Podstawy Programowania Funkcyjnego

Chlebowski Sz. Gajda A.

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

05.04.2023

1. Funkcja 'map' i 'foldl'
2. List comprehensions
   - Generatory
   - Strażnicy (guards)
   - Funkcja zip
3. Szyfr Cezara
4. Zadania

Funkcja 'map'

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

Funkcja 'foldl'

```
foldl :: (b -> a -> b) -> b -> t a -> b
```

- Np. suma elementów w liście

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

Generatory

- Potęgi liczb od 1 do 5

```
> [x^2 | x <- [1..5]]
[1, 4, 9, 16, 25]
```

Przykłady

- Potęgi liczb od 1 do 5

```
> [x^2 | x <- [1..5]]
[1, 4, 9, 16, 25]
```

- Lista wszystkich par, których pierwszy element należy do listy [1,2,3], zaś drugi do listy [4,5]

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]

> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4), (2,4), (3,4), (1,5), (2,5), (3,5)]
```

- Potęgi liczb od 1 do 5

```
> [x^2 | x <- [1..5]]
[1, 4, 9, 16, 25]
```

- Lista wszystkich par, których pierwszy element należy do listy [1,2,3], zaś drugi do listy [4,5]

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]

> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4), (2,4), (3,4), (1,5), (2,5), (3,5)]
```

- Generator:

```
x <- [k..m]
```

- Zbiór par elementów dwóch list

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

- Zbiór par elementów dwóch list

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

- Konkatenacja dowolnej ilości list (uogólnione $++$)

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

- Zbiór par elementów dwóch list

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

- Konkatenacja dowolnej ilości list (uogólnione $++$)

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

- Lista wszystkich pierwszych elementów par

```
firsts :: [(a,b)] -> [a]
firsts ps = [x | (x,_) <- ps]
```

- Zbiór par elementów dwóch list

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

- Konkatenacja dowolnej ilości list (uogólnione $++$)

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

- Lista wszystkich pierwszych elementów par

```
firsts :: [(a,b)] -> [a]
firsts ps = [x | (x,_) <- ps]
```

- Długośc listy

```
length :: [a] -> Int
length xs = sum [1 | _ <- xs]
```

Strażnicy (guards)

- Dzielniki *n*

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]

> factors 15
[1, 3, 5, 15]

> factors 7
[1,7]
```

- Test na liczby pierwsze

```
prime :: Int -> Bool
prime n = factors n == [1,n]

> prime 15
False
```

- Test na liczby pierwsze

```
prime :: Int -> Bool
prime n = factors n == [1,n]

> prime 15
False
```

- Lista *n* początkowych liczb pierwszych

```
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]

> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

- Lista wartości dla danego klucza

```
find :: Eq a => a -> [(a,b)] -> [b]
find k t = [v | (k',v) <- t, k == k']

> find 'b' [('a',1),('b',2),('c',3),('b',4)]
[2,4]
```

Funkcja zip

- Funkcja zip łączy ze sobą (w pary) elementy dwóch list, dopóki jedna z nich nie zostanie wyczerpana

```
> zip ['a', 'b', 'c'] [1,2,3,4]
[('a', 1), ('b', 2), ('c', 3)]

> zip ['a', 'b', 'c', 'd'] [1,2,3]
[('a', 1), ('b', 2), ('c', 3)]
```

- Lista par

```
pairs :: [a] -> [(a, a)]
pairs xs = zip xs (tail xs)

> pairs [1, 2, 3, 4]
[(1, 2), (2, 3), (3, 4)]
```

- Funkcja sort sprawdza, czy lista jest uporządkowana

```
sorted :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x, y) <- pairs xs]
```

- Funkcja sort sprawdza, czy lista jest uporządkowana

```
sorted :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x, y) <- pairs xs]
```

- Funkcja positions zwraca listę pozycji, na których występuje dana wartość

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs=[i|(x',i)<-zip xs [0..n],x==x']
              where
                  n = length xs - 1

> positions False [True, False, True, False]
[1,3]
```

- Liczba małych liter

```
lowers :: String -> Int
lowers xs = length [x | x <- xs, isLower x]

> lowers "Haskell"
6
```

- Liczba małych liter

```
lowers :: String -> Int
lowers xs = length [x | x <- xs, isLower x]

> lowers "Haskell"
6
```

- Ilość wystąpień danego symbolu w liście

```
count :: Char -> String -> Int
count x xs = length [x' | x' <- xs, x==x']

> count 's' "Mississippi"
4
```

Szyfr Cezara

Kod do „Szyfru Cezara”

Kod, który opisywany jest na następnych slajdach, dostępny jest do ściągnięcia pod tym linkiem:

`http://www.cs.nott.ac.uk/~pszgmh/Code.zip`

w pliku „cipher.hs”.

- Zamiana liter na liczby

```
let2int :: Char -> Int
let2int c = ord c - ord 'a'
```

- Zamiana liter na liczby

```
let2int :: Char -> Int
let2int c = ord c - ord 'a'
```

- Zamiana liczb na litery

```
int2let :: Int -> Char
int2let n = chr (ord 'a' + n)
```

- Zamiana liter na liczby

```
let2int :: Char -> Int
let2int c = ord c - ord 'a'
```

- Zamiana liczb na litery

```
int2let :: Int -> Char
int2let n = chr (ord 'a' + n)
```

- Przykłady

```
> let2int 'a'
0

> int2let 0
'a'
```

- Zamiana liter na liczby

```
shift :: Int -> Char -> Char
shift n c
 |isLower c = int2let((let2int c+n) `mod` 26)
 |otherwise = c
```

- Zamiana liter na liczby

```
shift :: Int -> Char -> Char
shift n c
 |isLower c = int2let((let2int c+n) `mod` 26)
 |otherwise = c
```

- Przykłady

```
> shift 3 'a'
'd'

> shift 3 'z'
'c'

> shift (-3) 'c'
'z'

> shift 3 ' '
' '
```

- Kodowanie

```
encode :: Int -> String -> String
encode n s  = [shift n x | x <- s]
```

- Kodowanie

```
encode :: Int -> String -> String
encode n s  = [shift n x | x <- s]
```

- Przykłady

```
> encode 3 "haskell is fun"
"kdvnhoo lv ixq"

> encode (-3) "kdvnhoo lv ixq"
"haskell is fun"
```

- Tabela częstości (język angielski i polski)

```
table :: [Float]
table = [8.2, 1.5, 2.8, 4.3, 12.7,
         2.2, 2.0, 6.1, 7.0, 0.2,
         0.8, 4.0, 2.4, 6.7, 7.5,
         1.9, 0.1, 6.0, 6.3, 9.1,
         2.8, 1.0, 2.4, 0.2, 2.0,
         0.1]

tablePol :: [Float]
tablePol = [8.91, 1.47, 3.96, 3.25, 7.66,
            0.30, 1.42, 1.08, 8.21, 2.28,
            3.51, 2.10, 2.80, 5.52, 7.75,
            3.13, 0.14, 4.69, 4.32, 3.98,
            2.50, 0.04, 4.65, 0.02, 3.76,
            5.64]
```

- Względny procent

```
percent :: Int -> Int -> Float
percent n m = (a/b) * 100
    where
    a = fromIntegral n :: Float
    b = fromIntegral m :: Float
```

Tabele częstości

- Względny procent

```
percent :: Int -> Int -> Float
percent n m = (a/b) * 100
    where
    a = fromIntegral n :: Float
    b = fromIntegral m :: Float
```

- Tabela częstości dla konkretnej listy

```
frequency :: String -> [Float]
frequency xs = [percent(count x xs) n |
                                x<-['a'..'z']]
  where
   n = length [x | x <- xs, isLower x]

> frequency "abbcccddddeeeeee"
[6.666667,13.333334,20.0,26.666668,
33.333336,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0]
```

- chi-square statictics

$$\sum_{i=0}^{n-1} \frac{(os_i - oe_i)^2}{es_i}$$

- chi-square statictics

$$\sum_{i=0}^{n-1} \frac{(os_i - oe_i)^2}{es_i}$$

- Implementacja

```
chisqr :: [Float] -> [Float] -> Float
chisqr os es = sum [((o - e)^2)/e |
         (o,e) <- zip os es]
```

Odkodowanie szyfru

- Rotacje

```
rotate :: Int -> [a] -> [a]
rotate n xs = drop n xs ++ take n xs

> rotate 3 [1,2,3,4,5]
[4,5,1,2,3]
```

- Łamanie szyfru

```
crack :: String -> String
crack xs = encode (-factor) xs
where
 factor=head(positions(minimum chitab)chitab)
 chitab=[chisqr (rotate n table') table |
              n <- [0..25]]
 table'=frequency xs
```

Zadania

1. Używając wyróżniania (list comprehension) zdefiniuj wyrażenie obliczające sumę $1^2 + 2^2 + \ldots + 100^2$ (pierwszych stu liczb naturalnych).
2. Inspirując się definicją funkcji length, zdefiniuj funkcję replicate, produkującą listę identycznych elementów.

```
> replicate 3 True
[True,True, True]
```

3. Trójkę liczb naturalnych (x, y, z) nazywamy *pitagorejską*, jeśli $x^2 + y^2 = z^2$. Używając wyróżniania, zdefiniuj funkcję:

```
pyths :: Int -> [(Int, Int, Int)]
```

zwracającą listę wszystkich pitagorejskich trójek, do pewnego limitu.

```
> pyths 10
[(3,4,5), (4,3,5), (6,8,10), (8,6,10)]
```

4. Liczba naturalna jest *doskonała*, jeśli jest równa sumie swoich dzielników (z wyłączeniem siebie samej). Używając wyróżniania oraz funkcji factor zdefiniuj funkcję:

```
perfects :: Int -> [Int]
```

   zwracającą listę wszystkich liczb doskonałych, do pewnego limitu.

5. Pokaż w jaki sposób wyrażenie (z jednym wyróznieniem i dwoma generatorami):

```
[(x,y) | x <- [1,2,3], y <- [4,5,6]]
```

   może być zdefiniowane przy użyciu dwóch wyróznień i jednego generatora (list comprehensions można zagnieżdżać w sobie).

6. Zdefiniuj ponownie funkcję positions uzywając funkcji find.

7. *Produktem skalarnym* dwóch list liczb naturalnych o długości n jest suma iloczynów odpowiadających sobie liczb, i.e.,

$$\sum_{i=0}^{n-1}(xs_i \cdot ys_i)$$

   Używając wyróżniania, zdefiniuj funkcję scalarProduct

```
scalarProduct :: [Int] -> [Int] -> Int

> scalarProduct [1, 2, 3] [4, 5, 6]
32
```

8. Zmodyfikuj program do deszyfrażu szyfrów Cezara w taki sposób, aby działał również dla wielkich liter.