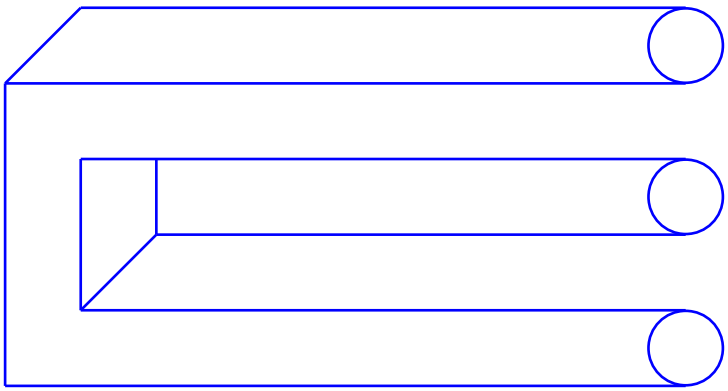
APF : Algorithmique et Programmation Fonctionnelle

Livret de TP



Jean-François Monin, Laurent Mounier et Benjamin Wack

Avant-propos

Ce document contient l'ensemble des énoncés des séances de TP proposés par l'équipe pédagogique d'APF pour l'année 2014-2015. Nous indiquons par des étoiles la difficulté des exercices proposés : plus il y a d'étoiles plus l'exercice est jugé difficile.

Plan : Nous commencerons par des exercices basiques permettant de s'approprier la syntaxe d'OCaml. Ensuite nous verrons comment écrire un `Makefile` et utiliser le compilateur d'OCaml. Nous verrons les principaux algorithmes de tris en programmation fonctionnelle. Nous parlerons des structures de données usuelles en informatique telles les files et les piles. Nous utiliserons aussi les modules et les foncteurs d'OCaml. Ensuite nous dessinerons des objets fractals et regarderons le format d'image PGM. Enfin nous utiliserons des arbres afin de trouver une stratégie gagnante dans le jeu de la gaufre empoisonnée. La fin du semestre sera consacrée à la réalisation d'un projet simulant l'évolution de chats et de souris dans une maison.

Préparation : Avant chaque séance de TD, des jeux de tests de vos différentes fonctions devront être réalisés et envoyés par email à votre enseignant de TP.

Pour certains TP, des fichiers sont à télécharger depuis l'URL :
`http://www-verimag.imag.fr/~wack/APF_2014_2015/`

La documentation complète de la bibliothèque standard est accessible par l'URL :
`http://caml.inria.fr/pub/docs/manual-ocaml/libref/`

Table des matières

1 Introduction à OCaml

Objectifs : Il s'agit dans ce premier TP de se familiariser avec l'environnement de développement Emacs couplé au mode Tuareg et de s'interroger sur les différents paradigmes du langage Caml à travers des exercices « jouets ». Il s'agit aussi de mettre en application différents algorithmes de manipulation de listes.

Toutes les constructions syntaxiques qui vous seront nécessaires ne sont pas données ou décrites en détail [1]. Il est **fortement conseillé** de tester vos fonctions avec des entrées judicieusement choisies.

1.1 Environnement de travail

Objective Caml est la principale implémentation du langage Caml. Pour travailler sur votre machine personnelle, veillez à ce que OCaml, Emacs, le mode Tuareg, Make et la bibliothèque LablGL soient installés sur votre système. Par exemple, dans les distributions Debian (et donc également Ubuntu), ils sont fournis par le méta-paquet `ocaml-core` (en suivant les recommandations et suggestions) et les paquets `make` (installé par défaut) et `liblablgl-ocaml-dev`.

Si votre machine n'est pas correctement configurée, vous pouvez vous connecter en ssh sur la machine Mandelbrot, sur laquelle est installée OCaml. Pour cela, tapez la commande suivante, en remplaçant `login` par votre login : `ssh -X login@mandelbrot2.e.ujf-grenoble.fr`

1.2 Personnalisation d'Emacs avec le mode Tuareg

Pour utiliser l'environnement de développement Emacs/OCaml, depuis la racine de votre compte, éditez (s'il existe, le créer sinon) le fichier `.emacs` et insérez le contenu du fichier `mode.tuareg` à récupérer sur le site web du cours.

Cette modification du fichier de configuration d'Emacs permet de charger automatiquement le mode Tuareg d'OCaml chaque fois que vous éditerez un fichier `.ml`.

1.3 Premiers pas avec OCaml

OCaml peut être utilisé en mode compilé ou en mode interactif.
- Le mode compilé permet de créer un fichier binaire exécutable. Nous en parlerons au TP7.
- Le mode interactif permet d'évaluer des expressions OCaml de manière incrémentale. C'est celui qui nous intéresse dans le cadre de ce premier TP. Nous l'utiliserons par l'intermédiaire de l'éditeur Emacs, grâce au mode Tuareg.

Tests pour vérifier que Tuareg fonctionne :

Utiliser la coloration syntaxique :

1. Ouvrir Emacs.
2. Ouvrir/créer un nouveau fichier `test.ml` (l'extension `.ml` indique à Emacs de lancer Tuareg).
3. Taper dans ce nouveau tampon vierge l'expression `let a = 1+1`. Chaque terme de cette expression apparaît dans une couleur différente.

1. Pour plus de précisions référez-vous à la documentation en ligne `http://caml.inria.fr` dans « Ressources » suivre « Manuel d'OCaml » puis « The core language »

Évaluer une expression :

1. Taper `C-c C-e` qui permet d'évaluer une expression dans la boucle interactive d'OCaml.
2. À la première utilisation de cette commande, Emacs demande dans la ligne de commande si on veut utiliser la boucle interactive standard (`ocaml`) ; confirmer en tapant entrée.
3. Une nouvelle fenêtre doit s'ouvrir et afficher le résultat de l'évaluation. Les résultats suivants seront toujours affichés dans ce nouveau tampon nommé `*caml-toplevel*`.

Note : pour stopper la boucle interactive, taper `C-c C-k`.

Indenter une expression sur plusieurs lignes :

1. Insérer un retour à la ligne entre `let a =` et `1+1`.
2. Taper `C-x h` pour sélectionner toutes les lignes.
3. Taper `C-M-\` pour indenter correctement la sélection.

Remarques sur Caml :
- Caml est *case-sensitive*, c'est à dire qu'il prend en compte les majuscules et les minuscules. En particulier, seuls les noms de constructeurs de types commencent par une majuscule alors que les autres noms commencent par une minuscule.
- Les commentaires sont placés entre `(*...*)`.

Quelques raccourcis Emacs utiles À utiliser sans modération. Il existe une abondante documentation sur internet [2].

– couper la région sélectionnée	`C-w`
– coller la région sélectionnée	`C-y`
– indenter une ligne	`Tab`
– indenter un paragraphe	`Esc Q`
– ouvrir un fichier	`C-x C-f`
– sauvegarder le buffer courant	`C-x C-s`
– quitter Emacs	`C-x C-c`
– changer de buffer	`C-x b + tab`
– aller en fin de ligne	`C-e`
– aller en début de ligne	`C-a`
– supprimer le buffer courant (n'affecte pas le fichier)	`C-x k`
– évaluer l'expression OCaml sous le curseur	`C-c C-e`
– évaluer l'ensemble des expressions OCaml de la région sélectionnée	`C-c C-r`
– évaluer l'ensemble des expressions OCaml du buffer	`C-c C-b`
– fermer l'interpréteur OCaml	`C-C C-k`
– commenter la région sélectionnée	`M-x comment-region`
– décommenter la région sélectionnée	`C-u M-x comment-region`

1.4 Découverte d'OCaml

En Caml, écrire un programme consiste à déclarer un ensemble d'expressions qui peuvent être des valeurs ou des fonctions. Comme le calcul correspond à l'évaluation des fonctions déclarées, on parle de programmation fonctionnelle.

Exercice 1 *Évaluer les expressions suivantes et donner le type du résultat :*

2. par exemple `http://refcards.com/docs/wingb/xemacs/xemacs-refcard-a4.pdf` ou encore `http://www.ocamlpro.com/files/tuareg-mode.pdf`

- `let r = let x = 7 in 6 * x`
- `let a = (r - 6) / 6 - 6`
- `let o = r * r - x * x - 51`
- `let u = let x = 9 in if (x < 9) then 9 / (x - x) else (x + x) / 9`
- `let l = let x,y = 3 + 4,6 in x * y`

Des fonctions Les fonctions peuvent être passées en argument et rendues en résultat comme tout autre élément. Cela permet en particulier de traiter les fonctions à plusieurs arguments comme des fonctions à un seul argument retournant une fonction. Cette dernière forme est dite forme *curryfiée*. Ainsi, on peut définir une fonction somme comme suit : `fun x -> fun y -> x + y`. Une fonction qui prend en argument une fonction, ou qui rend en résultat une fonction, est appelée fonction d'ordre supérieur.

Exercice 2 *Écrire sous forme curryfiée et tester une fonction* `f1` *qui calcule le produit de trois entiers et une fonction* `f2` *qui calcule leur somme.*

Du typage En CAML l'évaluateur, comme le compilateur, calcule le type de chaque expression et vérifie que celui-ci existe. Puisqu'il s'agit d'un langage fonctionnel, un type fréquent est celui des fonctions, que l'on écrit à l'aide de la flèche $t_1 \to t_2$ et qui signifie qu'une fonction ayant ce type calcule un élément de type t_2 à partir d'un élément de type t_1. Les types t_i peuvent être des types de base ou des types de fonctions.

Exercice 3 *Manipulons les types.*
- *Donner une fonction ayant le type int → int → int, et une autre ayant le type int → int → bool.*
- *Observer le type des fonctions* `f1` *et* `f2` *de l'exercice 2.*
- *Évaluer* `let f3 = fun x -> fun y -> x (y + 1) || false`.
  Expliquer ce que fait cette fonction et donner son type.
- *Évaluer* `let f4 = fun x -> fun y -> x (y + 1) + 1`.
  Expliquer ce que fait cette fonction et donner son type.
- *Évaluer* `let f5 = fun x -> x x` *et expliquer le résultat. (*)*

De l'évaluation partielle On peut évaluer certains paramètres d'une fonction et ainsi créer une instance spécialisée de celle-ci. Considérer la fonction suivante, qui incrémente une variable entière `x` de `i` : `f = fun i -> fun x -> x + i`. La fonction `g = f 2` est l'évaluation partielle de `f` pour `i` égal à 2. C'est la fonction à un argument qui ajoute 2 à ce dernier.

Exercice 4 *Observer le comportement des fonctions* `f` *et* `g`. *À partir de la fonction* `f1` *(respectivement* `f2`*) de l'exercice 2, construire la fonction* `g1` *(resp.* `g2`*) qui calcule le produit (resp. la somme) de deux entiers.*

1.5 Retour sur des exercices vus en TD

1.5.1 Types de base et fonctions

Exercice 5 *Prédicats.*
Écrire des fonctions qui déterminent :
- *si un entier est positif.*
- *si un entier est pair.*
- *si les trois paramètres entiers forment un triplet pythagoricien.*
- *si deux entiers sont de même signe.*

Exercice 6 *Minimum de deux entiers.*
Écrire une fonction `min2entiers` *qui calcule le minimum de deux entiers passés en paramètres.*

Exercice 7 *Maximum de deux entiers.*
Écrire une fonction `max2entiers` *qui calcule le maximum de deux entiers passés en paramètres.*

Exercice 8 *Min de trois entiers.*
Écrire une fonction `min3entiers` *qui calcule le minimum de trois entiers passés en paramètres.*

1.5.2 Types sommes

Exercice 9 *Jour de la semaine.*
– *Définir le type* `semaine` *qui représente chaque jour de la semaine.*
– *Écrire une fonction qui teste si un jour de la semaine est un jour du week-end.*

Exercice 10 *Aires.*
Écrire des fonctions qui calculent :
– *L'aire d'un carré de côté a.*
– *L'aire d'un rectangle de côtés a et b.*
– *L'aire d'un cercle de rayon r.*
– *L'aire d'un triangle rectangle de côté a et d'hypoténuse h (en utilisant le théorème de Pythagore).*

Exercice 11 *Figures géométriques.*
– *Définir un type somme pour représenter les figures géométriques carré, rectangle et cercle.*
– *Écrire une fonction aire permettant de calculer l'aire d'une figure géométrique.*

1.5.3 Types produits

Exercice 12 *Nombres complexes.*
– *Définir le type* `complexe`
– *Définir l'élément neutre pour l'addition des nombres complexes.*
– *Écrire une fonction qui additionne deux nombres complexes.*
– *Écrire une fonction qui donne le module d'un nombre complexe.*
– *Écrire une fonction qui donne l'opposé d'un nombre complexe.*

Exercice 13 *Point 2D.*

– *Définir un type* `point2D` *qui représente les points du plan.*
– *Écrire une fonction qui calcule la distance entre deux points.*
– *Définir un type* `segment`*.*
– *Écrire une fonction qui renvoie le milieu d'un segment.*
– *Définir un type* `vecteur`
– *Écrire une fonction qui renvoie le vecteur associé à deux points.*
– *Définir un type* `droite` *donné par un point de la droite et un vecteur directeur.*
– *Écrire une fonction qui calcule l'intersection de deux droites.*
– *Écrire une fonction qui calcule la droite perpendiculaire à une droite et passant par un point donné.*

1.6 Le robot jardinier (facultatif)

Exercice 14 (facultatif) *Récupérer sur*
`http://www-verimag.imag.fr/~wack/APF_2014_2015/`
la définition des types introduits en TD.
En utilisant ces types, écrivez la fonction `mouvoir` *demandée en TD.*

Exercice 15 (facultatif) *Définissez un exemple de terrain comportant un arrosoir, des robinets, et des végétaux sur certaines cases.*
Définissez alors une fonction `scenario` *permettant à un robot de prendre un arrosoir, le remplir et arroser un végétal.*
Complétez par d'autres scénarios de votre choix ...

2 Manipulation de listes

Objectifs : Mettre en application des algorithmes de manipulation de listes.

Définir des types pour modéliser.

2.0 Travail préliminaire : échauffement sur les listes

Exercice 16 *Écrire les fonctions suivantes, qui prennent toutes en argument une liste d'entiers :*
– `longueur` *renvoie le nombre d'entiers dans la liste*
– `somme` *renvoie la somme de tous les entiers de la liste*
– `tous_positifs` *renvoie un booléen exprimant si tous les entiers de la liste sont positifs ou non*

2.1 Les déménageurs Caml

Un appartement contenant plus d'une dizaine de paquets doit être vidé. Les déménageurs Caml font l'inventaire des paquets à transporter avant toute manipulation. Un paquet peut contenir un meuble, un objet, un cadre ou une plante. Certains paquets sont fragiles (ne peuvent supporter aucun poids) ou robustes (peuvent supporter plus de 20 kg), les autres paquets sont supposés pouvoir supporter jusqu'à 20 kg. Les déménageurs cherchent à savoir comment les empiler au mieux.

Exercice 17 *Définir les types* `contenu` *et* `solidite` *comme des types somme, grâce aux informations données ci-dessus. Définir le type* `paquet` *comme un type* produit *à trois composantes à l'aide des deux types précédents et du type* `int` *pour le poids.*

Exercice 18 *Écrire une fonction* `fragiles` *qui indique le nombre de paquets fragiles présents dans l'inventaire donné en paramètre.*

Exercice 19 *Écrire une fonction* `legers` *qui prend en paramètre un poids et l'inventaire et rend la liste des paquets pesant au plus ce poids.*

Exercice 20 *Écrire une fonction* `poids_plantes` *qui indique le poids total des plantes de l'inventaire donné en paramètre.*

Exercice 21 *Écrire une fonction* `exposition` *qui retire tous les cadres d'un inventaire.*

Exercice 22 *Écrire une fonction* `inventorie` *qui insère un paquet dans un inventaire par ordre croissant de poids. Nous supposons que l'inventaire est ordonné.*

Exercice 23 *Écrire une fonction* `dromadaire` *qui prend l'inventaire en paramètre et indique le paquet le plus lourd.*

> **Exercice 24 (facultatif)** *[*] Écrire une fonction* `chameau` *qui prend l'inventaire en paramètre et indique les deux plus lourds paquets.*

2.2 Chez Tardy

Marc est vendeur au magasin Tardy. Il est responsable du rayon électronique. Son rayon contient différents produits de différentes marques : il y a des lecteurs MP3, des appareils photo, des caméras numériques, des téléphones portables ainsi que des ordinateurs portables. Les marques vendues sont Alpel, Syno, Massung et Liphisp. Bien sûr, chaque appareil est caractérisé par un prix, en plus du nombre restant en stock.

Exercice 25 *Définir des types adaptés pour représenter la situation décrite ci-dessus.*

Si besoin, vous compléterez ces définitions au fur et à mesure des questions qui suivent.

Exercice 26 *Écrire une fonction* `est_en_stock` *qui indique si un élément est présent en stock (c'est-à-dire si le nombre d'articles disponibles est strictement positif). Elle prend en argument un produit, une marque et un prix (caractéristiques de l'élément) et la liste des articles répertoriés.*

Exercice 27 *Écrire une fonction* `ajoute_article` *qui ajoute un article dans la liste, en vérifiant qu'il n'y est pas déjà, et s'il y est déjà, modifie le nombre d'éléments en stock dans la liste en additionnant le nombre en stock de l'argument article. Elle prend en argument un article et la liste des articles répertoriés.*

Exercice 28 *Écrire une fonction* `enleve_article` *qui enlève un article de la liste. Elle prend en argument la liste des articles répertoriés et l'article à enlever.*

2.3 Aidons Marc : algorithmes de base (facultatif)

Aider un client

Exercice 29 (facultatif) *Écrire une fonction* `longueur` *qui calcule le nombre d'éléments d'une liste d'articles. Cette fonction prend comme argument une liste d'articles.*

Dans les 5 exercices suivants, on ne tient pas compte du fait qu'un produit soit en stock ou non (c'est-à-dire qu'on peut effectuer des réponses comportant des produits non présents en stock).

Exercice 30 (facultatif) *Écrire une fonction* `ces_produits` *qui prend en argument un produit et une liste d'articles et renvoie la liste des articles qui conviennent dans la liste d'articles (ex :* `ces_produits`*(MP3,L) renvoie tous les MP3 présents dans L).*

Exercice 31 (facultatif) *Écrire une fonction* `le_moins_cher` *qui renvoie l'élément d'une catégorie produit donnée le moins cher d'une liste d'articles. Elle prend en argument un produit, et la liste d'articles répertoriés.*

Exercice 32 (facultatif) ***(Le choix le plus courant)*** *Marc se rend compte que les clients choisissent généralement le deuxième produit le moins cher, i.e. si l'on classe tous les produits de la liste par ordre de prix croissant, le deuxième produit de cette liste. Sans classer tous les articles correspondant à un même produit par ordre croissant, écrivez une fonction* `deuxieme_moins_cher` *qui prend en argument une liste d'articles répertoriés, un produit, et renvoie le choix préféré des clients.*
Pouvez-vous écrire cette fonction en utilisant les fonctions précédentes ? Est-ce judicieux, et pourquoi ?

Exercice 33 (facultatif) *Écrire une fonction* `budget` *qui prend en argument deux entiers m, budget minimal, et M budget maximal, ainsi qu'une liste d'articles, et renvoie la liste des articles compris dans ce budget (i.e. dont le prix p est tel que $m \leq p \leq M$).*

Gestion des stocks

Exercice 34 (facultatif) *Écrire une fonction* `achete` *qui prend en argument une liste d'articles et les nom de produit, marque et prix pour un élément, et fait diminuer de 1 la quantité en stock d'un article.*

Exercice 35 (facultatif) *Écrire une fonction* `commande` *qui prend en argument la liste des articles et renvoie la liste des articles à commander au fournisseur (ceux dont le nombre en stock est nul).*

3 Trions

Objectifs : Coder des algorithmes de tri classiques.

3.0 Travail préliminaire : tests

Il est **fortement conseillé** de tester vos fonctions avec des entrées judicieusement choisies, et de respecter le prototype (signature de fonction) demandé. Vous utiliserez les fonctions suivantes pour tester toutes vos fonctions de tri.

Exercice 36 *Écrire une fonction* `est_trie` *de type* `'a list → bool` *qui teste si une liste est triée par ordre croissant. On rappelle que la liste vide est triée.*

Exercice 37 *Écrire des fonctions* `croissante`, `decroissante` *et* `aleatoire`, *toutes de type* `int → int list` *qui génèrent des listes d'entiers respectivement déjà triées en ordre croissant, en ordre décroissant, ou « aléatoires ».*
(voir `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Random.html` *)*

3.1 Tri sélection

Le principe du tri sélection est de trouver le minimum de la liste, de le mettre à sa place, puis de recommencer.

Exercice 38 *Écrire une fonction* `trouve_min` *de type* `'a list → 'a * 'a list` *qui calcule le minimum d'une liste non vide ainsi que la liste privée de son minimum.*

Exercice 39 *Écrire une fonction* `tri_selection` *de type* `'a list → 'a list` *qui trie la liste donnée en entrée en suivant l'algorithme du tri par sélection.*

3.2 Tri bulles (facultatif)

Le principe du tri à bulles est de parcourir la liste en échangeant lors du parcours deux éléments consécutifs s'ils sont rangés dans le mauvais ordre. Lorsqu'il n'y a plus d'échange lors d'un parcours, c'est que la liste est triée.

Par exemple lors d'un premier passage, on pourrait modifier la liste initiale `5 4 3 2 1` de la façon suivante :

État initial :	**5**	**4**	3	2	1
	4	**5**	**3**	2	1
	4	3	**5**	**2**	1
	4	3	2	**5**	**1**
État final :	4	3	2	1	5

Exercice 40 (facultatif) *Écrire une fonction* `bulles` *de type* `'a list → 'a list * bool` *qui effectue un parcours avec échange comme décrit précédemment. Votre fonction devra de plus renvoyer un booléen indiquant si la liste a été modifiée.*

Exercice 41 (facultatif) *Écrire une fonction* `tri_bulles` *de type* `'a list → 'a list` *qui trie la liste donnée en entrée en suivant l'algorithme du tri à bulles.*

3.3 Tri insertion

Le principe du tri insertion est d'insérer à leur place les éléments de la liste à trier dans une liste initialement vide. À la fin cette liste est la liste initiale triée.

Par exemple sur l'entrée `1 5 4 6 9 7` on aura les étapes suivantes :

État initial (vide)						
Insertion de 1	1					
Insertion de 5	1	5				
Insertion de 4	1	4	5			
Insertion de 6	1	4	5	6		
Insertion de 9	1	4	5	6	9	
Insertion de 7	1	4	5	6	7	9

Exercice 42 *Écrire une fonction* `insere` *de type* `'a list → 'a → 'a list` *qui insère un élément à sa place dans la liste donnée en argument.*

Exercice 43 *Écrire une fonction* `tri_insertion` *de type* `'a list → 'a list` *qui trie la liste donnée en entrée en suivant l'algorithme du tri par insertion.*

3.4 Tri rapide

Le principe du tri rapide est de découper la liste initiale en deux morceaux en fonction d'un pivot, trier récursivement puis joindre les deux morceaux triés.

Exercice 44 *Écrire une fonction* `decoupe` *de type* `'a → 'a list → ('a list * 'a list)` *qui à partir d'un pivot découpe une liste en deux listes constituées respectivement des éléments de la liste initiale plus petits que le pivot (respectivement plus grands).*

Exercice 45 *Écrire une fonction* `tri_rapide` *de type* `'a list → 'a list` *qui trie la liste donnée en entrée en suivant l'algorithme du tri rapide. On choisira toujours comme pivot le premier élément de la liste.*

3.5 Tri fusion (facultatif)

Le principe du tri fusion est découper la liste initiale en deux listes de taille égale, de trier récursivement ces deux listes puis de les fusionner. L'idée est que l'on peut efficacement fusionner deux listes déjà triées en une liste triée.

Exercice 46 (facultatif) *Écrire une fonction* `coupe` *de type* `'a list → ('a list * 'a list)` *qui découpe la liste initiale en deux sous-listes de même taille (à un près quand la liste initiale est de taille impaire).*

Exercice 47 (facultatif) *Écrire une fonction* `fusion` *de type* `'a list → 'a list → 'a list` *qui fusionne les deux listes triées données en argument en une unique liste triée.*

Exercice 48 (facultatif) *Écrire une fonction* `tri_fusion` *de type* `'a list → 'a list` *qui trie la liste donnée en entrée en suivant l'algorithme du tri fusion.*

3.6 Performances (facultatif)

Vous utiliserez les fonctions programmées en préliminaire de ce TP pour tester tous vos tris.

Exercice 49 (facultatif) *Comparer les temps d'exécution pour les différents tris implémentés, pour des listes d'entiers de tailles allant jusqu'à un million d'éléments. Faire plusieurs mesures, avec des listes d'entrée déjà triées, triées à l'envers, ou « aléatoires ».*

4 Pile, file et rang

Objectifs : Coder piles et files. Coder le médian (et le i-ème rang) d'une liste en temps linéaire.

4.0 Travail préliminaire : accès dans une liste

Exercice 50 *Écrire une fonction* `neme` *de type* `int → 'a list → 'a` *qui prend en argument un entier et retourne l'élément ayant cette position dans la liste. Par convention l'élément de position* 1 *sera la tête de liste.*

Exercice 51 *Écrire une fonction* `renverse` *de type* `'a list → 'a list` *qui prend en argument une liste et renvoie la liste constituée des mêmes éléments dans l'ordre inverse.*

Évaluer le nombre total *d'appels récursifs (y compris à d'éventuelles fonctions auxiliaires) que réalise votre implantation, en fonction de la taille de la liste initiale.*

4.1 Structure de pile

Une pile est une structure de données modélisant un contenant d'objets et dans lequel les objets sont retirés dans l'ordre inverse dans lequel on les a insérés (on parle de *LIFO* pour *Last In, First Out*). Dans la suite on s'intéresse uniquement aux piles d'entiers.

Exercice 52 *Définir un type* `pile` *et les opérations et constantes suivantes :*

```
val pile_vide: pile
val est_pile_vide: pile → bool
val empile: int → pile → pile
val depile: pile → (int * pile)
```

Exercice 53 *Écrire une fonction* `teste_pile` *de type* `int list → bool` *qui empile les entiers de la liste dans une pile initialement vide, reconstruit une liste en dépilant totalement la pile et vérifie que le résultat correspond bien à la liste initiale.*

4.2 Structure de file

Une file est une structure de données modélisant un contenant d'objets et dans lequel les objets sont retirés dans le même ordre que celui dans lequel on les a insérés (on parle de *FIFO* pour *First In, First Out*). Dans la suite on s'intéresse uniquement aux files d'entiers.

Exercice 54 *Écrire une fonction* `rac` *de type* `'a list → ('a * 'a list)` *qui retourne, pour une liste non vide, son dernier élément ainsi que la liste privée de son dernier élément.*

Exercice 55 *Définir un type* `file` *et les opérations et constantes suivantes :*

```
val file_vide: file
val est_file_vide: file → bool
val enfile: int → file → file
val defile: pile → (int * file)
```

Exercice 56 *Écrire une fonction* `teste_file` *de type* `int list → bool` *qui enfile les entiers de la liste dans une file initialement vide, reconstruit une liste en défilant totalement la file et vérifie que le résultat correspond bien au miroir de la liste initiale.*

4.3 Médian et rangs (facultatif)

Le i-ème rang d'une liste est le i-ème plus petit élément :
– le premier rang est le minimum ;
– le n-ème rang est le maximum d'une liste de taille n ;
– le médian est le $\lceil \frac{n}{2} \rceil$-ème rang d'une liste de taille n.

Exercice 57 (facultatif) *En utilisant un tri préalable de votre choix, écrire une fonction* `rang` *de type* `int → 'a list → 'a` *qui retourne l'élément du rang demandé de la liste passée en paramètre.*

Exercice 58 (facultatif) *Écrire une fonction* `median` *de type* `'a list → 'a` *qui calcule le médian d'une liste.*

Quitte à trier, il est toujours possible de calculer le i-ème rang en temps $O(n \log n)$. Mais il est possible de le calculer en temps linéaire, comme nous allons le faire ici.

Exercice 59 (facultatif) *Écrire une fonction* `median_naif` *de type* `'a list → 'a` *qui calcule le médian d'une liste sans la trier. Cette fonction n'a pas besoin d'être efficace.*

Exercice 60 (facultatif) *Écrire une fonction* `paquets` *de type* `'a list → int → 'a list list` *découpant une liste l en paquets de taille n (sauf éventuellement l'un d'entre eux) et renvoyant la liste des paquets.*

Exercice 61 (facultatif) *En utilisant les deux fonctions précédentes, écrire une fonction* `liste_medians` *de type* `'a list → int → 'a list` *qui calcule la liste des médians des paquets obtenus en coupant la liste initiale.*

Exercice 62 (facultatif) *Écrire une fonction* `decoupe` *de type* `'a list → 'a → ('a list × 'a list)` *qui découpe une liste l en deux parties, les éléments qui sont plus petits qu'un pivot p d'un côté et les plus grands dans l'autre.*

La stratégie du calcul du i-ème rang est la suivante :

1. Découper la liste initiale en paquets de taille 5.
2. Trouver le médian de chaque paquet (avec `median_naif`) puis (récursivement) le médian des médians p.
3. Découper la liste initiale en deux morceaux $(bas, haut)$ des éléments qui sont plus petits (pour bas) ou plus grand (pour $haut$) que p.
4. Chercher récursivement le i-ème rang de la liste initiale dans le bon morceau bas ou $haut$.

Exemple : on cherche le cinquième plus petit élément de la liste l. Le découpage de l autour du « médian des médians » p donne 2 éléments plus petits que p et 12 plus grands que p. Il suffit maintenant de chercher le deuxième plus petit élément de la liste $haut$.

Exercice 63 (facultatif) *Écrire une fonction* `rang` *de type* `'a list → int → 'a` *calculant le i-ème rang de la liste donnée un argument et utilisant la stratégie décrite plus haut.*

Exercice 64 (facultatif) *Écrire une fonction* `rang_naif` *de type* `'a list → int → 'a` *calculant le i-ème rang de la liste donnée un argument et utilisant comme stratégie le tri préalable de la liste initiale.*

Exercice 65 (facultatif) *Comparer les temps d'exécution de* `rang` *et de* `rang_naif` *pour des listes de taille variable, jusqu'au million d'éléments.*

5 Quadtree

Objectifs : Manipuler des arbres représentant des images en niveau de gris.

5.0 Travail préliminaire : apprivoiser les quadtrees

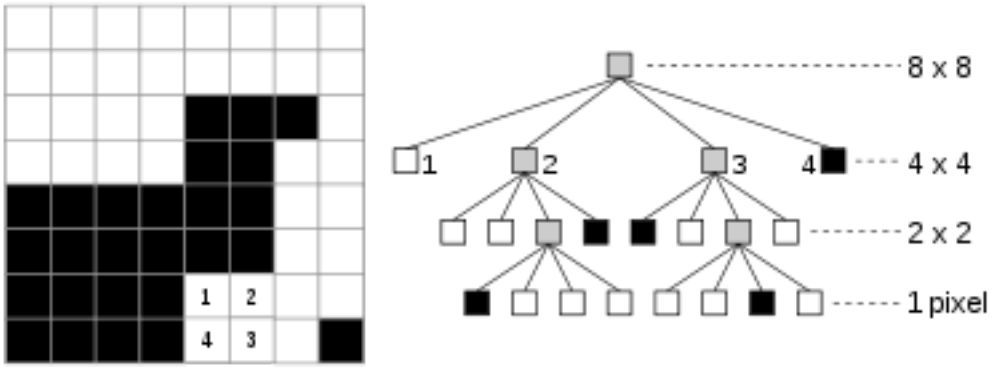
1. Dessiner (sur papier !) une image simple en noir et blanc de 8x8 pixels, par exemple un smiley ou une icône.
2. Construire, toujours sur papier, le quadtree correspondant (voir section 5.1).
3. Représenter (dans un fichier .ml) ce quadtree dans le type OCaml proposé dans cet énoncé.
4. Représenter cette même image sous forme d'un tableau de 8x8 pixels (voir section 5.2).

Cet exemple pourra vous servir de premier cas de test pour vos fonctions. Que ne permet-il pas de tester correctement ?

5.1 Quadtree

Dans une image il est fréquent que de larges portions contiguës aient la même couleur, et il est souhaitable d'en tirer parti pour représenter ces images de façon compressée. Nous allons étudier une représentation de ce type, appelée *quadtree*.

Pour simplifier, on suppose les images carrées, de côté 2^n, et en niveaux de gris. L'idée est la suivante : une image unie se représente par sa couleur, tandis qu'une image composite se divise naturellement en quatre images carrées.



Nous considérons le type suivant :

```
type quadtree = Feuille of int
              | Noeud of quadtree * quadtree * quadtree * quadtree
```

Par convention l'ordre des sous-images dans le quadruplet sera le suivant : Nord-Ouest, Nord-Est, Sud-Est, Sud-Ouest.

Exercice 66 *Écrire une fonction* `rot_pos` *de type* `quadtree` $\rightarrow$ `quadtree` *qui effectue une rotation de l'image dans le sens inverse des aiguilles d'une montre (c'est-à-dire dans le sens positif).*

Exercice 67 *Écrire une fonction* `rot_neg` *de type* `quadtree` $\rightarrow$ `quadtree` *qui effectue une rotation de l'image dans le sens des aiguilles d'une montre (c'est-à-dire dans le sens négatif).*

Exercice 68 *Écrire une fonction* `miroir_hori` *de type* `quadtree → quadtree` *qui effectue une symétrie selon l'axe horizontal.*

Exercice 69 *Écrire une fonction* `miroir_vert` *de type* `quadtree → quadtree` *qui effectue une symétrie selon l'axe vertical.*

Exercice 70 *Écrire une fonction* `inversion_video` *de type* `quadtree → int → quadtree` *qui modifie une image en inversant les niveaux de gris : le niveau* 0 *devient le niveau maximal (passé en argument) et inversement.*

Exercice 71 *Écrire une fonction* `max_gris` *de type* `quadtree → int` *qui parcourt une image et retourne la valeur du niveau de gris maximal présent dans l'image.*

Exercice 72 *Écrire une fonction* `min_quad` *de type* `quadtree → quadtree` *qui minimise le* `quadtree` *passé en entrée, l'idée étant que lorsque les quatre fils d'un nœud ont la même couleur, on peut les remplacer par une unique feuille de cette couleur.*

5.2 Tableau de pixels

On souhaite maintenant coder ou décoder une image en niveau de gris, vers et depuis sa représentation en quadtree. On se contentera pour l'instant de produire un tableau de valeurs entières correpondant à chaque pixel de l'image.

Des primitives sur les tableaux sont disponibles à l'adresse suivante :

`http://caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html`

(seule la première page devrait vous être utile).

Exercice 73 *Écrire une fonction* `tab_vers_quad` *de type* `int array → quadtree` *qui prend en argument un tableau d'entiers et renvoie le* `quadtree` *minimal correspondant à l'image lue. On supposera que l'image est carrée et de dimension une puissance de* 2*.*

Exercice 74 *Écrire une fonction* `quad_vers_tab` *de type* `int → quadtree → int array` *qui calcule le tableau des pixels de l'image représentée par le* `quadtree` *donné en 2è argument, le premier entier donnant la dimension du côté de l'image.*

5.3 Images rectangulaires (facultatif)

Afin de pouvoir représenter des images qui ne sont pas forcément des carrés, et dont les dimensions ne sont pas des puissances de 2, on introduit les types suivants :

```
type mixtree = Q of quadtree * int | T of recttree
 and recttree = NoeudT of mixtree * mixtree * mixtree * mixtree
```

Pour découper une image de taille $n \times m$, on commence par chercher le plus grand carré de taille une puissance de 2 qui rentre dans l'image. On le code par un `quadtree`, et on recommence récursivement avec les trois autres rectangles restants. Les quatre rectangles, dont un au moins est un carré, sont les quatre fils du `recttree` codant l'image entière.

Exercice 75 (facultatif) *Écrire une fonction* `taille_rect` *de type* `recttree → int * int` *qui donne les dimensions d'une image codée par un* `recttree`*.*

Exercice 76 (facultatif) *Écrire une fonction* `rect_valide` *de type* `recttree` $\rightarrow$ `bool` *qui vérifie si un* `recttree` *vérifie la contrainte de toujours contenir un* `quadtree` *de dimensions maximales dans son coin Nord-Ouest.*

Exercice 77 (facultatif) *Écrire une fonction* `pgm_vers_rect` *de type* `char stream` $\rightarrow$ `recttree` *qui lit un fichier PGM et retourne le* `recttree` *minimal correspondant à l'image lue.*

Exercice 78 (facultatif) *Écrire une fonction* `rect_vers_pgm` *de type* `recttree` $\rightarrow$ `string` *qui calcule le contenu d'un fichier PGM codant l'image représentée par le* `recttree` *donné en argument.*

6 Analyse syntaxique

6.0 Travail préliminaire : lecture d'un entier

Exercice 79 *Écrire une fonction* `horner` *de type* `int → char Stream.t → int` *qui consomme en début de flot le plus long préfixe de chiffres et rend l'entier dénoté par cette suite de chiffres lorsque l'entier donné en argument vaut 0. On considèrera que l'entier donné en argument correspond au décodage des chiffres déjà lus.*

Par exemple si le flux `s` débute par `"123 1981"` alors `horner 0 s` doit renvoyer 123 et `horner 42 s` doit renvoyer l'entier 42123.

6.1 Connect 6

Le jeu de Connect6 est un jeu de stratégie à deux joueurs (Noir et Blanc) se jouant sur un plateau quadrillé (de type Go) initialement vide :

1. Les joueurs jouent l'un après l'autre en posant des pierres de leur couleur, à commencer par Noir.
2. Au premier tour, Noir pose une pierre, après quoi chaque joueur pose à son tour deux pierres sur des intersections libres.
3. Le premier joueur à réaliser un alignement de 6 pierres adjacentes de sa couleur gagne la partie. Un alignement peut être horizontal, vertical ou diagonal.
4. Si le plateau est entièrement rempli sans qu'un joueur ait gagné, la partie est déclarée nulle.

Nous souhaitons parser un fichier qui contient une partie de Connect6 selon la grammaire suivante :

$$\begin{aligned} P &::= \text{noir} \mid \text{blanc} \\ C &::= (\ P\ \mathit{int}\ \mathit{int}\ ) \\ Cl &::= \varepsilon \mid C\ Cl \\ S &::= (\ \mathit{int}\ \mathit{int}\ )\ Cl \end{aligned}$$

Par exemple voici le contenu d'un fichier d'une partie de Connect6 :

```
(19 19) (noir 3 5) (blanc 5      8 )
    (blanc 5 9) (      noir 3 4)
(noir 3 6)
```

Dans cet exemple, la partie se joue sur un plateau de taille 19 par 19. Noir pose sa première pierre en position $(3, 5)$, puis Blanc en pose deux, et ainsi de suite. On numérote les positions à partir de 0.

Exercice 80
1. *Définir un type* `coup` *permettant de représenter un coup d'une partie de Connect6. Pour simplifier les choses on supposera qu'un coup est la pose d'une pierre et qu'à part au premier tour, chaque joueur joue deux coups.*
2. *Définir un type* `plateau` *qui représente l'état d'un plateau de jeu de Connect6.*

Exercice 81 *Écrire une fonction* `analex_partie` *qui réalise l'analyse lexicale d'un flux représentant une partie de Connect6.*

Exercice 82 *Écrire une fonction* `lit_partie` *qui analyse un flux représentant une partie de Connect6 et qui retourne la taille du plateau de jeu et la liste des coups de la partie.*

Exercice 83 *Écrire une fonction* `joue_coup` *qui calcule un nouvel état à partir d'un état du plateau et d'un coup.*

Exercice 84 (facultatif) *(**) Écrire une fonction* `resultat` *qui détermine l'issue de la partie pour un état du jeu passé en paramètre. L'issue pourra être une victoire (noire ou blanche), une partie nulle ou encore une partie non terminée.*

Attention : certaines entrées sont invalides, par exemple un état dans lequel Noir comme Blanc a réalisé un alignement de 6 pierres. Vous êtes libres de renvoyer ce que vous voulez sur une entrée invalide.

6.2 Quadtrees et format PGM (facultatif)

Le format PGM est un format de fichier permettant de stocker des images *bitmap* en niveaux de gris. Contrairement à beaucoup de formats de fichiers images, il est possible de stocker les données sous forme de texte, ce qui simplifie leur lecture. On souhaite pouvoir convertir une image dans ce format en `quadtree` et réciproquement.

Voici un exemple de fichier PGM (d'après Wikipédia) :

```
P2
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Le premier mot `P2` est fixe et désigne le format du fichier. Viennent ensuite les dimensions longueur × hauteur, la valeur du plus haut niveau de gris, puis l'image en elle même à raison d'un entier par pixel.

Exercice 85 (facultatif) *Écrire une fonction* `pgm_vers_quad` *de type* `char Stream.t → int * quadtree` *qui lit un fichier PGM et retourne le* `quadtree` *minimal correspondant à l'image lue, ainsi que la dimension du côté de l'image. On supposera que l'image sera carrée et de dimension une puissance de* 2*.*
Vous êtes évidemment invités à utiliser (voire à terminer !) votre fonction `tab_vers_quad` *écrite au TP précédent.*

Pour obtenir un flot de caractères à partir d'un fichier, on pourra utiliser `Stream.of_channel (open_in "nomdufichier")`.

Exercice 86 (facultatif) *Écrire une fonction* `quad_vers_pgm` *de type* `int * quadtree → string` *qui calcule le contenu d'un fichier PGM codant l'image représentée par le* `quadtree` *donné en argument, l'entier donnant la dimension du côté de l'image.*
De même on pourra (ré)utiliser la fonction `quad_vers_tab` *du TP5.*

Pour sauvegarder la chaîne de caractères obtenue on pourra utiliser `output_string (open_out "nomdufichier") chaine`.

7 Bibliothèque Graphique

7.0 Travail préliminaire : compilation

L'objet de ce TP est de vous familiariser avec la bibliothèque graphique que vous devrez utiliser dans le projet de fin de semestre. Dans ce TP nous compilerons les programmes écrits, au lieu de les interpréter dans emacs comme d'habitude. Pour cela, un `Makefile` vous est fourni : `http://www-verimag.imag.fr/~wack/APF_2014_2015/fichiers-TP07`

En guise de travail préliminaire, il vous est demandé de parcourir le précis de compilation fourni en annexe de ce cahier de TP, et de faire les exercices proposés à propos de `make`.

7.1 La bibliothèque graphique

Nous souhaitons écrire un module de calcul de primitives géométriques en 2D. Nous utilisons la bibliothèque graphique LablGL [3] en OCaml pour l'affichage. Vous pouvez lire le code source déjà écrit, car nous n'allons bien entendu pas redévelopper cette bibliothèque graphique. Nous vous fournissons un module encapsulant les fonctions permettant de dessiner dans une fenêtre carrée de taille `1.0` × `1.0`. Sur un système Debian/Ubuntu, cette bibliothèque est installée par les deux paquets suivants : `liblablgl-ocaml-dev` et `liblablgl-ocaml`. Si ces deux paquets sont absents de votre machine, installez-les avec la commande `sudo apt-get install liblablgl-ocaml-dev liblablgl-ocaml`

7.2 Premiers dessins

Nous rappelons qu'en OCaml il de bon goût de définir π par `let pi = 4. *. atan 1.`

Nous définissons un point par un enregistrement à deux valeurs flottantes. Un vecteur peut être représenté de la même manière. Nous considérons trois types de surfaces : un cercle, une ligne et un polygone plein. Nous avons aussi besoin de définir des opérations permettant de manipuler les vecteurs.

1. Implémentez le module `Geo` défini par sa signature donnée dans le fichier `geo.mli`.
2. Testez votre module en traçant un cercle, un segment, une ligne brisée, un triangle plein et un polygone à 5 côtés. Pour cela nous vous fournissons le module `Aff` dont la signature est dans le fichier `aff.mli` et l'implémentation dans le fichier `aff.ml`. La signature comporte une unique fonction `draw`, qui prend en paramètre le titre de la fenêtre d'affichage sous forme d'une chaîne de caractères et une liste de surfaces qui seront dessinées.
   Le code suivant permet de tester si la bibliothèque `LablGL` est bien installé sur votre machine.
   ```
   Aff.draw "Essai"
       [Geo.Line((Geo.point 0.2 0.5),(Geo.point 0.8 0.5));
        Geo.Line((Geo.point 0.5 0.2),(Geo.point 0.5 0.8))];;
   ```

7.3 Application : Dessiner des fractales

Nous allons maintenant dessiner des fractales en 2D, en utilisant les modules `Geo` et `Aff`. Une fractale est une courbe ou surface qui se dessine de manière itérative. Nous appelons *nombre d'itérations* le nombre d'étapes nécessaires à sa construction.

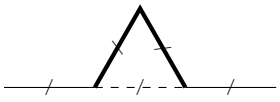
Pour chaque fractale décrite ci-dessous :

1. Implémentez son tracé par une fonction qui prend un segment et un nombre d'itérations fixé et produit la figure fractale associée.
2. Testez avec un nombre significatif d'itérations (selon la complexité de la fractale considérée).

3. `http://wwwfun.kurims.kyoto-u.ac.jp/soft/lsl/lablgl.html`

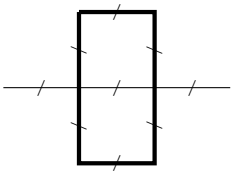
7.3.1 Flocon de Koch

Le flocon de Koch est un polygone (creux). Nous débutons avec un triangle équilatéral. À chaque itération et pour chaque côté du polygone, nous divisons ce côté en trois segments égaux. Nous traçons un triangle équilatéral de base le segment du milieu et orienté vers l'extérieur. Ensuite nous retirons ce segment.



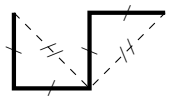
7.3.2 Courbe de Peano

La courbe de Peano est une ligne brisée, initialement constituée d'un seul segment. À chaque itération et pour chaque segment de la ligne brisée, nous divisons ce segment en trois parties égales. Nous traçons deux carrés de part et d'autre du segment ayant pour coté commun la partie du milieu.



7.3.3 Courbe du dragon (*) (facultatif)

La courbe du dragon est une ligne brisée entre deux points A et B, initialement constituée d'un seul segment $[AB]$. À chaque itération et pour chaque segment de la ligne brisée (dans l'ordre de A vers B), nous dessinons un triangle rectangle isocèle ayant ce segment pour hypoténuse. Le triangle est situé à droite du segment (en regardant vers B) si celui-ci est de rang impair (dans l'ordre des segments considérés à cette itération), sinon à sa gauche. Puis nous effaçons l'hypoténuse.

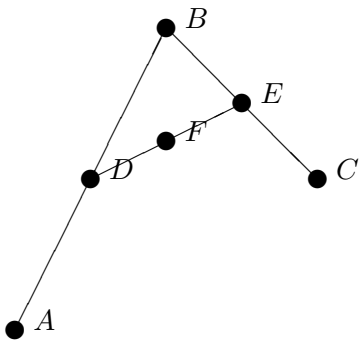


Remarque : Vous pouvez bien entendu coder d'autres fractales comme par exemple la fractale de Vicsek ou celle de Cesàro ou encore la courbe de Moore.

7.4 Courbe de Bézier quadratique : une fractale qui n'en est pas une ! (facultatif)

La courbe de Bézier de points de contrôle A, B et C est obtenue en remplaçant le polygone A, B, C par les deux polygones A, D, F, et F, E, C :
– D Etant le milieu de $[AB]$
– E Etant le milieu de $[BC]$
– F Etant le milieu de $[DE]$
et en itérant.

Le résultat est une courbe lisse, passant par A et C et tangente : $[AB]$ et $[BC]$.



Exercice 87 (facultatif) *Réaliser une fonction qui étant donné trois points et un nombre d'itérations, trace la courbe de Bézier associée.*

8 Modules et foncteurs

Certaines constructions syntaxiques qui vous seront nécessaires ne sont pas données ou décrites en détail [4]. Il est **fortement conseillé** de tester vos fonctions avec des entrées judicieusement choisies.

Objectifs : L'objet de ce TP est double. Il s'agit d'une part de s'entraîner avec le mode compilé d'OCaml, d'autre part de mettre en pratique sur des exemples les notions de modules, signatures et foncteurs introduites en TD.

8.0 Travail préliminaire : modules et compilation

Lire le précis de compilation en appendice.

Nous avons jusqu'ici utilisé le mode interprété pour effectuer nos calculs en Caml. Il est possible de *compiler* un programme Caml, c'est–dire générer un exécutable à partir d'un code source. Cela peut permettre par exemple de gagner en efficacité.

Considérons le programme suivant, permettant d'afficher la liste triée des éléments contenus dans un fichier passé en paramètre, et constitué des fichiers :
- `main.ml` qui prend en entrée le type des éléments à trier, le nom de l'algorithme de tri à utiliser et le nom du fichier contenant la liste à trier ; il affiche cette liste une fois triée en utilisant les fichiers `ordre.ml` et `tri.ml`.
- `ordre.mli` et `tri.mli` qui spécifient la signature des modules `Ordre` et `Tri` (correspondant respectivement aux fichiers `ordre.ml` et `tri.ml`) tels qu'ils sont utilisés par `main.ml`.
- `ordre.ml` et `tri.ml` qui réalisent les signatures définies dans les `.mli`.

Les fichiers sont fournis sur le site Web du cours, à part `ordre.ml` et `tri.ml` qui sont à réaliser.

Observer le contenu de ces fichiers. Remarquer en particulier que pour accéder au module Entier du fichier `ordre.ml` (par exemple depuis le fichier main.ml), on utilise Ordre.Entier. Le fichier ordre.ml définit donc implicitement un module Ordre qui contient plusieurs sous-modules.

Attention : un `.mli` indique seulement ce que contient (ou doit contenir) le `.ml` correspondant. Cela signifie que la signature du module type `TypeOrdonne` contenue dans `ordre.mli` ne *définit* pas `TypeOrdonne`, qui doit donc également être présent dans le `.ml` pour pouvoir être utilisé.

Exercice 88 *Dans le fichier* `ordre.ml`*, implémenter les modules Entier et Chaine à partir de leur signature donnée dans* `ordre.mli`*.*

Exercice 89 *Utiliser le fichier* `tri.ml`*, puis :*
- *Implémenter le module Quicksort à partir de la signature donnée dans* `tri.mli`*.*
- *Compiler puis tester le programme obtenu.*
- *Écrire un* `Makefile` *afin de préciser l'ordre de compilation choisi.*

8.1 Foncteurs

Nous présentons dans ce TP une application des foncteurs. Nous montrons comment, en instanciant astucieusement un module paramétré qui réalise des calculs sur des matrices, nous réalisons simplement deux autres tâches. Pour cela, nous construisons un foncteur que nous appliquerons dans un premier temps au calcul matriciel afin de le tester, puis à des calculs de code correcteur d'erreur d'une part, et sur des relations binaires d'autre part.

4. Pour plus de précisions référez-vous à la documentation en ligne `http://caml.inria.fr` dans « resources » suivre « Objective Caml manual » puis « The core language »

8.1.1 Produit matriciel

On rappelle que si $A = (a_{ij})$ est une matrice de taille $m \times n$ et $B = (b_{ij})$ est une matrice de taille $n \times p$, alors leur produit, noté $AB = (c_{ij})$, est une matrice de taille $m \times p$ donnée par :

$$\forall i, j : c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}$$

Exercice 90 *Adapter le module* `Groupe` *(voir feuille TD) de la façon suivante : à partir de* `Groupe`*, on crée* `SemiAnneau` *en supprimant l'opération* `oppose` *et en ajoutant une opération* `produit` *ainsi qu'un élément neutre pour ce produit. On obtient ainsi une structure qui n'est plus un groupe mais un semi-anneau.*

Si les opérateurs `plus` et `produit` définis dans notre signature `SemiAnneau` sont notés respectivement $\oplus$ et $\otimes$, le produit matriciel s'écrit ainsi :

$$\forall i, j : c_{ij} = a_{i1} \otimes b_{1j} \oplus a_{i2} \otimes b_{2j} \oplus \cdots \oplus a_{in} \otimes b_{nj}$$

On étend ces notations à la somme $(A \oplus B)$ et au produit $(A \otimes B)$ de matrices.

Exercice 91 *Adapter le foncteur* `Matrices` *en ajoutant l'opération* `produit`*, en utilisant les fonctions intermédiaires suivantes :*
- *une fonction qui prend une matrice A et renvoie le couple constitué de la première colonne de A et de A privée de sa première colonne.*
- *une fonction qui multiplie une ligne par une colonne.*
- *une fonction qui multiplie une ligne par une matrice.*

Exercice 92 *Tester votre produit sur des matrices d'entiers.*

8.1.2 Code correcteur d'erreurs

On considère un code de Hamming binaire donné par sa matrice génératrice G et sa matrice de contrôle H à coefficients dans $\{0, 1\}$, par exemple

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad H = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Exercice 93 *Définir un module* `Z2` *de signature* `SemiAnneau` *qui réalise les opérations dans $\mathbb{Z}/2\mathbb{Z}$. Autrement dit, toutes les opérations sont effectuées modulo 2 et donc à valeurs dans $\{0, 1\}$.*

On rappelle que le codage d'un message u est donné par le produit matriciel $u \otimes G$, à condition d'effectuer toutes les opérations dans $\mathbb{Z}/2\mathbb{Z}$.

Réciproquement, étant donné un mot transmis z comportant au maximum une erreur :
- $z \otimes {}^tH = 0$ si et seulement si z ne comporte pas d'erreur.
- Dans le cas où il y a une erreur, $z \otimes {}^tH$ est égal (à transposition près) à l'une des colonnes de H, et l'erreur commise porte sur le bit de z dont la position correspond à cette colonne.

Exercice 94 *Écrire les fonctions suivantes :*

– Dans *le foncteur* `Matrices`*, une fonction* `transpose` *qui calcule la transposée d'une matrice donnée. Puis* à l'aide *du foncteur* `Matrices`*,*
– `codage` *qui effectue le codage d'un mot donné par une matrice donnée ;*
– `corrige` *qui corrige (si nécessaire) tout mot comprenant au maximum une erreur.*
Tester enfin vos fonctions sur différents exemples : transmission sans erreur, avec une erreur, avec plus d'une erreur.

8.1.3 Calculs sur des relations (facultatif)

On change dans cette partie de contexte : on considère une relation binaire R sur un ensemble fini. Quitte à nommer $\{1, \ldots, n\}$ les éléments de cet ensemble, on peut représenter la relation R par une matrice de booléens $M(R)$: le coefficient m_{ij} vaut `true` si et seulement si la relation $i\,R\,j$ est vérifiée.

Ainsi, par exemple, la relation $<$ sur les entiers $\{1, \ldots, 5\}$ est représentée par la matrice

$$M = \begin{pmatrix} false & true & true & true & true \\ false & false & true & true & true \\ false & false & false & true & true \\ false & false & false & false & true \\ false & false & false & false & false \end{pmatrix}$$

On s'intéressera alors à vérifier certaines propriétés de la relation R et à calculer des relations engendrées par R. Ces propriétés et relations peuvent être pour la plupart obtenues par des opérations matricielles : on va donc à nouveau traiter le problème posé en étendant et en instanciant correctement le foncteur décrit dans la section 8.1.1.

Pour tester vos fonctions, il est conseillé de générer aléatoirement des relations sous forme de matrices à coefficients booléens, où pour tout couple d'éléments (x, y), la relation $x\,R\,y$ est vérifiée avec une probabilité $pr \in [0, 1]$ donnée.

Exercice 95 (facultatif) *1. Définir le module* `Boolean` *qui exprime que les booléens ont une signature de* `SemiAnneau` *pour les opérateurs booléens usuels.*

2. Ajouter au foncteur `Matrices` *une fonction* `identite` *qui construit la matrice identité de taille n donnée en argument. (elle a l'élément neutre pour le produit sur sa diagonale et l'élément neutre pour la somme partout ailleurs).*

Exercice 96 (facultatif) *Pour écrire les fonctions suivantes, on essayera au maximum d'utiliser des opérations matricielles déjà implantées (mais ce n'est pas toujours possible) :*
– `symetrique` *qui détermine si une matrice donnée représente une relation symétrique, c'est-à-dire* $\forall i, j \in \{1, \ldots, n\}, i\,R\,j \Leftrightarrow j\,R\,i$ *;*
– `reflexive` *qui détermine si une matrice donnée représente une relation réflexive, c'est-à-dire* $\forall i \in \{1, \ldots, n\}, i\,R\,i$ *;*
– `totale` *qui détermine si une matrice donnée représente une relation totale, c'est-à-dire* $\forall i, j \in \{1, \ldots, n\}, i\,R\,j \vee j\,R\,i$.

Exercice 97 (facultatif) *(**) Écrire une fonction qui calcule la relation de préordre engendrée par une relation R. On pourra se baser sur l'équation suivante (méthode de Horner) :*

$$I \oplus R \oplus \ldots \oplus R^n \quad = \quad I \oplus R \otimes (I \oplus R \otimes (\ldots (I \oplus R)) \cdots)$$

*Montrer que $I \oplus R \oplus \ldots \oplus R^N = (I \oplus R)^N$ dans le semi-anneau $(\vee, \wedge)$. En déduire un algorithme plus efficace que le précédent pour le calcul de la relation de préordre engendrée. (***Rappel :** *il existe une méthode d'*exponentiation rapide *pour calculer une puissance en temps logarithmique de l'exposant).*

A La compilation avec ocamlc

Pour compiler un fichier source `code.ml` en fichier binaire exécutable `prog`, tapez la commande :

`ocamlc -o` *`prog code`*`.ml`

Un fichier `prog` est créé. Celui-ci contient un appel à la machine virtuelle (`ocamlrun`) et le byte-code correspondant au programme lui-même. Ce fichier peut être exécuté en tapant :

`./prog`

Les tableaux suivants récapitulent l'ensemble des différents fichiers mis en jeu lors de la compilation de programmes CAML. Les fichiers sources sont :

`.ml`	code source d'un ensemble de définitions considérées dans l'ordre de leur déclaration
`.mli`	description du type de toutes les définitions du `.ml` correspondant que l'on souhaite rendre *visibles* de l'extérieur (fichier d'interface)

Les fichiers produits sont :

`.cmo`	*bytecode* résultant de la compilation d'un `.ml`
`.cmi`	version compilée d'un fichier d'interface `.mli`
`.cma`	collection de `.cmo` (librairie)

Reste l'exécutable final, par défaut a.out et sinon `xxx`, qui est produit à partir d'un ensemble de `.cmo` et de `.cma`. Cet exécutable est en fait du *bytecode* qu'interprète la machine virtuelle `ocamlrun` ; ce n'est donc pas un véritable *standalone executable*.

A.1 Compilation séparée avec ocamlc

Le compilateur `ocamlc` s'utilise de la façon suivante. On compile un fichier source `f.ml` par la commande :

`ocamlc -c f.ml` *effectue* `.ml` $\longrightarrow$ `.cmo`

Lors de cette compilation il est vérifié que le type *inféré* d'une définition est bien en accord avec celui déclaré dans le fichier d'interface.

Si dans le fichier source on souhaite utiliser une définition rendue visible par un autre `.mli`, il faut indiquer au compilateur où elle est spécifiée (*i.e.* `autre.mli`). Deux choix possibles : soit utiliser la directive `open Autre` au début du fichier source, soit *qualifier* toute utilisation de la définition par le préfixe « `Autre.` » (ce qui peut s'avérer meilleur pour la lisibilité du programme).

On compile de la même manière un fichier d'interface :

`ocamlc -c f.mli` *effectue* `.mli` $\longrightarrow$ `.cmi`

Pour gagner du temps dans l'écriture de vos fichiers d'interface vous pouvez générer automatiquement le `.mli` correspondant à l'ensemble d'un fichier source ainsi :

`ocamlc -i f.ml > f.mli` *effectue* `.ml` $\longrightarrow$ `.mli`

La création d'une librairie s'effectue simplement en donnant l'ensemble des `.cmo` qui la compose.

`ocamlc -a f1.cmo f2.cmo f3.cmo -o lib.cma` *effectue* `.cmo` $\longrightarrow$ `.cma`

Enfin on crée l'exécutable par :

`ocamlc f4.cmo f5.cmo lib.cma f6.cmo -o xxx` *effectue* `.cma + .cmo` $\longrightarrow$ `xxx`

L'ordre dans lequel on liste les `.cmo` a une importance. Si un certain `.cmo` utilise une définition introduite dans un autre `.cmo`, ce dernier doit figurer avant. Les `.cmi` interviennent donc ici. On notera d'ailleurs que si vous n'avez pas fourni de `.mli` pour un certain fichier source, le compilateur crée automatiquement un `.cmi` rendant visible toutes les définitions décrites dans ce dernier.

On peut s'interroger sur la réelle valeur ajoutée des librairies, qui ne sont qu'une concaténation de `.cmo`. Elle réside dans l'inclusion des seules définitions réellement utilisées dans l'exécutable final.

A.2 Compilation directe

La chaîne de compilation décrite précédemment permet la compilation séparée des différentes parties qui composent l'exécutable final. Hors de ce cadre, on peut directement crée l'exécutable à partir des sources comme suit :

`ocamlc f1.ml f2.ml f3.ml -o xxx` *effectue* `.ml` $\longrightarrow$ `xxx`

B La compilation avec `ocamlopt`

Le compilateur `ocamlopt` permet la production d'un exécutable plus efficace, qui est un véritable *standalone executable*.

Pour ce qui est des fichiers générés, les `.cmo` prennent l'extension `.cmx` et les `.cma` celle de `.cmxa`. Enfin un fichier `.o` (fichier objet compatible avec les compilateurs C) est créé avec tout `.cmx`.

B.1 Différences avec `ocamlc`

Chacun des deux compilateurs a sa raison d'être.
– l'efficacité du code produit est l'affaire d'`ocamlopt`.
– la rapidité de la compilation revient à `ocamlc`.
– la portabilité du code c'est le principe d'`ocamlc`.
L'utilisation d'`ocamldebug` n'est possible que sur les exécutables compilés avec `ocamlc` et l'option de débogage `-g`.

B.2 Options de compilation

Les compilateurs `ocamlc` et `ocamlopt` peuvent être invoqués avec plusieurs options dont les suivantes :
`-c` ne fait que la compilation en elle-même, c'est-à-dire que la création des fichiers objet, mais pas l'édition de liens et donc pas la création du fichier exécutable.
`-i` affiche les types inférés sans créer de fichier.
`-o` *nom* spécifie le nom du fichier de sortie.

C Compilation et Makefile

C.1 Introduction à Make

Make est un outil répandu pour la compilation de projet. Il permet d'exécuter des *commandes* afin de créer des fichiers, appelés *cibles*. La date de création des fichiers et les *dépendances* de construction entre eux sont prises en compte pour ne refaire que ce qui est nécessaire après que certains fichiers soient modifiés. Les principaux intérêts de Make sont de :

1. Ne recompiler que ce qui a été modifié.
2. Permettre de compiler un projet avec une seule commande, `make`, connue de tout le monde.
3. Éviter la plupart des erreurs de manipulation lors de la recompilation d'un projet.

Un fichier `Makefile` est constitué de règles de transformation. Les règles d'un projet sont constituées de *cibles*, *dépendances* et *commandes*, et sont décrites dans un fichier nommé `Makefile`. La syntaxe élémentaire de ce fichier (en tout cas celle de GNU Make) est présentée ci-dessous.

C.1.1 Règles

Une règle est constituée d'une *cible* (le fichier à construire), de *dépendances* (les fichiers dont dépend la construction de la cible) et de *commandes* (la méthode de construction de la cible).

```
cible:  dépendances ...
        commandes
        ...
```

Les commandes ne seront exécutées que si la cible n'existe pas ou qu'une des dépendances a une date de création plus récente que celle de la cible.

Remarque : Make est sensible à la casse, que ce soit pour le nom du fichier `Makefile` ou celui des variables, cibles, etc. De plus, chaque commande dans une règle **doit** être précédée d'une *tabulation*.

C.1.2 Variables

Pour faciliter l'écriture et la maintenance d'un `Makefile`, il est possible de définir des variables et de leur affecter une valeur, puis de les appeler plus tard, en suivant la notation ci-dessous :

```
NOM=valeur
...
$(NOM)
```

Il est d'usage de définir une variable `SRC` listant les fichiers source, une variable `BIN` donnant le nom de l'exécutable à construire, ainsi qu'une variable par commande externe utilisée (`RM`, etc.). De cette façon, il est facile de modifier ces paramètres en fonction des changements du projet et de l'environnement.

Remarque : Par défaut, toutes les variables d'environnement (`HOME`, `USER`, etc.) sont récupérées sous forme de variables par Make.

C.1.3 Variables automatiques

Pour faciliter l'écriture des commandes à l'intérieur d'une règle, plusieurs variables sont définies automatiquement par MAKE. :

`$@` cible de la règle	`$?` dépendances plus récentes que la cible
`$<` première dépendance	`$+` toutes les dépendances

C.1.4 Règles muettes

Il est possible d'écrire une règle dont les commandes ne génèrent aucun fichier, par exemple pour supprimer les fichiers temporaires générés par la compilation. Dans ce cas, il faut spécifier à MAKE que cette règle doit être exécutée même si la cible existe déjà. C'est le rôle de la directive `.PHONY`.

```
clean:
        rm -f -v tmpfile
.PHONY: clean
```

C.1.5 Invocation

Enfin, MAKE peut être invoqué par la commande `make` qui cherchera s'il existe un fichier `Makefile` dans le répertoire courant et exécutera sa première cible par défaut. Il est possible de spécifier des variables et des cibles à construire sur la ligne de commande comme suit :

```
VARIABLE=valeur make cible1 cible2 ...
```

Remarque : La première cible se nomme souvent `all` et liste les cibles à construire par défaut.

C.2 Exemple : Pi

Considérons par exemple un projet « simple », disons un programme qui calcule et affiche une approximation du nombre π. Le projet comporte le fichier `pi.ml` ci-dessous.

```
let pi = 4. *. atan 1. in
print_float pi;
print_string "\n"
```

Exercice 98 *Créez un fichier* `Makefile` *et écrivez y une règle pour compiler l'exécutable* `appropi` *à partir du fichier source* `pi.ml`*, en définissant au préalable les variables d'usage pour cette règle et en utilisant autant que possible les variables automatiques.*

Exercice 99 *Ajoutez y une règle* muette*, nommée* `clean`*, pour supprimer les fichiers intermédiaires générés par la compilation.*

Exercice 100 *Ajoutez une autre règle* muette*, nommée* `mrproper`*, pour supprimer tous les fichiers générés par la compilation, en utilisant la règle précédente comme dépendance.*

Exercice 101 *Ajoutez encore une règle* muette*, nommée* `run`*, qui exécute le programme compilé.*

Exercice 102 *Ajoutez enfin une règle* muette*, nommée* `all`*, qui compile le programme, puis l'exécute, en utilisant uniquement d'autres règles comme dépendance. Cette règle doit être exécutée par défaut lors de l'invocation de la commande* `make`*.*