Kurs języka Haskell 2024/25

LISTA NR 5 (TERMIN: 18.11.2024, godz 5:00)

Uwaga: Wszystkie rozwiązania należy umieścić w jednym module o nazwie `Lista5` zachowując sygnatury zgodne z szablonem rozwiązań zamieszczonym w SKOS-ie.

Zadanie 1. Zdefiniuj funkcję

```
combineErrors :: (Semigroup e)
 ⇒ Either e a → Either e b → Either e (a, b)
```

która pozwala kumulować błędy z dwóch różnych obliczeń. Zdefiniuj funkcję

```
eval :: Expr → Either [String] Int
```

która oblicza wartość wyrażenia kumulując wszystkie błędy, które wystąpiły w trakcie obliczania wartości wyrażenia.

Zadanie 2. (2 pkt) W tym zadaniu zaimplementujemy grę w kółko i krzyżyk w trybie hot seat. Zadaniem jest zdefiniowanie reprezentacji planszy, kilku funkcji zmieniającej stan gry i samej gry. Plansza składa się z dziewięciu pól, do których gracz może odnieść się przez ich indeksy:

```
0|1|2
-----
3|4|5
-----
6|7|8
```

Zdefiniuj:

```
data Symbol = X | O
data Board = ...
instance Show Board where ...
emptyBoard :: Board
draw :: Board → Bool
wins :: Board → Maybe Symbol
put :: Symbol → Int → Board → Maybe Board
play :: Symbol → Board → IO ()
```

gdzie:

- `emptyBoard` to pusta plansza
- `draw` ujawnia, czy w danym stanie planszy jest remis (plansza jest pełna, ale nikt nie wygrał)
- `wins` ujawnia, kto wygrywa w danym stanie planszy (**Nothing**, gdy nikt nie wygrywa)
- `put` wstawia symbol na danym polu na planszy reprezentowanym przez indeks (**Nothing**, gdy indeks jest nieprawidłowy lub pole jest już zajęte)
- `play` implementuje rozgrywkę (pierwszy argument to symbol kolejnego gracza i opcząkowy stan planszy): Gracze na zmianę wpisują indeksy pól, na których stawiają swój symbol. Co rundę system powinien wyświetlać planszę na ekranie i informować, do którego gracza należy następny ruch, np.

```
0|1|2
-----
3|4|5
-----
6|7|8

X's move: 0
X|1|2
-----
3|4|5
-----
6|7|8

O's move: 4
X|1|2
-----
3|O|5
-----
6|7|8

X's move: 3
X|1|2
-----
X|O|5
-----
6|7|8

O's move: 8
X|1|2
-----
X|O|5
-----
6|7|O

X's move: 6
X|1|2
-----
X|O|5
-----
X|7|O

X wins!!
```

Wskazówka: Żeby spłukać wyjście standardowe, można użyć **hFlush** z modułu **System.IO** dla uchwytu wyjścia standardowego **stdout** (również z modułu **System.IO**).

Zadanie 3. Rozważ typ drzew, w których węzły etykietowane są akcjami wejścia/wyjścia typu boolowskiego:

```
data IOTree a = Leaf a
              | Node (IO Bool) (IOTree a) (IOTree a)
```

Zdefiniuj funkcje

```
runTree :: IOTree a → IO a
yesNo :: String → IO Bool
```

takie, że

- runTree t przechodzi drzewo t „wewnątrz" monady **IO** od korzenia do liścia, zgodnie z zasadą z Zad. 3 z Listy 3: jeśli akcja zwraca **True**, idziemy w lewo, jeśli **False** – w prawo.
- yesNo s tworzy akcję, która:
  1. Wypisuje komunikat s na wyjście standardowe
  2. Wczytuje odpowiedź użytkownika `"y"` lub `"n"` (możesz użyć np. **getLine**)
  3. Zwraca **True** jeśli użytkownik odpowiedział `"y"` i **False** w p.p.

Przykładowo:

```
system :: IOTree String
system =
  Node (yesNo "Czy chcesz wydac pieniadze?")
    (Node (yesNo "Czy chcesz ogladac reklamy?")
      (Leaf "Windows")
      (Leaf "MacOS"))
    (Node (yesNo "Czy umiesz zrobic 'mount'?")
      (Leaf "Arch")
      (Leaf "Ubuntu"))
```

Następnie:

```
ghci> runTree system
Czy chcesz wydac pieniadze? (y/n):
n
Czy umiesz zrobic 'mount'? (y/n):
y
"Arch"
```

Zadanie 4. Przypomnij sobie typ Term z Listy 4:

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

Zainstaluj go w klasie **Monad** (przypomnij sobie funkcje var i subst).

Rozważmy trochę inną implementację monady opisującej wejście/wyjście. Niech będzie to Term MyIOSig:

```
data MyIOSig a = PutStr String a
               | GetLine (String → a)
  deriving (Functor)

type MyIO = Term MyIOSig
```

Zdefiniuj odpowiednie wersje funkcji **putStr** i **getLine**:

```
myPutStr :: String → MyIO ()
myGetLine :: MyIO String
```

Zdefiniuj funkcję

```
toRealIO :: MyIO a → IO a
```

która konwertuje program w naszej monadzie do prawdziwego **IO**.

Morał: Cały program, który kontaktuje się z zewnętrznym światem, można opisać przy pomocy monady MyIO (a więc instancji monady Term), która jest zwykłym algebraicznym typem danych.

Zadanie 5. Innym sposobem opisywania wejścia/wyjścia (obowiązującym przed monadycznym IO) jest zastosowanie strumieni zapytań i odpowiedzi. Sposób ten bazuje na leniwości. Zapytania, odpowiedzi i typ dla funkcji z wejściem/wyjściem definiujemy następująco:

```
data Request = PutStr String
             | GetLine

data Response = Unit
              | Value String

type DialogueIO = [Response] → [Request]
```

Wówczas funkcja, który ma wejście/wyjście (np. main) dostaje jako argument strumień odpowiedzi, a oblicza się do strumienia zapytań. Ważne, by funkcja produkowała zapytanie zanim nastąpi wymuszenie odpowiedzi, np.:

```
echo :: DialogueIO
echo ~(Value s : _) = [GetLine, PutStr s]
```

Zdefiniuj funkcję

```
dialogueToIO :: DialogueIO → IO ()
```

która konwertuje funkcję z wejściem/wyjściem opartym na strumieniach zapytań i odpowiedzi na monadyczne IO.