NFP 119 : Programmation Fonctionnelle

Seconde session – Juin 2019

Indications

Le rendu de ce devoir se fait par mail à l'adresse suivante : `Pierre.Courtieu@cnam.fr` avec le programme dans un fichier en attachement nommé `JeanDupont.ml` (adapter à votre nom). Pour vos remarque éventuelles utilisez `(* des commentaires *)` dans le code.

Vous devez faire ce travail seul sans chercher de solution sur internet. La soutenance a pour objectif entre autre d'évaluer votre compréhension du code que vous avez rendu. En particulier il vous sera demandé d'écrire de nouvelles fonctions pour le projet.

Codage de Huffman

On souhaite réaliser un programme de compression de texte. Par exemple, la chaîne de caractères `"Bonjour␣le␣monde"` (`'␣'` représente l'espace) contient 16 caractères. Dans le code ASCII chaque caractère est codé sur 8 bits, il faut donc $8 \times 16 = 128$ bits pour représenter cette chaîne de caractères en ASCII.

Le principe de la compression de Huffman est le suivant : au lieu de représenter chaque caractère sur 8 bits, on utilise un nombre de bits *différent selon les caractères*. Pour représenter les caractères les plus fréquents on utilisera un codage plus court, et pour les caractères les moins fréquents un codage plus long. On obtient ainsi une représentation plus compacte de la chaîne de caractère. Il faut pour cela remplacer le codage ASCII par *un codage spécialement calculé pour ce texte* (pour chaque texte à compresser un codage différent est calculé).

Dans ce devoir nous nous intéressons à l'opération de décompression : étant donnés une suite de bits `lb` et une *clé* de décodage `c` on calcule la chaîne représentée par `lb`. La clé est un arbre binaire. La figure **??** montre un arbre de décodage pour la chaîne de caractère `"Bonjour␣le␣monde"`.

Chaque caractère est codé par la suite de bits permettant d'aller de la racine de l'arbre à la feuille contenant ce caractère. Par exemple on voit dans cet exemple que le caractère `n` est codé par la suite `010` alors que `u` est codé par `1001`. Le principe du décodage est le suivant : l'arbre binaire est utilisé comme ≪ arbre de décision ≫ : à chaque fois qu'on lit un bit, si le bit est `0` on prend le fils gauche dans l'arbre, sinon on prend le fils droit. Lorsqu'on arrive sur une feuille cela signifie qu'on a reconnu un caractère. On enregistre alors le caractère et on recommence à lire `lb` en repartant du sommet de l'arbre binaire.

Supposons par exemple qu'on veut décoder la suite de bits suivante :`0011000` avec la clé de la figure **??**. On commence par suivre les bits dans l'arbre binaire et on reconnaît la séquence `001` qui correspond au caractère `'e'` dans l'arbre et il reste à reconnaître la suite de la liste : `1000`, on recommence en partant du haut de l'arbre et cette fois on reconnaît le caractère `r`. La chaîne décodée est donc `"er"`.

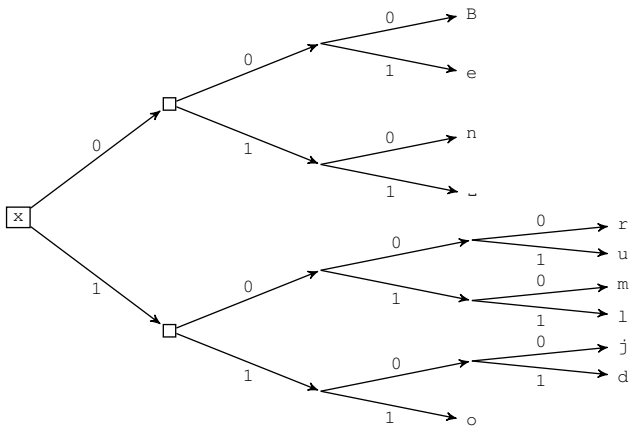

FIGURE 1 – Arbre binaire de décodage de la chaîne "Bonjour le monde"

Devoir

On se donne le type `bin` à 2 éléments, comparable au type `bool` mais avec les deux valeurs `O` (la lettre o majuscule) et `I` (la lettre i majuscule) pour représenter les bits.

```
type bit = O | I ;;
```

Par exemple la valeur `0011` sera représentée par la liste suivante (de type `bit list`) :

```
let l = [ 0 ; 0 ; I ; I ];;
```

On se donne également le type suivant pour les arbres de décodage :

```
type cle =
| Feuille of char
| Choix of cle * cle;;
```

Dans le fichier `huffman.ml` fourni, sans changer la définition de ces deux types, programmez les fonctions suivantes :

1. Fonction `encode_char (ch:char) (c:cle): (bit list) option` qui retourne `None` si le caractère n'apparaît pas dans l'arbre, et le code du caractère `ch` (ex : `Some [O;O;I]`) selon la clé `c`.
2. Fonction `decode_char (lb:bit list) (c:cle): char * bit list` qui reconnaît le premier caractère de la liste de bit `lb` selon la clé `c` et le reste de la liste `lb`.
3. Fonction `decode (lb:bit list) (c:cle): string` qui reconnaît toute la chaîne de caractères codée dans `lb` selon `c`. On utilisera la fonction demandée à la question précédente (même si vous n'avez pas réussi à l'écrire).

Le fichier `huffman.ml` contient la déclaration de ces types ainsi que des tests permettant de vérifier le fonctionnement de la dernière fonction sur des exemples. Vous pouvez bien entendu ajouter des tests.

Rappels

— La fonction `String.make n c` retourne la chaîne de caractère de longueur `n` dont tous les caractères sont le caractère `c`. Par exemple `List.make 3 'r'` retourne la chaîne `"rrr"` et `List.make 1 't'` retourne la chaîne `"t"`.

— Le type `'a option` est défini comme ceci :

```
type 'a option =
| None
|Some of 'a
```

et sert à représenter les valeurs *absentes* ou *présentes*. `None` représente la valeur absente, et `Some x` représente une valeur présente (données par `x`). Par exemple `Some 'a'` est de type `char option`, `Some 12` est de type `int option`.