Programowanie Funkcyjne 2018

Lista zadań nr 3 dla grupy TWI: funkcje wyższych rzędów i listy

Rozwiązania należy zgłosić w systemie SKOS do 4 listopada 2018

Z powodu zawodów AMPPZ zajęcia w pracowni 25 października 2018 nie odbędą się.

W poniższych zadaniach możesz używać funkcji zdefiniowanych w modułach `Pervasives` i `List`.

Jeśli w definicjach funkcji ocamlowych:

```
let h xs = List.fold_right f xs c
let h_TR xs = List.fold_left g c xs
```

gdzie

```
f : 'elem -> 'res -> 'res
g : 'res -> 'elem -> 'res
c : 'res
xs : 'elem list
```

rozwiniemy definicje funkcji `List.fold_right` i `List.fold_left`, to otrzymamy definicje funkcji `h` i `h_TR` wykorzystujące jawną rekursję:

```
let rec h xs =
   match xs with
   | [] -> c
   | x::xs -> f x (h xs)
```

oraz

```
let h_TR xs =
   let rec aux acc xs =
      match xs with
      | [] -> acc
      | x::xs -> aux (g acc x) xs
   in aux c
```

Zatem funkcje `List.fold_right` i `List.fold_left` to uniwersalne narzędzia do zwięzłego definiowania funkcji rekurencyjnych działających na listach, przy czym rekursja w `List.fold_right` jest nieogonowa, zaś w `List.fold_left` — ogonowa. Ta druga jest przez to bardziej efektywna, ale — podobnie jak w przypadku jawnej rekursji z akumulatorem — wymaga czasem „poprawienia" argumentu lub wyniku (zwykle odwrócenia listy), co jest związane z generowaniem nieużytków. Mamy np.

```
let length xs = List.fold_left (fun n _ -> n+1) 0 xs
let map f xs = List.fold_right (fun x xs -> f x :: xs) xs []
let rev_append xs ys = List.fold_left (fun xs x -> x::xs) xs ys
let (@) xs ys = List.fold_right List.cons ys xs
```

zaś żeby wyznaczać sumę elementów pewnej listy `xs : int list` wystarczy napisać

```
List.fold_left (+) 0 xs
```

Spósród dwóch funkcji *fold* dla list szczególnie użyteczna jest funkcja `List.fold_left`, gdyż w zwięzły sposób efektywnie realizuje obliczenie z akumulatorem. Ten sposób iteracji względem listy jest znacznie bardziej zwięzły, niż korzystanie z pomocniczej funkcji zdefiniowanej za pomocą jawnej rekursji, np.

```
let rec sum acc = function [] -> acc | x::xs -> sum (x+acc) xs in sum 0 xs
```

Zadanie 1 (1 pkt). Liczbę w zapisie dziesiętnym reprezentujemy w postaci listy znaków. Na przykład lista `['3';'7';'8']` oznacza liczbę 378. Korzystając z funkcji `int_of_char` (która ujawnia kod ASCII podanego znaku) zaprogramuj funkcję

```
atoi : char list -> int
```

ujawniającą liczbę całkowitą reprezentowaną przez podaną listę znaków. Użyj przy tym a) jawnie rekursji ogonowej, b) funkcji `List.fold_left`. *Wskazówka:* kody ASCII cyfr 0–9 są kolejnymi liczbami całkowitymi.

Zadanie 2 (1 pkt). Wielomiany o współczynnikach rzeczywistych reprezentujemy w postaci list współczynników w kolejności od najwyższej potęgi do najniższej. Np. `[1.;0.;-1.;2.]` oznacza wielomian $x^3 - x + 2$. Przyjmij, że lista pusta oznacza ten sam wielomian, co lista `[0.]`. Napisz funkcję

```
polynomial : float list -> float -> float
```

wykorzystującą schemat Hornera do wyznaczenia wartości wielomianu o podanych współczynnikach dla podanego argumentu. Zaprogramuj ją a) jawnie korzystając z rekursji ogonowej oraz b) używając funkcji `List.fold_left`.

Zadanie 3 (1 pkt). Zaprogramuj ogonową wersję funkcji `List.fold_right` używając przy tym a) jawnej rekursji i b) funkcji `List.fold_left`. W obu przypadkach będziesz pewnie potrzebował funkcji `List.rev`. Zauważ, że druga definicja jest równaniem pozwalającym na zamianę w dowolnym wyrażeniu wywołania funkcji `List.fold_right` na wywołanie `List.fold_left`.

Zadanie 4 (2 pkt). Zauważ, że schemat rekursji użyty w definicji funkcji `ins_everywhere` z poprzedniej listy ma postać

```
let rec h xs =
   match xs with
   | [] -> c
   | x::xs -> f x xs (h xs)
```

tj. do wyznaczenia wyniku za pomocą funkcji `f` w kroku indukcyjnym potrzebujemy nie tylko głowy listy (`x`) i wyniku wywołania rekurencyjnego (`h xs`), ale też samej listy `xs`. Z tego powodu funkcje *fold* nie są odpowiednie do zwięzłego zdefiniowania funkcji `ins_everywhere`. Możemy jednak wykorzystać następującą sztuczkę: definiowana funkcja `h` powinna zwracać parę złożoną z wyniku i oryginalnej listy, dla której ten wynik został obliczony, tj. niech `c' = (c,[])` oraz

```
f' : 'elem -> 'res * 'elem list -> 'res * 'elem list
```

Funkcja `f'` ma teraz dostęp do listy `xs`, gdyż jest ona częścią wyniku wywołania rekurencyjnego przekazywanego jej jako drugi argument i możemy napisać

```
let h xs = fst (List.fold_right f' xs c')
```

Ostateczny wynik możemy teraz wybrać za pomocą funkcji `fst`. Podczas obliczenia oryginalna lista jest kopiowana. Ta sztuczka powoduje więc powstanie pewnej ilości dodatkowych nieużytków.

Zaprogramuj w ten sposób funkcję `ins_everywhere` korzystając z funkcji a) `List.fold_right`, b) `List.fold_left`. *Uwaga:* nie rób tego w produkcyjnym kodzie. Celem tego zadania jest pokazanie, że *można*, a nie że *należy* tak robić.

Zadanie 5 (2 pkt). Współczynniki wielomianu z zadania 2 umieszczamy teraz w kolejności rosnących potęg. Przykładowy wielomian z zadania 2 reprezentujemy teraz za pomocą listy `[2.,-1.,0.,1.]`. Zaprogramuj funkcję `polynomial` dla tej reprezentacji a) jawnie korzystając z rekursji (nieogonowej), b) używając funkcji `List.fold_right`, c) jawnie korzystając z rekursji ogonowej, d) używając funkcji `List.fold_left`.

Zadanie 6 (1 pkt). Zaprogramuj funkcję `iperm` z poprzedniej listy używając zamiast jawnej rekursji funkcji a) `List.fold_left` i b) `List.fold_right`.

Zadanie 7 (2 pkt). Zauważ, że skoro w Ocamlu parametry są przekazywane przez wartość, to funkcje *fold* zawsze wykonują obliczenie dla każdego elementu listy, nawet jeśli wynik jest znany wcześniej. Na przykład czas obliczenia wyrażenia

```
List.fold_left (&&) true [false;...;false]
```

jest proporcjonalny do długości listy, podczas gdy funkcja

```
let rec and_all xs =
   match xs with
   | [] -> true
   | x::xs -> x && and_all xs
```

wywołana dla listy `[false;...;false]` zwraca wynik po wykonaniu jednego kroku (nie dochodzi do wywołania rekurencyjnego). Obliczenie w pierwszym wyrażeniu ma bowiem postać

```
let rec and_all xs =
   match xs with
   | [] -> true
   | x::xs -> (fun a b -> a && b) x (and_all xs)
```

i — w odróżnieniu od wcześniejszego schematu — wywołanie rekurencyjne nie jest tu ogonowe. Zatem do zaprogramowania obliczeń tego typu funkcje *fold* też nie są odpowiednie (inaczej jest w języku *non-strict*, takim jak Haskell, w którym funkcja `foldr` nadaje się do tego celu, bo jest inkrementacyjna, a funkcja `foldl` — nie, gdyż jest monolityczna). Efektywne zaimplementowanie za pomocą funkcji *fold* funkcji `ins_ord` wstawiającej element do posortowanej listy tak, żeby zachować porządek nie jest więc łatwe (choć jest możliwe, jeśli skorzystamy z konstrukcji wywołujących skutki uboczne, np. z wyjątków). Zaprogramuj zatem nieefektywną wersję funkcji `ins_ord` korzystając z funkcji a) `List.fold_right` i b) `List.fold_left` a następnie użyj jej oraz a) `List.fold_right` i b) `List.fold_left` do zdefiniowania funkcji `isort`. Jeśli nie brzydzisz się skutków ubocznych spróbuj zaprogramować też za pomocą funkcji *fold* funkcje

```
monotone : 'a list -> bool
and_all : bool list -> bool
ins_ord : 'a -> 'a list -> 'a list
```

które przerywają iterację gdy tylko znany jest wynik. Wyjątki grają tu rolę instrukcji skoku `break` przerywającej iterację pętli `for`. Ta analogia wynika z faktu, że np. obliczenie `List.fold_left g c xs` to w istocie (kod w Pseudo-Ocamlu):

```
acc := c;
for x in xs do
   acc := g acc x;
return acc;
```

Zadanie 8 (2 pkt). Zaprogramuj funkcje

```
sel_anything : 'a list -> ('a * 'a list) list
sperm : 'a list -> 'a list list
sel_min : 'a list -> 'a * 'a list
ssort : 'a list -> 'a list
```

z poprzedniej listy korzystając z funkcji a) `List.fold_right` i b) `List.fold_left`. Wskaż, w których przypadkach definicje są zwięzłe i naturalne, a gdzie nie warto było używać funkcji *fold*.

Zadanie 9 (1 pkt). *Podlistą* listy $[x_1; x_2; \ldots; x_n]$ jest lista $[x_{i_1}; x_{i_2}; \ldots; x_{i_k}]$, gdzie $0 \leq k \leq n$ oraz $1 \leq i_1 < i_2 < \ldots < i_k \leq n$. Lista n-elementowa posiada 2^n podlist. Zaprogramuj funkcję

```
sublist : 'a list -> 'a list list
```

tworzącą listę wszystkich podlist podanej listy. Użyj przy tym a) jawnej rekursji, b) funkcji *fold*.

Wiersze (o indeksach $1 \ldots m$) macierzy o m wierszach i n kolumnach:

$$\begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ \vdots & \vdots & & \vdots \\ x_{m1} & x_{m2} & \ldots & x_{mn} \end{bmatrix}$$

reprezentujemy w postaci list $[x_{i1}; x_{i2}; \ldots; x_{in}]$ typu `'a list`, a całe macierze — w postaci list wierszy:

```
type 'a mtx = 'a list list
```

Np. reprezentacją macierzy

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

jest lista

```
[[1;2;3;4];[5;6;7;8];[9;10;11;12]] : int mtx
```

Niech

```
exception Mtx of string
```

będzie wyjątkiem zgłaszanym przez twoje funkcje, jeśli ich dane wejściowe nie będą poprawne. Zgłoszenie wyjątku następuje wskutek obliczenia wyrażenia

```
raise (Mtx "opis błędu")
```

W poniższych zdaniach tam, gdzie to jest możliwe, uprość implementację korzystając z funkcji udostępnianych przez moduł `List`, szczególnie z funkcji `List.fold_left` oraz z funkcji zdefiniowanych we wcześniejszych zadaniach.

Zadanie 10 (1 pkt). Zaprogramuj funkcję

```
mtx_dim : 'a Mtx -> { rows : int; columns : int }
```

ujawniającą wymiary podanej macierzy. Nie każda lista list elementów jest poprawną reprezentacją macierzy — reprezentacje wszystkich wierszy powinny być tej samej długości, a każdy wiersz i ich lista nie mogą być listami pustymi (wykluczamy macierze zerowego wymiaru). Jeśli podany argument nie spełnia powyższych warunków, to zgłoś błąd rzucając wyjątek `Mtx`.

W poniższych zadaniach przyjmij zasadę ograniczonego zaufania: załóż, że dane wejściowe są poprawne i nie sprawdzaj osobno wszystkich warunków poprawności, a wyjątek `Mtx` zgłaszaj wówczas, jeśli natrafisz na błąd w danych uniemożliwiający dokończenie obliczenia.

Zadanie 11 (1 pkt). Zaprogramuj funkcje

```
mtx_row : row:int -> 'a Mtx -> 'a list
mtx_column : column:int -> 'a Mtx -> 'a list
mtx_elem : column:int -> row:int -> 'a Mtx -> 'a
```

ujawniające, odpowiednio, wiersz, kolumnę i element macierzy o podanych indeksach.

Zadanie 12 (1 pkt). Zaprogramuj funkcję

```
traspose : 'a Mtx -> 'a Mtx
```

wyznaczającą transpozycję podanej macierzy.

Zadanie 13 (1 pkt). Zaprogramuj funkcję

```
mtx_add : float Mtx -> float Mtx -> float Mtx
```

wyznaczającą sumę podanych macierzy.

Zadanie 14 (1 pkt). Zaprogramuj funkcję

```
scalar_prod : float list -> float list -> float
```

wyznaczającą iloczyn skalarny dwóch wektorów. Wykorzystaj tę funkcję do zdefiniowania alternatywnej wersji funkcji `polynomial` z zadania 2. Przyda się pewnie przy tym biblioteczna funkcja `List.fold_left2`.

Zadanie 15 (1 pkt). Zaprogramuj funkcję

```
mtx_apply : float Mtx -> float list -> float list
```

wyznaczającą iloczyn macierzy i wektora. Przyda się pewnie przy tym bilbioteczna funkcja `List.map2`. Zaprogramuj następnie funkcję

```
mtx_mul : float Mtx -> float Mtx -> float Mtx
```

wyznaczającą iloczyn podanych macierzy.

Zadanie 16 (1 pkt). Zaprogramuj funkcję

```
det : float Mtx -> float
```

obliczającą wyznacznik podanej macierzy kwadratowej.