

Correction du TD 3

Julien Reichert

Arithmétique

Exercice 1

Seules les formules récursives nous intéresseront ici, donc on écrit que

$$x \times y = \begin{cases} 0 & \text{si } y = 0 \\ x + x \times (y - 1) & \text{sinon} \end{cases}$$

(en pouvant échanger les rôles de x et y ici, contrairement à la suite), et

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

En optimisant ces deux opérations suivant le même principe, on a aussi :

$$x \times y = \begin{cases} 0 & \text{si } y = 0 \\ (x + x) \times \frac{y}{2} & \text{sinon avec } y \text{ pair} \\ x + (x + x) \times \frac{y-1}{2} & \text{sinon avec } y \text{ impair} \end{cases}$$

Le dernier choix peut se contenter de retrancher un à y sans perte d'efficacité asymptotiquement car on se ramène à une valeur paire permettant une division par deux à l'étape suivante, aboutissant à la même valeur que ce qu'on fait ici en une étape.

L'exponentiation est analogue :

$$x^y = \begin{cases} 1 & \text{si } y = 0 \\ (x \times x)^{\frac{y}{2}} & \text{sinon avec } y \text{ pair} \\ x \times (x \times x)^{\frac{y-1}{2}} & \text{sinon avec } y \text{ impair} \end{cases}$$

Le nombre d'additions pour multiplier par y passe de y à au plus $2 \log_2(y)$, comme le nombre de multiplications pour élever à la puissance y .

Implémentation :

```
let rec mult x y = if y = 0 then 0 else x + mult x (y-1);;
```

```
let rec puiss x y = if y = 0 then 1 else x * puiss x (y-1);;
```

```

let rec multrap x y =
  if y = 0 then 0
  else if y mod 2 = 0 then multrap (x+x) (y/2)
  else x + multrap (x+x) ((y-1)/2);;

```

```

let rec exporap x y =
  if y = 0 then 1
  else if y mod 2 = 0 then exporap (x*x) (y/2)
  else x * exporap (x*x) ((y-1)/2);;

```

Il n'est pas spécialement demandé d'optimiser la multiplication dans l'exponentiation rapide.

Exercice 2

Soient u et v non nuls. Le PGCD de u et de v est aussi le PGCD de v et de tout nombre de la forme $u + kv$ avec $k \in \mathbb{Z}$. En vue de se rapprocher d'un cas de base, on prendra k de sorte que $u + kv$ soit le reste dans la division euclidienne de u par v , une fois qu'on a organisé les valeurs de sorte que u et v soient strictement positifs avec $u \geq v$.

Ainsi, la formule résumant le tout est que le PGCD de u et v (avec u et v positifs) est :

$$\begin{cases} \text{indéterminé} & \text{si } u = v = 0 \\ u & \text{sinon si } v = 0 \\ \text{PGCD}(v, r) & \text{sinon, avec } r \text{ le reste de la division euclidienne de } u \text{ par } v \end{cases}$$

et on l'écrit ainsi en OCaml :

```

let pgcd u v =
  let rec pgcdpos u v =
    if u = 0 && v = 0 then failwith "Indéterminé";
    if v = 0 then u
    else pgcdpos v (u mod v)
  in pgcdpos (abs u) (abs v);;

```

Si en fait u est plus petit que v , le premier appel récursif échange en fait les deux valeurs. Cela nécessite une précaution pour la preuve de terminaison, mais le code est plus léger.

Exercice 3

On cherche a et b tels que $12a + 19b = 1$ (en ayant anticipé la valeur du PGCD). Or on sait que $19 = 12 \times 1 + 7$, que $12 = 7 \times 1 + 5$, que $7 = 5 \times 1 + 2$ et que $5 = 2 \times 2 + 1$. En remontant le calcul, on retrouve successivement $1 = 5 - 2 \times 2 = 5 - (7 - 5) \times 2 = 5 \times 3 - 7 \times 2 = (12 - 7) \times 3 - 7 \times 2 = 12 \times 3 - 7 \times 5 = 12 \times 3 - (19 - 12) \times 5 = 12 \times 8 - 19 \times 5$.

Pour programmer ceci, il s'agit de « descendre » vers le PGCD puis de réutiliser les quotients dans les divisions euclidiennes intermédiaires pour garder les égalités.

```

let bezout u v =
  let rec bezoutpos u v =
    if u = 0 && v = 0 then failwith "Indéterminé";
    if u = 0 then (0, 1) else if v = 0 then (1, 0)
    else let q = u / v and r = u mod v in
    let a, b = bezoutpos v r in (b, a-q*b)
  in let apos, bpos = bezoutpos (abs u) (abs v)
  in let a = if u >= 0 then apos else -apos
  in let b = if v >= 0 then bpos else -bpos
  in (a, b);;

```

Un peu d'ASCII Art

Exercice 4

Un escalier sur n lignes est vide si $n = 0$ (on peut aussi faire un cas de base à $n = 1$) et sinon c'est un escalier sur $n - 1$ lignes puis une ligne de n étoiles.

```

let rec escalier n =
  if n > 0 then
    begin
      escalier (n-1);
      print_endline (String.make n '*')
    end;;

```

Exercice 5

Un triangle sur n lignes est vide si $n = 0$ (même remarque) et sinon c'est un triangle sur $n - 1$ lignes suivi d'une ligne de $2n - 1$ étoiles. Cependant, le nombre d'espaces avant les étoiles dans le triangle sur $n - 1$ lignes nécessite d'avoir à disposition le nombre final de lignes au moment de gérer l'appel récursif, d'où l'utilisation d'une fonction principale contenant une sous-fonction récursive (on peut faire partir l'argument de cette dernière de l'argument de la fonction principale ou de zéro, au choix).

```

let triangle n =
  let rec aux k =
    if k > 0 then
      begin
        aux (k-1);
        print_string (String.make (n-k) ' ');
        print_endline (String.make (2*k-1) '*')
      end
    in aux n;;

```

Exercice 6

On reprend le principe précédent en compliquant un peu la formule. Un diamant sur n lignes (avec n impair) s'obtient en faisant une récursion allant de 1 à $\frac{n+1}{2}$ (ou vice-versa là aussi, en adaptant),

et dans cette récursion pour toute valeur k sauf $\frac{n+1}{2}$ on fait $\frac{n+1}{2} - k$ espaces puis $2k - 1$ étoiles, l'appel récursif pour $k + 1$ puis la même ligne qu'avant l'appel récursif, alors que pour k valant $\frac{n+1}{2}$ on ne fait que la ligne correspondante une fois.

Une fonction annexe sera la bienvenue pour des soucis de lisibilité.

```
let ligne m k =
  print_string (String.make (m-k) ' ');
  print_endline (String.make (2*k-1) '*');;

let diamant n =
  if n mod 2 = 0 then failwith "Nombre pair de lignes";
  let m = (n+1)/2 in
  let rec aux k =
    if k = m then ligne m m
    else (ligne m k; aux (k+1); ligne m k)
  in aux 1;;
```

Exercice 7

Rien de spécialement compliqué une fois les exercices précédents effectués. On peut en pratique envisager de faire une fonction faisant l'escalier à l'envers (il suffit d'échanger l'ordre entre l'impression et l'appel récursif) pour enchaîner un escalier de taille n à l'envers et un escalier de taille n en retirant à l'un des deux la ligne avec une seule étoile.

De manière assez intéressante, la figure demandée peut se faire sans sous-fonction cette fois. Il s'agit de faire, si on veut n lignes avec n impair, une ligne de taille $\frac{n+1}{2}$, la figure avec $n - 2$ lignes, et encore une ligne de taille $\frac{n+1}{2}$, sauf si $n = 1$ où on imprime juste une ligne avec une étoile.

```
let rec figure n =
  if n mod 2 = 0 then failwith "Nombre pair de lignes";
  let m = (n+1)/2 in
  print_endline (String.make m '*');
  if n > 1 then (figure (n-2); print_endline (String.make m '*'));;
```

Permutations, sous-ensembles, etc.

Exercice 8

Une liste de p éléments d'un ensemble à n éléments est simplement la liste vide si p est nul ou une liste dont la tête est un élément de l'ensemble et la queue une liste de $p - 1$ éléments de l'ensemble.

Pour toutes les engendrer (rappel : il y en a n^p), on doit prendre toutes celles qui sont données pour la valeur $p - 1$, tous les éléments de l'ensemble et produire les nouvelles listes.

Ici, l'ensemble à n éléments sera représenté par une liste, et on utilisera des fonctions du module `List` pour raccourcir le code.

```

let rec toutes_listes p ens =
  if p = 0 then [[]] (* attention, il faut une liste de taille 1 ici *)
  else
    let l = toutes_listes (p-1) ens in
    List.fold_left (fun accu elt -> (List.map (fun li -> elt::li) l)@accu) [] ens;;

```

Exercice 9

Un arrangement de p éléments d'un ensemble à n éléments est la liste vide si p est nul ou une liste dont la tête est un élément de l'ensemble et la queue un arrangement de $p - 1$ éléments de l'ensemble privé de ce qui a été mis dans la tête.

Ici, la récursion qui était gérée par le `List.fold_left` de la correction précédente sera faite explicitement pour montrer les deux versions (une troisième récursion pour remplacer le `List.map` rendrait la lecture plus difficile, mais rien n'empêche de créer des fonctions extérieures pour cela aussi).

```

let separations l = (* tous les couples (élément, reste) possibles pour une liste *)
  let rec aux reste liste = match liste with
  | [] -> []
  | a::q -> (a, reste@q) :: aux (a::reste) q
  in aux [] l;;

```

```

let rec arrangements p ens =
  (* éventuellement vérifier que p est inférieur à la taille de l'ensemble,
  mais une seule fois en englobant dans une fonction globale, vu le coût *)
  if p = 0 then [[]]
  else
    let rec aux accu l = match l with
    | [] -> accu
    | (elt, reste)::q -> let liste = arrangements (p-1) reste
      in aux ((List.map (fun li -> elt::li) liste)@accu) q
    in aux [] (separations ens);;

```

Exercice 10

Pour les permutations, il suffit de prendre $p = n$. Le programme se fera en une ligne.

```

let permutations ens = arrangements (List.length ens) ens;;

```

Exercice 11

Il est en pratique équivalent de considérer l'ensemble des combinaisons d'un ensemble et celui de ses arrangements croissants (forcément strictement, car il ne peut pas y avoir d'égalité dans un ensemble mathématique). Ceci vaut en fait pour n'importe quelle relation d'ordre et nous allons en profiter pour utiliser comme relation d'ordre l'endroit où un élément se trouve dans la liste qui le modélise (on évitera cependant les doublons dans la liste, qui feraient croire à la non unicité au sein du résultat).

Grâce à l'utilisation fortuite d'une fonction annexe dans ce qui précède, la fonction `arrangements` donne les combinaisons si on remplace l'appel à `separations` par l'appel à `separations_cro` définie ci-après :

```
let rec separations_cro l = (* couples (élément, suite) *)
  match l with
  | [] -> []
  | a::q -> (a, q) :: separations_cro q;;
```

Les tours de Hanoï

Exercice 12

La magie de la récursion, c'est que justement on peut utiliser la solution au problème pour résoudre le problème, tout comme une preuve par récurrence permet de dire que par exemple une propriété \mathcal{P} est vraie pour une valeur comme 254656 de n dès qu'on peut supposer (et l'acter ultérieurement par une bonne initialisation et une preuve d'hérédité) qu'elle l'est pour la valeur 254655.

Ainsi, on peut résoudre le problème avec un seul anneau, car il est déplaçable où l'on veut, et si on peut le résoudre avec $n - 1$ anneaux, la solution pour n anneaux est de mettre en œuvre cette résolution pour les $n - 1$ premiers, de déplacer le dernier sur le piquet resté libre, puis de remettre en œuvre la résolution pour les $n - 1$ déjà déplacés en les envoyant sur ce même troisième piquet.

On utilisera donc une fonction auxiliaire pour avoir le départ et l'arrivée (l'intermédiaire s'en déduisant).

```
let hanoi n =
  let rec hanoi_aux k dep arr =
    if k > 0 then begin
      hanoi_aux (k-1) dep (6-dep-arr); (* c'est beau *)
      Printf.printf "Déplacer l'anneau %d du piquet %d au piquet %d\n" k dep arr;
      hanoi_aux (k-1) (6-dep-arr) arr
    end
  in hanoi_aux n 1 3;; (* choix 1 et 3 arbitraires, 2 est l'autre intuitivement *)
```

Exercice 13

Ici il s'agit de considérer à chaque étape un anneau de plus, pour avoir les k premiers anneaux sur un même piquet pour k augmentant d'au moins un à chaque résolution du problème des tours de Hanoï avec un nombre restreint d'anneaux.

Il reste à déterminer les appels à effectuer. En pratique, à partir de trois listes croissantes, le problème devient de prendre la pile d'anneaux depuis la liste qui contient 1 et de la placer sur le sommet de la liste qui contient la plus petite valeur parmi les deux autres (ou la seule non vide sinon). On passera une fois de plus par des fonctions annexes.

Pour faciliter les choses, l'argument sera en OCaml un tableau avec les trois listes.

```

let tete l = match l with
| [] -> max_int
| a::q -> a;;

let arguments_hanoi tab =
  match tab.(0), tab.(1), tab.(2) with
  | _, [], [] | [], _, [] | [], [], _ -> -1, -1
  | 1::_, l, q -> 0, if tete l < tete q then 1 else 2
  | l, 1::_, q -> 1, if tete l < tete q then 0 else 2
  | l, q, 1::_ -> 2, if tete l < tete q then 0 else 1
  | _, _, _ -> failwith "Cas impossible !"
;;

let rec deblayer l k = match l with
| [] -> []
| a::q -> if a >= k then l else deblayer q k;;

let rec hanoi_a k dep arr = (* en fait on en a besoin aussi *)
  if k > 0 then begin
    hanoi_a (k-1) dep (3-dep-arr); (* indices de 0 à 2 cette fois *)
    Printf.printf "Déplacer l'anneau %d du piquet %d au piquet %d\n" k dep arr;
    hanoi_a (k-1) (3-dep-arr) arr
  end;;

let hanoi_partiel tab =
  let tab1 = Array.copy tab in
  let rec aux () = (* bon, en fait c'est un while *)
    let deb, fin = arguments_hanoi tab1 in
    if deb <> -1 then
      begin
        let k = List.hd tab1.(fin) in
        hanoi_a (k - 1) deb fin;
        tab1.(fin) <- 1::tab1.(fin); (* c'est une sorte de jeton, le 1 *)
        tab1.(deb) <- deblayer tab1.(deb) k;
        aux ()
      end
    in aux ();;

```

Attention : la solution proposée ici n'est pas optimale, on remarquera qu'on déplace deux fois de suite le plus petit anneau entre deux appels à `hanoi_a`. Ceci étant, on peut imaginer qu'à part cela on a l'optimalité, et l'adaptation nécessaire serait lourde en programmation alors que le code précédent est déjà bien long.

Le retour de la tortue

Exercice 14

Il n'est pas nécessaire d'utiliser une fonction auxiliaire dans l'absolu, mais chaque déplacement doit se faire sur un certain nombre de pixels, et une étape supplémentaire triple alors la taille de l'image obtenue, d'où la suggestion que le nombre de pixels dépende du nombre d'étapes.

Le module graphique disponible avec OCaml est hors programme et peu pratique, notamment car il faudrait avoir la position et l'angle en mémoire pour les déplacements, contrairement au pilotage de la tortue.

Dessiner un flocon d'indice 0 revient à faire un triangle équilatéral pointant vers le bas, donc (en admettant une orientation initiale vers la droite) tracer une ligne de taille arbitraire L , tourner à droite d'un tiers de tour, le tout trois fois de suite. En pratique, le flocon d'indice 0 est la ligne elle-même, dans l'optique de la récursion, donc on appellera dans une boucle de taille trois cette fonction récursive à chaque tour.

Dessiner un flocon d'indice $n+1$ revient à dessiner quatre flocons d'indice n , en séparant les dessins par trois rotations respectivement d'un sixième de tour vers la gauche, d'un tiers de tour vers la droite et de nouveau d'un sixième de tour vers la gauche (pour un angle global nul quel que soit l'indice, y compris 0).

Ici, la solution est écrite en Logo.

```
to flocon :n
  if :n < 1 [fd 5 stop]
  flocon :n-1 lt 60
  flocon :n-1 rt 120
  flocon :n-1 lt 60
  flocon :n-1
end
rt 90
repeat 3
[
  flocon 3
  rt 120
]
```

Exercice 15

Présentation d'autres courbes fractales sur suggestion (ultérieure) de la part de la classe...

Exemple par Ezequiel en 2024 : triangle de Sierpinski.

```
to sierpinski :l :resolution
  if :l > :resolution [
    pendown
    repeat 3 [
      fd :l / 2
      sierpinski :l / 2 :resolution
      fd :l / 2
      lt 120
    ]
    penup
  ]
end

cs
ht
rt 90
penup
setpensize [1 1]
setxy -200 -200
sierpinski 400 10
wait 1
```