TP 1 – Premières fonctions

OCaml vient avec plusieurs outils permettant d'évaluer des expressions intéractivement ou encore de compiler OCaml vers du code natif. Le plus souvent, pour écrire tester et évaluer rapidement des expressions, il est d'usage de passer par un REPL (Read-Eval-Print Loop) qui donne un retour intéractifs aux expressions entrées. L'accès au REPL d'OCaml est donné en utilisant le programme `ocaml` et est appelé top-level.

Même si l'utilisation du top-level est pratique pour tester et avoir un retour rapide, le top-level d'OCaml n'est pas spécialement pratique à utiliser. Nous allons vois pourquoi dans le prochain exercice.

Exercice 1. Prise en main de l'interpréteur

Le but de cet exercice est de vous familiariser avec l'interpréteur `ocaml` et les types de base. Vous allez pouvoir entrer des expressions que le "top-level" évaluera.

1. Lancer le top-level (programme `ocaml`) et évaluer l'addition $1 + 2$. Pouvez-vous utiliser les flèches directionnelles ?

2. Lancer le top-level amélioré avec `utop`. Vérifier que vous avez bien accès aux flèches directionnelles.

Remarque : De manière générale, si le programme `utop` n'est pas installé, on préfèrera écrire les fonctions et expressions dans un fichier et l'inclure pour le tester dans le top-level en utilisant : `#use "mon_fichier.ml";;`

3. Déterminer le type des expressions suivantes (lorsque c'est possible) et vérifier avec le top-level.

```
2   2.0   2,0   2;0   a   'a'   "a"   true   ()   []   [1]   [1, true]   [1; true]
```

4. Lorsque c'est possible, donner des exemples d'expressions ayant les types suivants :

a. `int * float`
b. `(int, float)`
c. `string list`
d. `bool list * string`
e. `'a * int`

5. Utiliser le top-level pour tester si les opérateurs booléens `and` et `or` sont séquentiels.

Exercice 2. Conditions

Donner les expressions pour effectuer les actions suivantes :

1. Calculer le maximum de `a` et `b`.

2. Calculer le minimum de `a`, `b` et `c`.

3. Tester si l'entier `a` est pair. Si oui, afficher sa valeur, sinon, afficher `"odd"`.

4. Que pensez-vous du code suivant ? Est-il correct ?

```
if a<10  then  let b="small" else  let b="large";;
```

5. Modifier le code précédent pour qu'il ait le comportement attendu.

Exercice 3. Premières fonctions

1. Écrire une fonction `average` qui calcule la moyenne de trois entiers. Cette fonction peut-elle être utilisée avec des flottants ?

2. Écrire une fonction `implies` qui prend deux expressions booléennes `a` et `b` et renvoie vrai si `a` implique `b` au sens logique.
3. À l'aide du top-level, donner le type des fonctions `fst` et `snd`.
4. Déduire leur utilisation.
5. Écrire une fonction `inv` qui prend un couple et renvoie le couple inversé. Faire deux versions, une utilisant `fst` et `snd` et l'autre en utilisant du pattern matching (si abordé en cours).

Exercice 4. La suite de Fibonacci
La suite de Fibonaci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence généralement par les termes 0 et 1 (parfois 1 et 1) et ses premiers termes sont : $0, 1, 1, 2, 3, 5, 8, 14, 21, ...$

```
# fib;;
- : int -> int = <fun>
# fib 0, fib 1, fib 2, fib 3, fib 4, fib 5;;
- : int * int * int * int * int * int = (0, 1, 1, 2, 3, 5)
```

1. Écrire la fonction `fib` à un argument entier `n` qui calcule $Fib(n)$.

Exercice 5. Plus grand diviseur commun
On a les relations suivantes pour le cacul du pgcd de deux nombres :

$$pgcd(m,n) = \begin{cases} n & si\ m = 0 \\ pgcd(n,m) & si\ m > n \\ pgcd(n\ mod\ m, m) & sinon \end{cases}$$

```
# pgcd;;
val pgcd : int -> int -> int = <fun>
# pgcd 1 1;;
- : int = 1
# pgcd 20 30;;
- : int = 10
# pgcd 77 34;;
- : int = 1
```

1. Écrire la fonction `pgcd` qui étant donné deux entiers retourne leur plus grand diviseur commun.

Exercice 6. Fonctions mutuellement récursives
Le but de cet exercice est d'écrire les fonctions recursives `pair` et `impair` sans utiliser le modulo ou la division et avec la seule information que 0 est pair (`n` est impair si `n - 1` est pair, `n` est pair si `n - 1` est impair, 0 est pair, 0 n'est pas impair). La fonction `pair` (resp. `impair`) retourne `true` si son argument entier est pair (res. impair).
indication : pour écrire ces fonctions, il est nécessaire de définir en même temps `pair` et `impair` en utilisant la notation `let rec pair n = ... and impair n = ...;;`

```
# pair;;
- : int -> bool = <fun>
# impair;;
- : int -> bool = <fun>
# pair 0, pair 1, pair 2, pair 3;;
- : bool * bool * bool * bool  = (true, false, true, false)
# impair 0, impair 1, impair 2, impair 3;;
- : bool * bool * bool * bool = (false, true, false, true)
```

Exercice 7. Récursivité terminale 1/2
Une fonction à récursivité terminale (dite tail-recursive en anglais) est une fonction où l'appel récursif est la dernière instruction à être évaluée. Cette instruction consiste en un simple appel à la fonction, et jamais à un calcul ou une composition avec le retour de celle-ci. La récursivité terminale économise de l'espace mémoire car aucun état (sauf l'adresse de la fonction appelante) n'a besoin d'être sauvé sur la pile d'exécution. Cela signifie également que le programmeur n'a pas à craindre l'épuisement de l'espace de pile ou du tas pour les récursions très profondes.
1. Écrire une fonction à récursivité terminale `fact` à un argument entier `n` qui calcule $n!$. *indication :* pour écrire ce type de fonction, on utilise des fonctions "internes" avec plus d'arguments que la fonction de plus "haut niveau".

```
# fact;;
- : int -> int = <fun>
# fact 0, fact 1, fact 2, fact 3, fact 4;;
- : int * int * int * int * int = (1, 1, 2, 6, 24)
```

Exercice 8. Récursivité terminale 2/2
Écrire une version à récursivité terminale de la fonction `fib` à un argument entier `n` qui calcule $Fib(n)$.

Exercice 9. Exponentielle
1. Écrire la fonction `exp x n` qui calcule x^n
2. Écrire la fonction `exp' x n` qui calcule x^n en utilisant une récursivité terminale.
3. Réécrire la fonction `exp x n` pour qu'elle calcule x^n ainsi que le nombre d'appels récursif effectués pour obtenir ce résultat.

Exercice 10. Sommes doubles
1. Écrire une fonction `sum1 n m` qui calcule la somme suivante en fonction de n et m :

$$\sum_{a=n}^{m}\sum_{b=a}^{m} b$$