Programowanie Funkcyjne 2025

Lista zadań nr 8

Na zajęcia 11 i 19 grudnia 2025

Zadanie 1 (1p). Zaimplementuj funkcję `echoLower :: IO ()` która będzie wczytywać znaki ze standardowego wejścia, zamieniać wielkie litery na małe i wypisywać z powrotem na standardowe wyjście, aż do napotkania końca strumienia wejściowego.

Zadanie 2 (3p). Funkcja z poprzedniego zadania jest przykładem transformatora strumieni: transformuje jeden strumień (standardowe wejście) na drugi (standardowe wyjście). Znów oddzielimy składnię od semantyki i zareprezentujemy transformatory strumieni za pomocą następującego algebraicznego typu danych.

```
data StreamTrans i o a
  = Return a
  | ReadS (Maybe i -> StreamTrans i o a)
  | WriteS o (StreamTrans i o a)
```

Typ ten reprezentuje całe obliczenia z użyciem dwóch strumieni: wejściowego i wyjściowego. Wartość `WriteS x c` oznacza wypisanie wartości `x` do strumienia wyjściowego i kontynuowanie obliczenia `c`. Wartość `ReadS f` oznacza próbę czytania ze strumienia wejściowego, gdzie funkcja `f` jest kontynuacją dla wyniku czytania. Jeśli zostanie przeczytana wartość `x`, to dalej zostanie wykonane obliczenie `f (Just x)`. Przeczytanie `Nothing` oznacza koniec strumienia wejściowego. W końcu `Return x` oznacza zakończenie obliczenia z wartością `x`.

Zdefiniuj transformator strumieni `toLower :: StreamTrans Char Char ()` podobny to tego z poprzedniego zadania. Następnie zdefiniuj funkcję

```
runIOStreamTrans :: StreamTrans Char Char a -> IO a,
```

która pozwoli na przetestowanie przykładowego transformatora strumieni.

Zadanie 3 (2p). Napisz funkcję `listTrans :: StreamTrans i o a -> [i] -> ([o], a)` pozwalającą transformować strumienie reprezentowane jako listy. Twoja implementacja powinna się dobrze zachowywać również dla list nieskończonych, np.

```
take 3 $ fst $ listTrans toLower ['A','A'..]
```

powinno zwrócić napis `"aaa"`.

Zadanie 4 (1p). Napisz funkcję `runCycle :: StreamTrans a a b -> b`, która uruchamia transformator strumieni poprzez skierowanie wyjścia na jego własne wejście.

Zadanie 5 (1p). Zdefiniuj operator

```
(|>|) :: StreamTrans i m a -> StreamTrans m o b -> StreamTrans i o b
```

pozwalający na przekierowanie wyjścia jednego transformatora na wejście drugiego.

Zadanie 6 (2p). Napisz funkcję

```
catchOutput :: StreamTrans i o a -> StreamTrans i b (a, [o])
```

która przekształca podany transformator strumieni w taki, który nic nie wypisuje na wyjściu, tylko zbiera wypisywane elementy na listę.

Zadanie 7 (4p). *Brainfuck* jest ekstremalnie prostym, ezoterycznym językiem programowania, którego składnię abstrakcyjną można przedstawić następującym typem danych.

```
data BF
  = MoveR      -- >
  | MoveL      -- <
  | Inc        -- +
  | Dec        -- -
  | Output     -- .
  | Input      -- ,
  | While [BF] -- [ ]
```

Składnia konkretna jest równie prosta: każda z sześciu instrukcji prostych jest reprezentowana pojedynczym znakiem (>,<,+,-,.,, — tak jak podano w komentarzu), natomiast jedyna złożona instrukcja `While` jest reprezentowana poprzez parę nawiasów kwadratowych, pomiędzy którymi znajduje się dowolny ciąg instrukcji (prostych i złożonych). Wszystkie pozostałe znaki są ignorowane (można z nich budować komentarze). Napisz transformator strumieni

```
brainfuckParser :: StreamTrans Char BF ()
```

który zamienia reprezentację konkretną programu w języku *Brainfuck* na ciąg instrukcji w składni abstrakcyjnej.

Wskazówka: Może (ale nie musi) okazać się przydatne rozwiązanie poprzedniego zadania oraz zadania 9.

Zadanie 8 (4p). *Brainfuck* ma nie tylko prostą składnię, ale również prostą semantykę. Programy operują na nieskończonej taśmie liczb[1], początkowo wypełnionej zerami. Na taśmie znajduje się kursor, który można przesuwać instrukcjami `MoveR` oraz `MoveL` (o jedną pozycję odpowiednio w lewo i w prawo). Instrukcje`Inc` oraz `Dec` odpowiednio zwiększają i zmniejszają liczbę na pozycji wskazywanej przez kursor. Instrukcja `Output` wypisuje na wyjście znak, o kodzie ASCII znajdującym się na pozycji wskazywanej przez kursor, natomiast `Input` wczytuje znak i umieszcza pod kursorem na taśmie. W końcu instrukcja `While` jest podobna do znanej wam pętli *while*: wykonuje podany ciąg instrukcji tak długo, jak pod kursorem znajduje się niezerowa liczba. Napisz funkcję

```
runBF :: [BF] -> StramTrans Char Char ()
```

interpretującą program w języku *Brainfuck*. Wygodnie będzie napisać pomocniczą funkcję, która jawnie przekazuje stan taśmy reprezentowanej jako dwie listy: przed i za kursorem.

```
type Tape = ([Integer], [Integer])
evalBF :: Tape -> BF -> StreamTrans Char Char Tape

evalBFBlock :: Tape -> [BF] -> StreamTrans Char Char Tape
evalBFBlock = foldM evalBF -- jeśli rozwiązałeś zadanie 9
```

Do konwersji pomiędzy typami `Char` oraz `Integer` może się przydać następująca funkcja.

```
coerceEnum :: (Enum a, Enum b) => a -> b
coerceEnum = toEnum . fromEnum
```

Wskazówka: Może (ale nie musi) okazać się przydatne rozwiązanie zadania 9.

Zadanie 9 (2p). Odkryj strukturę monady w typie `StreamTrans i o a` i zainstaluj go w klasie `Monad`[2]. Implementacja funkcji `(>>=)` powinna sekwencjonować obliczenia: `m >>= f` wykonuje najpierw wszystkie akcje z `m`, a dopiero na koniec wywołuje `f`.

[1] Oryginalnie, *Brainfuck* operował na bajtach, ale na potrzeby tego zadania możemy przyjąć, że na taśmie znajdują się liczby typu `Integer`.

[2] W SKOSie znajduje się przykładowy kod demonstrujący jak to zrobić dla monady identycznościowej.