Informatique *TD*
Programmation fonctionnelle avec OCaml Ipesup

1 Récursion

Exercice 1 — Somme et produit d'une liste

Écrire des fonctions récursives sur les listes d'entiers :

- `sum : int list -> int`
- `prod : int list -> int` (le produit de la liste vide vaut 1)
- `sum_prod : int list -> int * int` (une seule récursion pour calculer les deux à la fois)

Objectifs : distinguer cas de base et cas récursif ; éviter les parcours multiples.

Exercice 2 — Pair / Impair et récursion mutuelle

- Définir `even : int -> bool` et `odd : int -> bool` par récursion mutuelle.
- Définir `count_even_odd : int list -> int * int` qui renvoie le nombre de pairs et d'impairs en utilisant uniquement `even`/`odd`.
- Écrire `filter_even : int list -> int list` sans utiliser l'opérateur `mod`.

Objectifs : usage du mot-clé `and`, mutualisation des fonctions, traitement des entiers négatifs.

Exercice 3 — Journal du marcheur

Un marcheur part du niveau 0. On a une liste d'instructions `["UP"; "UP"; "DOWN"; ...]`.

Chaque "UP" augmente l'altitude de 1, chaque "DOWN" la diminue de 1, toute autre chaîne est ignorée. On souhaite calculer :

(i) l'altitude finale,
(ii) l'altitude maximale atteinte,
(iii) le nombre de vallées traversées (où l'on descend sous 0 puis on revient à 0).

Écrire une fonction **récursive terminale** qui parcourt la liste une seule fois et retourne le triplet `(altitude_finale, altitude_max, vallees)`.

Objectifs : conception d'accumulateurs, détection de transitions, preuve de récursion terminale.

2 Polymorphisme et ordre supérieur

Exercice 4 — `last_opt` et `rev_append`

- Implémenter `last_opt : 'a list -> 'a option` (renvoie `None` si la liste est vide).
- Implémenter `rev_append : 'a list -> 'a list -> 'a list` qui calcule `rev xs @ ys` en temps linéaire et de manière terminale.
- Vérifier les types inférés : les fonctions doivent être polymorphes.

Objectifs : usage du type `option`, récursion terminale, généralité des types.

Exercice 5 — Réductions et fonctions en argument

- Implémenter `fold_left : ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b`.
- À l'aide de `fold_left`, définir `length`, `sum_int`, `concat_strings`.
- Écrire `fold_map : ('b -> 'a -> 'b * 'c) -> 'b -> 'a list -> 'b * 'c list`.

Objectifs : sens des arguments, réutilisation d'ordre supérieur, typage générique.

Exercice 6 — Pipeline de traitement de données

On dispose d'une liste d'enregistrements `{name : string; score : int}`. Objectif : construire une *pipeline fonctionnelle* qui :

(i) filtre les enregistrements avec `score < t`,
(ii) met la première lettre du nom en majuscule,
(iii) produit :
  — la somme des scores filtrés,
  — la liste des noms triés par score décroissant,
  — une fonction `topk k` renvoyant les k premiers noms.

Implémenter des fonctions réutilisables : `filter`, `map`, `compose`. (N'utiliser ni `List.filter` ni `List.map` de la bibliothèque standard.)

Objectifs : composition fonctionnelle, modularité, clarté du code.

3 Aspects impératifs

Exercice 7 — Registre d'étudiants avec champ mutable

- Définir `type student = { id:int; mutable age:int }`.
- Écrire `birthday : student -> unit` qui incrémente l'âge.
- Écrire `ages : student list -> int array`.
- Écrire `inc_all : student list -> unit` qui ajoute 1 à tous les âges.

Objectifs : champs mutables, mise à jour in-place, conversion liste → tableau.

Exercice 8 — Itérateurs sur tableaux et références

- Pour un `t : int array`, écrire :
  - `exists_lt : int -> int array -> bool` avec `Array.exists` ou une boucle `while`.
  - `for_all_even : int array -> bool` avec `Array.for_all` ou une boucle `for`.
- Écrire `swap_if : int ref -> int ref -> unit` qui échange les contenus si `!a > !b`.
- Expliquer brièvement pourquoi la fonction peut modifier les valeurs malgré le passage par valeur.

Objectifs : compréhension des références, aliasing, modèle mémoire d'OCaml.

Exercice 9 — Simulateur de caisse « Self-Checkout »

Un kiosque vend des produits identifiés par des codes `string`. Chaque commande est une chaîne de la forme : `"ADD code q"`, `"SELL code q"`, `"PRICE code p"`, `"STOCK code"`, `"TOTAL"`, `"QUIT"`.

On souhaite écrire un simulateur maintenant l'état d'un inventaire.

Spécifications implicites :

- `type item = { price:int ref; stock:int ref }`.
- Boucle principale `while true do ... done` lisant les commandes et mettant à jour l'état.
- Exceptions personnalisées : `Unknown_code`, `Negative_qty`, `Out_of_stock`, `Bad_price`.

Exigences :

(i) `ADD` enregistre un nouveau produit.
(ii) `SELL` décrémente le stock (ou lève une exception).
(iii) `PRICE` met à jour le prix.
(iv) `STOCK` affiche la quantité.
(v) `TOTAL` affiche la valeur totale de l'inventaire.

Objectifs : gestion d'état, exceptions, boucles impératives et I/O simulées.

Exercice Bonus — Explorer `Array` et `Matrix`

Cartographie des fonctions `Array` :

A.1) À partir de la documentation (ou de l'environnement `utop`), dresser une table de correspondance entre des fonctions de `List` et leurs équivalents pour `Array`.

A.2) Écrire deux versions d'une transformation :
(i) en style liste avec `List.map`,
(ii) en style tableau avec `Array.mapi`.

Le module *Matrix* n'existe pas. On modélise une matrice 2D comme un tableau de tableaux :

```
type 'a matrix = 'a array array.
```

Proposer un **petit module** `Matrix` (fichier unique) avec au minimum :

- **matrice** — `type 'a t = 'a array array`
- **dimensions** — `dims : 'a t -> int * int`
  Rôle : renvoie $(n\_lignes, n\_colonnes)$.
- **accès** — `get : 'a t -> int -> int -> 'a`
  Rôle : lit l'élément à la position (i, j).
- **écriture** — `set : 'a t -> int -> int -> 'a -> unit`
  Rôle : écrit `v` à la position (i, j) en place (modifie la matrice).
- **initialisation** — `init : int -> int -> (int -> int -> 'a) -> 'a t`
  Rôle : crée une matrice de m lignes et n colonnes telle que la case (i, j) vaille `f i j`.
- **application** — `map : ('a -> 'b) -> 'a t -> 'b t`
  Rôle : construit une nouvelle matrice en appliquant `f` à chaque case, sans modifier l'originale.
- **application indexée** — `mapi : (int -> int -> 'a -> 'b) -> 'a t -> 'b t`
  Rôle : comme `map`, mais `f` reçoit aussi les indices (i, j).
- **itération** — `iter : ('a -> unit) -> 'a t -> unit`
  Rôle : parcourt la matrice (ordre ligne majeure) et exécute `f` pour chaque valeur (effets de bord admis).
- **itération indexée** — `iteri : (int -> int -> 'a -> unit) -> 'a t -> unit`
  Rôle : comme `iter`, mais `f` reçoit aussi (i, j).
- **réduction** — `fold_left : ('b -> 'a -> 'b) -> 'b -> 'a t -> 'b`
  Rôle : agrège toutes les cases en un accumulateur (ligne par ligne, de gauche à droite).
- **transposition** — `transpose : 'a t -> 'a t`
- **conversion (entrée)** — `of_listlist : 'a list list -> 'a t`
  Rôle : transforme une liste de listes (supposée rectangulaire) en matrice ; échoue si les longueurs diffèrent.
- **conversion (sortie)** — `to_listlist : 'a t -> 'a list list`
  Rôle : transforme la matrice en liste de listes, en préservant l'ordre des éléments.

Petit test d'usage :

1. Créer une matrice `m = init 3 4 (fun i j -> 10*i + j)` puis calculer sa transposée.
2. Écrire `map (fun x -> x*x) m` et `fold_left (+) 0 m`.