Exercice 1 On entre successivement les expressions suivantes. Indiquer le résultat de l'évaluation de chacune de ces expressions.

```
let x = 25 in x + 2 * x ;;
x + 1;;
let x = 1 ;;
let x = 4 in x + 2 * x ;;
x + 3 ;;
let t = 3 and y = let z = x + 3 in z + 2 ;;
let y = let z = y + 3 in z + 2 ;;
let x = x + 1 ;;
let x = 5 in let y = 3 + x in let z = y + x in x + y + z ;;
```

Exercice 2 Evaluer les expressions ci-dessous. On supposera qu'à chaque question correspond une nouvelle session OCaml interactive.

1. Manipulation d'expressions

```
4 < 3;;
"debut " ^ "fin";;
4.0 * 6;;
string_of_int 34;;
4.0 >= 45.6;;
4.0 >= 45;;
```

2. Déclarations de booléens

```
let x = 4 < 3;;
let x = x = true;;
```

3. Déclarations simultanées

```
let nb1 = 4 and nb2 = nb1;;
let nb1 = 4;;
let nb2 = nb1;;
let y = 1;;
let y = 3 and z = y;;
```

Exercice 3 Evaluer les expressions ci-dessous. On supposera qu'à chaque question correspond une nouvelle session OCaml interactive.

1.
```
let a = 3 and b = 1;;
let f x = a * x + b;;
f 1;;
let a = 0 and b = 0;;
f 1;;
```

2.
```
let const = 3 ;;
let f x = x * const ;;
f 1 ;;
let const = 6 ;;
f 1 ;;
```

Exercice 4 Evaluer les expressions ci-dessous. On supposera qu'à chaque question correspond une nouvelle session OCaml interactive.

1.
```
let x = false;;
let y = (x = true);;
let z = if x = true then false else true;;
let x = 3;;
let x = false;;
let x = x + 2;;
```

2.
```
if true then 1 else 2 ;;
let a = 5 ;;
let parite = (a mod 2) = 0 ;;
if parite then "pair" else "impair" ;;
if 2 < 4 then 1 else false ;;
let b = 10 ;;
let c = 20 ;;
if a < b
 then if a < c then a
               else c
 else if b < c then b
               else c ;;
if (a < b) & (a < c) then a
else if (b < a) & (b < c) then b
     else c ;;
```

Exercice 5

1. Montrer que les expressions :

   `if` a `then true else` $F(a)$
   `if` a `then` a `else` $F(a)$
   a `or` $F(a)$

   ont les mêmes valeurs pour tout booléen a et toute fonction booléenne F.

2. Montrer que les expressions :

   `if` a `then` $F(a)$ `else false`
   `if` a `then` $F(a)$ `else` a
   a `&` $F(a)$

   ont les mêmes valeurs pour tout booléen a et toute fonction booléenne F.

3. Comparer les valeurs des deux expressions suivantes :

```
if a = b then true else false ;;
a = b ;;
```

Exercice 6 Ecrire une fonction calculant le n-ième terme de la suite de Fibonacci définie par :

$$\begin{cases} F_0 = F_1 = 1 \\ F_{n+2} = F_n + F_{n+1} \end{cases}$$

Exercice 7 On entre successivement les expressions suivantes (tout ce qui est entre `(*` et `*)` est ignoré par OCaml et n'a aucun effet sur les résultats) :

```
let x=7 in x-1;;
        (* 1: x *)
let y = let a=5 in a+a;;
        (* 2: y, a *)
let f x = x*x in (f 3);;
        (* 3: x, f *)
let f x = y-x;;
        (* 4: x, y, f *)
let f x y =
  let tmp = y-x in
  if x>y then tmp
         else -tmp;;
        (* 5: x, y, tmp, f *)
let f x =
  let rec f' z y =
    if z = 1 then y else f' (z-1) (z*y)
  in f' x 1;;
        (* 6: x, z, y, f, f' *)
```

Indiquer, pour chaque définition, le statut de chacune des variables citées en commentaire (entre `(*` et `*)`) : variable globale ou variable locale. On rappelle que les *paramètres* d'une fonction sont des variables *locales* à la fonction : ainsi `x` est local à `f` dans `let f x = x+1`. Si l'on entre toutes ces définitions telles quelles, dans l'ordre ci-dessus, que va donner ensuite l'évaluation de `(f 3)` ? Pourquoi ?

Exercice 8 On entre successivement les expressions suivantes. Indiquer le résultat de l'évaluation et le type de chacune de ces expressions.

```
let surface_rectangle1 = function long -> function larg -> long *. larg;;
let surface_rectangle2 long larg  = long *. larg;;
let surface_rectangle3 (long,larg)  = long *. larg;;
surface_rectangle1(1.1,0.3);;
surface_rectangle3(1.1,0.3);;
surface_rectangle2 1.1 0.3;;
surface_rectangle3 1.1 0.3;;
surface_rectangle2 1.1;;
surface_rectangle3 1.1;;
let x = 1.1;;
let surface_rectangle = surface_rectangle1 x;;
surface_rectangle 0.3;;
let x = surface_rectangle 0.3;;
surface_rectangle 0.3;;
```

Exercice 9 Définir, à l'aide d'une fonction `compose` de composition de deux fonctions, une fonction `fact_o_carre` qui calcule la factorielle du carré d'un nombre entier.

Exercice 10 Définir une fonction `iterer` calculant la n-ième itérée d'une fonction f passée en argument (on rappelle que f^0 est la fonction identité).

$$\texttt{iterer}\ f\ n\ x = f^n(x)$$

Exercice 11 Evaluez les expressions ci-dessous.

```
let xxx = (3,"45",true);;
let zzz = (3, ("45", true)) and ttt = ((3, "45"), true);;
xxx = zzz;;
xxx = ttt;;
let f (x,y,z) = (x, string_of_int x ^ y, not z);;
f xxx;;
```

Exercice 12 Utiliser la fonction `iterer` (définie dans l'exercice 10) pour définir une fonction `fib` calculant le n-ième terme de la suite de Fibonacci de manière efficace, c.a.d. en un temps linéaire en fonction de n et non pas exponentiel comme c'est le cas lorsqu'on utilise la relation de récurrence donnée dans l'exercice 6. On cherchera une fonction f de type `int * int -> int * int` à passer à la fonction `iter`.

Exercice 13

1. Définir une fonction `until` qui étant donnés une fonction f et un prédicat p retourne la fonction telle que :

$$\texttt{until}\ f\ p\ x = f^n(x)$$

   où n est le plus petit entier tel que $p(f^n(x)) = \texttt{true}$.

2. e^x peut être calculé par approximations successives en utilisant la formule :

$$e^x = \sum_{n \geq 0} \frac{x^n}{n!}$$

   Définir une fonction `exp_approx` qui calcule une approximation de e^x à ε près.

3. On se propose dans ce qui suit de calculer la racine carrée d'un réel positif a à l'aide de méthodes itératives permettant de déterminer "le zéro" d'une fonction. Pour ce faire, on pose $f(x) = x^2 - a$ et on cherche la valeur x_s telle que $f(x_s) = 0 = x_s^2 - a$. Cette valeur correspond évidemment à $\sqrt{a}$. Les méthodes itératives sont des méthodes par cheminement : à partir d'un point arbitraire x_0 du domaine dans lequel se trouve la solution x_s, elles permettent de former une suite d'itérés x_1, x_2, $\cdots$, avec l'espoir que cette suite converge vers la solution cherchée. Lorsque c'est le cas, on a $\lim_{n \to \infty} x_n = x_s$. En utilisant la méthode de Newton-Raphson pour trouver "le zéro" d'une fonction f, la suite des itérés x_i est définie par récurrence par :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

   (f doit être continue à dérivée continue dans le voisinage de la solution.) Pour $f(x) = x^2 - a$, on a donc :

$$x_{k+1} = \frac{\left(\frac{a}{x_k} + x_k\right)}{2}$$

   (a) Définir une fonction `next` qui calcule x_{k+1} à partir de x_k et a.

(b) En utilisant la fonction `until` définir une fonction `racine` qui calcule une approximation r de $\sqrt{a}$ telle que $|\sqrt{a} - r| < \varepsilon$ (où ε est passé en argument).

Exercice 14 Evaluez les expressions ci-dessous.

```
type t = {x : int; y : int};;
let rec1 = {x = 3; y = 1};;
let rec2 = {y = 1; x = 3};;
rec1 = rec2;;
type t' = {xxx : int; yyy : t};;
let rec3 = {xxx = 1; yyy = {x = 4; y = 5}};;
rec3.yyy.x;;
```

Exercice 15

1. Un point p dans le plan est la donnée d'une abscisse x_p et d'une ordonnée y_p. Proposer un type de données pour les points.

2. Un segment dans le plan est la donnée des deux points qui constituent ses extrémités. Proposer un type de données pour les segments.

3. Définir une fonction `make_segment` qui étant donnés deux points retourne le segment défini par ces deux points.

4. Définir une fonction qui calcule la longueur d'un segment passé en argument (on rappelle que la longueur entre deux points p_1 et p_2 est donnée par $\texttt{sqrt}((y_1 - y_2)^2 + (x_1 - x_2)^2)$).

Exercice 16 Le type `bool` peut être défini par :

```
type bool = true | false
```

Définir une fonction de type `bool * bool -> bool` qui réalise la table de vérité d'une implication logique.

Exercice 17 Evaluez les expressions ci-dessous.

```
type C = C1 | C2 | C3 of int * float * string list;;
let x = C3(1, 1.2, ["aaa"; "bbb"]);;
let f c = match c with
| C1 -> 1
| C3(x,y,z) -> x
| C2 -> 0;;
f x;;
```

Exercice 18 On souhaite définir les opérations de manipulation d'entiers signés. On définit le type `entier_signe` par :

```
type entier_signe = Pos of int | Neg of int ;;
```

1. Ecrire une fonction `creer_signe`, qui prend un argument `n` de type `int` et rend la valeur `Pos(n)` si `(n >= 0)` et `Neg(-n)` dans le cas contraire.

2. On souhaite que les constructeurs `Pos` et `Neg` ne soient appliqués qu'à des entiers positifs ou nuls. Ecrire une fonction `correct` qui vérifie cette condition. Une valeur `v`, telle que `correct(v) = true`, est dite correcte.

3. Ecrire les fonctions `plus_signe`, `moins_signe` et `mult_signe` qui prennent un couple d'arguments de type `entier_signe` et, si ces arguments sont corrects, effectuent les opérations indiquées dans leur nom. (Indication : la construction `let rec . and .` permet de définir des fonctions mutuellement récursives)

Exercice 19 On entre successivement les expressions suivantes. Indiquer le résultat de l'évaluation de chacune de ces expressions.

1.
```
let l = [0; 1; 2];;
hd l;;
tl l;;
hd (tl l);;
hd (tl (tl l));;
hd (tl (tl (tl l)));;
```

2.
```
let l = 3::[];;
l=[3];;
1::l;;
[]@l@[4];;
l::[1];;
[l] :: [[1]];;
l@["34"; "2"];;
```

Exercice 20 Définir et donner le type des fonctions suivantes :

1. `head_liste`, `tail_liste` et `nth_liste` qui retournent respectivement le premier, le dernier et le n-ième élément d'une liste.

2. `rang_pair_liste` qui extrait la liste des éléments de rang pair d'une liste.

3. `for_all` qui détermine si un prédicat est vérifié par tous les éléments d'une liste.

4. `merge` : fusionne deux listes d'entiers triées

Exercice 21 Que fait la fonction suivante :

```
let rec u = match u with
          [] -> (function x -> x)
        | x::xs -> (function y -> x (u xs y))
```

Exercice 22 (Fonctions classiques sur les listes)

1. Ecrire une fonction `length` qui prend une liste en argument et qui rend la longueur de cette liste.

2. Ecrire une fonction `append` qui prend deux listes en arguments et qui retourne la concaténation des deux listes.

3. Ecrire une fonction `rev` qui prend une liste ℓ en argument et qui rend la liste "miroir" de ℓ.

4. Ecrire une fonction `somme` qui prend une liste d'entiers en argument et qui rend la somme des éléments de cette liste.

5. Ecrire une fonction `flat` qui prend une liste de listes en argument et qui rend la liste "applatie" correspondante.

Exercice 23 Ecrire une fonction `intervalle` qui retourne la liste des entiers consécutifs compris entre deux entiers n et m passés en arguments. Utiliser cette fonction pour définir une fonction qui calcule $n!$.

Exercice 24 On veut créer une fonction engendrant, pour un entier $k > 0$, la liste L_k définie comme suit :

$$\begin{array}{ll} L_1 & [1] \\ L_2 & [1;1] \\ L_3 & [2;1] \\ L_4 & [1;2;1;1] \\ L_5 & [1;1;1;2;2;1] \\ L_6 & [3;1;2;2;1;1] \\ \ldots & \end{array}$$

En fait, on obtient la liste L_k en "lisant" le contenu de la liste L_{k-1}. Par exemple, L_5 s'obtient en lisant la liste L_4 : 1 "1", 1 "2" et 2 "1".

1. Ecrire une fonction `calc_prefixe` qui prend en argument une liste d'entiers L et qui renvoie 0 si L est vide, ou le nombre de chiffres identiques au début de la liste L. Par exemple `calc_prefixe [1;1;1;2;2;1]` renvoie la valeur 3 car la liste commence par trois "1".

2. Ecrire une fonction `genere_liste` qui construit L_k à partir de L_{k-1}. On supposera, pour simplifier, que tous les entiers de la liste sont compris entre 0 et 9.

3. Ecrire une fonction qui étant donné un entier k construit la liste L_k,