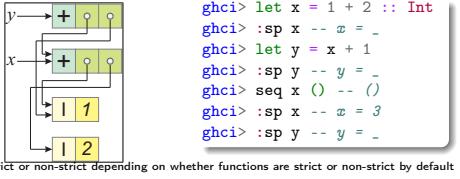


<p style="text-align: center;">Programowanie funkcyjne (wykład 2.)</p> <p style="text-align: center;">Roman Dębski</p> <p style="text-align: center;">Instytut Informatyki, AGH</p> <p style="text-align: center;">3 października 2025 r.</p> 	<p style="text-align: center;">Plan wykładu</p> <ul style="list-style-type: none"> ① Definicja funkcji (uzupełnienie) ② Leniwe obliczanie/wartościowianie ③ Rekursja i przetwarzanie list
--	---

<p>Roman Dębski (II, AGH) Programowanie funkcyjne (wykł.2) 3 października 2025 r. 1 / 14</p> <p>Plan wykładu</p> <ul style="list-style-type: none"> ① Definicja funkcji (uzupełnienie) ② Leniwe obliczanie/wartościowianie ③ Rekursja i przetwarzanie list 	<p>Roman Dębski (II, AGH) Programowanie funkcyjne (wykł.2) 3 października 2025 r. 2 / 14</p> <p>Definicja funkcji: funkcje wielu zmiennych</p> <p>Nazewnictwo zmiennych w Haskellu Nazwy zmiennych nie mogą zaczynać się wielką literą (te są zarezerwowane dla nazw typów).</p> <pre>mPi :: Fractional t => t mPi = 3.141592653589793 -- stała/wartość</pre> <pre>addT :: Num a => (a, a) -> a addT (x,y) = x + y</pre> <pre>addC :: Num a => a -> a -> a addC x y = x + y</pre> <pre>curry :: ((a, b) -> c) -> a -> b -> c uncurry :: (a -> b -> c) -> (a, b) -> c</pre> <pre>ghci> 2 + mPi ghci> addT (1,2) ghci> addC 1 2 ghci> (curry addT) 1 2 -- 3 ghci> (uncurry addC) (1,2) -- 3</pre> <p>Uwaga</p> <pre>f :: a -> b -> c -> d = a -> (b -> (c -> d)) -- prawostronna łączność f 1 2.2 `c` = ((f 1) 2.2) `c` -- lewostronna łączność</pre>
---	--

<p>Roman Dębski (II, AGH) Programowanie funkcyjne (wykł.2) 3 października 2025 r. 3 / 14</p> <p>Currying*, partially applied functions**, sections (przykłady)</p> <pre>addC :: Num a => a -> a -> a -> a addC x y z = x + y + z</pre> <pre>(~) :: (Num a, Integral b) => a -> b -> a flip :: (a -> b -> c) -> b -> a -> c</pre> <pre>ghci> (~) 2 5 -- = 2 ^ 5 = 32 ghci> twoToPower_ :: (2^) -- = (~) 2 ghci> twoToPower_ 5 -- = 32</pre> <pre>ghci> _ToPower5 = (~5) -- = flip (~) 5 ghci> _ToPower5 2 -- = 32</pre> <p>Uwaga</p> <pre>ghci> subtr5From = (-5) -- Num a => a ghci> subtr5From = flip (-) 5 -- c -> c ghci> subtr5From 6 -- = 1, Num c=>c->c -- Prelude.subtract :: Num a => a->a->a</pre> <p>* from Haskell Brooks Curry ** partially applied functions vs. partial functions</p>	<p>Roman Dębski (II, AGH) Programowanie funkcyjne (wykł.2) 3 października 2025 r. 4 / 14</p> <p>Plan wykładu</p> <ul style="list-style-type: none"> ① Definicja funkcji (uzupełnienie) ② Leniwe obliczanie/wartościowianie ③ Rekursja i przetwarzanie list
---	---

<p>Roman Dębski (II, AGH) Programowanie funkcyjne (wykł.2) 3 października 2025 r. 5 / 14</p> <p>Non-strictness* vs. laziness</p> <p>Strictness (semantyka denotacyjna) f is strict iff $f \perp = \perp$ otherwise non-strict, $\perp = \text{bottom}$</p> <p>Laziness (semantyka operacyjna)</p> <p>Leniwe wartościowianie/obliczanie (lazy evaluation) - realizacja/implementacja 'non-strictness' wykorzystująca 'thunks'**</p> <pre>ghci> f x = 42 ghci> f (1/0) -- 42 (f _/_ = 42) ghci> f non_terminating_expr -- 42</pre> <pre>ghci> import Data.Tuple ghci> let x = 1 + 2 :: Int ghci> let z = swap (x,x+1) ghci> :sp z -- z = _ ghci> seq z () -- () ghci> :sp z -- z = (_,_) ghci> seq x () -- () ghci> :sp x -- x = 3 ghci> :sp y -- y = _</pre> <p></p> <p>*the language is strict or non-strict depending on whether functions are strict or non-strict by default **thunks - unevaluated values with a recipe that explains how to evaluate them</p>	<p>Roman Dębski (II, AGH) Programowanie funkcyjne (wykł.2) 3 października 2025 r. 6 / 14</p> <p>Leniwe wartościowianie/obliczanie (przykłady)</p> <pre>ghci> let ys = [1..5] :: [Int] ghci> :sp ys -- ys = _ ghci> seq (length ys) () -- () ghci> :sp ys -- ys = [1,2,3,4,5]</pre> <pre>ghci> let xs = [1..] :: [Int] ghci> :sp xs -- xs = _ ghci> head xs -- 1 ghci> :sp xs -- xs = 1 : _ ghci> xs !! 2 -- 3 ghci> :sp xs -- xs = 1 : 2 : 3 : _</pre> <p>Ciąg Fibonacciego</p> <pre>ghci> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs) :: [Int] ghci> :sp fibs -- fibs = _</pre> <pre>ghci> take 6 fibs -- [0,1,1,2,3,5] ghci> :sp fibs -- fibs = 0 : 1 : 1 : 2 : 3 : 5 : _</pre>
---	--

Plan wykładu

① Definicja funkcji (uzupełnienie)

② Leniwe obliczanie/wartościowanie

③ Rekursja i przetwarzanie list

Roman Dębski (II, AGH)

Programowanie funkcyjne (wykł.2)

3 października 2025 r. 9 / 14

Lista: podstawowe operacje

[uwaga: niektóre z nich zgłoszą RT error, jeśli lista jest pusta]

```

Definicja listy
data [] a = [] | a : [a]

ghci>xs = 1:2:3:4:[] -- [1,2,3,4]
ghci>length xs -- 4
ghci>reverse xs -- [4,3,2,1]

ghci>head xs -- 1
ghci>tail xs -- [2,3,4]
ghci>last xs -- 4
ghci>init xs -- [1,2,3]

ghci>0 : xs -- [0,1,2,3,4]
ghci>xs ++ [5] -- [1,2,3,4,5]
ghci>xs !! 2 -- 3
ghci>[1,2] ++ [3,4] -- [1,2,3,4]

ghci>take 2 xs -- [1,2]
ghci>drop 2 xs -- [3,4]

```

```

ghci>(:) 1 ((:) 2 []) = [1:(2:[])] = [1,2]
-- [], (:) - konstruktor

ghci>null xs -- False
ghci>any (>2) xs -- True
ghci>all (>0) xs -- True

ghci>zip xs ['a','b'] -- [(1,'a'),(2,'b')]
ghci>splitAt 2 xs -- [[1,2],[3,4]]
ghci>sort [2,3,1,4] -- [1,2,3,4]
ghci>2 `elem` xs -- True

ghci>filter even xs -- [2,4]
ghci>map (*2) xs -- [2,4,6,8]
ghci>foldr (+) 0 xs -- 10

ghci>minimum xs -- 1
ghci>maximum xs -- 4
ghci>sum xs -- 10

```

List comprehensions*

Set-builder notation (set comprehension)
 $xs = \{x^2 \mid x \in \{1\dots10\}, x^5 < 1025\}$

List comprehension

$xs = [x^2 \mid x \leftarrow [1..10], x^5 < 1025]$

Generatory i ich zagnieżdżanie, guards

```
ghci> [(x,c) | x <- [1..3], c <- ['a','b']] -- x <- [1..3] - generator
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

```
ghci> [(x,y) | x <- [1..3], y <- [1..x]]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

```
ghci> [(a,b,c) | a <- [1..10], b <- [a..10], c <- [b..10], a^2+b^2==c^2]
[(3,4,5),(6,8,10)]
```

Napisy jako listy znaków

```
ghci> lowers xs = length [x | x <- xs, isLower x]
ghci> lowers "Hello" -- 4
```

*połączenie map i filter

Roman Dębski (II, AGH)

Programowanie funkcyjne (wykł.2)

3 października 2025 r. 11 / 14

Rekursja

Rodzaje rekursji

końcowa/"niekońcowa", bezpośrednia/pośrednia, zagnieżdżona, liniowa/drzewiasta, korekursja, ...

Prosty schemat definicji rekurencyjnej [uwaga: rekursja vs. rekurencja]

```
recurFun x = if f - base case(s) ? -}
              then {- handle base case(s) ... -}
              else {- handle recurrence case(s)... recurFun (f x)... -}
```

```
fact :: Integer -> Integer
-- assert (n >= 0)
fact n = -- **
-- the base case
if n == 0 || n == 1 then 1
else n * fact (n - 1)
```

fibb :: (Num a, Eq a) => a -> a

```
-- assert (n >= 0)
fibb n =
-- the base case
if n == 0 || n == 1 then n
else fibb (n - 2) + fibb (n - 1)
```

notTerm :: Int -> Bool
notTerm x = not (notTerm x)

* uwaga: tail call optimisation in Haskell

** "Never hire a developer who computes the factorial using recursion":)

Roman Dębski (II, AGH)

Programowanie funkcyjne (wykł.2)

3 października 2025 r. 12 / 14

Przetwarzanie list, rekursja, dopasowanie wzorców (przykł.)

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

```
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (x:xs) = qSort (leftPart xs) ++ [x] ++ qSort (rightPart xs)
  where
    leftPart xs = [y | y <- xs, y <= x]
    rightPart xs = [y | y <- xs, y > x]
```

Conditional evaluation with guards

```
fst2Eq :: Eq a => [a] -> Bool
fst2Eq (x : y : _) | x == y = True
fst2Eq _ = False
```

```
prod' :: Num a => [a] -> a
prod' = loop 1 where -- point-free
  loop acc [] = acc -- accumulator
  loop acc (x:xs) = loop (x * acc) xs
```

{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}

Roman Dębski (II, AGH)

Programowanie funkcyjne (wykł.2)

3 października 2025 r. 13 / 14

Bibliografia

- Graham Hutton, Programming in Haskell, Cambridge University Press, 2007
- Simon Marlow, Parallel and Concurrent Programming in Haskell, O'Reilly Media, Inc, 2013

Roman Dębski (II, AGH)

Programowanie funkcyjne (wykł.2)

3 października 2025 r. 14 / 14