Operacje na listach c.d.

zadania

Zadanie 1

Podać przykład działania, wartości początkowej i listy, gdzie wywołanie `foldl` i `foldr` da różne wyniki. Rozpisać, jak wygląda obliczenie w obu przypadkach.

Zadanie 2

Zaimplementować `reverse` na liście używając `foldl` i `foldr` (po jednej definicji; można założyć, że listy są skończone).

Zadanie 3

Zaimplementować `map` używając `foldl` i `foldr` (po jednej definicji; można założyć, że listy są skończone).
Uwaga: Zauważmy, że w drugą stronę (fold za pomocą map) nie jest to wykonalne, ponieważ `map` zwraca nam zawsze listę.

Zadanie 4

Zdefiniować używając `foldl` lub `foldr` funkcję

```
doDziesietnego :: Int -> [Int] -> Int
```

przyjmującą podstawę systemu liczbowego oraz reprezentację liczby w tym systemie (kolejne wyrazy to kolejne "cyfry") i zwracającą jej reprezentację w systemie dziesiętnym. Przykładowo `doDziesietnego 2 [1,0,1,1]` powinno zwrócić `11`, a `doDziesietnego 16 [15,15]` powinno zwrócić 255.

Zadanie 5

Zaimplementować używając `foldl` lub `foldr` funkcję

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

przyjmującą listę punktów na płaszczyźnie i zwracającą numer ćwiartki, w której jest najwięcej punktów (w przypadku remisu dowolną ćwiartkę, w której jest najwięcej; to zadanie było już na drugich ćwiczeniach, jednak bez użycia fold).

Zadanie 6

Napisać funkcję

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

przyjmującą warunek (funkcję jednoargumentową) oraz listę, i zwracającą prawdę, jeśli warunek jest spełniony dla każdego elementu listy oraz fałsz w przeciwnym przypadku. Zdefiniować też funkcję

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

przyjmującą warunek (funkcję jednoargumentową) oraz listę, i zwracającą prawdę, jeśli warunek jest spełniony dla przynajmniej elementu listy oraz fałsz w przeciwnym przypadku.
W obu przypadkach można założyć, że listy są skończone.

Zadanie 7

Rozważmy poniższą definicję:

```
fibonacci :: [Integer]
fibonacci = 0 : 1 : (zipWith (+)
                fibonacci (tail fibonacci))
```

Wytłumaczyć, dlaczego poprawnie oblicza ona ciąg Fibonacciego (wartością jest istotnie nieskończona lista zawierająca wszystkie jego elementy).
Wskazówka: Dla list nieskończonych Haskell liczy kolejny element dopiero wtedy, kiedy musi.