TP 7 - Codage de Huffman

3 février 2011

Pour toutes questions, suggestions, remarques ou autres, n'hésitez pas à m'envoyer un mail à edbonnet@hotmail.com en mettant en objet [TP Caml].

1 Introduction

Dans ce TP, nous allons implémenter le codage de Huffman qui est à l'origine de la compression zip et utilise la structure d'arbre binaire. Son principe général est que le code d'un caractère est d'autant plus court que son nombre d'occurrences est élevé (au lieu d'avoir un code de taille constante n pour les 2^n caractères à représenter). Pour simplifier les choses on travaillera avec les 26 lettres de l'alphabet en minuscule et 6 autres caractères dont l'espace, le point et la virgule (le choix des 3 autres vous étant laissé).

2 Construction de l'arbre

Question 1 *Écrire (ou reprendre d'un précédent TP) une fonction* `occurrences : string -> int vect` *qui renvoie le nombre d'occurrences dans un texte de chacun des 32 caractères autorisés.*

L'arbre binaire dit de Huffman se construit de la façon suivante. On initialise une forêt à 32 arbres réduits à une feuille contenant chaque caractère et son nombre d'occurrences (son poids). Tant qu'il y a plusieurs arbres dans la forêt, on prend les deux arbres a_0 et a_1 de poids minimum et on les fusionne en un arbre de poids la somme des poids de a_0 et a_1 où on lit "0" sur l'arête qui mène à a_0 et "1" sur l'arête qui mène à a_1 . Le code de Huffman d'un caractère est alors donné par l'arbre comme le mot lu sur la branche qui mène à la feuille contenant le caractère en question. On pourra travailler avec le type suivant : `type arbre = N of int * arbre * arbre | F of int * char`.

Question 2 *Comment est l'arbre de Huffman dans un texte où tous les caractères apparaissent autant de fois ? Comment est l'arbre de Huffman dans un texte où un seul caractère apparait ? Dans ces deux cas, comparer la taille du codage de Huffman est celle du codage naïf.*

Question 3 *Montrer qu'un code de Huffman ne peut pas être plus long que le texte non compressé.*

Question 4 *Écrire une fonction* `construit_arbre : string -> arbre` *qui prend un texte en entrée et renvoie son arbre de huffman.*

3 Compression et décompression

Une fois obtenu l'arbre de Huffman, l'essentiel est fait. Il reste cependant à écrire quelques menues fonctions auxiliaires pour compresser et décompresser un texte.

Question 5 *Écrire une fonction qui compresse un texte donné en argument.*

Question 6 *Écrire une fonction qui prend un texte compressé et son arbre de huffman et qui renvoie le texte décompressé.*

4 Affichage

On va finir ce TP en utilisant la bibliothèque graphique de caml pour afficher l'arbre de huffman d'un texte. La fonction `open_graph : string -> unit` ouvre la fenêtre graphique de caml. Vous aurez sans doute aussi besoin des fonctions : `moveto : x:int -> y:int -> unit` qui déplace le curseur vers le pixel de coordonnées (x,y), `lineto : x:int -> y:int -> unit` qui trace un trait depuis le point courant jusqu'au point (x,y) et change le point courant en (x,y) `draw_string : string -> unit` qui écrit un mot au point courant.

Question 7 *Écrire une fonction qui prend un texte en entrée et affiche relativement élégamment son arbre de huffman.*

Vous pouvez vérifier que vos fonctions marchent bien en comparant sur un même texte votre arbre affiché et celui généré à l'url suivante `http://huffman.ooz.ie/`.