TP 3 : Tableaux redimensionnables. Introduction à la récursivité.

1 Implémentation d'une structure de tableau redimensionnable

Dans cette section, on implémente une structure de tableau redimensionnable : on va obtenir une structure très proche des listes Python. En Python, une liste `L` supporte les opérations suivantes :

- accès aux éléments (via `L[i]` pour $0 \leq i <$ `len(L)`) en temps constant ;
- modification des éléments (via `L[i]=x`) en temps constant ;
- ajout d'un élément en fin de liste (via `L.append(x)`) en temps constant *amorti* ;
- suppression de l'élément en fin d'une liste non vide (via `L.pop()`) en temps constant.

Un tableau standard permet de réaliser les deux premières opérations (accès et modification des éléments). Pour pouvoir réaliser l'opération d'ajout lorsque le tableau est plein, il faut pouvoir changer le tableau avec lequel on travaille. Pour ce faire, on alloue un nouveau tableau de taille supérieure, dans lequel on copie les éléments de l'ancien tableau, ainsi que le nouvel élément. Passer d'un tableau de taille n à un tableau de taille $n+1$ est très mauvais pour l'ajout successif d'éléments, par contre doubler la taille du tableau alloué permet, lorsqu'on passe d'une taille n à une taille $2n$ (on prendra en fait $2n+1$), de pouvoir réaliser $O(n)$ ajouts en temps constant après le premier ajout (de coût $O(n)$). Ceci mène au type suivant :

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

Dans une instance de ce type, le champ `nb` contient le nombre d'éléments effectivement présents dans le tableau redimensionnable. Le tableau `tab` contient au moins `nb` éléments, ceux effectivement présents sont ceux d'indice variant entre 0 et `nb` -1.

Rappel. Si `t` est de type `tab_redim`, on accède aux champs via `t.nb` et `t.tab`. On peut modifier `nb` (par exemple) par `t.nb <- ...`. Les éléments d'un tableau (standard !) `a` sont accessibles via `a.(i)`, sa longueur par `Array.length a` et on peut modifier un élément par `a.(i) <- ...`.

Exercice 1. On donne la fonction de création (de type `unit -> 'a tab_redim`) d'un tableau redimensionnable vide (remarquez que le tableau vide `[| |]` est polymorphe).

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

Écrire les fonctions suivantes :

```
acces : 'a tab_redim -> int -> 'a
modif : 'a tab_redim -> int -> 'a -> unit
ajout : 'a tab_redim -> 'a -> unit
suppr : 'a tab_redim -> 'a
```

- `acces t i` et `modif t i x` doivent renvoyer le i-ème élément d'un tableau redimensionnable ou le modifier ;
- `ajout t x` rajoute un élément au tableau redimensionnable. Si le tableau `t.tab` est plein, on le remplaçera par un tableau de taille $2n+1$ où n est la taille actuelle de `t.tab`. (Rappel : `Array.make taille x` pour créer un tableau de taille `taille` contenant uniquement des `x`.)
- `suppr t` supprime un élément du tableau redimensionnable et le renvoie. En pratique, il suffit de décrémenter `nb`, pas besoin de toucher à `t.tab`.

Tester vos fonctions, par exemple :

```
let t=creer_tab () ;;
for i=0 to 9 do ajout t i done ;;
print_int (acces t 3) ;;
for i=0 to 9 do print_int (suppr t) ; print_string " "  done ;;
```

affiche à l'écran :

```
#3- : unit = ()
#9 8 7 6 5 4 3 2 1 0 - : unit = ()
```

Voici `t` après cette suite d'instructions dans mon implémentation (remarquez que `t` est vide) :

```
#t ;;
- : int tab_redim =
 {nb = 0; tab = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 7; 7; 7; 7; 7|]}
```

Remarque : mis à part l'ajout d'un élément, toutes les complexités sont bien en temps constant. De temps en temps, l'ajout d'un élément coûte cher (lorsqu'il faut changer `tab`), mais comme ce cas se produit relativement peu souvent, le coût moyen d'un ajout reste borné. La figure 1 montre le coût moyen d'un ajout dans une série de n (pour $n \leq 1000$). Les sauts correspondent aux tailles où il y a changement de tableau `tab`, mais on voit que la complexité *amortie* est bornée par 2.

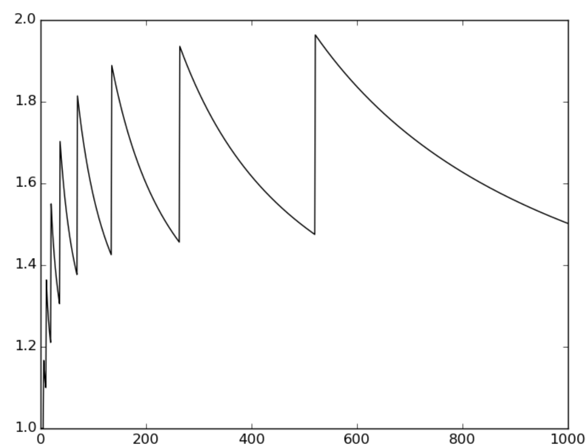

Figure 1: Coût moyen d'un ajout dans une série de n ajouts successifs à partir d'un tableau vide.

2 Récursivité : quelques fonctions basiques

On rappelle qu'on déclare en Caml qu'une fonction est récursive avec `let rec...`. La syntaxe pour un filtrage est la suivante :

```
  match truc with
  | cas 1 -> action
  | cas 2 -> action
  | ...
  | cas p -> action
  | _ -> action (avec _, on est sur que le filtrage est exhaustif).
```

La barre verticale s'obtient avec `alt gr + 6`. Rappel : dans un filtrage, on filtre *sur motifs*. En particulier un nom de variable absorbe tout cas de filtrage !

Exercice 2. Écrire une fonction récursive, donnant le nombre de chiffres en base 10 d'un entier strictement positif (on pourra convenir que zéro n'a aucun chiffre). Généralisez à une base b quelconque.

Exercice 3. ***1.*** Écrire une fonction puissance sur les entiers, récursive, utilisant l'égalité :

$$x^n = \begin{cases} 1 & \text{si } n = 0. \\ x \times x^{n-1} & \text{sinon.} \end{cases}$$

```
#puissance 2 10 ;;
- : int = 1024
```

2. Donner une version utilisant une fonction récursive terminale interne (utiliser un accumulateur).

Exercice 4. Écrire une fonction d'exponentiation rapide récursive basée sur l'égalité

$$x^n = \begin{cases} 1 & \text{si } n = 0. \\ (x^2)^{\frac{n}{2}} & \text{si } n \text{ est pair.} \\ x \times (x^2)^{\frac{n-1}{2}} & \text{si } n \text{ est impair.} \end{cases}$$

On rappelle à toute fin utile que `/` est la division entière sur les entiers, `mod` le modulo.

Exercice 5. Réécrire les fonctions de sommation vues en cours pour sommer les entiers de 1 à n, l'une récursive terminale et l'autre non. Vérifiez la différence sur un grand entier, pour obtenir une exception de dépassement mémoire pour la version non récursive terminale.

Exercice 6. *L'option trace*. Il est possible de *tracer* les appels à une fonction à l'aide de `trace`, elle s'utilise ainsi :

```
#trace fonction;;
```

où `fonction` est bien sûr le nom de la fonction à tracer. Coder une fonction récursive `fibo n` calculant le n-ème terme de la suite de Fibonacci définie par :

$$F_n = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1. \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

On fera deux appels récursifs. Regardez ce que donne le calcul de F_6 (après avoir tracé la fonction).

3 Introduction aux fractales

Exercice 7. *Fractales et graphisme*. Le triangle de Sierpinski de la figure ci-dessous est obtenu de la façon suivante :
- on trace un triangle équilatéral noir ;
- on trace un triangle équilatéral blanc central (de taille divisée par un facteur 3 par rapport au premier) ;
- on recommence le tracé de triangles équilatéraux blancs sur les trois triangles noirs restants.

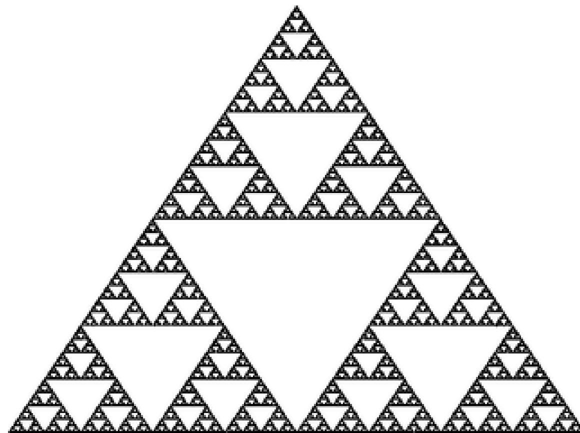

Figure 2: Triangle de Sierpinski

La figure limite obtenue est appelée *triangle de Sierpinski*, mais en pratique, pour y voir quelque chose on ne fait qu'un nombre fini d'étapes.

Pour tracer des figures, on utilise la librairie `graphics`. Pour initialiser un graphique dans une fenêtre 1000×1000, on tapera les lignes suivantes.

```
#load "graphics.cma" ;;
open Graphics ;;
open_graph " 1000x1000" ;;
```

Pour tracer un triangle, on utilisera par exemple la fonction suivante (qui utilise la fonction `fill_poly` du memento).

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

Vous pouvez essayer la commande suivante pour tracer un magnifique triangle noir :

```
triangle (212,84) (812,84) (512,700) black;;
```

Écrire une fonction **`sierpinski n`** permettant d'obtenir une approximation du triangle de Sierpinski, après un nombre n d'étapes. Il suffit de tracer un grand triangle noir comme ci-dessus, et de tracer en blanc des petits triangles par dessus. On pourra définir une fonction donnant le milieu d'un segment délimité par 2 points, utiliser une fonction récursive (interne) qui fera 3 appels récursifs...

Remarque : la complexité de votre fonction devrait être en $O(3^n)$. On évitera donc de prendre n trop grand...

Exercice 8. Générer d'autres fractales : flocon de Von Koch, carré de Sierpinski, etc...