Travaux Pratiques sur Caml Light n°4

Cours d'Informatique de l'École Centrale d'Électronique

—Ingénieur Deuxième Année—

Cette feuille porte sur l'algorithme de codage de Huffman. Cet algorithme permet notamment de compresser un fichier. Nous voyons comment réaliser un programme implémentant cette technique. Les objectifs sont la mise en évidence des avantages de la programmation fonctionnelle au sein d'un groupe de développement de logiciels.

Le codage statique d'un fichier est réalisé en deux passes (lectures). La première initialise les fréquences d'apparition de chaque symbole contenu au moins une fois dans le texte source : les symboles possibles forment l'ensemble du code *ASCII*-8 *bits*. La lecture de ces fréquences permet de construire un arbre de poids minimal, dit arbre de Huffman. C'est lors de la construction de cet arbre que le fichier de sauvegarde des fréquences est constitué (`.frq`).

Ensuite, le codage du fichier est réalisé lors d'une deuxième lecture du fichier source : à chaque caractère correspond un codage dans l'arbre, qui est émis dans un *buffer* (tableau) d'entiers ; dès que le *buffer* est plein, on émet le caractère *ASCII* correspondant à la suite de 1 et de 0 contenue dans le *buffer* : ceci nous permet donc de produire un fichier compressé.

L'arbre de Huffman est implémenté sous forme d'un arbre binaire, dont les nœuds sont constitués d'un poids, d'un pointeur sur le fils droit, d'un pointeur sur le fils gauche, de la liste de toutes les lettres contenues dans le fils gauche et enfin de la liste de toutes les lettres du fils gauche et du fils droit : cette liste est utilisée lors de la construction de l'arbre (`int * HuffmanTree * HuffmanTree * int list * int list`). Les feuilles contiennent une fréquence d'apparition associée à une lettre dans cet ordre dans le couple `int * int`. Le type choisi est donc :

```
#type HuffmanTree =
   Leaf of int * int
 | Node of int * HuffmanTree * HuffmanTree * int list * int list;;
Type HuffmanTree defined.
```

Pour le compte des fréquences, un tableau sera déclaré afin d'accéder à une case de ce tableau en temps constant, ce qui aurait été impossible avec une liste. Enfin, un *buffer* de type tableau a été défini pour filtrer l'émission des codes dans le fichier destination.

Signalons que nous allons avoir besoin de définir un caractère en plus de l'alphabet standard, pour indiquer au décodeur la fin du codage. En effet, cette fin ne correspond pas nécessairement à la fin du fichier codé : le dernier octet du fichier peut ainsi comporter moins de 8 *bits* significatifs. Ce caractère sera représenté par la constante `EOF` dans notre programme. L'alphabet standard comporte 256 caractères :

```
#let NL = 256;;
NL : int = 256
```

et la définition du caractère `EOF` s'écrit donc :

```
#let EOF = NL + 1;;
EOF : int = 257
```

Dans la mesure où les opérations de codage et de décodage présentent des similitudes, certaines des fonctions sont définies pour servir aux deux opérations. Le sens dans lequel elles sont utilisées est géré par un argument de type :

```
#type HuffmanWay = C | D;;
Type HuffmanWay defined.
```

le constructeur `C` désignant un codage, le constructeur `D` un décodage.

Nous divisons le développement du programme est plusieurs parties. Chaque partie devra être programmée en suivant scrupuleusement toutes les spécificités des questions. En particulier, le nom de chaque fonction, l'ordre des arguments ou encore le type des objets devront être respectés à la lettre afin que l'ensemble soit cohérent.

1 Lecture dans le fichier source, stockage des fréquences

L'idée est que les fréquences d'apparition des lettres d'un fichier source sont stockées dans un tableau, lequel sera réduit en éliminant les lettres non utilisées, et transformé en une liste de couples.

▸ **Exercice 1.** *Donnez une fonction qui crée un tableau de* `n` *couples (fréquence, lettre) correspondant à* `n` *lettres avec (par défaut) une fréquence d'apparition nulle :*

```
BuildCellTab : int -> (int * int) vect = <fun>
```

On définit alors `ref_FreqTab` *comme une référence sur un tel tableau, initialisé avec les* `NL` *lettres possibles accompagnées de la lettre spéciale* `EOF` *apparaissant (par nature) une fois :*

```
#let ref_FreqTab = ref (concat_vect (BuildCellTab NL)
                                    (make_vect 1 (1,EOF)));;
ref_FreqTab : (int * int) vect ref = ref [| ...; ...|]
```

▸ **Exercice 2.** *On s'occupe maintenant du remplissage du tableau. Donnez une fonction* `FreqCount_rec` *qui se charge de lire dans l'input-channel* `ic` *(flot d'entrée) un caractère et de modifier la fréquence d'apparition de la lettre dans le tableau* `!ref_FreqTab`.

```
FreqCount_rec : in_channel -> 'a = <fun>
```

La fonction `FreqCount` *lance la lecture et attend la bulle* `End_of_file` *en fin de lecture :*

```
#let FreqCount ic =
   try FreqCount_rec ic with
     End_of_file -> flush std_out;;
FreqCount : in_channel -> unit = <fun>
```

2 Gestion des fréquences

▸ **Exercice 3.** *Donnez une fonction* `VectInsert` *qui insère l'élément (fréquence,lettre) dans une liste. La liste est triée par fréquences croissantes :*

```
VectInsert : 'a * 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

▸ **Exercice 4.** *Donnez une fonction* `VectInsertDump` *dont l'objet est l'élimination des lettres dont la fréquence est nulle et l'émission dans le flot de sortie (out-channel)* `file_of_freq` *des lettres apparues au moins une fois dans le fichier source ; cette sauvegarde s'effectuera selon le format* `lettre fréquence` *(*`int int`*). De plus, ces lettres seront insérées dans la liste des couples (fréquence, lettre) triées par fréquences croissantes :*

```
VectInsertDump : out_channel -> bool -> int * int ->
                 (int * int) list -> (int * int) list = <fun>
```

La fonction utilise un booléen qui gère le fait que l'on écrit ou pas dans le flot de sortie.

▸ **Exercice 5.** *Donnez une fonction* `SortedList_of_vect` *qui crée la liste triées des couples (fréquence, lettre) dont nous venons de parler dans l'exercice précédent. La fonction utilise en particulier la fonction* `VectInsertDump` *:*

```
SortedList_of_vect : out_channel -> bool -> (int * int) vect -> int ->
                     (int * int) list = <fun>
```

Le premier argument est le flot de sortie, le second contrôle l'écriture dans ce flot, le troisième le tableau de couples, le quatrième le nombre d'éléments du tableau et le résultat est la liste triée des couples.

▸ **Exercice 6.** *Les fonctions suivantes forment la partie symétrique des opérations précédentes. Cela correspond à la phase de lecture du fichier de fréquences et de reconstruction de la liste de couples lors de la phase de décodage. Donnez une fonction* `FreqRead_rec` *lisant un flot d'entrée au format de sauvegarde décrit précédemment et reconstituant le tableau de couples :*

```
FreqRead_rec : in_channel -> 'a = <fun>
```

En fin de fichier, le système produira une exception et par conséquent :

```
#let FreqRead file_of_freq = try FreqRead_rec file_of_freq with
                               End_of_file -> flush std_out;;
FreqRead : in_channel -> unit = <fun>
```

La fonction `Init` *est utilisée en codage et décodage. Elle initialise la liste des couples (lettre,fréquence) selon qu'il s'agit d'un codage (*`C`*) ou d'un décodage (*`D`*) :*

```
#let Init ic oc = function
    C -> FreqCount ic;
         SortedList_of_vect oc true !ref_FreqTab
                                (vect_length !ref_FreqTab)
  | D -> FreqRead ic;
         SortedList_of_vect oc false !ref_FreqTab
                                    (vect_length !ref_FreqTab);;
Init : in_channel -> out_channel -> HuffmanWay -> (int * int) list = <fun>
```

3 Construction de l'arbre de Huffman

On donne tout d'abord quelques fonctions de base :

```
#let BuildLeaf (a,b) = Leaf(a,b);;
BuildLeaf : int * int -> HuffmanTree = <fun>
#let Weight_of_HuffmanTree = function Leaf(p,_) -> p
                              | Node(p,_,_,_,_) -> p;;
Weight_of_HuffmanTree : HuffmanTree -> int = <fun>
#let Letters_of_HuffmanTree = function Leaf(_,l) -> [l]
                               | Node(_,_,_,_,l) -> l;;
Letters_of_HuffmanTree : HuffmanTree -> int list = <fun>
```

▸ **Exercice 7.** *La construction de l'arbre de Huffman se fait en insérant dans une liste triée le nouveau nœud créé à partir des deux premiers éléments de la liste de feuilles et nœuds servant à la construction. Donnez une fonction* `NodeListInsert` *insérant un nœud dans une liste triée de nœuds :*

```
NodeListInsert : HuffmanTree -> HuffmanTree list ->
                 HuffmanTree list = <fun>
```

▸ **Exercice 8.** *Donnez une fonction* `BuildNode` *qui construit simplement un nœud de l'arbre à partir d'un sous-arbre gauche et d'un sous-arbre droit :*

```
BuildNode : HuffmanTree -> HuffmanTree -> HuffmanTree = <fun>
```

▸ **Exercice 9.** *On arrive alors à la fonction de construction proprement dite, c'est-à-dire à l'algorithme qui prend les deux premiers éléments de la liste de nœuds pour former un nouveau nœud inséré à son tour dans la liste. Donnez la fonction* `BuildHuffmanTree` *correspondante :*

```
BuildHuffmanTree : HuffmanTree list -> HuffmanTree = <fun>
```

Finalement, la fonction `Huffman` *crée l'arbre de Huffman en lançant la fonction* `Build-HuffmanTree` *avec les paramètres apropriés : fichier source, fichier d'émission, sens de l'opération :*

```
#let Huffman ic oc way = BuildHuffmanTree (map BuildLeaf
                                             (Init ic oc way));;
Huffman : in_channel -> out_channel -> HuffmanWay -> HuffmanTree = <fun>
```

4 Émission sur le flot de sortie

Cette partie traite de l'émission des bits sur le canal de sortie. On définit tout d'abord un *buffer* de `SIZ_BUFF` = 8 *bits* qui sera vidé chaque fois qu'il est plein. L'algorithme va donc produire des bits dans ce *buffer*, qui lui-même alimentera sous forme d'un flot de caractères le canal de sortie :

```
#let SIZ_BUFF = 8;;
SIZ_BUFF : int = 8
#let ref_buffer = ref (make_vect SIZ_BUFF 0);;
ref_buffer : int vect ref = ref [|0; 0; 0; 0; 0; 0; 0; 0|]
#let ref_buffer_ind = ref 0;;
ref_buffer_ind : int ref = ref 0
```

le nom `ref_buffer_ind` désignant un curseur sur le tableau pointé par le nom `ref_buffer`.

▸ **Exercice 10.** *Donnez des fonctions* `int_of_list`, `int_of_vect`, `list_of_int` *et* `vect_of_int` *permettant de transformer une liste ou un vecteur de* 1 *et* 0 *en un entier (ou vice versa) dont la représentation en binaire est codée par cette liste—bits de poids faible en début de liste :*

```
int_of_list : int list -> int = <fun>
int_of_vect : int vect -> int = <fun>
list_of_int : int -> int list = <fun>
vect_of_int : int -> int vect = <fun>
```

La fonction principale d'émission d'un bit de codage est la fonction `put_byte`. *Elle remplit le buffer et le vide lorsqu'il est plein, en émettant en sortie un caractère sur* `oc` *:*

```
#let put_byte oc byt = if (!ref_buffer_ind)=SIZ_BUFF then
                         begin
                           let car = int_of_vect !ref_buffer in
                             output_char oc (char_of_int car);
                             fill_vect !ref_buffer 0 SIZ_BUFF 0;
                             ref_buffer_ind := 1;
                             vect_assign !ref_buffer 0 byt
                         end
                       else
                         begin
                           vect_assign !ref_buffer !ref_buffer_ind byt;
```

```
                          incr ref_buffer_ind
                        end;;
put_byte : out_channel -> int -> unit = <fun>
```

La fonction suivante se charge de vider complètement le buffer en fin de codage :

```
#let FlushVect oc = let car = int_of_vect !ref_buffer in
                      output_char oc (char_of_int car);;
FlushVect : out_channel -> unit = <fun>
```

5 Génération du code

On déclare tout d'abord quelques exceptions afin de gérer les erreurs : `NotFound` pour une lettre non trouvée dans l'arbre, `End_of_code` pour indiquer que le décodage est terminé :

```
exception NotFound of int;;
exception End_of_code;;
```

▸ **Exercice 11.** *Donnez une fonction* `Letter2Buffer` *qui recherche le code d'une lettre* `lettre_int` *dans un arbre de Huffman : elle émet des 1 et 0 dans le buffer* `!ref_buffer` *:*

```
Letter2Buffer : out_channel -> int -> HuffmanTree -> unit = <fun>
```

▸ **Exercice 12.** *Donnez une fonction* `ReadCode` *qui lit des caractères sur un flot d'entrée* `ic` *puis émet leurs codages (à partir de l'arbre de Huffman* `tree`*) sur le flot de sortie* `oc` *:*

```
ReadCode : HuffmanTree -> out_channel -> in_channel -> 'a = <fun>
```

La fonction suivante teste l'existence d'un fichier :

```
#let exist_file name =
  try (let ic = open_in name in close_in ic;true) with
    sys__Sys_error _ -> false
  |                _ -> true;;
exist_file : string -> bool = <fun>
```

La fonction `PrintInfo` affiche des informations sur la compression :

```
#let PrintInfo ic oc t =
  print_string("Source file size (bytes)     : ");
  print_int(in_channel_length ic);
  print_string("\nTarget file size (bytes)     : ");
  print_int(out_channel_length oc);
  print_string("\nCompression                  : ");
```

```
  let r = (100*(t + (out_channel_length oc))) /
          (in_channel_length ic) in
    print_int(r);print_string("%\n");;
PrintInfo : in_channel -> out_channel -> int -> unit = <fun>
```

Enfin, la fonction `HuffmanCode` est la fonction lançant le codage d'un fichier. Elle crée en particulier un fichier `.frq` qui représente la sauvegarde des couples (lettre, fréquence d'apparition) :

```
#let HuffmanCode src dest =
  let ic = open_in src and file_of_freq = open_out (dest^".frq") in
    let h = (Huffman ic file_of_freq C) in
      seek_in ic 0;
      let oc = open_out dest in
        try (ReadCode h oc ic) with
           NotFound a -> print_string("Error on "); print_int a;
                         exit 1
        | End_of_file -> Letter2Buffer oc EOF h;
                         FlushVect oc; flush oc;
                         if ((exist_file src) & (exist_file dest) &
                             (exist_file (dest^".frq"))) then
                         begin
                           flush file_of_freq;
                           let t = out_channel_length file_of_freq in
                             PrintInfo ic oc t;
                             print_string
                               "Frequencies file size (bytes): ";
                             print_int(t); print_string("\n")
                         end
                         else
                           print_string "Can't find one needed file...\n";
                         close_in ic; close_out oc;
                         close_out file_of_freq;;
HuffmanCode : string -> string -> unit = <fun>
```

6 Décodage d'un flot d'entrée

Nous traitons maintenant de la partie décodage du fichier codé. Nous donnons tout d'abord une fonction initialisant un *buffer* à partir du premier caractère d'un flot d'entrée :

```
#let ReadVect ic = vect_of_int
                     (((int_of_char (input_char ic))+NL) mod NL);;
ReadVect : in_channel -> int vect = <fun>
```

▸ **Exercice 13.** *Donnez une fonction* `ReadDecodeWrite` *qui se charge de lire dans* `ic` *et d'écrire dans* `oc` *la "traduction" par l'arbre de Huffman du buffer : lecture du buffer, émisssion d'un caractère et arrêt à la rencontre du caractère* `EOF` *de fin de codage :*

```
ReadDecodeWrite : in_channel -> out_channel -> HuffmanTree ->
                  int -> HuffmanTree -> int vect -> 'a = <fun>
```

Le troisième paramètre est l'arbre de Huffman complet alors que le cinquième est un sous-arbre de cet arbre qui évolue au fur et à mesure de la lecture des 0 et 1 dans le buffer. La fonction est ainsi récursive et remplit le buffer dès qu'il a été lu entièrement. Chaque fois que le cinquième argument est une feuille on émet le caractère correspondant sur le flot de sortie ou l'exception `End_of_code` *si on est sur le caractère* `EOF`*. Le quatrième argument est le curseur du buffer d'émission.*

La fonction principale de décodage s'écrit alors :

```
#let HuffmanDecode src dest =
   let ic = open_in src and file_of_freq = open_in (src^".frq") in
     let h = Huffman file_of_freq std_out D and oc = open_out dest in
       let buff = make_vect 0 0 in
         try ReadDecodeWrite ic oc h 0 h buff with
           End_of_code -> flush oc; close_in ic; close_out oc;
                          close_in file_of_freq;;
HuffmanDecode : string -> string -> unit = <fun>
```

7 Analyse de la ligne de commande et compilation

Les lignes suivantes analysent la ligne de commande et lancent le codage ou décodage :

```
let Usage command =
  print_string("Usage: "^command.(0)^" <-[cd]> <src>");
  print_string " <target>\n";
  print_string "is a static Huffman (de)coder program.\n";;

let command = sys__command_line in if vect_length command = 4
  then
     match command.(1) with
       "-c" -> HuffmanCode (command.(2)) (command.(3))
     | "-d" -> HuffmanDecode (command.(2)) (command.(3))
     | "-h" -> Usage command
     |    _ -> print_string("illegal option\n"); Usage command
  else Usage command;;
```

▸ **Exercice 14.** *Utilisez* `camlc` *pour compiler le programme complet. Essayez le programme pour voir ses performances.*