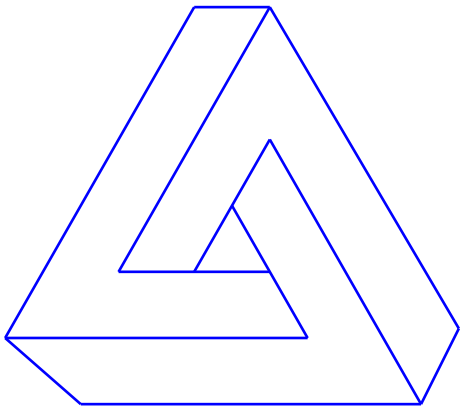
Polytech Grenoble

PF : Programmation Fonctionnelle

Livret d'exercices



Pierre Corbineau, 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

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://v2.ocaml.org/api/.

1 Bases

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

1.1 Type somme et filtrage

Exercice 1.1 [1] : Aires

Écrire le calcul de l'aire :
- d'un carré de côté a.
- d'un rectangle de côtés a et b.
- d'un cercle de rayon r.
- d'un triangle rectangle de côté a et d'hypoténuse h (en utilisant le théorème de Pythagore).

Exercice 1.2 [2] : Type somme pour des figures géométriques

- Définir un type somme `geom` pour représenter les figures géométriques considérées dans l'exercice 1.
- Décrire le calcul de l'aire d'une figure g de type `geom`.

1.2 Fonctions

Exercice 1.3 [3] : Fonctions de calcul d'aire

- Écrire une fonction pour chaque calcul de l'exercice 1.
- Écrire deux fonction `aire1` et `aire2` qui calculent l'aire d'une figure de l'exercice 2, en utilisant ou non les fonctions auxiliaires de l'item précédent.
- Reprendre les questions précédentes en ajoutant une figure constituée d'un simple point (sans coordonnées).

Exercice 1.4 [4] : `if` fonctionnel, minimum de deux entiers

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

Exercice 1.5 [5] : Minimum de trois entiers

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

1.3 Récursivité

Exercice 1.6 [6] : Factorielle

Écrire une fonction qui calcule $n!$.

Calculer $(2022 - 2 * 5!) * 5! + 2 * 5! * (5! - 1)$ en évitant les répétitions de calculs.

1.4 Types produits : couples

Exercice 1.7 [7] : 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.

1.5 Portée, typage, évaluation

Exercice 1.8 [8] : Portées dans des expressions simples

1. Dans chacune des trois sessions suivantes, dire ce que vaut y après avoir évalué les définitions données. Donner le type de chaque définition.

   (a)
   $$\begin{array}{l}\mathsf{let}\ x\ =\ 5\\ \mathsf{let}\ y\ =\ x\ +\ 3\\ \mathsf{let}\ x\ =\ 7\end{array}$$

   (b)
   $$\begin{array}{l}\mathsf{let}\ a\ =\ 5\\ \mathsf{let}\ f\ =\mathsf{fun}\ x\ \rightarrow\ x\ +\ a\\ \mathsf{let}\ a\ =\ 10\\ \mathsf{let}\ y\ =\ f\ a\end{array}$$

   (c)
   $$\begin{array}{l}\mathsf{let}\ a\ =\ 5\\ \mathsf{let}\ f\ =\mathsf{fun}\ x\ \rightarrow\ x\ +\ a\\ \mathsf{let}\ a\ =\ a\ +\ 1\\ \mathsf{let}\ y\ =\ f\ a\end{array}$$

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 1.9 [9] : Portées et fonctions

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

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. let $f\ =$ fun $x\ \rightarrow$ let $a = 1$ in $a + x$ in $f\ a$

3. let $a = 1$ in let $f\ =$ fun $x\ \rightarrow\ a + x$ in let $a = 2$ in $f\ a$

4. let $g\ =$ fun $x\ \rightarrow\ 2 + (f\ x)$ in let $f\ =$ fun $x\ \rightarrow\ 1 + x$ in $g\ 0$

5. let $f\ =$ fun $x\ \rightarrow\ 1 + x$ in let $g\ =$ fun $x\ \rightarrow\ 2 + (f\ x)$ in let $f\ =$ fun $x\ \rightarrow\ 4 + x$ in $g\ 0$

6. let rec $f\ =$ fun $x\ \rightarrow\ 1\ +\ (f\ (x - 1))$ in $f\ 3$

7. let rec $f\ =$ fun $x\ \rightarrow$ if $x = 0$ then 0 else $1\ +\ (f\ (x - 1))$ in
   let $g\ =$ fun $x\ \rightarrow\ f\ (x + 1)$ in $g\ 3$

2 Structures arborescentes

2.1 Arbres binaires

Exercice 2.1 [10] **: Fonctions simples sur un arbre**

1. Définir un type `abin` pour un arbre binaire d'entiers (les nœuds doivent être étiquetés, pas les feuilles).
2. Écrire une fonction qui calcule le nombre de nœuds d'un arbre binaire.
3. Écrire une fonction qui calcule la hauteur d'un arbre binaire.
4. Écrire une fonction qui calcule le produit de tous les éléments d'un arbre binaire.
5. Écrire une fonction qui calcule la somme de tous les éléments d'un arbre binaire.
6. Écrire une fonction qui détermine si un élément appartient à un arbre binaire.
7. Écrire une fonction qui calcule le maximum d'un arbre binaire.
8. 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 2.2 [11] **: Preuve d'une propriété simple sur les arbres**

1. 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. 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. Prouver que pour tout arbre binaire a : `taille (double a) = taille a`

Exercice 2.3 [12] **: Arbres et listes**

1. 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.
2. Réécrire cette fonction afin de construire la liste correspondant au parcours infixe mais de droite à gauche.
3. 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.

2.2 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 2.4 [13] **: 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 2.5 [14] : 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 2.6 [15] : 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 2.7 [16] : vérification d'ABR

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

Exercice 2.8 [17] : 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.

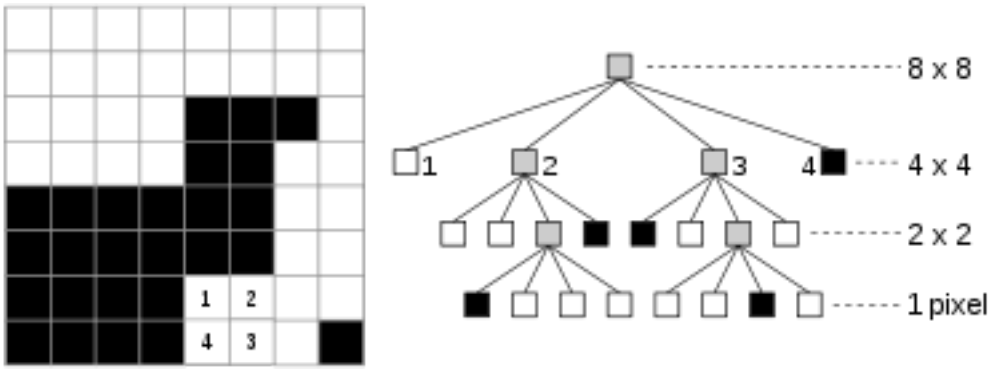
- Écrire une fonction qui prend un arbre binaire de recherche et renvoie son minimum.
- Écrire une fonction qui prend un arbre binaire de recherche et renvoie son deuxième plus grand élément.
- Écrire une fonction qui prend un arbre binaire de recherche et calcule la médiane de ses éléments.

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 2.9 [18] : Quadtrees et compression d'image

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. 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. 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. Écrire une fonction `quadtree_vers_liste` de type `quadtree -> bit list` qui transforme un quadtree en une liste de `bit` selon le codage[1].
4. Écrire une fonction de décodage `liste_vers_quadtree` de type `bit list -> quadtree`, qui transforme une liste de `bit` en le quadtree correspondant.

1. vous pouvez utiliser `@`.

3 Listes

3.1 Exercices élémentaires sur les listes

Exercice 3.1 [19] : 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 3.2 [20] : Appartenir à une liste

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

Exercice 3.3 [21] : 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 3.4 [22] : Max d'une liste

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

Exercice 3.5 [23] : 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 3.6 [24] : 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 3.7 [25] : 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 3.8 [26] : Concaténation de deux listes

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

Exercice 3.9 [27] : Égalité de deux listes

Écrire une fonction qui teste si deux listes sont égales.

Exercice 3.10 [28] : Dernier élément d'une liste

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

Exercice 3.11 [29] : Nombre pair d'éléments

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

Exercice 3.12 [30] : Renversement d'une liste l

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

3.2 Preuves de programmes sur les listes

Exercice 3.13 [31] : Correction de minimum

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

Exercice 3.14 [32] : Duplication et longueur

Montrer la propriété suivante pour les fonctions `duplique` et `longueur` écrites vers la page 11.

$$\forall u, 2 \times \mathit{longueur}\, u = \mathit{longueur}(\mathit{duplique}\, u)$$

Exercice 3.15 [33] : Concaténation, longueur et renversement

Montrer les propriétés suivantes pour les fonctions `append`, `longueur` et `renverse` écrites vers la page 12.

— $\forall u \forall v,\ \mathit{longueur}\ (u @ v) = \mathit{longueur}\ u + \mathit{longueur}\ v$
— $\forall u,\ \mathit{longueur}\, u = \mathit{longueur}\ (\mathit{renverse}\, u)$

Exercice 3.16 [34] : Concaténation de deux listes

Propriétés algébriques de la concaténation

— Rappeler la définition de la fonction concaténation.
— Démontrer que $[\,]$ est un élément neutre, c'est-à-dire : $\forall u,\ [\,] @ u = u = u @ [\,]$.
— Démontrer que la concaténation est associative, c.-à-d. : $\forall u_1\, u_2\, u_3,\ (u_1 @ u_2) @ u_3 \ = \ u_1 @ (u_2 @ u_3)$.
Indication : u_2 et u_3 étant fixés arbitrairement, procéder par récurrence structurelle sur u_1.

Exercice 3.17 [35] : Renversement et concaténation

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

Exercice 3.18 [36] : Renversement du renversement

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

Exercice 3.19 [37] : Renversement, version efficace (**)

On propose la fonction suivante pour renverser une liste.

```
let rec renv_acc u a = match u with
  | [] -> a
  | x :: q -> renv_acc q (x :: a)

let renv_efficace l = renv_acc l []
```

Pour quelles raisons est-elle plus efficace que la fonction `renverse` écrite précédemment ? Indiquer et justifier la complexité en temps de ces deux fonctions.
Montrer que cette fonction produit le même résultat que celle écrite précédemment, c'est-à-dire que

$$\forall u, renv\_acc\, u\, [] = renverse\, u.$$

3.3 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 éléments 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 $n = 4$.

3.4 Tri rapide (Quicksort)

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

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 3.20 [38] : 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 3.21 [39] : 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.

Exercice 3.22 [40] : 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 $\alpha\ ab\ =\ F$ of $\alpha\ \mid\ N$ of $\alpha\ ab\ \times\ \alpha\ ab$

let rec $to\_list\ a\ =$ match a with
$\mid\ F\ (x)\ \rightarrow\ [x]$
$\mid\ N\ (g,\ d)\ \rightarrow\ to\_list\ g\ @\ to\_list\ d$

let rec $tle\ a\ u\ =$ match a with
$\mid\ F\ (x)\ \rightarrow\ x\ ::\ u$
$\mid\ N\ (g,\ d)\ \rightarrow\ tle\ g\ (tle\ d\ u)$

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

$$\forall a\, u, tle\, a\, u = to\_liste\, a\ @\ u$$

3.5 Programmes variés sur les listes

Exercice 3.23 [41] **: 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 3.24 [42] **: Découpage d'une liste**

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

Exercice 3.25 [43] **: 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 3.26 [44] **: 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 3.27 [45] **: Second maximum d'une liste ***

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

Exercice 3.28 [46] **: 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`.

4 Exceptions

Exercice 4.1 [47] **: 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.

Que se passe-t-il si le calcul de p lève lui-même une exception, par exemple sur une division par 0 ?

Exercice 4.2 [48] **: Attraper une exception**

Utiliser la fonction précédente pour définir une fonction `trouve_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.

Même exercice avec comme type de résultat `'a option` au lieu de `'a list`.
Rappel : le type `'a option` comprend deux constructeurs :
- `None`
- `Some of 'a`

et on peut le voir comme codant des listes avec seulement 0 ou exactement 1 élément.

Exercice 4.3 [49] **: 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 ?

5 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 5.1 [50] : somme de fonctions

Indiquer deux autres écritures de la fonction *somme_fonctions*.

Exercice 5.2 [51] : composition de fonctions

- Donner la définition d'une fonction qui rend la composée de deux fonctions de type `'a -> 'b` et `'b -> 'c`.
- Donner la définition d'une fonction qui rend la composée de deux fonctions de types `'a -> 'b * 'c` et `'b -> 'c -> 'd`.

Exercice 5.3 [52] : factorielle avec combinateurs

- 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 5.4 [53] : 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 égales :

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

- Formaliser et démontrer que pour tout f, $uncurry(curry\, f)$ et f sont extensionnellement égales.

Exercice 5.5 [54] : 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 $n, n-1, \dots 1$.

Exercice 5.6 [55] : 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 5.7 [56] : Utilisation

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

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

Exercice 5.8 [57] : 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.

6 Listes paresseuses

Exercice 6.1 [58] **: Listes infinies**

On définit le type des listes infinies de la façon suivante :

type α *inflist* $=$ *unit* $\rightarrow$ α *contentsil*
and α *contentsil* $=$ *Cons* of $\alpha \times \alpha$ *inflist*

On note que l'évaluation de chaque maillon de la liste est suspendue, ce qui permet de définir des listes « infinies » (en réalité, plutôt construites « à la demande »), comme cette liste de tous les entiers :

let rec *from* n $=$ fun () $\rightarrow$ *Cons* $(n,$ *from* $(n+1))$
let *nat* $=$ *from* 0

Écrire les termes et fonctions suivants :
- la liste infinie `bits = [0; 1; 0; 1; 0; 1; 0; 1; ... ]` (en vous inspirant de `nat`)
- la fonction `peek` qui prend une liste infinie et renvoie son premier élément
- la fonction `suppr : int -> 'a inflist -> 'a inflist` qui prend un entier n et une liste infinie et renvoie la liste (infinie) privée de ses n premiers éléments
- la fonction `get : int -> 'a inflist -> 'a list` qui prend un entier n et une liste infinie et renvoie la liste (OCaml) de ses n premiers éléments
- la fonction `iterate` qui prend une valeur x et une fonction f et renvoie la liste infinie `[x ; f x ; f (f x) ; ... ]`

Exercice 6.2 [59] **: Listes paresseuses**

On peut aller un peu plus loin en construisant des listes *paresseuses* :

type α *lazylist* $=$ *unit* $\rightarrow$ α *contentsll*
and α *contentsll* $=$ *Nil* $\mid$ *Cons* of $\alpha \times \alpha$ *lazylist*

Cette fois les listes peuvent être au choix finies ou infinies, mais la construction de leurs éléments se fait toujours à la demande. Ainsi on peut définir :

let rec *range* i j $=$ fun () $\rightarrow$ if $i = j$ then *Nil* else *Cons* $(i,$ *range* $(i+1)$ $j)$

- Si $a \leq b$ alors `range a b` construit l'intervalle d'entiers $[a, b[$.
- Si $a > b$ alors `range a b` construit la liste infinie `[a ; a+1 ; a+2 ; ...]`.

Écrire les fonctions suivantes :
- la fonction `peek : 'a lazylist -> 'a option` qui prend une liste paresseuse et renvoie `Some` de son premier élément, ou bien `None` si la liste est vide
- la fonction `get : int -> 'a lazylist -> 'a list` qui prend un entier n et une liste paresseuse et renvoie la liste (OCaml) de ses n premiers éléments (si la liste est trop courte, on renvoie tout ce qui est disponible)
- la fonction `evens : 'a lazylist -> 'a lazylist` qui prend une liste et renvoie la liste de tous ses éléments d'indice pair
- la fonction `splice : 'a lazylist -> 'a lazylist -> 'a lazylist` qui prend deux listes `[a0 ; a1 ; a2 ; ...]` et `[b0 ; b1 ; b2 ; ...]` et renvoie `[a0 ; b0 ; a1 ; b1 ; a2 ; b2; ...]`.
  Si une des listes est épuisée avant l'autre, on terminera par tous les éléments restants dans la liste la plus longue.
- la fonction `enumerate : 'a lazylist -> (int * 'a) lazylist` qui numérote les éléments d'une liste : `[a0 ; a1 ; a2 ; ...]` devient `[(0, a0) ; (1, a1) ; (2, a2) ; ...]`
- la fonction `append` qui concatène deux listes paresseuses

— la fonction `flat_map : ('a -> 'b lazylist) -> 'a lazylist -> 'b lazylist` telle que `flat_map f al` calcule une liste de `'b` en appliquant f à chaque élément de `la` et concatène les listes ainsi obtenues (le résultat doit évidemment être construit à la demande).

7 Analyseurs sur des listes ou des listes paresseuses

Dans un premier temps les seuls terminaux considérés seront des caractères, on analysera donc des listes de terminaux. On utilisera alors les types et primitives suivants.

```
(* Le type des fonctions qui épluchent une liste de caractères. *)
type analist = char list -> char list

(* Le type des fonctions qui épluchent une liste de caractères et rendent un résultat. *)
type 'res ranalist = char list -> 'res * char list

(* L'exception levée en cas de tentative d'analyse ratée. *)
exception Echec

(* Ne rien consommer *)
let epsilon : analist = fun l -> l
(* Un epsilon informatif *)
let epsilon_res (info : 'res) : 'res ranalist =
  fun l -> (info, l)

(* Terminaux *)
let terminal c : analist = fun l -> match l with
  | x :: l when x = c -> l
  | _ -> raise Echec

let terminal_cond (p : char -> bool) : analist = fun l -> match l with
  | x :: l when p x -> l
  | _ -> raise Echec

(* Le même avec résultat *)
(* f ne retourne pas un booléen mais un résultat optionnel *)
let terminal_res (f : 'term -> 'res option) : 'res ranalist =
  fun l -> match l with
  | x :: l -> (match f x with Some y -> y, l | None -> raise Echec)
  | _ -> raise Echec
```

Exercice 7.1 [60] : Suite de chiffres, somme des chiffres, schéma de Horner

1. Écrire une grammaire d'un langage reconnaissant une suite de chiffres.

   Recommandation : un non-terminal U pour « un chiffre exactement », un non-terminal A pour « au moins un chiffre », un non-terminal C pour « chiffres » (zéro ou plusieurs).
2. Écrire sur ce schéma une fonction `chiffres` de type `analist` qui à partir d'une liste rend celle-ci privée de son plus long préfixe de chiffres.
3. Écrire sur ce même schéma une fonction `sommechiffres` de type `int ranalist` qui à partir d'une liste rend celle-ci privée de son plus long préfixe de chiffres en rendant également l'entier correspondant à la somme de ces chiffres.

   Il est utile de définir préalablement la fonction `un_chiffre` de type `int ranalist` correspondant au non-terminal U : à partir d'une liste l, `un_chiffre` l rend l privée de son premier caractère si c'est un chiffre, ainsi que l'entier correspondant à ce chiffre. Le code suivant donne l'entier correspondant à un caractère chiffre (explication : dans le codage ASCII les chiffres sont placés consécutivement) : `Char.code c - Char.code '0'`.
4. Toujours sur ce même schéma, écrire une fonction `horner` de type `int -> int ranalist` prenant un accumulateur a en premier argument puis une liste en second argument et rend celle-ci

privée de son plus long préfixe de chiffres en rendant également l'entier dénoté par cette suite de chiffres lorsque a vaut 0. Dans le cas général, a correspond au décodage des chiffres déjà lus.
Par exemple si la liste `s` fournie en entrée débute par les caractères de la chaîne `"123 1981"` alors `horner 0 s` doit renvoyer 123 ainsi que la liste des caractères de `" 1981"`, tandis que `horner 42 s` doit renvoyer l'entier 42123 (ainsi que la liste des caractères de `" 1981"`).
Rappel : $1664 = (((1 \times 10 + 6) \times 10) + 6) \times 10 + 4$.

5. (*) Variante fonctionnelle, d'un côté plus sophistiquée mais d'un autre côté, elle suit la même structure que `sommechiffres` : l'idée est de rendre une fonction au lieu d'un entier. Écrire une fonction `horner_f` de type `(int -> int) ranalist`. Par exemple si la liste `s` fournie en entrée débute par les caractères de la chaîne `"123 1981"` alors `horner_f s` doit renvoyer une fonction qui appliquée à 0 rend 123 ainsi que la liste des caractères de `" 1981"`.
Il est utile de définir préalablement la composition de deux fonctions et la fonction identité.

Exercice 7.2 [61] : Initiation aux combinateurs

Reprendre l'exercice précédent en utilisant les combinateurs d'analyseurs.

Exercice 7.3 [62] : Analyse d'expressions arithmétiques

L'objectif de l'exercice est de travailler sur un "grand classique" de l'analyse syntaxique.
Pour cette partie, on pourra se baser sur la version `analist.ml` Mais on préfèrera la version `anacomb.ml` qui donne un code plus concis et proche de la grammaire.
On commence par remarquer que La grammaire la plus naturelle pour des expressions arithmétiques est récursive à gauche. En effet, une expression comme `10 - 3 + 5` se lit naturellement "10 auquel on retranche 3 puis auquel on ajoute 5". Pour simplifier on va d'abord se limiter à des expressions additives et non parenthésées. On part donc de la grammaire :

```
E ::= E '+' n | n
```

Pour ne pas s'encombrer d'une analyse lexicale préalable, on se contente de terminaux qui sont des caractères, et les entiers seront de simples chiffres '3' ou '8'. Dans la grammaire ci-dessus, `n` sera donc remplacé par `C` avec la grammaire

```
C ::= '3' | '8'
```

On propose de récupérer des AST de type suivant (ce type est un peu trop riche à ce stade, mais permet les extensions à venir de la grammaire).

```
type aexp =
  | Acst of int
  | Apl of aexp × aexp (* addition *)
  | Amo of aexp × aexp (* soustraction *)
  | Amu of aexp × aexp (* multiplication *)
```

1. On commence par programmer sur la base de la grammaire ci-dessus *SANS* modifier la grammaire. Écrire un programme `pa_E` (et `pa_C`) de type `char analist` pour cette grammaire *non transformée*, puis un programme `pr_E` de type `(aexp, char) ranalist` (avec `pr_C` de type `(int, char) ranalist`) pour cette grammaire *non transformée*.
Pourquoi ces programmes sont-ils inappropriés ?
2. Pour résoudre le problème posé par la récursion à gauche, il convient de traiter le problème à la racine, c'est-à-dire au niveau de la grammaire elle-même. On prend donc une grammaire équivalente à la grammaire ci-dessus (elle décrit le même langage). `SE` signifie "suite de l'expression".

```
E ::= C SE
SE ::=  '+' C SE | ε
```

Redéfinir les programmes `pa_E` et `pr_E`. Pour le second il convient de faire en sorte que les AST rendus soient ceux qui auraient été rendus par le programme `pr_E` précédent s'il ne bouclait pas. Ainsi l'AST de correspondant à `"8+3+3"` doit être `Apl (Apl (Acst(8), Acst(3)), Acst(3))` et non `Apl (Acst(8), (Apl (Acst(3), Acst(3))))`.

3. Compléter la grammaire puis les programmes, de façon à analyser des expressions additives comportant des parenthèses, comme `"3+(8+3)+(8)"`.
4. Compléter la grammaire puis les programmes, en ajoutant l'opérateur de soustraction (de même niveau précédence que l'addition) puis l'opérateur de multiplicatives (plus prioritaire que les précédents). On pourra à cet effet commencer par compléter la gramaire récursive à gauche, plus simple, puis éliminer la récursion à gauche de la même façon que précédemment, et enfin écrire les programmes d'analyse correspondants.

8 Aspects impératifs

8.1 Références

Exercice 8.1 [63] : Compteur

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.

8.2 Tableaux

Exercice 8.2 [64] : 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 8.3 [65] : Type Array et listes

Écrire les fonctions suivantes (certaines sont présentes dans la bibliothèque Array) :
- `array_change : ('a -> 'a) -> 'a array -> unit` applique une fonction f à tous les éléments d'un tableau a, qui aura pour valeur finale $[|fa.(0); fa.(1); ...|]$.
- `array_map : ('a -> 'a) -> 'a array -> 'a array` applique une fonction f à tous les éléments d'un tableau a, et construit le tableau $[|fa.(0); fa.(1); ...|]$. On pourra utiiiliser, ou non, la fonction `array_change` ; discuter.
- `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.

9 Modules et foncteurs

9.1 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 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
```

nb : en Ocaml, le corps de la signature du module (l'interface) peut être mis dans un fichier `.mli` (ex : `unModule.mli`), et le corps du module dans un un fichier `.ml` (ex : `unModule.ml`).

Exercice 9.1 [66] : Écriture d'un module de gestion d'une pile

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*.

2) Écrire un module implémentant cette interface.

9.2 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 9.2 [67] : Écriture d'un foncteur de tri

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.

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.

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

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.

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 $x \to U$ 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 ;
   dans ce terme, x est dite *libre*.
2. Abstraction : si x est une variable, et U un λ-terme, alors $\lambda x.U$ est un λ-terme ;
   les occurrences de x qui étaient libres dans U deviennent *liées* dans $\lambda x.U$, le statut (*libre* ou *lié*) des autres variables reste inchangé.
3. Application : si U et V sont des λ-termes, alors $(U\,V)$ est un λ-terme ;
   le statut (*libre* ou *lié*) des occurrences de variables dans U et V reste inchangé dans $(U\,V)$.

Conventions de simplification d'écriture (similaires à celles de OCaml)

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 $\lambda x_1 x_2 \ldots x_n.U$
   Ainsi,comme en OCaml, les fonctions à plusieurs arguments se ramènent à des fonctions à un seul argument.

α-conversion : on peut 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 : c'est l'unique mécanisme de calcul, et il formalise l'appel d'une fonction sur un paramètre effectif. Il s'écrit $(\lambda x.U)V \to_\beta U[V/x]$, où $U[V/x]$ dénote le terme U dans lequel ses occurrences de x libres ont été substituées par V.
Attention, les variables libres de V ne doivent pas être capturées dans U. Pour se préserver de ce danger, on suivra consciencieusement l'intention de l'appel de fonction que l'on a intuitivement. Cela revient, techniquement, à appliquer préalablement des α conversions dans U en cas de besoin : si V contient des occurrences de y libres, et que U contient un sous-terme $\lambda y.U'$, α-convertir ce dernier en prenant une nouvelle variable à la place de y.

Il est remarquable qu'à partir de ces seules règles, c'est-à-dire en utilisant comme seul mécanisme de calcul 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.

10.1 Termes très simples

Exercice 10.1 [68] : Beta-réductions

Voici quelques combinateurs standards :

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

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

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

1. Vérifier que $I = (SKK)$
2. β-réduire $SI(S(KK)Iu)v$.

Exercice 10.2 [69] : Booléens et fonctions booléennes

Les booléens sont représentés par : $V = \lambda xy.x$ et $F = \lambda xy.y$.

1. β-réduire les termes suivants.
   — F F V
   — V F V
   — F V V
2. Donner un terme représentant la fonction $ifthenelse$.
3. Donner les termes représentant les fonctions booléennes suivantes :
   — *not*
   — *and*
   — *or*
   — *nand*
   — *nor*
   Rappel des 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

10.2 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 10.3 [70] : 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 10.4 [71] : 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 $exp\ C_n\ C_m = C_{n^m}$
Indications : $n\ f$ est f^n ; $a^{b+c} = a^b\ a^c$, $a^{bc} = (a^b)^c$.

Exercice 10.5 [72] : 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 10.6 [73] : (*) 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 10.7 [74] : 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.

10.3 Une autre representation des entiers

Exercice 10.8 [75] : Entiers de Barendregt

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 :
— $B_0 = \lambda p.p\,V\,V$
— $B_n = \lambda p.p\,F\,B_{n-1}$ pour $n \geq 1$

1. Écrire B_2, et β-réduire $B_2\,F$.
2. $S = \lambda xp.p\,F\,x$ définit le successeur d'un entier de Barendregt. Vérifier que $S\ B_n$ se réduit en B_{n+1}.
3. Définir le prédécesseur P d'un entier naturel de Barendregt.
4. Vérifier que $P\,B_n$ se réduit en B_{n-1}.

A Annexe : compléments, annales

Cette annexe reprend, plus ou moins en vrac, des exercices variés traités au cours des années précédentes.

A.1 Bases

Exercice A.1 [76] : 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 A.2 [77] : 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 A.3 [78] : Maximum de deux entiers

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

Exercice A.4 [79] : 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 A.5 [80] : Pgcd

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

Exercice A.6 [81] **: Somme des n premiers entiers**

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

Exercice A.7 [82] **: Somme des n premiers entiers impairs**

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

Exercice A.8 [83] **: Division Euclidienne**

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

Exercice A.9 [84] **: Prédicats "again"**

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

Exercice A.10 [85] **: 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.

Exercice A.11 [86] **: 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 A.12 [87] **: 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 A.13 [88] **: 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.

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).

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.

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.

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

Exercice A.14 [89] : Cartes

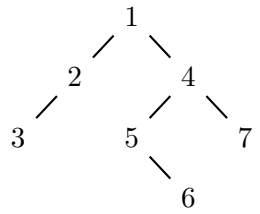
— Définir un type `couleur`, pour les quatre couleurs d'un jeu de carte.
— Définir un type `valeur`, pour les valeurs des 8 cartes d'un jeu de trente deux cartes.
— Définir un type `carte` qui compte en plus un joker.
— Écrire une fonction qui donne la couleur d'une carte.

Exercice A.15 [90] : 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.

A.2 Arbres

Exercice A.16 [91] : 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.

Exercice A.17 [92] : 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.

Exercice A.18 [93] : 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
```

A.3 Listes

Exercice A.19 [94] : 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 A.20 [95] : 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 A.21 [96] : 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 A.22 [97] : 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 A.23 [98] : 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.) [2]

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 A.24 [99] : 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);;
```

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

Exercice A.25 [100] : 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},
      {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
```

   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 [])
```

Exercice A.26 [101] : Division Euclidienne

— Rappeler la fonction euclide (voir page 32).
— Pourquoi calcule-t-elle le bon résultat ?
— Pourquoi se termine-t-elle ?

Exercice A.27 [102] : 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 $f(l)$).

Exercice A.28 [103] : Tri crêpe (Examen 2010)

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 [3].

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

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 [4].
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.
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 vrai 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.

A.4 Ordre supérieur

Exercice A.29 [104] : 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 A.30 [105] : 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>
```

4. Vous pouvez utiliser `@`.

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 A.31 [106] : 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 A.32 [107] : 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.

A.5 Analyse lexicale et syntaxique

Exercice A.33 [108] : Connect 6

Énoncé complet placé sur le poly de TP.

1. Définir un type `coup`.
2. **Analyse lexicale**.
3. **Analyse syntaxique**.
4. (**) Définir un type `plateau`.
5. (**) Écrire une fonction `joue_coup` qui calcule un nouvel état à partir d'un état du plateau et d'un coup.
6. (**) Écrire une fonction `resultat`.

A.6 Aspects impératifs

Exercice A.34 [109] : Références

— 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 A.35 [110] : Factorielle

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

Exercice A.36 [111] : 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
$\quad | \; 0 \rightarrow 0$
$\quad | \; n \rightarrow 2 \times (n-1) + 1 \; + \; simp1 \; (n-1)$

let rec *simp2* n = match n with
$\quad | \; 0 \rightarrow 0$
$\quad | \; n \rightarrow 2 \times n - 1 \; + \; simp2 \; (n-1)$

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 de 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`.

Exercice A.37 [112] : 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 A.38 [113] : 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 A.39 [114] : 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
```

A.7 Modules, foncteurs

Exercice A.40 [115] : 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
```

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.

2) Implémenter ce module.

Exercice A.41 [116] : 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 A.42 [117] : 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 A.43 [118] : 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 A.44 [119] : 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 A.45 [120] : 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 A.46 [121] : Listes paresseuses

Voici une signature d'un module ListT :

```
module type ListT =
  sig
    type α t
    val vide : α t
    val cons : α → α t → α t
    val hd : α t → α (* may raise Failure "hd" *)
    val tl : α t → α t (* may raise Failure "tl" *)
    val map : (α → α) → α t → α t
    val to_list : α t → α list
  end
```

1) Écrivez-en une implémentation utilisant les listes d'Ocaml
Voici maintenant un type représentant des listes dites paresseuses.

```
type α lazylist = unit → α contentsll
and α contentsll = Nil | Cons of α × α lazylist
```

2) Proposez une autre implémentation du module Liste basée sur ces listes paresseuses.

3) Finissez les fonctions suivantes, dont le nom et le type constituent une spécification suffisante :

```
let (char_liste_of_string : string → 'char Liste.t) = fun s → (* à finir *)

let (char_liste_of_file : string → 'char Liste.t) = fun fn → (* à finir *)

let (string_of_list : (α → string) → α Liste.t → string) = fun a2s l → (* à finir *)

let (string_of_char_liste : char Liste.t → string) =(* à finir *)
```

Rappels :

- `"une chaine".[1]` renvoie le caractère 'n' qui est à l'index 1 de la chaîne `"une chaine"`
- la fonction prédéfinie `open_in : string -> input_channel` permet d'ouvrir un fichier en lecture
- la fonction prédéfinie `input_char :input_channel -> char` permet de lire un caractère dans un fichier ouvert en lecture.
- la fonction prédéfinie `Char.escaped : char -> string` permet de transformer un caractère en une chaîne.

```
let test str = assert(str = string_of_char_liste (char_liste_of_string str))
let _ = test "voici␣un␣test␣possible␣pour␣quelques␣unes␣des␣fonctions␣prédécentes␣!"
```

4) Pourquoi les listes paresseuses sont-elles nommées ainsi ? Voyez-vous leur intérêt ?

5) Redéfinissez dans votre projet les types `analist` et `ranalist` ainsi :

```
type 'term analist = 'term Liste.t → 'term Liste.t
type ('res, 'term) ranalist = 'term Liste.t → 'res × 'term Liste.t
```

Il vous faudra bien sûr légèrement modifier quelques fonctions dans votre projet pour que tout fonctionne comme avant.

Exercice A.47 [122] : 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
```

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])`.

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"])]`.

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 A.48 [123] : Foncteur de datagramme

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.

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.

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.