TP 2 : Récursion terminale et structures de données

L'objectif de ce TP est de consolider les acquis de la dernière séance en implémentant plusieurs algorithmes simples en OCaml. Une notion très importante en programmation fonctionnelle est également introduite : la récursion terminale.

Algorithmes sur les listes

Comme beaucoup d'autres langages, OCaml est fourni avec un ensemble de bibliothèques (modules) donnant accès à des fonctionnalités de base telles que l'appel de primitives système (*Sys*), l'affichage sur console (*Printf*) ou la manipulation de structures de données (*Array*, *List*, ...). Les modules disponibles sont répertoriés dans la documentation du langage : https://ocaml.org/api/index.html.

Question 1. Grâce à la documentation du module *List*, expliquez les paramètres d'entrée et de sortie ainsi que l'utilité des fonctions suivantes : *map*; *fold_left*; *fold_right*; *filter*.

Question 2. Implémentez en OCaml les 4 fonctions précédentes.

Une fonction définie en **récursion terminale** permet de simuler une exécution itérative à partir d'appels récursifs. En d'autres termes, il s'agit d'une manière récursive de déclarer une boucle. Ce type de fonction a notamment l'avantage de ne pas surcharger la pile d'exécution quand la récursion est très profonde, ce qui permet l'utilisation d'algorithmes récursifs même quand le nombre d'étapes intermédiaires est très important (par exemple sur de grandes listes).

La fonction *rev* ci-dessous inverse l'ordre des éléments d'une liste sans récursion terminale. Pour inverser l'ordre d'une liste à une étape donnée, il faut d'abord obtenir une version inversée du reste de la liste et ensuite placer le premier élément à sa fin. À chaque appel récursif, la pile d'exécution conserve le premier élément en attendant que le reste soit inversé pour pouvoir continuer. Si la taille de la liste à inverser est 100 000 000, alors la pile devra conserver en mémoire 99 999 999 "premiers éléments" en attendant d'arriver à un cas trivial, ce qui la fera en général surcharger bien avant d'en arriver là et provoquera l'échec de l'exécution. Pour éviter ce phénomène, il faudrait faire en sorte de ne pas avoir besoin de la pile, par exemple en supprimant les opérations qui s'exécutent après l'appel récursif (ici la fusion) et nécessitent donc le besoin de garder en mémoire l'état intermédiaire avant l'appel.

```
let rec rev liste =
    match liste with
    | [] -> []
    | [_] -> liste
    | elt1::reste -> (rev reste) @ [elt1] ;;
```

Ainsi, pour qu'une fonction soit en récursion terminale, **l'appel récursif doit toujours être le dernier élément à être exécuté à chaque étape de la récursion.** Une technique très souvent utilisée pour transformer une fonction de sorte à ce qu'elle respecte la récursion terminale consiste à lui ajouter un paramètre en entrée, l'accumulateur (souvent appelé *acc*), qui va recevoir à chaque appel récursif le résultat intermédiaire calculé. Au fur et à mesure, l'accumulateur est transformé pour former le résultat final. Il est finalement retourné lors de l'appel correspondant au cas trivial.

Question 3. La fonction *rev_rt* suivante est-elle en récursion terminale ? Pourquoi ?

```
let rec rev_rt liste acc =
    match liste with
    | [] -> acc
    | elt1::reste -> rev_rt reste (elt1::acc) ;;
```

Question 4. Implémentez en récursion terminale une fonction *concat* équivalente à celle proposée par le module *List* (voir documentation).

Suite de Fibonacci

La suite de Fibonacci est une fonction qui prend en entrée un entier positif n, appelé rang, et qui est définie de la manière suivante :

$$\begin{cases} Fibo(0) = 0, \\ Fibo(1) = 1, \\ Fibo(n) = Fibo(n-1) + Fibo(n-2) \end{cases}$$

Question 5. À partir de la définition mathématique de la suite, implémentez une fonction récursive *fibonacci* qui calcule la valeur correspondant au rang n. Pour le moment, pas besoin qu'elle soit en récursion terminale.

La fonction récursive naïve (*c.-à-d.* qui traduit directement l'énoncé mathématique) pour calculer la valeur de la suite de Fibonacci à un rang n donné possède une complexité exponentielle en temps. Avec un algorithme itératif, il est possible de réduire la complexité en un temps linéaire d'ordre O(n) :

```
let fibonacci n =
    if n < 2 then n
    else
        let f1 = ref 1 in let f2 = ref 0 in let fn = ref 1 in
        for i=0 to (n-2) do
            fn := !f1 + !f2 ;
            f2 := !f1 ;  f1 := !fn
        done ;
        !fn ;;
```

Question 6. Maintenant que vous êtes des pros de la récursion terminale, vous savez qu'elle permet de déclarer récursivement une boucle. Proposez une nouvelle fonction *fibonacci* qui sera cette fois-ci en récursion terminale.

Un autre moyen pour réduire la complexité en temps du calcul des valeurs de la suite de Fibonacci consiste à utiliser un tableau ou une table de hachage pour stocker les valeurs intermédiaires et ainsi ne les calculer qu'une seule fois.

En effet, comme illustré sur la figure 1, le calcul de la suite de Fibonacci au rang n avec l'algorithme naïf implique de recalculer plusieurs fois certaines valeurs antérieures. Par exemple, la suite de Fibonacci au rang 4 nécessite d'additionner la valeur de la suite au rang 3 avec celle au rang 2. Mais pour obtenir la valeur au rang 3, il y a une nouvelle fois besoin de celle au rang 2. Pour éviter de recalculer deux fois Fibonacci au rang 2, on peut enregistrer sa valeur dans une structure de données et ensuite simplement la récupérer si une étape suivante en a besoin.

Remarque : L'idée d'utiliser un tableau de valeurs intermédiaires pour éviter de ré-exécuter plusieurs fois une même fonction est à la base du concept qui sera traité dans le TP 4 : ***la programmation dynamique****.*

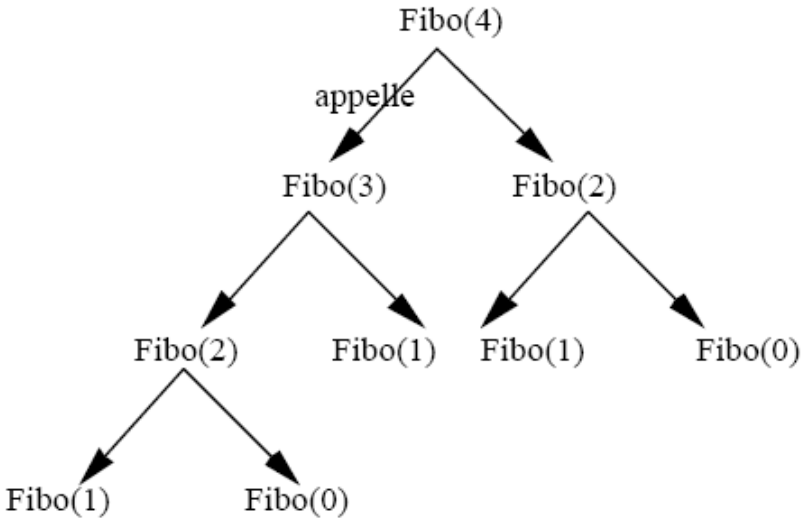

FIGURE 1. Source de la figure : http://igm.univ-mlv.fr/~dr/XPOSE2012/La%20programmation%20dynamique/fibonacci.html

Quand on utilise une table de hachage plutôt qu'un tableau, on parle en général de ***mémoïsation*** *(ce qui fait +13 points au Scrabble et permet de briller en société, de rien).*

Le code ci-dessous propose une version optimisée de la fonction *fibonacci* par programmation dynamique.

```
let fibonacci n =
    let tab_dyn = Array.make ___ (-1) in
    tab_dyn.(0) <- ___ ; tab_dyn.(1) <- ___ ; tab_dyn.(2) <- ___ ;
    let rec fibonacci_aux n =
        if tab_dyn.(n) < 0 then ___ else () ;
        tab_dyn.(n)
    in fibonacci_aux n ;;
```

Question 7. Complétez le code de la fonction *fibonacci*.

Le code suivant propose lui une version optimisée de la fonction *fibonacci* par mémoïsation.

```
let fibonacci n =
    let ht_dyn = (*1*) in
    let rec fibonacci_aux n =
        try (*2*)
        with Not_found ->
            let fn = match n with
            | 0 -> 0
            | 1 | 2 -> 1
            | _ -> fibonacci_aux (n-1) + fibonacci_aux (n-2)
            in (*3*) ; fn
    in fibonacci_aux n ;;
```

Question 8. Complétez le code de la fonction *fibonacci* en remplaçant :

- *(*1*)* par la fonction du module *Hashtbl* qui permet de créer une nouvelle table de hachage ;
- *(*2*)* par la fonction du module *Hashtbl* qui permet de récupérer dans une table de hachage la valeur enregistrée à un indice donné ;
- *(*3*)* par la fonction du module *Hashtbl* qui permet d'ajouter un élément dans une table de hachage.

N'oubliez pas d'ajouter les bons paramètres derrière bien sûr !

Enfin, une dernière optimisation possible pour le calcul de Fibonacci à un rang donné consiste à utiliser des produits de matrices. Pour être plus précis, on peut obtenir les valeurs *Fibo(n-1)*, *Fibo(n)* et *Fibo(n+1)* pour n'importe quelle valeur de n en sachant calculer la puissance n de la matrice 2x2 suivante :

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} Fibo(n+1) & Fibo(n) \\ Fibo(n) & Fibo(n-1) \end{bmatrix} \quad (1)$$

Pour ceux qui le souhaitent, la démonstration de l'égalité se fait assez facilement par récurrence ... :)

Le code suivant déclare un nouveau type *matrice* et définit quelques fonctions utiles comme la multiplication de matrices ou le calcul de puissances :

```
(* hg = haut-gauche, hd = haut-droite, bg = bas-gauche, bd = bas-droite. *)
type matrice = { hg: int; hd: int; bg: int; bd: int } ;;

let mult_matrices { hg = hg1; hd = hd1; bg = bg1; bd = bd1 } { hg = hg2; hd =
    hd2; bg = bg2; bd = bd2 } =
    {
        hg = hg1 * hg2 + hd1 * bg2 ;
        hd = hg1 * hd2 + hd1 * bd2 ;
        bg = bg1 * hg2 + bd1 * bg2 ;
        bd = bg1 * hd2 + bd1 * bd2
    } ;;

let puissance_2 matrice =
    mult_matrices matrice matrice ;;

let rec puissance_n matrice n =
    if 0 = n then { hg = 1; hd = 0; bg = 0; bd = 1 } (* Matrice identite. *)
    else if 0 = n mod 2 then puissance_2 (puissance_n matrice (n/2))
        else mult_matrices matrice (puissance_n matrice (n-1)) ;;
```

Question 9. Uniquement à l'aide des fonctions précédentes, proposez une dernière implémentation de *fibonacci* qui exploite l'équation 1. Challenge : l'algorithme peut tenir sur une seule ligne.

Factorielles de grands entiers

Le code ci-dessous permet de calculer la factorielle d'un entier naturel n.

```
let rec factorielle n =
    if n < 0 then failwith "n doit etre positif."
    else if 0 = n then 1
    else n * factorielle (n-1) ;;
```

Pour ne pas être embêté par la limite de taille des entiers, il est possible d'utiliser le module *Num*. On peut alors déclarer un entier comme grand entier en le castant en *Int*. La fonction précédente devient ainsi :

```
#load "nums.cma" ;;
open Num ;;

(* Pour grands entiers. *)
let factorielle n =
    let rec factorielle_aux n =
        if n < 0 then failwith "n doit etre positif."
        else if 0 = n then (Int 1)
        else (Int n) */ factorielle (n-1)
    in string_of_num (factorielle_aux n) ;;
```

Question 10. Assurez-vous de bien comprendre le code ci-dessus et cherchez dans la documentation du module *Num* à quoi correspond le symbole infixe */.

Question 11. Proposez une version récursive terminale de la fonction *factorielle*.