Informatique *TD*
Programmation fonctionnelle avec OCaml - Données structurées Ipesup

Exercise 1

Écrire trois fonctions :

`fst2 : a * b -> a`, `snd2 : a * b -> b`, `swap : a * b -> b * a`

en utilisant uniquement le filtrage par motif. Tester sur `(1, "ocaml")`.

Exercise 2

Soit `t = ('a', 1.2, (true, 0))`.
1) Par `let`-motifs, extraire `'a'` et `0` dans des variables `x` et `y`.
2) Écrire `third : a * b * c -> c`.
3) Écrire `reorder : a * b * c -> c * a * b`.

Exercise 3

Donner le type des trois valeurs suivantes et écrire des fonctions de conversion explicites entre elles :

`x1 : int * int * int`, `x2 : (int * int) * int`, `x3 : int * (int * int)`

Expliquer pourquoi ces types représentent des mises en mémoire distinctes.

Exercise 4

Définir

`type complex = { re : float ; im : float }`.

Écrire `add`, `mul`, `mod2` (module au carré) sur ce type. Illustrer l'accès aux champs via la *notation pointée* et via *let*-motifs.

Exercise 5

Créer `c = {re = 1.2; im = 0.5}` puis définir `c` en ne changeant que `im` en `-0.5` via la notation `{ c with im = -0.5 }`. Vérifier que `c.re = c'.re`. Créer `c2 = { re = 1.2 ; im = 0.5 }` et `c3 = { im = 0.5 ; re = 1.2 }`.
1) Vérifier que `c2 = c3`.
2) Discuter brièvement pourquoi l'ordre des *champs renseignés* lors de la construction n'influe pas sur l'égalité structurelle.

Exercise 6

Définir deux types

`type t1 = { a:int; b:int }`, `type t2 = { b:int; a:int }`.

Construire trois valeurs `v1,v2,v3` pour chaque type (mêmes champs, ordre de construction quelconque) et comparer `v1 < v2 < v3`.
Expliquer en quoi l'ordre des *champs dans la définition du type* influe sur l'ordre lexicographique.

Exercise 7

Définir

```
type address = { street:string; city:string; postal_code:int }
type date = { day:int; month:int; year:int }
type form = { last_name:string; first_name:string; address:address; birthday:date }
```

Écrire : `full_name`, `same_city`, `age_in_year : form -> int -> int` (âge à la fin d'une année donnée), et `move_to_city : form -> string -> form` qui met à jour la ville (avec `with`).

Exercise 8

Définir `type enseigne = Pique | Coeur | Carreau | Trefle`.
1) Écrire `symbole : enseigne -> string`.
2) Écrire `est_noire : enseigne -> bool` en utilisant un *motif ou* (`|`) pour regrouper `Pique` et `Trefle`.
3) Vérifier l'exhaustivité du `match`. *(Observer l'avertissement si un cas manque.)*

Exercise 9

Définir `type carte = Roi | Reine | Valet | Point of int`.
1) Écrire `valeur : carte -> int` en considérant `Roi=13, Reine=12, Valet=11, Point x = x`.
2) Écrire `est_as : carte -> bool` (vrai si `Point 1`).
3) Écrire `to_string : carte -> string`. Tester sur `[Roi; Point 4; Point 1; Valet]`.

Exercise 10

Définir

`type coord = { x:int; y:int }`, `type forme = Point of coord | Cercle of coord * int`.

1) Écrire `centre : forme -> coord`.
2) Écrire `deplace : forme -> int -> int -> forme` qui translate d'un vecteur `(dx,dy)`.
3) Écrire `rayon : forme -> int option`. *(Renvoie `None` pour un `Point`.)*

Exercise 11

Soit `p3 = ((1,2), ('a', 4.9))`.
1) Par `let ((_, y), _) = p3` récupérer uniquement `y`.
2) Donner un exemple d'avertissement `Warning 11 [redundant-case]` avec un `match` contenant un motif jamais atteint, et corriger.

Exercise 12

En repartant du type `forme` (ex. 10), écrire :

```
est_sur_axeX : forme -> bool,

distance_centre_origine : forme -> float,

projX : forme -> int.
```

Implémenter chaque fonction par `match`-with et motifs en profondeur.

Exercise 13

Écrire `partition_formes : forme list -> (coord list) * ((coord * int) list)` qui renvoie séparément la liste des centres des `Point` et la liste `(centre, rayon)` des `Cercle`. Utiliser des compréhensions (`List.filter`/`List.map`) et/ou `match`.