Notatki Programowanie Funkcyjne

Jakub Kogut

11 marca 2025

Spis treści

1 Wstęp **2**

2 Wykład 11-03-2025 **2**
2.1 Struktura kodu w Haskell 2
2.2 Typy w Haskellu 2
2.3 Listy 3
2.3.1 Operacje na listach 4
2.3.2 Podstawowe funkcje operujace na listach 4
2.3.3 List comprehension 5

3 Podsumowanie **5**

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.

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

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 Podsumowanie

Podsumowanie lub zakończenie notatek.