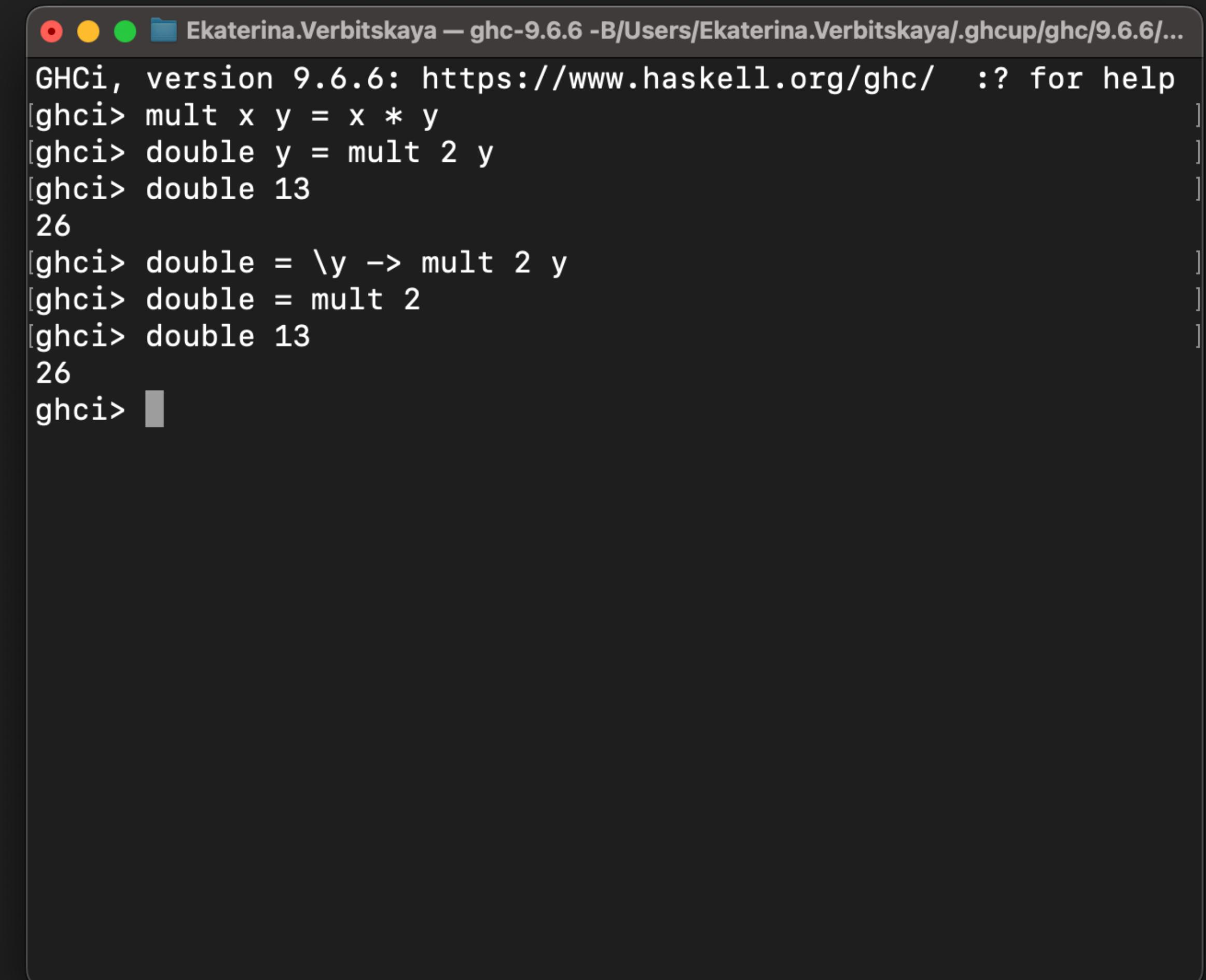
- ▶ What happens if we don't provide enough values for all arguments of a function?

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
- ▶ $\lambda x \rightarrow f x \equiv f$



A screenshot of a terminal window titled "Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6...". The window shows a Haskell GHCi session. The user defines a function "mult" that takes two arguments and returns their product. Then, they define a function "double" that takes one argument and applies "mult" with 2 as the second argument. When "double 13" is evaluated, it prints 26. Finally, the user defines "double" as a lambda expression that takes one argument "y" and applies "mult" with 2 as the second argument. When "double 13" is evaluated again, it prints 26.

```
GHCi, version 9.6.6: https://www.haskell.org/ghc/ :? for help
[ghci] mult x y = x * y
[ghci] double y = mult 2 y
[ghci] double 13
26
[ghci] double = \y -> mult 2 y
[ghci] double = mult 2
[ghci] double 13
26
ghci>
```

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`

```
L04 — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib -...  
GHCi, version 9.6.6: https://www.haskell.org/ghc/ :? for help  
[1 of 2] Compiling Main           ( Main.hs, interpreted )  
Ok, one module loaded.  
[ghci]> triple 1  
3  
[ghci]> triple 3  
9  
[ghci]> triple 42  
126  
ghci> █
```

OUTLINE

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

OPERATOR SECTIONS

- ▶ $(+1) \equiv \lambda x \rightarrow x + 1$
- ▶ $(2 *) \equiv \lambda y \rightarrow 2 * y$
- ▶ $(0 -) \equiv \lambda y \rightarrow 0 - y$
- ▶ $(`div` 2) \equiv \lambda x \rightarrow x `div` 2$
 $\equiv \lambda x \rightarrow div\ x\ 2$

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> inc = (+1)
[ghci]> inc 13
14
[ghci]>
[ghci]> double = (2*)
[ghci]> double 13
26
[ghci]>
[ghci]> negate = (0-)
[ghci]> negate 13
-13
[ghci]>
[ghci]> halve = (`div` 2)
[ghci]> halve 26
13
ghci>
```

BEWARE OF (-1)

- ▶ Unary - messes with sections
- ▶ $(-n)$ is a negation of n , not a section
- ▶ Use anonymous functions instead

- ▶ $\lambda x \rightarrow x - 1$
- ▶ $f x = x - 1$

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci] pred = (-1)
[ghci] pred 13
<interactive>:2:1: error: [GHC-3999]
  • Could not deduce 'Num t0'
    from the context: (Num t1, Num (t1 -> t2))
      bound by the inferred type for 'it':
                  forall {t1} {t2}. (Num t1, Num (t1 -> t2)) =
> t2
      at <interactive>:2:1-7
The type variable 't0' is ambiguous
Potentially matching instances:
  instance Num Integer -- Defined in 'GHC.Num'
  instance Num Double -- Defined in 'GHC.Float'
  ...plus three others
  ...plus one instance involving out-of-scope types
    (use -fprint-potential-instances to see them all)
  • In the ambiguity check for the inferred type for 'it'
    To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
    When checking the inferred type
      it :: forall {t1} {t2}. (Num t1, Num (t1 -> t2)) => t2
ghci>
```

OPERATOR SECTIONS

EXERCISE

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

```
L04 — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib -...
GHCi, version 9.6.6: https://www.haskell.org/ghc/ :? for help
[1 of 2] Compiling Main                         ( Main.hs, interpreted )
Ok, one module loaded.
[ghci]> triple 1
3
[ghci]> triple 2
6
[ghci]> triple 42
126
[ghci]>
[ghci]> putStrLn (indent "abc")
      abc
[ghci]> putStrLn (indent "")
[ghci]> indent ""
"\t"
ghci> █
```

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci]
[ghci| apply f x = f x
[ghci| :}
[ghci]> apply (+1) 13
14
[ghci]>
[ghci]> :{
[ghci]
[ghci| (.) f g = \x -> f (g x)
[ghci| :}
[ghci]>
[ghci]> nthOdd = (+1) . (*2)
[ghci]> nthOdd 3
7
[ghci]>
[ghci]> nthOdd n = n * 2 + 1
[ghci]> nthOdd 3
7
ghci> █
```

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| apply :: (a -> b) -> a -> b
[ghci| apply f x = f x
[ghci| :}
[ghci]> apply (+1) 13
14
[ghci]>
[ghci]> :{
[ghci| (.) :: (b -> c) -> (a -> b) -> a -> c
[ghci| (.) f g = \x -> f (g x)
[ghci| :}
[ghci]>
[ghci]> nthOdd = (+1) . (*2)
[ghci]> nthOdd 3
7
[ghci]>
[ghci]> nthOdd n = n * 2 + 1
[ghci]> nthOdd 3
7
ghci> █
```

FUNCTIONS AS ARGUMENTS

FLIP

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci]
[ghci| flip f x y = f y x
[ghci| :}
[ghci]>
[ghci]> prefixOf xs n = take n xs
[ghci]> prefixOf xs = \n -> take n xs
[ghci]> prefixOf = flip take
[ghci]>
[ghci]> prefixOf [1,2,3] 2
[1,2]
ghci> █
```

FUNCTIONS AS ARGUMENTS

FLIP

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| flip :: (a -> b -> c) -> b -> a -> c
[ghci| flip f x y = f y x
[ghci| :}
[ghci]>
[ghci]> prefixOf xs n = take n xs
[ghci]> prefixOf xs = \n -> take n xs
[ghci]> prefixOf = flip take
[ghci]>
[ghci]> prefixOf [1,2,3] 2
[1,2]
ghci> █
```

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci]
[ghci| curry f x y = f (x, y)
[ghci| :}
[ghci]>
[ghci]> :{
[ghci]
[ghci| uncurry f (x, y) = f x y
[ghci| :}
ghci> █
```

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| curry :: ((a, b) -> c) -> a -> b -> c
[ghci| curry f x y = f (x, y)
[ghci| :}
[ghci]>
[ghci]> :{
[ghci| uncurry :: (a -> b -> c) -> (a, b) -> c
[ghci| uncurry f (x, y) = f x y
[ghci| :}
ghci> █
```

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**

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci]
[ghci| curry f x y = f (x, y)
[ghci| :}
[ghci]>
[ghci]> :{
[ghci]
[ghci| uncurry f (x, y) = f x y
[ghci| :}
ghci> █
```

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**

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  ?: for help
[ghci]> :{
[ghci| curry :: ((a, b) -> c) -> a -> b -> c
[ghci| curry f x y = f (x, y)
[ghci| :}
[ghci]>
[ghci]> :{
[ghci| uncurry :: (a -> b -> c) -> (a, b) -> c
[ghci| uncurry f (x, y) = f x y
[ghci| :}
[ghci]>
[ghci]> :t (curry . uncurry)
(curry . uncurry) :: (a -> b -> c) -> a -> b -> c
[ghci]>
[ghci]> :t (uncurry . curry)
(uncurry . curry) :: ((a, b) -> c) -> (a, b) -> c
[ghci]>
[ghci]> :t id
id :: a -> a
ghci> █
```

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?

```
L04 — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib -...  
GHCi, version 9.6.6: https://www.haskell.org/ghc/ :? for help  
[1 of 2] Compiling Main           ( Main.hs, interpreted )  
Ok, one module loaded.  
[ghci]> applyTwice (+1) 42  
44  
[ghci]> applyTwice tail "abcde"  
"cde"  
[ghci]>  
[ghci]> applyN 3 tail "abcde"  
"de"  
[ghci]> applyN 3 (+1) 42  
45  
ghci>
```

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?

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| map :: (a -> b) -> [a] -> [b]
[ghci| map f [] = []
[ghci| map f (h:t) = f h : map f t
[ghci| :}
[ghci]>
[ghci]> evens = map (*2) [0..]
[ghci]> odds = map (+1) evens
[ghci]>
[ghci]> take 10 evens
[0,2,4,6,8,10,12,14,16,18]
[ghci]>
[ghci]> take 10 odds
[1,3,5,7,9,11,13,15,17,19]
ghci>
```

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| map :: (a -> b) -> [a] -> [b]
[ghci| map f [] = []
[ghci| map f (h:t) = f h : map f t
[ghci| :}
[ghci]>
[ghci]> evens = map (*2) [0..]
[ghci]> odds = map (+1) evens
[ghci]>
[ghci]> take 10 evens
[0,2,4,6,8,10,12,14,16,18]
[ghci]>
[ghci]> take 10 odds
[1,3,5,7,9,11,13,15,17,19]
[ghci]>
[ghci]> map f xs = [ f x | x <- xs ]
ghci> █
```

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?

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| filter :: (a -> Bool) -> [a] -> [a]
[ghci| filter _ [] = []
[ghci| filter p (h:t)
[ghci|   | p h = h : t'
[ghci|   | otherwise = t'
[ghci|     where t' = filter p t
[ghci|   }
[ghci]>
[ghci]> take 10 $ filter even [0..]
[0,2,4,6,8,10,12,14,16,18]
ghci> █
```

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| filter :: (a -> Bool) -> [a] -> [a]
[ghci| filter _ [] = []
[ghci| filter p (h:t)
[ghci|   | p h = h : t'
[ghci|   | otherwise = t'
[ghci|     where t' = filter p t
[ghci|   }
[ghci]>
[ghci]> take 10 $ filter even [0..]
[0,2,4,6,8,10,12,14,16,18]
[ghci]>
[ghci]> filter p xs = [ x | x <- xs, p x ]
[ghci]>
[ghci]> take 10 $ filter even [0..]
[0,2,4,6,8,10,12,14,16,18]
ghci> █
```

ZIPWITH

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
[ghci| zipWith _ [] _ = []
[ghci| zipWith _ _ [] = []
[ghci| zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
[ghci| :}
[ghci]>
[ghci]> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
ghci>
```

ZIPWITH

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

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
[ghci| zipWith _ [] _ = []
[ghci| zipWith _ _ [] = []
[ghci| zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
[ghci| :}
[ghci]>
[ghci]> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
[ghci]>
[ghci]> fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
[ghci]> take 15 fibs
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
ghci>
```

HIGHER-ORDER FUNCTIONS ON LISTS

FOLD

► $\text{foldr } f \text{ acc } [x, y, \dots, z] \equiv$
 $x `f` (y `f` \dots (z `f` acc) \dots)$

► $\text{foldl } f \text{ acc } [x, y, \dots, z] \equiv$
 $(\dots((acc `f` x) `f` \dots) `f` z$

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci]
[ghci| foldr _ acc [] = acc
[ghci| foldr f acc (h:t) = h `f` foldr f acc t
[ghci| :}
[ghci]>
[ghci]> foldr (+) 0 [0..10]
55
[ghci]>
[ghci]> :{
[ghci]
[ghci| foldl _ acc [] = acc
[ghci| foldl f acc (h:t) = foldl f (acc `f` h) t
[ghci| :}
[ghci]>
[ghci]> foldl (+) 0 [0..10]
55
ghci> █
```

HIGHER-ORDER FUNCTIONS ON LISTS

FOLD

► $\text{foldr } f \text{ acc } [x, y, \dots, z] \equiv x `f` (y `f` \dots (z `f` acc) \dots)$

► $\text{foldl } f \text{ acc } [x, y, \dots, z] \equiv (\dots((\text{acc} `f` x) `f` \dots) `f` z$

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| foldr :: (a -> b -> b) -> b -> [a] -> b
[ghci| foldr _ acc [] = acc
[ghci| foldr f acc (h:t) = h `f` foldr f acc t
[ghci| :}
[ghci]>
[ghci]> foldr (+) 0 [0..10]
55
[ghci]>
[ghci]> :{
[ghci| foldl :: (b -> a -> b) -> b -> [a] -> b
[ghci| foldl _ acc [] = acc
[ghci| foldl f acc (h:t) = foldl f (acc `f` h) t
[ghci| :}
[ghci]>
[ghci]> foldl (+) 0 [0..10]
55
ghci> █
```

HIGHER-ORDER FUNCTIONS ON LISTS

FOLD

- ▶ $\text{foldr } f \text{ acc } [x, y, \dots, z] \equiv x `f` (y `f` \dots (z `f` acc) \dots)$
- ▶ $\text{foldl } f \text{ acc } [x, y, \dots, z] \equiv (\dots((acc `f` x) `f` \dots) `f` z$
- ▶ Which of the functions is safe to be applied to an infinite list?

```
Ekaterina.Verbitskaya — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.6.6/...
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
[ghci]> :{
[ghci| foldr :: (a -> b -> b) -> b -> [a] -> b
[ghci| foldr _ acc [] = acc
[ghci| foldr f acc (h:t) = h `f` foldr f acc t
[ghci| :}
[ghci]>
[ghci]> foldr (+) 0 [0..10]
55
[ghci]>
[ghci]> :{
[ghci| foldl :: (b -> a -> b) -> b -> [a] -> b
[ghci| foldl _ acc [] = acc
[ghci| foldl f acc (h:t) = foldl f (acc `f` h) t
[ghci| :}
[ghci]>
[ghci]> foldl (+) 0 [0..10]
55
ghci> █
```

HIGHER-ORDER FUNCTIONS ON LISTS

EXERCISES

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

```
L04 — ghc-9.6.6 -B/Users/Ekaterina.Verbitskaya.ghcup/ghc/9.6.6/lib/ghc-9.6.6/lib -...
GHCi, version 9.6.6: https://www.haskell.org/ghc/ :? for help
[1 of 2] Compiling Main           ( Main.hs, interpreted )
Ok, one module loaded.
[ghci]> sumList []
0
[ghci]> sumList [1,2,3,4,5]
15
[ghci]>
[ghci]> prodList []
1
[ghci]> prodList [1,2,3,4,5]
120
ghci>
```