Rapport de projet

Audric Schiltknecht

Résumé

Implémentation de la compression de Huffman en **CaML**.

Table des matières

1 Présentation générale **2**
1.1 Introduction . . . 2
1.2 Fichiers fournis . . . 2
1.3 Spécifications . . . 2

2 Codage **2**
2.1 Méthode de Burrows-Wheeler . . . 2
2.1.1 Présentation . . . 2
2.1.2 Codage . . . 3
2.1.3 Décodage . . . 4
2.2 Algorithme de transformation Move To Front . . . 7
2.2.1 Présentation . . . 7
2.2.2 Encodage . . . 7
2.2.3 Décodage . . . 9
2.3 Codage de Huffman . . . 10
2.3.1 Principe . . . 10
2.3.2 Représentation d'un arbre de Huffman en *CaML* . . . 11
2.3.3 Construction de l'arbre . . . 11
2.3.4 Compression d'un texte . . . 12
2.3.5 Décodage . . . 14

3 Tests **15**

4 Conclusion **18**

5 Listings **18**

1 Présentation générale

1.1 Introduction

L'objectif de ce projet est de réaliser une compression suivant la méthode de Huffman. Cette méthode s'appuie sur une compression par arbre. Pour augmenter l'efficacité de la compression, on implémentera des algorithmes de « réorganisation »et de codage des données.

1.2 Fichiers fournis

Les fichiers suivants nous ont été fournis :

`huffman.mli` interface du module de compression/décompression de Huffman

`burrows_wheeler.mli` interface du module de transformation de Burrows-Wheeler

`movetofront.mli` interface du module de transformation « Move To Front »

`defaut.cmo`, `defaut.cmi`, `defaut.mli` Modules des fonctions de codage - décodage par défaut, utilisées pour le deboggage.

`main.cmo` module implémentant l'interface de l'application

1.3 Spécifications

Ce projet devra être implémenté de façon fonctionnelle, c'est-à-dire sans utiliser de tableaux, références ou procédures.
La clarté et la concision du code seront primordiales, en dépit de sa rapidité lors de l'exécution.

2 Codage

Dans les explications, on prendra souvent des caractères alphanumériques pour illustrer les propos. Cependant, il est demandé de coder l'ensemble du projet de façon générique.

2.1 Méthode de Burrows-Wheeler

2.1.1 Présentation

On applique en premier lieu un codage dit de `Burrows-Wheeler`. Cette méthode permet de rapprocher les diverses occurrences d'un même caractère.

2.1.2 Codage

Dans la pratique, il faut engendrer l'ensemble des rotations de la séquence initiale, les ranger dans l'ordre usuel (lexicographique), et retourner la position de la séquence dans cet ensemble, ainsi que la liste formée de l'ensemble des derniers caractères de chacune des rotations.

Exemple Soit la séquence TEXTE à coder.

Position	$s1$	$s2$	$s3$	$s4$	$s5$
1	E	T	E	X	T
2	E	X	T	E	T
3	T	E	T	E	X
4	T	E	X	T	E
5	X	T	E	T	E

La fonction devra alors retourner le couple (4,TTXEE).

La fonction *CaML* devra vérifier la spécification suivante :

```
val encode : 'a list -> int * 'a list
```

Analyse de l'algorithme Nous avons besoin de définir plusieurs fonctions auxiliaires pour coder cet algorithme :

- Générer la séquence des rotations
- Les classer dans l'ordre lexicographique
- Trouver la position de la séquence dans la liste
- Récupérer le dernier caractère de chaque nouvelle séquence.

Génération de la séquence Le principal problème dans le codage de cette fonction est de savoir quand arrêter les rotations. Plusieurs méthodes sont possibles, j'ai décidé de passer en paramètre la taille de la séquence initiale, ainsi que le nombre de rotations déjà effectuées. En effet, pour une séquence de taille n, il y a n rotations différentes possibles.

On obtient donc la fonction de prototype suivant :

```
val create_seq : 'a list -> int -> int -> 'a list list
```

Classement Il faut ensuite classer l'ensemble de ces séquences dans l'ordre lexicographique. J'ai choisi d'utiliser l'algorithme de *tri par fusion*, qui constitue l'algorithme de tri le plus efficace que nous ayons déjà vu et étudié en cours. De plus, cet algorithme possède une complexité d'exécution de l'ordre de $\Theta(n \log_2 n)$. Comme notre algorithme devra s'appliquer à des textes relativement importants, la séquence initiale, qui sera constituée de l'ensemble des caractères de ce texte, engendrera une séquence de rotations de taille importante (n^2 caractères au total !).

L'implantation des fonctions nécessaires à la réalisation de ce tri ne pose pas de problème particulier :

- `val cut : 'a list -> 'a list * 'a list` qui découpe une liste en deux sous-liste de taille égale à un élément près.
- `val merge : ('a -> 'a -> bool)-> 'a list -> 'a list -> 'a list` qui joint deux liste trié suivant un ordre en une liste triée
- `val sort : ('a -> 'a -> bool)-> 'a list -> 'a list` qui réalise le tri

Recherche de la séquence d'origine Maintenant que nous disposons d'une liste triée des diverses séquences obtenues par rotation de l'initiale, il nous faut chercher cette dernière dans la liste. La liste est parcourue élément par élément : si l'élément courant est celui cherché, la fonction retourne 1, sinon, elle ajoute 1 à son appel récursif. Si l'élément n'est pas dans la liste, c'est-à-dire que l'on obtient la liste vide, alors il y a une exception.

On obtient la fonction suivante :

```
val find_pos : 'a -> 'a list -> int
```

Récupération des derniers caractères Pour pouvoir constituer la liste des derniers caractères, on va stocker dans un accumulateur le caractère concerné, obtenu par extraction du premier dans la liste obtenue par `reverse` de la liste courante. L'accumulateur est obligatoire ici car l'on souhaite obtenir les derniers caractères dans l'ordre.

La fonction possède le prototype suivant :

```
val last_char : 'a list list -> 'a list
```

La fonction principale Cette fonction vérifie juste si l'on est pas dans n cas particulier. Sinon, elle appelles les fonctions précédentes pour faire passer les paramètres adéquats à chacune.

Dans cette transformation, il n'y a qu'un cas particulier : si la séquence initiale est vide. J'ai choisi de renvoyer le couple $(0, [])$.

2.1.3 Décodage

Le décodage est légèrement plus complexe. Il faut commencer à trier la séquence de caractères reçue.

Exemple Soit le couple reçu (4,TTXEE).

Position	$s5$	$s1$
1	T	E
2	T	E
3	X	T
4	E	T
5	E	X

La position transmise nous permet d'obtenir le premier caractère du mot décodé : c'est celui de la liste triée. Il faut ensuite compter le nombre d'occurrence de ce caractère dans la liste triée, et trouver la position dans la liste reçue de cette occurrence. Le caractère de la liste triée correspondant à cette position est le deuxième caractère du mot décodé. On continue ainsi jusqu'à retrouver la position reçue.

Exemple Dans notre exemple, le caractère correspondant à la position 4 dans la liste triée est T. C'est aussi la seconde occurrence de ce caractère dans cette liste. On va en chercher la seconde dans la liste reçue : elle est à la position 2. Le caractère correspondant est E. On a donc reconstitué le mot : TE. On continue comme ça jusqu'à retomber sur la position 4, qui traduit le fait que le mot est complet.

La fonction devra avoir le type suivant :

```
val decode : int * 'a list -> 'a list
```

Analyse Nous allons avoir besoin de diverses fonctions pour cet algorithme :

- Chercher l'occurrence d'un caractère à une position dans une liste
- Trouver la k^{e} occurrence d'un caractère dans une liste
- Reconstruire le mot à partir de ces deux fonctions précédentes

Nombre d'occurrences Il nous faut définir ici une fonction comptant le nombre d'occurrences d'un caractère à une position donnée. On parcourt la liste de façon récursive :

- Si l'élément est celui cherché : on regarde sa position (la position courante est passée en paramètre lors de l'appel). Si la position actuelle est inférieure à celle cherchée, on appelle récursivement la fonction, en incrémentant sa valeur de 1. Sinon, c'est qu'on est sur la position cherchée, on retourne 1.
- Si l'élément n'est pas celui cherché : on regarde la position. Si l'on se trouve avant celle voulue, on continue par un appel récursif. Sinon, on retourne 0.
- Si la liste est vide, on retourne 0.

Elle possède le prototype suivant :

```
val find_oc : 'a -> int -> 'a list -> int -> int
```

Recherche de la k^e occurrence Cette fois, c'est l'opération inverse qui se produit : il faut la k^e occurrence d'un élément donné. Pour simplifier la recherche de la prochaine occurrence, on retournera en fait le couple suivant : (élément de $s2$, position dans la liste).
On effectue le parcourt *simultané* des deux listes élément par élément, bien que l'on ne s'intéresse vraiment qu'à la valeur de l'élément de la première liste (c'est celui sur lequel se base la recherche, et qui provient de la liste reçue).

- Si l'élément est le caractère cherché, on regarde son occurrence. Si elle correspond, on retourne le couple constitué de l'élément de la seconde liste et de sa position. Sinon on appelle récursivement la fonction en incrémentant la position et l'occurrence.
- Si l'élément ne correspond pas : on appelle récursivement la fonction
- Si l'une des listes est vide avant l'autre : il y a eu un problème. On génère un exception.

Elle possède le prototype suivant :

```
val get_couple : 'a -> int -> 'a list -> 'b list -> int -> int -> 'b
* int
```

Reconstruction du mot Il suffit en fait d'alterner les appels aux deux fonctions précédentes. On utilise la fonction `find_oc` pour trouver l'occurrence de l'élément que l'on vient d'ajouter au mot, puis on cherche cet élément avec la fonction `get_couple`, qui nous donne ainsi le prochain caractère et sa position. On itère cette construction jusqu'à ce que l'on ait contruit un mot de même taille que celle passée en paramètre :

```
let rec recompose liste liste_sort car next_pos taille cur_taille =
       let oc = find_oc car next_pos liste_sort 1
       in let (c2,cp) = get_couple car oc liste liste_sort 1 1
       in     if cur_taille = taille
              then [car]
              else car::(recompose liste liste_sort c2 cp taille (
                  cur_taille+1))
;;
```

Listing 1 – Implémentation de la fonction de recomposition

La fonction possède le prototype suivant :

```
val recompose : 'a list -> 'a list -> 'a -> int -> int -> int -> 'a list
```

Il nous faut aussi pouvoir initialiser cet algorithme. En effet, nous ne pouvons connaître que la position du premier élément, qui est transmise, mais pas son occurrence. Il faut donc un fonction qui sera appelée avant la précédente pour pouvoir connaître ce caractère.

Cette fonction parcourt simplement la liste en incrémentant un compteur, et ressort l'élément courant lorsque ce compteur atteint la valeur passée en paramètre.

Elle possède le prototype :

```
val get_car : 'a list -> int -> int -> 'a
```

Fonction principale La fonction principale se contente juste de vérifier les cas particuliers, et de faire un premier appel à la fonction précédente avant d'appeler la fonction `recompose`.

Le seul cas particulier est celui où la liste passée en paramètre est vide. Dans ce cas, c'est que le mot à décoder est vide.

Remarque Cette fonction suppose que les données transmises sont valides, en particulier, elle ne teste pas si la position n'est pas plus grande que la taille de la liste.

2.2 Algorithme de transformation Move To Front

2.2.1 Présentation

Dans cette transformation, on remplace le caractère de la séquence par un indice donné stocké dans un tableau qui sera mis à jour tout au long de l'exécution de l'algorithme.

2.2.2 Encodage

On va transformer une liste de caractère en une liste de « distances »entre les divers caractères : lorsque l'on croise un caractère une première fois, on lui attribue un code arbitraire, et lors du prochain passage, on le codera en lui donnant le nombre d'éléments qui diffèrent entre sa première occurrence et lui.

Intérêt Puisque que la séquence de caractère que l'on doit traiter par cet algorithme est issue de l'algorithme de Burrows-Wheeler, les caractères identiques seront proches. Ainsi, la séquence obtenue par transformation « Move To Front »possédera beaucoup de zéro.

La première occurrence sera codé à l'aide d'une fonction de codage passée en paramètre (par exemple la fonction qui à un caractère associe son code *ASCII* : `Char.code`)

Remarque En fait, le code fourni par la fonction pour cet élément (par exemple, son code ASCII) ne sera pas stocké dans la table. On ne stockera que sa position par rapport à sa dernière occurrence.

On demande le prototype suivant :
`val encode : ('a -> int)-> 'a list -> int list`

Analyse de l'algorithme Nous aurons besoin de la fonction auxiliaire qui effectuera le parcours de la liste de codes obtenus.

Codage d'un élément L'idée est de remplacer dans l'appel à la fonction `encode` la fonction de codage donnée en paramètre par une fonction qui donnera le code de l'élément dans la table de codage.

Voici l'implémentation *CaML* de cette fonction :

```
let move_front_encode fct car_code =
       fun x -> let code = fct x
              in     if code = car_code
                     then 0
                     else   if code < car_code
                            then code+1
                     else code
;;
```

Listing 2 – Implémentation de la fonction de codage

Cette fonction prend en paramètre la fonction de codage, le code du dernier élément, ainsi que l'élément courant.

Elle lui associe son code grâce à la fonction de codage, et effectue divers test sur cette valeur :

- ▷ Si son code est le même que celui du dernier caractère, alors c'est qu'ils sont identiques : on retourne 0
- ▷ Sinon, si son code est inférieur à celui de dernier caractère, alors on décale de 1 celui ci : il se situe avant lui dans la table de codage.
- ▷ Sinon, c'est que c'est la première fois qu'on voit ce caractère.

Cette fonction devant remplacer la fonction de codage à certains appel, elle possède le prototype suivant :
`val move_front_encode : ('a -> int)-> int -> 'a -> int`

Fonction principale La fonction de codage aura l'implémentation suivante :

```
let rec encode fonction liste_char =
       match liste_char with
       |[] -> []
       |t::q -> let code = fonction t
                     in     if code = 0
                            then 0::(encode fonction q)
```

```
                              else code::(encode (move_front_encode
                                 fonction code) q)
;;
```

Listing 3 – Fonction de codage

Lors du premier appel, c'est la fonction choisie par l'utilisateur qui est utilisée pour coder l'élément (dans nos tests, ce sera la fonction `Char.code`).

– Si le code de l'élément est différent de 0, c'est-à-dire que l'élément c'est pas le même que le dernier codé, alors on insère son code dans la table, et on fait un appel récursif à la fonction de codage définie au-dessus. On décalera ainsi le code de tous les éléments suivants.
– Si le code est 0, alors l'élément est le même que le précédent. On insère donc 0 dans la table, et l'on fait appel à la *même* fonction pour coder le reste de la liste, puisqu'il n'y a pas besoin de décalage dans la table de codage.

2.2.3 Décodage

Le décodage est quasiment identique : il suffit de parcourir la liste, et de retourner le caractère associé au code, et de mettre à jour la table.

Décodage d'un élément On va utiliser une fonction de décodage du code d'un élément similaire à celle utilisée pour le codage.

Cette fonction possède l'implémentation suivante :

```

let move_front_decode fct_decodage code =
  fun n-> if n = 0
    then (fct_decodage code)
    else  if n <= code then fct_decodage (n-1)
      else fct_decodage n
;;
```

Listing 4 – Implémentation de la fonction de décodage

- Si le code de l'élément est 0, alors c'est que l'élément est le même que celui d'avant : on le décode avec la fonction de décodage passée en paramètre
- Sinon si son code est inférieur à celui ci, on décale son code de 1 dans la table de codage, puisqu'il va remonter en haut de la table.
- Sinon on ressort son code

Cette fonction va faire office de fonction de décodage. Elle a donc le prototype suivant :

```
val move_front_decode : (int -> 'a)-> int -> int -> 'a
```

Fonction principale La fonction principale est complètement similaire à la fonction d'encodage, il n'y a aucune difficulté sur ce point.

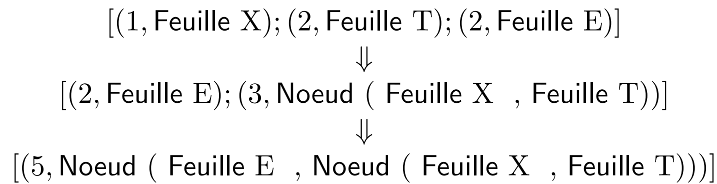
2.3 Codage de Huffman

Le codage de Huffman est en fait la véritable étape de compression. Parfois qualifiée de compressions « statistique », elle code les éléments de façon à ce que les plus fréquents aient un code binaire le plus petit possible.

2.3.1 Principe

On recherche le nombre d'occurrence de chaque caractère dans le texte. Ensuite, on va construire un arbre de la façon suivante : on associe les noeuds de poids (ici, d'occurrence) les plus faibles pour former un noeud dont le poids sera égal à la somme des poids précédents. On itère ce procédé jusqu'à ce qu'il n'y ait plus qu'un seul noeud : c'est l'arbre de Huffman.

Exemple Soit le texte TEXTE à coder. On va tout d'abord créer la liste de couples (occurrences, éléments) suivante : $[(1, X); (2, T); (2, E)]$. On crée ensuite l'arbre, comme expliqué dans le schéma 1.



$$[(1, \mathsf{Feuille}\ \mathrm{X}); (2, \mathsf{Feuille}\ \mathrm{T}); (2, \mathsf{Feuille}\ \mathrm{E})]$$
$$\Downarrow$$
$$[(2, \mathsf{Feuille}\ \mathrm{E}); (3, \mathsf{Noeud}\ (\ \mathsf{Feuille}\ \mathrm{X}\ ,\ \mathsf{Feuille}\ \mathrm{T}))]$$
$$\Downarrow$$
$$[(5, \mathsf{Noeud}\ (\ \mathsf{Feuille}\ \mathrm{E}\ ,\ \mathsf{Noeud}\ (\ \mathsf{Feuille}\ \mathrm{X}\ ,\ \mathsf{Feuille}\ \mathrm{T})))]$$

Tab. 1 – Création de l'arbre

On obtient alors l'arbre de Huffman représenté en figure 1

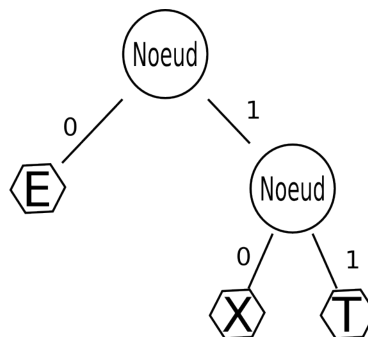


Fig. 1 – Arbre de Huffman

Chaque lettre est codée par son chemin dans l'arbre : un bit 0 codera pour une branche gauche, et un bit 1 pour une branche droite.

De plus, aucun code ne peut être préfixe d'un autre, c'est-à-dire que seuls les chemins allant jusqu'aux feuilles correspondent effectivement le code d'un élément de l'arbre.

Le code pour le mot TEXTE est le suivant : 11 0 10 11 0, ce qui se transmet en 11010110.

2.3.2 Représentation d'un arbre de Huffman en *CaML*

J'ai choisi de définir le type « Arbre de Huffman »de la façon suivante en *CaML* :

```
type 'a huffmantree =
  |Leaf of 'a
  |Node of 'a huffmantree * 'a huffmantree
;;
```

Listing 5 – Type Arbre de Huffman

2.3.3 Construction de l'arbre

La première étape est de construire l'arbre de Huffman associé au texte. Pour cela, il faut :

- Calculer les occurrences de chacun de éléments dans le texte à coder
- Utiliser l'algorithme de construction de l'arbre décrit précédemment

La fonction devra avoir le type suivant :

```
val build_tree : 'a list -> 'a huffmantree
```

Calcul des occurrences Cette fonction prend en paramètre une liste d'élément, et retourne la liste formée des couples (occurrence,élément). Pour cela, elle fait appel à une fonction auxiliaire `incremente`.

La fonction `incremente` augmente l'occurrence d'un élément dans une liste (occurrence, élément).

Elle est du type :

```
val incremente : 'a -> (int * 'a)list -> (int * 'a)list
```

La fonction de calcul des occurrences possède la signature suivante :

```
val occurrence : 'a list -> (int * 'a)list -> (int * 'a)list
```

Construction de l'arbre D'après l'algorithme, il nous faudra insérer au fur et à mesure les nouveaux couples (poids, arbres) dans une liste. De plus, cette liste devra être triée par ordre croissant d'occurrence, pour augmenter l'efficacité de l'algorithme.

Étudions en premier lieu l'implémentation de la fonction nécessaire à l'insertion d'élément dans une liste triée, suivant un ordre passé en paramètre :

- Si la liste est vide, on retourne l'élément
- Sinon, on compare l'ordre de l'élément courant et de celui que l'on cherche à insérer
  - Si ils sont dans le bon ordre, on ajoute l'élément courant à la liste obtenue par appel récursif sur la queue.
  - Sinon, on ajoute l'élément en première position.

Cette fonction possède le prototype suivant :

```
val insert : ('a -> 'a -> bool)-> 'a -> 'a list -> 'a list
```

Ensuite, il nous faut implémenter la fonction qui réalisera la fusion des arbres : cette fonction prend en paramètre une liste de couple (poids, arbre), et parcourt chacun de ces éléments :

- Si la liste ne contient qu'un seul couple, c'est que l'arbre est complet : on le retourne
- Sinon, c'est qu'elle contient deux éléments $(oc1, arbre1)$ et $(oc2, arbre2)$ on appelle récursivement la fonction sur la nouvelle liste, obtenue par insertion (à l'aide de la fonction `insert`) du couple suivant : $(oc1 + oc2,$ Noeud (arbre1 , arbre2)) dans le reste de la liste.

Cette fonction est de type :

```
val merge_tree : (int * 'a huffmantree)list -> 'a huffmantree
```

Enfin, il reste à créer la fonction qui à partir d'une liste (occurrence,caractère), générera l'arbre de Huffman. Cette fonction applique simplement la fonction `merge_tree` précédente à la liste obtenue par la fonction : `List.map (fun (n,car)-> (n, Leaf car))` sur la liste passée en paramètre. Ainsi, on obtient bien une liste (poids,arbre), qui correspond au type de l'argument attendu par la fonction de fusion des arbres.

Cette fonction est de type :

```
val make_tree : (int * 'a)list -> 'a huffmantree
```

Fonction principale La fonction principale vérifier que la liste n'est pas vide. Si ce n'est pas le cas, alors elle génère la liste des occurrences, la tri à l'aide d'un tri par fusion, et ensuite fait appel à la fonction `make_tree`.

2.3.4 Compression d'un texte

Il faut maintenant coder les caractère du texte d'origine par leur code obtenu en parcourant l'arbre de Huffman obtenu.

Pour cela, nous allons générer un « alphabet » : c'est une table dans laquelle on notera pour chaque caractère de la séquence originale son code dans l'arbre de Huffman. Étant donné que l'arbre de Huffman est construit à partir du texte initial, l'ensemble des caractères qui le constitue est utilisé au moins une fois, donc cet alphabet n'est pas gaspilleur de temps.

Ensuite, il s'agira de trouver chacun des caractères dans cet alphabet, et d'assembler les codes obtenus.

Il faudra implémenter les fonctions suivantes :
- Génération de l'alphabet
- Recherche d'un caractère
- Encodage d'un« mot »

Génération de l'alphabet Pour générer l'alphabet, nous allons parcourir l'arbre.

```
let rec create_alphabet h_tree =
       match h_tree with
       |Leaf car -> [(car,[])]
       |Node (g,d) -> (List.map (fun (e,c) -> e,(false::c)) (
           create_alphabet g))
5              @(List.map (fun (e,c) -> e,(true::c)) (create_alphabet
                   d))
;;
```

Listing 6 – Génération de l'alphabet

La fonction parcourt l'arbre en associant `false` (donc 0) lors du passage dans une branche gauche, et `true` (donc 1) dans le passage dans une branche droite.

Remarque Un caractère seul (donc une feuille) n'a pas de code associé.

Cette fonction possède la signature suivante :
`val create_alphabet : 'a huffmantree -> ('a * bool list)list`

Recherche d'un caractère On parcourt récursivement l'alphabet obtenu en cherchant le caractère passé en paramètre Lorsque il est trouvé, on renvoit son code associé.

Cette fonction est de prototype suivant :
`val find_car : 'a -> ('a * 'b)list -> 'b`

Encodage On va en fait appliquer la fonction de recherche de code d'un caractère en concaténant le code pour le caractère (obtenu par appel à la fonction `find_car`), à la liste issu de l'appel récursif sur le reste de la liste.

Cette fonction est de type :
```
val find_bin_word : 'a list -> ('a * 'b list)list -> 'b list
```

Fonction principale Une remarque s'impose : que faire si l'on reçoit un texte composé d'une suite d'un seul caractère (exemple : AAAAAAAAA). En effet, j'ai décidé de renvoyer le code vide en cas d'unique caractère dans l'arbre.

En fait, ce cas ne se produira pas en pratique, puisque le texte que l'on reçoit est issu de la transformation par « Move To Front ».

Donc, si le texte est composé d'au moins deux fois le même caractère, le texte en entrée de la compression d'Huffman sera composé d'au moins deux caractères : le code arbitraire choisi pour ce caractère, et 0 !

Ce cas n'est donc pas gênant.

2.3.5 Décodage

Il nous faut cette fois réaliser l'opération inverse. À partir d'une liste de bits, il faut recomposer par un parcours dans l'arbre le mot initial.

On demande une fonction de prototype :
```
val decode : 'a huffmantree * bool list -> 'a list
```

Pour cela, nous n'aurons besoin que d'une unique fonction intermédiaire qui parcourt l'arbre selon une liste de bits passé en paramètre et ressort le couple (caractère, liste restante).

Cette fonction s'implémante en *CaML* de la façon suivante :

```
let rec read_letter h_tree liste =
       match h_tree,liste with
       |Leaf car,_ -> (car, liste)
       |Node _, [] -> failwith "Erreur dans le codage"
       |Node (g,d), t::q -> if t
                            then read_letter d q
                            else read_letter g q
;;
```

Listing 7 – Parcourt de l'abre

La seul erreur possible est que la liste soit vide alors que l'on est sur un noeud.

Elle possède le prototype suivant :
```
val read_letter : 'a huffmantree -> bool list -> 'a * bool list
```

Fonction principale Elle passe à la fonction `read_letter` la liste de booléen et l'arbre de Huffman associé, et en récupère le caractère lu, ainsi que la nouvelle liste de booléen, qui sera son nouveau paramètre lors de son appel récursif.

3 Tests

Pour les tests, j'ai adopté la procédure suivante :

* Création du fichier contenant le texte à compresser
* Appel du programme `huffman`, avec l'option `-e` pour compresser ce fichier
* Relevé des données affichées par le programme
* Appel au programme `huffman` avec l'option `-d` pour décompresser le fichier
* Vérification de l'intégrité du fichier décodé, par la commande `diff`

Remarque Parmi les données fournies par le programme, se trouve le taux de compression. Or pour de petits fichiers, ce taux est supérieur à 100 %. Cela s'explique par le fait que le fichier compressé contient aussi les données nécessaires au décodage, notamment l'arbre de Huffman.

Ce taux n'est donc significatif que lorsque sa valeur descend au dessous de 100%.

Fichier vide

J'ai tenté de compresser un fichier vide, obtenu avec la commande `touch` par exemple.

Il en a résulté une exception, prévue dans le module `huffman.ml`.

Fichier contenant un caractère

Ce test est utile pour connaître le comportement du module `huffman.ml` Pas de problème particulier.

Répétition d'un unique caractère

Testons les opérations de compression/décompression avec un fichier contenant un unique caractère répété plusieurs fois. En effet, j'ai volontairement écarté le cas d'un unique caractère à coder dans le module `huffman.ml`, puisque son entrée provenait de la sortie du module `movetofront.ml`, et donc posséderait dans ce cas ci un unique caractère.

Même ligne

Un premier fichier contiendra le caractère répété plusieurs fois sur la même ligne (une centaine de fois)

J'obtiens un taux de compression de 67%, et le fichier original et celui décompressé sont identiques.

Même colonne

En fait, dans ce cas là, il n'y a pas un unique caractère à compresser. En effet, il y a aussi le caractère « saut de ligne »que l'on doit coder : une colonne de 10 caractères contient en fait 20 caractères. Ce cas ne devrait théoriquement pas poser de problème.

Or, il s'avère que le fichier décodé ne contient qu'une seule fois le caractère ! Après plusieurs tests, il s'avère que si l'on répète en colonne le même motif, on obtiendra en décompressant une seule fois ce motif. Il suffit d'insérer une variation dans une des lignes pour retourner sur deux fichiers identiques.

Causes possibles Le fait frappant est que la répétition en ligne ne produit aucune erreur. Or une colonne n'est rien d'autre qu'une ligne dans laquelle on trouve le caractère
`n` de retour à la ligne. Logiquement, il ne devrait donc pas y avoir d'erreur.

Remarque Ce problème est maintenant résolu. Il était du à un mauvais codage de la fonction de décodage par algorithme Burrows Wheeler. En effet, cette dernière ne décodait pas les motifs répétés. Je n'ai cependant pas bien compris pourquoi une ligne de motifs répétés ne posait pas de problème particuliers à l'exécution de cet algorithme.

Motif

On peut aussi étendre ces tests à des fichiers contenant un même motif répeté plusieurs fois, ces cas étendant ceux réalisés lors de la répétition d'un même caractère.

Il n'y a pas eu de problème lors de la réalisation de ces tests, malgré une remarque similaire concernant les motifs « en colonne », problèmes réglés après mise au point du code.

Fichier texte

J'ai pris plusieurs paragraphes du fameux texte *Lorem ipsum*, disponibles sur cette page : `http://www.lipsum.com/`.

Voici quelques ordre de grandeurs (ceux concernant le fichier initial sont obtenus par la commande `wc`) :

1 paragraphe

Nombre de caractères : 682

Nombre de mots : 98

Taux de compression : 81 %

Temps moyen de compression : 0.6 sec

Temps moyen de décompression : 0.3 sec

Fichiers identiques : Oui

2 paragraphes

Nombre de caractères : 1152

Nombre de mots : 151

Taux de compression : 64

Temps moyen de compression : 1.9 sec

Temps moyen de décompression : 0.9 sec

Fichiers identiques : Oui

5 paragraphes

Nombre de caractères : 3048

Nombre de mots : 452

Taux de compression : 44 %

Temps moyen de compression : 15.9 sec

Temps moyen de décompression : 5.8 sec

Fichiers identiques : Oui

10 paragraphes

Nombre de caractères : 8120

Nombre de mots : 1210

Taux de compression : 33 %

Temps moyen de compression : 590 sec

Temps moyen de décompression : 41 sec

Fichiers identiques : Oui

Conclusion des tests

La majorité du temps nécessaire à la compression est utilisé par l'algorithme de Burrows-Wheeler. Par exemple, voici le détail pour la compression et la décompression de 10 paragraphes :

◇ Compression :

> **Burrows-Wheeler :** 570.13 sec
>
> **Move To Front :** 19.901 sec
>
> **Huffman :** 0.54433 sec

◇ Décompression :

Huffman : 0.03561 sec

Move To Front : 3.9206 sec

Burrows-Wheeler : 37.692 sec

Chose notable à remarquer, l'algorithme de Burrows Wheeler est considérablement plus rapide lors du décodage que lors du codage. On peut expliquer cela par la quantité de donnée engendrée lors du codage, pour la création de la séquence des rotations, tandis que l'algorithme ne manipule « que »deux séquences pour la décompression.

4 Conclusion

Ce sujet m'est apparu comme très intéressant. En effet, il s'agit d'une application concrète de nos connaissances théoriques, sur le domaine qu'est la compression de données. Même si l'algorithme développé ici reste basique, il n'en constitue pas moins un premier pas vers des projets d'envergures plus importantes.

Concernant mon algorithme, je regrette de ne pas trouver d'où peut venir le problème énoncé lors de la compression « en colonne ».

Il faut aussi noter que cet algorithme est très gourmand en resources, aussi bien processeur (calcul) que mémoire (stockage de valeurs). Ne trouvant pas de solutions permettant de conserver un code « convenablement lisible », point essentiel dans la réalisation d'un projet, j'ai du me résoudre à laisser mes algorithmes initiaux.

Vis à vis du choix du langage, le fait de devoir adopter un style de programmation fonctionnel impose la lourdeur des deux algorithmes de Burrows-Wheeler et de Move To Front. Le choix d'un langage possédant des références (tels que *C*, par exemple), aurait considérablement alléger le temps d'exécution de ces deux algorithmes. Cependant, le choix de programmer de façon fonctionnelle, et notamment en utilisant la récursivité est tout à fait adapté à l'algorithme de Huffman.

5 Listings

```
(*************************************************************************)
(*************************************************************************)
(*******                 Transformée de Burrows-Wheeler           *******)
(*************************************************************************)
(*************************************************************************)



(*************************************************************************)
(*******                 Fonctions d'encodage                     *******)
(*************************************************************************)



(* Fonction create_seq
 * Argument :
        * liste : 'a list
        * taille : int
        * cur : int : nombre de rotations effectuées
 * Résultat : 'a list
 * Sémantique : Crée la séquence des rotations d'une liste de taille taille
 * Exceptions : Aucune
 *)
 (* Test :
create_seq ['t';'e';'x';'t';'e'] 5 0;;
- : char list list =
[['e'; 'x'; 't'; 'e'; 't']; ['x'; 't'; 'e'; 't'; 'e'];
 ['t'; 'e'; 't'; 'e'; 'x']; ['e'; 't'; 'e'; 'x'; 't'];
  ['t'; 'e'; 'x'; 't'; 'e']]
*)
let rec create_seq liste taille cur =
        match liste with
        |[]-> [[]]
        |t::q ->if cur = taille
                then []
                else let new_list = q@[t]
                in new_list::(create_seq new_list taille (cur+1))
;;

(******                Implémantation du tri par fusion             ******)

(* Fonction cut
 * Argument :
        * liste : 'a list
 * Résultat : 'a list
 * Sémantique : Coupe une liste de taile n en deux sous-liste de taille
                proche de n/2
 * Exception : Aucune
*)
(* Tests :
cut [1;2;3;4;5;6;7;8;9;10];;
- : int list * int list = ([1; 3; 5; 7; 9], [2; 4; 6; 8; 10])
*)

let rec cut liste =
        match liste with
        |[]-> ([],[])
        |[t]-> ([t],[])
        |t1::t2::q -> let l1,l2 = cut q
                        in (t1::l1),(t2::l2)
;;
```



```
(* Fonction merge
 * Argument :
        * Ordre : 'a -> 'a -> bool
        * l1 : 'a list
        * l2 : 'a list
 * Résultat : 'a list
 * Sémantique : Fusionne deux listes triées en une liste triée
 * Exception : Aucune
*)
(* Tests :
merge (<) [1; 3; 5; 7; 9] [2; 4; 6; 8; 10];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
*)

let rec merge ordre l1 l2 =
        match (l1,l2) with
        |[],_ -> l2
        |_,[] -> l1
        |(t1::q1),(t2::q2) ->   if ordre t1 t2
                                then t1::(merge ordre q1 l2)
                                else t2::(merge ordre q2 l1)
;;



(* Fonction sort
 * Argument :
        * ordre : 'a -> 'a -> bool
        * liste : 'a list
 * Résultat : 'a list
 * Sémantique : Trie la liste suivant l'ordre passé en paramètre
 * Exception : Aucune
*)
(* Test :
sort (<) [9;2;7;10;3];;
- : int list = [2; 3; 7; 9; 10]
*)

let rec sort ordre liste =
        match liste with
        |[]->liste
        |[_]->liste
        |_->    let l1,l2 = cut liste
                in merge ordre (sort ordre l1) (sort ordre l2)
;;

(*************************************************************************)

(* Fonction find_pos
 * Argument :
        * seq : 'a list
        * l_seq : 'a list list
 * Résultat : int
 * Sémantique : Trouve la position de liste seq dans la liste de listes l_ seq
 * Exception : Si le motif n'est pas trouvé
*)
(* Tests :
find_pos [4;1;3] [[1; 3; 4]; [4; 1; 3]; [3; 4; 1]];;
- : int = 2
*)
```

```
let rec find_pos seq l_seq =
        match l_seq with
        |[]-> failwith "Motif non trouvé"
        |t::q ->if t = seq
                then 1
                else 1 + (find_pos seq q)
;;



(* Fonction last_char
 * Argument :
        * l_seq : 'a list list
        * acc : 'a list : accumulateur
 * Résultat : 'a list
 * Sémantique : Récupère le dernier élément de chaque liste d'une liste
                de listes
 * Exception : Aucune
*)
(* Test :
last_char [[1; 3; 4]; [4; 1; 3]; [3; 4; 1]] [];;
- : int list = [4; 3; 1]
*)
(*
let rec last_char l_seq acc =
        match l_seq with
        |[]->acc
        |t::q->last_char q (acc@[(List.hd (List.rev t))])
;;
*)


let rec last_char l_seq =
        match l_seq with
        |[]->[]
        |t::q->(List.hd (List.rev t))::(last_char q)
;;


(*******                       Fonction principale                      *******)

(* Fonction encode
 * Argument :
        * liste : 'a list
 * Résultat : int * 'a list
 * Sémantique : Renvoie l'indice de position
                et la séquence associée
 * Exception : Aucune
 *)
 (* Test :
 encode ['t';'e';'x';'t';'e'];;
 - : int * char list = (4, ['t'; 't'; 'x'; 'e'; 'e'])
 *)

let encode liste =
        match liste with
        |[] -> (0,[])
        |_ ->   let l_seq = sort (<) (create_seq liste (List.length liste) 0)
                        in (find_pos liste l_seq , last_char l_seq)
;;
```



```

(*****************************************************************************)
(***                        Fonctions de décodage                          ***)
(*****************************************************************************)


(* Fonction find_oc
 * Argument :
        * car : 'a  : élément
        * pos : int : position
        * liste : 'a list : liste d'éléments
        * cur_pos : int : position courante
 * Résultat : int
 * Sémantique : Trouve l'occurence d'un élément dans une liste
                        à une position donnée
 * Exception : Aucune
 *)
 (* Test :
# find_oc 'e' 4 ['t'; 't'; 'x'; 'e'; 'e'] 1;;
- : int = 1
# find_oc 'e' 5 ['t'; 't'; 'x'; 'e'; 'e'] 1;;
- : int = 2
*)
let rec find_oc car pos liste cur_pos =
        match liste with
        |[] -> 0
        |t::q -> if t = car
                        (* Si l'élément est celui cherché *)
                        then    if cur_pos < pos
                                (* On est avant la position cherchée *)
                                then 1 + (find_oc car pos q (cur_pos+1))
                                (* Sinon, on a finit *)
                                else 1
                        (* Sinon *)
                        else    if cur_pos < pos
                                (* On est avant la position cherchée *)
                                then find_oc car pos q (cur_pos+1)
                                (* Sinon, c'est plus la peine de chercher
                                                plus loin, on arrete *)
                                else 0
;;

(* Fonction get_couple
 * Argument :
        * car : 'a  : élément
        * occur : int : occurrence
        * liste : 'a list : liste d'éléments
        * list_sort : 'a list : liste d'éléments (triée en pratique)
        * cur_occur : int : occurrence courante
        * cur_pos : int : position courante
 * Résultat : 'a * 'a * int
 * Sémantique : Ressort le couple contenant l'élément de la liste liste_sort
        d'occurrence occur et sa position
 * Exception : Si les deux listes ne sont pas de la même taille
 *)
 (* Tests :
get_couple 3 2 [1;3;6;7;1;4;3] [3;4;6;2;1;4;6] 1 1;;
- : int * int * int = (6, 7)
*)

let rec get_couple car occur liste liste_sort cur_occur cur_pos =
```

```
        match liste, liste_sort with
        |[],_ -> failwith "Les listes ne sont pas compatibles"
        |_,[] -> failwith "Les listes ne sont pas compatibles"
        |t1::q1,t::q2 -> if t1 = car
                        then    if cur_occur = occur
                                then (t,cur_pos)
                                else get_couple car occur q1 q2 (cur_occur+1)
                                        (cur_pos+1)
                         else   get_couple car occur q1 q2 cur_occur (cur_pos+1)
;;



(* Fonction recompose
 * Argument :
        * liste : 'a list : liste d'éléments
        * list_sort : 'a list : liste d'éléments triée
        * car : 'a  : élément
        * next_pos : int : position pour le prochain appel récursif
        * taille : int : taille de la séquence d'origine
        * cur_taille : int : Taille du mot actuellement formé
 * Résultat : 'a * int
 * Sémantique : Fonction qui applique le décodage de Burrows-Wheeler.
        Il faut lui passer en paramètre les deux listes (celle reçue,
        et celle obtenue par tri), ainsi que le premier caractère du mot
        et sa position.
 * Exception : Aucune
 *)
 (* Tests :
recompose ['t'; 't'; 'x'; 'e'; 'e'] ['e'; 'e'; 't'; 't'; 'x'] 't' 4 5 1;;
- : char list = ['t'; 'e'; 'x'; 't'; 'e']
*)

let rec recompose liste liste_sort car next_pos taille cur_taille =
        let oc = find_oc car next_pos liste_sort 1
        in let (c2,cp) = get_couple car oc liste liste_sort 1 1
        in      if cur_taille = taille
                then [car]
                else car::(recompose liste liste_sort c2 cp taille (cur_taille+1
))
;;

(* Fonction get_car
 * Argument :
        * liste : 'a list : liste d'éléments
        * pos : int : position
        * cur_pos : int : position courrante
 * Résultat : 'a
 * Sémantique : Récupère l'élément de la liste à la position pos
 * Exception : Si la liste est vide
*)
(* Test :
# get_car ['t'; 'e'; 'x'; 't'; 'e'] 3 1;;
- : char = 'x'
*)

let rec get_car liste pos cur_pos =
        match liste with
        |[]->failwith "Pas de caractère"
        |t::q -> if cur_pos = pos
                then t
                else get_car q pos (cur_pos+1)
```



```
;;


(*******                        Fonction principale                    *******)


(* Fonction decode
 * Argument :
        * int*'a list
 * Résultat : 'a list
 * Sémantique : Applique le décodage de Burrows-Wheeler au doublet pos,liste.
 On suppose que les données fournies sont valides.
 * Exception : Aucune
*)
(* Test :
decode (4, ['t'; 't'; 'x'; 'e'; 'e']);;
- : char list = ['t'; 'e'; 'x'; 't'; 'e']
*)
let decode (pos,liste) =
        match liste with
        |[]->[]
        |_-> let liste_sort = sort (<) liste
                in let (car) = get_car liste_sort pos 1
                in recompose liste liste_sort car pos (List.length liste) 1
;;
```

```
(***************************************************************************)
(***************************************************************************)
(*******                    Encode Move To Front                    *******)
(***************************************************************************)
(***************************************************************************)



(***************************************************************************)
(*******                    Fonctions d'encodage                    *******)
(***************************************************************************)

(* Fonction move_front_encode
 * Argument :
        * fct_encode : ('a -> int) : fonction de codage d'un caractère
        * car_code : int : code du caractère précédent
 * Résultat : 'a -> int : Nouvelle fonction de codage
 * Sémantique : On compare le code du nouveau caractère à celui de l'ancien.
 *              Retourne 0 si le code est le même, sinon on appelera
 *              récursivement la fonction sur le reste de la liste.
 * Exceptions : Aucune
 *)
(* Tests :
# move_front_encode Char.code (Char.code 'A') 'A';;
- : int = 0
# move_front_encode (move_front_encode Char.code (Char.code 'A') (Char.code 'B')
 'A';;
- : int = 1
*)


let move_front_encode fct car_code =
        (* On définit une nouvelle fonction de codage *)
        fun x -> let code = fct x
                in      if code = car_code (* Le caractère est le même que le
                précédent *)
                        then 0 (* On le code donc par un 0 *)
                        else    if code < car_code (* Le caractère a déjà été vu
                        : on décale le code de ceux qui le précède *)
                                then code+1
                        else code (* Sinon, on ressort son code *)
;;



(* Fonction encode
 * Argument :
        * fonction : 'a -> int : fonction de codage
        * liste_char : 'a list : liste d'élément à coder
 * Résultat : int list : liste codée
 * Sémantique : Code la liste d'élément en utilisant l'algorithme MTF
 * Exceptions : Aucune
 *)
 (* Tests :
# encode (Char.code) ['a';'e';'e';'e';'e';'a'];;
- : int list = [97; 101; 0; 0; 0; 1]
*)

let rec encode fonction liste_char =
        match liste_char with
        |[] -> []
        |t::q -> let code = fonction t
```



```
                        in      if code = 0 (* Si le caractère est le même *)
                                then 0::(encode fonction q)
                                (* Sinon, on appel récursivement cette fonction
                                avec la nouvelle fonction de codage *)
                                else code::(encode (move_front_encode fonction c
ode) q)
;;


(***************************************************************************)
(*******                    Fonctions de décodage                    *******)
(***************************************************************************)


(* Fonction move_front_decode
 * Argument :
        * fct_decodage : (int ->'a) : fonction de décodage d'un caractère
        * code : int : code du caractère précédent
 * Résultat : int -> 'a : Nouvelle fonction de décodage
 * Sémantique : Si le code est 0, on décode le caractère.
 *              Sinon, on compare le code par rapport à celui de l'élément
 *              précédent
 * Exceptions : Aucune
 *)
(* Tests :
# move_front_decode (Char.chr) 65 0;;
- : char = 'A'
# move_front_decode (move_front_decode Char.chr 65) 66 1;;
- : char = 'A'
*)

let move_front_decode fct_decodage code =
        fun n-> if n = 0 (* Le caractère est le même que le précédent *)
                then (fct_decodage code)
                else    if n <= code (* Si on l'a déjà vu *)
                        then fct_decodage (n-1) (* On décrémente le code de ceux
                        qui le précède *)
                        else fct_decodage n (* Sinon, on ressort son code *)
;;



(* Fonction decode
 * Argument :
        * fonction : int -> 'a : fonction de décodage
        * liste_int : int list : liste d'entier servant de codage
 * Résultat : 'a list
 * Sémantique : Réciproque de la fonction encode : applique le décodage
        suivant l'algorithme de MTF.
 * Exceptions : Aucune
 *)
 (* Tests :
# decode (Char.chr) [97; 101; 0; 0; 0; 1];;
- : char list = ['a'; 'e'; 'e'; 'e'; 'e'; 'a']
*)

let rec decode fonction liste_int =
        match liste_int with
        |[] -> []
        |t::q -> let car = fonction t
                in      if t = 0 (* Même caractère *)
                        then car::(decode fonction q)
```

```
                        (* On fait appel à la nouvelle fonction de codage *)
                        else car::(decode (move_front_decode fonction t) q)
;;
```

```
(**************************************************************************)
(**************************************************************************)
(******                        Code Huffman                         ******)
(**************************************************************************)
(**************************************************************************)


(**************************************************************************)
(******                    Type arbre de Huffman                    ******)
(**************************************************************************)
type 'a huffmantree =
        (* On définit une feuille *)
        |Leaf of 'a
        (* On définit le type récursif *)
        |Node of 'a huffmantree * 'a huffmantree
;;


(**************************************************************************)
(******                Fonction de création de l'arbre              ******)
(**************************************************************************)

(* Fonction insert
 * Argument :
        * ordre : 'a -> 'a -> bool : ordre sur les éléments
        * liste : 'a  list : liste d'élément
 * Résultat : 'a list
 * Sémantique : Ajoute l'élément x dans la liste triée en gardant l'ordre
 * Exceptions : Aucune
 *)
(* Test :
# insert (<) 5 [2;4;6];;
- : int list = [2; 4; 5; 6]
# insert (<) 5 [2;4];;
- : int list = [2; 4; 5]
*)

let rec insert ordre x liste  =
        match liste with
        |[] -> [x] (* Si la liste est vide, on insert x *)
        |t::q -> if ordre t x (* Si t et x sont dans le bon ordre *)
                (* On insère x dans le reste de la liste *)
                 then t::(insert ordre x q)
                 (* Sinon on le place en tête de liste *)
                 else x::liste
;;



(* Fonction merge_tree
 * Argument :
        * liste : (int * 'a huffmantree)  list : liste d'éléments
                (pd, arbre associé)
 * Résultat : 'a huffmantree
 * Sémantique : Construit l'arbre de Huffman en partant des feuilles
 * Exceptions : Si la liste passée en paramètre est vide
 *)
(* Test :
# merge_tree [(2, Leaf 1);(3,Leaf 0); (5,Leaf 8)];;
- : int huffmantree = Node (Leaf 8, Node (Leaf 1, Leaf 0))
*)
```



```
let rec merge_tree liste =
        match liste with
        |[] -> failwith "Aucun élément à coder"
        |[p,tree] -> tree (* Si la liste ne contient plus qu'un arbre *)
        (* Sinon, on additionne les poids, et on crée le nouvel arbre *)
        |(p1,tree1)::(p2,tree2)::q -> merge_tree (insert (<)
                ((p1+p2),Node(tree2,tree1)) q)
;;




(* Fonction make_tree
 * Argument :
        * liste : (int * 'a)  list : liste d'éléments
                (occurrence, caractère)
 * Résultat : 'a huffmantree
 * Sémantique : Construit l'arbre de Huffman à l'aide d'une liste de
                couples (occurrence,caractère)
 * Exceptions : Si la liste passée en paramètre est vide
 *)
(* Test :
# make_tree [(2,2);(2,(-1));(3,9);(7,0)];;
- : int huffmantree = Node (Leaf 0, Node (Leaf 9, Node (Leaf 2, Leaf (-1))))
*)

let make_tree liste =
        match liste with
        |[] -> failwith "La liste est vide"
        (* Si la liste n'est pas vide, on applique la fonction merge_tree à la
        liste obtenue en utilisant les caractères pour créer une feuille *)
        |_ -> merge_tree (List.map (fun (n,car) -> (n, Leaf car)) liste)
;;


(* Fonction incremente
 * Argument :
        * x : 'a : élément
        * liste : (int * 'a)  list : liste d'éléments
                (occurrence, caractère)
 * Résultat : (int * 'a) list
 * Sémantique : Incrémente l'occurrence de x dans la liste
 * Exceptions : Aucune
 *)
(* Test :
# incremente 2 [(1,1);(1,8);(2,9);(2,3);(6,2)];;
- : (int * int) list = [(1, 1); (1, 8); (2, 9); (2, 3); (7, 2)]
*)

let rec incremente x liste =
        match liste with
        |[] -> [(1,x)]
        |(oc,e)::q -> if e = x
                (* Si l'élément est celui que l'on cherche *)
                then (oc+1,x)::q
                (* Sinon, on parcourt le reste de la liste *)
                else (oc,e)::(incremente x q)
;;
```

```
(* Fonction occurrence
 * Argument :
        * liste : 'a  list : liste d'éléments
        * acc : (int * 'a) list : accumulateur contenant la liste des couples
        * (occurrence, élément)
 * Résultat : (int * 'a) list
 * Sémantique : Construit à partir d'une liste d'élément la liste des couples
 *              (occurrences, éléments)
 * Exceptions : Aucune
 *)
(* Test :
# occurrence [2;3;4;5;2;5;6;5;0] [] ;;
- : (int * int) list = [(2, 2); (1, 3); (1, 4); (3, 5); (1, 6); (1, 0)]
*)

let rec occurrence liste acc =
        match liste with
        |[] -> acc
        (* On augmente l'occurrence de t dans l'accumulateur *)
        |t::q -> occurrence q (incremente t acc)
;;


(******                 Implémantation du tri par fusion                ******)

(* Fonction cut
 * Argument :
        * liste : 'a list
 * Résultat : 'a list
 * Sémantique : Coupe une liste de taile n en deux sous-liste de taille
                proche de n/2
 * Exception : Aucune
*)
(* Tests :
cut [1;2;3;4;5;6;7;8;9;10];;
- : int list * int list = ([1; 3; 5; 7; 9], [2; 4; 6; 8; 10])
*)

let rec cut liste =
        match liste with
        |[]-> ([],[])
        |[t]-> ([t],[])
        |t1::t2::q -> let l1,l2 = cut q
                        in (t1::l1),(t2::l2)
;;



(* Fonction merge
 * Argument :
        * Ordre : 'a -> 'a -> bool
        * l1 : 'a list
        * l2 : 'a list
 * Résultat : 'a list
 * Sémantique : Fusionne deux listes triées en une liste triée
 * Exception : Aucune
*)
(* Tests :
merge (<) [1; 3; 5; 7; 9] [2; 4; 6; 8; 10];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
*)
```



```
let rec merge ordre l1 l2 =
        match (l1,l2) with
        |[],_ -> l2
        |_,[] -> l1
        |(t1::q1),(t2::q2) ->   if ordre t1 t2
                                then t1::(merge ordre q1 l2)
                                else t2::(merge ordre q2 l1)
;;



(* Fonction sort
 * Argument :
        * ordre : 'a -> 'a -> bool
        * liste : 'a list
 * Résultat : 'a list
 * Sémantique : Trie la liste suivant l'ordre passé en paramètre
 * Exception : Aucune
*)
(* Test :
sort (<) [9;2;7;10;3];;
- : int list = [2; 3; 7; 9; 10]
*)

let rec sort ordre liste =
        match liste with
        |[]->liste
        |[_]->liste
        |_->    let l1,l2 = cut liste
                in merge ordre (sort ordre l1) (sort ordre l2)

(*****************************************************************************)
(******                     Fonction Principale                        ******)
(*****************************************************************************)

(* Fonction build_tree
 * Argument :
        * liste : 'a  list : liste d'éléments
 * Résultat : 'a huffmantree
 * Sémantique : Construit l'arbre de Huffman à partir d'une liste d'éléments
 * Exceptions : Si la liste passée en paramètre est vide
 *)
(* Tests :
# build_tree ['t';'e';'x';'t';'e'];;
- : char huffmantree = Node (Leaf 't', Node (Leaf 'x', Leaf 'e'))
# build_tree ['e'];;
- : char huffmantree = Leaf 'e'
*)

let rec build_tree liste =
        match liste with
        |[] -> failwith "Rien à coder !"
        |_ ->   (* On crée la liste des occurrences *)
                let list_oc = occurrence liste []
                (* Que l'on trie par ordre lexicographique *)
                in let list_oc_sort = sort (<) list_oc
                (* et l'on génère l'arbre associé à cette liste *)
                in make_tree list_oc_sort
;;
```

```
(****************************************************************************)
(******                    Fonction de codage                        ******)
(****************************************************************************)



(* Fonction create_alphabet
 * Argument : 'a huffmantree : arbre de Huffman
 * Résultat : ('a * bool list) list
 * Sémantique : Construit "l'alphabet" de tous les caractères et leur chemin
 *              dans l'arbre passé en paramètre
 * Exceptions : Aucune
 *)
(* Test :
# create_alphabet (Node (Leaf 't', Node (Leaf 'x', Leaf 'e')));;
- : (char * bool list) list =
[('t', [false]); ('x', [true; false]); ('e', [true; true])]
*)

let rec create_alphabet h_tree =
        match h_tree with
        |Leaf car -> [(car,[])] (* Si l'arbre est constitué d'un seul caractère,
        il n'a pas de code *)
        (* Sinon, on retourne false pour la branche gauche, et true pour la
        branche droite *)
        |Node (g,d) -> (List.map (fun (e,c) -> e,(false::c)) (create_alphabet g)
)
                @(List.map (fun (e,c) -> e,(true::c)) (create_alphabet d))
;;




(* Fonction find_car
 * Argument :
        * car : 'a : caractère à trouver
        * liste : ('a  * 'b) list :
 * Résultat : 'b
 * Sémantique : Ressort le chemin dans l'arbre pour obtenir le caractère
 * Exceptions : Si la liste passée en paramètre est vide
 *)
(* Test :
# find_car 't' [('t', [false]); ('x', [true; false]); ('e', [true; true])];;
- : bool list = [false]
*)

let rec find_car car liste =
        match liste with
        |[]-> failwith "Pas de caractère dans la liste"
        |(x,bin)::q ->  if x = car (* Si le caractère est trouvé *)
                        then bin (* On retourne le chemin *)
                        else find_car car q (* Sinon on continue de parcourir la
                        liste *)
;;




(* Fonction find_bin_word
 * Argument :
        * liste_car : 'a  list : liste de caractères
        * alphabet : ('a * bool list) list : alphabet
```



```
 * Résultat : bool list
 * Sémantique : Construit le chemin à parcourir dans l'arbre pour obtenir le
 *              mot en entier
 * Exceptions : Aucune
 *)
(* Test :
# find_bin_word ['t';'x'] [('t', [false]); ('x', [true; false]); ('e', [true;
# true])];;
- : bool list = [false; true; false]
*)

let rec find_bin_word liste_car alphabet =
        match liste_car with
        |[] -> []
        |t::q -> (find_car t alphabet)@(find_bin_word q alphabet)
;;




(* Fonction encode
 * Argument :
        * liste : 'a list : liste d'éléments à coder
 * Résultat : 'a huffmantree * bool list
 * Sémantique : Construit l'arbre de Huffman ainsi que le code du mot passé en
 *              paramètre
 * Exceptions : Si la liste passée en paramètre est vide
 *)
(* Test :
#  encode ['t';'e';'x';'t';'e'];;
- : char huffmantree * bool list =
(Node (Leaf 't', Node (Leaf 'x', Leaf 'e')),
 [false; true; true; true; false; false; true; true])
*)

let encode liste =
        match liste with
        |[] -> failwith "Rien à compresser"
        |_ -> let tree = build_tree liste (* Fabrication de l'arbre *)
                in (tree,find_bin_word liste (create_alphabet tree))
                (* On retourne l'arbre et le chemin à parcourir *)
;;



(****************************************************************************)
(******                    Fonction de décodage                      ******)
(****************************************************************************)

(* Fonction read_letter
 * Argument :
        * h_tree : 'a huffmantree : arbre de Huffman
        * liste : bool list : chemin dans l'arbre
 * Résultat : 'a * bool list
 * Sémantique : À partir d'un chemin dans l'arbre, cette fonction ressort le
 *              caractère obtenu, et le reste de la liste de booléen
 * Exceptions : Si la liste est vide alors qu'on est sur un Noeud.
 *)
(* Test :
# read_letter (Node (Leaf 't', Node (Leaf 'x', Leaf 'e'))) [false; true; true;
# true; false; false; true; true] ;;
- : char * bool list = ('t', [true; true; true; false; false; true; true])
```

```
*)

let rec read_letter h_tree liste =
        match h_tree,liste with
        |Leaf car,_ -> (car, liste) (* On est sur une feuille, on retourne le
        caractère et le reste de la liste *)
        |Node _, [] -> failwith "Erreur dans le codage" (* La liste n'est pas
        vide alors que l'on est sur un noeud : problème *)
        |Node (g,d), t::q -> if t (* Si t vaut true *)
                                then read_letter d q (* On parcourt la branche
                                droite *)
                                else read_letter g q (* Sinon la gauche *)
;;


(* Fonction decode
 * Argument :
        * ('a huffmantree * bool list) : couple (Arbre de Huffman, chemins dans
 *                                                      l'arbre)
 * Résultat : 'a list
 * Sémantique : Décode le mot.
 * Exceptions : Aucune
 *)
(* Test :
# decode (Node (Leaf 't', Node (Leaf 'x', Leaf 'e')),
 [false; true; true; true; false; false; true; true])  ;;
 - : char list = ['t'; 'e'; 'x'; 't'; 'e']
*)

let rec decode (arbre,liste_bool) =
        match liste_bool with
        |[] -> []
        |_ -> let (car,n_list) = read_letter arbre liste_bool (* On lit la
        première lettre, et on récupère le reste de la liste *)
                in car::(decode (arbre,n_list))
;;
```