UNIVERSITÉ D'ORLÉANS **DEUG MIAS**
IF 2.01 - Informatique **Année 2001-2001**

Feuille d'exercices n°1bis

Exercice 1

```
# let g = function (x,y,z) ->
        ((if x then y else 1),
        if y>10 then "exemple" else z^"a");;
val g : bool * int * string -> int * string = <fun>

# let f  = function x -> (function y -> x+y);;
val f : int -> int -> int = <fun>
# let g = f 2;;
val g : int -> int = <fun>
# g 5;;
- : int = 7
# let g1 = function f -> 2*(f (function x-> x+1));;
val g1 : ((int -> int) -> int) -> int = <fun>
# let h = function g -> (g 7)/3;;
val h : (int -> int) -> int = <fun>
# g1 h;;
- : int = 4
# let g = function f -> (function x -> f(function y -> x+y)+x);;
val g : ((int -> int) -> int) -> int -> int = <fun>
# let g = function f -> function x -> f(function y -> x+y)+x;;
val g : ((int -> int) -> int) -> int -> int = <fun>
# g h 6;;
- : int = 10
```

Exercice 2

```
# let rec f n = if n = 0
        then true
        else g(n-1)
      and g n = if n=0
        then false
        else f (n-1);;
val f : int -> bool = <fun>
val g : int -> bool = <fun>
```

`f` est la fonction `est_pair` et `g` est la fonction `est_impair`. Cela se montre aisément par récurrence : notons H_n l'hypothèse de récurrence suivante :

> (H_n) : $f(n)$ est vrai si, et seulement si, n est pair, et $g(n)$ est vraie si, et seulement si, n est impair.

Le résultat est vrai pour $n = 0$. Supposons le vrai à l'ordre n. Alors $f(n+1)$ prendra la valeur de $g(n)$ puisque $n+1$ ne peut pas être égal à 0. De même, $g(n+1) = f(n)$.
Maintenant, supposons n pair. Alors $n+1$ est impair, et on a bien $g(n+1) = f(n) =$ `true` et $f(n+1) = g(n) =$ `false`. Le raisonnement est identique pour le cas où n est impair.
ATTENTION : cette preuve est faite par récurrence croissante, à partir de 0, et n'est donc vraie que pour les entiers positifs. Les fonctions f et g ne sont pas correctement définie pour les entiers négatifs. Il vaut mieux définir les fonctions suivantes :

```
# let pair = f (abs n)
  and impair = g (abs n);;
```

On peut alors s'intéresser à la parité des nombres négatifs :

```
# pair 3;;
- : bool = false
# pair (-23);;
- : bool = false
# pair (-24);;
- : bool = true
```

Exercice 3

```
# let rec exp n = if n =0 then 1 else exp(n-1) + exp (n-1);;
val exp : int -> int = <fun>
```

Cette fonction crée deux « processus fils » à chaque itération. En effet, on lui demande, à chaque fois, de calculer `exp (n-1)` et `exp (n-1)`, et de les additionner. Le calcul de `exp (n-1)` est donc fait deux fois, celui de `(exp (n-2)` sera fait quatre fois, et ainsi de suite. La complexité est alors exponentielle.
Voici une méthode plus rapide :

```
# let double x = x + x;;
val double : int -> int = <fun>
# let rec exp n = if n=0 then 1 else double (exp (n-1));;
val exp : int -> int = <fun>
```

ATTENTION : on peut alors calculer facilement maintenant la valeur de 2^{29} et 2^{30} :

```
# exp 29;;
- : int = 536870912
# exp 30;;
- : int = -1073741824
```

Explication : les entiers en CAML doivent être compris entre -2^{30} et $2^{30} - 1$. On peut alors faire des choses très bizarres, comme par exemple :

```
# - exp 30;;
- : int = -1073741824
# exp 30;;
- : int = -1073741824

# 1073741823+1;;
- : int = -1073741824
```

C'est une limitation qu'ont tous les langages de programmation. Noter que l'utilisation des `floats` permet d'aller au dela de cette limite[2] :

```
# 2.**30.;;
- : float = 1073741824
# 2.**39.;;
- : float = 549755813888
# 2.**40.;;
- : float = 1.09951162778e+12
# 2.**100.;;
- : float = 1.26765060023e+30
```

Exercice 4

```
# let d = function f -> f 0;;
val d : (int -> 'a) -> 'a = <fun>
# d (function x -> x+1);;
- : int = 1
# d (function x-> "argument"^(if x=0 then "" else "non")^" nul");;
- : string = "argument nul"

# let id x = x;;
val id : 'a -> 'a = <fun>
# ((id true), (id 2));;
- : bool * int = true, 2
# let id x = x in (id true), (id 2);;
- : bool * int = true, 2
# let g f = (f true), (f 2);;
This expression [2] has type int but is here used with type bool
```

Noter dans ce dernier exemple que CAML force la fonction `f` à avoir un type `bool -> 'a`, et ne peut donc pas l'appliquer à l'entier 2.

Exercice 5

```
let entier_it f a =
        let rec g = function 0 -> a
                        |    n -> f ( g (n-1))
        in g;;
```

C'est une fonction de type

```
val entier_it : ('a -> 'a) -> 'a -> int -> 'a = <fun>
```

Elle prend en premier argument une fonction `f` d'un type dans lui-même et en deuxième argument un élément `a` de ce même type, et renvoie une fonction qui à un entier `n` associe l'itérée de la fonction sur l'élément, c'est à dire $f^n(a)$.

Exercice 6

[2] La fonction `**` est la fonction puissance, elle est définie sur les flottants.

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

Exercice 7

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