Lista zadań nr 4

Zadanie 1. (2 pkt)

Rozważ następującą sygnaturę:

```
module type HUFFMAN = sig
  type 'a code_tree
  type 'a code_dict
  val code_tree : 'a list -> 'a code_tree
  val dict_of_code_tree : 'a code_tree -> 'a code_dict
  val encode : 'a list -> 'a code_dict -> int list
  val decode : int list -> 'a code_tree -> 'a list
end
```

Zmodyfikuj implementację kodowania Huffmana z wykładu tak, aby była zdefiniowana przy pomocy funktora `Huffman` sparametryzowanego dwoma modułami o sygnaturach odpowiednio `DICT` oraz `PRIO_QUEUE`.

Zadanie 2. (2 pkt)

Wbudowany w bibliotekę standardową moduł `Map` udostępnia słownik zaimplementowany przy pomocy drzewa zrównoważonego. Używanym tam porządkiem porównań nie jest domyślny porządek określony przez operatory `(<)` oraz `(=)`. Zamiast tego słowniki są sparametryzowane porządkiem, definiowanym następującą sygnaturą (`Map.OrderedType`):

```
module type OrderedType = sig
    type t
    val compare : t -> t -> int
end
```

Dla dwóch zadanych elementów typu `t`, funkcja `compare` zwraca wartość ujemną, zero lub dodatnią wtedy i tylko wtedy gdy, odpowiednio, pierwszy element jest mniejszy, równy lub większy od drugiego. Biblioteka standardowa języka

OCaml udostępnia funkcje `compare` dla różnych typów, włączając w to `int` (`Int.compare`), `string` (`String.compare`) i `char` (`Char.compare`).

Zmodyfikuj sygnaturę `DICT` z wykładu tak, aby typ kluczy słownika nie był już parametrem typowym typu `dict`, ale ustalonym (choć nieznanym) typem. Zmodyfikowana sygnatura powinna zaczynać się następująco:

```
module type DICT = sig
  type key
  type 'a dict
```

Zadanie 3. (2 pkt)

Zamień definicję modułu `ListDict` podaną na wykładzie na definicję funktora `MakeListDict`, sparametryzowanego modułem o sygnaturze `Map.OrderedType`, i zwracającego moduł o sygnaturze `DICT` z zadania 2. Następnie, korzystając z napisanego funktora, utwórz moduł `CharListDict`.

Zwróć uwagę, że zdefiniowany właśnie moduł nie pozwala (między innymi) na dodawanie nowych elementów do słownika. Dlaczego?

Popraw definicję funktora `MakeListDict` przez modyfikację sygnatury zwracanego modułu. Możesz wykorzystać konstrukcję **`with type`**, umożliwiającą dodanie do sygnatury definicji typu, który był w niej abstrakcyjny (ukryty). Przykładowo, `DICT` **`with type`** `key = char` oznacza sygnaturę słownika, w którym typ klucza jest typem znakowym, czyli:

```
sig
  type key = char
  type 'a dict
  ...
end
```

Zadanie 4. (2 pkt)

Utwórz otypowany analogicznie do funktora `MakeListDict` z zadania 3 funktor `MakeMapDict`, wykorzystujący wbudowane w bibliotekę standardową OCamla słowniki z modułu `Map`. Następnie, korzystając z napisanego funktora, utwórz moduł `CharMapDict`. Pamiętaj o użyciu **`with type`**.

Zadanie 5. (2 pkt)

Kopce lewicowe (znane też jako drzewa lewicowe) to prosta i efektywna struktura danych implementująca kolejkę priorytetową (którą na wykładzie zaimplementowaliśmy używając nieefektywnej struktury listy posortowanej). Podobnie jak w przypadku posortowanej listy, chcemy móc znaleźć najmniejszy element w stałym czasie, jednak chcemy żeby pozostałe operacje (wstawianie, usuwanie minimum i scalanie dwóch kolejek) działały szybko – czyli w czasie logarytmicznym. W tym celu, zamiast listy budujemy *drzewo binarne*, w którym wierzchołki zawierają elementy kopca wraz z wagami. Dodatkowym niezmiennikiem struktury danych, który umożliwi efektywną implementację jest to, że każdemu kopcowi przypisujemy *rangę*, którą jest długość „prawego kręgosłupa" (czyli ranga prawego poddrzewa zwiększona o 1 – lub zero w przypadku pustego kopca), i że w każdym poprawnie sformowanym kopcu ranga lewego poddrzewa jest nie mniejsza niż ranga prawego poddrzewa.

Pozwala to nam zdefiniować następującą implementację:

```
module LeftistHeap = struct
  type ('a, 'b) heap =
    | HLeaf
    | HNode of int * ('a, 'b) heap * 'a * 'b * ('a, 'b) heap

  let rank = function HLeaf -> 0 | HNode (n, _, _, _, _) -> n

  let heap_ordered p = function
    | HLeaf -> true
    | HNode (_, _, p', _, _) -> p <= p'

  let rec is_valid = function
    | HLeaf -> true
    | HNode (n, l, p, v, r) ->
        rank r <= rank l
        && rank r + 1 = n
        && heap_ordered p l
        && heap_ordered p r
        && is_valid l
        && is_valid r

  let make_node p v l r = ...
end
```

Wierzchołki reprezentujemy przy użyciu konstruktora `HNode`, którego polami są, kolejno: ranga wierzchołka, lewe poddrzewo, priorytet elementu, element, prawe poddrzewo. Funkcja `is_valid` sprawdza czy zachowany jest porządek kopca (używając `heap_ordered`) i czy własność rangi opisana powyżej jest spełniona.

Zaimplementuj funkcję („inteligentny konstruktor") `make_node`. Zwróć uwagę, że `make_node` nie przyjmuje rangi tworzonego kopca, ale musi ją wyliczyć. Oznacza też, że musimy stwierdzić w funkcji konstruktora który z kopców powinien zostać prawym, a który lewym poddrzewem (możemy natomiast założyć że porządek kopca zostanie zachowany).

Zaimplementuj następnie funkcję `heap_merge` złączającą dwa kopce. Idea scalania kopców jest następująca: jeśli jeden z kopców jest pusty, scalanie jest trywialne (bierzemy drugi kopiec). Jeśli oba są niepuste, możemy znaleźć najmniejszy priorytet elemenu każdego z nich. Mniejszy z tych dwóch priorytetów i skojarzony z nim element powinny znaleźć się w korzeniu wynikowego kopca — łatwo go znaleźć. Mamy zatem cztery obiekty:

- element o najniższym priorytecie (nazwiemy go e),
- jego priorytet (nazwiemy go p),
- lewe poddrzewo kopca z którego korzenia pochodzi $e - h_l$
- prawe poddrzewo kopca z którego korzenia pochodzi $e - h_r$
- drugi kopiec, h, którego korzeń miał priorytet większy niż e.

Aby stworzyć wynikowy kopiec wystarczy teraz scalić h_r i h (rekurencyjnie), a następnie stworzyć wynikowy kopiec z kopca otrzymanego przez rekurencyjne scalanie, kopca h_l elementu e i priorytetu p. Implementując `heap_merge`, wykorzystaj funkcję `make_node`.

Zadanie 6. (2 pkt)

Wykorzystaj strukturę danych kopca z poprzedniego zadania, aby zaimplementować sygnaturę `PRIO_QUEUE` z wykładu. – to znaczy, zaproponuj definicje `empty`, `insert`, `pop`, `min_with_prio` wykorzystujące, zamiast list uporządkowanych, kopce lewicowe.

Używając sygnatury kolejki priorytetowej, zaimplementuj (w formie funktora) funkcję `pqsort xs`, działającą w następujący sposób:

- Utwórz kolejkę priorytetową składającą się z elementów listy `xs` (priorytetem elementu niech będzie on sam).
- Skonstruuj posortowaną listę wynikową przez wyjmowanie kolejnych elementów z kolejki.

Implementacja powinna działać poprawnie dla obu implementacji kolejek priorytetowych (listy uporządkowane i kopce lewicowe).