Chapitre 1

Correction des exercices

Exercice 1 Dans les deux cas la réponse de l'interprète est :

```
- : int = 2
```

Dans le premier cas, cette réponse provient du fait qu'en Caml les liaisons sont *statiques* : la valeur associée à un nom est fixée définitivement à la première définition de ce nom ; elle ne peut être affectée par des liaisons ultérieures.
Dans le deuxième cas, la réponse provient du fait que les valeurs ne deviennent visibles qu'*après* toutes les déclarations simultanées.

Exercice 2 La première définition est équivalente à :

```
let a =
   let aa = 3 and bb = 2 in
      let aaa = aa + bb and bbb = aa - bb in
         aaa - bbb ;;
```

donc $aa = 3$, $bb = 2$, $aaa = 5$, $bbb = 1$, et la réponse de l'interprète est :

```
a : int = 4
```

La seconde réponse de l'interprète est alors :

```
- : int = 0
```

Exercice 3 Il s'agit tout d'abord de définir une fonction de $\mathscr{F}(\mathbb{Z},\mathbb{Z})$ dans $\mathbb{Z}$; par exemple

$$\left(\begin{array}{rcl} \mathscr{F}(\mathbb{Z},\mathbb{Z}) & \longrightarrow & \mathbb{Z} \\ f & \longmapsto & f(0)+1 \end{array}\right).$$

```
# function f -> f 0 + 1 ;;
- : (int -> int) -> int = <fun>
```

On demande ensuite une fonction de $\mathbb{Z}$ dans $\mathscr{F}(\mathbb{Z},\mathbb{Z})$; par exemple $\left(\begin{array}{rcl} \mathbb{Z} & \longrightarrow & \mathscr{F}(\mathbb{Z},\mathbb{Z}) \\ n & \longmapsto & (x \mapsto x+n) \end{array}\right)$:

```
# function n -> (function x -> x + n) ;;
- : int -> int -> int = <fun>
```

Cette exemple convient aussi pour la troisième question puisque la règle d'associativité à droite du typage indique que le type *int -> int -> int* est équivalent à *int -> (int -> int)*.
Enfin, on nous demande la forme curryfiée d'une application de $\mathbb{Z}\times\mathscr{F}(\mathbb{Z},\mathbb{Z})$ dans $\mathbb{Z}$, par exemple

$$\left(\begin{array}{rcl} \mathbb{Z}\times\mathscr{F}(\mathbb{Z},\mathbb{Z}) & \longrightarrow & \mathbb{Z} \\ (n,f) & \longmapsto & f(n+1)-1 \end{array}\right).$$

```
# fun n f -> f (n + 1) - 1 ;;
- : int -> (int -> int) -> int = <fun>
```

Exercice 4 La première définition correspond à la forme curryfiée d'une application

$$\begin{pmatrix} \mathscr{F}(\mathrm{A}\times\mathrm{B},\mathrm{C})\times\mathrm{A}\times\mathrm{B} & \longrightarrow & \mathrm{C} \\ (f,x,y) & \longmapsto & f(x,y) \end{pmatrix}.$$

```
# fun f x y -> f x y ;;
- : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

La seconde définition correspond à l'application $\begin{pmatrix} \mathscr{F}(\mathrm{A},\mathrm{B})\times\mathscr{F}(\mathrm{B},\mathrm{C})\times\mathrm{A} & \longrightarrow & \mathrm{C} \\ (f,g,x) & \longmapsto & g\circ f(x) \end{pmatrix}$.

```
# fun f g x -> g (f x) ;;
- : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

Enfin, la troisième définition correspond à l'application $\begin{pmatrix} \mathscr{F}(\mathrm{A},\mathbb{Z})\times\mathscr{F}(\mathrm{A},\mathbb{Z})\times\mathrm{A} & \longrightarrow & \mathbb{Z} \\ (f,g,x) & \longmapsto & f(x)+g(x) \end{pmatrix}$.

```
# fun f g x -> (f x) + (g x) ;;
- : ('a -> int) -> ('a -> int) -> 'a -> int = <fun>
```

Exercice 5 Il s'agit de la définition non curryfiée de la fonction $h:(f,g)\mapsto f\circ g$:

```
# let h (f, g) = function x -> f (g x) ;;
h : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
```

La forme curryfiée de cette définition serait : `let h f g = function x -> f (g x)`.

Exercice 6 Dans cet exercice le typage est toujours de la même forme :

type_de_x -> type_de_y -> type_de_z -> type_de_l'expression

Il s'agit donc de déterminer les types de **x**, **y** et **z** pour que l'expression ait un sens.

a) Si on note *'b* de type de **z**, il faut que `x y` soit une fonction de type *'b -> 'c* pour que l'expression `(x y) z` ait un sens. Si on note *'a* le type de **y**, il faut donc que le type de **x** soit *'a -> 'b -> 'c*. D'où :

```
# fun x y z -> (x y) z ;;
- : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

b) Pour que l'expression `x (y z)` ait un sens, il faut que **x** soit une fonction de type *'a -> 'b*, et donc que `y z` soit de type *'a*. Il faut donc que **z** soit de type *'c* et **y** une fonction de type *'c -> 'a*. D'où :

```
# fun x y z -> x (y z) ;;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

c) `x y z` est équivalent à `(x y) z` donc la réponse est identique à celle de la question 1.

d) **x** doit être une fonction de type *'a -> 'b* donc l'expression `y z x`, équivalente à `(y z) x`, doit être de type *'a*. Il est donc nécessaire que `y z` soit de type *('a -> 'b) -> 'a*. Si on note *'c* le type de **z**, le type de **y** est donc *'c -> ('a -> 'b) -> 'a*. D'où :

```
# fun x y z -> x (y z x) ;;
- : ('a -> 'b) -> ('c -> ('a -> 'b) -> 'a) -> 'c -> 'b = <fun>
```

e) Enfin, pour que l'expression `(x y) (z x)` ait un sens, il faut que `x y` soit de type *'b -> 'c* et `z x` de type *'b*. Si on note *'a* le type de **y**, alors **x** doit être de type *'a -> 'b -> 'c*, et donc **z** de type *('a -> 'b -> 'c) -> 'b*. D'où :

```
# fun x y z -> (x y) (z x) ;;
- : ('a -> 'b -> 'c) -> 'a -> (('a -> 'b -> 'c) -> 'b) -> 'c = <fun>
```

Exercice 7 Observons la table de vérité de l'assertion logique $A \Rightarrow B$:

A	B	$A \Rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

On procède alors par filtrage :

```
# let implique a b = match (a, b) with
    | (true, false) -> false
    | _             -> true ;;
implique : bool -> bool -> bool = <fun>
```

Exercice 8 Il n'est pas précisé à quel ensemble appartiennent les éléments de la suite $(u_n)_{n\in\mathbb{N}}$; il est seulement nécessaire que cet ensemble soit muni d'une structure de groupe additif pour pouvoir soustraire u_n à u_{n+1}.

Si on utilise l'opérateur – du type *int*, la suite $(u_n)_{n\in\mathbb{N}}$ sera représentée par un élément de type *int -> int* et l'opérateur Δ par le type *(int -> int) -> int -> int* ; si on utilise l'opérateur **–.** le type de Δ sera *(int -> float) -> int -> float* :

```
# let delta u = function n -> u (n+1) - u n ;;
delta : (int -> int) -> int -> int = <fun>
# let delta u = function n -> u (n+1) -. u n ;;
delta : (int -> float) -> int -> float = <fun>
```

Exercice 9 La version non curryfiée d'une fonction à deux variables doit être de type *'a * 'b -> 'c*, tandis que la version curryfiée doit être de type *'a -> 'b -> 'c*. Cela conduit aux deux définitions suivantes :

```
# let curry f = fun x y -> f (x, y) ;;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let uncurry f = function (x, y) -> f x y ;;
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

Exercice 10 Un entier n est un nombre de Hamming s'il est égal à 1 ou s'il existe un nombre de Hamming p tel que n soit égal à $2p$, $3p$ ou $5p$. Ceci conduit à la définition suivante :

```
let rec hamming = function
  | 1                    -> true
  | n when n mod 2 = 0 -> hamming (n / 2)
  | n when n mod 3 = 0 -> hamming (n / 3)
  | n when n mod 5 = 0 -> hamming (n / 5)
  | _                    -> false ;;
```

Exercice 11 On définit les fonctions récursives suivantes :

```
let rec f = fun
  | 0 q -> q
  | p q -> f (p - 1) q + 1 ;;

let rec g = fun
  | 0 q -> q
  | p q -> g (p - 1) (q + 1) ;;
```

On constate sans peine qu'elles calculent toutes deux la somme des entiers p et q, résultat qui peut par exemple se démontrer par récurrence sur p.

Exercice 12 Pour que cette définition du produit soit valable, il faut rajouter la règle : $0 \times q = 0$, ce qui conduit à la définition :

```
let rec produit = fun
  | 0 q                   -> 0
  | p q when p mod 2 = 0 -> produit (p / 2) (2 * q)
  | p q                   -> produit (p / 2) (2 * q) + q ;;
```

La preuve de validité de cette définition peut se faire par récurrence forte sur p.
Notons que le filtrage n'agissant que sur **p**, on aurait pu écrire :

```
let rec produit p q = match p with
  | 0                   -> 0
  | p when p mod 2 = 0 -> produit (p / 2) (2 * q)
  | p                   -> produit (p / 2) (2 * q) + q ;;
```

Pour calculer q^p, on adopte les règles suivantes : $q^p = \begin{cases} 1 & \text{si } p = 0 \\ q^{p/2} \times q^{p/2} & \text{si } p \text{ est pair} \\ q \times q^{(p-1)/2} \times q^{(p-1)/2} & \text{si } p \text{ est impair} \end{cases}$

```
let rec puissance q = function
  | 0                   -> 1
  | p when p mod 2 = 0 -> let a = puissance q (p / 2) in a * a
  | p                   -> let a = puissance q (p / 2) in q * a * a ;;
```