Kurs języka Haskell 2024/25

LISTA NR 2 (TERMIN: 19.10.2024, godz 5:00)

Uwaga: Wszystkie rozwiązania należy umieścić w jednym module o nazwie `Lista2` zachowując sygnatury zgodne z szablonem rozwiązań zamieszczonym w SKOS-ie.

Zadanie 1. (2 pkt) W tym zadaniu należy zaimplementować uproszczony algorytm szyfrujący i deszyfrujący Enigmy:

Konfiguracja Enigmy składa się z pewnej liczby wirników ustawionych w kolejności. Każdy z wirników implementuje permutację, która zmienia pojedynczy znak wejściowy na inny znak. By uzyskać ostateczny zaszyfrowany znak, należy „przepuścić” znak wejściowy przez wszystkie wirniki zgodnie z kolejnością. W tym zadaniu będziemy reprezentować permutację implementowaną przez wirnik jako listę zawierającą znaki ['A'..'Z']. Przykładowo, załóżmy, że w konfiguracji są dwa wirniki, które implementują następujące permutacje:

	ABCDEFGHIJKLMNOPQRSTUVWXYZ
1	CKMFLGDQVZNTOWYHXUSPAIBREJ
2	JPGVOUMFYQBENHZRDKASXLICTW

Oznacza to, że znak 'A' zmieniany jest przez pierwszy wirnik na znak 'C', który zmieniany jest przez drugi wirnik na znak 'G'.

Komplikacja polega na tym, że z każdym kolejnym zaszyfrowanym znakiem niektóre wirniki obracają się w określony sposób o jeden ząbek. Obrót wirnika oznacza, że permutacja przemieszcza się cyklicznie o jeden znak. Np. pierwsza permutacja zmienia się z

CKMFLGDQVZNTOWYHXUSPAIBREJ

na

KMFLGDQVZNTOWYHXUSPAIBREJC

Zmiany wirników definiowane są przez same wirniki: z każdym wirnikiem związany jest zbiór symboli takich, że gdy w czasie obrotu symbol opuszcza pierwszą pozycję w permutacji, obracany jest też kolejny wirnik. Jeśli kolejny wirnik akurat przesunął się z ustawienia, na którym jeden z jego wybranych symboli był na pierwszej pozycji, przesuwany jest też kolejny wirnik, i tak dalej. Dla przykładu załóżmy, że nasze wirniki wyposażone są w następujące zestawy symboli:

	ABCDEFGHIJKLMNOPQRSTUVWXYZ	
1	CKMFLGDQVZNTOWYHXUSPAIBREJ	K, M
2	JPGVOUMFYQBENHZRDKASXLICTW	J

Po zaszyfrowaniu pierwszego znaku, obraca się tylko pierwszy wirnik i układ zmienia się na:

1	KMFLGDQVZNTOWYHXUSPAIBREJC
2	JPGVOUMFYQBENHZRDKASXLICTW

Po zaszyfrowaniu drugiego znaku, pierwszy wirnik obraca się jak zawsze, ale tym razem obraca się i drugi wirnik, bo znak 'K' jest z biorze pierwszego wirnika. Układ zmienia się więc na:

1	MFLGDQVZNTOWYHXUSPAIBREJCC
2	PGVOUMFYQBENHZRDKASXLICTWJ

Ponieważ znak 'J' jest w zbiorze drugiego wirnika, gdybyśmy mieli trzeci wirnik, i on obróciłby się w tym momencie. Jak widać, algorytm obrotu wirników nie zależy od tego, jaki znak był szyfrowany.

Zadanie: Definiujemy typ reprezentujący wirniki jako:

```
data Rotor = Rotor { wiring   :: [Char]
                   , turnover :: [Char] }
```

przy czym zakłądamy, że wiring to opisana wyżej permutacja alfabetu. Zdefiniuj funkcje

```
encode :: [Rotor] → String → String
decode :: [Rotor] → String → String
```

które szyfrują i deszyfrują wiadomości złożone ze znaków ze zbioru ['A'..'Z'] (bez spacji i innych znaków). Przykładowo, dla

```
rotors :: [Rotor]
rotors =
  [ Rotor "ABCDEFGHIJKLMNOPQRSTUVWXYZ" "C"
  , Rotor "ABCDEFGHIJKLMNOPQRSTUVWXYZ" "" ]
```

uzyskujemy:

```
ghci> encode rotors "AAAAAA"
"ABCEFG"
```

(Widzimy w szyfrogramie, jak pierwszy wirnik przesuwał się zmieniając znak 'A' na kolejne litery alfabetu. Po zaszyfrowaniu trzeciego znaku, przesuwa się też drugi wirnik, przez co z szyfrogramu „wypada” znak 'D'.)

Wskazówka nr 1: Ponieważ algorytm obracania wirników jest niezależny od szyfrowanej wiadomości, rozwiązanie można podzielić na dwa etapy: najpierw wygenerować listę wszystkich kolejnych konfiguracji (lista taka musi mieć długość równą przynajmniej długości szyfrowanej wiadomości, ale chyba łatwiej w tym wypadku jest wygenerować listę nieskończoną, np. używając funkcji z Zad 2.), a dopiero potem przetwarzać kolejne znaki przy użyciu kolejnych konfiguracji. Oczywiście jest to tylko jedno z bardzo wielu możliwych rozwiązań.

Wskazówka nr 2: Bardzo dużo przydatnych funkcji można znaleźć w module Data.**List**. Przykłady funkcji użytych przez prowadzącego w jego rozwiązaniu to **zip**, zipWtih, **lookup**, **cycle**. (W funkcjach **zip** i zipWtih roszę zwrócić uwagę, jak zachowują się dla list różnej długości, w szczególności, gdy jedna z list jest nieskończona.)

Zadanie 2. Zdefiniuj funkcję

```
unfoldStream :: (seed → (seed, val)) → seed → [val]
```

w której pierwszy argument jest funkcją, której argumentem jest ziarno, a wynikiem nowe ziarno i kolejny element produkowanej listy. Drugim argumentem jest początkowe ziarno. Na przykład

```
unfoldStream (λn → (n+1, n)) 0
```

jest listą wszystkich liczb naturalnych, a

```
unfoldStream (λ(p,q) → ((q, q+p), p)) (0,1)
```

jest listą wszystkich liczb Fibonacciego.

Wskazówka: Zadanie 2. można rozwiązać używając funkcji unfoldStream.

Zadanie 3. Zdefiniuj wartość

```
pascal :: [[Integer]]
```

reprezentującą (nieskończony) trójkąt Pascala:

```
ghci> take 6 pascal
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],[1,5,10,10,5,1]]
```

Zadanie 4. Rozważmy drzewa binarne bez etykiet zdefiniowane jako:

```
data Tree = Leaf | Node Tree Tree

instance Show Tree where
  show Leaf       = "."
  show (Node l r) = "(" ++ show l ++ show r ++ ")"
```

Zdefiniuj wartość

```
trees :: [Tree]
```

będącą listą wszystkich skończonych drzew typu Tree. Przykładowo:

```
ghci> take 8 trees
[.,(..),(.(..)),((..).),(.(.(..))),(.((..).)),((..)(..)),((.(..)).)]
```

Wskazówka: Jedno z możliwych rozwiązań to najpierw zdefiniowanie funkcji

```
treesN :: Integer → [Tree]
```

która generuje wszystkie drzewa o n węzłach wewnętrznych, a potem użycie standardowej funkcji

```
concatMap :: (a → [b]) → [a] → [b]
```

na liście wszystkich liczb naturalnych.

Zadanie 5. Listę podwójnie łączoną (czyli taką, która ma wskaźniki na następny i na poprzedni element) można wyrazić następującym typem:

```
data DList a = DCons { val  :: a
                     , prev :: Maybe (DList a)
                     , next :: Maybe (DList a) }
```

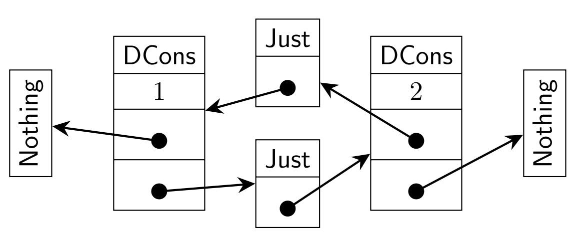
Zdefiniuj funkcję

```
toDList :: [a] → Maybe (DList a)
```

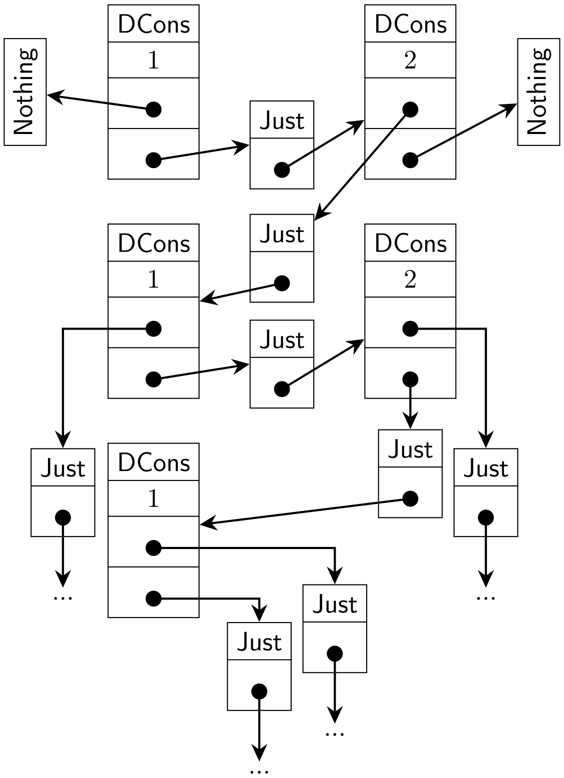
która:

- Dla listy pustej zwraca **Nothing**,
- Dla list niepustej zwraca jej odpowiednik w postaci listy podwójnie łącznej owinięty w konstruktor **Just**.

Zadbaj, by w pamięci wskaźniki tworzyły cykl. Np. toDList [1,2] powinna wyglądać tak:



a nie być nieskończonym drzewem w stylu:



Wskazówka nr 1: Jak zrobić cykl w pamięci? Np. listę `[1,2,1,2,1,2,1,...]` można stworzyć w pamięci jako leniwie rozwijającą się, nieskończoną listę, albo jako listę zacykloną:

```
lazy, cycleA, cycle B :: [Int]

-- leniwa nieskonczona lista
lazy = aux 1
  where aux n = n : aux ((n+1) `mod` 2)

-- cykl w pamieci (A)
cycleA = let x = 1 : 2 : x in x

 -- cykl w pamieci (B)
cycleB = 1 : 2 : cycleB
```

Oczywiście definicje `A` i `B` są równoważne, ale w tym zadaniu raczej przyda się **let** – proszę zwrócić uwagę, że gdyby `cycleB` miało argument, już nie powstałby cykl, nawet gdyby wywoływało się z tym samym argumentem (chyba, że wkroczyłby kompilator z jakąś optymalizacją).

Wskazówka nr 2: Jak zbadać, jaki jest kształt struktury powstałej w pamięci? Semantycznie, obie powyższe struktury niczym się nie różnią. Możemy zbadać sytuację mierząc zużycie pamięci. W GHCi możemy poprosić o to, by zawsze razem z obliczoną wartością wyrażenia, w odpowiedzi REPL-a pojawiła się też informacja o pamięci zaalokowanej w trakcie obliczania wartości zapytania:

```
ghci> :set +s
ghci> length [1..1000000000]
1000000000
(17.33 secs, 72,000,071,440 bytes)
```

Oczywiście jest to alokacja sumaryczna, więc powyższa liczba nie oznacza, że w którymkolwiek momencie powyższy program zajmował 72GB.

Żeby zobaczyć, czy wyprodukowana podwójnie łączona lista jest cyklem czy drzewem, musimy przejść się po niej w obie strony kilka razy i zobaczyć czy alokujemy pamięć (rozwijając leniwie nieskończone drzewo) czy chodzimy wskaźnikiem po istniejącym w pamięci cyklu:

```
bounce :: Int → DList a → a
bounce = go prev next where
  go dir anty 0 ds = val ds
  go dir anty n ds = case dir ds of
                       Just ds'  → go dir  anty (n−1) ds'
                       Nothing → go anty dir  n      ds
```

Niestety, nie ma sensu uruchamiać tej funkcji bezpośrednio w REPL-u, bo GHCi zawsze dodaje do kodu informacje potrzebne do debugowania, które alokują pamięć przy każdym wywołaniu funkcji, więc nie otrzymalibyśmy wiarygodnego wyniku. Z tego powodu należy naszą funkcję najpierw skompilować i dopiero tak skompilowany moduł załadować. Na szczęście wszystko to można zrobić z poziomu GHCi:

```
ghci> :! ghc −O2 −c −dynamic Lista2.hs
ghci> :l Lista2
ghci> :m +Data.Maybe
ghci> bounce 10000000 $ fromJust $ toDList [1..10]
9
(0.12 secs, 66,424 bytes)
```

Jeśli wynik oscyluje w kilobajtach, to mamy cykl. Jeśli w gigabajtach – mamy nieskończone drzewo.