Interrogation n° 1 — Corrigé

QUESTION 1 Un langage est fortement typé si tous les objets/expressions ont un type, défini sans ambiguïté, et ne peuvent pas en changer sans conversion explicite. Un inconvénient est que l'écriture de code est moins flexible. Un avantage est une plus grande sûreté de programmation et la possibilité d'une analyse mathématique.

QUESTION 2 Les fonctions sont des valeurs, avec un type, et peuvent être utilisées comme toute autre valeur. Il n'y a pas de différence entre une valeur de type fonctionnel et une autre. En particulier, on peut écrire des fonctions qui prennent ou renvoient d'autres fonctions.

QUESTION 3 `expr1` doit être de type **`bool`** ; `expr2` et `expr3` doivent être de même type.

QUESTION 4 Il s'agit d'une application de fonction ; `expr1` doit être de type fonctionnel `'a -> 'b` avec `'a` et `'b` deux types quelconques (ou des paramètres de types) ; `expr2` doit alors être de type `'a` ; l'expression est alors de type `'b`.

EXERCICE 1 On cherche une fonction de type `(float -> float) -> float -> float` (ou un type plus général).

```
let exercice1 f x =
  (f (1. +. x)) *. sqrt (1. +. (f x) ** 2.)
;;
```

On peut bien sûr écrire

```
let exercice1 f =
  function x -> (f (1. +. x)) *. sqrt (1. +. (f x) ** 2.)
;;
```

ce qui revient *exactement* au même !

EXERCICE 2

1. `let addition x y = x + y;;`
2. `let en_zero f = 0 + f 0;;` où on ajoute `0 +` pour assurer un type `int -> int`, mais un type plus général (e.g. `int -> 'a`) était bien sûr accepté.
3. C'est exactement le même type que le premier !

EXERCICE 3

1. `f1 : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c`
2. `f2 : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`
3. C'est exactement la même fonction que la première !

EXERCICE 4 Si $m < n$ alors le produit est vide et par convention vaut 1, le neutre pour la multiplication. Sinon on peut décomposer

$$\prod_{k=n}^{m} f(k) = f(n) \times \prod_{k=n+1}^{m} f(k)$$

On traduit directement cela par un programme CAML, en prenant soin de convertir l'entier n en flottant pour pouvoir appliquer f.

```
let rec produit f n m =
  if m < n then
    1.
  else
    (f (float_of_int n)) *. (produit f (n + 1) m)
;;
```

Si $m < n$ la complexité est en $\Theta(1)$. Sinon, on note $C_f(n)$ le coût de l'appel de la fonction `f` sur un entier n (converti en flottant). On a une relation de récurrence $C(f,n,m) = C(f,n+1,m) + \Theta(1) + C_f(n)$ et on en déduit $C(f,n,m) = \Theta(\sum_{k=n}^{m} C_f(k))$. Si on suppose que l'appel à f est de coût constant, on trouve une complexité dans le pire des cas en $\Theta(m)$.