TP OCaml basique

Dans ce TP d'introduction, toutes les variables sont persistantes, aucune liste n'intervient et la récursion n'est pas nécessairement terminale.

Exercice 1. écrire une fonction `sigma : (int -> int) -> int -> int` qui prend en paramètre une fonction f et retourne la somme

$$\Sigma_{i=0}^{n} f(i).$$

```
# sigma (fun x -> x*x) 3;;
- : int = 14
```

Exercice 2.

1. Écrire la fonction `opp : int -> int` qui renvoie l'opposé d'un entier. Par exemple `opp 6` renvoie -6.
2. Écrire une fonction `incr : int -> int -> int` telle que `incr a x` renvoie $x+1$ si $a > 0$, $x-1$ si $a < 0$ et x sinon.

```
# incr 3 6, incr 0 6, incr (opp 8) 6;;
- : int * int * int = (7, 6, 5)
```

3. Écrire une fonction `somme : int -> int -> int` qui retourne la somme de deux nombres. Aucun symbole `+` ou `-` ne doit être utilisé.
   La fonction doit avoir un coût linéaire.

Exercice 3. Écrire une fonction `prod: int -> int -> int` qui renvoie le produit de deux entiers positifs. Les seules opérateurs arithmétiques autorisés sont `+`, `mod`, `lsr` (décalage à droite), `lsl` (décalage à gauche).

La fonction doit avoir un coût logarithmique en nombre d'opérations arithmétiques (divisions par 2 ; produits par 2 ; additions ; décalages).

Exercice 4. On veut afficher des triangles :

1. Écrire la fonction `triangle1 (n:int) (c:char):unit` qui réalise l'affichage suivant :

```
# triangle1 5 '*';;
*****
****
***
**
*
- : unit = ()
```

2. Écrire la fonction `triangle2 (n:int) (c:char):unit` qui réalise l'affichage suivant :

```
# triangle2 5 '*';;
*
**
***
****
*****
- : unit = ()
```

3. Écrire la fonction `triangle3 (n:int) (c:char):unit` qui réalise l'affichage suivant :

```
# triangle3 5 '*';;
    *
   ***
  *****
 *******
*********
- : unit = ()
```

4. Écrire la fonction `losange (n:int) (c:char):unit` qui réalise l'affichage suivant :

```
# losange 5 '*';;
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
- : unit = ()
```

Exercice 5. On rappelle que pour deux nombres entiers a, b, on a $\text{pgcd}(a, b) = \text{pgcd}(b, r)$ où r est le reste de la division euclidienne de a par b. De plus, $\text{pgcd}(a, 0) = a$.

Ces deux observations sont au cœur de l'*algorithme d'Euclide* pour la recherche du pgcd.

1. Écrire une fonction OCaml `pgcd : int -> int -> int` qui calcule par l'algorithme d'Euclide le pgcd de deux nombres.

```
# pgcd 21 15;;
- : int = 3
# pgcd (3*3*2*5*7*7*7*11) (3*3*3*3*5*7);;
- : int = 315
```

2. On admet que si a, b sont deux entiers positifs tels que $a \geq b > 0$, alors $a\%b \leq \lfloor \frac{a}{2} \rfloor$ en notant % l'opérateur de calcul du reste.

(a) Dans le cours, on considère les opérations arithmétiques (addition, division ...) comme des opérations élémentaires : c'est sans doute vrai si la taille des entiers est bornée, mais pas dans l'absolu.

Dans cet exercice, nous évaluons la complexité du calcul du pgcd en fonction du nombre de bits nécessaires à l'écriture des 2 nombres.

On suppose a, b écrits sur n bits et $a > b$ (donc b est en fait écrit sur moins de n bits).

Majorer le nombre total d'appels à `pgcd` pour l'appel initial `pgcd a b` en fonction de n.

(b) Majorer le nombre total d'appels à `pgcd` pour l'appel initial `pgcd a b` en fonction de a.

(c) On admet que la recherche du reste pour deux nombre écrits sur au plus n bits est majorée par un $O(n^2)$ opérations élémentaires ; que la comparaison à 0 est en $O(n)$ (il y a n bits à comparer) et que les écritures (comme celle à l'adresse de retour) sont majorées par un $O(n)$ (il y a n bits à écrire).

Donner, en fonction de a, une majoration pas trop grossière de la complexité au pire de l'appel `pgcd a b` (on ne prend en compte que les calculs de restes, les écritures et les compraisons à 0).