Lista zadań nr 3

Zadanie 1.

Wzorując się na funkcji z wykładu sumującej liczby przy użyciu `fold_left`, zdefiniuj funkcję `product`, obliczającą iloczyn elementów listy. Jaką wartość powinien zwracać `product` dla listy pustej?

Zadanie 2.

Złożenie funkcji f i g definiujemy (jak pamiętamy z przedmiotu „Logika dla informatyków") jako funkcję $x \mapsto f(g(x))$. Zdefiniuj dwuargumentową funkcję `compose`, której wynikiem jest złożenie (jednoargumentowych) funkcji przekazanych jej jako argumenty. Używając modelu podstawieniowego, prześledź wykonanie wyrażeń:

```
compose square inc 5
compose inc square 5
```

Zakładamy, że funkcja `square` oblicza kwadrat swojego argumentu, natomiast `inc` – wartość argumentu powiększoną o 1.

Zadanie 3. (2 pkt)

Zaimplementuj funkcję `build_list n f`, która konstruuje `n`-elementową listę, aplikując `f` do wartości od 0 do $\mathsf{n} - 1$. Dokładniej:

```
build_list n f = [f 0; f 1; ...; f (n - 1)]
```

Wykorzystaj funkcję `build_list` oraz funkcje anonimowe (**`fun`**), aby napisać następujące funkcje:

- `negatives n`, zwracającą listę liczb ujemnych od -1 do $-\mathsf{n}$,
- `reciprocals n`, zwracającą listę odwrotności liczb od 1 do `n` (czyli $1, \ldots, \frac{1}{\mathsf{n}}$),

- `evens n`, zwracającą listę pierwszych `n` liczb parzystych,
- `identityM n`, zwracającą macierz identycznościową o wymiarach $n \times n$ w postaci listy list:

```
# identityM 3
- : int list list = [[1; 0; 0]; [0; 1; 0]; [0; 0; 1]]
```

Zadanie 4. (2 pkt)

Zareprezentuj zbiory przy użyciu predykatów charakterystycznych – tzn. funkcji o typie `'a -> bool`, zwracających **`true`** wtedy i tylko wtedy, gdy argument należy do zbioru. Zdefiniuj:

- `empty_set` – reprezentacja zbioru pustego,
- `singleton a` – zwraca zbiór zawierający wyłącznie element `a`,
- `in_set a s` – zwraca `true` gdy `a` należy do zbioru `s`, w przeciwnym wypadku wynikiem jest `false`,
- `union s t` – zwraca sumę zbiorów `s` i `t`,
- `intersect s t` – zwraca przecięcie zbiorów `s` i `t`.

Zadanie 5.

Rozważ drzewa binarne z wykładu. Narysuj, jak w pamięci reprezentowane jest drzewo `t` zdefiniowane poniżej:

```
let t =
  Node (Node (Leaf, 2, Leaf),
        5,
        (Node (Node (Leaf, 6, Leaf)),
               8,
               (Node (Leaf, 9, Leaf))))
```

Zaimplementuj funkcję `insert_bst` wstawiającą element do drzewa BST, zachowując własność BST. Pokaż, jak będzie wyglądał stan pamięci po wykonaniu wstawienia BST wartości 7. Które fragmenty drzewa t są współdzielone między drzewem `t` i `insert_bst 7 t`?

Zadanie 6.

Wykorzystując funkcję `fold_tree` z wykładu, zdefiniuj następujące funkcje:

- `tree_product t` – iloczyn wszystkich wartości występujących w drzewie,
- `tree_flip t` – odwrócenie kolejności: zamiana lewego i prawego poddrzewa wszystkich węzłów w drzewie,
- `tree_height t` – wysokość drzewa (liczba węzłów na najdłuższej ścieżce od korzenia do liścia),
- `tree_span t` – para złożona z wartości skrajnie prawego i skrajnie lewego węzła w drzewie (czyli najmniejszej i największej wartości w drzewie BST),
- `preorder t` – lista wszystkich elementów występujących w drzewie, w kolejności preorder. Kolejność ta polega na tym, że elementy drzewa posiadającego w korzeniu węzeł listuje się zaczynając od wartości w węźle, po której występują elementy lewego poddrzewa a następnie prawego, również w kolejności preorder. Inaczej można powiedzieć, że kolejność preorder to kolejność odwiedzania węzłów przez przeszukiwanie w głąb (DFS) drzewa.

Zadanie 7. (2 pkt)

Najprostsza implementacja funkcji `flatten` z wykładu posiada poważną wadę – tworzy ona duże ilości nieużytków oraz wykonuje nadmiarowe obliczenia. Tę wadę można szczególnie dobrze zaobserwować na przykładzie drzew, które „rosną tylko w lewo”:

```
let left_tree_of_list xs =
  List.fold_left (fun t x -> Node (t, x, Leaf)) Leaf xs
let test_tree = left_tree_of_list (build_list 20000 Fun.id)
```

Napisz inną implementację `flatten`, która nie posiada tej wady. Nie używaj funkcji `append` (ani operatora `@`)!

Wskazówka: zaimplementuj najpierw dwuargumentową funkcję `flat_append t xs`, której wynikiem jest lista elementów `t` w kolejności infiksowej scalona z listą `xs`. Przykład:

```
# flat_append t [10; 11]
- : int list = [2; 5; 6; 8; 9; 10; 11]
```

Zadanie 8.

Zmodyfikuj funkcję `insert_bst` z zadania 5 (wstawiającą element do drzewa BST) tak, aby możliwe było tworzenie drzew BST z duplikatami. Możesz założyć, że elementy równe elementowi w korzeniu drzewa będą trafiać do prawego poddrzewa.

Zaimplementuj funkcję `tree_sort xs`, implementującą algorytm sortowania przy użyciu drzew BST:

- Utwórz drzewo przeszukiwania składające się z elementów listy `xs`.
- Zwróć listę elementów drzewa w kolejności infiksowej.

Zadanie 9.

Zaimplementuj funkcję `delete`, zwracającą drzewo z usuniętym danym kluczem, dla reprezentacji drzew przeszukiwań binarnych z wykładu. Wskazówka: Aby stworzyć drzewo przeszukiwań binarnych z którego usunęliśmy korzeń, najlepiej znaleźć (jeśli istnieje) najmniejszy element większy od tego korzenia.