Informatique MP
TP
Compression d'image Bitmap 256 couleurs

Antoine MOTEAU
antoine.moteau@wanadoo.fr

. . . /TP-Bitmap-C.tex (2003)
. . . /TP-Bitmap-C.tex Compilé le jeudi 05 juillet 2018 à 13h 39m 17s avec PDFLaTeX.
Compilable avec LaTeX, PDFLaTeX, LuaLaTeX, XeLaTeX.

Amélioration forte du sujet CCS 2006 (partie I, compression de données)
TP simple de début d'année (application du cours)
Ordre du TP : début d'année (TP 1 ou 2))

Eléments utilisés :

- listes, chaines
- enregistrements, avec champs nommés mutables
- boucles, récursivité
- utilisation de fichiers (en complément et pour information : tout est écrit).

Documents relatifs au TP :

- Texte du TP avec squelette : TP-Bitmap-S.tex, TP-Bitmap-S.dvi, TP-Bitmap-S.pdf
- Texte du TP avec squelette et corrigé : TP-Bitmap-C.tex, TP-Bitmap-C.dvi, TP-Bitmap-C.pdf
- Eléments de programmation initiale : Bitmap-0.ml
- Squelette de programme Caml : Bitmap-S.ml
- Programme corrigé Caml : Bitmap-C.ml
- Fichiers exemples :
  - Test-1.txt (élémentaire)
  - Image1-1.bmp (image .bmp profondeur 8, ide pixel sur 8 bits = 256 couleurs)
  - (les autres sont créés par le programme)
- Autres fichiers exemples :
  - Image3-1.bmp (image .bmp profondeur 3*8, ide pixel sur 3*8 bits = xxxxxx couleurs), de taille multiple de 3 !
  - . . .

Antoine MOTEAU – Lycée POTHIER – Option Informatique MP
TP n^o

COMPRESSION DE DONNEES
IMAGES BITMAP 256 COULEURS.

1 Introduction

1.1 Image couleur bitmap (256 couleurs ou 256 niveaux de gris)

Les images couleurs bitmap 8 bits sont composées de pixels (points colorés), chaque pixel étant représenté par (ou assimilé à) un octet (entier sur 8 bits), qui est le code de la couleur du pixel (de 0 à 255).
L'image est stockée dans un fichier (format .bmp 8 bits), avec en tête de fichier une zone contenant les informations de gestion de l'image (type d'image, longueur d'une ligne, nombre de lignes et autres), suivie des pixels (sous forme d'octets de code de couleur), organisés en ligne, ligne après ligne.

1.2 Octets et caractères

Le type octet (entier sur 8 bits) n'est pas pris en compte sur les machines, mais le type `char` (caractère sur 8 bits) étant pris en compte, on assimilera le type octet au type `char` :

- `int_of_char c` donne le code ascii sur 8 bits (valeur entière de 0 à 255) du caractère `c`.
  Par exemple, `int_of_char 'a';;` renvoie la valeur 97 (entier décimal) .
- `char_of_int i` donne le caractère dont le code ascii vaut `i` .
  Par exemple `char_of_int 97;;` renvoie le caractère `'a'` et `char_of_int 13;;` renvoie le caractère `'\013'` (forme spéciale du caractère de code ascii 13 (retour à la ligne), qui n'est pas éditable).

1.3 Principe de la compression choisie pour ce TP

La nature même des images fait que, sur une même ligne, de nombreux pixels concécutifs sont de même couleur (ciel bleu, nuage blanc, . . .) et on peut espérer réduire sensiblement la taille du fichier en remplaçant une séquence de plusieurs pixels concécutifs égaux par une information du genre (nombre de pixels, couleur), un pixel isolé étant (en général) représenté par lui même.
Dans ce remplacement, pour que l'information "nombre de pixels" ne soit pas confondue avec des valeurs de pixels uniques, il faut utiliser une stratégie particulière :

- le nombre de pixels (facteur de répétition) sera codé sur un octet, donc de valeur au plus 255 (s'il y a plus de 255 répétitions, il faudra s'y prendre en plusieurs fois) ;
- le caractère `'['` sert d'indicateur de répétition : une séquence de n pixels de couleur c (de type `char`) sera codée par la séquence de trois octets : (`'[' , char_of_int n, c`) si $3 < n \leqslant 255$ et sinon elle est inchangée, sauf cas spécial (voir point suivant) ;
- les pixels de couleur codée par `'['`, isolés ou en nombre au plus 3, ne pouvant plus être pris en compte (à cause du choix précédent), seront systématiquement codés avec un facteur de répétition ($n = 1$ ou 2 ou 3) par la séquence de trois octets : (`'[', char_of_int n, '['`);
- la taille de l'image étant connue (information lue en tête de fichier), il n'est pas utile d'envisager de traiter à part un caractère spécial de fin de ligne ni un caractère spécial de fin d'image éventuels.

Remarque. Avec un indicateur de répétition supplémentaire, on pourrait avoir un facteur de répétition codé sur 2 octets (pour les valeurs de 256 à 65535), ou même sur 4 octets (pour les valeurs de 256 à 4294967295).

1.4 Objectif du TP, fichiers initiaux

On se propose d'effectuer le codage et le décodage d'une image bitmap selon le principe exposé ci-dessus.

- Dans un premier temps, on va traiter une image "bidon" sous forme d'une liste de caractères pouvant être répétés, choisis parmi les caractères "éditables" (que l'on peut voir à l'écran).
- Ultérieurement, on pourra s'attaquer à de vraies images couleurs 8 bits, stockées dans un fichier . . . .

Les fichiers de ressources du TP sont initialement dans le dossier MPx/ · · · /TP-Info/BitMap/,
à copier dans votre dossier personnel. Ensuite, vous trouverez normalement dans votre dossier BitMap/

- le texte de ce TP au format .pdf : TP-BitMap-S.pdf,
- Squelette de programme (fonctions à compléter et exemples) : Bitmap-S.ml
- Une image, sous forme de fichier bitmap 256 couleurs : Image1-1.bmp

2 Ressources

2.1 Rappel de quelques fonctions usuelles Caml (extrait de l'aide Caml)

`value int_of_char : char -> int`
Return the ASCII code of the argument.

`value char_of_int : int -> char`
Return the character with the given ASCII code. Raise Invalid_argument "char_of_int" if the argument is outside the range 0 – 255.

`value char_for_read : char -> string`
Return a string representing the given character, with special characters escaped following the lexical conventions of Caml Light.

`value print_char : char -> unit`
Print the character on standard output.

`value print_string : string -> unit`
Print the string on standard output.

`value print_int : int -> unit`
Print the integer, in decimal, on standard output.

`value print_newline : unit -> unit`
Print a newline character on standard output, and flush standard output. This can be used to simulate line buffering of standard output.

- longueur d'une chaîne de caractères `s` : `string_length s;;`
- i-ème élément (à partir de 0) d'un chaîne `s` : `s.[i] <- char_of_int (1 + int_of_char s.[i]);;`
- concaténation de chaînes : `s3 := (string_of_char 'a') ^ s1 ^ ( s2 ^ !s3);;`
- longueur d'un vecteur `v` : `vect_length v;;`
- i-ème élément (à partir de 0) d'un vecteur `v` : `v.(i) <- v.(i) + 1;;`
- longueur d'une liste `L` : `list_length L;;` (à éviter) ;
- tête et queue de liste : `hd L;; tl L;;`
- ajout d'un élément `a` en tête de la liste `L`, par `a::L ;;`
- ajout multiple en tête de liste, par `a::b::c::Q` ou `a::b::c::[d];;`
- concaténation de listes : `L3 := L1 @ ( L2 @ !L3 );;` (à éviter).

2.2 Ressources complémentaires, à construire

2.2.1 Conversion d'une chaîne de caractères en liste de caractères

Pour éviter la saisie fastidieuse d'une liste de caractères, par exemple :

```
let image = ['a'; 'b'; 'u'; 'u'; 'u'; 'u'; 'u'; 'u'; 'c'; 'd'; 'e'; 'f'; 'f'; 'f'; 'f';
             'f'; 'g'; 'h'; 'i'; 'i'; 'i'; 'i'; 'i'; 'i'; 'j'; 'k'; 'l'; 'l'; 'l'; 'l';
             'l'; 'l'; 'l'; 'l'; 'l'; 'l'; 'm'; 'n'; 'o'; 'o'; 'o'; 'p'];;
```

on va créer une fonction `char_list_of_string` , qui convertit une chaîne de caractères (`string`) en la liste de ces mêmes caractères (`char list`). La liste précédente s'obtiendra par :

```
let image = char_list_of_string "abuuuuuucdeffffghiiiiiijkllllllllllmnooop";;
```

Écrire la fonction `char_list_of_string : string -> char list` (en 3 lignes élémentaires). Cette fonction peut utiliser (exceptionnellement) une variable référence (`ref`), utiliser une boucle d'itération (`for`), mais n'utilisera pas la concaténation de listes (@).

De même, pour faciliter la lecture d'une liste de caractères, on introduit la fonction `string_of_char_list`, réciproque de la précédente. Par exemple, avec la liste précédente (`image`), `string_of_char_list image` renvoie la chaîne de caractères associée (`"abuuuuuucdeffffghiiiiiijkllllllllllmnooop"`).

Écrire la fonction `string_of_char_list : char list -> string` (en 4 lignes élémentaires). Cette fonction n'utilisera pas de variables référence (`ref`) ni de structure itérative (`while, for, ...`) mais sera récursive et utilisera la concaténation de chaines (∧).

3 Format temporaire de description de répétitions

Comme on est intéressé par des séquences où un même caratère est répété de nombreuses fois (éventuellement plus de 255 fois), pour faciliter la saisie des exemples de test, on introduit un format temporaire pour décrire les listes de caratères avec répétition.

Par exemple, la séquence (ou liste) `abbbbbbbbcddd ...ddee66666g[xyz`, où `d` est répété 1664 fois, difficile à saisir, sera représentée initialement par la séquence (ou liste) `a[8]bc[1664]dee[5]6g[1][xyz` où

- le caractère `[` sert d'indicateur de début de facteur de répétition ;
- le facteur de répétition est représenté par la séquence de ses chiffres décimaux (0 à 9)
- la fin du facteur de répétition est marquée par le caractère `]` suivi du caractère à répéter
- le caractère effectif `[` est représenté avec répétition systématique, soit, s'il est isolé, par `[1][`

La séquence (liste) `a[8]bc[1664]dee[5]6g[1][xyz` sera transformée par la fonction `expand_temp` (que l'on va écrire) en la séquence (liste) `abbbbbbbbcddd ...ddee66666g[xyz`, où `d` est répété 1664 fois,

Ainsi, la séquence (liste) `abbbbbbbbcddd ...ddee66666g[xyz`, où `d` est répété 1664 fois, s'obtiendra facilement par l'instruction : `expand_temp ( char_list_of_string "a[8]bc[1664]dee[5]6g[1][xyz" );;`

3.1 Expansion d'une liste avec répétitions sous format temporaire

On rappelle que les codes ASCII des caractères 0 à 9 sont (inconnus) concécutifs.

Écrire la fonction `lire_nombre : char list -> int * char list` (5 lignes élémentaires) ,

dont le paramètre est une liste comportant en tête des chiffres décimaux, suivis par le caractère `]` et qui renvoie le couple formé par le nombre de tête et le reste de la liste (privée des chiffres décimaux de tête et du caractère `]` qui les suit).

Par exemple, `lire_nombre ( [ '1'; '9'; '4'; '6'; ']' ] @ q);;` renvoie le couple `(1946,q)`.

Cette fonction

- fera appel à une fonction interne récursive,
- ne traitera pas les cas d'erreurs (on supposera que le paramètre est correctement constitué),
- n'utilisera pas de structure itérative (`while, for, ...`) ni de variable référence, ni @ .

Écrire la fonction `multiple : int -> char -> char list` (4 lignes élémentaires) ,

telle que l'appel `(multiple i a)` renvoie la liste composée de `i` caractères identiques à `a`.

Cette fonction

- sera récursive ou fera appel à une fonction interne récursive,
- ne traitera pas les cas d'erreurs (on supposera que les paramètres sont correctement constitués),
- n'utilisera pas de structure itérative (`while, for, ...`) ni de variable référence, ni @ .

Écrire la fonction `expand_temp : char list -> char list` (4 lignes élémentaires).

Cette fonction, qui a été décrite précédement,

- sera récursive,
- ne traitera pas les cas d'erreurs (on supposera que le paramètre est correctement constitué),
- n'utilisera pas de structure itérative (`while, for, ...`) ni de variable référence,
- pourra utiliser la concaténation de liste (@) (on pourrait l'éviter avec une "acrobatie").

TESTS : Bien qu'un algorithme bien pensé et bien conçu conduise toujours à un programme juste, on n'est pas à l'abri de quelques surprises .... Un test se déroule en plusieurs étapes répétitives :

1. Tester chaque fonction élémentaire, a) sur un exemple simple b) sur un exemple complexe
   (attendre que tout un programme soit fini pour le tester conduit à beaucoup de confusion).
2. Tester chaque assemblage de fonctions, a) sur un exemple simple b) sur un exemple complexe.
3. Tester tout le programme, a) sur un exemple simple b) sur un exemple complexe.
   En cas de problème(s), il faut revenir en arrière pour cerner la zone (ou les zones) erronée(s).

Un programme qui "tourne" est un programme dans lequel on n'a pas su voir les erreurs ou pas encore eu l'occasion de rencontrer les erreurs.

3.2 Réciproque : compression avec répétitions sous format temporaire

Pendant qu'on y est, juste pour le plaisir, on va écrire la fonction `compress_temp`, réciproque de la précédente (`expand_temp`), telle que, par exemple, avec `u = char_list_of_string "abbbbbbbbcddd...dde66666g[xyz"`, `string_of_char_list (compress_temp u);;` donne comme résultat `"a[8]bc[1664]de[5]6g[1][xyz"`.

(1664 fois)

Remarque : on ne fera la compression que lorsqu'elle est pertinente, c'est à dire lorsque le caractère est répété au moins 5 fois (à l'exception du caractère `[` qui est toujours répété au moins une fois) et ainsi,

`string_of_char_list (compress_temp (char_list_of_string "aaabbbbbbbbbccddddddde66666g[xyz"))` donne comme résultat `"aaa[8]bcc[6]de[5]6g[1][xyz"`.

3.2.1 Structure de donnée pour le comptage

On définit la structure de données `Accumulateur` permettant de représenter, sous forme d'un *ensemble à champs nommés* (ou "*enregistrement*") modifiables en place (déclarés `mutable`), l'état de la lecture d'un caractère avec son facteur de répétition, par la déclaration :

```
type Accumulateur = { mutable caractere : char; mutable compte : int };;
```

On remarquera que la modification d'une variable de type `Accumulateur`, passée en paramètre à une fonction, se fait en place (par effet de bord). Par exemple, avec la fonction (qui ne renvoie rien) :

```
let truc accu = .... ; accu.caractere <- 'b'; accu.compte <- 1 + accu.compte; () ;;
(* truc : Accumulateur -> unit *)
```

à l'issue des instructions suivantes :

```
let accu = { caractere = 'a' ; compte = 2 };; (* création et initialisation *)
truc accu;;                                   (* effet de bord sur accu     *)
```

les champs `caractere` et `compte` de la variable `accu` seront aux valeurs `'b'` et `3` .

3.2.2 Transformation d'un entier (positif) en la liste de ses chiffres décimaux.

Écrire la fonction `char_list_of_int : int -> char list` (4 lignes élémentaires) ,

telle que l'appel `(char_list_of_int i)` renvoie la liste composée des chiffres de l'écriture décimale de l'entier positif `i`. Par exemple, `char_list_of_int 1946` renvoie la liste `[ '1'; '9'; '4'; '6' ]`.

Cette fonction, qui fera appel à une fonction interne récursive,

- ne traitera pas les cas d'erreurs (on supposera que le paramètre est correctement constitué),
- n'utilisera pas de structure itérative (`while, for, ...`) ni de variable référence, ni `@` .

3.2.3 Vidage d'un accumulateur au format temporaire.

Écrire la fonction `vide_accumulateur_temp : Accumulateur -> char list` (en 4 ou 5 lignes) ,

telle que l'appel `(vide_accumulateur_temp accu)` renvoie

- si `accu.compte > 4` ou `accu.caractere = '['` , la liste composée de `[` , suivie des chiffres de l'écriture décimale de `accu.compte` , puis de `]` et enfin de `accu.caractere`,
- sinon, la liste composée de `accu.compte` fois `accu.caractere`,

le champ `compte` du paramètre `accu` étant remis à `0` avant la terminaison.

Par exemple,

avec `let accu = { caractere = 'a' ; compte = 1946 };; vide_accumulateur_temp accu;;`
on obtient la liste `[ '['; '1'; '9'; '4'; '6'; ']'; 'a' ]` ,

avec `let accu = { caractere = 'a' ; compte = 3 };; vide_accumulateur_temp accu;;`
on obtient la liste `[ 'a'; 'a'; 'a' ]`

avec `let accu = { caractere = '[' ; compte = 1 };; vide_accumulateur_temp accu;;`
on obtient la liste `[ '['; '1'; ']'; '[' ]`,

et dans tous ces cas, `accu.compte` se retrouve à la valeur `0` .

Cette fonction

- ne traitera pas les cas d'erreurs (on supposera que le paramètre est correctement constitué),
- n'utilisera pas de structure itérative (`while, for, ...`) ni de variable référence,
- pourra utiliser la concaténation de liste (`@`) (on pourrait l'éviter avec une "acrobatie").

3.2.4 La fonction de compression temporaire.

Écrire la fonction `compress_temp : char list -> char list` (en au plus 9 lignes) ,

telle que l'appel `(compress_temp u)` renvoie la liste déduite de u en remplaçant toute sous-liste de u qui est la répétition n fois ($n \geqslant 5$) d'un même caractére c par la sous-liste formée par `'['` suivi des caractères de l'écriture décimale de n puis de `']'` et enfin de c. Une sous-liste de u constituée du caractère `'['` isolé ou répété n fois étant systématiquement traitée comme une répétition.

Par exemple,
`string_of_char_list (compress_temp (char_list_of_string "aabbbbbccddddde66666g[x"));;`
donne la chaîne `"aa[5]bcc[6]de[5]6g[1][x"` .

Cette fonction, qui fera appel à une fonction interne récursive,

- ne traitera pas les cas d'erreurs (on supposera que le paramètre est correctement constitué),
- n'utilisera pas de structure itérative (`while, for, ...`) ni de variable référence,
- pourra utiliser la concaténation de liste (@) (on pourrait l'éviter avec une "acrobatie").

TESTS : Mêmes consignes que précédemment.

Conclusion. En fait, on a pratiquement réalisé le TP (dans le cas des listes de caractères).
Il reste à adapter ce que l'on a déjà écrit au cas où le facteur de compression est codé sur 1 octet, après l'indicateur de répétition `'['`, puis à passer au cas des fichiers image bitmap 256 couleurs ....

4 Codage, décodage avec répétitions codées sur un octet

Comme dit dans l'introduction, on utilise le caractère `[` comme indicateur de répétition, le nombre de répétitions ($\geqslant 4$) étant codé sur 1 octet, quitte à s'y prendre en plusieurs fois.
Par exemple, la séquence (liste) `aabbbbbbbbcddd...dde66666g[x` (1664 fois) que l'on peut obtenir par l'instruction
`expand_temp (char_list_of_string "aa[8]bc[1664]de[5]6g[1][x");;` sera compressée en :

```
'a' 'a' '[' '008' 'b' 'c' '[' '255' 'd' '[' '255' 'd' '[' '255' 'd' '[' '255' 'd'
'[' '255' 'd' '[' '255' 'd' '[' '134' 'd' 'e' '[' '005' '6' 'g' '[' '001' '[' 'x'
```

les espaces étant ici non significatifs, la répétition $1664 > 255$ se décomposant en 7 codages de répétitions ($1664 = 6 \times 255 + 134$) et où `'008'` (par exemple) désigne le caractère de code ascii 8 (non éditable).

4.1 Compression (codage)

On utilise la même stucture de donnée (type `Accumulateur`) que précédemment pour le même objectif.

4.1.1 Vidage d'un accumulateur.

Écrire la fonction `vide_accumulateur : Accumulateur -> char list` (en au plus 8 lignes) ,

telle que l'appel `(vide_accumulateur accu)` renvoie

- si `accu.compte > 3` ou `accu.caractere = '['` , une liste composée de `'['`, suivi d'un octet de valeur puis de `accu.caractere`, répétés autant de fois que nécessaire,
- sinon, la liste composée de `accu.compte` fois `accu.caractere`,

le champ `compte` du paramètre `accu` étant remis à `0` avant la terminaison.

Par exemple,
avec `let accu = { caractere = 'a' ; compte = 1664 };; vide_accumulateur accu;;`
on obtient la liste `[ '['; '255'; 'a'; '['; '255'; 'a'; '['; '255'; 'a'; '['; '255'; 'a'; '['; '255'; 'a'; '['; '255'; 'a'; '['; '134'; 'a' ]` ,

avec `let accu = { caractere = 'a' ; compte = 3 };; vide_accumulateur accu;;`
on obtient la liste `[ 'a'; 'a'; 'a' ]` ,

avec `let accu = { caractere = '[' ; compte = 1 };; vide_accumulateur accu;;`
on obtient la liste `[ '['; '001'; '[' ]`,
et dans tous ces cas, `accu.compte` se retrouve à la valeur `0` .

Cette fonction, qui fera appel à une fonction auxiliaire récursive,

- ne traitera pas les cas d'erreurs (on supposera que le paramètre est correctement constitué),
- n'utilisera pas de structure itérative (`while, for, ...`) ni de variable référence, ni @ .

4.1.2 La fonction de compression (codage).

Écrire la fonction `codage : char list -> char list` (en au plus 9 lignes) ,

telle que l'appel `(codage u)` renvoie la liste déduite de `u` en remplaçant toute sous-liste de `u` qui est la répétition n fois ($n \geqslant 4$) d'un même caractére c par autant de sous-listes, formées par `'['` suivi d'un octet de valeur (sous forme de `char`) puis de c, que nécessaire. Une sous-liste de `u` constituée du caractère `'['` isolé ou répété n fois étant systématiquement traitée comme une répétition.

Par exemple, `codage (char_list_of_string "aabbbbbcddddde66666g[x");;` donne la séquence (liste) `'a' 'a' '[' '005' 'b' 'c' '[' '006' 'd' 'e' '[' '005' '6' 'g' '[' '001' '[' 'x'` .

Cette fonction, qui fera appel à une fonction interne récursive,

- ne traitera pas les cas d'erreurs (on supposera que le paramètre est correctement constitué),
- n'utilisera pas de structure itérative (`while, for, ...`) ni de variable référence,
- pourra utiliser la concaténation de liste (@) (on pourrait l'éviter avec une "acrobatie").

4.2 Décompression (décodage).

Écrire la fonction `decodage : char list -> char list` (en au plus 3 lignes) ,

telle que l'appel `decodage v` où `v` est une liste qui a été codée par let `v := codage u;;` restaure toutes les séquences compactées de `v` sous leur forme initale, ide renvoie la même liste que `u`.

Par exemple, après exécution des instructions suivantes

```
let s = "[2]a[13]b[3]5[1][xyz";;
let u = expand_temp (char_list_of_string s);;
let v = codage u;;
let w = decodage v;;
let x = string_of_char_list (compress_temp w);;
```

la variable `x` contient la chaîne `"aa[13]b555[1][xyz"`.

Cette fonction, qui sera récursive,

- ne traitera pas les cas d'erreurs (on supposera que le paramètre est correctement constitué),
- n'utilisera pas de structure itérative (`while, for, ...`) ni de variable référence,
- pourra utiliser la concaténation de liste (@) (on pourrait l'éviter avec une "acrobatie").

TESTS : Mêmes consignes que précédemment.

5 Facultatif : Compression de fichiers .bmp 8 bits (256 couleurs)

5.1 Informations sur la gestion des fichiers en Caml

On se limite ici au minimum sur les fichiers : fichiers de caractères, que l'on lit ou écrit caractère par caractères, du début à la fin.

L'information utile se trouve dans "CAML–light Help" aux rubriques :

"Index // the core library // io : buffered input and output"

"Index // the core library // io : buffered input and output // General output functions"

"Index // the core library // io : buffered input and output // General input functions"

5.1.1 Ressources élémentaires de manipulation de fichiers

`type in_channel`
`type out_channel`
The abstract types of input channels and output channels.

`exception End_of_file`
Raised when an operation cannot complete, because the end of the file has been reached.

`value open_out : string -> out_channel`
Open the named file for writing, and return a new output channel on that file, positionned at the beginning of the file. The file is truncated to zero length if it already exists. It is created if it does not already exists. Raise sys__Sys_error if the file could not be opened.

`value open_out_bin : string -> out_channel`
Same as `open_out`, but the file is opened in binary mode, so that no translation takes place during writes. On operating systems that do not distinguish between text mode and binary mode, this function behaves like `open_out`.

`value output_char : out_channel -> char -> unit`
Write the character on the given output channel.

`value close_out : out_channel -> unit`
Close the given channel, flushing all buffered write operations. The behavior is unspecified if any of the functions above is called on a closed channel.

`value open_in : string -> in_channel`
Open the named file for reading, and return a new input channel on that file, positionned at the beginning of the file. Raise sys__Sys_error if the file could not be opened.

`value open_in_bin : string -> in_channel`
Same as `open_in`, but the file is opened in binary mode, so that no translation takes place during reads. On operating systems that do not distinguish between text mode and binary mode, this function behaves like `open_in`.

`value input_char : in_channel -> char`
Read one character from the given input channel. Raise End_of_file if there are no more characters to read.

`value close_in : in_channel -> unit`
Close the given channel. Anything can happen if any of the functions above is called on a closed channel.

5.1.2 Illustration par des exemples élémentaires

On utilise la programmation avec gestion d'erreur(s) :

```
try
   séquence d'instructions
with End_of_file -> une instruction OU begin séquence d'instructions end;
instruction suivante; ...
```

où on exécute la séquence d'instructions située juste après `try` et,

- si on n'y rencontre pas d'erreur, à la fin de cette séquence, le programme continue avec "`instruction suivante`",
- en cas d'erreur (l'erreur attendue ici est "`End_of_file`"), le programme abandonne la séquence en cours et se branche sur l'instruction (ou le bloc d'instructions) qui suit "`with End_of_file ->` ".
  Ensuite, le programme pourra continuer avec "`instruction_suivante`",

1. **Lecture dans un fichier de caractères, caractère par caractère.**

   Supposons que l'on ait créé (avec le bloc-notes), dans le dossier "h :\InfoMP\TP-info\Bitmap", un fichier "test-1.txt" contenant la simple phrase "Ceci est le contenu du fichier test-1.txt." .

```
let rec lire i =
   try let a = input_char i in a :: (lire i)
   with End_of_file -> [ ]
;;
(* lire : in_channel -> char list = <fun> *)

let Myfile_in = open_in "h:/InfoMP/TP-info/Bitmap/test-1.txt";;
let u = string_of_char_list (lire Myfile_in);;
close_in Myfile_in;;
```

   On doit trouver dans u la chaîne "`Ceci est le contenu du fichier test-1.txt.`" .

   PROBLEME : Si le fichier contient le caractère `EOF` (End Of File), de code ascii `'\026'`, la lecture s'arrête à ce caractère (non éditable), puisque c'est le caractère spécial indiquant la fin d'un fichier de caractères. Par exemple, si le contenu du fichier précédent avait été

```
"Ceci est le contenu du fi" ^ char_for_read (char_of_int 26) ^ "chier test-1.txt."
```

   la lecture précédente n'aurait redonné que "`Ceci est le contenu du fi`" !

   Il faut ouvrir le fichier comme un fichier binaire, avec `open_in_bin`, pour pouvoir traiter aussi les caractères spéciaux, jusqu'à la fin réelle du fichier.

2. **Écriture dans un fichier de caractères, caractère par caractère.**

   On va écrire une suite de caractères dans le fichier "test-2.txt", situé dans le répertoire "h :\InfoMP\TP-info\Bitmap". Si le fichier n'existe pas, il sera créé et **s'il existe déjà, il sera vidé** à l'ouverture.

```
let rec ecrire o = function
    | [ ]   -> ()
    | a::q  -> output_char o a; ecrire o q
;;
(* ecrire : out_channel -> char list -> unit = <fun> *)

let Myfile_out = open_out "h:/InfoMP/TP-info/Bitmap/test-2.txt";;
let u = char_list_of_string "Et maintenant, le fichier test-2.txt.";;
ecriture Myfile_out u;;
close_out Myfile_out;;
```

   En ouvrant (avec le bloc-notes ou en le lisant comme précédemment) le fichier test-2.txt, on y trouve la phrase "Et maintenant, le fichier test-2.txt." (du moins je l'espère).

   PROBLEME : si la chaîne u contient le caractère `EOF` (End Of File), de code ascii `'\026'`, l'écriture (et la lecture) s'arrête à ce caractère (non éditable), puisque c'est le caractère spécial indiquant la fin d'un fichier de caractères. Par exemple, si le contenu de la chaîne u avait été

```
"Et maintenant, le fi" ^ char_for_read (char_of_int 26) ^ "chier test-2.txt."
```

   le contenu du fichier aurait été limité à `"Et maintenant, le fi"` !

   > Il faut ouvrir le fichier comme un fichier binaire, avec `open_out_bin`, pour pouvoir traiter aussi les caractères spéciaux, jusqu'à la fin réelle des caractères.

5.2 Compression de l'image contenue dans un fichier .bmp 8 bits

Il suffit d'adapter ce que l'on a déjà écrit, dans le cas des listes, au cas des fichiers de caractères, ouverts en binaire, mais **sans utiliser la récursivité** :

> sinon, à chaque caractère, la fonction récursive se réappelle, en montant (au moins) son adresse de retour dans la pile (limitée) et, dans le cas d'un fichier un pneu gros, on arrive à l'erreur `Out_of_memory` !

On remplace donc l'écriture récursive par une écriture **itérative** (ici, une boucle `while`).

On assure dans tous les cas la fermeture des fichiers (pour ne pas avoir de fichiers qui restent ouverts).
On utilise le type `Accumulateur`, la constante `indic (= '[')`, les foonctions `multiple`, `vide_accu` (déjà écrits) et la fonction auxiliaire interne `vider` (récursive, mais sur un nombre faible d'appels).
Remarque : on pourrait ré-écrire `vide_accu` pour qu'elle vide effectivement dans le fichier de sortie.

```
let coder_fichier ch_in ch_out =
  let file_in = open_in_bin ch_in and file_out = open_out_bin ch_out   (* bin *)
  in
  let rec vider = function      (* output la liste fournie par (vide_accu enc) *)
    | []   -> ()
    | a::q -> output_char file_out a; vider q
  in
  let enc = {caractere = 'a'; compte = 0}
  and fini = ref false in
  while not !fini
  do try let a = input_char file_in in
         if a = enc.caractere
         then enc.compte <- 1 + enc.compte
         else ( vider (vide_accu enc);
                enc.caractere <- a; enc.compte <- 1  )
     with End_of_file -> ( vider (vide_accu enc); fini := true )
  done;
  close_in file_in; close_out file_out
;;
(* coder_fichier : string -> string -> unit = <fun> *)
```

5.3 Décompression de l'image contenue dans un fichier .bmp 8 bits

Il suffit d'adapter ce que l'on a déjà écrit, dans le cas des listes, au cas des fichiers de caractères, ouverts en binaire, mais, de même, **sans utiliser la récursivité** !

On assure dans tous les cas la fermeture des fichiers (pour ne pas avoir de fichiers qu restent ouverts).

```
let decoder_fichier ch_in ch_out =
  let file_in = open_in_bin ch_in and file_out = open_out_bin ch_out in    (* bin *)
  let fini = ref false in
  while not !fini
  do try let a = input_char file_in in
         if a = '['
         then let i = int_of_char (input_char file_in) in
              let c = input_char file_in in
              for k = 1 to i do output_char file_out c done
         else output_char file_out a
     with End_of_file -> fini := true
  done;
  close_in file_in; close_out file_out
;;
(* decoder_fichier : string -> string -> unit = <fun> *)
```

5.4 Tests

```
let dir_name = "h:/InfoMP/TP-info/Bitmap";;
```
(chemin à adapter).

5.4.1 Test élémentaire (fichiers crées automatiquement)

```
let ch_file_1 = dir_name ^ "/file-1.txt";;   (* fichier à coder                 *)
let ch_file_2 = dir_name ^ "/file-2.txtc";;  (* ext + c = codage (du précédent) *)
let ch_file_3 = dir_name ^ "/file-3.txt";;   (* décodage du précédent           *)

let Myfile_out = open_out_bin ch_file_1;;
let u1 = expand_temp (char_list_of_string "[2]a[513]b[3]5[1][xyz" );;
ecrire Myfile_out u1;;
close_out Myfile_out;;

coder_fichier ch_file_1 ch_file_2;;

let Myfile_in = open_in_bin ch_file_2;;
let v2 = lire Myfile_in;;
close_in Myfile_in;;

decoder_fichier ch_file_2 ch_file_3;;

let Myfile_in = open_in_bin ch_file_3;;
let s3 = string_of_char_list (compress_temp (lire Myfile_in));;
close_in Myfile_in;;
```

5.4.2 Big Test sur big fichier image bitmap (.bmp 8, 256 couleurs)

Remarque : un fichier image réelle sera peu lisible ! On vérifiera, après codage du fichier 1 dans le fichier 2, que le décodage du fichier 2 dans le fichier 3 donne la même taille et la même image que le fichier 1.

- Fichier .bmp de 17 062 octets $\xrightarrow{\text{coder}}$ 16 964 octets $\xrightarrow{\text{decoder}}$ 17 062 octets = IDEM mais...
- Fichier .bmp de 2 534 454 octets $\xrightarrow{\text{coder}}$ 2 551 586 octets $\xrightarrow{\text{decoder}}$ 2 534 454 octets = IDEM mais...

En fait, ces deux fichiers étaient des fichiers .bmp à profondeur de couleur 24 (pixel codé sur 3 octets).

Il faut travailler avec des fichiers .bmp à profondeur de couleur 8 (pixel codé sur 1 octet).

- Fichier **Image1-1.bmp** (8), de 845 878 octets $\xrightarrow{\text{coder}}$ 228 808 octets $\xrightarrow{\text{decoder}}$ 845 878 octets = IDEM

Ce fichier .bmp 8 a été compressé au quart de sa taille initiale, sans perte d'informations !

6 Compression de fichiers .bmp $x \times 8$ bits (2^{8x} couleurs)

Un fois ce qui précède fait, j'ai du galérer pour trouver des exemples d'images .bmp 8 bits à peu près intéressantes. C'est là que l'on se rend compte que les choses évoluent **...**

- Avant 1960, on n'avait même pas la télé. En 1960, on regardait "Rintintin" à la vitrine du marchand de télé et j'avais 15 ans quand j'ai vu pour la première fois une télé couleur.
- En 1970, si on avait la télé, parfois en couleur, il n'y avait pas de micro-ordinateurs et, sur les (énormes) ordinateurs de l'époque, il n'y avait pas d'écran ... et les claviers étaient rudimentaires !
- En 1980 (au siècle dernier, vous n'étiez pas nés et la souris non plus), les rares possesseurs d'un micro-ordinateur étaient fort heureux d'avoir un écran monochrome (noir et blanc, 350×200) !
  Parfois, on pouvait se brancher sur la télé couleur et avoir (oh miracle !) 16 couleurs, mais c'était la galère. On voyait bien des écrans (on disait "moniteurs") couleurs (16 couleurs) dans les catalogues, mais j'te dis pas le prix ... et certains évoquaient même des écrans 256 couleurs (un rève !)
- Dix ans plus tard (1990), on avait presque tous des écrans CRT 256 couleurs(hé oui, maintenant cela fait ringard), avec plein d'images au format .bmp 8 bits.
- Maintenant, il n'y a plus que des écrans plats. On ne trouve pratiquement plus d'images natives au format .bmp 8 bits et j'ai du rechercher dans mes vieilles archiches des exemples d'images couleur .bmp 8 bits à peu près intéressantes (comme quoi il ne faut jamais rien jeter).

En 1980, la mémoire s'exprimait en Kilo octets (Ko), (le Mo était encore un rève et le Go n'était même pas inventé), quelle soit interne (4 Ko, 48 Ko, ...), de stockage (magnétophone, disquette 90 Ko, 360 Ko, ...). La vitesse du processeur s'exprimait en Khz (sans M ni G) ! Enfin, c'était le bon temps ... .

Dans un fichier .bmp z bits (profondeur de couleur z), les couleurs sont codées sur z bits, ce qui permet permet d'avoir 2^z couleurs par pixel, chaque pixel étant représenté par son code de couleur, donc codé sur z bits. Ainsi, on a différents formats .bmp, selon les profondeurs de couleur, par exemple :

$z = 8 \ = 1 \times 8$: 256 couleurs (ou niveaux de gris), avec un pixel par octet (cela se fait rare ...),
$z = 24 = 3 \times 8$: $2^24 = 16777216$ couleurs, un pixel sur 3 octets (assez courant),
$z = 32 = 4 \times 8$: $2^32 = 4294967296$ couleurs, un pixel sur 4 octets, etc ...

On va adapter ce qui précède au cas des formats .bmp $z = x \times 8$ bits.

Remarques. Si la méthode de compression utilisée ici pour le format .bmp 8 bits semble intéressante, il ne devrait pas en être de même pour les formats .bmp $x \times 8$ bits avec $x > 1$: avec beaucoup de couleurs, il y a beaucoup de nuances et **... le ciel bleu n'est pas ... si bleu que ça**, c'est à dire pas bleu uniforme.

Les pixels isolés étant rares, il pourrait être pertinent de grouper systématiquement les pixels concécutifs même isolés, sans l'indicateur `[`, avec deux octets (facteur de répétition, valeur) **... à voir ...**.

Les fichiers .bmp $z = x \times 8$ n'ont pas forcément une taille multiple de x en octets (à cause des informations de gestion en tête de fichier). Pour simplifier, je me limite à ceux dont la taille est multiple de x (sinon il faudrait traiter à part l'entête du fichier) et, dans ce cas, je compresse aussi les informations de gestion.

6.1 Programme de compression et décompression de fichier .bmp $x \times 8$

6.1.1 Environnement et adaptation des concepts précédents)

```
let dir_name = "h:/InfoMP/TP-info/Bitmap";;              (* directory des fichiers à traiter *)
let indic = '[';;                                        (* octet indicateur de répétition   *)
type Accumulateur_pixel = { mutable pixel : char list;   (* sur x octets *)
                            mutable compte : int }
;;
let rec out_char_list f_out = function
  | []   -> ()
  | a::q -> output_char f_out a; out_char_list f_out q
;;
(* out_char_list : out_channel -> char list -> unit = <fun> *)

let in_pixel f_in x = rev (lire x)                           (* lire x octets *)
  where rec lire = function
    | 0 -> []
    | i -> (input_char f_in) :: (lire (i-1) )
;;
(* in_pixel : in_channel -> int -> char list = <fun> *)
(* je ne sait pas faire sans le "rev" : il y a un truc bizarre en Caml *)
```

```
let vide_accu_pixel f_out enc  =              (* directement dans le fichier *)
   let p = enc.pixel and i = enc.compte in
   enc.pixel <- []; enc.compte <- 0;
   vide i
   where rec vide = function
     | i when i <= 0     -> ()
     | i when (i < 4) & (hd p <> indic)
                         -> for k = 1 to i do out_char_list f_out p done
     | i when (i <= 255) -> out_char_list f_out (indic :: (char_of_int i) :: p)
     | i                 -> out_char_list f_out (indic :: (char_of_int i) :: p); vide (i-255)
;;
(* vide_accu_pixel : out_channel -> Accumulateur_pixel -> unit = <fun> *)
```

6.1.2 Compression fichier .bmp $x \times 8$ bits, de taille multiple de 3

```
let coder_fichier_bmp x ch_in ch_out =          (* compression .bmp x fois 8 bits *)
  let file_in = open_in_bin ch_in and file_out = open_out_bin ch_out in    (* bin *)
  let enc = {pixel = [] ; compte = 0}
  and fini = ref false in
  while not !fini
  do try let a = in_pixel file_in x in
         if a = enc.pixel                        (* égalité de listes *)
         then enc.compte <- 1 + enc.compte
         else ( vide_accu_pixel file_out enc;
                enc.pixel <- a ; enc.compte <- 1
              )
     with End_of_file -> ( vide_accu_pixel file_out enc; fini := true )
  done;
  close_in file_in; close_out file_out
;;
(* coder_fichier_bmp : int -> string -> string -> unit = <fun> *)
```

6.1.3 Décompression fichier .bmp $x \times 8$ bits, de taille initiale multiple de 3

```
let decoder_fichier_bmp x ch_in ch_out =    (* décompression .bmp x fois 8 bits *)
  let file_in = open_in_bin ch_in and file_out = open_out_bin ch_out in  (* bin *)
  let fini = ref false in
  while not !fini
  do try let a = input_char file_in in
         if a = '['
         then let i = int_of_char (input_char file_in) in
              let p = in_pixel file_in x in for k = 1 to i do out_char_list file_out p done
         else ( output_char file_out a;
                for k = 1 to x-1 do let b = input_char file_in in output_char file_out b done )
     with End_of_file -> fini := true
  done;
  close_in file_in; close_out file_out
;;
(* decoder_fichier_bmp : int -> string -> string -> unit = <fun> *)
```

6.2 Test de compression et décompression d'un fichier .bmp 3×8

On trouve la taille exacte et la valeur de $z = x \times 8$ dans les propriétés (Windows) du fichier :

- sous l'onglet "Général", on a la taille exacte (en octets),
- sous l'onglet "Résumé", option avancé, on a la profondeur de couleur z.

```
let ch_file_1 = dir_name ^ "/Image3-1.bmp";;  (* 1 440 054 octets, profondeur 24 = 3 x 8 *)
let ch_file_2 = dir_name ^ "/Image3-2.bmpc";; (* ext + c pour compressé *)
let ch_file_3 = dir_name ^ "/Image3-3.bmp";;
coder_fichier_bmp 3 ch_file_1 ch_file_2;;    (* 1 440 054 octets --> 1 162 437 octets *)
decoder_fichier_bmp 3 ch_file_2 ch_file_3;;  (* 1 162 437 octets --> 1 440 054 octets)*)
```

La taille initiale est bien, dans ce cas particulier, un facteur de 3. Après compression puis décompression, on retrouve la même taille et la même image (ouf !). Comme prévu, la réduction est moins significative que dans le cas des fichiers .bmp 256 couleurs : on a compressé à 80% de la taille initiale (au lieu de 25%).

$< \quad \mathscr{FIN} \quad >$ (énoncé du TP)

7 Squelette de programme (non exécutable)

```
(* Directory : du programme et des fichiers exemples *)
let dir_name = "h:/InfoMP/TP-info/Bitmap";;              (* A ADAPTER ! *)

(*-- 2.2. Ressources complémentaires -------------*)
let char_list_of_string s =                            A ECRIRE (3 lignes)
;;
(* char_list_of_string : string -> char list = <fun> *)
let v = char_list_of_string "abuuuuuucdeffffghiiiiiijkllllllllllmnooop";;

let rec string_of_char_list = function                                    A ECRIRE (2 lignes)
    | []   ->                                    A ECRIRE (1 ligne)
    | a::q ->                                    A ECRIRE (1 ligne)
;;
(* string_of_char_list : char list -> string = <fun> *)
string_of_char_list v;;

(*== 3. LISTES DE CARACTERES : format temporaire ==*)
(*-- 3.1. Listes: Expansion du format temporaire de répétitions ----*)
let lire_nombre q = lire 0 q                                              A ECRIRE (4/5 lignes)
   where rec lire i = function                     A ECRIRE (en 3 lignes)
;;
(* lire_nombre : char list -> int * char list = <fun> *)
lire_nombre (char_list_of_string "1664]def");;

let multiple i a = duplique i                                             A ECRIRE (3/4 lignes)
   where rec duplique = function                   A ECRIRE (en 2 lignes)
;;
(* multiple : int -> 'a -> 'a list = <fun> *)
multiple 10 `a`;;

let rec expand_temp = function                                            A ECRIRE (4 lignes)
;;
(* expand_temp : char list -> char list = <fun> *)
expand_temp (char_list_of_string "[2]a[16]b[3]5[1][xyz");;

(*-- 3.2. Listes: Compression au format temporaire de répétition ----*)
type Accumulateur = { mutable caractere : char; mutable compte : int }
;;
(*---------------------------------------*)
let char_list_of_int i =  reverse [] i                                    A ECRIRE (3/4 lignes)
  where rec reverse accu = function          A ECRIRE (2 lignes)
;;
(* char_list_of_int : int -> char list = <fun> *)
char_list_of_int 1664;;

let vide_accu_temp enc =                                                  A ECRIRE (5 lignes)
;;
(* vide_accu_temp : Accumulateur -> char list = <fun> *)

vide_accu_temp {caractere = `a`; compte = 5};;    vide_accu_temp {caractere = `a`; compte = 3};;
vide_accu_temp {caractere = `a`; compte = 0};;    vide_accu_temp {caractere = `[`; compte = 1};;
vide_accu_temp {caractere = `a`; compte = 1664};;

let compress_temp image =                                                 A ECRIRE (9 lignes)
  let enc = {caractere = `a`; compte = 0} in
  comprime image
  where rec comprime = function   A ECRIRE (6 lignes)
;;
(* compress_temp : char list -> char list = <fun> *)

let s = "aaabbbbbbbbccddddddde66666g[xyz";;
string_of_char_list (compress_temp (char_list_of_string s) );;
```

```
(*== 4. LISTES DE CARACTERES : répétitions sur 1 octets ==*)
(*-- 4.1. Listes: Compression avec répétitions sur 1 octet -----*)

let indic = '[';;   (* indicateur de répétition *)

let vide_accu enc =                                                    A ECRIRE (8 lignes)
   let c = enc.caractere and i = enc.compte in
   enc.compte <- 0;
   vide i
   where rec vide = function
     | i when i <= 0                 -> A ECRIRE (1 ligne)
     | i when (i < 4) & (c <> indic) -> A ECRIRE (1 ligne)
     | i when (i <= 255)             -> A ECRIRE (1 ligne)
     | i                             -> A ECRIRE (1 ligne)
;;
(* vide_accu : Accumulateur -> char list = <fun> *)

vide_accu {caractere = 'a'; compte = 5};;    vide_accu {caractere = 'a'; compte = 3};;
vide_accu {caractere = 'a'; compte = 0};;    vide_accu {caractere = indic; compte = 1};;
vide_accu {caractere = 'a'; compte = 1664};;

let codage image =                                                     A ECRIRE (9 lignes)
  let enc = {caractere = 'a'; compte = 0} in
  coder image
  where rec coder = function      A ECRIRE (6 lignes)
;;
(* codage : char list -> char list = <fun> *)

let u = expand_temp (char_list_of_string "[2]a[513]b[3]5[1][xyz");;
codage u;;

(*-- 4.2. Listes: Décompression des répétitions sur 1 octet -----*)
let rec decodage = function                                            A ECRIRE (3 lignes)
;;
(* decodage : char list -> char list = <fun> *)

let s = "[2]a[513]b[3]5[1][xyz";;
let u = expand_temp (char_list_of_string s);; let v = codage u;;
let w = decodage v;;                          let x = string_of_char_list (compress_temp w);;

(*== 5. FICHIERS DE CARACTERES : répétitions sur 1 octets ====*)
(*--- 5.1. Illustration par des exemples élémentaires ----*)
let rec lire i =
   try let a = input_char i in a :: (lire i)
   with End_of_file -> []
;;
(* lire : in_channel -> char list = <fun> *)

let ch_in = dir_name ^ "/test-1.txt";;
let Myfile_in = open_in ch_in;;
string_of_char_list (lire Myfile_in);;
close_in Myfile_in;;

(*--------------*)
let rec ecrire o = function
    | []   -> ()
    | a::q -> output_char o a; ecrire o q
;;
(* ecrire : out_channel -> char list -> unit = <fun> *)

let ch_out = dir_name ^ "/test-2.txt";;
let Myfile_out = open_out ch_out;;
let s = "Et maintenant, ceci est le contenu du fichier test-2.txt.";;
let u = char_list_of_string s;;
ecrire Myfile_out u;;
close_out Myfile_out;;
```

```
(*-- 5.2. CODAGE d'un fichier (binaire) vers un autre ---*)
let coder_fichier ch_in ch_out =
  let file_in = open_in_bin ch_in and file_out = open_out_bin ch_out   (* bin *)
  in
  let rec vider = function      (* écrit la liste fournie par (vide_accu enc) *)
    | []   -> ()
    | a::q -> output_char file_out a; vider q
  in
  let enc = {caractere = 'a'; compte = 0}
  and fini = ref false in
  while not !fini
  do try let a = input_char file_in in
           if a = enc.caractere
           then enc.compte <- 1 + enc.compte
           else ( vider (vide_accu enc);
                  enc.caractere <- a ; enc.compte <- 1
                )
       with End_of_file -> (vider (vide_accu enc) ; fini := true)
  done;
  close_in file_in; close_out file_out
;;
(* coder_fichier : string -> string -> unit = <fun> *)

(*-- 5.3. DECODAGE d'un fichier (binaire) vers un autre ---*)
let decoder_fichier ch_in ch_out =
  let file_in = open_in_bin ch_in and file_out = open_out_bin ch_out   (* bin *)
  in
  let fini = ref false in
  while not !fini
  do try let a = input_char file_in in
         if a = '['
         then let i = int_of_char (input_char file_in) in
              let c = input_char file_in in
              for k = 1 to i do output_char file_out c done
         else output_char file_out a
     with End_of_file -> fini := true
  done;
  close_in file_in; close_out file_out
;;
(* decoder_fichier : string -> string -> unit = <fun> *)

(*-- 5.3. TESTS : compression de fichiers .bmp 8 -----*)
(*-- 5.3.1. TEST élémentaire  ----*)
let ch_file_1 = dir_name ^ "/file-1.txt";;  let ch_file_2 = dir_name ^ "/file-2.txtc";;
let ch_file_3 = dir_name ^ "/file-3.txt";;

let u1 = expand_temp (char_list_of_string "[2]a[513]b[3]5[1][xyz" );;
let Myfile_out = open_out ch_file_1;;  ecrire Myfile_out u1;;  close_out Myfile_out;;

coder_fichier ch_file_1 ch_file_2;;
let Myfile_in = open_in ch_file_2;; let v2 = lire Myfile_in;; close_in Myfile_in;;

decoder_fichier ch_file_2 ch_file_3;;
let Myfile_in = open_in ch_file_3;;
let s3 = string_of_char_list (compress_temp (lire Myfile_in));;
close_in Myfile_in;;

(*-- 5.3.2. BIG TEST sur BIG fichier .BMP 8 (256 couleurs) ------*)
let ch_file_1 = dir_name ^ "/Image-1.bmp";;  let ch_file_2 = dir_name ^ "/Image-2.bmpc";;
let ch_file_3 = dir_name ^ "/Image-3.bmp";;
coder_fichier ch_file_1 ch_file_2;;
decoder_fichier ch_file_2 ch_file_3;;

(*=== Fin (squelette), Cas des fichiers .bmp > 8 : non reporté ==*)
```

< $\mathscr{FIN}$ (énoncé + squelette) >

8 Corrigé (TP BitMap)

```
(*============================================================*)
(* TP Info MP : Compression image bitmap 8 bits (256 couleurs) *)
(*                              CORRIGE                       *)
(* Antoine MOTEAU -- Lycée Pothier - Orléans --               *)
(*============================================================*)
(* Directory : du programme et des fichiers exemples *)
let dir_name = "h:/InfoMP/TP-info/Bitmap";;               (* A ADAPTER ! *)

(*-- 2.2. Ressources complémentaires -------------*)
let char_list_of_string s =
  let l = ref [] in for i = string_length s - 1 downto 0 do l := s.[i]::!l done; !l
;;
(* char_list_of_string : string -> char list = <fun> *)
let v = char_list_of_string "abuuuuuucdeffffghiiiiiijkllllllllllmnooop";;

let rec string_of_char_list = function
    | []   -> ""
    | a::q -> (char_for_read a) ^ (string_of_char_list q)
;;
(* string_of_char_list : char list -> string = <fun> *)
string_of_char_list v;;

(*== 3. LISTES DE CARACTERES : format temporaire ==*)
(*-- 3.1. Listes: Expansion du format temporaire de répétitions ----*)
let lire_nombre q = lire 0 q
   where rec lire i = function
     | []               -> failwith "erreur"
     | a::q when a = ']' -> (i, q)
     | a::q              -> lire (i * 10 + (int_of_char a - int_of_char '0')) q
;;
(* lire_nombre : char list -> int * char list = <fun> *)

lire_nombre (char_list_of_string "1664]def");;

let multiple i a = duplique i
  where rec duplique = function
      0 -> []
    | i -> a::(duplique (i-1) )
;;
(* multiple : int -> 'a -> 'a list = <fun> *)

multiple 10 'a';;

let rec expand_temp = function
     []                -> []
   | a::q when a = '[' -> let i,r = lire_nombre q
                          in (multiple i (hd r) ) @ (expand_temp (tl r))
   | a::q              -> a::(expand_temp q)
;;
(* expand_temp : char list -> char list = <fun> *)

expand_temp (char_list_of_string "[2]a[16]b[3]5[1][xyz");;

(*-- 3.2. Listes: Compression au format temporaire de répétition ----*)
type Accumulateur = { mutable caractere : char;  mutable compte : int }
;;
let char_list_of_int i = reverse [] i
  where rec reverse accu = function
     0 -> accu
   | i -> reverse (char_of_int ((i mod 10) + int_of_char '0') :: accu) (i/10)
;;
(* char_list_of_int : int -> char list = <fun> *)

char_list_of_int 1664;;
```

```
let vide_accu_temp enc =
   let c = enc.caractere and i = enc.compte in
   enc.compte <- 0;
   if ( c = '[') or (i > 4) then ( '[' :: (char_list_of_int i) ) @ ( ']' :: [c] )
   else multiple i c
;;
(* vide_accu_temp : Accumulateur -> char list = <fun> *)

vide_accu_temp {caractere = 'a'; compte = 5};;    vide_accu_temp {caractere = 'a'; compte = 4};;
vide_accu_temp {caractere = 'a'; compte = 0};;    vide_accu_temp {caractere = '['; compte = 1};;
vide_accu_temp {caractere = 'a'; compte = 1664};;

let compress_temp image =
  let enc = {caractere = 'a'; compte = 0} in
  comprime image
  where rec comprime = function
    | []   -> vide_accu_temp enc
    | a::q -> if a = enc.caractere
              then (enc.compte <- 1 + enc.compte ; comprime q)
              else let u = vide_accu_temp enc in
                   enc.caractere <- a ; enc.compte <- 1;
                   u @ comprime q
;;
(* compress_temp : char list -> char list = <fun> *)

let ch = "aa[8]bc[1664]de[5]6g[1][xyz";;
let u = expand_temp (char_list_of_string ch);;
string_of_char_list (compress_temp u);;

(*== 4. LISTES DE CARACTERES : répétitions sur 1 octets ==*)
(*-- 4.1. Listes: Compression avec répétitions sur 1 octet -----*)
let indic = '[';;    (* indicateur de répétition *)

let vide_accu enc =
   let c = enc.caractere and i = enc.compte in
   enc.compte <- 0;
   vide i
   where rec vide = function
     | i when i <= 0                -> []
     | i when (i < 4) & (c <> indic) -> multiple i c
     | i when (i <= 255)             -> indic :: (char_of_int i)   :: [c]
     | i                             -> indic :: (char_of_int 255) :: c :: (vide (i-255))
;;
(* vide_accu : Accumulateur -> char list = <fun> *)

vide_accu {caractere = 'a'; compte = 5};;    vide_accu {caractere = 'a'; compte = 3};;
vide_accu {caractere = 'a'; compte = 0};;    vide_accu {caractere = indic; compte = 1};;
vide_accu {caractere = 'a'; compte = 1664};;

let codage image =
  let enc = {caractere = 'a'; compte = 0} in
  coder image
  where rec coder = function
    | []   -> vide_accu enc
    | a::q -> if a = enc.caractere
              then (enc.compte <- 1 + enc.compte ; coder q)
              else let u = vide_accu enc in
                   enc.caractere <- a ; enc.compte <- 1;
                   u @ coder q
;;
(* codage : char list -> char list = <fun> *)

let u = expand_temp (char_list_of_string "[2]a[513]b[3]5[1][xyz");;
codage u;;


(*-- 4.2. Listes: Décompression des répétitions sur 1 octet -----*)
```

```
let rec decodage = function
     []                      -> []
   | a::o::c::q when a = '[' -> (multiple (int_of_char o) c) @ (decodage q)
   | a::q                    -> a::(decodage q)
;;
(* decodage : char list -> char list = <fun> *)

let s = "[2]a[513]b[3]5[1][xyz";;
let u = expand_temp (char_list_of_string s);;
let v = codage u;;
let w = decodage v;;
let x = string_of_char_list (compress_temp w);;

(*== 5. FICHIERS DE CARACTERES : répétitions sur 1 octets ====*)
(* Aide Caml : voir document du TP *)

(*--- 5.1. Illustration par des exemples élémentaires ----*)

let rec lire i =
   try let a = input_char i in a :: (lire i)
   with End_of_file -> []
;;
(* lire : in_channel -> char list = <fun> *)

let ch_in = dir_name ^ "/test-1.txt";;
let Myfile_in = open_in ch_in;; string_of_char_list (lire Myfile_in);; close_in Myfile_in;;

(*--------------*)
let rec ecrire o = function
    | []   -> ()
    | a::q -> output_char o a; ecrire o q
;;
(* ecrire : out_channel -> char list -> unit = <fun> *)

let ch_out = dir_name ^ "/test-2.txt";;
let Myfile_out = open_out ch_out;;
let s = "Et maintenant, ceci est le contenu du fichier test-2.txt.";;
let u = char_list_of_string s;; ecrire Myfile_out u;;  close_out Myfile_out;;

(*-- 5.2. CODAGE d'un fichier (binaire) vers un autre ---*)

let coder_fichier ch_in ch_out =
  let file_in = open_in_bin ch_in       (* bin *)
  and file_out = open_out_bin ch_out in   (* bin *)
  let rec vider = function        (* écrit la liste fournie par (vide_accu enc) *)
    | []   -> ()
    | a::q -> output_char file_out a; vider q
  in

  let enc = {caractere = 'a'; compte = 0}
  and fini = ref false in
  while not !fini
  do try let a = input_char file_in in
           if a = enc.caractere
           then enc.compte <- 1 + enc.compte
           else ( vider (vide_accu enc);
                  enc.caractere <- a ; enc.compte <- 1
                )
       with End_of_file -> (vider (vide_accu enc) ; fini := true)
  done;
  close_in file_in; close_out file_out
;;
(* coder_fichier : string -> string -> unit = <fun> *)


(*-- 5.3. DECODAGE d'un fichier (binaire) vers un autre ---*)
```

```
let decoder_fichier ch_in ch_out =
  let file_in = open_in_bin ch_in               (* bin *)
  and file_out = open_out_bin ch_out in         (* bin *)
  let fini = ref false in
  while not !fini
  do try let a = input_char file_in in
         if a = '['
         then let i = int_of_char (input_char file_in) in
              let c = input_char file_in in
              for k = 1 to i do output_char file_out c done
         else output_char file_out a
     with End_of_file -> fini := true
  done;
  close_in file_in; close_out file_out
;;
(* decoder_fichier : string -> string -> unit = <fun> *)

(*-- 5.3. TESTS : compression de fichiers .bmp 8 -----*)
(*-- 5.3.1. TEST élémentaire  ----*)

let ch_file_1 = dir_name ^ "/file-1.txt";;  let ch_file_2 = dir_name ^ "/file-2.txtc";;
let ch_file_3 = dir_name ^ "/file-3.txt";;

let s = "[2]a[513]b[3]5[1][xyz";;
let u1 = expand_temp (char_list_of_string s );;
let Myfile_out = open_out ch_file_1;;  ecrire Myfile_out u1;;  close_out Myfile_out;;

coder_fichier ch_file_1 ch_file_2;;

let Myfile_in = open_in ch_file_2;;   let v2 = lire Myfile_in;;  close_in Myfile_in;;

decoder_fichier ch_file_2 ch_file_3;;

let Myfile_in = open_in ch_file_3;;
let s3 = string_of_char_list (compress_temp (lire Myfile_in));;
close_in Myfile_in;;

(*-- 5.3.2. BIG TEST sur BIG fichier .BMP 8 (256 couleurs) ------*)

let ch_file_1 = dir_name ^ "/Image1-1.bmp";;  let ch_file_2 = dir_name ^ "/Image1-2.bmpc";;
let ch_file_3 = dir_name ^ "/Image1-3.bmp";;

coder_fichier ch_file_1 ch_file_2;;
decoder_fichier ch_file_2 ch_file_3;;

(*== 6. Cas des fichiers .bmp > 8 : non reporté (en intégralité dans l'énoncé) ==*)

(*=== Fin (corrigé) ===*)
```

$<$ $\mathscr{FIN}$ (énoncé + squelette + corrigé) $>$