Premier contact avec OCaml

1 Des boucles sans boucles

On peut afficher une étoile dans la sortie standard d'OCaml en écrivant « `print_string "*"` », et effectuer un retour à la ligne avec « `print_newline ()` ».

1. Proposer une fonction `n_etoiles` récursive de signature `int -> unit` prenant en argument un entier n et affichant une suite de n étoiles suivi d'un retour à la ligne. Faute d'outils pour écrire des boucles pour le moment, il faudra ici se tourner vers une fonction récursive ! Elle devra se comporter de la sorte :

```
# n_etoiles 7;;
*******
- : unit = ()
```

2. Proposer deux fonctions récursives `triangle_bas` et `triangle_haut`, toutes deux de signature `int -> unit`, prenant en argument un entier positif n, et traçant des « triangles » constitués d'étoiles, de hauteur et largeur n caractères, tels qu'illustrés ci-dessous :

```
# triangle_bas 4;;
*
**
***
****
- : unit = ()

# triangle_haut 4;;
****
***
**
*
- : unit = ()
```

2 Séquences particulières

La suite des nombres de Fibonacci est une suite $(f_n)_{n\in\mathbb{N}}$ d'entiers positifs définie par :

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ f_{n+2} = f_n + f_{n+1} \quad \text{pour tout } n \geqslant 0 \end{cases}$$

Les premiers termes de cette suite sont $f_0 = 0, f_1 = 1, f_2 = 1, f_3 = 2, f_4 = 3, f_5 = 5$...

3. Proposer une fonction `fibonacci` de signature `int -> int` prenant en argument un entier positif n et retournant f_n. On ne se préoccupera pas, exceptionnellement, de proposer des fonctions particulièrement efficace, le but de cette séance étant de se familiariser avec l'écriture et la manipulation de fonctions OCaml. Nous verrons plus tard comment il est possible d'obtenir des solutions de meilleure complexité temporelle. On verifiera que les premiers termes sont les bons.

La suite des nombres de Stern est une suite $(s_n)_{n\in\mathbb{N}}$ d'entiers positifs définie par :

$$\begin{cases} s_0 = 0 \\ s_1 = 1 \\ s_{2n} = s_n \quad \text{pour tout } n \geqslant 1 \\ s_{2n+1} = s_n + s_{n+1} \quad \text{pour tout } n \geqslant 1 \end{cases}$$

Les premiers termes de cette suite sont $s_0 = 0, s_1 = 1, s_2 = 1, s_3 = 2, s_4 = 1, s_5 = 3$...

4. Proposer une fonction `stern` de signature `int -> int` prenant en argument un entier positif n et retournant s_n.

La suite de Stern présente un intérêt particulier : elle permet de construire la suite $w_n = \frac{s_n}{s_{n+1}}$, dite suite de Calkin-Wilf, qui présente la particularité d'énumérer tous les rationnels positifs ou nuls, chacun n'apparaissant qu'une seule et unique fois.

5. Proposer une fonction `calkin_wilf` de signature `int -> (int*int)` prenant en argument un entier positif n et retournant un couple (p_n, q_n) correspondant à $w_n = p_n/q_n$.

6. En déduire une fonction de signature `int -> int` (comme précédemment, exceptionnellement, on ne se préoccupe pas de complexité temporelle) prenant en argument un entier positif n et retournant le nombre de rationnels parmi $w_0, \ldots, w_{n-1}$ qui se trouvent dans l'intervalle $[0, 1]$. Peut-on conjecturer quelque chose du comportement asymptotique de cette fonction quand n tend vers l'infini ?

La suite des nombres de Catalan [1] est une suite $(c_n)_{n\in\mathbb{N}}$ d'entiers positifs définie par :

$$\begin{cases} c_0 = 1 \\ c_{n+1} = \displaystyle\sum_{k=0}^{n} c_k c_{n-k} \end{cases}$$

Les premiers termes de cette suite sont $c_0 = 1, c_1 = 1, c_2 = 2, c_3 = 5, c_4 = 14, c_5 = 42$...

1. En hommage au mathématicien belge Eugène Catalan qui l'a étudiée, même si cette suite était connue des chinois bien plus tôt, et était apparue dans les écrits de Léonard Euler.

7. Proposer une fonction `catalan` de signature `int -> int` prenant en argument un entier positif n et retournant c_n.

3 Nombres de Lychrel

3.1 Introduction rapide aux chaînes de caractères

Les chaînes de caractères en OCaml sont des objets de type « `string` », représentées entre guillemets doubles.

```
# let ma_chaine = "Hello World!";;
val ma_chaine : string = "Hello World!"
```

On peut accéder à la longueur (nombre de caractères) d'une chaîne en utilisant la fonction `String.length` :

```
# String.length ma_chaine;;
- : int = 12
```

On peut obtenir un caractère de la chaîne grâce à son index en faisant suivre le nom désignant la chaîne d'un point, suivi de l'index entre crochets, de la sorte :

```
# ma_chaine.[6];;
- : char = 'W'
```

On remarquera qu'un caractère a son propre type, `char` : il ne s'agit pas d'une chaîne de caractères de longueur 1, comme c'est par exemple le cas en Python. On peut comparer des caractères entre eux, des chaînes entre elles, mais il est impossible de comparer un caractère et une chaîne.

On peut également obtenir un morceau de la chaîne en précisant un index de début et une *longueur* souhaitée au moyen de la fonction `String.sub`, en écrivant par exemple :

```
# String.sub ma_chaine 6 5;;
- : string = "World"
```

Enfin, on peut obtenir la chaîne résultant de la concaténation de deux chaînes avec l'opérateur « `^` » :

```
# "Again, " ^ ma_chaine;;
- : string = "Again, Hello World!"
```

Cette fois encore, attention : on ne peut concaténer que des chaînes de caractères, mais il est impossible de concaténer une chaîne de caractères et un caractère, ou bien enore deux caractères.

3.2 Palindromes

Un palindrome est une chaîne de caractères qui se lit indifféremment de gauche à droite et de droite à gauche, telle que « `"radar"` »

8. Écrire une fonction (récursive) `est_palindrome` qui prend en argument une chaîne de caractères et retourne un booléen indiquant si la chaîne est un palindrôme.

9. Écrire une fonction (récursive) `retourne` qui prend en argument une chaîne de caractères et retourne une chaîne de caractère correspondant au renversement de l'argument (la chaîne « `"Hello"` » doit donner comme résultat « `"olleH"` »).

Note : on pourra se servir, pour la fonction `retourne` d'autant de concaténations que nécessaires, même si l'on verra ultérieurement que cette approche n'est pas très efficace. Encore une fois, la recherche de complexité temporelle raisonnable n'est pas le but de cette séance.

3.3 Nombres de Lychrel

On considère la suite d'entiers défini par la relation de récurrence $u_{n+1} = u_n + f(u_n)$ où la fonction f est la fonction qui « retourne » les chiffres de son paramètre (par exemple, $f(1023) = 3201$).

Il existe deux fonctions permettant de convertir des entiers en chaînes de caractères et inversement, appelées respectivement `string_of_int` et `int_of_string`.

10. En se servant des fonctions de la section précédente et des outils de conversion, écrire une fonction `suivant` qui prend en argument un entier correspondant à u_n et retourne l'entier u_{n+1}.

11. Écrire une fonction `nb_iter` prenant en argument un entier correspondant à u_0 et indiquant après combien d'itérations on obtient, pour la première fois, un nombre u_n palindrômique.

12. Que donne la fonction pour les valeurs initiales 19, 59, 89 ?

13. Quel problème survient si le nombre d'itérations est trop grand ?

3.4 Au-delà des entiers

Une telle situation se produit par exemple pour `u0=10911`. Afin de résoudre ce problème, on souhaite pouvoir manipuler des entiers de taille arbitraire. On les représentera par des chaînes de caractères (`"1234"` représentera l'entier 1234). Mais pour ce faire, il nous faut redéfinir une addition !

On souhaite écrire une fonction `somme` prenant en argument deux chaînes de caractères contenant deux entiers positifs ou nuls, et retournant une chaîne contenant leur somme, en utilisant la méthode vue en primaire pour effectuer l'addition.

Pour faciliter l'écriture de cette fonction, on crée tout d'abord une fonction `string_of_char` de signture `char -> string` convertissant un caractère (`char`) en une chaîne de caractères de longueur 1 contenant ce seul caractère :

```
let string_of_char c = String.make 1 c;;
```

Et une fonction `digit_of_char` de signature `char -> int` prenant en argument un caractère correspondant à un chiffre, et retournant un entier égal à ce même chiffre :

```
let digit_of_char c = int_of_string (string_of_char c);;
```

Il est à présent temps d'écrire notre fonction somme :

```
let somme ch1 ch2 =
    let rec aux = function
          |
          |
          |
          | (ch1, ch2, car) ->
             let a = digit_of_char ch1.[String.length ch1 - 1]
             and b = digit_of_char ch2.[String.length ch2 - 1]
                 in let s = a + b + car
                     in (aux ((String.sub ch1 0 (String.length ch1 - 1)),
                              (String.sub ch2 0 (String.length ch2 - 1)),
                              (         )) ) ^ string_of_int (         )
    in aux (ch1, ch2, 0);;
```

14. Comprendre le fonctionnement de la fonction (qui repose sur la méthode étudiée en primaire !) et compléter les blancs. Le nombre de cas manquants dans le filtrage n'est qu'indicatif, vous pouvez en ajouter ou en retirer.

On pourra utiliser cette fonction pour déterminer le nombre d'itérations nécessaires pour obtenir un palindrôme à partir de 10911. Cependant, certains nombres semblent ne jamais conduire à un palindrôme, le plus petit étant le nombre 196. Après plus de 700 millions d'itérations, le nombre, comprenant 300 millions de chiffres, n'est toujours pas un palindrôme. On appelle ces nombres *nombres de Lychrel*. On connaît de nombreux candidats, mais la démonstration que l'on n'arrive jamais à un palindrôme n'a pas encore pu être trouvée.

4 Retour sur les tracés

On souhaite tracer quelques figures intéressantes, à l'aide d'étoiles « * » et d'espaces, dans le terminal.

15. Proposer une fonction appelée `trace`, dont la signature OCaml serait `int -> (int -> int -> bool) -> unit` et prenant donc deux arguments, un entier `n` et une fonction `f`, cette dernière prenant successivement deux entiers et retournant un booléen. La fonction `trace` devra tracer, au moyen de caractères dans le terminal, une figure de hauteur `n` et de largeur `n`, telle que le caractère à la ligne `i` ($0 \leqslant i < n$) et à la colonne `j` ($0 \leqslant j < n$) de cette figure est une étoile si `f i j` retourne `true` et une espace sinon.

16. Quels résultats donnent les deux appels suivants ?

```
trace 10 (fun i j -> true)|

trace 10 (fun i j -> (i + j) mod 4 = 2)|
```

17. Proposer des implémentations de `triangle_bas` et `triangle_haut` à partir de `trace`.

18. Proposer une fonction `foo` de signature `int -> int -> bool` prenant en argument deux entiers `i` et `j` et retournant `true` si et seulement si tous les bits à 1, dans la représentation binaire de `j` correspondent à un bit à `1` à la même place dans la représentation de `i`.

19. Proposer une fonction `pow` de signature `int -> int -> int` prenant en argument un entier `n` et un entier `p` et retournant n^p.

20. Définir et tester la fonction suivante (par exemple pour $n = 4$) :

```
let trace_foo n = trace (pow 2 n) foo
```

21. Proposer une fonction `bar` de signature `int -> int -> bool` prenant en argument deux entiers `i` et `j` et retournant `true` si et seulement si le chiffre 1 n'apparait ni dans l'écriture ternaire de `i`, ni dans celle de `j`.

22. Implémenter et tester la fonction suivante (par exemple pour $n = 3$) :

```
let trace_bar n = trace (pow 3 n) bar
```

23. Trouver une fonction de signature `int -> int -> bool` permettant d'obtenir un tapis de Sierpinski.