TP 3 : Tableaux dimensionnables. Introduction à la récursivité: Corrigé

1 Implémentation d'une structure de tableau redimensionnable

Attention : comme pour la structure de pile ou de file vue en cours, il faut faire la différence entre la capacité (nombre d'éléments que la structure peut contenir) et le nombre d'éléments effectivement présents. Avec le type choisi :

```
type 'a tab_redim = {mutable nb: int ; mutable tab: 'a array}
```

Le nombre d'éléments présents est stocké dans `nb`, le nombre d'éléments qu'un élément de type `tab_redim` peut contenir est la taille du tableau `tab`. La différence avec le type pile vue en cours est qu'on a rendu le champ `tab` mutable, pour le remplacer par un tableau plus grand lorsque c'est nécessaire. On pourrait très bien faire ceci pour les piles et files du cours et obtenir ainsi une structure non bornée (capacité infinie), dans laquelle l'ajout se fait en temps constant *amorti*. Voici les fonctions de l'exercice 1 :

```
let creer_tab () = {nb = 0; tab = [| |]} ;;

let acces t i =
  if i< t.nb then
    t.tab.(i)
  else
    failwith "depassement d'indice"
;;

let modif t i x =
  if i< t.nb then
    t.tab.(i) <- x
  else
    failwith "depassement d'indice"
;;

let ajout t x=
  if t.nb = Array.length t.tab then
    let u=Array.make (2*t.nb+1) x in
    for i=0 to t.nb-1 do
      u.(i) <- t.tab.(i)
    done ;
    t.nb <- t.nb + 1 ;
    t.tab <- u
  else
    (t.tab.(t.nb) <- x ; t.nb <- t.nb + 1)
;;

let suppr t =
  if t.nb > 0 then
    (t.nb <- t.nb - 1 ; t.tab.(t.nb))
  else
    failwith "tableau vide"
;;
```

2 Récursivité : quelques fonctions basiques

Exercice 2. La fonction suivante renvoie le nombre de chiffres en base b d'un entier strictement positif. On convient que 0 a zéro chiffre pour que ce soit notre cas de base.

```
let rec nbc n b=match n with
  | 0 -> 0
  | _ -> 1+ nbc (n/b) b
;;
```

Exercice 3. Une fonction puissance :

```
let rec puissance x n=match n with
  | 0 -> 1
  | _ -> x*(puissance x (n-1))
;;
```

Et avec une fonction auxiliaire récursive terminale, faisant usage d'un accumulateur :

```
let puissance2 x n=
  let rec aux acc n=match n with
    | 0 -> acc
    | _ -> aux (acc*x) (n-1)
  in aux 1 n
;;
```

Exercice 4.

```
let rec somme n = match n with
  | 0 -> 0
  | _ -> n + somme (n-1)
;;

let somme2 n =
  let rec aux acc n = match n with
  | 0 -> acc
  | _ -> aux (acc+n) (n-1)
  in aux 0 n
;;
```

Tests :

```
# somme 1000000 ;;
Stack overflow during evaluation (looping recursion?).
# somme2 1000000 ;;
- : int = 500000500000
```

Exercice 5. Exponentiation rapide :

```
let rec expo x n = match n, n mod 2 with
  | 0,_ -> 1
  | _,0 -> expo (x*x) (n/2)
  | _ -> x*expo (x*x) (n/2)
;;
```

Exercice 6. *L'option trace*. Voici le code de la fonction `fibo` :

```
let rec fibo n=match n with
  | 0 | 1 -> 1
  | _ -> fibo (n-1) + fibo (n-2)
;;
```

L'exécution de `fibo 6` après avoir tracé la fonction montre beaucoup d'appels effectués : en fait la complexité de la fonction est exponentielle en `n`, ce qui est mauvais. Voici l'appel `fibo 6` :

```
# #trace fibo ;;
fibo is now traced.
# fibo 6 ;;
fibo <-- 6
fibo <-- 4
fibo <-- 2
fibo <-- 0
fibo --> 1
fibo <-- 1
fibo --> 1
fibo --> 2
fibo <-- 3
fibo <-- 1
```

```
fibo --> 1
fibo <-- 2
fibo <-- 0
fibo --> 1
fibo <-- 1
fibo --> 1
fibo --> 2
fibo --> 3
fibo --> 5
fibo <-- 5
fibo <-- 3
fibo <-- 1
fibo --> 1
fibo <-- 2
fibo <-- 0
fibo --> 1
fibo <-- 1
fibo --> 1
fibo --> 2
fibo --> 3
fibo <-- 4
fibo <-- 2
fibo <-- 0
fibo --> 1
fibo <-- 1
fibo --> 1
fibo --> 2
fibo <-- 3
fibo <-- 1
fibo --> 1
fibo <-- 2
fibo <-- 0
fibo --> 1
fibo <-- 1
fibo --> 1
fibo --> 2
fibo --> 3
fibo --> 5
fibo --> 8
fibo --> 13
- : int = 13
```

3 Introduction aux fractales

L'idée de la fonction `sierpinski` est de tracer le triangle noir, la fonction `aux` (récursive) s'occupant des triangles blancs : elle fait 3 appels récursifs à chaque étape.

```
let triangle p1 p2 p3 couleur =
  set_color couleur;
  fill_poly [|p1; p2; p3|]
;;

let milieu p1 p2=
  let x1, y1 = p1 and x2, y2 = p2 in
  (x1+x2)/2, (y1+y2)/2
;;

let sierpinski n=
  let p1, p2, p3 = (212,84), (812,84), (512,700) in
  triangle p1 p2 p3 black ;
  let rec aux n p1 p2 p3=
    let m1, m2, m3 = milieu p2 p3, milieu p1 p3, milieu p2 p1 in
    triangle m1 m2 m3 white ;
    if n>0 then
      (aux (n-1) p1 m2 m3 ;
      aux (n-1) p2 m1 m3 ;
      aux (n-1) p3 m2 m1)
  in aux n p1 p2 p3
;;
```

Bien entendu, la complexité de `sierpinski n` est exponentielle en n (en $O(3^n)$), mais c'est souvent le cas pour les fractales, et inhérent à ce que l'on souhaite tracer.

Cadeau : le tapis de Sierpinski, avec 8 appels récursifs (le carré délimité par a, b, c et d dans `aux` est divisé en 9, on colorie en noir le carré central, puis on se rappelle sur les 8 petits carrés).

```
let tiers a b=let x,y=a and z,t=b in ((2*x+z)/3, (2*y+t)/3), ((x+2*z)/3, (y+2*t)/3) ;;

let tapis n=
  let a=(100,100) and b=(100,600) and c=(600,600) and d=(600,100) in
  let rec aux a b c d n=match n with
    | 0 -> ()
    | _ -> let ab, ba=tiers a b and bc, cb=tiers b c and cd, dc=tiers c d and ad, da=tiers a d in
           let a1, b1=tiers ad bc and c1, d1=tiers cb da in
           fill_poly [|a1; b1; c1; d1|] ;
           let k=n-1 in
           aux a ab a1 ad k ; aux ab ba b1 a1 k ; aux ba b bc b1 k ; aux b1 bc cb c1 k ;
           aux c1 cb c cd k ; aux d1 c1 cd dc k ; aux da d1 dc d k ; aux ad a1 d1 da k
  in aux a b c d n
;;
```

L'appel `tapis 6` produit :