Exercices de Programmation OCaml

Eliana Carozza

IpeSup MP2I-2025/2026

Contents

Exercice 0.1. Évaluation de déclarations imbriquées

Quelle est la valeur de `z` après les déclarations suivantes ?

```
let x = 10
let x =
  let y = 10 + x in
  let y = let x = y * x in y + x in
  y + x
let z = x + 100
```

1 Typage

Exercice 1.1. Typage de programmes simples

Dire si les programmes suivants sont bien typés ou mal typés. S'ils sont bien typés, donner le type de toutes les variables et fonctions globales. S'ils sont mal typés, indiquer précisément la ligne et la nature de l'erreur de typage.

1. ```
   let x = 3
   let y = 4
   let z = string_of_int x + 4
   ```

2. ```
   let f x y = (x + 1) * (y - 1)
   let u = f (f 2 3) (f 4 5)
   ```

3. ```
   let x = 47
   let y =
     if x mod 2 = 0 then 3.0
     else 4.0
   let z = x + y
   ```

4. ```
   let f x = x / 17
   let g x =
     if x mod 2 = 0 then "pair"
     else false
   let u = f 42
   ```

5. ```
   let rec f n =
     if n <= 0 then 1
     else n * f (n - 1)
   let g n =
     Printf.printf "%d! = %d\n" n (f n)
   ```

6. ```
   let rec f n =
     if n >= 0 then (
       Printf.printf "%d\n" n;
       f (n - 1)
     )
   ```

Exercice 1.2. Typage de fonctions récursives

Les fonctions suivantes sont-elles bien typées ? Si oui, donner leur type ; sinon, expliquer pourquoi.

```
let rec f1 x = if x > 0 then [] :: (f1 (x - 1)) else []
let rec f2 a b c = if c <= 0 then a else f2 a b (b c)
let rec f3 x y z = f3 (y, y) z x
```

Exercice 1.3. Filtrage par motifs et exhaustivité

Étant donnée la définition de type suivante :

```
type t = A of t | B of int * t | C
```

On définit la fonction :

```
let g v = match v with
  | A (C) -> 0
  | A (B (x, C)) -> 2
  | A (_) -> 1
  | B (0, _) -> 3
  | B (y, _) -> 4
  | C -> 5
```

La fonction `g` est-elle exhaustive ? Si non, donner un exemple de valeur non filtrée.

Exercice 1.4. Bibliothèque de fractions

Le but de cet exercice est d'écrire une petite bibliothèque permettant de manipuler des fractions. Une fraction $\frac{a}{b}$ est représentée par deux entiers : le numérateur a et le dénominateur b. On souhaite que les fractions vérifient les propriétés suivantes :

- la fraction est irréductible, c'est-à-dire que le PGCD de a et b vaut 1 ;
- le dénominateur b est toujours positif, autrement dit le signe de la fraction est donné par le signe de a.

1. Écrire une fonction récursive `gcd : int -> int -> int` qui calcule le PGCD de deux entiers a et b, en utilisant l'algorithme récursif d'Euclide :
   - si b est nul, renvoyer a ;
   - sinon, renvoyer le PGCD de b et a mod b.

   La fonction est-elle récursive terminale ?
2. Définir un type enregistrement `frac` possédant deux champs entiers : `num` et `denom`.
3. Définir une fonction `simp : frac -> frac` qui simplifie la fraction et s'assure que le dénominateur est positif.
4. Définir les fonctions suivantes :
   - `frac : int -> int -> frac` — construit une fraction à partir de deux entiers ;
   - `add_frac` — additionne deux fractions ;
   - `neg_frac` — renvoie l'opposé d'une fraction ;
   - `sub_frac` — soustrait deux fractions ;
   - `mul_frac` — multiplie deux fractions ;

- `inv_frac` — renvoie l'inverse d'une fraction ;
- `div_frac` — divise deux fractions ;
- `float_of_frac` — convertit la fraction en nombre flottant ;
- `string_of_frac` — convertit la fraction en chaîne de caractères.

Toutes les fonctions qui renvoient des fractions doivent renvoyer des fractions simplifiées. On évitera au maximum de dupliquer du code en réutilisant les fonctions précédemment définies.

Exercice 1.5. Type unifié de nombres

On souhaite maintenant définir un type unique pour représenter trois sortes de nombres : entiers, flottants et fractions (cf. exercice précédent).

```
type num =
  | Int of int
  | Float of float
  | Frac of frac
```

1. Écrire une fonction `string_of_num : num -> string` qui renvoie une chaîne de caractères représentant le nombre donné en argument.
2. Écrire des opérations génériques entre valeurs du type `num`. Pour une expression `n1 ı n2` (où ı représente une opération arithmétique), on adopte les règles suivantes :
   - Si `n1` et `n2` sont du même type, le résultat est de ce type.
   - Si l'un des deux opérandes est un flottant, l'autre est converti en flottant, et le résultat est un flottant.
   - Sinon, si l'un des deux opérandes est une fraction, l'autre est converti en fraction, et le résultat est une fraction.
3. Pour éviter la duplication de code, écrire une fonction générique :

   ```
   exec_op : num -> num ->
             (int -> int -> int) ->
             (frac -> frac -> frac) ->
             (float -> float -> float) ->
             num
   ```

   prenant en argument deux nombres et trois fonctions correspondant respectivement à une opération entre entiers, entre fractions et entre flottants.

   Exemple de squelette :

```
let exec_op n1 n2 op_i op_fr op_fl =
  match n1, n2 with
  | Float f1, Float f2 -> Float (op_fl f1 f2)
  | Float f1, Frac fr2 -> Float ( ... )
  | Frac fr1, Float f2 -> ...
  | ...
```

4. Compléter la fonction `exec_op`.
5. Définir enfin les fonctions :
   - `add_num` — additionne deux nombres ;
   - `sub_num` — soustrait deux nombres ;
   - `mul_num` — multiplie deux nombres ;
   - `div_num` — divise deux nombres ;
   - `neg_num` — renvoie l'opposé d'un nombre.

2 Listes

Exercice 2.1. Maximum d'une liste

Écrire une fonction qui retourne le plus grand élément d'une liste.

Exercice 2.2. Inversion d'une liste

Écrire une fonction qui inverse une liste, de préférence en place.

Exercice 2.3. Appartenance à une liste

Écrire une fonction qui vérifie si un élément apparaît dans une liste.

Exercice 2.4. Éléments en positions impaires

Écrire une fonction qui retourne les éléments situés aux positions impaires d'une liste.

Exercice 2.5. Somme cumulée d'une liste

Écrire une fonction qui calcule la somme cumulée des éléments d'une liste.

Exercice 2.6. Statistiques sur une liste d'altitudes

Un *altimètre* mesure la distance verticale par rapport à un niveau de référence. Lors d'une randonnée en montagne, Alice étalonne son altimètre au départ, puis relève *chaque heure* l'altitude relative atteinte.

À la fin de la randonnée, elle obtient une liste OCaml d'entiers, que l'on notera `alt : int list`. Par exemple (10 heures de marche) :

```
alt = [300; 500; 600; 1000; 800; 900; 500; 600; 200; 0].
```

Alice souhaite calculer différentes grandeurs à partir de ces mesures.

1. **Durée de la randonnée.** Quelle fonction standard d'OCaml (sur les listes) permet d'obtenir la *durée* de la randonnée à partir de `alt` ?

2. **Altitude maximale.** Écrire une fonction `altmax : int list -> int` qui retourne l'altitude maximale atteinte au cours de la randonnée.
   Ex. : sur la liste d'exemple, la fonction doit renvoyer `1000`.

3. **Dénivelé maximal sur une heure.** On appelle *dénivelé horaire* la différence `alt.(i+1) - alt.(i)` (qui peut être négative). Écrire `denivmax : int list -> int` qui renvoie la plus grande différence entre deux *heures consécutives*.
   Ex. : sur l'exemple, le dénivelé maximal vaut `400` (entre les 3^{e} et 4^{e} heures).

4. **Heure de réalisation du dénivelé maximal.** Modifier la question précédente pour écrire `heure_denivmax : int list -> int` qui renvoie *l'indice horaire* i où débute ce dénivelé maximal (on suppose l'indice de départ à 0). *Ex. :* sur l'exemple, la fonction doit renvoyer `2`.

5. **Dénivelé total positif.** On définit le *dénivelé total positif* comme la somme de toutes les hausses d'altitude :

   $$\sum\nolimits_{-} i \,:\, \texttt{alt}(i+1) > \texttt{alt}(i)\big(\texttt{alt}(i+1) - \texttt{alt}(i)\big).$$

   Écrire `denivtotal : int list -> int` qui calcule cette somme.
   Ex. : sur l'exemple, la valeur attendue est `1200`.

6. **Sommets locaux.** On appelle *sommet* toute altitude strictement supérieure à celle qui la précède *et* à celle qui la suit. Écrire `nb_sommets : int list -> int` qui renvoie le nombre de sommets rencontrés.
   Ex. : sur l'exemple, Alice rencontre trois sommets (atteints aux indices 3, 5 et 7).

7. **Altitude minimale.** Alice souhaite également connaître le point le plus bas atteint au cours de sa marche. Écrire `altmin : int list -> int` qui renvoie l'altitude minimale de la liste.
   Ex. : sur l'exemple, la valeur attendue est `0`.

8. **Altitude moyenne.** Pour évaluer la tendance générale de son parcours, Alice veut calculer l'altitude moyenne. Écrire `moyenne : int list -> float` qui renvoie la moyenne des valeurs de `alt`.
   Ex. : sur l'exemple, la moyenne vaut environ `540.`

9. **Écart-type des altitudes.** Afin de mesurer la variabilité de son itinéraire, Alice souhaite calculer l'*écart-type* des altitudes mesurées, c'est-à-dire la racine carrée de la moyenne des carrés des écarts à la moyenne. Écrire `ecart-type : int list -> float` qui renvoie cette valeur.

10. **Médiane des altitudes.** Enfin, pour disposer d'une mesure moins sensible aux valeurs extrêmes, Alice décide de calculer la *médiane* des altitudes, obtenue après tri croissant des valeurs. Écrire `mediane : int list -> float` qui renvoie la médiane de la liste.
    Ex. : sur l'exemple, la médiane vaut `550.`

Exercice 2.7. Plus long bloc de zéros

Écrire une fonction qui, dans une liste constituée de 0 et de 1, renvoie la longueur du plus grand plateau de zéros consécutifs.

Exercice 2.8. Liste aléatoire : éléments manquants

On génère aléatoirement une liste L de longueur n dont les éléments appartiennent à $\{0, \ldots, m-1\}$ (tirages indépendants avec remise).

1. Écrire une fonction qui renvoie le nombre d'éléments de $\{0, \ldots, m-1\}$ absents de L.
2. Estimer par simulation (plusieurs essais) la valeur moyenne de ce nombre en fonction de n et m.

Exercice 2.9. Mélanges parfaits : out-shuffle et in-shuffle

Une méthode traditionnelle pour mélanger un paquet de cartes consiste à couper le paquet en deux moitiés, puis à entrelacer ces deux parties carte par carte. Lorsque les deux moitiés sont de même taille et que l'entrelacement est parfait, on parle de *mélange parfait*.

Il existe deux variantes :

- **Out-shuffle** : on commence par la première carte de la première moitié, puis on alterne avec la seconde moitié.
- **In-shuffle** : on commence par la première carte de la seconde moitié, puis on alterne avec la première moitié.

Exemple (liste de 8 cartes numérotées de 0 à 7) :

`[0;1;2;3;4;5;6;7]`

- L'out-shuffle donne `[0;4;1;5;2;6;3;7]`.
- L'in-shuffle donne `[4;0;5;1;6;2;7;3]`.

Travail demandé :

1. Écrire une fonction `split : 'a list -> 'a list * 'a list` pour décomposer une liste en deux moitiés égales.
2. À partir de cette décomposition, écrire une fonction `out_shuffle : 'a list -> 'a list` qui réalise le mélange en commençant par la première moitié. Vérifier sur l'exemple que l'on obtient `[0;4;1;5;2;6;3;7]`.

3. De même, écrire une fonction `in_shuffle : 'a list -> 'a list` qui commence par la seconde moitié. Vérifier que l'on obtient `[4;0;5;1;6;2;7;3]`.

4. Pour un jeu de 52 cartes, combien de *out-shuffles* successifs faut-il pour retrouver la liste dans son ordre initial ? Même question avec l'in-shuffle.

3 Matrices

Exercice 3.1. Déplacements dans le puzzle de l'âne rouge

On considère le célèbre jeu du *puzzle de l'âne rouge* (ou *Klotski*), dans lequel des pièces rectangulaires de différentes tailles doivent être déplacées dans une grille de 5 lignes et 4 colonnes, afin de libérer la pièce principale (l'âne rouge) par la sortie.

Une configuration du puzzle est représentée sous forme textuelle, par exemple

```
CBBC
CDDC
CDDC
CAAC
AABB
```

où :

- la lettre `A` représente les pièces de taille 1×1 ;
- la lettre `B` représente les pièces de taille 1×2 ;
- la lettre `C` représente les pièces de taille 2×1 ;
- la lettre `D` représente la pièce rouge de taille 2×2.

4. Écrire une fonction `moves : state -> state list` qui, pour une configuration donnée, renvoie la liste de toutes les configurations accessibles par un déplacement valide. On veillera à ce que les configurations renvoyées soient triées.

   Indication : Commencer par remplir une matrice 5×4 de booléens indiquant les cases occupées. On pourra tirer avantage du fait que les pièces ne dépassent jamais la taille 2×2.

4 Récursivité

Exercice 4.1. Somme d'une liste : for, while, récursif

Écrire trois fonctions qui calculent la somme des éléments d'une liste : en utilisant une boucle `for`, une boucle `while` et la récursivité.

Exercice 4.2. Application d'une fonction à toute une liste

Écrire une fonction `on_all` qui applique une fonction à tous les éléments d'une liste. Utiliser cette fonction pour afficher les vingt premiers carrés parfaits.

Exercice 4.3. Deux listes : égalité, inclusion, opérations

Étant données deux listes (de même type), écrire les fonctions suivantes :

1. Tester l'égalité de deux listes,
2. Tester l'inclusion d'une liste dans une autre,
3. Calculer l'union, l'intersection et la différence entre les deux listes,
4. **Concaténation.** Écrire une fonction qui concatène les deux listes (on conserve l'ordre),
5. **Entrelacement (interleaving).** Écrire une fonction qui combine les deux listes en alternant les éléments. Par exemple, $[a, b, c], [1, 2, 3] \to [a, 1, b, 2, c, 3]$.
6. **Fusion triée (merge).** Écrire une fonction qui fusionne les deux listes triées en une nouvelle liste triée. Par exemple, $[1, 4, 6], [2, 3, 5] \to [1, 2, 3, 4, 5, 6]$.

Exercice 4.4. Rotation d'une liste

Écrire une fonction qui effectue une rotation d'une liste de k éléments. Par exemple, $[1, 2, 3, 4, 5, 6]$ tournée de deux devient $[3, 4, 5, 6, 1, 2]$.

Exercice 4.5. Somme des carrés

Écrire un algorithme qui calcule la somme des carrés des n premiers entiers $S_n = \sum_{k=1}^{n} k^2$.

Exercice 4.6. -Factorielle et coefficients binomiaux-

On suppose $n \geq 0$ et $0 \leq k \leq n$.

1. Écrire une fonction `factorielle : int -> int` qui calcule $n!$ sans utiliser la récursivité.
2. Écrire une fonction `factorielle_rec : int -> int` qui calcule $n!$ en version récursive.
3. Écrire une fonction `binomial_fact : int -> int -> int` qui calcule $\binom{n}{k}$ en utilisant `factorielle`.

4. Écrire une fonction `binomial_rec : int -> int -> int` qui calcule $\binom{n}{k}$ récursivement, en utilisant la relation de Pascal :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{pour } n \geq 1,\ 1 \leq k \leq n-1.$$

5. Écrire une fonction `binomial_iter : int -> int -> int` qui calcule $\binom{n}{k}$ de manière itérative, en construisant progressivement le triangle de Pascal (par exemple sous forme d'une liste de listes).

Exercice 4.7. -Fibonacci-

On définit la suite (u_n) par $u_0 = 0$, $u_1 = 1$, et pour tout $n \geq 0$,

$$u_{n+2} = u_{n+1} + u_n.$$

C'est la *suite de Fibonacci*. Les premiers termes sont $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

1. Définir une fonction récursive naïve `fib_rec : int -> int`.
2. Ajouter des impressions pour visualiser la pile d'appels.
3. Pourquoi `fib_rec 1200` est-elle impossible à calculer ?
4. Estimer le temps pour calculer u_{42} avec `fib_rec`.
5. Écrire une version itérative `fib_iter` et une version récursive terminale `fib_tail`.
6. Comparer les performances des trois versions.

Exercice 4.8. -Hanoï-

On dispose de trois piquets et de n disques de tailles différentes, empilés par ordre croissant de bas en haut sur le premier piquet. Règles du jeu :

- déplacer un disque à la fois,
- ne jamais poser un grand disque sur un plus petit.

Problème : déplacer toute la tour du premier piquet vers un autre, en respectant les règles.

1. Définir une fonction `hanoi` qui renvoie la liste des coups à effectuer.
2. Indiquer le type de récursivité utilisée.
3. Montrer par récurrence que le nombre minimal de coups est $2^n - 1$.
4. Vérifier expérimentalement que l'algorithme produit bien $2^n - 1$ coups.

5. Que se passe-t-il si $n = 64$ (ordre de grandeur du temps si on joue 1 coup par seconde) ?

Exercice 4.9. Manipulation de listes typées

Étant donnée la définition :

```
type t = B | N | R
```

Écrire les fonctions suivantes (de manière récursive ou à l'aide d'un itérateur) :

1. **Permutation des symboles.** `permute : t list -> t list` renvoie une nouvelle liste dans laquelle les valeurs `B` sont remplacées par des `N`, les `N` par des `R`, et les `R` par des `B`.
2. **Comptage des symboles.** La fonction récursive terminale `compte_B : t list -> int` compte le nombre de `B` dans la liste passée en paramètre.
3. **Séquence maximale.** La fonction `plus_grande_sequence : t list -> int` renvoie la longueur de la plus grande séquence consécutive de `B`. *Exemple :* `plus_grande_sequence [B; N; N; B; B; R; N; N; B; B; R]` renvoie `3`.

5 Fonctions d'ordre supérieur

Exercice 5.1. Conversion de bases

Écrire une fonction qui prend une liste de chiffres, une base de départ b_1 et une base cible b_2, interprète la liste comme un nombre en base b_1 et le convertit en base b_2. Par exemple, $[2, 1, 0]$ en base 3 est converti en $[2, 1]$ en base 10.

Exercice 5.2. Grands entiers représentés par des listes de chiffres

Dans ce problème, on représente un entier naturel non pas sous forme numérique, mais comme une *liste de ses chiffres décimaux*. Par exemple :

$$1234 \quad \longleftrightarrow \quad [1; 2; 3; 4].$$

Cette représentation permet de manipuler des nombres beaucoup plus grands que ceux gérés nativement par le type `int` d'OCaml.

Objectif. Écrire des fonctions qui effectuent les opérations arithmétiques de base (addition, soustraction, multiplication) sur ces entiers représentés sous forme de listes de chiffres, et qui renvoient le résultat sous la même forme.

1. **Addition.** Écrire une fonction `add_list : int list -> int list -> int list` qui additionne deux listes de chiffres et retourne une nouvelle liste correspondant à leur somme. *Ex. :* $[1; 2; 3] + [9; 9] = [2; 2; 2]$.

2. **Soustraction.** Écrire `sub_list : int list -> int list -> int list` qui soustrait la deuxième liste de la première (en supposant que le résultat est positif). *Ex. :* $[2;0;0] - [1;2;3] = [7;7]$.

3. **Multiplication (naïve).** Écrire `mul_list : int list -> int list -> int list` qui implémente la multiplication "à la main", comme on la ferait sur papier, en multipliant chaque chiffre de la première liste par chaque chiffre de la seconde.

4. **(Optionnel) Multiplication de Karatsuba.** Pour les très grands entiers, la méthode précédente devient lente : elle effectue $O(n^2)$ multiplications élémentaires pour deux nombres de n chiffres.

   En 1962, le mathématicien russe **Anatoly Karatsuba** a découvert un algorithme beaucoup plus rapide, appelé *multiplication de Karatsuba*, qui réduit la complexité à $O(n^{\log_2 3}) \approx O(n^{1.585})$.

   Le principe repose sur une décomposition astucieuse : pour deux entiers x et y de n chiffres, on les écrit comme

   $$x = 10^m a + b, \quad y = 10^m c + d, \quad \text{avec } m = \lfloor n/2 \rfloor,$$

   puis on utilise la formule :

   $$x \times y = 10^{2m}(a \times c) + 10^m\big((a+b)(c+d) - ac - bd\big) + bd.$$

   Cette méthode nécessite seulement trois multiplications récursives de nombres de taille $n/2$ (au lieu de quatre dans la méthode classique).

   Exemple : Soient $x = 1234$ et $y = 5678$. On sépare : $a = 12, b = 34, c = 56, d = 78$. Les trois produits intermédiaires sont :

   $$ac = 12 \times 56, \quad bd = 34 \times 78, \quad (a+b)(c+d) = 46 \times 134.$$

   En combinant ces résultats selon la formule ci-dessus, on retrouve le produit complet $1234 \times 5678 = 7006652$.

   Écrire une fonction `karatsuba : int list -> int list -> int list` qui implémente cet algorithme récursif sur des listes de chiffres.

Exercice 5.3. -Table de multiplication-

Écrire une fonction `table_mul : int -> unit` qui, pour un entier n, affiche la table de multiplication de n de 0 à 9 (inclus), sous la forme :

```
k x n = (k*n)
```

pour chaque k de 0 à 9, un résultat par ligne.

Ex. : pour $n = 7$, le programme affiche notamment
`0 x 7 = 0`,
⋮,
`9 x 7 = 63`.

Exercice 5.4. -Multiples de 3 et 5-

Écrire un programme interactif qui :

1. Demande à l'utilisateur un entier naturel $n \geq 0$.
2. Calcule la somme des entiers i entre 1 et n (inclus) tels que i soit multiple de 3 *ou* de 5.
3. Affiche le résultat au format :
   `Somme des multiples de 3 et 5 de 1 à n :  <valeur>`

Exemple : pour $n = 10$, il doit emprimer: "Somme des multiples de 3 et 5 de 1 à 10 : 33"

Exercice 5.5. -Somme ou produit de 1 à n-

Écrire un programme interactif qui :

1. Demande à l'utilisateur un entier naturel $n \geq 0$.
2. Demande ensuite à l'utilisateur de choisir l'opération :
   entrer `"P"` pour le **produit** ou `"S"` pour la **somme**.
3. Calcule, selon le choix :
   - la somme $\sum_{i=1}^{n} i$ si l'utilisateur saisit `"S"`;
   - le produit $\prod_{i=1}^{n} i$ si l'utilisateur saisit `"P"`.
4. Affiche le résultat au format : `Résultat :  <valeur>`.

Remarques :

- Par convention, le produit vide ($n = 0$) vaut 1 et la somme vide vaut 0.
- Toute autre réponse que `"P"` ou `"S"` est considérée invalide.

Exercice 5.6. Sous-liste maximale – Algorithme de Kadane

Le problème de la *sous-liste maximale* consiste à trouver, dans une liste d'entiers (positifs ou négatifs), la sous-liste ayant la plus grande somme. Par sous-liste, on entend ici une liste d'éléments *contigus*.

Exemple. Pour la liste d'entiers `[-2; 1; -3; 4; -1; 2; 1; -5; 4]`, la sous-liste ayant la plus grande somme est `[4; -1; 2; 1]`, de somme 6.

Pour résoudre ce problème, Jay Kadane (Carnegie Mellon University) a proposé en 1984 un algorithme efficace ne parcourant la liste qu'une seule fois. Le fonctionnement récursif est donné par :

- Si ℓ est vide, la plus grande somme vaut 0.
- Si $\ell = [v_1; v_2; \ldots; v_n]$, et m_{i-1} est la plus grande somme se terminant à l'indice $i - 1$, alors $m_i = \max(v_i, m_{i-1} + v_i)$.

1. Écrire une fonction `max_kadane : int list -> int` qui implémente cet algorithme et renvoie la plus grande somme d'une sous-liste.

   Indication : on peut utiliser une fonction auxiliaire `ma_liste : ('a -> 'a -> int) -> 'a list -> 'a` renvoyant le plus grand élément d'une liste selon une fonction de comparaison donnée.

2. Étendre la fonction précédente pour écrire `kadane : int list -> int list`, qui renvoie la sous-liste elle-même, pas seulement sa somme maximale.

Exercice 5.7. Quantificateur universel sur un intervalle

On souhaite écrire une fonction `for_all : int -> int -> (int -> bool) -> bool` qui vérifie que `p(k)` est vrai pour tous les entiers $i \leq k < j$.

1. Première version :

   ```
   let for_all i j p =
     List.fold_left (fun b k -> b && p k) true (interval i j)
   ```

   Expliquer pourquoi cette version n'est pas une bonne idée.

2. Deuxième version :

   ```
   let for_all i j p =
     List.for_all p (interval i j)
   ```

   Expliquer en quoi cette version est meilleure, mais encore imparfaite.

6 Programmation impérative en OCaml

Exercice 6.1. Somme croissante jusqu'à 100

Écrire un programme qui utilise une boucle `while` pour calculer et afficher la somme croissante des entiers à partir de 0, jusqu'à ce que la somme atteigne ou dépasse 100.

On utilisera deux références :

- `acc` pour la somme courante,
- `k` pour le compteur.

Exercice 6.2. Inversion en place d'un tableau

Écrire une fonction `rev_inplace : int array -> unit` qui inverse un tableau d'entiers *en place*, c'est-à-dire sans allouer de nouveau tableau. *Indication :* utiliser deux indices `i` et `j` et une boucle `while` pour permuter `a.(i)` et `a.(j)` jusqu'à `i >= j`.

Exercice 6.3. Tri impératif et compteurs

Écrire une fonction `insertion_sort : int array -> int * int` qui trie un tableau d'entiers *en place* par insertion, et renvoie un couple `(comparaisons, echanges)` donnant le nombre total de comparaisons et d'échanges effectués. *Contraintes :* utiliser des références pour les compteurs ; boucles `for`/`while` ; pas de fonctions de tri de la bibliothèque.

Exercice 6.4. Matrices impératives

On représente une matrice d'entiers par un tableau de tableaux `int array array`.

1. Écrire `mat_add : int array array -> int array array -> int array array` qui renvoie la somme de deux matrices de même dimension (allouer un nouveau résultat).
2. Écrire `mat_mul : int array array -> int array array -> int array array` qui renvoie le produit $A \times B$. Lever `Invalid_argument` en cas de dimensions incompatibles.
3. Écrire `mat_rand : int -> int -> int -> int -> int array array` qui crée une matrice aléatoire de taille `m x n` avec des valeurs uniformes dans `[lo, hi]`.

Indication : triple boucle pour la multiplication ; attention aux accès `a.(i).(k)` et `b.(k).(j)`.

Exercice 6.5. Histogramme de mots avec `Hashtbl`

Écrire un programme qui lit un fichier texte (nom passé dans `Sys.argv` ou sinon `stdin`), calcule la fréquence des mots (insensibilité à la casse ; séparer sur tout caractère non alphabétique), puis affiche les **10** mots les plus fréquents avec leur compte, triés par fréquence décroissante puis ordre alphabétique. *Contraintes :* utiliser `Hashtbl.t` pour cumuler les comptes ; allouer un tableau temporaire pour trier le résultat ; pas de dépendances externes.

Exercice 6.6. File d'attente interactive (module `Queue`)

Écrire un mini-interpréteur qui maintient une file d'attente d'entiers à l'aide de `Queue.t`. Il lit des commandes sur `stdin` et réagit comme suit :

- `ENQUEUE x` : ajoute `x` en fin de file ;
- `DEQUEUE` : retire et affiche l'élément en tête, ou affiche `EMPTY` si la file est vide ;

- `PEEK` : affiche l'élément en tête sans le retirer, ou `EMPTY` si vide ;
- `SIZE` : affiche la taille courante ;
- `CLEAR` : vide la file ;
- `QUIT` : termine le programme.

Indication : utiliser une boucle `while true` avec `read_line ()`, un `match` sur les mots de la ligne, et les opérations du module `Queue`.

Exercice 6.7. Exceptions et erreurs d'exécution

Tester et analyser le comportement du code suivant :

```
print_endline "avant";
let n = 3 / 0 in
print_endline "après";
print_int n;
print_newline ()
```

Que se passe-t-il à l'exécution ? Pourquoi l'instruction `"après"` n'est-elle pas affichée ?

Exercice 6.8. Définition et levée d'exceptions

Définir et tester les fonctions suivantes :

```
let fonction_erreur x =
  raise (Invalid_argument "argument incorrect")

let failwith s =
  raise (Failure s)
```

1. Expliquer la différence entre les exceptions `Invalid_argument` et `Failure`.
2. Tester ces exceptions dans une fonction `max : 'a list -> 'a` définie comme suit :

   ```
   let rec max l = match l with
     | [] -> raise (Invalid_argument "liste vide")
     | [t] -> t
     | t::q -> ...
   ```

3. Expliquer comment intercepter une exception pour éviter l'arrêt du programme.

Exercice 6.9. Lecture sécurisée avec `try ... with`

Écrire un programme qui demande à l'utilisateur son âge, puis :

- lit la saisie à l'aide de `read_int ()` ;

- intercepte les erreurs de conversion (`Failure "int_of_string"`) ;
- affiche un message différent selon l'âge saisi.

Exemple de comportement attendu :

```
Ton âge : dix-huit
Tape ton âge correctement
Ton âge : 17
Tu es mineur
Ton âge : 25
Tu es majeur
```

Exercice 6.10. Lecture d'un fichier ligne par ligne

Écrire un programme qui ouvre le fichier `"nouvelle.txt"` et affiche son contenu ligne par ligne. À la fin du fichier, le programme doit afficher le message `"Finito"`.

Indication : utiliser `open_in`, `input_line`, et intercepter l'exception `End_of_file`.