Podstawy Programowania Funkcyjnego

Chlebowski Sz. Gajda A.

Zakład Logiki i Kognitywistyki
Wydział Psychologii i Kognitywistyki
Uniwersytet im. Adama Mickiewicza

17.03.2020

Plan na dziś

1. Wyrażenia warunkowe
2. Guarded equations
3. Pattern matching
4. N-tki
5. Listy
6. Wyrażenia lambda
7. Operatory

Co już umiemy i wiemy

- Zarys kontekstu historycznego i teoretycznego.
- Proste, rekurencyjne definicje funkcji.
- Obiekty, typy, klasy.
- Stworzyć plik źródłowy Haskella i uruchomić go w GHCi w terminalu.

- Liczby podzielne przez 2

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0
```

- Liczby podzielne przez 2

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0
```

- Podziel listę w punkcie

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)
```

- Liczby podzielne przez 2

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0
```

- Podziel listę w punkcie

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)
```

- Odwrotność

```
recip :: Fractional a => a -> a
recip n = 1/n
```

Wyrażenia warunkowe

- Wartość absolutna

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

- Wartość absolutna

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

- Signum

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
             if n == 0 then 0 else 1
```

Guarded equations

- Wartość absolutna

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

- Wartość absolutna

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

- Signum

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

Pattern matching

- Nieprawda, że

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

- Nieprawda, że

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

- Koniunkcja

```
(&&) :: Bool -> Bool -> Bool
True  && True   = True
True  && False  = False
False && True   = False
False && False  = False
```

- Koniunkcja

```
(&&) :: Bool -> Bool -> Bool
True && True     = True
_ && _           = False
```

- Koniunkcja

```
(&&) :: Bool -> Bool -> Bool
True && True     = True
_ && _           = False
```

- Koniunkcja

```
(&&) :: Bool -> Bool -> Bool
True && b    = b
_ && _       = False
```

- Koniunkcja

```
(&&) :: Bool -> Bool -> Bool
True && True     = True
_ && _           = False
```

- Koniunkcja

```
(&&) :: Bool -> Bool -> Bool
True && b    = b
_ && _       = False
```

- Koniunkcja (zła definicja)

```
(&&) :: Bool -> Bool -> Bool
b && b   = b
_ && _   = False
```

N-tki

- Pierwszy element

```
fst :: (a,b) -> a
fst (x,_) = x
```

- Pierwszy element

```
fst :: (a,b) -> a
fst (x,_) = x
```

- Drugi element

```
snd :: (a,b) -> b
snd (_,y) = y
```

Listy

- 3 elementy, pierwszy to litera ‘a’

```
test :: [Char] -> Bool
test ['a',_,_]  = True
test _          = False
```

- 3 elementy, pierwszy to litera ‘a’

```
test :: [Char] -> Bool
test ['a',_,_]  = True
test _          = False
```

- Dowolna ilość elementów, pierwszy to litera ‘a’

```
test' :: [Char] -> Bool
test' ('a':_)   = True
test' _         = False
```

Wyrażenia lambda

Wyrażenia lambda (1/3)

- Funkcja

```
> (\x -> x + x) 2
4
```

Wyrażenia lambda (1/3)

- Funkcja

```
> (\x -> x + x) 2
4
```

- Dodawanie

```
add :: Int -> Int -> Int
add x y = x + y

add' :: Int -> (Int -> Int)
add' = \x -> (\y -> x + y)
```

Wyrażenia lambda (2/3)

- Stała

```
const :: a -> b -> a
const x _ = x

const' :: a -> (b -> a)
const' x = \_ -> x
```

Wyrażenia lambda (3/3)

- Pierwsze `n` liczb nieparzystych

```haskell
odds :: Int -> [Int]
odds n = map f [0..n-1]
             where f x = x*2 + 1

odds' :: Int -> [Int]
odds' n = map (\x -> x*2 + 1) [0..n-1]
```

Operatory

- Suma elementów w liście

```
sum :: [Int] -> Int
sum = foldl (+) 0
```

Zadania (1/5)

1. Zdefiniuj przy pomocy funkcji dostępnych w bibliotece funkcję:

```
> halve [1,2,3,4,5,6]
([1,2,3],[4,5,6])

halve :: [a] -> ([a],[a])
```

2. Zdefiniuj funkcję `third`, która zwraca trzeci element listy (która z kolei zawiera co najmniej 3 elementy):
   (a) użyj `head` i `tail`,
   (b) użyj `!!`,
   (c) użyj *pattern matching*.

3. Zdefiniuj funkcję

```
safetail :: [a] -> [a]
```

zwracającą ten sam rezultat, co funkcja `tail`, przy czym zamiast zwracać błąd przy pustej liście, mapuje ją w pustą listę. Uzyj w tym celu funkcji, która sprawdza, czy lista jest pusta

```
null :: [a] -> Bool
```

oraz:

   (a) wyrażenia warunkowego,
   (b) *guarded equations*,
   (c) *pattern matching*.

4. Zdefiniuj alternatywę || w podobny sposób, jak została zdefiniowana koniunkcja &&.

5. Zdefiniuj koniunkcję && przy pomocy tylko i wyłącznie wyrażeń warunkowych:

```
True  &&  True  =  True
_  &&  _        =  False
```

6. Zdefiniuj koniunkcję && przy pomocy tylko i wyłącznie wyrażeń warunkowych:

```
True   &&  b    =  b
False  &&  _    =  False
```

7. Pokaż, że funkcja

```
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

może zostać zdefiniowana przy pomocy wyrażeń lambda.

8. Zdefiniuj algorytm *Luhna*, który stosowany jest do sprawdzania prostych błędów w numerach kart kredytowych:
   - każda cyfra traktowana jest jako osobny numer,
   - poruszając się w lewo, podwojony jest każdy co drugi numer rozpoczynając od przedostatniego,
   - odjęte zostaje 9 od każdego numeru który jest teraz większy od 9,
   - dodane zostają do siebie wszystkie otrzymane numery,
   - jeżeli otrzymana wartość jest podzielna przez 10, to znaczy, że numer karty jest poprawny.

   Jako pomocniczą funkcję, zdefiniuj

```
luhnDouble :: Int -> Int
```

   podwajającą numer i odejmującą od rezultatu 9, jeżeli rezultat ten jest większy niż 9, np.:

```
> luhnDouble 3
6

> luhnDouble 6
3
```

Zadania (5/5)

Używając `luhnDouble` i `mod` zdefiniuj funkcję `luhn`:

```
luhn :: Int -> Int -> Int -> Int -> Bool

> luhn 1 7 8 4
True

> luhn 4 7 8 3
False
```