Notatki Programowanie Funkcyjne

Jakub Kogut

11 marca 2025

Spis treści

1 Wstęp

Notatki z programowania funkcyjnego prowadzone przez GOATA profesora Jacka Cichonia na semestrze 4 2025. Zajęcia laboratoryjne prowadzone są przez dr Dominika Bojko.

2 Wykład 11-03-2025

Na tym wykładzie skupimy się na przygotowaniu środowiska pracy do programowania funkcyjnego w języku Haskell.

2.1 Struktura kodu w Haskell

Przykładowy kod wygląda następująco:

```
{-  file = W2.hs
    autor = JK
    date = 11-03-2025
-}
module W2 where
id' x = x
```

Następnie w terminalu, w którym mamy odpalone GHCI wpisujemy:

```
>:l W2.hs
>:r
>id' 5
5
>:t id'
id' :: a -> a //co oznacza id :: forall a => a->a
```

Co matematycznie można zapisać jako:

$$exp = (\lambda a : Typ \to (a \to a))$$

- Przykład:
  - $exp(Int) :: Int \to Int$
  - $exp(Bool) :: Bool \to Bool$
  - $exp(Double) :: Double \to Double$

Cichoń radzi, aby narpiew zastanowić się jaki powinnen być typ funkcji, a dopiero potem zastanawiać się nad implementacją, ponoć oszczędza to *czas i nerwy*.

2.2 Typy w Haskellu

- Typy proste:
  - Int
  - Double
  - Char
  - Bool
- Typy złożone:
  - Listy

- Krotki
- Funkcje

- Przykład:
  - funkcja Collatz'a $coll :: Int \to Int$

```
coll n
    | n==1 = 1
    | even n = coll (n `div` 2)
    | odd n = coll (3*n+1)
```

Symbol | oznacza wyrażenie z wykożystaniem strażników *guards*. Zapis taki jest podobny do matematycznego zapisu funkcji:

$$coll(n) = \begin{cases} 1 & \text{gdy } n = 1 \\ coll(\frac{n}{2}) & \text{gdy } n \text{ jest parzyste} \\ coll(3n+1) & \text{gdy } n \text{ jest nieparzyste} \end{cases}$$

Nie jest to bezpieczna funkcja, ponieważ dla liczb ujemnych zapętli się ona w nieskończoność. Można zauważyć, że funkcja ta zwraca zawsze liczbę 1. Ciekawa jest liczba kroków, które są potrzebne do osiągnięcia tej wartości. Dla $n = 27$ potrzeba 111 kroków, dla $n = 28$ potrzeba 18 kroków, dla $n = 29$ potrzeba 111 kroków.

  - Nowa funkcja Collatz'a
    collatz::(Int,Int) $\to$ (Int,Int)

```
collatz (n,steps)
    | n==1 = (n,steps)
    | even n = collatz (n `div` 2,steps+1)
    | odd n = collatz (3*n+1,steps+1)
```

Funkcja ta zwraca parę liczb, pierwsza to wynik funkcji Collatz'a, a druga to liczba kroków potrzebna do osiągnięcia tej wartości.
Spróbujmy ją sobie odpalić:

```
>collatz (97,0)
(1,118)
```

Jak widać dla $n = 97$ potrzeba 118 kroków, aby osiągnąć wartość 1.

  - Funkcja lenz *lenght of collatz* zwracająca długość ciągu Collatz'a dla danej liczby:
    lenz::Int $\to$ Int

```
lenz n = snd (collatz (n,0))
```

2.3 Listy

Definicja listy w Haskellu:
[a] - lista elementów typu a

$$[\text{a}] = \{[a_1, \ldots, a_k] \mid a_1, \ldots, a_k \in a, k \in \mathbb{N}\}$$

```
>:t [1,2,3]
[1,2,3] :: Num a => [a]
>:t [1::Integer, 2, 3]
[1,2,3] :: [Integer]
```

2.3.1 Operacje na listach

- Dodawanie elementu na początku listy

```
>:t (1:[2,3])
(1:[2,3]) :: Num a => [a]
```

- Konkatenacja list

```
>:t [1,2]++[3,4]
[1,2]++[3,4] :: Num a => [a]
```

2.3.2 Podstawowe funkcje operujace na listach

- length::[a] → Int
  - length [] = 0
  - length (x:xs) = 1 + length xs
- head::[a] → a
  zwraca pierwszy element listy
  - head (x:xs) = x
  - head [] = error "empty list"
- tail::[a] → [a]
  zwraca listę bez pierwszego elementu
  - tail (x:xs) = xs
  - tail [] = error "empty list"
- last::[a] → a
  zwraca ostatni element listy
  - last [x] = x
  - last (x:xs) = last xs
  - last [] = error "empty list"
- filter::(a → Bool) → [a] → [a]
  - filter p [] = []
  - filter p (x:xs) = if p x then x : filter p xs else filter p xs
  - filter (n → n>0) [-1,2,-3,4] = [2,4]
  - filter even [1..10] = [2,4,6,8,10]
    Jak zdefiniować funkcje filter:

```
filter p [] = []
filter p (x:xs)
    | p x = x : filter p xs
    | otherwise = filter p xs
```

- map::(a → b) → [a] → [b]
  zwraca listę, która powstaje zastosowaniem funkcji do każdego elementu listy
  - map f [] = []
  - map f (x:xs) = f x : map f xs

- map (n → n*n) [1,2,3] = [1,4,9]
- map (n → n^3) [1..10] = [1..1000]

gdzie **[1..10]** to skrót od **[1,2,3,4,5,6,7,8,9,10]**

2.3.3 List comprehension

Polega na tworzeniu listy na podstawie innych list.

$$\boxed{[fx_1, x_2, x_3 \mid x_1 \leftarrow xs, x_2 \leftarrow ys, x_3 \leftarrow zs]}$$

Przykład:

- chcemy stworzyć listę wszystkich trójek pitagorejskich ponizej liczby n.

```
pitagorasTrzy n = [(x,y,z) | x <- [1..n], y <- [1..n], z <- [1..n],
                   x^2 + y^2 == z^2, gcd xy == 1]
```

3 Ćwiczenia

W tym miejscu będa pojawiały się notatki z laboratorów (ćwiczeń)

3.1 Ćwiczenia 11-03-2025

3.1.1 Zadanie 1

```
power :: Int => Int => Int
power x y = y ^ x

p2 = power 4
p3 = power 3
```

1. Wyznacz w GCHI wartość wyrażenia $(p2 \circ p3)^2$ i wyjaśnij, dlaczego otrzymałeś ten wynik.
2. Zbadaj typy funkcji $p2$, $p3$ i $(p2 \circ p3)$.
3. Zapisz powyższe funkcje za pomocą wyrażeń lambda.

$$Int \to Int \to Int$$

Zapis strzałkowy definuje nam typ funkcji operacja => jest wiążaca z prawej strony, wiec można by było to również zapisać jako:

$$power :: Int \to (Int \to Int)$$

1. podpunkt 1

$$(p2 \circ p3)^2 = p2(p3(x))^2 = 4(3^x)^2 = 4 \cdot 9^x$$

2. podpunkt 2

```
>:t p2
p2 :: Int -> Int
>:t p3
p3 :: Int -> Int
>:t (p2 . p3)
(p2 . p3) :: Int -> Int
```

3. podpunkt 3

```
p2 = \x -> power 4 x
p3 = \x -> power 3 x
```

3.1.2 Zadanie 2

$$2 \wedge 3 \wedge 2, \quad (2 \wedge 3) \wedge 2, \quad 2 \wedge (2 \wedge 3).$$

Dowiedz się, jaka jest łączność oraz siła operatora $\wedge$ za pomocą polecenia:

$$: i(\wedge).$$

```
>:i (^)
(^) :: (Num a, Integral b) => a -> b -> a
        -- Defined in ‘GHC.Real’
infixr 8 ^
```

Operator $\wedge$ jest prawostronnie łączny, a jego siła wynosi 8 (najwyższa możliwa wartość, wyłącznie wyższe jest nałożenie funkcji na zmienną). W nawiasie **Num a, Integraf b** oznacza, że operator $\wedge$ bierze jeden argument typu **Num** i drugi typu **Integral**.

$$2 \wedge 3 \wedge 2 = 2 \wedge (3 \wedge 2) = 2 \wedge 9 = 512$$

$$(2 \wedge 3) \wedge 2 = 8 \wedge 2 = 64$$

$$2 \wedge (2 \wedge 3) = 2 \wedge 8 = 256$$

3.1.3 Zadanie 3

```
f : : Int => Int
f x = x ^ 2
g : : Int => Int => Int
g x y = x+2*y
h : : . . . .
h x y = f ( g x y )
```

1. Jaki jest typ funkcji h? (tzn. uzupełnij ... w powyższym listingu)
2. Czy $h = f \circ g$?
3. Czy $h\,x = f(g\,x)$?

---

1. Typ funkcji h to:
$$h :: Int \to Int \to Int$$
2. Nie, ponieważ:
$$h(x, y) = f(g(x, y)) = f(x + 2y) = (x + 2y)^2$$
3. Tak, ponieważ:
$$h(x) = f(g(x)) = f(x + 2x) = (x + 2x)^2 = 9x^2$$

3.1.4 Zadanie 4

Zapisz operacje binarne $(+)$, $(*)$ za pomocą lambda wyrażeń.

```
add = \x -> (\y -> x + y)
mul = \x -> (\y -> x * y)
```

Co to daje? Mozna teraz zapisać 2+3 jako:

- add 2 3
- (add 2) 3
- add 2 (3)
- ($3) (2 add)

3.1.5 Zadanie 5

Zapisz funkcje:

$$f(x) = 1 + x \cdot (x + 1), \quad g(x, y) = x + y^2, \quad h(y, x) = x + y^2$$

za pomocą lambda wyrażeń w językach C++, Python, JavaScript oraz Haskell.

W języku Haskell:

```
f = \x -> 1 + x * (x + 1)
g = \x -> \y -> x + y^2
h = \y -> \x -> x + y^2
```

W języku Python:

```
f = lambda x: 1 + x * (x + 1)
g = lambda x, y: x + y**2
h = lambda y, x: x + y**2
```

W języku JavaScript:

```
f = x => 1 + x * (x + 1)
g = (x, y) => x + y**2
h = (y, x) => x + y**2
```

W języku C++:

```
auto f = [](int x) { return 1 + x * (x + 1); };
auto g = [](int x, int y) { return x + y*y; };
auto h = [](int y, int x) { return x + y*y; };
```

3.1.6 Zadanie 6

Ustalmy zbiory A, B, C. Niech

$$\text{curry} : C^{B\times A} \to (C^B)^A$$

będzie funkcją zadaną wzorem:

$$\text{curry}(\varphi) = \lambda a \in A \to (\lambda b \in B \to \varphi(b, a)).$$

oraz niech

$$\text{uncurry} : (C^B)^A \to C^{B\times A}$$

będzie zadana wzorem:

$$\text{uncurry}(\psi)(b, a) = (\psi(a))(b).$$

1. Pokaż, że $\text{curry} \circ \text{uncurry} = \text{id}_{(C^B)^A}$ oraz $\text{uncurry} \circ \text{curry} = \text{id}_{C^{B\times A}}$.
2. Wywnioskuj z tego, że $|(C^B)^A| = |C^{B\times A}|$. Przypomnij sobie dowód tego twierdzenia, który poznałeś na pierwszym semestrze studiów.
3. Spróbuj zdefiniować w języku Haskell odpowiedniki funkcji `curry` i `uncurry`.

---

1. Pokażemy, że $\text{curry} \circ \text{uncurry} = \text{id}_{(C^B)^A}$ oraz $\text{uncurry} \circ \text{curry} = \text{id}_{C^{B\times A}}$.

   - $\text{curry} \circ \text{uncurry}$

   $$\begin{aligned}(\text{curry} \circ \text{uncurry})(\psi) &= \text{curry}(\text{uncurry}(\psi)) \\ &= \text{curry}(\lambda a \in A \to (\lambda b \in B \to \psi(a)(b))) \\ &= \lambda a \in A \to (\lambda b \in B \to \psi(a)(b)).\end{aligned} \tag{1}$$

   - $\text{uncurry} \circ \text{curry}$

   $$\begin{aligned}(\text{uncurry} \circ \text{curry})(\varphi) &= \text{uncurry}(\text{curry}(\varphi)) \\ &= \text{uncurry}(\lambda a \in A \to (\lambda b \in B \to \varphi(b, a))) \\ &= \lambda b \in B \to (\lambda a \in A \to \varphi(b, a)).\end{aligned} \tag{2}$$

   Z powyższych równań wynika, że $\text{curry} \circ \text{uncurry} = \text{id}_{(C^B)^A}$ oraz $\text{uncurry} \circ \text{curry} = \text{id}_{C^{B\times A}}$. □

2. Możemy pokazać że `curry` i `uncurry` są iniekcjami niewprost, nakładając odpowiednio przeciwne funkcje na obie strony równości:

   - Załóżmy, że $\text{curry}(\varphi_1) = \text{curry}(\varphi_2)$. Wtedy:

   $$\begin{aligned}\text{curry}(\varphi_1)(a)(b) &= \text{curry}(\varphi_2)(a)(b) \\ \varphi_1(b, a) &= \varphi_2(b, a) \\ \varphi_1 &= \varphi_2.\end{aligned} \tag{3}$$

   - Załóżmy, że $\text{uncurry}(\psi_1) = \text{uncurry}(\psi_2)$. Wtedy:

   $$\begin{aligned}\text{uncurry}(\psi_1)(b, a) &= \text{uncurry}(\psi_2)(b, a) \\ \psi_1(a)(b) &= \psi_2(a)(b) \\ \psi_1 &= \psi_2.\end{aligned} \tag{4}$$

A więc istnieje biekcja między $(C^B)^A$ i $C^{B\times A}$, co oznacza, że te zbiory mają taką samą moc. □

3. W języku Haskell funkcje `curry` i `uncurry` można zdefiniować następująco:

```
curry :: ((b, a) -> c) -> a -> b -> c
curry f x y = f (y, x)
```

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y
```

3.1.7 Zadanie 7

Podaj przykłady funkcji następujących typów:

$$(\text{Int} \Rightarrow \text{Int}) \Rightarrow \text{Int}$$

$$(\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int})$$

$$(\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int})$$

- Funkcja typu $(\text{Int} \Rightarrow \text{Int}) \Rightarrow \text{Int}$:

```
f :: (Int -> Int) -> Int
f g = g 0
```

- Funkcja typu $(\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int})$:

```
f :: (Int -> Int) -> (Int -> Int)
f g x = g (g x)
```

- Funkcja typu $(\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int})$:

```
f :: (Int -> Int) -> (Int -> Int) -> (Int -> Int)
f g h x = g (h x)
```

3.1.8 Zadanie 8

Załóżmy, że chcesz oprogramować funkcję, która dla danych liczb a, b oraz funkcji $f : \mathbb{R} \to \mathbb{R}$ oblicza

$$\int_a^b f(x)\,dx.$$

Jaki powinien być typ tej funkcji?

Typ tej funkcji powinien być następujący:

$$\text{Num(a)} \implies a \to a \to (a \to a) \to a$$

mogłaby ona wyglądać następująco:

```
integral :: (Double -> Double) -> Double -> Double -> Double
integral f a b = undefined
```

3.1.9 Zadanie 9 – (Eliminacja Pętli)

Wybierz jeden z języków Python, C++ lub JavaScript.

1. Masz daną (czyli oprogramowaną) funkcję $f : \mathbb{N} \to \mathbb{N}$. Oprogramuj funkcję, która dla danego $n \in \mathbb{N}$ oblicza
$$\sum_{k=0}^{n} f(k).$$
Zrób to najpierw (standardowo) za pomocą pętli, a potem oprogramuj ją bez użycia pętli, za pomocą rekursji.

2. Rozważamy następującą funkcję napisaną w pseudokodzie:

```
FUNCTION f(x: DOUBLE): DOUBLE
BEGIN
    DOUBLE y = sin(x);
    RETURN y*y + y + x;
ENDFNC
```

Oprogramuj tę funkcję w wybranym języku i następnie wyeliminuj zmienną lokalną y z tego kodu, bez pogarszania jego efektywności.

---

1. Oto rozwiązanie w języku C++:

```cpp
#include <iostream>
using namespace std;

// Example implementation of function f: N -> N.
// You can replace this with any function of type int -> int.
int f(int x) {
    // Example: f(x) = x + 1
    return x + 1;
}

// Function that sums using a loop:
int sumLoop(int n) {
    int sum = 0;
    for (int k = 0; k <= n; ++k) {
        sum += f(k);
    }
    return sum;
}

// Function that sums using recursion:
int sumRec(int n) {
    if(n == 0)
        return f(0);
    else
        return sumRec(n - 1) + f(n);
}

int main() {
    int n;
```

```
    cout << "Enter n: ";
    cin >> n;
    cout << "Sum computed with loop: " << sumLoop(n) << endl;
    cout << "Sum computed recursively: " << sumRec(n) << endl;
    return 0;
}
```

funkcja `sumLoop` oblicza sumę za pomocą pętli, a funkcja `sumRec` oblicza sumę rekurencyjnie.

$$\text{sumRec}(n) = \begin{cases} f(0) & \text{gdy } n = 0 \\ \text{sumRec}(n-1) + f(n) & \text{gdy } n > 0 \end{cases}$$

2. Rozwiazanie w Haskellu:

```
f :: Double -> Double
f x = sin x * sin x + sin x + x
```

```
f' :: Double -> Double
f' x = sin x * sin x + sin x + x
```

3.1.10 Zadanie 10

Zaimplementuj samodzielnie następujące funkcje działające na listach z Prelude:

1. map
2. zip
3. zipWith
4. filter
5. take
6. drop
7. fib

---

1. Funkcja `map`:

```
map' :: (a -> b) -> [a] -> [b]
map' f [] = []
map' f (x:xs) = f x : map' f xs
```

2. Funkcja `zip`[1]:

```
zip' :: [a] -> [b] -> [(a, b)]
zip' [] _ = []
zip' _ [] = []
zip' (x:xs) (y:ys) = (x, y) : zip' xs ys
```

[1]Funkcja `zip` zwraca listę par, które są złożone z elementów listy wejściowej. Jeśli jedna z list jest krótsza, to wynikowa lista będzie miała długość krótszej z nich.
Przykład: `zip [1,2,3] ['a','b','c','d']` zwróci `[(1,'a'),(2,'b'),(3,'c')]`.

3. Funkcja `zipWith`[2]:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

4. Funkcja `filter`[3]:

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' p (x:xs)
    | p x = x : filter' p xs
    | otherwise = filter' p xs
```

5. Funkcja `take`[4]:

```
take' :: Int -> [a] -> [a]
take' 0 _ = []
take' _ [] = []
take' n (x:xs) = x : take' (n - 1) xs
```

6. Funkcja `drop`[5]:

```
drop' :: Int -> [a] -> [a]
drop' 0 xs = xs
drop' _ [] = []
drop' n (_:xs) = drop' (n - 1) xs
```

7. Funkcja `fib`[6]:

```
fib :: Int -> [Int]
fib n = take' n (map' fib' [0..])
    where
        fib' 0 = 0
        fib' 1 = 1
        fib' n = fib (n - 1) + fib (n - 2)
```

Fajne złożenie funkcji `fib` z `zipWith`:

```
fib :: [Int]
fib = 0 : 1 : zipWith (+) fib (tail fib)
```

co pozwala na generowanie listy liczb Fibonacciego w nieskończoność. Na przykład `take 10 fib` zwróci `[0,1,1,2,3,5,8,13,21,34]`.

[2]Funkcja `zipWith` działa podobnie jak `zip`, ale zamiast zwracać parę elementów, zwraca wynik funkcji, która jest podana jako argument.
Przykład: `zipWith (+) [1,2,3] [4,5,6]` zwróci `[5,7,9]`.

[3]Funkcja `filter` zwraca listę elementów, które spełniają warunek podany jako argument.
Przykład: `filter even [1..10]` zwróci `[2,4,6,8,10]`.

[4]Funkcja `take` zwraca listę składającą się z n pierwszych elementów listy wejściowej.
Przykład: `take 3 [1,2,3,4,5]` zwróci `[1,2,3]`.

[5]Funkcja `drop` zwraca listę, która jest wynikiem usunięcia n pierwszych elementów z listy wejściowej.
Przykład: `drop 3 [1,2,3,4,5]` zwróci `[4,5]`.

[6]Funkcja `fib` zwraca listę liczb Fibonacciego do n-tego elementu.