

Programowanie funkcyjne (wykład 1.)

Roman Dębski

Instytut Informatyki, AGH

5 października 2021



Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykład.1)

5 października 2021

1 / 34

Plan wykładu

- 1 Organizacja i program zajęć
- 2 Paradygmat funkcyjny na tle innych paradygmatów
- 3 Podstawy programowania w języku Haskell

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykład.1)

5 października 2021

2 / 34

Plan wykładu

- 1 Organizacja i program zajęć
- 2 Paradygmat funkcyjny na tle innych paradygmatów
- 3 Podstawy programowania w języku Haskell

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykład.1)

5 października 2021

3 / 34

Organizacja zajęć

Ocena wyznaczana wg regulaminu studiów na podstawie sumy punktów (s_p) z quizów:

$$s_p = \sum_{i=1}^6 q_i$$

gdzie: $s_p \in [0, 100]$, a $\max(q_i)$:

i	1	2	3	4	5	6
$\max(q_i)$	12	14	16	18	20	20

s_p	ocena
[50, 60)	3.0
[60, 70)	3.5
[70, 80)	4.0
[80, 90)	4.5
[90, 100]	5.0

Uwagi:

- quizy – zadania z zakresu poprzedniego laboratorium,
- w przypadku zajęć zdalnych należy w ustalonym terminie¹ przesłać na platformę moodle potwierdzenie wykonania ćwiczeń (plik **.ipynb** lub pliki **.hs** jako rozwiązania zadań).

¹ uwaga na punkty karne!

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykład.1)

5 października 2021

4 / 34

Program zajęć

- 1 Paradygmat funkcyjny na tle innych paradygmatów. Krótka charakterystyka języka Haskell. Podstawowe typy danych. Składnia definicji funkcji. Polimorfizm. Inferencja typów.
- 2 Leniwe wartościowanie/obliczanie. Rekursja. Przetwarzanie list. Dopasowanie wzorców.
- 3 Funkcje anonimowe. Funkcje wyższego rzędu. Wzorzec "Collection Pipeline".
- 4 Definicje typów. Typy algebraiczne. Typy wyższego rzędu. Klasy typów i ich instancje. Moduły i importy. Organizacja kodu źródłowego (narzędzie stack).
- 5 Operacje wejścia/wyjścia. Funktory. Funktory aplikatywne. Monoidy.
- 6 Monady.
- 7 Elementy teorii kategorii. Kierunki rozwoju języków funkcyjnych.

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykład.1)

5 października 2021

5 / 34

Materiały pomocnicze



<https://wiki.haskell.org>, <https://www.haskell.org>, <https://haskell-lang.org>



Miran Lipovaca, Learn You a Haskell for Great Good!: A Beginner's Guide



Bryan O'Sullivan, Real World Haskell



Graham Hutton, Programming in Haskell



Simon Thompson, Haskell: The Craft of Functional Programming



Christopher Allen, Julie Moronuki, Haskell programming: From First Principles

źródła zdjęć: <https://www.amazon.com>, <https://books.google.pl>

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykład.1)

5 października 2021

6 / 34

Plan wykładu

- 1 Organizacja i program zajęć
- 2 Paradygmat funkcyjny na tle innych paradygmatów
- 3 Podstawy programowania w języku Haskell

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykład.1)

5 października 2021

7 / 34

Paradygmat funkcyjny i Haskell: tło historyczne (1/3)



1) lata 30.

Alonzo Church – rachunek lambda



2) lata 50.

John McCarthy – Lisp



3) lata 60.

Peter Landin – ISWIM (1. język funkcyjny (pure))

źródła zdjęć: en.wikipedia.org, <http://www.cs.nott.ac.uk/~psgmh/book.html>

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykład.1)

5 października 2021

8 / 34

Paradygmat funkcyjny i Haskell: tło historyczne (2/3)

4) lata 70.



John Backus – FP (funkcje wyższego rzędu)

5) lata 70.



Robin Milner i inni – ML (inferencja typu, polimorfizm)

6) lata 70. i 80.



David Turner – "leniwe" języki funkcyjne → Miranda

7) 09.1987



Rozpoczęcie prac nad Haskellem (01.04.1990 Haskell 1.0)

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykł.1)

5 października 2021

9 / 34

Paradygmat funkcyjny i Haskell: tło historyczne (3/3)

7) lata 90.



Philip Wadler i inni – klasy typu i monady

8) 2003



Definicja stabilnej wersji języka Haskell

9) 2010



Haskell 2010

źródła zdjęć: en.wikipedia.org, http://www.cs.nott.ac.uk/~pszgmh/book.html

Roman Dębski (KI, AGH)

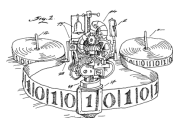
Programowanie funkcyjne (wykł.1)

5 października 2021

10 / 34

Programowanie imperatywne vs. funkcyjne

Programowanie imperatywne
przygotowanie ciągu instrukcji
zmieniających stan programu



```
int sum = 0;
for (int i = 1; i <= 10; ++i)
    sum += i;
```

Programowanie funkcyjne
składanie funkcji (obliczanie
wartości wyrażeń)



```
sum [1..10]
```

A programowanie obiektowe? Deklaratywne? Proceduralne? Generyczne? ...

źródła zdjęć: en.wikipedia.org, www.felienne.com

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykł.1)

5 października 2021

11 / 34

Programowanie imperatywne vs. funkcyjne (przykł.)

C/C++

```
void qsort(int tab[], int left,
           int right) {
    int i = left; int j = right;
    int x = tab[(left+right) >> 1];
    do {
        while (tab[i] < x) i++;
        while (tab[j] > x) j--;
        if (i <= j) {
            int temp = tab[i];
            tab[i] = tab[j];
            tab[j] = temp;
            i++; j--;
        }
    } while (i < j);
    if (left < j) qsort(tab, left, j);
    if (right > i) qsort(tab, i, right);
}
```

Haskell*

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++
               [x] ++
               qsort larger
    where smaller = filter (<= x) xs
          larger = filter (> x) xs
```

*gdzie jest różnica w algorytmie?

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykł.1)

5 października 2021

12 / 34

Języki funkcyjne

"A functional programming language should permit only *pure functions* and should not allow *side effects*. According to that definition, there are hardly any functional languages in use [...]. So I prefer the broad definition: that a *functional language* makes programming centered around functions easy and natural."

— Martin Odersky, creator of Scala

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykł.1)

5 października 2021

13 / 34

Wybrane cechy/mechanizmy języków funkcyjnych

- funkcje jako "pierwszorzędne" elementy języka*,
- domknięcia funkcji (closures),
- "list comprehensions",
- "rozwijanie" funkcji (currying, Haskell Curry),
- leniwe wartościowanie,
- dopasowanie wzorców (pattern matching),
- optymalizacja rekursji ogonowej (tail-call recursion),

oraz dodatkowo, dla języków statycznie typowanych:

- rozbudowane mechanizmy programowania generycznego (w tym "higher-kinded types"),
- klasy typu,
- inferencja typu.

* funkcje mogą być: wartościami zmiennych, argumentami funkcji, wynikami funkcji lub elementami struktur danych

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykł.1)

5 października 2021

14 / 34

Funkcje w matematyce i programowaniu

```
double f1(double x) {
    return 2 * x;
}
double f2() {
    return 4;
}
```

$$y = f(x) = x^2, x \in \mathbb{R}$$

```
int rand(void);
int printf(const char *format, ...);
double f3() {
    printf("f3()\n");
    return 4;
}
int f4() {
    return 4 + rand();
}
```

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykł.1)

5 października 2021

15 / 34

Motywacja

Dlaczego warto nauczyć się co najmniej jednego języka funkcyjnego?

Roman Dębski (KI, AGH)

Programowanie funkcyjne (wykł.1)

5 października 2021

16 / 34

Plan wykładu

- 1 Organizacja i program zajęć
- 2 Paradygmat funkcyjny na tle innych paradygmatów
- 3 Podstawy programowania w języku Haskell

Roman Dębski (KI, AGH) Programowanie funkcyjne (wykł.1) 5 października 2021 17 / 34

Strona domowa i środowisko programistyczne

Strona domowa języka

<https://www.haskell.org/>

Alternatywna strona domowa :)

<https://haskell-lang.org>

Środowisko programistyczne, trzy warianty

- 1 <https://hub.docker.com/r/crosscompass/ihaskell-notebook>
- 2 Stack, <https://docs.haskellstack.org>
- 3 Haskell Platform, <https://www.haskell.org/downloads#platform>

Roman Dębski (KI, AGH) Programowanie funkcyjne (wykł.1) 5 października 2021 18 / 34

Klasyfikacja języka

Na stronie domowej możemy przeczytać, że Haskell to

"an advanced, purely functional programming language"

Kluczowe cechy języka

"czystość", statyczne typowanie, leniwe wartościowanie (domyślnie), inferencja typów, efektywne mechanizmy programowania współbieżnego,...

Roman Dębski (KI, AGH) Programowanie funkcyjne (wykł.1) 5 października 2021 19 / 34

Podstawowe typy danych (typ w Haskellu to kolekcja elementów o wspólnych cechach)

Typy proste

- `1 :: Int` -- (skończona długość reprezentacji)*
- `123~230 :: Integer` -- (dowolne liczby całkowite)
- `3.141592653589793 :: Double/Float`
- `True :: Bool`
- `'a' :: Char`
- `"Ala" :: String` -- `String = [Char]`, typ złożony? :)

Typy złożone

- `(1, 'a', True) :: (Int, Char, Bool)` -- krotka (tuple)
- `[1,2,3,4] :: [Int]` -- lista
- `not :: Bool -> Bool` -- funkcja

*uwaga: dostępne są także: `Int8`, `Int16`, `Int32`, `Int64` oraz ten sam zestaw dla `Word` (całkowite, bez znaku)

Roman Dębski (KI, AGH) Programowanie funkcyjne (wykł.1) 5 października 2021 20 / 34

Podstawowe operatory

```
a == b, a /= b
a < b, a > b
a <= b, a >= b
a && b, a || b, not a
a + b, a - b
a * b, a ^ b, a ** b
a / b, a `mod` b, a `div` b
```

Uwaga

Oba argumenty muszą być tego samego typu (nie dotyczy operatorów `^` i `**`)

Roman Dębski (KI, AGH) Programowanie funkcyjne (wykł.1) 5 października 2021 21 / 34

Komentarze

```
-- ten komentarz rozciąga się do końca linii
main = putStrLn "Hello, hello!" -- instead of Hello World! :)
```

```
{-
  Komentarz blokowy, który może rozciągać
  się na {- Komentarz zagnieżdżony -}
  kilka linii.
-}
main = putStrLn "Hello, hello!"
```

Roman Dębski (KI, AGH) Programowanie funkcyjne (wykł.1) 5 października 2021 22 / 34

Wywołanie funkcji: matematyka vs. Haskell*

Matematyka	Haskell
$f(x)$	<code>f x</code>
$f(x, y)$	<code>f x y</code>
$f(g(x))$	<code>f (g x)</code>
$f(x, g(y))$	<code>f x (g y)</code>
$f(x)g(y)$	<code>f x * g y</code>

Uwaga

wywołanie ("aplikacja") funkcji ma wyższy priorytet niż jakiegokolwiek inny operator w Haskellu

*funkcje wielu zmiennych będą omawiane podczas kolejnego wykładu

Roman Dębski (KI, AGH) Programowanie funkcyjne (wykł.1) 5 października 2021 23 / 34

Definicja funkcji: proste przykłady

```
volume :: Double -> Double
volume r = 4/3 * pi * r^3
volume 3 -- 113.09733552923254
```

```
-- uwaga: ile parametrów ma funkcja areEqual?
areEqual :: (Int, Int) -> Bool
areEqual (x, y) = x == y
areEqual (1, 2) -- False
```

Uwaga

Jeżeli pominiemy w definicji funkcji jej typ, zostanie on wywnioskowany przez kompilator. W przypadku pierwszej funkcji będzie to

```
volume :: Floating a => a -> a
a w przypadku drugiej
areEqual :: Eq a => (a, a) -> Bool
```

Roman Dębski (KI, AGH) Programowanie funkcyjne (wykł.1) 5 października 2021 24 / 34

Definicja funkcji: wyrażenie warunkowe

```
abs :: Int -> Int
abs n = if n >= 0 then n else (-n)

sgn :: Int -> Int
sgn n = if n < 0
      then -1
      else if n == 0 -- zagnieżdżony 'if'
            then 0
            else 1
```

Uwaga

Wyrażenie warunkowe w Haskellu zawsze musi mieć zdefiniowane obie gałęzie. Dlaczego?

Definicja funkcji: guards

```
abs :: Int -> Int
abs n | n >= 0    = n
      | otherwise = -n

sgn :: Int -> Int
sgn n | n < 0    = -1
      | n == 0   = 0
      | otherwise = 1
```

Uwaga

'otherwise' jest zdefiniowane w bibliotece standardowej (Prelude) w następujący sposób:

```
otherwise = True
```

Definicja funkcji: dopasowanie wzorców i wyrażenie case...of

```
not :: Bool -> Bool
not True  = False
not False = True -- not _ = True ?

isTheName :: String -> Bool
isTheName "Rumpelstilkskin" = True
isTheName _                  = False

not :: Bool -> Bool
not b = case b of
  True  -> False
  False -> True  -- _ -> True ?
```

Uwaga

- 1 Dopasowywanie odbywa się od góry, pierwsze dopasowanie kończy proces → kolejność reguł jest istotna! :)
- 2 _ odpowiada 'otherwise' (lub 'default' z innych języków)

Definicja funkcji: klauzula where i wyrażenie let...in (1/2)

```
roots :: (Double, Double, Double) -> (Double, Double)
roots (a, b, c) = ( (-b - d) / e, (-b + d) / e )
  where d = sqrt (b * b - 4 * a * c)
        e = 2 * a

roots' :: (Double, Double, Double) -> (Double, Double)
roots' (a, b, c) =
  let d = sqrt (b * b - 4 * a * c)
      e = 2 * a
  in ( (-b - d) / e, (-b + d) / e )
```

Uwaga

Kolejne lokalne definicje w blokach klauzuli where i wyrażenia let...in można umieścić w nawiasach, oddzielając je średnikami, np.

```
roots :: (Double, Double, Double) -> (Double, Double)
roots (a, b, c) = ( (-b - d) / e, (-b + d) / e )
  where { d = sqrt (b * b - 4 * a * c); e = 2 * a }
```

Definicja funkcji: klauzula where i wyrażenie let...in (2/2)

Lokalność definicji po where i let

```
a = 1.0
f x = a * x where a = 0.5
g x = a * x

f 2.0 -- = 1.0
g 2.0 -- = 2.0
```

where i let razem

```
-- Definicje po let przestają być po where
f x = let a = 10 * x
      in a
      where a = 100 * x

f 1 -- = 10
```

Formatowanie kodu

off-side rule

- pierwszy symbol w serii definicji ustala lewą granicę bloku (→ definicje najwyższego poziomu zaczynają się w tej samej kolumnie)
- definicja może być "złamana" w dowolnym miejscu pod warunkiem, że wcięcia będą większe niż w pierwszej linii (rozpoczynającej definicję)
- jeżeli po where lub let występuje więcej niż jedna definicja lokalna, to wszystkie muszą zaczynać się w tej samej kolumnie
- wyrażenia po of muszą zaczynać się w tej samej kolumnie

Uwaga

Bloki można definiować używając nawiasów i średników. Taki styl jest jednak rzadko spotykany.

Polimorfizm* w Haskellu

Rodzaje polimorfizmu

- statyczny vs. dynamiczny
- ad-hoc, parametryczny, inkluzyjny

Polimorfizm parametryczny (Haskell, inne języki funkcyjne) fragmenty programu (funkcje i/lub struktury danych) mogą być parametryzowane typami. Typ (struktura danych, funkcja) jest polimorficzny, jeśli zawiera co najmniej jedną zmienną typu, np.

```
id :: a -> a
id x = x
```

*z gr. wielość form, wielopostaciowość

Inferencja (wnioskowanie) typu (Hindley-Milner type system)

```
(+) :: Num a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
(/) :: Fractional a => a -> a -> a
```

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

```
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
```

Przykład wyniku wnioskowania

```
f x = if x > 3 then 2 * x else 42
f :: (Num a, Ord a) => a -> a
```

Przykłady wnioskowania typu

```
swap(x,y) = (y,x)           f1 (x,y,z) = (y,x,z)
swap :: (t1, t) -> (t, t1)  f1 :: (t1, t, t2) -> (t, t1, t2)
```

```
f2 (x,y,z) = if (x > y) then (x,y,z) else (y,x,z)
f2 :: Ord t => (t, t, t1) -> (t, t, t1)
```

```
f3 x = if x > 3 then 2 * x else x / 42
f3 :: (Fractional a, Ord a) => a -> a
```

Jaki będzie wynik wnioskowania typu?

```
f4 x = if x > 3 then 2 * x else False

A f4 :: (Num a, Ord a) => a -> a
B f4 :: (Num a, Ord a) => a -> Bool
C żaden z powyższych :)
```

Praca z kodem: warianty uruchamiania

myprog.hs - przykładowy plik źródłowy

```
c = 1
f x = x * x
main = putStrLn "Hello, from myprog.hs!"
```

```
ghci
$ ghci
GHCi, version 7.10.3: ... :? for help
ghci> :load myprog.hs
[1 of 1] Compiling Main (myprog.hs, interpreted)
Ok, modules loaded: Main.

ghci> c
1
ghci> f 5
25
ghci> main
Hello, from myprog.hs!
```

```
runghc
$ runghc myprog.hs
Hello, from myprog.hs!

ghc
$ ghc myprog.hs [-o myprog]
$ ./myprog
Hello, from myprog.hs!

Script interpreter (stack jako interpreter)
#!/usr/bin/env stack
```