Programmation Fonctionnelle
TP n°1 : Retrouver ces petits

Le but de ce TP est de vous initier à OCaml et de vous faire expérimenter les différentes façons de faire en OCaml ce que vous savez déjà faire en C ou Javascool. Bien entendu, on se restreindra au comportement « pure » (sans effet de bord, ni modification de mémoire globale ou externe, pas d'affichage, rien que le calcul d'une sortie en fonction d'une entrée ou de rien). Le but est également d'explorer ce que vous pouvez faire en fonctionnelle et pas en C / Javascool, ce qui apparaitra : utiliser la currification ou le passage de fonction en paramètre pour réduire la quantité de code à écrire.

Exercice 0: Expressions et types

Soit la liste d'expression suivantes :

1. `10`	11. `(*)`	21. `(=) "me"`
2. `10.0`	12. `(*.)`	22. `fun x -> "me" = x`
3. `10.5`	13. `(+)`	23. `fun x -> "hello" ^ x`
4. `"hello"`	14. `(+.)`	24. `fun x -> x ^ "hello"`
5. `int_of_float`	15. `(/)`	25. `(^) "hello"`
6. `float_of_int`	16. `true`	26. `let a = 5 in a + b`
7. `string_of_int`	17. `false`	27. `(fun a -> a + b) 5`
8. `string_of_float`	18. `(&&)`	28. `(fun a -> fun a -> a + a) 10 20`
9. `(^)`	19. `(=)`	29. `let a = 10 in let a = 20 in a + a`
10. `sqrt`	20. `(=) 5`	30. `let a = 10 in let b = 20 in b + b`

Commencez par donner le type de toutes les expressions. Ensuite, pour chaque expression fonctionnelle, donc la liste des autres expressions sur lesquelles elle peut être appliquer ainsi que le type du résultat de l'application. Par exemple, l'expression `10` est de type `int` ; l'expression `(+)` est de type `int -> int -> int`, et l'application `(+) 10` a pour type `int -> int`.

Exercice 1: Traduction de C / Javascool

Écrire en OCaml les fonctions suivantes en respectant le plus possible leur structure:

```
int distance_squared (int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    return dx * dx + dy * dy;
}


// sqrtf (SQuare RooT Float) calcule la racine carrée d'un float
// en C, il faut faire #include <math.h> pour l'avoir, et ajouter -lm
à la compilation
// en OCAML, il est directement disponible sous le nom sqrt
```

```
float distance (int x1, int y1, int x2, int y2) {
    return sqrtf (distance_squared (x1, y1, x2, y2));
}

// Pour le suivant, vous pouvez faire quelque chose de malin
(currification ?)

float vector_norm (int x, int y) {
    return distance (0, 0, x, y);
}

// Faire: float distance_squared_f (float x1, float y1, float x2,
float y2)
// Faire: float distance_f (float x1, float y1, float x2, float y2)
// Faire: float vector_norm_f (float x, float y)

float constant1 = 30;

int constant2 () {
    return 3;
}

float constant3 () {
    int a = 3;
    int b = 5;
    return vector_norm (a, b);
}

int test (int a, int b, int c) {
    if (a >= 0) {
        return a * b;
    } else {
        return -a * c;
    }
}

// Dans le suivant il faut modifier un peu la structure,
// et il y a plusieurs façon de le faire.

int test2 (int a, int b, int c) {
    int r;
```

```
    if (a >= 0) {
        r = b * b;
    } else {
        r = c * c;
    }
    return a * r;
}

float power (int a, int b) {
    if (b == 0) {
        return 1;
    } else if (b > 0) {
        return power (a, b - 1) * b;
    } else {
        // ici on a forcement b < 0
        return power (a, b + 1) / b;
    }
}

// asprintf, c'est un peu comme printf, mais au lieu d'afficher la chaîne de caractère,
// il retourne la chaîne construite dans le premier argument.

char* optim (int a, int b) {
    char *p;
    asprintf(&p, "Anciennes valeurs: %d %d", a, b);
    a = a + b;
    b = a - b;
    a = a - b;
    char *q;
    asprintf(&q, "%s\nNouvelles valeurs: %d %d", p, a, b);
    free(p);
    return q;
}

// En Javascool

String optim (int a, int b) {
    String p = "Anciennes valeurs: " + a + " " + b;
    a = a + b;
```

```
    b = a - b;
    a = a - b;
    String q = p + "\nNouvelles valeurs: " + a + " " + b;
    free(p);
    return q;
}
```

Exercice 2: Factorisation de code

Exemple de factorisation en C / Javascool:

Avant:

```
int testA (int a, int b) {
    if (a < 0) a = -a;
    if (b > 0) b = b * b;
    return a * 4 + b;
}

int testB (int a) {
    if (a < 0) a = -a;
    if (b > 0) b = b * b;
    return a * 2 - b;
}
```

Après:

```
int test (int k, int m, int a) {
    if (a < 0) a = -a;
    if (b > 0) b = b * b;
    return a * k + m * b;
}

int testA (int a) { return test (4, +1, a); }
int testB (int a) { return test (2, -1, a); }
```

== Faire l'exemple précédent en OCAML:

== Il y a un truc malin à écrire pour le testA et testB factoriser

Factoriser les 4 fonctions suivantes en OCAML:

```
char* message_haut (int x, int y) {
    return sprintf ("Déplacement de %d %d vers %d %d", x, y, x, y -
1);
```

```
}

char* message_bas (int x, int y) {
    return sprintf ("Déplacement de %d %d vers %d %d", x, y, x, y +
1);
}

char* message_gauche (int x, int y) {
    return sprintf ("Déplacement de %d %d vers %d %d", x, y, x - 1,
y);
}

char* message_droite (int x, int y) {
    return sprintf ("Déplacement de %d %d vers %d %d", x, y, x + 1,
y);
}

== Factoriser les 4 fonctions suivantes en OCAML:
== Ici, ce qui varie entre les 4 codes n'est pas un nombre, mais du
code!

float calculA (float dx, float x) {
    float a = 3 * (x + dx);
    float b = 3 * x;
    return a - b;
}

float calculB (float dx, float x) {
    float a = (x + dx) * (x + dx);
    float b = x * x;
    return a - b;
}

float calculC (float dx, float x) {
    float a = sqrtf (x + dx);
    float b = sqrtf x;
    return a - b;
}

float calculD (float dx, float x) {
    float a = (x + dx) * (x + dx) - (x + dx);
```

```
    float b = x * x - x;
    return a - b;
}
```