

# Metody programowania 2016

## Lista zadań nr 12

Na zajęcia 25 i 30 maja 2016

**Zadanie 1 (1 pkt).** Oto program w Prologu generujący permutacje przez wstawianie:

```
permi([], []).
permi([H|T], S) :-
    permi(T, R),
    select(H, S, R).
```

gdzie

```
select(H, [H|T], T).
select(X, [H|T], [H|S]) :-
    select(X, T, S).
```

Przepisz predykat `permi` w Haskellu, tj. zaprogramuj funkcję

```
permi :: [a] -> [[a]]
```

używając

1. funkcji `foldr`, `unfoldr`, `foldl`, `map` itp., wzorców i jawnej rekursji;
2. wyrażeń listowych;
3. monady `[]` i notacji `do`.

Porównaj czytelność wszystkich trzech rozwiązań.

**Zadanie 2 (1 pkt).** Rozwiąż poprzednie zadanie dla predykatu

```
perms([], []).
perms(S, [H|T]) :-
    select(H, S, R),
    perms(R, T).
```

**Zadanie 3 (1 pkt).** Rozwiąż poprzednie zadanie dla predykatu

```
sublist([], []).
sublist([H|T], [H|S]) :-
    sublist(T, S).
sublist([_|T], S) :-
    sublist(T, S).
```

który generuje wszystkie podlisty podanej listy.

**Zadanie 4 (1 pkt).** Oto program w SML-u obliczający iloczyn elementów podanej listy:

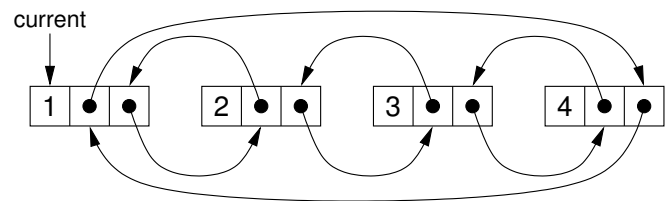
```
exception E
fun prod xs = foldr
    (fn (n,p) => if n=0 then raise E else n*p)
    1
    xs
handle E => 0
```

Wyjaśnij, czemu rzucanie wyjątku czyni ten program bardziej efektywnym. Przepisz go w Haskellu używając monady `Maybe` do symulowania wyjątków. Wyjaśnij, czemu ten program może być w Haskellu uproszczony do

```
prod :: [Integer] -> Integer
prod = foldr (\ n p -> if n=0 then 0 else p) 1
```

bez straty efektywności, podczas gdy w SML-u potrzebujemy wyjątków lub podobnego mechanizmu (kontynuacji itp.).

**Zadanie 5 (1 pkt).** Dwukierunkowa lista cykliczna, to struktura danych o dostępie sekwencyjnym, w której każdy element posiada wskaźnik do elementu poprzedniego oraz następnego i elementy te tworzą cykl (być może nieskończony). Ponieważ w trwałej strukturze danych modyfikacja elementu pociąga za sobą konieczność skopiowania wszystkich elementów z których dany element jest osiągalny, więc modyfikacja dwukierunkowej listy cyklicznej zawsze prowadzi do konieczności skopiowania całej struktury, co jest bardzo nieefektywne. Ograniczmy się więc do zaprogramowania selektorów i obserwatorów, tj. operacji, które nie modyfikują struktury.



Niech zatem

```
data Cyclist a = Elem (Cyclist a) a (Cyclist a)
```

Zdefiniuj funkcje

```
fromList :: [a] -> Cyclist a
forward, backward :: Cyclist a -> Cyclist a
label :: Cyclist a -> a
```

Funkcja `fromList` tworzy cyklistę zawierającą elementy podanej listy. Bieżącym elementem jest pierwszy element listy. Funkcje `forward` i `backward` przemieszczają wskaźnik elementu bieżącego w przód i w tył, a `label` ujawnia etykietę bieżącego elementu, `np`.

```
label . forward . forward . forward .
backward . forward . forward
$ fromList [1,2,3] = 2
```

Jeśli `xs` jest listą nieskończoną, to

```
backward . fromlist $ xs = ⊥.
```

**Zadanie 6 (1 pkt).** Zdefiniuj nieskończoną cyklistę

```
enumInts :: Cyclist Integer
```

która zawiera wszystkie liczby całkowite w naturalnym porządku i której bieżącym elementem jest zero.

**Zadanie 7 (1 pkt).** Niech `Cyclist a` będzie naszym stanem obliczeń. Obliczenie modyfikuje stan i zwraca wynik typu `b`, ma więc typ

```
Cyclist a -> (b, Cyclist a)
```

Obliczenie z cyklistą jako stanem jest więc wartością typu

```
newtype Cyc a b =
    Cyc (Cyclist a -> (b, Cyclist a))
```

Uczyń typ `Cyc` a monadą. Zdefiniuj operacje:

```
runCyc :: Cyclist a → Cyc a b → b
fwd  :: Cyc a ()
bkw  :: Cyc a ()
lbl  :: Cyc a a
```

tak by dało się obliczenia zapisywać następująco:

```
example :: Integer
example = runCyc enumInts (do
  bkw
  bkw
  bkw
  bkw
  x ← lbl
  fwd
  fwd
  y ← lbl
  fwd
  z ← lbl
  return (x+y+z))
```