Kurs języka Haskell 2024/25

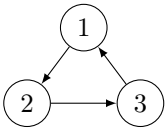
LISTA NR 4 (TERMIN: 9.11.2024, godz 5:00)

Uwaga: Wszystkie rozwiązania należy umieścić w jednym module o nazwie `Lista4` zachowując sygnatury zgodne z szablonem rozwiązań zamieszczonym w SKOS-ie.

Zadanie 1. Rozważ dwie reprezentacje grafów skierowanych z (unikatowymi) etykietami w węzłach. Pierwsza reprezentacja to:

```
type FlatGraph a = [(a, [a])]
```

Graf w tej reprezentacji to lista wierzchłków. Każdy wierzchołek to para składająca się z identyfikatora i listy identyfikatorów wierzchołków, do których istnieją krawędzie z tego wierzchołka. Przykładowo, graf



można zareprezentować następującą wartością:

```
g1 :: FlatGraph Int
g1 =   [(1,[2]),(2,[3]),(3,[1])]
```

Drugą reprezentacją jest

```
data Node a = Node { lbl :: a, ns :: [Node a] }
type Graph a = [Node a]
```

Ten sam graf można zareprezentować jako:

```
g2 :: Graph Int
g2 = let n1 = Node 1 [n2]
         n2 = Node 2 [n3]
         n3 = Node 3 [n1]
     in [n1, n2, n3]
```

Zdefiniuj funkcje:

```
makeGraph   :: (Eq a) ⇒ FlatGraph a → Graph a
flattenGraph :: (Eq a) ⇒ Graph a → FlatGraph a
```

które konwertują pomiędzy dwoma reprezentacjami. Zadbaj, żeby makeGraph tworzył cykle w pamięci, a nie nieskończone struktury.

Zadanie 2. Na wykładzie widzieliśmy, że strzałka jest funktorem w swoim drugim (wyjściowym) argumencie. A co z funkjami w stylu kontynuacyjnym? Pokaż, że

```
newtype CPS o a = CPS { run :: (a → o) → o }
```

jest funktorem.

Zadanie 3. Rozważ typ funktorów **kontrawariantnych**, czyli takich, które odwracają kierunek strzałki:

```
class CoFunctor f where
  cofmap :: (a → b) → f b → f a
```

Zainstaluj typ

```
newtype Predicate a = Predicate (a → Bool)
```

w klasie CoFunctor.

Przetestuj rozwiązanie na następującym przykładzie. Rozważ poniższe dwie definicje:

```
filterPred :: Predicate a → [a] → [a]
filterPred (Predicate p) = filter p

hasThreeChars :: Predicate String
hasThreeChars = Predicate (λx → length x == 3)
```

Użyj filterPred , hasThreeChars i cofmap, żeby pozostawić na liście tylko te liczby, które mają w zapisie dziesiętnym trzy cyfry, np.

```
ghci> filterPred ??? [1,12,123,1234,12345,321,21]
[123,321]
```

Zadanie 4. W tym zadaniu rozważamy termy nad sygnaturą. Przypomnijmy, że tradycyjnie sygnaturę definiuje się jako zbiór symboli funkcyjnych wraz z ich arnością. Natomiast w Haskellu za sygnaturę posłuży nam dowolny funktor.

Przykładowo, sygnaturę dla wyrażeń w logice zdaniowej można zdefiniować jako:

$$\{\top^{(0)}, \bot^{(0)}, \vee^{(2)}, \wedge^{(2)}, \neg^{(1)}\}$$

W Haskellu możemy ją zareprezentować jako następujący typ danych:

```
data BoolSig a = Top | Bot | Or a a | And a a | Neg a
  deriving (Functor)
```

Termy nad sygnaturą sig ze zmiennymi pochodzącymi ze „zbioru" x możemy zdefiniować przy użyciu następującego typu danych:

```
data Term sig x = Var x
                | Op (sig (Term sig x))
```

Przykładowo, formuła $x \wedge (\top \vee \neg y)$ reprezentowana jest przez następującą wartość typu Term BoolSig **String**

```
Op (And (Var "x") (Op (Or (Op Top) (Op (Neg (Var "y"))))))
```

W tym zadaniu:

- Zainstaluj Term sig w klasie **Functor** (dla dowolnego funktora sig)
- Zdefiniuj następujące funkcje:

```
var :: x → Term sig x

subst :: Functor sig
      ⇒ Term sig x
      → (x → Term sig y)
      → Term sig y

foldTerm :: Functor sig
         ⇒ (sig a → a)
         → (x → a)
         → Term sig x
         → a
```

  gdzie:
  - var to funkcja awansująca zmienną do termu składającego się z pojedynczej zmiennej,
  - subst to funkcja podstawiająca za zmienne termy (być może ze zmiennymi pochodzącymi z innego zbioru),
  - foldTerm to homomorficzna interpretacja termu, której dostarczamy interpretację operacji w sygnaturze (czyli tzw. algebrę) i interpretację zmiennych, a w zamian dostajemy wyliczoną wartość termu.

Czy typy funkcji var i subst mają znajomy kształt (być może znajomy po wykładzie nr 5)?

Zadanie 5. (na podstawie dyskusji w czasie wykładu) Zdefiniuj klasę Applicative f ⇒ ApplicativeError f zawierającą metodę err :: **String** → f a, która umożliwia funkcji korzystającej z efektów w klasie ApplicativeError zgłoszenie błędu parametryzowanego komunikatem.

Zainstaluj w klasie ApplicativeError konstruktory typów **Maybe** i **Either String**. Niech **Maybe** ignoruje wartość komunikatu.

Przerób funkcję eval z wykładu tak, by miała typ

```
eval :: ( ApplicativeError f) ⇒ Expr → f Int
```

Przykładowo:

```
ghci> let ex1 = Op "+" (Op "*" (Val 2) (Val 2)) (Val 3)
ghci> eval ex1 :: Maybe Int
Nothing
ghci> eval ex1 :: Either String Int
Left "*"
```

Zadanie 6. (2 pkt) Rozważmy typ danych, który użyjemy do zaimplementowania interaktywnej obsługi błędów w ewaluatorze z wykładu. Tym razem, gdy ewaluator napotka na nieznany sobie operator, spyta o jego definicję użytkownika. Nośnikiem naszego efektu będzie następujący typ:

```
data Interactive
  = Done Int
  | NeedMoreInfo
      { operator :: String
      , continue :: (Int → Int → Int) → Interactive }

instance Show Interactive where
 show (Done n)              = "Done: " ++ show n
 show (NeedMoreInfo s _) = "Need more info on " ++ s
```

Zaimplementuj wersję ewaluatora tak, by miał typ

```
evalI :: Expr → Interactive
```

Wyrażenie powinno się obliczać do Done, jeśli znajdują się w nim tylko operatory znane interpreterowi. Jeśli interpreter napotka nieznany symbol, powininen zwrócić wartość NeedMoreInfo s k, gdzie s to nieznany operator, a k to kontynuacja obliczenia, która oczekuje interpretacji nieznanego operatora. (Nie trzeba spamiętywać odpowiedzi: jeśli ten sam nieznany operator występuje w wyrażeniu wiele razy, niech ewaluator pyta o jego interpretację za każdym razem.)

W ten sposób można przeprowadzić z ghci dialog:[1]

```
ghci> exI = Op "+" (Op "^&" (Val 2) (Val 10))
                   (Op "%%" (Val 4) (Val 3))
ghci> evalI exI
Need more info on ^&
ghci> continue it (*)
Need more info on %%
ghci> continue it (−)
Done: 21
```

[1] W ghci zmienna it przechowuje wynik poprzedniego zapytania.

Czy rozumiesz, czemu Interactive nie może być prawdziwym funktorem aplikatywnym?

Uwaga: Dwa punkty przydzielone zostaną eleganckim rozwiązaniom, w których kod evalI ma kształt podobny do eval z wykładu, który, przypomnijmy, korzysta z funktora aplikatywnego. Co prawda Interactive nie jest funktorem aplikatywnym, ale można dla niego zdefiniować funkcje odpowiadające metodom klasy Applicative.

Wskazówka: Przyjrzyj się metodzie liftA2 z klasy Applicative.