Projet d'Algorithmique et Programmation
AP3
—Licence Informatique, 3ième année, 1ier semestre—

Ce projet est à réaliser à deux ou trois. Il est à rendre au plus tard le 04 février par le dépôt du fichier `img_bw.ml` à l'emplacement prévu sur UPdago et correctement renseigné avec le numéro du groupe et les noms des membres des groupes.

Trois fichiers vous sont fournis sur UPedago. Le fichier `img_bw.ml` contient les fonctions qui vous sont fournies. Vous **devez** répondre aux questions en insérant votre code dans ce fichier aux endroits prévus. Les deux autres fichiers `portrait.pbm` et `avion.pbm` sont des exemples d'images.

Dans cette partie, on va représenter les images en noir et blanc avec des arbres. Cette représentation permet de compresser les images et facilite les opérations de zoom et de rotation d'un quart de tour (rotations de $\pm 90°$).

Pour simplifier le problème, les image sont supposées carrées et de coté 2^n pixels. L'idée, illustrée sur la figure 1, est la suivante : pour une image de taille 2^n, si l'image est monochrome alors c'est une feuille d'un arbre ; sinon l'image est composée de quatre images de taille 2^{n-1}.

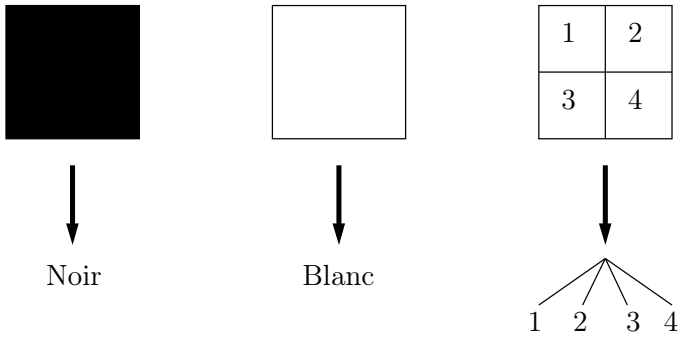


FIGURE 1 – Représentation d'une image noir et blanc sous la forme d'un arbre.

Pour représenter les couleurs, on se donne le type `couleur` composé des deux constructeurs constants `Noir` et `Blanc`. Les images noir et blanc sont représentées à l'aide du type `picture` qui est un tableau à deux dimension d'éléments du type `couleur`. Enfin les images sous forme d'arbres sont représentées par le type `arbre`.

```
type couleur = Noir | Blanc ;;

type picture = couleur array array ;;
```

```
type arbre = Feuille of couleur
           | Noeud of arbre * arbre * arbre * arbre ;;
```

Le type `array` est le type des tableaux en `OCaml`. Quelques explications sur ce type vous sont fournies dans l'annexe technique. Pour plus de détails, vous pouvez toujours vous reporter à la documentation du module `Array` de la bibliothèque standard d'`OCaml`.

▸ **Question 1.** *Écrivez une fonction* `is_puiss_2` *qui teste si un entier est une puissance de* 2.

▸ **Question 2.** *Quels sont les arbres qui correspondent aux deux images ci-dessous ?*

```
let img_test = [|
  [| Blanc; Noir; Blanc; Blanc |];
  [| Noir; Blanc; Blanc; Blanc |];
  [| Noir; Noir; Blanc; Noir |];
  [| Noir; Noir; Noir; Blanc |]
|] ;;
```

```
let img_test1 = [|
  [| Noir; Noir; Blanc; Blanc |];
  [| Noir; Noir; Blanc; Blanc |];
  [| Noir; Noir; Noir; Noir |];
  [| Noir; Noir; Noir; Noir |]
|] ;;
```

▸ **Question 3.** *À l'aide de la fonction* `forloop` *(voir l'annexe), écrivez une fonction* `random_img` *qui prend en argument une taille d'image (i.e. une puissance de* 2*) et un nombre* n *de pixels noirs, et qui renvoie une image du type* `picture` *avec les* n *pixels noirs placés aléatoirement. Utilisez la fonction* `draw_picture` *pour visualiser le résultat.*

▸ **Question 4.** ***Conversions entre les images et les arbres***

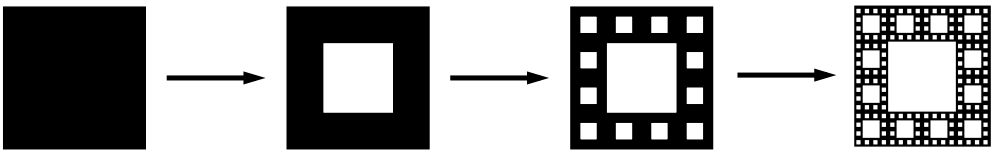
1. ***Arbre correspondant à une image.*** *Écrire une fonction* `image_vers_arbre` *qui prend un entier* k *et une image de taille* $k \times k$ *(on rappelle que* k *est de la forme* 2^n*) et qui renvoie l'arbre représentant cette image.*
2. ***Image correspondant à un arbre.*** *Écrire une fonction* `arbre_vers_image` *qui prend un arbre représentant une image de taille* $k \times k$ *et qui renvoie l'image représentée par cet arbre. Pour écrire cette fonction, on commencera par écrire une fonction* `remplir_carre` *qui prend en argument une image, une taille* k*, deux indices* i *et* j *et une couleur* c *et qui renvoie l'image remplie d'un carré de couleur* c *dont le coin inférieur gauche est indiqué par* i *et* j *et dont le coté est de longueur* k*.*

▸ **Question 5.** ***Dessin d'un arbre.*** *Écrivez une fonction* `draw_tree` *qui dessine une image à partir de sa représentation sous forme d'arbre. Vous utiliserez la fonction* `fill_rect` *pour dessiner une feuille et la fonction* `resize_window` *pour vous assurer que la fenêtre graphique soit de la bonne taille.*

▸ **Question 6.** ***Manipulation d'images sous forme d'arbres.*** *On veut maintenant manipuler les images directement en utilisant leur représentation sous forme d'arbre. Pour cela, on introduit un type* `type img_arb` *dont les valeurs contiennent un arbre qui représente une image et un entier qui est la taille de l'image.*

1. **Type** *Définissez le type* `img_arb`
2. **Agrandissement.** *Comment peut-on faire pour agrandir une image représentée sous la forme d'un arbre ? Programmez des exemples.*
3. **Rotation de** $90°$**.** *Écrivez une fonction* `rotation` *qui effectue une rotation d'une image représentée sous forme d'arbre de* $90°$ *sur la droite.*

▸ **Question 7.** ***Application : dessin d'une fractale.*** *Écrivez une fonction* `fractale` *qui prend deux entiers* k *et* n *et qui construit l'arbre correspondant à l'image de taille* $k \times k$ *obtenue à par* n *itérations du processus suivant :*



▸ **Question 8.** ***Enregistrement et chargement d'images.*** *Afin d'écrire et de lire les images manipulées dans des fichiers, on veut transformer une image sous forme d'arbre en une liste de bits (i.e. de* 0 *et de* 1*) puis d'octets (i.e. une liste de* 8 *bits). On veut aussi pouvoir transformer une liste d'octets en un arbre représentant une image.*

Pour cela, on code un arbre de la façon suivante :

$$\begin{aligned} code(\texttt{Feuille Blanc}) &= 00 \\ code(\texttt{Feuille Noir}) &= 01 \\ code(\texttt{Noeud (a1, a2, a3, a4)}) &= 1 code(\texttt{a1}) code(\texttt{a2}) code(\texttt{a3}) code(\texttt{a4}) \end{aligned}$$

1. *Écrivez une fonction* `arbre_vers_bits`*, qui transforme un arbre en une liste de* 0 *et de* 1 *selon le codage choisi.*
2. *Écrivez une fonction* `bits_vers_octets`*, qui prend une liste de* 0 *et de* 1 *comme précédemment, puis retourne une liste d'entiers compris entre* 0 *et* 256*. Chaque entier de cette nouvelle liste correspondra au codage en base* 10 *des sous listes de* 8 *bits (des octets) de la liste initiale. Si la longueur de la liste n'est pas un multiple de* 8*, on peut compléter avec des* 0 *pour le dernier octet.*
3. *Écrire les fonctions réciproques* `bits_vers_arbre` *et* `octet_vers_bit`*. Comment traitez vous les erreurs dans la fonction* `bits_vers_arbre` *(si la liste ne représente pas un arbre) ? Les* 0 *éventuellement ajoutés en fin de liste, dans la fonction* `bits_vers_octets`*, ne doivent pas être considérés comme des erreurs.*
4. *À l'aide des fonctions précédentes et des fonctions de lecture (*`open_in_bin`*,* `close_in` *et* `input_byte`*) et d'écriture (*`open_out_bin`*,* `close_out` *et* `output_byte`*) dans des fichiers binaires, écrivez les fonctions d'écriture* `write_arbre` *et de lecture* `read_arbre` *d'un arbre dans un fichier. Consultez la documentation du module* `Pervasive` *ou* `Stdlib` *pour les versions récentes de*`OCaml`*, pour connaître le fonctionnement de ces fonctions.*

Annexes

Le module `Graphics`

- ✖ Pour charger le module `Graphics` utilisez la commande `#load "graphics.cma"` et tapez ensuite `open Graphics`. Cela vous permettra de ne pas écrire `Graphics.` devant chaque fonction du module.
  ATTENTION : Si vous utilisez une version récente d'`OCaml` (i.e >= 4.09) le module `Graphics` n'est plus intégré dans la distribution d'`OCaml`. Il faut installer le module avec `opam` (`opam install graphics`) et utiliser la commande `#require "graphics" ;;`. Si vous avez des difficultés, contactez-moi (Il peut y avoir des cas particuliers selon votre installation d'`OCaml`).
- ✖ Pour ouvrir une fenêtre de dessin, utilisez `open_graph ""`. Vous pouvez préciser la taille de la fenêtre en remplaçant la chaîne de caractères vide (`""`) par `" 200x300"` ou une autre taille qui vous convient (attention : l'espace au début de la chaîne de caractères est obligatoire).
- ✖ Pour effacer la fenêtre de dessin utilisez `clear_graph ()`, et pour fermer la fenêtre de dessin, utilisez `close_graph ()`.
- ✖ La documentation du module `Graphics` se trouve
  — `https://ocaml.org/releases/4.08/htmlman/libref/Graphics.html` pour les versions <= 4.08
  — `https://ocaml.github.io/graphics/graphics/Graphics/index.html` pour les versions >= 4.09

Un petit point technique sur `OCaml`. Lorsque vous utiliserez les fonctions `open_graph`, `close_graph` et `clear_graph`, vous remarquerez que le type de leur valeur de retour est `unit`. Ce type ne contient qu'une valeur qui est `()`. Ainsi le type de `open_graph` est `string -> unit` et celui de `close_graph` est `unit -> unit`. Le type `unit` permet d'indiquer qu'il y a un *effet de bord* et correspond par exemple au `void` du langage `C`.

À propos des boucles

L'utilisation de fonctions avec effets de bords et de structures de données comme les tableaux (cf. ci-dessous) nous amène « naturellement » à vouloir programmer avec des boucles.

L'objectif de cette question est de vous montrer que les boucles `while` et `for` sont ds fonctions récursives terminales et peuvent donc se programmer comme telles. On peut ensuite les utiliser pour modifier des structures de données mutables tout comme les boucles programmées dans le langage (quel qu'il soit).

En plus de la boucle `while` que l'on a vue en cours implantée par la fonction `whileloop`, vous trouverez ci-dessous une implantation de la boucle `for`.

```
let rec forloop(r, n, next : 'a * int* ('a -> 'a)) : 'a =
  if n = 0 then r
  else forloop (next(r), n-1, next)
;;
```

Pour l'utiliser, on donne n le nombre d'itérations, *next* la fonction à itérer et r la valeur initiale. Généralement l'argument `r` est un couple `(i, res)` où `i` est la variable de boucle que l'on incrémente à chaque tour de boucle et `res` est la variable qui contient le résultat à chaque tour de boucle et à la fin de l'itération. Ci-dessous un exemple d'utilisation :

```
(forloop((1, 1),
         5,
         (function (i, k) -> print_int i;
                             print_newline ();
                             (i+1, i*k)
         )
   )
)
;;
```

Dans cet exemple, on utilise une séquence d'expressions séparées par des points-virgules, et éventuellement encadrées par des parenthèses. Le type d'une séquence d'expression est le type de sa dernière expression. Les résultats et les types des expressions intermédiaires ne sont pas pris en compte. Cependant, il est préférable qu'elles soient de type `unit` (i.e. on utilise des fonctions à effets de bords). Si ce n'est pas le cas `OCaml` vous l'indiquera par un message.

Quand on utilise des fonction à effet de bord la valeur de `res` est généralement `()` comme dans l'exemple suivant [(1)]

```
let affiche_compteurs(n, m : int * int) : unit =
  snd (forloop((0, ()),
               n,
               (function (i,()) ->
                  snd (forloop((0, ()),
                               m,
                               (function (j, ()) -> print_int i;
                                                    print_string " ";
                                                    print_int j;
                                                    print_newline (); (j+1, ()))));
                  (i+1, ()))
    )
;;
```

La fonction `draw_picture` qui vous est fournie pour le projet est un autre exemple d'utilisation de la boucle `for`.

À propos des tableaux du module `Array`

Le module `Array` implante la structure de données des tableaux. C'est une structure de données « mutable » (i.e. on peut modifier un élément comme dans les langages impératifs). Pour construire un tableau à deux entrées, on utilise la fonction `Array.make_matrix` qui prend en argument les dimensions du tableau et la valeur initiale des cases.

(1). Si on ne fait pas comme ça, cela fonctionne aussi mais `OCaml` rouspète un peu (à juste titre).

Pour accéder à un élément d'un tableau `tab` à deux entrées, on utilise la notation `tab.(i).(j)`. Pour modifier une case, on utilise `tab.(i).(j) <- val` où `val` est la nouvelle valeur de la case. La fonction `<-` est une fonction à effet de bord.

Les fonctions `draw_picture` et `read_pbm` qui vous sont fournies pour le projet sont des exemples d'utilisation des tableaux.

À propos des fichiers d'images

Deux fichiers d'image au format `pbm` ascii vous sont fournis pour le projet. La fonction `read_pbm` permet de lire ces fichiers et de d'obtenir une image de type `picture` que vous pouvez ensuite utiliser pour tester les fonctions de votre projet.

Le format `pbm` ascii est un format de fichier pour les images en noir et blanc. Il est très simple :

- la première ligne contient l'identifiant `P1` qui indique qu'il s'agit d'un fichier `pbm` ascii ;
- les lignes suivantes sont soit vides, soit elles commencent par le caractère `#` pour indiquer un commentaire ;
- vient ensuite une ligne avec la largeur, un caractère espace, puis la hauteur ;
- enfin, les lignes suivantes de 0 et de 1, éventuellement de longueurs différentes, composent l'image.

Pour convertir une image d'un format quelconque au format `pbm`, on utilise la commande `convert` dans un interpréteur de commande :

```
convert -resize 256x256! mon_image.xx mon_image.pbm
```

où `xx` est une extension d'un format d'image, par exemple, `jpg`, `tiff`, `png`, `bmp` et l'option `-resize 256x256!` indique qu'on veut une image de 256 pixels par 256 pixels.

Il faut ensuite convertir le fichier `pbm` obtenu en un fichier `pbm` en format ascii :

```
pnmtoplainpnm mon_image.pbm > mon_image_ascii.pbm
```

La commande `convert` fait partie de la bibliothèque `ImageMagick`, voir le site internet `http://www.imagemagick.org`.

La commande `pnmtoplainpnm` fait partie de la bibliothèque `Netpbm`, voir le site internet `http://netpbm.sourceforge.net`.

Ces deux ensembles de programmes fournissent des outils très puissants de manipulation d'images.

Si la manipulation d'images avec `OCaml` vous intéresse, vous pouvez consulter le site internet `http://gallium.inria.fr/camlimages/`.