Programowanie funkcyjne 2016

Grupa kzi
Lista 3
(25.10.2016)

Uwaga†: Jeśli zadanie jest niedospecyfikowane, to braki w specyfikacji można uzupełnić w dowolny sensowny(!) sposób.

Zadania w OCamlu

Semnatyka funkcji w zadaniach 1-7 jest taka sama jak funkcji o tych samych nazwach w Haskellu, tyle że gorliwa. W zadaniach nie wolno korzystać z funkcji bibliotecznych na listach (tylko dopasowanie wzorca i konstruktory).

1. Zdefiniuj funkcję `unfoldr : ('a -> ('a * 'b) option) -> 'a -> 'b list` . Jeśli $f\ a =$ `Some`(a_1, b_1), $f\ a_1 =$ `Some`(a_2, b_2) , . . . , $f\ a_n =$ `None` , to `unfoldr` od f, a zwraca $[b_1, ..., b_n]$, w.p.p. nie zwraca nic (może rzucać wyjątek albo się zapętlać). (2 pkt)

2. Zdefiniuj ogonowo funkcję

   ```
   scanl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a list,
   ```

   która generuje listę wszystkich kolejnych wyników pośrednich funkcji `fold_left` wywołanej na tych samych argumentach, z dodaną na początku wartością bazową. Czyli `scanl` $f\ b\ [e_1, e_2, ..., e_n]$ powinno zwracać listę

   $$[b,\ f\ b\ e_1,\ f\ (f\ b\ e_1)\ e_2,\ ...,\ f\ (...(f\ (f\ b\ e_1)\ e_2)...)\ e_n].$$

   (3 pkt)

3. Analogiczną funkcję dla `fold_right` można zdefiniować w następujący sposób

   ```
   let scanr (f : 'a -> 'b -> 'b) (l : 'a list)
             (base : 'b) : 'b list =
       reverse (scanl (flip f) base (reverse l)),
   ```

gdzie `flip f = fun x y -> f y x`, a `reverse` jest funkcją z kolejnego zadania. Ta implementacja ma asymptotycznie optymalna złożoność, ale robi niepotrzebnie dużo przebiegów po listach pośrednich. Popraw ją tak, żeby tego uniknąć. Twoja implementacja powinna być ogonowa. (4 PKT)

Wszystkie funkcje od tego miejsca należy zdefiniować za pomocą foldów z biblioteki standardowej.

4. Zdefiniuj funkcję `reverse : 'a list -> 'a list`, która odwraca listę. (1 PKT)

5. Zdefiniuj funkcję `map : ('a -> 'b) -> 'a list -> 'b list`. Funkcja `map` zaaplikowana do f i $[e_1, e_2, ..., e_n]$ zwraca $[f\ e_1, f\ e_2, ..., f\ e_n]$. (2 PKT)

6. Zdefiniuj funkcję `all : ('a -> bool) -> 'a list -> bool`, która sprawdza, czy predykat zachodzi dla wszystkich elementów listy. (1 PKT)

7. Zdefiniuj funkcję `any : ('a -> bool) -> 'a list -> bool`, która sprawdza, czy predykat zachodzi dla jakiegoś elementu listy. (1 PKT)

   a) +1 PKT jeśli `any` zaaplikowana do predykatu p i listy $[e_1, ..., e_n]$ będzie miała złożoność
   $$O(min_i(p\ e_i \vee i = n)).$$

8. Zdefiniuj typ `aritm_literal` z zeroarnymi konstruktorami `Neg`, `Add`, `Sub`, `Mul`, `Div`, `Pow` i jednoarnym `Num` opakowującym `float`-a. (1 PKT)

9. Zdefiniuj funkcję `eval_rpn : aritm_literal list -> float` , która bierze listę reprezentującą wyrażenie w odwrotnej notacji polskiej (Reverse Polish Notation) i zwraca wynik obliczenia tego wyrażenia. Wartości `Neg`, `Add`, `Sub`, `Mul`, `Div`, `Pow` reprezentują odpowiednio $\sim$, $+$, $-$, $\cdot$, $:$, $\wedge$, gdzie $\sim$ to unarny minus, a $\wedge$ to potęga. Wartość postaci `Num` x reprezentuje liczbę x.

   Przykład: Lista

   ```
   [Num 2.0; Neg; Num 3.0; Num 0.5; Mul; Add; Num (-1.0); Pow]
   ```

   reprezentuje wyrażenie
   $$2\ \sim\ 3\ \frac{1}{2}\ \cdot\ +\ -1\ \wedge$$
   które oblicza się do -2. Zadbaj o to, żeby `eval_rpn` rzucała wyjątki `TooFewOperations`, `TooManyOperations`, gdy wyrażenie ma nieprawidłową składnię (np. `[Num 2.0; Neg; Num 3.0]`) i `IllegalOperation` (op, a, b), gdy próbujemy wykonać operację op na niedozwolonych argumentach a, b (np. op = `Pow`, $a = -2.0$, $b = 0.5$). (4 PKT)