Operacje na listach

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

map

Funkcja `map` o sygnaturze

```
map :: (a -> b) -> [a] -> [b]
```

przyjmuje funkcję f oraz listę $(x_1, \ldots, x_n)$ i zwraca listę $(f(x_1), \ldots, f(x_n))$. Przykładowo:

```
map (10+) [1..10]
map (10*) [1..10]
map (\x->x*x) [1..10]
```

zwrócą kolejno

```
[11,12,13,14,15,16,17,18,19,20]
[10,20,30,40,50,60,70,80,90,100]
[1,4,9,16,25,36,49,64,81,100]
```

filter

Funkcja `filter` o sygnaturze

```
filter :: (a -> Bool) -> [a] -> [a]
```

przyjmuje funkcję f zwracającą wartość logiczną oraz listę $(x_1, \ldots, x_n)$ i zwraca listę tylko tych elementów z oryginalnej listy, na których f zwróciła prawdę. Przykładowo:

```
filter (odd) [1..10]
filter (\x->x `mod` 3 == 1) [1..10]
filter (>=5) [1..10]
```

zwrócą kolejno

```
[1,3,5,7,9]
[1,4,7,10]
[5,6,7,8,9,10]
```

filter na listach nieskończonych

Używając filter możemy działać na listach nieskończonych, na przykład `filter (even) [1..]` jest poprawną listą nieskończoną, na której możemy operować; przykładowo `take 10 (filter (even) [1..])` zwróci poprawnie `[2,4,6,8,10,12,14,16,18,20]`. Problematyczne jednak mogą być wywołania typu `filter (<5) [1..]`. W przypadku list nieskończonych Haskell oblicza kolejny element dopiero wtedy, kiedy musi, cztery pierwsze elementy wyniku to więc 1, 2, 3 i 4. Jeśli jednak będziemy chcieli skorzystać z piątego elementu listy, Haskell nigdy go nie obliczy (choć będzie próbować), ponieważ będzie przeszukiwał kolejne liczby całkowite (i nigdy nie znajdzie kolejnej, która spełnia warunek).

takeWhile/dropWhile

Przydatną funkcją w sytuacji powyżej może być `takeWhile`, która pobiera elementy z listy, póki spełniają podany warunek.
Przykładowo:

```
takeWhile (<10) [1..]
```

poprawnie zwróci skończoną listę

```
[1,2,3,4,5,6,7,8,9]
```

Analogiczna funkcja `dropWhile` działa podobnie, jednak ignoruje elementy zamiast je pobierać, na przykład

```
dropWhile (<5) [1..10]
```

zwraca

```
[5,6,7,8,9,10]
```

concatMap

Funkcja `concatMap` działa podobnie do `map`, jednak przyjmuje listę list, po zastosowaniu na każdej z nich podanej funkcji scala wyniki w jedną listę. Przykładowo:

```
concatMap (take 2) ["Ala","ma","kota"]
```

zwraca

```
"Almako"
```

iterate

Funkcja `iterate` o sygnaturze

```
iterate :: (a -> a) -> a -> [a]
```

przyjmuje funkcję f oraz element x, na którym można wywołać funkcję f i zwraca nieskończoną listę $(x, f(x), f(f(x)), \ldots)$. Przykładowo `iterate (+1) 0` zwróci nieskończoną listę wszystkich liczb naturalnych (od 0).

zip

Funkcja `zip` o sygnaturze

```
zip :: [a] -> [b] -> [(a, b)]
```

przyjmuje dwie listy i zwraca listę par postaci `(<elementLewej>,<elementPrawej>)`. Przykładowo

```
zip [1,2,3] "abc"
```

zwraca

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

Jeśli listy nie są równej długości, elementy dłuższej "bez pary" z elementem listy krótszej są pomijane.

zipWith

Funkcja `zipWith` o sygnaturze

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

przyjmuje funkcję oraz dwie listy i zwraca listę elementów postaci $f(x, y)$, gdzie f jest podaną funkcją, a x oraz y to kolejne elementy list. Przykładowo

```
zipWith (+) [1,2,3] [4,5,6]
```

zwraca

```
[5,7,9]
```

Jeśli listy nie są równej długości, elementy dłuższej "bez pary" z elementem listy krótszej są pomijane.