Arbres préfixes

OPTION INFORMATIQUE - TP nº 1.5 - Olivier Reynet

On se dote d'un type pour représenter les arbres préfixe pour élaborer un code de Huffmann :

```
type btree = Leaf of char
    | Node of int * btree * btree;;
```

Ces arbres sont constitués de feuilles étiquetées par des types `char` (les symboles à encoder) et des nœuds étiquietés par des types `int` (les nombre d'occurrences).

1. Écrire une fonction de signature `occurences : string -> (btree * int)list` dont le paramètre est un message à encoder. Cette fonction renvoie la liste des couples feuilles et nombre d'occurrences associés à chaque symbole. On utilisera une table de hachage pour effectuer le décompte des occurrences de chaque caractère dans le message. Cette table sera transformée en liste par la commande `Hashtbl.fold (fun k v acc -> (Leaf k, v):: acc)dict []`.Par exemple, `occurences "Hello"` renvoie `[(Leaf 'l', 2); (Leaf 'o', 1); (Leaf 'e', 1); (Leaf 'H', 1)]`.

   Solution :

   ```
   let occurences s =
       let dict = Hashtbl.create 64 in
       let n = String.length s in
       for k = 0 to n - 1 do
           let c = s.[k] in
           if Hashtbl.mem dict c
           then Hashtbl.replace dict c ((Hashtbl.find dict c) + 1)
           else Hashtbl.add dict c 1
       done;
       Hashtbl.fold (fun k v acc -> (Leaf k, v) :: acc) dict [];;
   ```

   Pour construire l'arbre d'Huffmann, il faut Implémenter une file de priorités. On se propose de réaliser celle-ci en utilisant :

   - une liste dont les éléments sont des couples de type `(btree * int)`,
   - une fonction `insert_elem` qui insère un élément dans une telle liste, au bon endroit, c'est à dire d'après l'entier qui code le nombre d'occurrences du symbole : les plus petits nombres en tête.
   - une fonction `insert_sort` qui implémente le tri par insertion pour une liste de type `(btree * int)list`.

2. Écrire une fonction de signature `compare : 'a * int -> 'b * int -> int` dont les paramètres sont deux couples (symbole,occurrences) de la liste et qui renvoie le résultat de `n1-n2` si `n1` et `n2` sont les occurrences associés aux symboles des deux couples.

Solution :

```
let compare (_,n1) (_,n2) = n1 - n2 ;;
```

3. Écrire une fonction récursive de signature `insert_elem : ('a * int)list -> 'a * int -> ('a * int)list` qui insère un couple (symbole,occurrences) à la bonne place dans la liste passée en paramètre.

Solution :

```
let rec insert_elem sorted e =
    match sorted with
        | [] -> [e]
        | h::t when compare h e <= 0 -> h::(insert_elem t e)
        | h::t ->  e::h::t;;
```

4. Écrire une fonction récursive de signature `insert_sort : ('a * int)list -> ('a * int)list` qui implémente le tri par insertion.

Solution :

```
let rec insert_sort l =
    match l with
        | [] ->  []
        | e::t ->  insert_elem (insert_sort t) e;;
```

5. Écrire une fonction de signature `merge : btree * int -> btree * int -> btree * int` qui fusionne les deux sous-arbres en un arbre `btree` et renvoie le couple constitué du nouveau nœud et de la somme des occurrences de sous-arbres. Par exemple, `merge (Leaf 'a',5)(Leaf 'b',9)` renvoie `(Node (14, Leaf 'a', Leaf 'b'), 14)`.

Solution :

```
let merge a1 a2 = let (n1, o1) = a1  and (n2, o2) = a2 in
                    (Node((o1 + o2), n1, n2), o1 + o2);;
```

6. Écrire une fonction de signature `huffmann_tree : (btree * int)list -> btree` qui construit l'arbre de Huffmann associé à une liste de couples (symbole, occurrences). Par exemple :

```
let q = [(Leaf 'a',5); (Leaf 'b',9); (Leaf 'c',12); (Leaf 'd',13); (Leaf 'e',16);
     (Leaf 'f',45)];;
let ht = huffmann_tree q;;
(* Node (100, Leaf 'f',
    Node (55, Node (25, Leaf 'c', Leaf 'd'),
    Node (30, Node (14, Leaf 'a', Leaf 'b'), Leaf 'e')))  *)
```

7. Écrire une fonction de signature `h_decode : btree -> string -> string` qui décode un message donné sous la forme d'une chaîne de caractères ne comportant que des 0 et des. Cette fonction renvoie la chaîne de caractères correspondant au message initial. On pourra utiliser l'arbre de Huffmann défini à la question précédente et décoder `"0100111"`.

Solution :

```
let h_decode htree msg =
    let to_decode = bin_string_to_char_list msg in
    let rec down acc ht m =
        match (ht, m) with
            | (Leaf c, []) -> acc ^ String.make 1 c (* msg done *)
            | (Leaf c, r) -> down (acc ^ String.make 1 c) htree r (* restart
               at root, new char *)
            | (Node(_, t1, _), b::q) when b = '0' -> down acc t1 q
            | (Node(_, _, t2), b::q) when b = '1' -> down acc t2 q
            | _ -> failwith "Decoding error !"  in
    down "" htree to_decode;;
```

8. Écrire une fonction de signature `encode_map : btree -> (char, string)Hashtbl.t` dont le paramètre est un arbre de Huffmann et qui renvoie une table de hachage associant chaque symbole à son encodage binaire sous la forme d'une chaîne de caractères. On pourra s'appuyer sur les fonction `String.to_seq` et `List.of_seq`.

Solution :

```
let encode_map htree =
    let map = Hashtbl.create 64 in
    let rec down acc ht =
        match ht with
            | Leaf c -> Hashtbl.add map c (String.of_seq (List.to_seq (List.
               rev acc)))
            | Node(_, t1, t2) -> down ('0'::acc) t1; down ('1'::acc) t2 in
    down [] htree;
    map;;
```

9. Écrire une fonction de signature `h_encode : (char, string)Hashtbl.t -> string -> string` dont les paramètres sont une table d'encodage et un message sous la forme d'une chaîne de caractères et qui renvoie la chaîne binaire qui encode le message. On pourra s'appuyer sur les fonction `String.to_seq` et `List.of_seq`.

Solution :

```
let h_encode h_map msg =
    let to_encode = string_to_list msg in
    let rec aux bits to_e =
        match to_e with
            | [] -> bits
            | c::t -> aux (bits ^ (Hashtbl.find h_map c)) t in
    aux "" to_encode;;
```

10. Écrire une fonction de signature `compression_rate : string -> float` qui calcule le taux de compression d'une chaîne de caractère encodée par l'arbre préfixe de Huffmann. Le taux de compression se calcule : $t = 1 - \frac{c}{8 \times n}$, si c est la longueur de la chaîne binaire compressée et n la longueur de la chaîne du message de départ

Solution :

```
let compression_rate text =
    let occ = occurences text in
    (*    let q = List.sort occ_compare occ in*)
    let q = insert_sort occ in
    let ht = huffmann_tree q in
    let hmap = encode_map ht in
    let emsg = h_encode hmap text in
    1. -. (float(String.length emsg) /. float(8 * String.length text));;
```

R La limite théorique de compression est atteignable avec un code d'Huffmann. On peut calculer le nombre de bits moyen nécessaire pour encoder un symbole. Celui-ci s'écrit :

$$H = -\sum_{i=0}^{n-1} -p_i \log_2 p_i \tag{1}$$

où n est le nombre de symboles et p_i la probabilité d'apparition du symbole i.