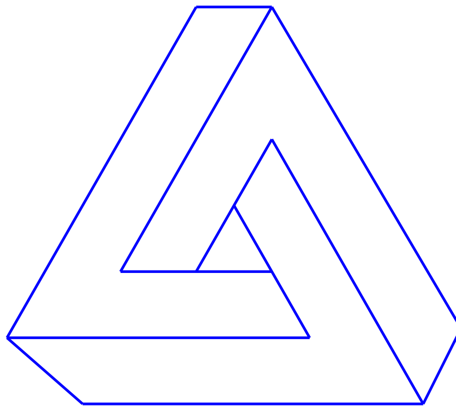
Polytech Grenoble

PF : Programmation Fonctionnelle

Livret d'exercices



Erwan Jahier, Jean-François Monin et Benjamin Wack

Avant Propos

Ce document contient l'ensemble des exercices proposés par l'équipe pédagogique de PF. Le nombre d'exercices est considérable et nous ne traiterons pas tous les exercices en séance, ceci afin de permettre aux étudiants de s'exercer par eux mêmes grâce aux exercices non traités. Nous indiquons par des étoiles la difficulté des exercices proposés : plus il y a d'étoiles plus l'exercice est jugé difficile. Enfin nous indiquons les exercices proposés les années passées (comme des parties d'examen ou bien des sujets de devoir maison).

Nous avons choisi le langage OCaml pour illustrer la programmation fonctionnelle, mais la plupart des exercices pourraient être résolus en utilisant un autre langage fonctionnel comme par exemple Haskell, Lisp ou encore Scheme. Nous donnerons en séance la complexité des solutions proposées ainsi qu'une preuve de terminaison et de correction.

Plan : Nous commençons par des exercices sur les principes de base de la programmation fonctionnelle. Nous proposons ensuite une série d'exercices concernant les preuves de programme qui en programmation fonctionnelle sont très proches des preuves mathématiques. Puis, nous abordons rapidement certains algorithmes de tris avant de parler de structure arborescente. Nous traiterons des notions de module, foncteur et d'ordre supérieur pour clore les exercices caractéristiques de l'approche fonctionnelle. Enfin nous finirons le semestre en parlant de parseur, d'aspect impératif et de lambda-calcul.

Table des matières

1 Bases **4**
1.1 Fonctions 4
1.2 Récursivité 5
1.3 Types sommes 6
1.4 Types produits 8
1.5 Types sommes et produits récursifs : Listes 8
1.6 Applications 10
1.7 Évaluation 13

2 Preuves de programme **14**

3 Tris de liste **16**
3.1 Tri par insertion 16
3.2 Tri rapide (*Quicksort*) 16
3.3 Tri crêpe (Examen 2010) 17

4 Exceptions **18**

5 Structures arborescentes **19**
5.1 Arbres binaires 19
5.2 Arbres binaire de recherche 20

6 Modules et foncteurs **22**
6.1 Modules 22
6.2 Foncteurs 27

7 Ordre supérieur **28**

8 Parseurs et flots **31**

9 Aspects impératifs **33**
9.1 Programmation impérative 33
9.2 Référence 33
9.3 Tableaux 34

10 λ-Calcul **35**

La bibliothèque standard OCaml Le compilateur OCaml est fourni avec une bibliothèque standard comprenant un ensemble de modules, dont par exemple :

- le module *List*, regroupant les opérations standard sur les listes (*map*, *mem*, *sort*...) ;
- le module *Array*, regroupant les opérations standard sur les tableaux ;
- le module *Arg* permettant l'accès aux arguments de la ligne de commande ;
- les modules *Stack* (piles), *Queue* (files), les foncteurs *Set* et *Map* permettant de gérer respectivement des ensembles et des associations d'éléments ordonnés ;
- etc.

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

1 Bases

1.1 Fonctions

Exercice 1 : Typage des expressions

Si l'expression est typable dans le contexte proposé alors précisez son type et donnez sa valeur.

CONTEXTE	EXPRESSION	TYPE	VALEUR
$let\ a = 4\ and\ b = 7$	$a + b$	*int*	11
$let\ a = 4\ and\ b = 2.3$	$a + b$		
$let\ a = 4\ and\ b = true$	$a + b$		
$let\ a = 7\ and\ b = 4$	$a < b$		
$let\ a = 4\ and\ b = 7\ and\ c = 5$	$a - b - c$		
$let\ a = 4\ and\ b = 7\ and\ c = 5$	$a < b < c$		
$let\ a = 4\ and\ b = 7\ and\ c = 5$ $and\ d = 3$	$a < b \wedge c < d$		
$let\ a = 4\ and\ b = 7\ and\ c = 5$	$(a > b \wedge a < c) \vee a < b$		
$let\ a = 4\ and\ b = 7\ and\ c = 5$	$a > b \wedge (a < c \vee a < b)$		
$let\ a = 4\ and\ b = 7\ and\ c = 5$	$(a > b \wedge a < c) \vee a > b$		
$let\ a = 4\ and\ b = 7\ and\ c = 5$	$(a > b \vee a <= b) \wedge a < c$		
	$(a < b) \vee (a > b)$		*true*
	$(a < b) \vee (a >= b)$		

Exercice 2 : 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 3 : 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 4 : Minimum de deux entiers

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

Exercice 5 : Maximum de deux entiers

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

Exercice 6 : Min de trois entiers

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

1.2 Récursivité

Exercice 7 : Factorielle

Écrire une fonction qui calcule $n!$.

Exercice 8 : Arithmétique (Examen 2010 10 points)

1. (5 points) Écrire une fonction non récursive terminale qui prend deux entiers et calcule le premier à la puissance le second sans utiliser `**` ou les *floats*.
2. (5 points) Réécrire cette fonction de manière récursive terminale, toujours sans utiliser `**` ni les *floats*.

Exercice 9 : Pgcd

Écrire une fonction qui calcule le pgcd de deux entiers positifs.

Exercice 10 : Somme des n premiers entiers

Écrire une fonction qui calcule la somme des n premiers entiers strictement positifs.

Exercice 11 : Somme des n premiers entiers impairs

Écrire une fonction qui calcule la somme des n premiers entiers impairs positifs.

Exercice 12 : Division Euclidienne

Écrire une fonction qui calcule le reste et le quotient de la division entière de deux entiers.

Exercice 13 : Prédicats “again”

Écrire une fonction qui détermine :
- (*) si un entier est un cube parfait.
- (**) si un entier est premier.

Exercice 14 : Nombres de Fibonacci

Nous rappelons que la suite de Fibonacci est définie par :
- $u_0 = 0$
- $u_1 = 1$
- $u_{n+1} = u_n + u_{n-1}$

- Écrire une fonction qui calcule u_n en fonction de n.
- Écrire une fonction qui détermine si un entier x apparaît dans la suite de Fibonacci.

1.3 Types sommes

Dans tous les exercices de cette partie, on devra donner, après chaque définition de type, un exemple de valeur de ce type.

Exercice 15 : 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 16 : Figure

- Définir un type somme pour représenter les figures géométriques.
- Écrire dans une seule fonction *aire* les fonctions permettant de calculer les aires, en utilisant les résultats des exercices précédents.
- Reprendre les 2 questions précédentes en ajoutant une figure représentant un point (sans coordonnées).

Exercice 17 : Pays et capitales

- Définir le type `pays` qui représente les pays limitrophes de la France.
- Écrire une fonction qui donne la capitale de chacun de ces pays.

Exercice 18 : Robot jardinier

Cet exercice est long et extensible à volonté. On veut représenter un robot jardinier, dont le terrain d'action est une grille de cases pouvant contenir différentes plantes ou outils de jardinage. Le robot est capable de transporter des plantes ou des outils ; il obéit à des ordres de déplacement ou d'actions diverses qui dépendent de son chargement et du contenu de la case où il est positionnée.

La première partie de l'exercice consiste à concevoir un système de types pour les données en jeu. Ces types seront ensuite utilisé dans différentes fonctions exprimant les actions du robot.

18.1. Le robot possède une direction absolue (vers un point cardinal) dans laquelle il est susceptible d'avancer, où à partir de laquelle il peut tourner de 90 degrés dans un sens ou l'autre.
- Définir le type `direction_absolue` indiquant une direction cardinale
- Définir le type `direction_relative` indiquant un changement de direction de 90 degrés vers la gauche ou la droite
- Définir une fonction `changer_direction` effectuant un tel changement de direction (indiquer tout d'abord le type de la fonction).
- Définir un type `coordonnees` pour la position dans une grille infinie de cases
- Définir une fonction `mouvoir` permettant de se déplacer dans une direction cardinale donnée sur une distance donnée sur une grille (indiquer tout d'abord le type de la fonction).

18.2. Le jardin est composé de cases dont chacune peut contenir soit un robinet, soit un objet qui peut être une plante ou un outil. Le robot peut transporter un tel objet, mais pas de robinet. Une plante peut être un légume, une fleur de couleur ou un arbre fruitier. Parmi les légumes, on a par exemple les haricots, les carottes, les courges...

- Définir des types adéquats `legume`, `fleur`, `fruit`, `couleur`, `plante`. Donner plusieurs exemples de valeurs de type `plante`.

Pour les outils, on a des bêches de trois différentes tailles, des pioches et des arrosoirs pouvant être vides ou remplis d'un volume donné d'eau.

- Définir des types adéquats `taille`, `contenu_arrosoir`, `outil`. Donner plusieurs exemples de valeurs de type `outil`.

Un objet peut être une plante, un outil, un panier vide, un panier contenant 1 légume ou un panier contenant 2 légumes. Le contenu des cases ou le chargement d'un robot est indiqué ci-dessus.

- Définir des types adéquats `objet`, `contenu_case` et `chargement_robot`. Donner plusieurs exemples de valeurs de ces types.

18.3.
- Donner un type pour l'état du terrain.
- Donner un type pour l'état du robot.
- Donner un type pour les actions possibles du robot.

18.4. Définir des états possibles du terrain, ainsi que des fonctions ajoutant différents éléments sur un terrain donnné en argument.

1.4 Types produits

Exercice 19 : 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 20 : Cartes

— Définir un type `couleur`, pour les quatre couleurs d'un jeu de carte.
— Définir un type `valeur`, pour les valeurs des 8 cartes d'un jeu de trente deux cartes.
— Définir un type `carte` qui compte en plus un joker.
— Écrire une fonction qui donne la couleur d'une carte.

Exercice 21 : 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.

1.5 Types sommes et produits récursifs : Listes

Exercice 22 : Taille d'une liste

Écrire une fonction qui renvoie le nombre d'éléments d'une liste :
— d'entiers en utilisant le type suivant `type listesint = Nil | Cons of int * listesint`
— d'entiers en utilisant le type par défaut par OCaml pour les listes.
— générique en utilisant le type suivant `type 'a listes = Nil | Cons of 'a * 'a listes`

Exercice 23 : Appartenir à une liste

Écrire une fonction qui détermine si un élément appartient ou non à une liste.

Exercice 24 : Min d'une liste

Écrire une fonction qui calcule le minimum d'une liste non vide, sinon lève une exception.

Démontrer que le résultat appartient à la liste, et qu'il est plus petit que tous les éléments de la liste.

Exercice 25 : Max d'une liste

Écrire une fonction qui calcule le maximum d'une liste non vide, sinon lève une exception.

Exercice 26 : MaxMin d'une liste

Écrire une fonction qui calcule en un seul passage le maximum et le minimum d'une liste non vide sinon lève une exception.

Exercice 27 : Supprimer un élément e d'une liste

Écrire des fonctions qui rendent une liste dans laquelle
- une occurrence de l'élément donné e est supprimée
- toutes les occurrences de e sont supprimées.

À chaque fois on renverra la liste initiale si l'élément à supprimer n'apparaît pas dans la liste.

Exercice 28 : Duplication des éléments d'une liste

Écrire une fonction qui rend la liste où tous les éléments de liste en argument sont dupliqués. Par exemple à partir de la liste `[1;2;3]` on obtient `[1;1;2;2;3;3]`.

Exercice 29 : Concaténation de deux listes

Écrire une fonction qui prend deux listes et renvoie la concaténation des deux listes.

Démontrer que `[]` est élément neutre à gauche, élément neutre à droite, et que l'opération est associative (ces propriétés fondamentales sont fréquemment utilisées).

Exercice 30 : Égalité de deux listes

Écrire une fonction qui teste si deux listes sont égales.

Exercice 31 : Dernier élément d'une liste

Écrire une fonction qui donne le dernier élément d'une liste non vide et lève une exception sinon.

Exercice 32 : Nombre pair d'éléments

Écrire une fonction qui détermine si une liste contient un nombre pair d'éléments.

Exercice 33 : Renversement d'une liste l

Écrire une fonction qui renvoie la liste des élements de l dans l'ordre inverse.

Exercice 34 : Tous sauf le dernier

Écrire une fonction qui renvoie une liste privée de son dernier élément.

Soit `l` une liste. Démontrer qu'en concaténant `l` privée de son dernier élément et la liste `[d]`, où `d` est précisément le dernier élément de `l`, on retrouve bien `l`.

Exercice 35 : Découpage d'une liste

Écrire une fonction qui renvoie les n premiers éléments d'une liste.

Exercice 36 : Découpage d'une liste

Écrire une fonction qui renvoie les éléments d'une liste entre les position n inclus et m exclus. On suppose que la liste contient au moins $m - 1$ éléments, et on numérote les éléments à partir de 0.

Exercice 37 : Palindrome d'une liste

Écrire une fonction qui détermine si une liste est un palindrome. Proposer plusieurs versions de cet algorithme en utilisant les fonctions écrites précédemment.

Exercice 38 : Second maximum d'une liste *

Écrire une fonction donnant le deuxième plus grand élément d'une liste.

Exercice 39 : Devoir maison 2009

— Écrire une fonction `application` qui prend en arguments une fonction `f` et une liste `l` et qui retourne une liste construite en appliquant la fonction `f` à chaque élément de `l`.
— Définir une fonction `selection` qui prend en arguments une propriété `p` de type `'a -> bool` et une liste `l` et qui retourne la liste des éléments de `l` qui satisfont `p`.

1.6 Applications

Exercice 40 : Dictionnaire naïf Devoir Maison 2010

Nous souhaitons construire un dictionnaire afin de pouvoir vérifier l'orthographe d'un texte. Pour cela nous cherchons une représentation d'un dictionnaire. Un dictionnaire peut contenir par exemple les mots suivants : art, article, artifice, balle, ballon, la, langage, langue. Une solution naïve est de représenter ce dictionnaire par la séquence de ses mots : ["art" ; "article" ; "artifice" ; "balle" ; "ballon" ; "la" ; "langage" ; "langue"].

- (3 points) Définir le type d'une lettre, d'un mot et d'un dictionnaire.
- (8 points) Nous nous familiarisons avec cette structure de données en réalisant les fonctions suivantes :
  - Écrire une fonction `nbmots` qui compte le nombre de mots présents dans un dictionnaire. (2 points)
  - Écrire une fonction `taillemot` qui compte le nombre de lettre d'un mot. (2 points)
  - Écrire une fonction `nbkmots` qui compte le nombre de mots de taille k présents dans un dictionnaire. (4 points)
- (3 points) Écrire une fonction `ajoutmot` qui ajoute un mot dans un dictionnaire.
- (3 points) Écrire une fonction `estdansdico` qui vérifie si un mot est pr
- (3 points) Écrire une fonction `supprimemot` qui enlève un mot d'un dictionnaire.

Note : Cette représentation peut être améliorée en considérant des structures de données arborescentes ou des graphes acycliques.

Exercice 41 : Le codage R.L.E. (Run-Length Encoding) (Examen 2010 15 points)

Le codage R.L.E. (Run Length Encoding) est une méthode de compression de listes très simple. Son principe est de remplacer dans une liste une suite de caractères identiques par le couple constitué du nombre de caractères identiques et du caractère.

Ainsi, la liste ['a' ; 'a' ; 'a' ; 'b' ; 'a' ; 'b' ; 'b'] est compressée en [(3,'a') ; (1,'b') ; (1,'a') ; (2,'b')].

1. (8 points) Écrire une fonction de décompression qui prend une liste de données compressées et retourne la liste initiale (vous pouvez utiliser la concaténation (append : `@`) de deux listes).
2. (7 points) Écrire la fonction de compression, qui prend une liste de données l et retourne la liste de données compressée au maximum par le codage R.L.E associée à l.

Exercice 42 : Conversion

On représente l'écriture hexadécimale d'un entier par une liste d'entier de 0 à 15 et son écriture binaire par une liste de 0 et 1.

Écrire une fonction qui convertit

- un entier en sa repr
- une écriture binaire en l'entier correspondant ;
- un entier en sa repr
- une écriture hexadécimale en l'entier correspondant.

Exercice 43 : Matrices Devoir Maison 2009

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 :

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

Une matrice peut être repr

- Définir le type `Matrice` et `Vecteur`.

— Écrire une fonction qui prend une matrice A et renvoie le couple constitué du vecteur de la première colonne de A et de A privée de sa première colonne.
— Écrire une fonction qui multiplie un vecteur ligne par un vecteur colonne.
— Écrire une fonction qui multiplie un vecteur ligne par une matrice.
— Écrire une fonction qui multiplie deux matrices.

Exercice 44 : Typage et lecture de code (Examen Janvier 2012)

Pour chaque fonction OCaml, indiquez son type. Si une expression est mal typée, indiquez le. (2 points par question.) [1]

1. let f l = match l with | [] -> 2 | t : :[] -> 3 | s : :_ : :t : :q -> q
2. let c f g = g f
3. let d f g = f g
4. ```
   let rec x (y,z) = match (y,z) with
          | (y,true) -> y
          | (0,z)  -> x (0,true)
          | (y,z) -> x (y-1,z);;
   ```
5. let s u v w = u v w
6. (**) let h f g (x,y) = g (x,f(y)) ; ;

Exercice 45 : Sens d'une expression (Examen Janvier 2012)

Dans les deux questions suivantes, on vous demande de donner le type et d'expliquer en une ou deux phrases ce que fait le programme (qui est un programme correct, utile, et bien typé). Vous pourrez vous appuyer sur des exemples.

1. Première fonction.
   ```
   let rec tutu l = match l with
   | [] -> ([],[])
   | [t] -> ([t],[])
   | t1::t2::q -> let (l1,l2) = tutu q in (t1::l1,t2::l2);;
   ```
2. On rappelle que la fonction @ $:' alist- >' alist- >' alist$ concatène deux listes.
   ```
   let rec toto l = match l with
   | [] -> []
   | t1::q -> t1 @ (toto q);;
   ```

Exercice 46 : Graphique (Examen Janvier 2012)

En considérant le module `Geo` utilisé en TP et lors du projet, indiquez ce que font les fonctions suivantes :

1. (5 points) La fonction `s` est définie par :
   ```
   let s p c =
      let p1,p2,p3,p4 =
         {x = p.x; y = p.y},{x = p.x+.c; y = p.y},
   ```

1. `fst` et `snd` correspondent respectivement aux fonctions de première et seconde projection d'un tuple.
   ```

```
{x = p.x+.c; y = p.y+.c},{x = p.x; y = p.y+.c} in
[Line(p1,p2);Line(p2,p3);Line(p3,p4);Line(p4,p1)]
```

Le type de `s` est :

```
val s : Geo.point -> float -> Geo.surface list = <fun>
```

Décrivez ce que fait cette fonction.

2. (6 points) La fonction `spc` utilise la fonction `s` et est définie par :

```
let rec spc pin c p i r =
    if i = 0 then (s pin c)@r
             else spc (pin +| p) (c-. 2. *. p.x ) p (i-1) ( (s pin c)@r)
```

Le type de `spc` est :

```
val s : Geo.point -> float -> Geo.point -> int -> Geo.surface  list list -> Geo.surface l
```

Nous rappelons que `+|` est l'opérateur d'addition de deux points.

3. (4 points) Que trace la commande suivante ?

```
Aff.draw "Courbe examen" ( spc {x=0.1;y=0.1} 0.8  {x = 0.1; y=0.1} 5 [])
```

1.7 Évaluation

Exercice 47 : Évaluation

1. Dans chacun des trois cas suivants, dire ce que vaut y après avoir évalué les définitions données :

(a)

let $x = 5$
let $y = x + 3$
let $x = 7$

(b)

let $a = 5$
let $f =$ fun $x \rightarrow x + a$
let $a = 10$
let $y = f\ a$

(c)

let $a = 5$
let $f =$ fun $x \rightarrow x + a$
let $a = a + 1$
let $y = f\ a$

2. Simplifier l'expression suivante :

   $\mathsf{let}\ y\ = \mathsf{let}\ x\ =\ x\ +\ 1\ \mathsf{in}\ x$

3. Indiquer si l'expression suivante est correcte et si oui, la simplifier :

   $\mathsf{let}\ y\ = \mathsf{let}\ x\ =\ x\ >\ 1\ \mathsf{in}\ x$

4. Indiquer si l'expression suivante est correcte et si oui, la simplifier :

   $\mathsf{let}\ y\ = \mathsf{let}\ x\ =\ x\ =\ 1\ \mathsf{in}\ x$

5. Pour chacune des trois expressions précédentes qui est correcte, donner la valeur obtenue pour y dans un environnement où x est lié à 100.

Exercice 48 : Évaluation avec des fonctions

Dans chacun des cas suivants, indiquer si l'expression est correcte et si oui, donner sa valeur :

1. $\mathsf{let}\ a = 0\ \mathsf{in}\ \mathsf{let}\ a = 1\ \mathsf{in}\ \mathsf{let}\ f\ = \mathsf{fun}\ x\ \rightarrow\ a + x\ \mathsf{in}\ f\ 2$
2. $\mathsf{let}\ f\ = \mathsf{fun}\ x\ \rightarrow\ \mathsf{let}\ a = 1\ \mathsf{in}\ a + x\ \mathsf{in}\ f\ a$
3. $\mathsf{let}\ a = 1\ \mathsf{in}\ \mathsf{let}\ f\ = \mathsf{fun}\ x\ \rightarrow\ a + x\ \mathsf{in}\ \mathsf{let}\ a = 2\ \mathsf{in}\ f\ a$
4. $\mathsf{let}\ g\ = \mathsf{fun}\ x\ \rightarrow\ 2 + (f\ x)\ \mathsf{in}\ \mathsf{let}\ f\ = \mathsf{fun}\ x\ \rightarrow\ 1 + x\ \mathsf{in}\ g\ 0$
5. $\mathsf{let}\ f\ = \mathsf{fun}\ x\ \rightarrow\ 1 + x\ \mathsf{in}\ \mathsf{let}\ g\ = \mathsf{fun}\ x\ \rightarrow\ 2 + (f\ x)\ \mathsf{in}\ \mathsf{let}\ f\ = \mathsf{fun}\ x\ \rightarrow\ 4 + x\ \mathsf{in}\ g\ 0$
6. $\mathsf{let\ rec}\ f\ = \mathsf{fun}\ x\ \rightarrow\ 1\ +\ (f\ (x - 1))\ \mathsf{in}\ f\ 3$
7. $\mathsf{let\ rec}\ f\ = \mathsf{fun}\ x\ \rightarrow\ \mathsf{if}\ x = 0\ \mathsf{then}\ 0\ \mathsf{else}\ 1\ +\ (f\ (x - 1))\ \mathsf{in}$
   $\mathsf{let}\ g\ = \mathsf{fun}\ x\ \rightarrow\ f\ (x + 1)\ \mathsf{in}\ g\ 3$

2 Preuves de programme

Exercice 49 : Correction de minimum

Reprendre le code de la fonction qui calcule le minimum d'une liste et montrer qu'il est correct.

Exercice 50 : Duplication et longueur

Montrer la propriété suivante pour les fonctions `duplique` et `longueur` écrites précédemment.

$$\forall l, 2 \times longueur\, l = longueur(duplique\, l)$$

Exercice 51 : Renversement et longueur

Montrer la propriété suivante pour les fonctions `renverse` et `longueur` écrites précédemment.

$$\forall l, longueur\, l = longueur(renverse\, l)$$

Exercice 52 : Concaténation de deux listes

Propriétés algébriques de la concaténation

- Rappeler la fonction concaténation du TD1.
- Démontrer que [ ] est un élément neutre, c'est-à-dire : $\forall l,\ [\,] \,@\, l = l = l \,@\, [\,]$.
- Démontrer que la concaténation est associative, c.-à-d. : $\forall l_1 l_2 l_3,\ (l_1 \,@\, l_2) \,@\, l_3 \;=\; l_1 \,@\, (l_2 \,@\, l_3)$.
  Indication : l_2 et l_3 étant fixés arbitrairement, procéder par récurrence structurelle sur l_1.

Exercice 53 : Renversement et concaténation

Démontrer : $\forall u\, v, renverse(u \,@\, v) = renverse\, v \,@\, renverse\, u$.

Exercice 54 : Renversement du renversement

Démontrer : $\forall u, renverse\,(renverse\, u) = u$.

Exercice 55 : Renversement, version efficace (**)

On propose la fonction suivante pour renverse une liste.

```
let rec renv_aux u l = match l with
  | [] → u
  | x :: q → renv_aux (x :: u) q
let renv_bis l = renv_aux [] l
```

Pour quelles raisons est-elle plus efficace que la fonction `renverse` écrite précédemment ?
Montrer que cette fonction produit le même résultat que celle écrite précédemment, c'est-à-dire que

$$\forall l, renv_aux\ [\,]\ l = renverse\, l$$

Exercice 56 : Listes et arbres binaires

On se donne un type d'arbres binaires et deux fonctions construisant la liste des feuilles d'un arbre binaire, une version naïve et une version plus efficace.

```
type α ab = F of α | N of α ab × α ab

let rec to_list a = match a with
  | F (x) → [x]
  | N (g, d) → to_list g @ to_list d

let rec tle a u = match a with
  | F (x) → x :: u
  | N (g, d) → tle g (tle d u)
```

Indiquer pourquoi la première version est peu efficace et démontrer l'équivalence entre ces deux versions. Plus précisément, démontrer :

$$\forall a\, u, tle\, a\, u = to_liste\, a \,@\, u$$

Exercice 57 : Division Euclidienne

— Rappeler la fonction euclide du TD1.
— Pourquoi calcule-t-elle le bon résultat ?
— Pourquoi se termine-t-elle ?

Exercice 58 : Preuves (Examen 2010 16 points)

Soit la fonction f de type $\alpha\ list \longrightarrow \alpha\ list$ définie par l'induction suivante :
— $f([\,]) = [\,]$
— $f(x :: [\,]) = [\,]$
— $f(x :: y :: l) = x :: (f\ l)$
Démontrer que pour toute liste l, $|f(l)| \leqslant \frac{|l|}{2}$.
(Rappel : $|l|$ et $|f(l)|$ désignent les longueurs des listes l et $f(l)$).

3 Tris de liste

On cherche à ranger les éléments d'une liste par ordre croissant.

3.1 Tri par insertion

Le principe du tri par insertion est d'insérer à leur place les éléments de la liste à trier dans une liste initialement vide.
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

— Écrire une fonction qui, partant d'une liste triée l et d'un élément e, rend une liste triée comportant tous les éléments de l ainsi que e.
— Écrire une fonction qui rend la liste des élements d'une liste triés par ordre croissant, suivant la technique du tri par insertion.
— Pourquoi la fonction de tri par insertion se termine-t-elle ?
— Combien faut-il de comparaisons pour trier une liste de taille n dans le meilleur des cas ? Donner un exemple de liste pour laquelle l'algorithme utilise ce nombre de comparaisons pour $n = 4$.
— Combien faut-il de comparaisons pour trier une liste de taille n dans le pire des cas ? Donner un exemple de liste pour laquelle l'algorithme utilise ce nombre de comparaisons pour $n = 4$.

3.2 Tri rapide (*Quicksort*)

Le tri rapide est un tri ayant de bonnes performances en moyenne. C'est une application typique du principe « diviser pour régner » :

1. On choisit un élément de la liste initiale, que l'on appelle le pivot ;
2. On extrait de la liste initiale deux sous-listes, la première contenant les éléments strictement plus petits que le pivot et la deuxième tous les autres ;
3. On obtient récursivement deux listes triées à partir de ces deux sous-listes ;
4. On les ré-assemble en une unique liste triée.

Exercice 59 : Implémentation du tri rapide

— Écrire une fonction qui produit les deux sous-listes d'une liste à partir d'un pivot donné en argument.
— Écrire une fonction qui trie une liste selon l'algorithme du tri rapide. On prendra le premier élément de la liste comme pivot.

Exercice 60 : Preuve et complexité du tri rapide (*)

— Prouver par récurrence que la fonction de tri rapide trie correctement l'entrée (il faut prouver que la liste de sortie contient les même éléments que la liste d'entrée, et que la liste de sortie est triée).
— Détailler l'exécution du tri rapide sur l'entrée `[5; 3; 1; 2]`. Combien de comparaisons entre entiers sont effectuées en tout ?
— Pour une liste en entrée de taille n, combien fait-on de comparaisons en tout dans le pire des cas ?
— Si on suppose que l'entrée est de taille n, et que tous les découpages pour tous les appels récursifs et le premier niveau coupent la liste en deux parties de même taille, combien fait-on de comparaisons en tout ?

Pour comprendre pourquoi le tri rapide porte ce nom, on admet que la complexité asymptotique trouvée pour cette dernière question correspond aussi à la complexité en moyenne de l'algorithme.

3.3 Tri crêpe (Examen 2010)

Exercice 61 : Ordonner une pile de crêpe (Examen 48 points)

Dans la suite vous pouvez utiliser les fonctions du module standard `List`.

1. (5 points) Écrire une fonction `split` qui étant donné une liste l et un entier n rend deux listes dont la première contient les n premiers éléments de la liste l et la seconde les autres éléments de l. Les éléments de chaque liste résultat devront apparaître dans le même ordre que dans la liste initiale [2].
2. (3 points) En utilisant `split` et `inverse` donner une fonction `retourne` qui prend une liste l et un entier n et rend une liste dans laquelle uniquement les n premiers éléments de l sont inversés [3].
3. (5 points) Écrire une fonction `rangMax` qui prend une liste l et renvoie un couple constitué de la position de l'élément maximum de l et de la valeur de cet élément maximum. Noter que nous renverrons $(0, 0)$ pour la liste vide.
4. (5 points) En utilisant les fonctions déjà construites, écrire une fonction `plusGrandElement` qui prend un entier k et une liste l et renvoie le rang du maximum dans la partie de la liste des k premiers éléments.

   Application : Nous cherchons à trier une pile de crêpes de tailles différentes. Notre pile peut être représentée par une liste d'éléments. Chaque crêpe est représentée par une entier, si nous avons k crêpes, la crêpe la plus petite est l'entier 1 et la plus grande l'entier k.

5. (10 points) Écrire une fonction `lecture` qui lit un char Stream contenant tous les nombres entre 1 et k séparés par une espace. Cette liste repr crêpes. Vous devrez gérer les espaces et les retours à la ligne superflus.

2. Vous pouvez utiliser `List.tl` et `List.hd`.
3. Vous pouvez utiliser `@`.

6. (10 points*) Nous souhaitons trier la pile de crêpes, et notre seule opération possible (outre l'inspection visuelle du tas de crêpes) est de retourner avec une spatule les n premières crêpes de la pile, pour un n de notre choix.
   Nous avons tous les éléments pour pouvoir ordonner notre pile de crêpes. Écrire une fonction qui ordonne une pile en utilisant `retourner` et `plusGrandElement`.
   L'idée étant de d'abord amener le plus grand élément de la liste au sommet de pile, ensuite de retourner l'ensemble de la pile ainsi modifiée et positionner de fait le plus grand élément à la base.
7. (10 points) Écrire une fonction qui prend une liste et renvoie vraie si tous les nombres de la liste sont présents uniquement une fois dans la liste.
8. BONUS (10 points) Écrire une fonction qui prend une liste d'entier où tous les nombres sont uniques et renvoie vrai si tous les nombres de 1 à la taille de la liste sont présents dans la liste.

4 Exceptions

Exercice 62 : Lever une exception

Définir une fonction OCaml `trouve` qui prend en arguments un prédicat p et une liste l, et qui :
- renvoie le premier élément de l qui vérifie la propriété p ;
- si aucun élément ne convient, lève une exception que vous aurez définie préalablement.

Exercice 63 : Attraper une exception

Utiliser la fonction précédente pour définir une fonction `trouver_bis` qui renvoie :
- une liste contenant uniquement le premier élément de l qui vérifie la propriété p.
- une liste vide si aucun élément ne convient.

Exercice 64 : Couplage de listes

Écrire une fonction `zip` qui reçoit deux listes `[a1 ; ... ; an]` et `[b1 ; ... ; bn]` en arguments et renvoie la liste des couples `[(a1,b1) ; ... ; (an;bn)]`.
En quoi cette fonction fait-elle usage des exceptions ?

Exercice 65 : Arbre équilibré

Un arbre binaire est dit *équilibré* si, pour chaque nœud, les fils gauche et droit sont des arbres de même hauteur.

1. Version naïve
   (a) Écrire une fonction `hauteur : arbre -> int` qui renvoie la hauteur d'un arbre binaire.
   (b) Écrire une fonction `equilibre : arbre -> bool` qui vérifie si un arbre binaire est équilibré à l'aide de `hauteur`.
   (c) Pourquoi la fonction précédente n'est-elle pas efficace ?
2. Version améliorée
   (a) On définit une exception `Desequilibre`, que l'on utilise pour programmer une nouvelle fonction `equi_hauteur : arbre -> int`.
   Cette fonction renvoie la hauteur d'un arbre équilibré, et lève l'exception `Desequilibre` si elle détecte un déséquilibre dans l'arbre. Cette exception doit être levée dès que possible
   (b) Définir enfin une fonction `equilibre_rapide : arbre -> bool` qui utilise la précédente et renvoie un booléen indiquant si l'arbre reçu en paramètre est équilibré ou non.

5 Structures arborescentes

5.1 Arbres binaires

Exercice 66 : Fonctions simples sur un arbre

- Définir un type `abin` pour un arbre binaire d'entiers.
- Écrire une fonction qui calcule le nombre de nœuds d'un arbre binaire.
- Écrire une fonction qui calcule le nombre de nœuds binaires "vrais" d'un arbre binaire.
- Écrire une fonction qui calcule la hauteur d'un arbre binaire.
- Écrire une fonction qui calcule le produit de tous les éléments d'un arbre binaire.
- Écrire une fonction qui calcule la somme de tous les éléments d'un arbre binaire.
- Écrire une fonction qui détermine si un élément appartient à un arbre binaire.
- Écrire une fonction qui calcule le maximum d'un arbre binaire.
- Écrire une fonction qui calcule le nombre de nœuds binaires "vrais" d'un arbre binaire (autrement dit le nombre de nœuds ayant 2 fils non vides).

Exercice 67 : Arbres et listes

- Écrire une fonction qui transforme un arbre binaire en une liste de ses éléments selon l'ordre infixe : les éléments du sous-arbre gauche d'abord, la racine au "milieu" et enfin les éléments du sous-arbre droit.
- Réécrire cette fonction afin de construire la liste correspondant au parcours infixe mais de droite à gauche.
- Réécrire cette fonction afin d'afficher la liste dans l'ordre de parcours préfixe : la racine est traitée en premier, puis les sous-arbres gauche et droit.

Exercice 68 : Preuve sur les arbres (Examen 2010 15 points)

Nous considérons les arbres binaires d'entiers de type :

```
type abr = F | N of abr * int *  abr
```

1. (2 points) Écrire une fonction `taille` qui prend un arbre binaire et rend le nombre de nœuds de cet arbre, sachant qu'une feuille a une taille nulle.
2. (3 points) Écrire une fonction `double` qui prend un arbre binaire a et rend un arbre de même structure que a dont la valeur de chaque nœud est le double de la valeur du nœud correspondant de a.
3. (10 points) Prouver que pour tout arbre binaire a : `taille(double a) = taille a`

Exercice 69 : Devoir maison 2009

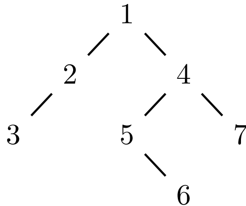
Les arbres sont une structure de donnée classique en algorithmique dont il existe de nombreuses variantes. Nous allons ici nous intéresser aux arbres binaires **non étiquetés** : un arbre est soit une feuille, soit un nœud auquel on associe deux arbres appelés fils. Les nœuds pères sont reliés à leurs nœuds fils par une arête. La racine de l'arbre est l'unique nœud ne possédant pas de parent. La hauteur d'un arbre est la plus grande distance en nombre d'arêtes de la racine à une feuille. On dit qu'un arbre de hauteur h est complet si et seulement si toutes ses feuilles sont à distance h de la racine.

- Définir le type `arbreNonEtiquete`.
- Écrire une fonction `hauteur` qui calcule la hauteur d'un arbre non étiqueté.
- Écrire une fonction `estcomplet` qui utilise `hauteur` pour tester si un arbre non étiqueté est complet.

— Démontrer que pour tout arbre a, `estcomplet a = eqht (hauteur a) a`, où `eqht` est la fonction qui prend en argument un entier et un arbre, et qui rend un booléen, définie comme suit.

```
let rec eqht h a  =  match a with
    | F ⟶  h=0
    | N(g, d) ⟶  eqht (h-1) g ∧ eqht (h-1) d
```

Exercice 70 : Niveau d'un arbre



Le niveau d'un nœud e dans un arbre est le nombre de nœuds sur la branche qui conduit de la racine de l'arbre jusqu'au nœud e inclus. La racine est donc de niveau 1.
Soit e un nœud de niveau $n > 1$ dans un arbre $Ab(g, r, d)$ alors il se situe soit dans g, soit dans d. Le nœud e est de niveau $n - 1$ dans le sous-arbre (g ou d) auquel il appartient.

— Définir la fonction nbf_deNiv qui donne le nombre de feuilles à un niveau donné dans un arbre.
— Définir la fonction nivElt qui donne le niveau d'un élément dans un arbre. On conviendra que le niveau d'un élément qui n'est pas présent dans l'arbre est 0.

5.2 Arbres binaire de recherche

Un arbre binaire a est un arbre binaire de recherche si, pour tout nœud s de a, les contenus des nœuds du sous-arbre gauche de s sont strictement inférieurs au contenu de s, et que les contenus des nœuds du sous-arbre droit de s sont strictement supérieurs au contenu de s.

Exercice 71 : recherche

Écrire la fonction *mem* qui recherche si un élément donné appartient à un arbre binaire de recherche donné. Justifier sa correction et sa complétude.

Exercice 72 : insertion

Écrire une fonction qui ajoute un élément donné à un arbre binaire de recherche donné tout en garantissant que l'arbre reste un arbre binaire de recherche. Justifier.

Exercice 73 : suppression

Différentes stratégies peuvent être suivies. Écrire la fonction *fusion_sup* qui fusionne deux arbres binaire de recherche donnés a et b. On suppose que les éléments de a sont inférieurs à ceux de b.
Écrire la fonction qui supprime un élément donné dans un arbre binaire de recherche donné tout en garantissant que l'arbre reste un arbre binaire de recherche.

Exercice 74 : vérification d'ABR

Écrire une fonction qui vérifie si un arbre donné est bien un arbre binaire de recherche. Différentes stratégies sont possibles.

Exercice 75 : équivalence

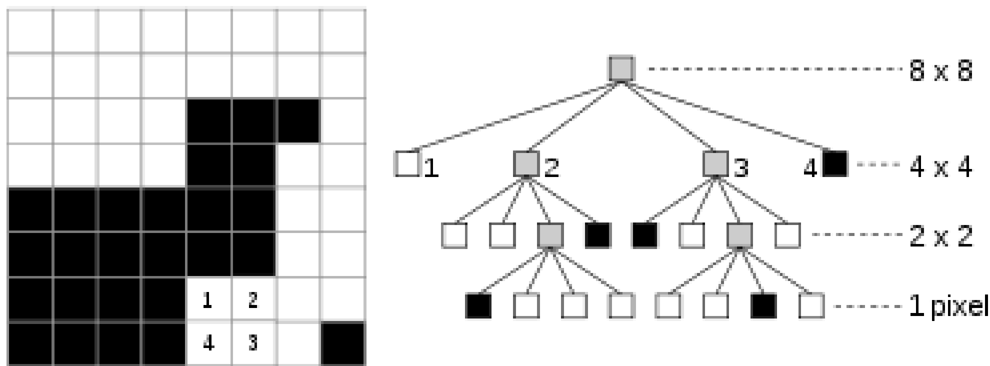
On dit que 2 arbres binaires de recherche sont équivalents s'ils possèdent les mêmes étiquettes. Écrire une fonction qui renvoie `true` si et seulement si deux ABR sont équivalents.

— Écrire une fonction qui prend un arbre binaire de recherche et renvoie son minimum.
— Écrire une fonction qui prend un arbre binaire de recherche et renvoie son deuxième plus grand élément.
— Écrire une fonction qui prend un arbre binaire de recherche et calcule la médiane de ses éléments.

Remarque : Il existe de nombreux autres types d'arbres comme par exemple les arbres AVL (Adelson-Velskii et Landis) : un AVL est un ABR pour lequel, en tout nœud, les hauteurs des sous-arbres gauche et droit diffèrent au plus de 1.

Exercice 76 : Quadtrees et compression d'image (Examen 2010 30 points)

On présente ici une représentation d'images sous forme d'arbres. Cette représentation donne une m et facilite certaines opérations sur les images.
Pour simplifier, on suppose les images carrées, de côté 2^n, et en noir et blanc. L'idée est la suivante : une image toute blanche ou toute noire se représente par sa couleur, tandis qu'une image composite se divise naturellement en quatre images carrées.



Nous considérons les types suivants :

```
type couleur = Blanc | Noir

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

1. (5 points) Écrire une fonction `inverse` qui prend un quadtree a représentant une image i et renvoie un quadtree représentant l'image i' obtenue à partir de i en échangeant noir et blanc.
2. (5 points) Écrire une fonction `rotate` qui prend un quadtree a représentant une image i et renvoie un quadtree représentant l'image i tournée d'un quart de tour vers la gauche.

On souhaite pouvoir transformer un quadtree en une liste de 0 et de 1, et réciproquement.
On note `code(a)` la liste de 0 et de 1 représentant un quadtree a. On choisit le codage suivant :

```
code(Feuille Blanc)        = 00
code(Feuille Noir)         = 01
code(Noeud (a1,a2,a3,a4)) = 1 code(a1) code(a2) code(a3) code(a4)
```

Considérons le type suivant

```
type bit = Zero | Un
```

3. (10 points) Écrire une fonction `quadtree_vers_liste` de type `quadtree -> bit list` qui transforme un quadtree en une liste de `bit` selon le codage [4].
4. (10 points *) Écrire une fonction de décodage `liste_vers_quadtree` de type `bit list -> quadtree`, qui transforme une liste de `bit` en le quadtree correspondant.

6 Modules et foncteurs

6.1 Modules

La programmation modulaire permet la décomposition d'un programme en *unités logiques* plus petites, ainsi que la *réutilisation* plus aisée d'unités logiques indépendantes.
En OCaml, la déclaration d'un module suit la syntaxe suivante :

```
module Complex = struct
  type t = float × float
  let zero = (0., 0.)
  let cons r i = (r, i)
  let oppose (r, i) = (−.r, −.i)
  let plus (r_1, i_1) (r_2, i_2) = (r_1 +.r_2, i_1 +.i_2)
  let modu (r, i) = sqrt (r *.r +. i *.i)
end
```

Le module *Complex* définit le type t, la valeur *zero*, ainsi que les fonctions *cons*, *oppose*, *plus* et *modu*.
Il y a ensuite deux manières d'appeler ces fonctions :
- — de manière explicite, par *Complex.zero* ;
- — après l'appel à open *Complex*, de manière implicite par *zero*.

On peut de plus « cacher » la définition de t en forçant le *type module* de *Complex* :

```
module Complex : sig
  type t
  val zero : t
  val cons : float → float → t
  val oppose : t → t
  val plus : t → t → t
  val modu : t → float
end = struct
  type t = float × float
  let zero = (0., 0.)
  let cons r i = (r, i)
  let oppose (r, i) = (−.r, −.i)
  let plus (r_1, i_1) (r_2, i_2) = (r_1 +.r_2, i_1 +.i_2)
  let modu (r, i) = sqrt (r *.r +. i *.i)
end
```

Le type module peut aussi être déclaré séparément, ce qui permet de séparer l'interface de l'implémentation, afin de fournir plusieurs implémentations pour la même interface. Penser par exemple, à

4. vous pouvez utiliser `@`.

la manière dont sont définis les paquetages en Ada : un fichier `.ads` contenant l'*interface* (en OCaml, le *type module*), et un fichier `.adb` contenant l'implémentation.

```
module type TComplex = sig
  type t
  val zero : t
  val cons : float -> float -> t
  val oppose : t -> t
  val plus : t -> t -> t
  val modu : t -> float
end

module Complex : TComplex = struct
  ...
end
```

Exercice 77 : Écriture d'un module de gestion d'une pile

77.1. Définir le type module correspondant à l'interface d'un module fournissant un type polymorphe « pile », ainsi que les fonctions *pile_vide*, *est_vide*, *push* et *pop*.

77.2. Écrire un module implémentant cette interface.

Exercice 78 : Module Exam 2013

On donne la signature d'un module `Urne` permettant de faire un vote à bulletin secret :

- Il y a deux candidats A et B.
- Chaque électeur est doté d'un numéro entier (unique), ce qui permet de s'assurer qu'un même électeur ne peut pas comptabiliser deux votes
- En cours de scrutin, un électeur peut changer d'avis et re-voter (l'ancien vote est alors détruit).
- On peut dépouiller l'urne pour connaître les scores de chaque candidat et déterminer le vainqueur.

Le type `t` est celui de l'urne proprement dite.

```
module type URNE =
sig
  type electeur = int
  type candidat = A | B
  type t (* type de l'urne *)
  val vide : t
  val voter : electeur -> candidat -> t -> t
  val depouiller : t -> int × int
  val vainqueur : t -> candidat
end
```

78.1. Dire pourquoi à l'extérieur du module il n'est possible à aucun moment de savoir qui a déjà voté, ni quel est le vote de quelqu'un.

78.2. Implémenter ce module.

Exercice 79 : Module et foncteur Exam 2014

Le foncteur de type `MEMOIRE`, dont la signature est donnée ci-dessous, spécifie une structure de données, que l'on appelera une **mémoire**, permettant de mémoriser des **éléments** de type quelconque (`'a` en OCaml) à chacun desquels est associée une **clé** (de type `cle`). On suppose que la mémoire *peut contenir plusieurs éléments ayant même clé.* La signature de ce foncteur contient :

- le type `'a mem`, représentant une mémoire ;
- la fonction `memVide` qui renvoie une mémoire vide (ne contenant aucun élément) ;
- la fonction `(estVide m)`, qui vaut vrai si et seulement si `m` est une memoire vide ;
- la fonction `(inserer m e c)` qui renvoie une mémoire `m` à laquelle l'élément `e` de clef `c` a été ajouté ;
- la fonction `(extraire m c)` qui renvoie la liste de tous les éléments de `m` dont la clé est « égale » à celle de `c` (selon la fonction `Cle.egal`).

Ce foncteur `MEMOIRE` est paramétré par un module de type `CLE` dont la signature contient :

- le type `cle` ;
- une fonction d'égalité entre clés, la fonction `egal`.

```
module type CLE =
sig
  type cle
  val egal : cle → cle → bool
end
```

```
module type MEMOIRE = functor (Cle : CLE) →
sig
  type α mem
  val memVide : α mem
  val estVide : α mem → bool
  val inserer : α mem → α → Cle.cle → α mem
  val extraire : α mem → Cle.cle → α list
end
```

79.1. En complétant le code OCaml ci-dessous, donner une première implémentation du module `MEMOIRE` dans laquelle le type `'a mem` est implémenté par une liste de couples (clé, élément).

```
module MEMOIRE_1 : MEMOIRE = functor (Cle : CLE) →
struct
  type α mem = (α × Cle.cle) list
  let memVide = ...
  let estVide m = ...
  ...
end
```

Exemple : Si `m1` est une mémoire contenant les élément `"a"`, `"b"` et `"c"` de clés respectives `2`, `3` et `2` alors `m` sera représenté par la liste `[(2, "a"); (3, "b"); (2, "c")]`.

On pourra *si on le souhaite* utiliser (sans les écrire) les fonctions prédéfinies suivantes sur les listes OCaml :

```
val filter : ('a -> bool) -> 'a list -> 'a list
val split : ('a * 'b) list -> 'a list * 'b list
```

`(filter p l)` renvoie tous les éléments de la liste `l` satisfaisant le predicat `p`.
`split` transforme une liste de couples en un couple de listes : `split [(a1,b1); ...; (an,bn)]` renvoie `([a1; ...; an], [b1; ...; bn])`.

79.2. En complétant le code OCaml ci-dessous, donner une deuxième implémentation du module `MEMOIRE` dans laquelle le type `'a mem` est implémentée par une liste de couples (clé c, liste d'éléments e ayant même clé c).

```
module MEMOIRE_2 : MEMOIRE = functor (Cle : CLE) →
struct
  type α mem = (Cle.cle × (α list)) list
  let memVide = ...
  let estVide m = ...
  ...
end
```

Exemple : Dans cette implémentation, si on suppose que les clés `2` et `3` sont différentes (selon la fonction `Cle.egal`), la mémoire `m1` contenant les éléments `"a"`, `"b"` et `"c"` de clés respectives `2`, `3` et `2` est représentée par la liste `[(2, ["a"; "c"]); (3, ["b"])]`.

79.3. Donner une implémentation du module `CLE` dans laquelle :
- — le type `cle` est le type `int`
- — la fonction `(egal c1 c2)` vaut vrai ssi `c1` et `c2` ont même parité.

Exercice 80 : Module Point Devoir Maison 2010

Le but de ce DM est de manipuler les notions de *signature*, *module* et *foncteur*, à travers l'exemple concret de la géométrie euclidienne.

- — (3 points) Écrire une signature `POINT` qui définit l'interface pour des modules de type point (d'un espace affine). On doit pouvoir calculer la `distance` entre deux points, avoir un type `vecteur` de l'espace vectoriel sous-jacent (dont on pourra calculer la `norme`), et pouvoir faire des additions/soustractions entre points et vecteurs, lorsque cela a un sens.
- — (3 points) Écrire un module `Point2D` réalisant la signature `POINT`. Comme son nom l'indique, nous considérons l'espace affine $\mathbb{R}^2$ (ou plutôt son approximation $\mathbb{F}^2$ représentable en OCaml, où $\mathbb{F}$ est l'ensemble des valeurs accessibles avec le type `float`).
- — (3 points) Écrire un module `Point3D` réalisant la signature `POINT`. On s'intéresse ici à l'espace affine $\mathbb{R}^3$ (ou plutôt son approximation $\mathbb{F}^3$).
- — Nous allons maintenant définir un foncteur `Geometrie` permettant de construire, à partir d'un module respectant la signature `POINT`, un module calculant quelques fonctions de géométrie. Chaque définition composant ce foncteur est demandée dans une question séparée, mais dans votre fichier OCaml tout est regroupé dans une unique définition du foncteur.
  (2 points) Ajouter à votre foncteur une définition du type `triangle`.
- — (2 points) Ajouter à votre foncteur une définition de la fonction `aire_triangle` calculant l'aire d'un `triangle`. Rappel : l'aire d'un triangle ne dépend que de ses longueurs, à vous de chercher une formule adéquate.
- — (2 points) Ajouter à votre foncteur une fonction `plus_proche`, qui pour un point `x` et une liste de points `l` calcule un point de `l` le plus proche de `x`. Évidemment, quelques tests sont de rigueur...
- — (2,5 points) Appliquer votre foncteur `Geometrie` au module `Point2D` pour obtenir le module `Geometrie2D`. Tester toutes les fonctionnalités du module `Geometrie2D` sur quelques exemples.
- — (2,5 points) Appliquer votre foncteur `Geometrie` au module `Point3D` pour obtenir le module `Geometrie3D`. Tester toutes les fonctionnalit exemples.

Exercice 81 : Utilisation d'un module

Le module Random de la bibliothèque standard OCaml propose un générateur de nombres pseudo-aléatoires. Voici un extrait de sa signature :

```
module Random : sig
  val self_init : unit -> unit
  val int : int -> int
end
```

— self_init initialise le générateur,
— int x génère un entier dans $(0, 1, \ldots, x-1)$,

Écrire une fonction qui génère n notes entre 0 et 20.

Exercice 82 : Luhn

Écrire un module Luhn qui définit une fonction clef calculant le nombre de Luhn pour une suite de nombres entiers :

$$Luhn(a_0, a_1, \ldots, a_k) = \sum_{i=0}^{\lfloor k/2 \rfloor} (a_{2i+1}) + \sum_{i=0}^{\lfloor k/2 \rfloor} \sum_{j=0}^{\lfloor \log_{10} a_{2i} \rfloor} (\lfloor \frac{a_{2i}}{10^j} \rfloor \bmod 10)$$

Indications : Chaque somme peut être codée par une fonction récursive. La dernière somme est celle des chiffres de a_{2i} écrit en base 10.

Exercice 83 : Datagramme

Un datagramme est typiquement constitué d'en-têtes et de données :
— numéro du port d'origine
— numéro du port de destination
— longueur des données
— somme de contrôle des données
— données elles-mêmes

Écrire la signature du type module TDatagram qui propose des fonctions pour remplir et lire un datagramme : datagram, orig, dest, length, checksum, data.
Écrire un module Datagram qui implémente cette interface avec, un type entier pour les numéros de port, une liste d'entiers pour les données, et le nombre de Luhn pour la somme de contrôle.

Exercice 84 : Extension

Étendre le module Datagram pour lui ajouter une fonction verify qui vérifie que la cohérence de la somme de contrôle et une fonction reply qui crée un datagramme de réponse.

Exercice 85 : Module (Examen Janvier 2012)

— (4 points) Écrivez une signature Point qui définit l'interface pour des fonctions de la géométrie euclidienne. On doit pouvoir calculer la distance entre deux points et faire des additions entre deux points.
— (4 points) Écrivez un module Point2D réalisant la signature Point représentant $\mathbb{R}^2$ (ou plutôt son approximation $\mathbb{F}^2$ représentable en OCaml, où $\mathbb{F}$ est l'ensemble des valeurs accessibles avec le type float).

6.2 Foncteurs

Les *foncteurs* sont des fonctions du domaine des modules vers le domaine des modules. Ils permettent la définition de modules *paramétrés par un ou plusieurs autres modules.*
Un foncteur est défini par le mot-clé functor, suivi de la signature du module paramètre (ici P), puis de la définition du foncteur en fonction de ce paramètre.

```
module type Groupe = sig
  type t
  val zero : t
  val oppose : t → t
  val plus : t → t → t
end
```

```
module Matrices = functor (P : Groupe) →
  struct
    type t = P.t list list
    let plus = List.map2 (List.map2 P.plus)
  end
```

Le résultat de l'instanciation d'un foncteur avec un module respectant la signature donnée (c'est-à-dire comportant *au moins* les types et valeurs de cette signature — ce module peut être plus complet !) est un module, que l'on peut lui-même nommer, utiliser, ouvrir avec le mot-clé open comme n'importe quel module.

```
module ComplexMat = Matrices(Complex)
```

Exercice 86 : Écriture d'un foncteur de tri

86.1. Donner la définition, sous forme de signature, d'un *type ordonné*, c'est-à-dire d'un type sur lequel on peut effectuer des comparaisons.

86.2. Soit la signature :

```
module type TTri = functor (E : TypeOrdonne) → sig
  val trier : E.t list → E.t list
end
```

Écrire un foncteur *QuickSort* implémentant l'interface *TTri* avec l'algorithme du QuickSort.

86.3. Écrire un foncteur *ABRSort* implémentant l'interface *TTri* par un tri par arbre binaire de recherche.

86.4. Définir, à l'aide d'un de ces foncteurs et en utilisant le module *Complex* vu précédemment, un module permettant de trier des nombres complexes par module croissant.

Exercice 87 : Foncteur de datagramme

87.1. Écrire la signature d'un type module `TChecksum` proposant les fonctions `checksum` - qui calcule la somme de contrôle d'une liste d'entiers - et `equal` qui compare deux sommes de contrôle.

87.2. Écrire des modules implémentant cette signature avec les algorithmes suivants :
— `Index` : somme des éléments, pondérés par leur index.
— `Frac` : division (en flottant) du produit des éléments pairs non nuls par le produit des éléments impairs non nuls.
— `Luhn` : somme de Luhn, définie précédemment.

87.3. Écrire un foncteur `Protocol` proposant de définir un datagramme en fonction d'un module implémentant une somme de contrôle. Définir les trois datagrammes correspondants à l'aide de ce foncteur.

7 Ordre supérieur

On appelle *fonction d'ordre supérieur*, ou *fonctionnelle* une fonction qui prend des fonctions en argument ou qui rend une fonction.
Par exemple, la fonction qui prend en argument deux fonctions et qui retourne la somme de celles-ci s'écrit de la façon suivante en OCaml :

let $somme_fonctions\ f\ g\ =$ fun $x\ \rightarrow\ (f\ x)\ +\ (g\ x)$

Exercice 88 : somme de fonctions

Indiquer deux autres écritures de la fonction *somme_fonctions*.

Exercice 89 : composition de fonctions

Donner la définition d'une fonction qui rend la composée de deux fonctions.

Exercice 90 : Préparation

— Écrire une fonction OCaml qui calcule pour toute fonction f, avec $n \in \mathbb{N}$: $\prod_{i=1}^{n} f(i)$.
— Utiliser la fonction précédente pour définir la fonction factorielle.

Exercice 91 : curryfication

Une fonction à plusieurs arguments $a_1 \dots a_n$ peut se ramener à une fonction à un seul argument de deux manières :

1. une fonction prenant en argument un n-uplet $(a_1 \dots a_n)$;
2. une fonction prenant en argument a_1 et rendant une fonction, prenant elle-même en argument a_2 et rendant une fonction ... prenant elle-même en argument a_n et rendant un résultat dépendant de $a_1 \dots a_n$.

Dans cet exercice on prend $n = 2$.
— Écrire la fonctionnelle *curry* prenant en argument une fonction dans le premier style et rendant sa représentation dans le second.
— Écrire la fonctionnelle *uncurry* effectuant la transformation inverse.
— Vérifier que pour tout f, $curry(uncurry f)$ et f sont extensionnellement

$$\forall xy, curry\ (uncurry\ f)\ x\ y = fx\ y.$$

— Formaliser et démontrer que pour tout f, $uncurry(curry f)$ et f sont extensionnellement

Exercice 92 : liste de fonctions

Notation : la fonction d'addition se note (+) et sous cette forme s'utilise de manière préfixe ; par exemple ((+) 2 1) est une autre écriture de (2 + 1).
Que représente l'expression (+) 2 ?
Écrire une fonction qui prend en argument un entier n et rend une liste des n fonctions qui ajoutent respectivement $n, n-1, \ldots 1$.

Exercice 93 : Combinateurs de liste

On retrouve souvent les mêmes schémas de parcours pour des structures de données telles que listes, arbres binaires, etc. Il est possible, en utilisant l'ordre supérieur (le passage de fonctions en argument) de programmer de tels schémas une fois pour toutes. Les schémas les plus fréquents sont :

- *map*, le morphisme structurel, où l'on recopie une structure de donnée en une structure de même forme, mais où les éléments x sont remplacés par des éléments $f\,x$;
- *fold*, le pliage, consistant à appliquer une opération binaire aux éléments de la structure, tour à tour ;
- *iter*, l'itération, consistant à appliquer une fonction à effet de bord à tous les éléments de la structure ;
- *filter*, le filtre des éléments d'une liste en fonction d'une propriété donnée.

Donner le type puis une définition de *map*, *fold*, et *filter* pour les listes.

Exercice 94 : Utilisation

Choisir et utiliser convenablement les combinateurs précédents pour réaliser les programmes suivants :

- calculer la somme des éléments d'une liste d'entiers ;
- calculer le maximum d'une liste d'entiers ;
- calculer la longueur d'une liste ;
- élever au carré chaque élément d'une liste ;
- calculer la conjonction d'une liste de booléens ;
- renverser une liste ;
- déterminer si une liste d'éléments vérifient tous un prédicat donné ;
- écrire map avec fold ; [***]
- écrire fold avec map ; [**]
- trier une liste.

Exercice 95 : Utilisation pour les arbres

- Écrire une fonction qui double la valeur de chaque nœud d'un arbre binaire.
- Écrire une fonction qui applique une fonction g sur l'ensemble des éléments d'un arbre. (map pour les arbres)
- Récrire la première fonction avec l'aide de la seconde.

Exercice 96 : For all 2

- Écrire une fonction qui prend en paramètre une fonction de test (un prédicat) à deux paramètres p, ainsi que deux listes $[a_1; a_2; ...; a_n]$ et $[b_1; b_2; ...; b_n]$, et calcule $(p\ a_1\ b_1)\&\&(p\ a_2\ b_2)\&\&...\&\&(p\ a_n\ b_n)$. Elle lève une exception si les deux listes sont de tailles différentes.
- Utiliser cette fonction pour définir une fonction qui teste si deux listes sont identiques.

Exercice 97 : Lecture de code

Soit la fonction suivante :

```
# let rec traiteliste init f = function
                  [] -> init
                | e :: l -> f e (traiteliste init f l) ;;
    val traiteliste : 'a -> ('b -> 'a -> 'a) -> 'b list -> 'a = <fun>
```

On ajoute la définition suivante :

```
# let trouve p = traiteliste false (fun e -> fun appel -> (p e) || appel) ;;
```

— Quel est le type de la fonction `trouve` ?
— Que fait la fonction `trouve` ?
— Quel sera le résultat de l'évaluation de l'expression suivante ?
`# trouve (fun x -> x mod 2 = 0) [1;2;3;4;5] ;;`
— Utiliser la fonction `traiteliste` pour définir, sans récursivité, la fonction `forall` qui teste si tous les éléments d'une liste ont une propriété donnée.

Exercice 98 : Lecture de code

Soit la fonction suivante :

```
# let term init f l =
     let rec traite accu = function
           [] -> accu
         | e :: l -> traite (f e accu) l
     in traite init l ;;
```

— Quel est le type de la fonction `term` ?
— Que donnera l'évaluation de l'expression suivante ?
`# term 0 (+) [1;2;3;4] ;;`
— Utiliser la fonction `term` pour définir, sans récursivité, une fonction qui inverse une liste.

Exercice 99 : Reconstruction d'un arbre (Examen Janvier 2012)

Tout graphe peut être parfaitement décrit par une liste de triplets (i, s, v), où chaque triplet est de la forme :
— i, un entier représentant l'identifiant d'un nœud ;
— s, la somme des identités des voisins de ce nœud ;
— v, le nombre de ses voisins.

```
type triplet = int*int*int
```

Cette liste représente complètement un graphe. Ici nous ne considérerons que les arbres binaires. Rappelons que dans un arbre binaire on considère qu'un nœud et son père sont voisins.

1. Liste valide (15 points).
   Une liste est valide si est seulement si elle vérifie les 4 critères suivants. Pour chaque critère écrire une fonction qui vérifie qu'il est vérifié. Dans chaque cas vous préciserez le type de vos fonctions et leur complexité pour une taille de liste de n éléments (sans justification).
   (a) (2 points) Tous les nœuds ont des identités supérieurs ou égaux à 1. Rappel donner la complexité et le type de cette fonction.

(b) (4 points) La somme des nombres de voisins de tous les triplets est inférieure au carré du nombre de nœuds. Rappel donner la complexité et le type de cette fonction.

(c) (5 points) Il n'existe qu'un seul triplet pour chaque identifiant de nœud. Autrement dit, il n'y a pas de redondance dans la liste. Rappel donner la complexité et le type de cette fonction.

(d) (6 points) Pour une liste de taille n la liste contient exactement tous les identifiants de 1 à n, c'est à dire une seule fois. Rappel donner la complexité et le type de cette fonction.

Nous considérons les arbres binaires de type :

```
type abin = F | N of abin* int*abin
```

2. Exemple (2 points) :
   Nous considérons la liste suivante représentant un graphe avec 5 nœuds :

   ```
   [(1,2,1);(3,4,1);(5,2,1);(2,10,3);(4,5,2)]
   ```

   Donnez un arbre associé à cette liste :
3. Élimination des triplets inutiles (8 points) :
   — (2 points) Écrire une fonction qui étant donné une liste de triplets, enlève les triplets ayant 0 voisin. Donner son type et sa complexité.
   — (6 points) Réécrivez cette fonction en utilisant une fonction d'ordre supérieur. Sachant que :
     — `val map : ('a -> 'b) -> 'a list -> 'b list`
       List.map f [a1 ; ... ; an] applique la fonction f à a1, ..., an, et construit la liste [f a1 ; ... ; f an] avec les résultats retourn
     — `val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
       List.fold_left f a [b1 ; ... ; bn] est f (... (f (f a b1) b2) ...) bn.
     — `val filter : ('a -> bool) -> 'a list -> 'a list`
       filter p l retourne tous les éléments de la liste l qui satisfont le prédicat p. L'ordre des éléments dans la liste donnée est préservé.
     — `val iter : ('a -> unit) -> 'a list -> unit`
       List.iter f [a1 ; ... ; an] applique la fonction f successivement à a1 ; ... ; an. C'est équivalent à begin f a1 ; f a2 ; ... ; f an ; () end.
4. Construction d'un arbre *** (10 points) :
   Écrivez une fonction qui prend une liste valide et renvoie l'arbre binaire associé, sachant que la liste code bien un arbre binaire.

8 Parseurs et flots

Exercice 100 : Lecture d'un entier

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.
Rappel : $1664 = (((1 \times 10 + 6) \times 10) + 6) \times 10 + 4$.

Exercice 101 : Lecture d'un fichier : Devoir Maison 2009

Nous cherchons à analyser un fichier contenant des triplets :

`(x0,y0,z0) (x1,y1,z1) (x2,y2,z2) ...`

où `x,y,z` sont des nombres entiers représentant les coordonnées d'un point dans l'espace.

1. Donner un type `token` représentant un lexème pour le format de fichier à lire, ainsi qu'un type `point`.
2. Écrire une fonction `analex` de type `char Stream.t -> token Stream.t` réalisant l'analyse lexicale d'un fichier.
3. Écrire une fonction `anasyn` de type `token Stream.t -> point list` qui donne la liste des points lus dans le fichier.
4. Écrire une fonction calculant l'isobarycentre d'une liste de points.
5. Écrire une fonction `bary_single_pass` de type `token Stream.t -> point` qui calcule l'isobarycentre des points lus en espace mémoire réduit (en ne stockant pas la liste des points dans une liste intermédiaire).

Remarques :

— L'isobarycentre est le point défini par

$$x = \frac{1}{n+1}\sum_{i=0}^{n} x_i,\ y = \frac{1}{n+1}\sum_{i=0}^{n} y_i,\ z = \frac{1}{n+1}\sum_{i=0}^{n} z_i.$$

— Pour obtenir un flot de caractères depuis un fichier, vous pouvez utiliser la fonction suivante.

```
let stream_from_file name = Stream.of_channel (open_in name)
val stream_from_file : string -> char Stream.t
```

— Les différents éléments lexicaux peuvent être séparés par des espaces ou retours à la ligne.

— La fonction `float_of_int` permet de convertir en nombre entier en un flottant.

Exercice 102 : 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 :

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

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.

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.
3. 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.
4. Écrire une fonction `joue_coup` qui calcule un nouvel état à partir d'un état du plateau et d'un coup.
5. (**) 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.

9 Aspects impératifs

9.1 Programmation impérative

Exercice 103 : Somme des impairs (Examen Janvier 2012)

Nous donnons deux versions d'une fonction récursive (non terminale) `sumimp1` et `sumimp2` qui prend en argument un entier n et calcule la somme des n premier entiers impairs. Par exemple pour $n = 3$ nous obtenons $1 + 3 + 5 = 9$

```
let rec simp1 n = match n with
    | 0 → 0
    | n → 2 × (n − 1) + 1 + simp1 (n − 1)
let rec simp2 n = match n with
    | 0 → 0
    | n → 2 × n − 1 + simp2 (n − 1)
```

1. (1 point) Prouvez que ces algorithmes terminent.
2. (6 points) Prouvez par une preuve par récurrence que ces deux algorithmes sont corrects. C'est à dire qu'ils calculent bien n^2.
3. (4 points) Écrivez une version récursive terminale d'une des ces deux fonctions.
4. (3 points) Écrivez une version de cette fonction en utilisant une boucle `while`.
5. (3 points) Écrivez une version de cette fonction en utilisant une boucle `for`.

9.2 Référence

Exercice 104 : Référence

— Écrire une fonction qui prend deux arguments, une chaîne et un entier et affiche la chaîne suivie de l'entier.

— Quel est le type de cette fonction ?
— Quel est la valeur de retour ?

Exercice 105 : Factorielle

Réécrire la fonction factorielle dans un style impératif. Comparer l'utilisation de la mémoire avec la version fonctionnelle naïve.

Exercice 106 : Objet

On peut définir un objet par une fonction qui peut accéder et modifier un état. Définir un tel objet `point2d` comme une fonction qui prend un message et renvoie un entier dépendant du message et de l'état interne.
Nous donnons le type message :
`type message = Get_x | Get_y | Set_x of int | Set_y of int`
Définir une fonction `point_constr` qui prend deux entiers en arguments et construit un objet point initialisé avec ces deux valeurs. Quel est le type de cette fonction ?

Exercice 107 : Compteur Devoir Maison 2009

1. Écrire une fonction `compteur` de type : `unit -> int` qui renvoie un au premier appel, puis à chaque appel retourne la valeur précédente plus un.
2. Écrire une fonction `genere_name` qui génère un nouveau nom de variable à chaque appel, par exemple $x1, x2, x3$...
3. Proposer une second fonction qui permet de remettre à zéro le générateur de nom de variable.

9.3 Tableaux

Exercice 108 : Type Array

On veut écrire une fonction `reverse` qui prend un tableau en entrée et inverse l'ordre de ses éléments.
— Quel doit être le type de cette fonction ?
— Écrire cette fonction.
— Comment la tester sur le tableau `[|1; 2; 3|]` ?
— Quel est l'espace mémoire utilisé par cette fonction ?
— Comparer avec la fonction `reverse` sur les listes.

Exercice 109 : Type Array et listes

Écrire les fonctions suivantes (présentes dans la bibliothèque Array) :
— `array_map : ('a -> 'b) -> 'a array -> 'b array` applique une fonction f à tous les éléments d'un tableau a, et construit le tableau $[|fa.(0); fa.(1); ...|]$.
— `array_to_list : 'a array -> 'a list` renvoie la liste des éléments de a.
— `array_of_list : 'a list -> 'a array` renvoie un nouveau tableau contenant les éléments de l.

Exercice 110 : Tri par insertion sur des tableaux

Écrire une fonction `tri` de type `'a array -> unit` qui trie un tableau suivant l'algorithme de tri par insertion.

Exercice 111 : Module matrice d'entier avec tableaux

Implanter la signature suivante à l'aide de tableaux :

```
module type Mat =
 sig
   type t
   val make n -> t
   val get : t -> i -> j -> int
   val set : t -> i -> j -> n -> unit
   val to_string : t -> string
 end
```

10 λ-Calcul

Les langages fonctionnels, comme OCaml, sont directement basés sur les concepts du λ-calcul, inventé par A. Church en 1930. Le λ-calcul fournit une notation condensée pour la fonction notée fun $x \rightarrow U$ en OCaml : $\lambda x.U$. On dit que la variable x est liée et que l'expression U est la portée de cette liaison.

Construction des termes du λ-calcul :

1. une variable x est un λ-terme.
2. abstraction : si x est une variable, et U un λ-terme, alors $\lambda x.U$ est un λ-terme.
3. application : si U et V sont des λ-termes, alors (UV) est un λ-terme.

Convention de simplification d'écriture :

1. Associativité à gauche de l'application :
   $(((uv)(wt)s)$ s'abrège en $uv(wt)s$
2. Associativité à droite de l'abstraction :
   $\lambda x.(\lambda y.(\lambda z.U))$ s'abrège en $\lambda x.\lambda y.\lambda z.U$
3. Regroupement des λ : $\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.U))$ s'abrège en $\lambda x_1 x_2 \ldots x_n.U$

α-conversion : renommer toutes les occurrences d'une variable liée dans la portée de sa liaison. Ex : $\lambda y.((xz)(\lambda x.xy)) \equiv_\alpha \lambda y.((xz)(\lambda s.sy))$
β-réduction : $(\lambda x.U)V \rightarrow_\beta U[V/x]$, où $U[V/x]$ dénote la substitution des occurrences libres de x par V dans U. Attention, les variables libres de V ne doivent pas être capturées dans U (appliquer éventuellement une α conversion dans $\lambda x.U$ si nécessaire).

Il est remarquable qu'à partir de ces seules règles, qui représentent essentiellement le passage de paramètres, il s'avère possible de représenter n'importe quelle fonction calculable. Ce qui suit a pour objectif de montrer quelques codages.

Exercice 112 : Beta-réductions (Examen Janvier 2012)

Nous rappelons les combinateurs standards :

$$S := \lambda xyz.(xz(yz)).$$

$$K := \lambda ab.a$$

$$I := \lambda e.e$$

1. (3 points) Vérifiez que $I = (SKK)$
2. (7 points) β-réduire $SI(S(KK)Iu)v$.

Exercice 113 : Booléens et fonctions booléennes

On rappelle que les booléens sont définis par : $V = \lambda xy.x$ et $F = \lambda xy.y$.

1. (3 points) β-réduire les termes suivants.
   — F F V
   — V F V
   — F V V
2. (4 points) Donner un terme représentant la fonction $ifthenelse$.
3. (10 points) Donner les termes représentant les fonctions booléennes suivantes :
   — *not*
   — *and*
   — *or*
   — *nand*
   — *nor*
   Nous rappelons les tables de vérité du nor et du nand.

x	y	x nor y	x nand y
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	0

Entiers de Church

On veut représenter les entiers naturels dans le λ-calcul. Il existe plusieurs codages, mais le plus simple est dû à Church. Les entiers de Church peuvent être définis comme des opérateurs sur les fonctions. Par exemple, 3 est l'opérateur $\lambda fx.f(f(fx))$ que nous pourrons abréger en $\lambda f.f^3$, où f^3 est une abréviation pour trois fois l'application de f. L'entier de Church C_n qui représente l'entier n est donc la fonctionnelle qui prend une fonction f et un argument x, et qui retourne f composée n fois appliquée à x.

Exercice 114 : Entiers de Church

— Écrire, en λ-calcul et en OCaml, les entiers de Church C_0, C_1, C_2, C_3 et C_n.
— Calculer $C_n f x$
— Écrire la fonction *successeur* qui, appliquée à C_n, rend C_{n+1}.
— Réduire *successeur* C_2.

Exercice 115 : arithmétique sur les entiers de Church

— Réduire $C_n f$
— Réduire $C_m f x$
— Écrire, en λ-calcul et en OCaml, la fonction *plus* adaptée aux entiers de Church.
— Vérifier que $plus C_n C_m = C_{n+m}$
— Écrire, en λ-calcul et en OCaml, la fonction *mult* adaptée aux entiers de Church.
— Vérifier que $mult C_n C_m = C_{n*m}$
— Réduire $C_1 C_2$ et $C_2 C_1$.
— Écrire, en λ-calcul et en OCaml, la fonction *exp* (exponentielle) adaptée aux entiers de Church. On veut que *exp* C_n C_m s'évalue en $C_{(n^m)}$.

— Vérifier que $expC_nC_m = C_{n^m}$
Indications : nf est f^n ; $a^{b+c} = a^b a^c$, $a^{bc} = (a^b)^c$.

Exercice 116 : conversions

Écrire, en OCaml, les deux fonctions de conversions entre entiers de Church et entiers naturels.
Dans la suite, on aura besoin d'un codage des couples, il nous faut donc une repr d'introduction et d'élimination, qui sont ici respectivement le constructeur de couples *cpl* et les deux projections *pr1* et *pr2*. Les voici en syntaxe OCaml.

```
let cpl x y = fun c -> c x y
let pr1 c = c (fun x y -> x)
let pr2 c = c (fun x y -> y)
```

Exercice 117 : (*) factorielle

Définir la factorielle d'un entier de Church n.
Indication : itérer n fois une transformation bien choisie.

NB On peut utiliser la syntaxe OCaml, mais sans la construction `let rec` !

Exercice 118 : tests sur des entiers

On rappelle que les booléens sont définis par : $V = \lambda xy.x$ et $F = \lambda xy.y$.

— Donner une fonction *test0* qui, appliquée à un argument n, rend V si n est l'entier de Church codant 0 et qui rend F si n est un autre entier de Church.
— Vérifier que

test0 C_n est vrai si et seulement si $n = 0$

Écrire des fonctions calculant :

— le prédécesseur d'un entier de Church n,
— la différence entre deux entiers de Church,
— testant l'égalité de deux entiers de Church.

(indication : itérer n fois une transformation bien choisie),

NB On peut tenter d'utiliser la syntaxe OCaml dans l'exercice précédent, mais les λ-expressions ne sont pas typables, du moins dans le système de types de ML. Elles le sont dans des systèmes de types plus complexe que l'on ne considère pas ici.

Exercice 119 : Entiers de Barendregt (Examen 2010 14 points)

On rappelle que les booléens sont définis par : $V = \lambda xy.x$ et $F = \lambda xy.y$. On considère le codage de Barendregt pour les entiers naturels :

— $B_0 = \lambda p.pVV$
— $B_n = \lambda p.pFB_{n-1}$ pour $n \geq 1$

1. (4 points) Écrire B_2, et β-réduire B_2F.
2. (3 points) $S = \lambda xp.pFx$ définit le successeur d'un entier de Barendregt. Vérifier que SB_n se réduit en B_{n+1}.
3. (4 points) Définir le prédécesseur P d'un entier naturel de Barendregt.
4. (3 points) Vérifier que PB_n se réduit en B_{n-1}.

Problème d'entraînement : Vote (Devoir Maison 2010)

Nous nous intéressons à un système de vote à n candidats dans lequel chaque votant inscrit sur son bulletin de vote une liste ordonnée de candidats, et non pas un candidat unique comme cela se fait (par exemple) pour une élection présidentielle en France.
Quand un votant remplit son bulletin de vote en listant dans cet ordre :

1. Alice
2. Charlie
3. Bob

cela signifie qu'il préfère Alice à Bob ou Charlie, qu'il préfère Charlie à Bob, et qu'il pr candidat. Nous supposons qu'il y a n candidats désignés par leur numéro de 0 à $n-1$. Ce nombre de candidats sera un paramètre de certaines des fonctions à écrire, si besoin.
Nous définissons aussi la notion de matrices des préférences : il s'agit d'une matrice $A = (a_{i,j})$ de $\mathcal{M}_n(\mathbb{N})$ telle que le coefficient $a_{i,j}$ est égal au nombre de fois qu'un votant a préféré le candidat i au candidat j.

1. (4 points) Écrire une fonction `depouille` qui transforme un bulletin de vote représenté par une liste d'entiers en une matrice de préférences. Nous penserons à tenir compte des candidats non inscrits sur le bulletin.
2. (3 points) Écrire une fonction `accumule` qui prend deux matrices A et B de mêmes dimensions et modifie A en $A+B$.
3. (4 points) En déduire une fonction `preferences` qui calcule la matrice globale des préférences à partir de la liste des bulletins des votants.
4. Nous définissons un vainqueur de Condorcet comme un candidat qui, si on l'avait fait concourir contre chaque autre candidat séparément, aurait gagné chacun de ses duels. Un tel vainqueur est unique s'il existe, et il n'existe pas toujours (paradoxe du vote de Condorcet).

   (2 points) Construire un exemple d'élection, avec le détail de chaque bulletin, dans laquelle il n'existe pas de vainqueur de Condorcet. Votre exemple n'a pas besoin d'être très gros ; néanmoins il doit y avoir un nombre non nul de candidats et au moins un bulletin exprimant une liste valide et non vide.
5. (2 points) Donner deux exemples complets d'élections en explicitant toutes les étapes :
   — bulletins exprimés par les votants ;
   — matrice globale des préf
   — résultat de l'appel à `condorcet`.