TP 7 : Arbres binaires et codage de Huffman

I Arbres binaires stricts étiquetés aux feuilles

I.1 Rappels

Soit E un ensemble non vide. On définit un arbre binaire strict étiqueté (aux feuilles) par E de la manière suivante :

- Si x est un élément de E alors $a = \bullet x$ est un arbre binaire strict étiqueté. (x est alors l'unique feuille de a.)
- Si a_0 et a_1 sont deux arbres binaires alors $a = a_0 \blacksquare a_1$ est un arbre binaire strict étiqueté. (a_0 est le sous-arbre gauche de a et a_1 le sous-arbre droit. Les feuilles de a sont exactement celles de ces deux sous-arbres.)

Il est usuel de donner une représentation graphique des arbres sous forme arborescente. Dans celles-ci, nous représenterons les nœuds par ■ et les feuilles par •.

La racine d'un arbre est le nœud (ou éventuellement l'unique feuille) située au sommet de ces constructions

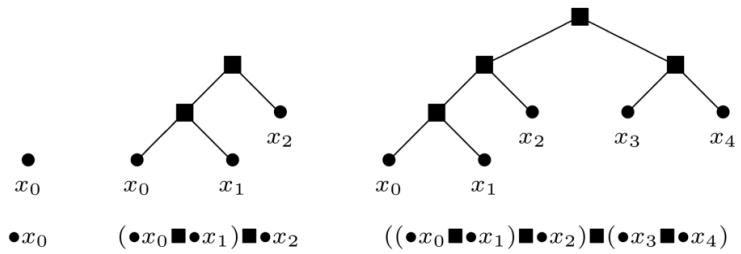

FIGURE 1 – Trois exemples d'arbres

Nous représenterons en CAML un arbre binaire strict étiqueté par des valeurs de type $'a$ grâce au type somme récursif `'a arbre` défini par la déclaration :

```
type 'a arbre =
    Feuille of 'a
  | Noeud of 'a arbre *  'a arbre
;;
```

Cette définition de type correspond à la définition formelle de la notion d'arbre que nous avons donnée précédemment. `Feuille x` représente en CAML l'arbre à une feuille $\bullet x$. De même, si `a0` et `a1` représentent les arbres a_0 et a_1 , `Noeud (a0, a1)` représente l'arbre $a_0 \blacksquare a_1$.

I.2 Manipulations élémentaires

Question 1. Donner la représentation en CAML des arbres donnés en exemple ci-dessus (figure 1). On définira des liaisons globales que l'on appelera respectivement `t0` , `t1` et `t2` . De plus on fabriquera en fait des arbres d'entiers en remplaçant x_0 par 0, x_1 par 1 etc...

Question 2. Écrire une fonction `nb_noeuds` `'a arbre -> int` qui calcule le nombre de noeuds d'un arbre. Vérifier qu'elle fonctionne correctement sur les trois arbres d'entiers que vous avez définis.

Question 3. Écrire une fonction `hauteur` `'a arbre -> int` qui calcule la hauteur d'un arbre. Vérifier qu'elle fonctionne correctement sur les trois arbres d'entiers que vous avez définis.

Question 4. On suppose que l'on travaille avec un arbre de nombres. Écrire une fonction `max_arbre` `'a arbre -> 'a` qui calcule la plus grande étiquette présente dans l'arbre. Vérifier qu'elle fonctionne correctement sur les trois arbres d'entiers que vous avez définis.

II Mots binaires

Un mot binaire mb est une liste finie de 0 et 1. Par exemple 0010 est un mot binaire représenté en CAML par la liste `[0; 0; 1; 0]` . Nous considérerons également le mot vide, noté ϵ et représenté par la liste vide.

Étant donné un arbre binaire a, un mot binaire mb permet de désigner un sous-arbre de a. Celui-ci est obtenu en parcourant a depuis la racine tout en lisant mb bit par bit : à la lecture d'un 0 on descend vers le fils gauche du nœud courant et à la lecture d'un 1, vers le fils droit.

Considérons à nouveau l'arbre :

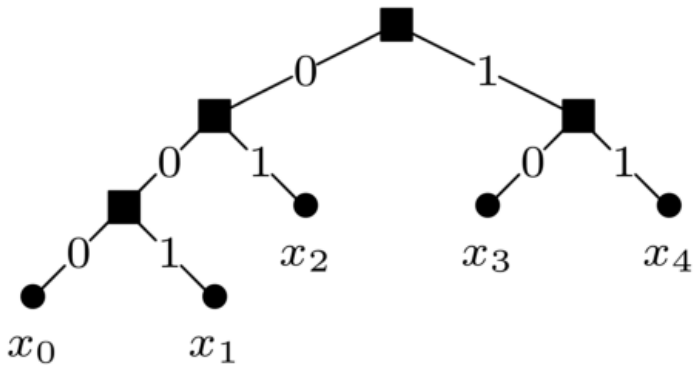

FIGURE 2 – Parcours guidé par un mot binaire

Dans cet exemple, le mot 00 désigne le sous arbre ($\bullet x_0 \blacksquare \bullet x_1$) alors que 001 désigne $\bullet x_1$. Cependant, les mots 010, 101 ou 0010 ne correspondent à aucun sous-arbre.

Question 5. Écrire une fonction `sous_arbre` `'a arbre -> int list -> 'a arbre` qui prend pour arguments un arbre a et un mot binaire mb. Cette fonction retourne le sous-arbre de a désigné par mb. Si le mot binaire mb ne désigne pas un sous-arbre de a, une exception sera levée.

Question 6. Écrire maintenant une fonction `read` `'a arbre -> int list -> ('a * int list)` qui prend en argument un arbre a et un mot binaire mb. Cette fonction parcourt l'arbre a en lisant le mot mb jusqu'à arriver sur une feuille. La fonction retourne alors l'étiquette de la feuille atteinte et la partie du mot mb qui n'a pas été lue. Si le mot mb ne permet pas d'atteindre une feuille, une exception sera levée.

III Code de HUFFMAN

III.1 Décodage

Pour représenter les lettres de l'alphabet, les chiffres, les symboles de ponctuation, et d'une manière générale tous les caractères qui apparaissent dans un fichier informatique, on associe à chacun d'entre-eux un mot binaire. Dans une représentation habituelle, la longueur du mot binaire est la même pour tous les caractères, un octet par exemple.

Le principe du codage de HUFFMAN est d'associer à chaque symbole du texte à encoder un code binaire qui est d'autant plus court que le caractère correspondant a un nombre d'occurrences élevé dans le texte à encoder, ce qui permet de diminuer la taille nécessaire à la représentation du texte.

Un code de HUFFMAN consiste en un arbre binaire dont les feuilles sont étiquetées par des caractères. Le mot binaire associé à chaque caractère c est celui qui mène de la racine de l'arbre à la feuille étiquetée par c selon la méthode vue dans la partie précédente (mots binaires).

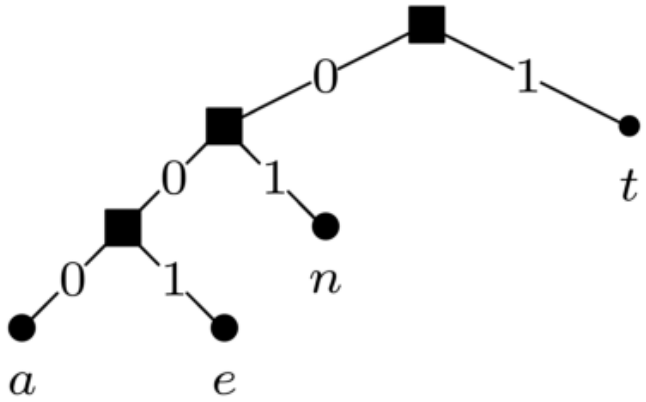

FIGURE 3 – Arbre du code de HUFFMAN du mot `"tentant"`

On représente alors une chaîne de caractères par la concaténation des mots binaires correspondant à chacun de ses caractères. Par exemple, dans le code précédent, la chaîne `"tentant"` est représentée par 1001011000011 ($1 \cdot 001 \cdot 01 \cdot 1 \cdot 000 \cdot 01 \cdot 1$).

Question 7. Écrire une fonction `decode` `char arbre -> int list -> string` qui prend pour argument un code de HUFFMAN et un mot binaire. La fonction retourne la chaîne de caractères représentée par le mot binaire dans le code.

On rappelle que pour concaténer deux chaînes de caractères `s1` et `c2` en OCAML il faut écrire `s1 ^s2` . Par exemple `"bon"^"jour"` va rendre `"bonjour"` .

Par ailleurs pour transformer un caractère en chaîne de caractères de longueur 1, on utilisera l'astuce `String.make 1 c` .

III.2 Construction du code

Nous nous intéressons maintenant à la construction d'un code de HUFFMAN permettant de représenter un texte d'une manière la plus concise possible. La difficulté réside dans le fait que, pour obtenir un codage efficace, il faut choisir les codes des différents caractères en tenant compte de leurs fréquences respectives. La méthode proposée ici nécessite de calculer dans un premier temps le nombre d'occurences nc de chaque caractère c présent dans le texte, afin de former la liste m des couples $(nc, \bullet c)$ (le deuxième élément du couple est une arbre-feuille étiqueté par le caractère c).

L'algorithme consiste alors à itérer le processus suivant sur la liste m jusqu'à ce que celle-ci soit réduite à un élément :

— Retirer de la liste m les deux couples (n_1, a_1) et (n_2, a_2) tels que n_1 et n_2 soient minimaux (i.e. les deux plus petites valeurs des nombres d'occurrences) ;

— Ajouter à la liste m le couple $(n_1 + n_2, a_1 \blacksquare a_2)$.

L'algorithme se termine lorsque la liste m est réduite à un couple (n, a) : a est alors l'arbre du code de HUFFMAN recherché. Pour obtenir une implémentation raisonnablement efficace, on maintiendra la liste m triée dans l'ordre des nc croissants. Ainsi, les éléments minimaux apparaîtront toujours en tête.

Question 8. Quelle est la liste m obtenue si le texte considéré est le mot `"tentant"` ? Quel est l'arbre construit par l'algorithme ?

Question 9. Écrire une fonction `insert` `int * 'a -> (int * 'a) list -> (int * 'a) list` prenant pour argument un couple (n, x) où n est un entier et x un objet de type $'a$ quelconque, et une liste l supposée triée selon les n croissants. La fonction retourne la liste obtenue à partir de l en ajoutant (n, x) de manière à maintenir le tri.

Question 10. Écrire une fonction `merge` (int * char arbre) list -> char arbre prenant pour argument une liste de couples de la forme $(nc, \bullet c)$ supposée triée selon les nc croissants. Cette fonction réduit la liste comme décrit dans l'algorithme ci-dessus afin d'obtenir le code de HUFFMAN correspondant.

Question 11. En Déduire une fonction `huffman` (int * char) list -> char arbre qui calcule l'arbre de HUFFMAN correspondant à la liste de couples (nc, c) (pas nécessairement triée) indiquant le nombre d'occurrences de chaque caractère dans un texte.

III.3 Codage

Question 12. Écrire une fonction `extract` qui prend pour argument l'arbre d'un code de HUFFMAN et qui calcule la liste des couples (c, mbc) où mbc est le mot binaire représentant le caractère c dans le code.

Question 13. En déduire enfin une fonction `code` qui prend pour argument un code de HUFFMAN ainsi qu'une chaîne de caractères et qui retourne le mot binaire représentant la chaîne de caractères dans le code de HUFFMAN.

Question 14. Tester vos fonctions avec le code proposé dans le fichier accompagnant cet énoncé. On y trouvera en particulier une fonction toute prête fabriquant la liste des occurrences (sous la forme d'une liste de (n, c), avec n entier et c caractère) des caractères d'une chaîne de caractères.