Corrigé de
Programmation fonctionnelle

1 Bases

1.
```
#let p = fun a b p -> p a b;;
val p : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c = <fun>

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

#let s = fun p -> p (fun x y -> y);;
val s : (('a -> 'b -> 'b) -> 'c) -> 'c = <fun>
```

Pour évaluer x, il s'agit d'effectuer des β-réductions :

```
let x = f (s (p 1 (p "deux" '3')))
      = f (s (p 1 ((fun a b p -> p a b) "deux" '3')))
      = f (s (p 1 (fun p -> p "deux" '3')))
      = f (s (fun p' -> p' 1 (fun p -> p "deux" '3')))
      = f ((fun p' -> p' 1 (fun p -> p "deux" '3')) (fun x y -> y))
      = f ((fun x y -> y) 1 (fun p -> p "deux" '3'))
      = f (fun p -> p "deux" '3')
      = (fun p -> p "deux" '3') (fun x y -> x)
      = (fun x y -> x) "deux" '3'
      = "deux" : string
```

Ici p sert à fabriquer une paire, "f" est l'accès au premier élément (`fst`)) et s l'accès au second.

2. Le plus simplement possible :

```
#type t = X of (int * string) list array * float;;
type t = X of (int * string) list array * float

#X ([|[(0, "0")]|],0.0);;
- : t = X ([|[(0, "0")]|], 0)
```

3.
```
#type x = I of int | F of float | S of string * x;;
```

L'itérateur prend 3 fonctions en argument qui correspondent aux 3 constructeurs du type :

```
#let rec iter = fun i f s x ->
#  match x with
#    I int -> i int
#  | F float -> f float
#  | S (string, x') -> s string (iter i f s x');;
val iter : (int -> 'a) -> (float -> 'a) -> (string -> 'a -> 'a) -> x -> 'a =
  <fun>

#let mem = iter (fun _ -> 4) (fun _ -> 8) (fun s x -> String.length s + 4 + x);;
val mem : x -> int = <fun>
```

2 Arithmétique en précision arbitraire

1. Pour additionner, il s'agit de parcourir les 2 listes et d'additionner les chiffre deux à deux ... en tenant compte de la retenue. Pour simplifier l'écriture, j'utilise ici un *pattern disjonctif* pour les deux cas où un nombre est plus court que l'autre.

```
#let somme = fun l1 l2 ->
#  let rec s = fun l1 l2 retenue ->
#    match l1, l2 with
```

```
#      (([], l) | (l, [])) -> if retenue <> 0 then s [retenue] l 0 else l
#     | x::xs, y::ys ->
#       let z = x+y+retenue in
#       z mod base::s xs ys (z / base) in
#   s l1 l2 0;;
 val somme : int list -> int list -> int list = <fun>
```

2. Les fonctions de division entière (/) et reste (mod) sont appliquées récursivement pour fabriquer la listes des chiffres d'un nombre, jusqu'à temps d'obtenir un nombre inférieur à la base.
   Pour la transformation inverse, il s'agit d'ajouter le premier chiffre avec le produit de la base par la transformation du reste.

```
#let rec grand_entier = fun n ->
#  if n < base then [n] else n mod base :: grand_entier (n / base);;
val grand_entier : int -> int list = <fun>

#let rec petit_entier = fun l ->
#  List.fold_right (fun x r -> x + base * r) l 0;;
val petit_entier : int list -> int = <fun>
```

3. L'affichage se réalise en imprimant en séquence les chiffres ... dans l'ordre inverse. On peut utiliser l'itérateur de séquence en renversant auparavant la liste.

```
#let rec afficher = fun l ->
#  match l with
#    [] -> ()
#  | x::xs -> afficher xs; print_int x;;
val afficher : int list -> unit = <fun>

#let afficher' = fun l -> List.iter print_int (List.rev l);;
val afficher' : int list -> unit = <fun>
```

4. On peut effectuer un produit comme « à la main »: multiplication des chiffres du second nombre par le premier et somme avec décalage. Une première fonction permet donc de multiplier un chiffre par un nombre. Comme pour l'addition, il faut propager une retenue (qui est toujours inférieure à la base).

```
#let rec produit_chiffre = fun i c retenue ->
#  match i with
#    [] -> if retenue > 0 then [retenue] else []
#  | x::xs ->
#    let y = x * c + retenue in
#    y mod base :: produit_chiffre xs c (y / base);;
val produit_chiffre : int list -> int -> int -> int list = <fun>
```

   La fonction de produit elle même utilise la fonction précédente et la somme. Le décalage est réalisé par l'ajoût d'un 0 en début de nombre. L'usage d'un itérateur permet d'éviter la fonction récursive.

```
#let rec produit = fun i1 i2 ->
#  match i2 with
#    [] -> []
#  | x::xs -> somme (produit_chiffre i1 x 0) (0::produit i1 xs);;
val produit : int list -> int list -> int list = <fun>

#let produit'= fun i1 i2 ->
#  List.fold_right (fun x r -> somme (produit_chiffre i1 x 0) (0::r)) i2 [];;
val produit' : int list -> int list -> int list = <fun>
```

5. Pour la factorielle, rien d'original mis à part les conversions en grand entier. On peut également utiliser le récurseur vue en cours.

```
#let rec fact = fun n ->
```

```
#  if n = 0 then grand_entier 1 else produit (grand_entier n) (fact (n-1));;
val fact : int -> int list = <fun>

#let rec recurseur = fun f n z ->
#  if n = 0
#  then z
#  else f (n-1) (recurseur f (n-1) z);;
val recurseur : (int -> 'a -> 'a) -> int -> 'a -> 'a = <fun>

#let fact' = fun n -> recurseur (fun n -> produit (grand_entier (n+1))) n (grand_entier 1);;
val fact' : int -> int list = <fun>
```

3 Document structuré

1. Une solution avec deux types, un premier pour le contenu des cadres et un second pour les cadres eux-mêmes.

```
#type item =
#    Texte of string
#  | Titre of string
#  | Enum of item list
#type document =
#    Cadre of document * document
#  | Items of item list;;

#
#let exemple =
#  Cadre (Items [Titre "Titre1"; Texte "Du texte"; Texte "Une enumeration:";
#                Enum [Texte "Une pierre"; Texte "deux ..."; Texte "un raton..."]],
#         Cadre (Items [Texte "xxx"],
#                Items [Enum [Texte "Un"; Texte "Deux";
#                             Enum [Texte "Trois"; Titre "Quatre"]];
#                       Texte "Cinq"]));;
```

2. Deux fonctions récursives pour les deux types. On utilise un paramètre supplémentaire (le *reste*) pour éviter les concaténations. Le traitement des listes est avantageusement effectué avec l'itérateur `fold_right`.

```
#let rec a_plat_item = fun i r ->
#  match i with
#    Texte s | Titre s -> s::r
#  | Enum l -> List.fold_right a_plat_item l r;;
val a_plat_item : item -> string list -> string list = <fun>

#let rec a_plat_doc = fun d r ->
#  match d with
#    Items l -> List.fold_right a_plat_item l r
#  | Cadre (d1, d2) -> a_plat_doc d1 (a_plat_doc d2 r);;
val a_plat_doc : document -> string list -> string list = <fun>

#let a_plat = fun d -> a_plat_doc d [];;
val a_plat : document -> string list = <fun>

#a_plat exemple;;
- : string list =
["Titre1"; "Du texte"; "Une enumeration:"; "Une pierre"; "deux ...";
 "un raton..."; "xxx"; "Un"; "Deux"; "Trois"; "Quatre"; "Cinq"]
```