

TD de *Logique et Circuits* n° 2  
(Correction)

## Introduction à Caml (2)

**Exercice 1** On veut définir un type `carte` qui représente un jeu de 32 cartes (quatre couleurs et des valeurs égales à 7, 8, 9, 10, Valet, Dame, Roi, As). Définir un type `couleur` dont les valeurs possibles sont Trèfle, Cœur, Carreau et Pique.

Définir le type `carte` et écrire une fonction de construction pour ce type qui, à un couple `int * couleur` associe une carte (11 → Valet, 12 → Dame, 13 → Roi, 1 → As) et retourne une erreur s'il n'y a pas de valeur correspondante.

On suppose qu'on joue à la belote. À ce jeu, une des couleurs est privilégiée (c'est l'« atout ») et la valeur des cartes de cette couleur est différente :

	As	Roi	Dame	Valet	10	9	8,7
Couleur atout	11	4	3	20	10	14	0
Couleur non atout	11	4	3	2	10	0	0

Écrire une fonction qui, étant données une carte et la couleur de l'atout, renvoie la valeur de la carte.

**Correction :** Comme il est précisé dans les notes de cours, il y a de multiples façons de définir un type carte. En voici une :

```
type couleur = Trefle | Pique | Coeur | Carreau;;
type hauteur = Petite of int | Valet | Dame | Roi | As;;
type carte = Carte of hauteur * couleur;;
```

```
let cree_carte (x,y) =
  if x = 1 || (x >= 7 && x <= 13) then
    match x with
    | 1 -> Carte(As, y)
    | 11 -> Carte(Valet, y)
    | 12 -> Carte(Dame, y)
    | 13 -> Carte(Roi, y)
    | _ -> Carte(Petite x, y)
  else failwith("mauvaise valeur");;
```

```
let valeur_carte c atout =
  match c with
  | Carte(As, _) -> 11
  | Carte(Roi, _) -> 4
  | Carte(Dame, _) -> 3
  | Carte(Valet, x) -> if x = atout then 20 else 2
  | Carte(Petite 10, _) -> 10
  | Carte(Petite 9, x) -> if x = atout then 14 else 0
  | _ -> 0;;
```

**Exercice 2** On considère les expressions suivantes, extraites d'un programme qui peut comporter des définitions antérieures.

```
type nombre = Ent of int | Dec of float;;
```

```
let x =  
  if y = f z (y = h z) then  
    match (g y) with  
    | z, true -> Dec y  
    | _ -> Ent z  
  else Dec (h z);;
```

Déterminer les types des différents identifiants qui apparaissent dans la dernière expression.

**Correction :** Les différents identifiants sont `x`, `y`, `z`, `f`, `g` et `h`.

1. Le type de `x` est celui de l'expression, c'est-à-dire `nombre` (constructeurs `Dec` et `Ent`).
2. Le type de `y` est `float` (`Dec y`).
3. Le type de `z` est `int` (`Ent z`).
4. `f` est une fonction qui s'applique sur une suite de deux éléments, le premier (`z`) est un `int`, le second est un `bool` (test d'égalité). Le type de retour de `f` est `float` (comparaison avec `y`).
5. `g` est une fonction dont l'argument est un `float`. Son type de retour est un couple `int * bool` (`((z,true))`).
6. `h` est une fonction dont l'argument est un `int` et le résultat un `float`.

**Exercice 3** On définit la fonction `applique` par :

```
let applique f a = f a;;
```

Quel est le type de `applique` ? de `applique applique` ?

**Correction :** `applique` est une fonction qui prend une suite de deux arguments et applique le premier (qui est donc une fonction `'a->'b`) au second (de type `'a`). `applique` est donc de type `('a -> 'b) -> 'a -> 'b`.

Pour toute fonction `f`, `applique f` est équivalent à `f`, puisqu'il s'agit d'une fonction qui attend un argument pour lui appliquer `f`. En particulier, `applique applique` est équivalent à `applique` et a donc le même type.

**Exercice 4** Écrire une fonction `compose` qui, à partir d'une suite de deux fonctions `x` et `y` calcule la fonction composée `z` (définie pour tout `t` par  $z(t) = x(y(t))$ ). Quel est le type de `compose` ?

**Correction :** `let compose x y t = x (y t);;` ou `let compose x y = fun t -> x (y t);;` Le type de retour de `y` doit être le même que l'argument de `x`, d'où :

```
compose : ('a->'b) -> ('c->'a) -> 'c->'b
```

**Exercice 5** On utilise la fonction `compose` définie dans l'exercice précédent et on définit le type `fct` et (de manière incomplète) la fonction `f` :

```
type ('a, 'b) fct = Dec of (float -> 'a) | Ent of (int -> 'b);;
```

```
let f g h =
```

```

match (g,h) with
| Dec a, Ent b -> ... (compose a b)
| Dec a, Dec b -> ... (compose b a)
| Ent a, Dec b -> ... (compose a b)
| Ent a, Ent b -> ... (compose ... ...);;

```

Compléter la définition de `f` et donner son type.

**Correction :** Les types de `g` et `h` sont définis par leur emploi. Les constructeurs à compléter dépendent du type du paramètre de la composition, c'est à dire du second argument, donc le constructeur devant `(compose a b)`, par exemple, est le même que celui qui est devant `b`.

1. La première ligne nous apprend que lorsque `h` est un "Ent", il peut être composé avec une fonction qui prend un `float` en argument, dont c'est un `Ent of (int->float)`.
2. De la même manière, la seconde ligne nous indique que `g` doit être un `Dec of (float->float)` : Ce qui, en revenant à la première ligne, nous apprend que si le résultat de `f` est un "Ent", c'est un `Ent of (int->float)`.
3. De la troisième ligne, on déduit `h : Dec of (float->int)`. On en déduit que le type de retour de la seconde ligne est un `Dec of (float->int)`, ce qui implique dans la troisième ligne (qui a le même type de retour) `g : Ent of (int->int)`.
4. Les trois premières lignes permettent de trouver tous les types. Dans la dernière ligne, on a `g : Ent of (int->int)` et `h : Ent of (int->float)`. La seule façon de les composer est de faire : `Ent a, Ent b -> Ent (compose b a);;` On vérifie que le résultat est bien un `Ent of (int->float)`.

Pour résumer :

```

let f g h =
  match (g,h) with
  | Dec a, Ent b -> Ent (compose a b)
  | Dec a, Dec b -> Dec (compose b a)
  | Ent a, Dec b -> Dec (compose a b)
  | Ent a, Ent b -> Ent (compose b a);;

```

```

val f : (float, int) fct -> (int, float) fct -> (int, float) fct = <fun>

```

**Exercice 6** On définit un type énuméré `jour` dont les valeurs sont : `Lundi`, `Mardi`,... et un type énuméré `mois` dont les valeurs sont aussi des constantes : `Janvier`,...

Écrire la fonction `jsuivant` qui calcule le jour suivant. On supposera qu'il existe aussi une fonction `msuivant` pour les mois. Écrire une fonction qui, étant donnés une suite de deux paramètres `mois` et `int` (représentant l'année) rend la longueur du mois (on supposera, pour simplifier, que les années multiples de 4 sont bissextiles).

Définir un type abstrait `date` tel que, pour chaque élément de type `date`, on puisse accéder aux informations suivantes : le jour de la semaine, le numéro du jour dans le mois, le mois et l'année. Écrire une fonction qui, étant donné une date, retourne la date du lendemain. On pourra écrire cette fonction de manière concrète ou abstraite.

**Correction :**

```

type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche;;
type mois =
  | Janvier |Fevrier| Mars | Avril | Mai | Juin
  | Juillet | Aout | Septembre | Octobre | Novembre | Decembre;;

```

```

let jsuivant x =
  match x with
  | Lundi -> Mardi
  | Mardi -> Mercredi
  | Mercredi -> Jeudi
  | Jeudi -> Vendredi
  | Vendredi -> Samedi
  | Samedi -> Dimanche
  | Dimanche -> Lundi;;

```

```

let longmois x t =
  match x with
  | Fevrier -> if t mod 4=0 then 29 else 28
  | Avril -> 30
  | Juin -> 30
  | Septembre -> 30
  | Novembre -> 30
  | _ -> 31;;

```

Le type `date` doit contenir les informations nécessaires pour pouvoir répondre aux opérations d'accès : on peut donc le représenter concrètement par un quadruplet; le jour de la semaine et le mois seront respectivement de type `jour` et `mois`, le numéro dans le mois et l'année seront des entiers. La définition du type abstrait est donc :

```
(* Domaine *)
```

```
type date = Dat of jour * int * mois * int;;
```

```
(* Operation de construction *)
```

```

let cons_date x =
  match x with
  Dat(j, n, m, a) ->
    if n > 0 && n <= (longmois m a)
    then Dat x else failwith "numero de jour incorrect";;

```

```
(* Operations d'accès *)
```

```

let jour_de_date = function Dat (j, n, m, a) -> j
let njour_de_date = function Dat (j, n, m, a) -> n
let mois_de_date = function Dat (j, n, m, a) -> m
let annee_de_date = function Dat (j, n, m, a) -> a

```

```
(* Version abstraite *)
```

```

let demain d =
  let dj = jsuivant (jour_de_date d)
  and m = mois_de_date d
  and dn = njour_de_date d + 1
  and a = annee_de_date d in
  let tm = longmois m a in
  if dn <= tm then cons_date (dj,dn,m,a) else
  if dn > tm + 1 then failwith("le mois est plus court") else
  let dm = msuivant m in
  if m = Decembre
  then cons_date (dj, 1, dm, a + 1)
  else cons_date (dj, 1, dm, a);;

```

```
(* Version concrete *)
let demain d =
  match d with
  | Dat (j, n, m, a) ->
    let dj = jsuivant j
    and tm = longmois m a in
    if n < tm then Dat (dj,n+1,m,a) else
    if n > tm then failwith("le mois est plus court") else
    let dm = msuivant m in
    if m = Decembre
    then Dat (dj, 1, dm, a + 1)
    else Dat (dj, 1, dm, a);;
```

**Exercice 7** Une entreprise de transports publics dispose de trois types de tickets :

- les **coupons** qui ont une durée (en nombre de mois, maximum un an) et un nombre de zones d'utilisation fixés (représenté par un entier  $n \in [1;8]$ , ce qui signifie que le coupon est valable de la zone 1 à la zone  $n$ ),
- les **cartes** qui sont utilisables un nombre fixé de fois dans une période d'un an, mais pour n'importe quel trajet,
- les tickets **simples** qui ne servent qu'à un trajet.

1. Définir le type abstrait **ticket**.

2. Le prix d'un ticket (arrondi à l'euro inférieur) est donné par les formules suivantes :

- coupon :  $20(m + 1)z$  ( $m$  : nbe de mois,  $z$  : nbe de zones),
- carte :  $5 + t/2$ ,
- simple : 1.

Écrire la fonction **prix** en utilisant les opérations abstraites dont on dispose sur les tickets.

3. Écrire une fonction **choix** qui prend comme suite de paramètres la durée d'utilisation, la zone et le nombre de trajets à effectuer, et retourne le ticket le moins cher (s'il est moins cher d'acheter des tickets simples, on retourne un seul ticket simple).

**Correction :**

```
(* Domaine *)
type ticket = Coupon of int * int | Carte of int | Simple;;
```

```
(* Operations de construction *)
let cons_coupon (m, z) =
  if m >= 1 && m <= 12 && z >= 1 && z <= 8 then Coupon(m,z)
  else failwith("mois dans [1;12], zone dans [1;8]");;
```

```
let cons_carte t =
  if t > 0 then Carte t
  else failwith("le nombre de trajets est negatif ou nul");;
```

```
let cons_simple = Simple;;
```

```
(* Operations de test *)
let est_coupon x =
  match x with
  | Coupon _ -> true
  | _ -> false;;
```

```

let est_carte x =
  match x with
  | Carte _ -> true
  | _ -> false;;

(* Operations d'accès *)

let duree x =
  match x with
  | Coupon (m, z) -> m
  | _ -> failwith("Mauvais type de ticket");;

let zone x =
  match x with
  | Coupon (m,z) -> z
  | _ -> failwith("Mauvais type de ticket");;

let nb_trajets x =
  match x with
  | Carte t -> t
  | _ -> failwith("Mauvais type de ticket");;

(* Version abstraite *)
let prix a =
  if est_coupon a then 20 * (duree a + 1) * zone a else
  if est_carte a then 5 + nb_trajets a else 1;;

let choix m z t =
  if m > 12 then failwith "Duree superieure a un an" else
  let co = cons_coupon(m, z)
  and ca = cons_carte(t)
  and si = cons_simple in
  let a =
    if prix co < prix ca then co else ca in
  if t * prix si < prix a then si else a;;

(* Version concrete cf TP *)

```

**Exercice 8** On définit les types suivants :

```

type place = Pl of int * int;;
type deplacement = Dep of (place->place);;

```

On considère un point sur un écran. Une `place` est caractérisée par ses coordonnées. Un déplacement est une opération qui permet d'aller d'une place à une autre en faisant certain nombre de pas horizontaux et un certain nombre de pas verticaux. Ainsi, `d x` est la place atteinte à partir de la place `x` avec le déplacement `Dep d`.

Écrire une fonction qui, à tout couple de places, associe le déplacement qui permet d'aller de la première à la seconde.

Écrire une fonction qui prend comme argument un déplacement et calcule le déplacement inverse.

**Correction :**

```
let mouvement (Pl(a, b), Pl(c, d)) =  
  Dep (function Pl(x,y) -> Pl(x + c - a, y + d - b));;
```

La fonction mouvement retourne une fonction à laquelle on applique le constructeur Dep pour en faire un déplacement. De même,

```
let inverse = function  
| Dep y ->  
  match y (Pl(0, 0)) with  
  | Pl(c, d) -> Dep ( function Pl(a, b) -> Pl(a - c, b - d) );;
```