Option informatique, DS 1 : Corrigé

Exercice 1. *Reconnaître le type.* — `f` est clairement de type `int -> int`.

— `g` est de type `int array -> int -> int`. En effet, `x` est un indice du tableau (donc un `int`), et `t.(x)` est de type `int` car `*` est un opérateur sur les entiers.

— `h` est de type `(int -> 'a) -> 'a`. En effet, on applique la fonction `x` à un `int` (4).

— `i` est de type `((float -> float) -> 'a) -> 'a`.

— `j` est de type `'a list -> 'a -> 'a list`.

— `k` est de type `bool -> int -> int`.

Remarque : pas besoin de préciser `=<fun>` lorsque vous décrivez un type fonctionnel.

Exercice 2. *Fonctionnelles.*

— `let comp f g x = f (g (f x)) ;;` Puisqu'on compose f et g des deux côtés dans $f \circ g \circ f$, le type d'entrée de f est le type de sortie de g est réciproquement. D'où le type `('a -> 'b) -> ('b -> 'a) -> 'a -> 'b`.
Remarque : écrire `let comp f g = f (g f)` est incorrect, car cette fonction a le type `('a -> 'b) -> (('a -> 'b) -> 'a) -> 'b`, incompatible avec le précédent !

— `let mini f g x = min (min (f x) (g x)) 0 ;;` Ici, le type de sortie des deux fonctions est un entier, d'où le type `('a -> int) -> ('a -> int) -> 'a -> int`.
Remarque : `min` est une fonction curryfiée à deux arguments. Donc on ne peut écrire ni `min (f x) (g x) 0` ni `min (f x, 0)` ni un mélange des deux.

Exercice 3. *Pile d'entiers bornée.*

```
let creer_pile c = Array.make (c+1) 0 ;;
let pile_vide p = p.(0) = 0 ;;
let empiler p x =  p.(0) <- p.(0) + 1 ; p.(p.(0)) <- x ;;
let depiler p = p.(0) <- p.(0) - 1 ; p.(p.(0)+1) ;;
let sommet p = p.(p.(0)) ;;
```

Remarque : c'est bien de connaître son cours, savoir l'adapter c'est mieux. Il n'y avait pas de type enregistrement ici, donc pas de champ `nb` ou `t`. Ne pas oublier d'incrémenter ou décrémenter `p.(0)` lorsqu'on modifie un élément. Certains proposent une version où les opérations ne sont manifestement pas en $O(1)$: ça n'aurait aucun intérêt d'implémenter une telle structure. Enfin, dans `depiler p`, certains écrivent `p.(p.(0)) ; p.(0) <- p.(0) - 1 ;;`. C'est incorrect : dans une succession d'insructions, toutes doivent avoir le type `unit` sauf peut-être la dernière.

Exercice 4. *Pile de Fibonacci.*

1. Il est nécessaire de dépiler au moins une fois, car seul le sommet est accessible. Ici, on dépile deux fois, et on fait attention à remettre la pile dans le bon ordre :

```
let maj p =
  let x=depiler p in
  let y=depiler p in
  empiler p y ;
  empiler p x ;
  empiler p (x+y)
;;
```

Remarque : attention à n'utiliser que des fonctions de pile.

2. On appelle simplement la fonction précédente dans une boucle.

```
let pile_fibo n=
  let p = creer_pile (n+1) in
  empiler p 1 ;
  empiler p 1 ;
  for i=2 to n do
    maj p
  done ;
  p
  ;;
```

Remarque : la fonction `maj` précédente ne renvoie rien (le type est `int array -> unit`). La bonne utilisation est celle du corrigé.

3.
```
let afficher p =
  while not (pile_vide p) do
    print_int (depiler p) ;
    print_string " "
  done ;;
```

Remarque : certains écrivent `print_int (sommet p) ; depiler p`. Ce n'est pas correct, car dans une boucle les instructions doivent avoir type `unit`. Ici (comme dans le cours), `depiler p` a comme double effet de sortir un élément de la pile et de s'évaluer en cet élément.

Exercice 5. *Nombres de Catalan.*

```
let rec catalan n = match n with
  | 0 -> 1
  | _ -> (catalan (n-1)) * 2*(2*n-1) / (n+1)
;;
```

Remarque : la division doit être faite en dernier, car $2(2n-1)$ n'est pas toujours divisible par $n+1$, bien que $2c_{n-1}(2n-1)$ le soit, comme indiqué dans l'énoncé.

Exercice 6. *n-ème élément d'une liste.*

```
let rec nth n q=match (n,q) with
  | _, [] -> failwith "pas assez d'éléments"
  | 0,x::_ -> x
  | _,x::r -> nth (n-1) r
;;
```

Remarque : il est préférable de fonctionner par filtrage que d'utiliser `List.hd` et `List.tl`.

Exercice 7. *Deux plus petits éléments d'une liste.*

```
let rec min2list q=match q with
  | [] | [_] -> failwith "pas assez d'éléments"
  | [a;b] -> min a b, max a b
  | x::r -> let a,b=min2list r in if x<a then x,a else if x<b then a,x else a,b
;;
```

Exercice 8. *Section croissante.* On peut donner des réponses de différentes qualités ici. Une première idée consiste à tester toutes les sous-suites du tableau, et garder en mémoire le début et la longueur de la plus longue sous-suite croissante rencontrée. On fait usage d'une fonction annexe dans la solution suivante :

```
let est_croissante t i j =
  (* teste si la portion entre i et j inclus est croissante *)
  let k=ref i in
  while !k<j && t.( !k) <= t.( !k+1) do
    incr k
  done ;
  !k=j
;;

let max_sec t =
  let im, lm = ref 0, ref 1 and n=Array.length t in
  for i=0 to n-1 do
    for j=i+1 to n-1 do
      if j - i + 1>= !lm && est_croissante t i j then (im:=i ; lm:=j-i+1)
    done
  done ;
  !im, !lm
;;
```

La complexité est $O(n^3)$, mais on sent bien que trop d'opérations sont effectuées ici. Déja, plutôt que de tester toutes les séquences démarrant à l'indice i, on peut optimiser en cherchant pour chaque i la plus longue démarrant à l'indice i, en temps $O(n)$. C'est la version suivante :

```
let max_sec t =
  let im, lm = ref 0, ref 1 and n=Array.length t in
  for i=0 to n-1 do
    let j=ref i in
    while !j<n-1 && t.( !j) <= t.( !j+1) do
      incr j
    done ;
    if !lm < !j - i + 1 then (im:=i; lm:= !j - i + 1)
  done ;
  !im, !lm
;;
```

On obtient une amélioration car la complexité est réduite à $O(n^2)$. Enfin, on peut observer qu'un tableau est naturellement partitionné en sous-suites croissantes successives. Si on parcourt le tableau de gauche à droite, lorsqu'on examine un élément :

— soit il est supérieur au précédent, auquel cas la sous-suite croissante « courante » est de taille augmentée de 1 ;

— soit il est strictement inférieur, auquel cas une nouvelle sous-suite démarre avec l'élément en question.

On peut donc obtenir une solution en $O(n)$ (dans le code suivant, `!i` est le début de la sous-suite croissante courante).

```
let max_sec t =
  let n=Array.length t and i=ref 0 and im=ref 0 and lm=ref 1 in
  for j=1 to n-1 do
    if t.(j)>=t.(j-1) then
      (if j - !i + 1 > !lm then (im := !i ; lm := j - !i +1) )
    else i:= j
  done ;
  !im, !lm
;;
```

Remarques additionnelles. Outre les remarques tout au long du corrigé, voici quelques points supplémentaires.

- Listes et tableaux sont différents, il ne faut pas les confondre, et faire attention à respecter l'énoncé. Pour mémoire, un tableau est un type mutable (modifiable), de taille fixée, et l'accès (et la modification) aux éléments se fait en temps $O(1)$ pour tous les éléments via `t.(i)` (ou `t.(i) <- x`). Une liste (chaînée) est statique (immuable). Seul l'élément de tête est accessible (via `List.hd` ou filtrage) et on peut construire une nouvelle liste par extraction de la queue (via `List.tl` ou filtrage) ou rajout d'un nouvel élément via `::`, la liste précédente étant inchangée (car immuable !).
- N'utilisez des références que si nécessaire. Une référence vers un tableau est en général inutile, (on peut toujours modifier les éléments d'un tableau, mais on n'a pas à changer de tableau, en général !).
- `incr` et `decr` sont des fonctions de type `int ref -> unit`. En conséquence de quoi elles ne s'appliquent qu'à des références et pas aux champs d'un enregistrement.
- Encore une fois, lorsqu'on utilise une pile, restreignez vous aux opérations de pile sans rien supposer de la structure.
- Les instructions conditionnelles sont `if ... then ... else ...` et les boucles `for i=d to f do ... done` et `while ... do ... done`. Pas de `if ... do` ou autre !
- Les motifs de filtrage sont :
  — le motif universel `_` ;
  — un identificateur quelconque `x` ;
  — un constructeur de type somme constant `C` ;
  — un constructeur de type somme non constant `C x`, avec x un motif ;
  — un produit cartésien de motifs (n-uplet $(m_1, \dots, m_n)$) ;
  — une liste de motifs `[a; b; c; ..]` ;
  — `x::q` où `x` et `q` sont des motifs ;
  — une disjonction $m_1|m_2$ de motifs.

  et grosso modo c'est tout. Dès que l'expression filtrée colle à un motif, les identificateurs présents dans le motifs sont localement alloués aux valeurs de l'expression, et les instructions à droite du filtrage est exécuté. Si vous voulez faire une comparaison sur valeurs, le rajout de `when` et d'une expression booléenne est utilisable.