TP : Compression de Huffman

Langage OCaml

Dans ce TP on implante l'algorithme de Huffman tel que présenté en cours. Le langage choisi est OCaml. Le fichier à compresser est supposé écrit en ASCII (type **Char** en OCaml). En appendice, on trouve des rappels sur les fonctions des modules **Char**, **Buffer** et des indications sur le contenu d'une archive traitant des files de priorité.

Dans tout le TP on manipule des arbres implantés dans le type suivant :

```
type tree = Leaf of char | Node of tree * tree ;;
```

Ces arbres ne comportent pas de mention de la fréquence des caractères.

1 Encodage

Q1 Écrire la fonction `freq_table (s:string) : int array` qui remplit le tableau des effectifs des caractères dans le texte. Les caractères utilisés sont de type **Char**.

```
# let occ = freq_table "satisfaisant" in Array.sub occ 97 26;;
- : int array =
[|3; 0; 0; 0; 0; 1; 0; 0; 2; 0; 0; 0; 0; 1;
0; 0; 0; 0; 3; 2; 0; 0; 0; 0; 0; 0|]
```

Q2 Écrire la fonction `compute_tree (s:string) : tree` qui calcule un arbre de Huffman associé à la chaîne s selon l'algorithme vu en cours.

```
# compute_tree "satisfaisant";;
- : tree =
Node (Node (Leaf 'i', Leaf 's'),
 Node (Leaf 'a', Node (Leaf 't', Node (Leaf 'f', Leaf 'n'))))
```

Q3 Écrire la fonction `build_dict (t:tree) : (char, string) Hashtbl.t` qui calcule l'encodage des caractères situés dans les feuilles de l'arbre passé en paramètre.

```
# let t = compute_tree "satisfaisant" in
    let d = build_dict t in Hashtbl.find d 'i', Hashtbl.find d
        'f';;
- : string * string = ("00", "1110")
```

Q4 Écrire la fonction `encode (s : string) : tree * string` qui retourne l'arbre de Huffman et un encodage du texte.

```
# encode  "satisfaisant";;
- : tree * string =
(Node (Node (Leaf 'i', Leaf 's'),
  Node (Leaf 'a', Node (Leaf 't', Node (Leaf 'f', Leaf 'n'))))
      ,
 "011011000011110100001101111110")
 # String.length @@ snd @@ encode "satisfaisant";;
- : int = 30
```

Le code de Huffman obtenu par la fonction `encode` est une chaîne de caractères `'0'` et `'1'`. Comme la plupart[1] des codes des caractères du texte initial ont une longueur plus grande que 2, on se retrouve avec un texte encodé plus long (en terme de nombre de caractères) que le texte initial ! Cela vient du fait que les bits **0** et **1** sont représentés par des caractères : leur représentation est donc 8 fois plus longue que nécessaire.

On décide de découper la chaîne encodée s en paquets de huit caractères. Chacun de ces paquets est l'expression d'un nombre binaire (nous choisissons la notation big-endian) dans $[\![0, 255]\!]$ qui est transformé en caractère.

- Si $|s|$ n'est pas divisible par 8, il reste à la fin un dernier paquet de moins de 8 caractères. On complète ce dernier paquet par un nombre n de `'0'` de façon à obtenir huit bits. Il reste à ajouter la conversion en caractère de la sous-chaîne obtenue au résultat. Le nombre de zéro ajouté (moins de 8) est lui même transformé en caractère et devient le dernier caractère du résultat.
- Si $|s|$ est divisible par 8, on ajoute la conversion de zéro en caractère à la fin du résultat (on n'a ajouté aucun 0).

Dans tous les cas, le dernier caractère du résultat est le nombre de zéros ajoutés.

Q5 Fonctions outils :

(a) Écrire la fonction `makechar (s:string) (n:int) : char` qui transforme la sous chaîne de longueur 8 de s commençant en position n en un caractère. La chaîne s est supposée constituée uniquement de caractères `'0'` et `'1'`.

```
# makechar "01100001" 0;;
- : char = 'a'
# makechar "11101100001000" 3;;
- : char = 'a'
```

(b) Écrire la fonction `char2bin (c:char) (n:int) : string` qui donne, exprimé par un `string`, le code ASCII de c si $n = 8$ et les n premiers bits de ce code si $n < 8$.

```
# char2bin 'a' 8;;
- : string = "01100001"
# char2bin 'a' 4;;
- : string = "0110"
```

1. En fait, tous les caractères sauf éventuellement un ou deux ont un code de longueur plus grande que 2.

Q6 Écrire la fonction `codes2string (s:string) : string` qui transforme s (censé être le résultat de la fonction `encode`) en une suite de caractères correspondant au regroupement par paquets de 8 décrit ci-dessus.

```
# codes2string ("01100001"^"01100010"^"01100011");;
- : string = "abc\000"
# codes2string ("01100001"^"01100010"^"01100011"^"101");;
- : string = "abc\240\005"
```

Dans le premier exemple, la longueur de la chaîne est un multiple de 8. On renvoie donc la chaîne `"abc"` suivit de l'encodage de 0 en `char`. Dans le second cas, la longueur de la chaîne n'est pas divisible par 8. On renvoie donc toujours `"abc"` puis le caractère qui correspond à `"101"` suivi de 5 zéros et enfin la conversion de 5 en `char`.

.

Q7 Écrire la fonction `string2code (s:string) : string` réciproque de la précédente.

```
# ("01100001"^"01100010"^"01100011") =
  string2code @@ codes2string ("01100001"^"01100010"^"01100011
      ");;
- : bool = true
# ("01100001"^"01100010"^"01100011"^"101") =
  string2code @@ codes2string ("01100001"^"01100010"^"01100011
      "^"101");;
- : bool = true
```

2 Décodage

Q8 écrire la fonction `next_char (s:string) (i:int) (t:tree) : char * int` qui prend en paramètre une position dans le texte compressé et l'arbre de Huffman. La fonction progresse SIMULTANÉMENT dans le texte compressé et dans l'arbre. Elle s'arrête lorsqu'une feuille est atteinte et retourne le caractère trouvé ainsi que la nouvelle position atteinte dans le texte compressé.

```
# let t , cod = encode "satisfaisant" in next_char cod 0 t;;
- : char * int = ('s', 2)
```

Q9 Écrire la fonction `decode (t, s : tree * string) : string` qui prend en paramètre un tuple (arbre de Huffman, texte compressé) et retourne le texte original.

```
# let t , cod = encode "satisfaisant" in decode (t,cod);;
- : string = "satisfaisant"
```

Appendice

- Les caractères traités sont de type **Char**.

Pour obtenir le code ASCII d'un caractère :

```
# Char.code 'a';;
- : int = 97
```

— Pour passer du code ASCII au caractère :

```
# Char.chr 97;;
- : char = 'a'
```

- Pour retourner l'encodage d'un texte, on a besoin d'ajouter l'encodage du caractère courant à ce qui a déjà été compressé. La bonne bibliothèque pour cela est le module **Buffer**. Il s'agit de tableaux redimensionnables dont on peut faire croître la taille en coût amorti $O(1)$. On peut, une fois le tableau rempli, retourner son contenu sous forme de **string**.

  — Créer un **Buffer** :

```
# let b = Buffer.create 1024;;
val b : Buffer.t = <abstr>
```

  — Faire grossir ce buffer en lui ajoutant des chaînes de caractères :

```
# let s = "abc" in Buffer.add_string b s;  Buffer.
    add_string b s;;
- : unit = ()
```

  Cette fonction est utile pour l'encodage.

  — Pour récupérer le contenu du Buffer sous forme de **string** :

```
# Buffer.contents b;;
- : string = "abcabc"
```

  — On peut alternativement faire grossir un buffer en lui ajoutant des caractères un à un :

```
# Buffer.add_char b 'z';;
- : unit = ()
# Buffer.contents b;;
- : string = "abcabcz"
```

- Pour obtenir les fonctionnalités de files de priorité, télécharger l'archive en entrant dans un terminal :

wget https ://nuss-baumcpge.be/public_html/Sup/MP2I/archivePqueue.zip

Cette archive contient une implémentation des files de priorité. Notamment :

  — Le type des données gérées par les files de priorité est

```
type ('a, 'b) data = {priority: 'a; value: 'b} ;;
let d = {priority = 12; value = Leaf 'a'};; (*exemple de
    data*)
```

  Les données sont gérées dans la file de priorité en fonction de leur champ **priority**.

— Pour créer une file de priorité de type **(int, tree) priority_file**, il faut donner la taille (indicative) du tableau sous-jacent, une priorité et une valeur. La file ainsi créée est vide mais possède un espace de stockage dédié initialisé à la valeur donnée. Le tableau sous-jacent est rempli avec une donnée non significative mais du type souhaité.

```
# let q = createpq 3 18 (Leaf 'a');;
val q : (int, tree) priority_file =
{n = 0;
 tbl =
  [|{priority = 18; value = Leaf 'a'}; {priority = 18;
      value = Leaf 'a'};
    {priority = 18; value = Leaf 'a'}; {priority = 18;
        value = Leaf 'a'}|];
 pos = <abstr>}
```

— Pour insérer une nouvelle donnée, on utilise **insert** :

```
# insert {priority = 32; value = Leaf 'z'};;
- : (int, tree) priority_file -> unit = <fun>
# q;;
- : (int, tree) priority_file =
{n = 0;
 tbl =
  [|{priority = 18; value = Leaf 'a'}; {priority = 18;
      value = Leaf 'a'};
    {priority = 18; value = Leaf 'a'}; {priority = 18;
        value = Leaf 'a'}|];
 pos = <abstr>}
```

— Pour retirer la donnée la plus prioritaire :

```
# let d = pop q in d.priority, d.value;;
- : int * tree = (18, Leaf 'a')
# q;;
- : (int, tree) priority_file =
{n = -1;
 tbl =
  [|{priority = 18; value = Leaf 'a'}; {priority = 18;
      value = Leaf 'a'};
    {priority = 18; value = Leaf 'a'}; {priority = 18;
        value = Leaf 'a'}|];
 pos = <abstr>}
#
```