Chapitre 6

option informatique

Récursivité

1. Introduction

Jusqu'à présent, nous nous sommes contentés d'une définition informelle de la récursivité : est récursive toute fonction qui intervient dans sa définition. Bien entendu, ce qualificatif est à mettre en rapport avec le principe de récurrence, dont on peut rappeler l'énoncé :

Théorème (principe de récurrence simple). — *Soit $\mathscr{P}$ un prédicat défini sur $\mathbb{N}$, tel que $\mathscr{P}(0)$ est vrai, ainsi que l'implication $\forall n \in \mathbb{N}^*, \mathscr{P}(n-1) \Longrightarrow \mathscr{P}(n)$. Alors pour tout entier $n \in \mathbb{N}$, $\mathscr{P}(n)$ est vrai.*

Ce principe permet de démontrer que la donnée d'un élément $a \in \mathrm{E}$ et d'une application $f : \mathrm{E} \to \mathrm{E}$ permet de définir sans ambiguïté une suite $(u_n)_{n\in\mathbb{N}}$ à l'aide des relations :

$$u_0 = a \quad \text{et} \quad \forall n \in \mathbb{N}^*,\ u_n = f(u_{n-1})$$

et que tout élément de cette suite peut être *calculé* à l'aide d'une définition qui suivra le schéma suivant :

```
let rec u = function
  | 0 -> a
  | n -> f (u (n-1)) ;;
```

Nous dirons dans ce cas que la fonction se *termine*, autrement dit que l'appel `u n` lorsque n est un entier naturel retournera une réponse en un temps fini.

Mais toutes les fonctions récursives ne suivent pas un modèle aussi simple. La fonction Q de Hofstadter, par exemple, est « définie » sur $\mathbb{N}^*$ par les relations :

```
let rec q = function
  | 1 -> 1
  | 2 -> 1
  | n -> q (n-q (n-1)) + q (n-q (n-2)) ;;
```

mais la preuve de sa terminaison reste un problème ouvert. Autrement dit, il n'a pas encore été prouvé que pour un entier n quelconque, le calcul de $\mathrm{Q}(n)$ retournera un résultat en un nombre fini d'étapes.

• Le problème de l'arrêt

Le problème de l'arrêt consiste à déterminer si le calcul d'une fonction donnée se termine ou pas. Turing a démontré qu'il s'agit d'un problème indécidable (au sens algorithmique du terme), autrement dit qu'il ne peut exister de moyen algorithmique de déterminer à l'avance si un calcul va se terminer ou pas. Sa preuve repose sur le principe de la diagonale de Cantor.
Il faut commencer par observer qu'un programme n'est autre qu'une suite finie de bits ; autrement dit, si on se restreint par exemple à l'ensemble $\mathscr{F}$ des fonctions Caml de type *int -> int*, alors $\mathscr{F}$ est en bijection avec un sous-ensemble de l'ensemble des suites finies sur $\{0,1\}$. Ce dernier ensemble étant dénombrable, il en est de même de $\mathscr{F}$. Il existe donc une bijection $\varphi : \mathbb{N} \to \mathscr{F}$.
Nous allons maintenant supposer qu'il existe une fonction **`termine`** de type *int -> int -> bool* capable de déterminer si un calcul se termine ou pas. Autrement dit,

$$\texttt{termine p q} = \begin{cases} \texttt{true} & \text{si le calcul de } \varphi(p)(q) \text{ se termine} \\ \texttt{false} & \text{sinon} \end{cases}$$

Considérons alors la fonction :

```
let rec f = function
  | p when termine p p -> f p
  | p                  -> 0 ;;
```

et notons $r \in \mathbb{N}$ tel que $f = \varphi(r)$. Il reste à considérer la valeur de **`termine r r`** pour aboutir à une absurdité : si **`termine r r = true`** alors **`f r`** bouclera indéfiniment ce qui est absurde, mais si **`termine r r = false`** alors **`f r`** retournera 0, ce qui est tout aussi contradictoire.

Sans prétendre à l'exhaustivité, nous allons maintenant définir un cadre formel plus général que le simple principe de récurrence pour prouver la terminaison des fonctions récursives que nous seront conduits à rencontrer dans la suite de ce cours.

2. Fonctions inductives

2.1 Ensembles bien fondés

Commençons par rappeler la différence entre élément minimal et plus petit élément. Si $(E, \leqslant)$ est un ensemble ordonné, A une partie non vide de E et $a \in A$, on dit que :

- a est un *élément minimal* de A lorsque pour tout $x \in A$, $x \leqslant a \Longrightarrow x = a$;
- a est le *plus petit élément* de A lorsque pour tout $x \in A$, $a \leqslant x$.

Bien entendu, le plus petit élément, s'il existe, est un élément minimal. En revanche, l'implication réciproque n'est pas toujours vraie dès lors que l'ordre n'est pas total. En outre, une partie peut posséder plusieurs éléments minimaux, mais au plus un plus petit élément.

Définition. — *On dit qu'un ensemble ordonné* $(E, \leqslant)$ *est* bien fondé *lorsque toute partie non vide possède (au moins) un élément minimal, et* bien ordonné *lorsque toute partie non vide possède un plus petit élément.*

Bien entendu, tout ensemble bien ordonné est a fortiori bien fondé ; c'est la cas par exemple de $(\mathbb{N}, \leqslant)$. En revanche, $(\mathbb{Z}, \leqslant)$ n'est pas bien fondé.

On peut remarquer enfin qu'un bon ordre est nécessairement total. En effet, si $(E, \leqslant)$ est bien ordonné, alors pour tout couple $(a, b) \in E^2$, l'ensemble $\{a, b\}$ admet un plus petit élément, donc $a \leqslant b$ ou $b \leqslant a$.

Lorsqu'on munit $\mathbb{N}^2$ de l'ordre produit : $(a, b) \leqslant (a', b') \iff a \leqslant a'$ et $b \leqslant b'$, on obtient un ensemble bien fondé mais pas bien ordonné (puisque l'ordre n'est pas total). En effet, si A est une partie non vide de $\mathbb{N}^2$, l'élément (a_0, b_0) défini par :

$$a_0 = \min\Big\{a \in \mathbb{N} \;\Big|\; \exists b \in \mathbb{N} \text{ tq } (a, b) \in A\Big\} \quad \text{et} \quad b_0 = \min\Big\{b \in \mathbb{N} \;\Big|\; (a_0, b) \in A\Big\}$$

en est un élément minimal.
Muni de l'ordre lexicographique : $(a, b) \leqslant (a', b') \iff a < a'$ ou $(a = a'$ et $b \leqslant b')$, $\mathbb{N}^2$ est cette fois bien ordonné (l'élément (a_0, b_0) est le plus petit élément de A).

Remarque. Plus généralement, on peut définir l'ordre lexicographique sur l'ensemble des suites finies d'éléments d'un ensemble bien ordonné. C'est bien entendu l'ordre utilisé dans les dictionnaires.

Le principe de récurrence se généralise alors de la manière suivante :

Théorème (principe d'induction). — *Soit* $(E, \leqslant)$ *un ensemble bien fondé,* A *une partie non vide de* E, *et* $\varphi : E \setminus A \to E$ *une application vérifiant :* $\forall x \in E \setminus A$, $\varphi(x) \prec x$. *On considère un prédicat* $\mathscr{P}$ *défini sur* E, *vérifiant :*

- *pour tout* $a \in A$, $\mathscr{P}(a)$ *est vrai ;*
- *pour tout* $x \in E \setminus A$, $\mathscr{P}\big(\varphi(x)\big) \Longrightarrow \mathscr{P}(x)$.

Alors $\mathscr{P}(x)$ *est vrai pour tout élément* x *de* E.

Preuve. Soit X l'ensemble des éléments x de E tel que $\mathscr{P}(x)$ soit faux, et supposons X non vide. X possède un élément minimal x_0. $\mathscr{P}(x_0)$ est faux, donc $x_0 \in E \setminus A$. Mais alors $\varphi(x_0) \prec x_0$, ce qui implique que $\varphi(x_0) \notin X$ (de par le caractère minimal de x_0). $\mathscr{P}\big(\varphi(x_0)\big)$ est donc vrai, ce qui implique que $\mathscr{P}(x_0)$ l'est aussi, *i.e.* $x_0 \notin X$. On aboutit bien à une contradiction. □

Lorsqu'on pose $E = \mathbb{N}$, $A = \{0\}$ et $\varphi : n \longmapsto n - 1$, on retrouve le principe de la récurrence simple.

2.2 Fonctions inductives

Le principe d'induction permet de justifier qu'on définit sans ambiguïté une fonction $f : \mathrm{E} \to \mathrm{F}$ en procédant de la manière suivante : si $g : \mathrm{A} \to \mathrm{F}$ et $h : \mathrm{E} \setminus \mathrm{A} \times \mathrm{F} \to \mathrm{F}$ sont deux fonctions quelconques, on pose :

$$\begin{aligned} \forall a \in \mathrm{A}, \quad & f(a) = g(a) \\ \forall x \in \mathrm{E} \setminus \mathrm{A}, \quad & f(x) = h\Big(x, f \circ \varphi(x)\Big) \end{aligned}$$

Une fonction ainsi définie est dite *inductive*.

Exemple. La fonction factorielle est définie inductivement par les relations :

$$0! = 1 \quad \text{et} \quad \forall n \geqslant 1,\ n! = n \times (n-1)!.$$

Nous avons ici $\mathrm{E} = \mathbb{N}$, $\mathrm{A} = \{0\}$, $\varphi : n \longmapsto n-1$, $g : x \longmapsto 1$ et $h : (x, y) \longmapsto xy$.

Exemple. La fonction pgcd est définie inductivement par les relations :

$$\forall q \in \mathbb{N},\ \mathrm{pgcd}(0, q) = q \quad \text{et} \quad \forall (p, q) \in \mathbb{N}^* \times \mathbb{N},\ \mathrm{pgcd}(p, q) = \mathrm{pgcd}(q \bmod p, p).$$

Nous avons ici $\mathrm{E} = \mathbb{N}^2$, $\mathrm{A} = \Big\{(0, q) \;\Big|\; q \in \mathbb{N}\Big\}$, $\varphi : (p, q) \longmapsto (q \bmod p, p)$, $g : (0, x) \longmapsto x$ et $h : (x, y) \longmapsto y$. Lorsqu'on munit $\mathbb{N}^2$ de l'ordre lexicographique, on a bien : $\forall (p, q) \in \mathrm{E} \setminus \mathrm{A}$, $\varphi(p, q) \prec (p, q)$. Ceci nous assure que la fonction définie ci-dessous se termine :

```
let rec pgcd = fun
  | 0 q -> q
  | p q -> pgcd (q mod p) p ;;
```

et le principe d'Euclide nous assure que le résultat calculé est bien le pgcd des entiers p et q.

Remarque. Toute fonction définie inductivement va être calculable à l'aide d'une définition suivant peu ou prou le schéma suivant :

```
let rec f = function
  | x when dans_A x -> g x
  | x               -> h x (f (phi x)) ;;
```

à condition d'avoir au préalable défini les fonctions **dans_A** : *'a -> bool*, **phi** : *'a -> 'a*, **g** : *'a -> 'b* et **h** : *'a -> 'b -> 'b*.

• Appels récursifs multiples

La définition que nous avons adoptée ne permet qu'un seul appel récursif, mais la généralisation à plusieurs appels est aisée. Si l'on souhaite par exemple définir inductivement une fonction faisant deux fois appel à elle-même dans sa définition, il faudra considérer deux fonctions φ_1 et φ_2 de $\mathrm{E} \setminus \mathrm{A}$ dans E vérifiant $\varphi_1(x) \prec x$ et $\varphi_2(x) \prec x$ et suivre le schéma suivant :

$$\begin{aligned} \forall a \in \mathrm{A}, \quad & f(a) = g(a) \\ \forall x \in \mathrm{E} \setminus \mathrm{A}, \quad & f(x) = h\Big(x, f \circ \varphi_1(x), f \circ \varphi_2(x)\Big) \end{aligned}$$

où h est une fonction définie sur $\mathrm{E} \setminus \mathrm{A} \times \mathrm{F} \times \mathrm{F}$ à valeurs dans F.

Par exemple, la suite de Fibonacci est définie inductivement par les relations :

$$u_0 = 0,\ = u_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ u_n = u_{n-1} + u_{n-2}.$$

Il suffit de poser $\mathrm{E} = \mathbb{N}$, $\mathrm{A} = \{0, 1\}$, $\varphi_1 : x \longmapsto x - 1$ et $\varphi_2 : x \longmapsto x - 2$ pour rentrer dans le cadre énoncé ci-dessus.

3. Récursivité terminale

3.1 Pile d'exécution d'une fonction

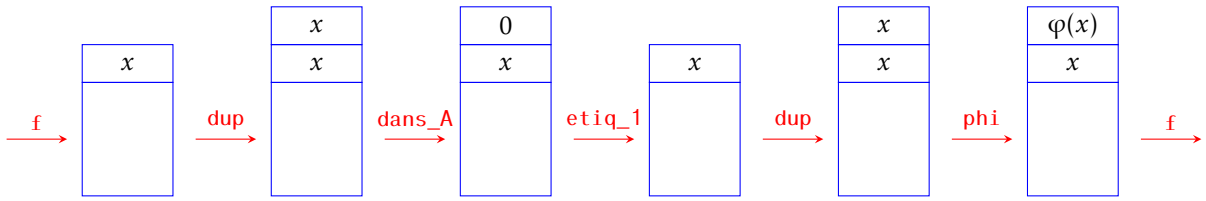
Revenons à la définition que nous avons adopté d'une fonction inductive :

```
let rec f = function
  | x when dans_A x -> g x
  | x               -> h x (f (phi x)) ;;
```

et cherchons à comprendre comment se déroule l'exécution d'une telle fonction.
Tout d'abord, une fonction écrite dans un langage de programmation de haut niveau doit être traduite en langage machine par le compilateur. Nous n'allons bien entendu pas rentrer dans les détails de cette traduction, mais on retiendra qu'à chaque fonction est associée une *pile d'exécution* qui est utilisée pour emmagasiner un certain nombre de valeurs, par exemple l'adresse de l'endroit où cette fonction devra retourner à la fin de son exécution, ou encore les paramètres et variables locales de cette fonction.
En simplifiant, la traduction en assembleur de la définition d'une fonction inductive ressemble à une suite d'instructions ayant l'allure suivante :

```
fonc_f :
   dup               ;   (* duplique le sommet de la pile *)
   call fonc_dans_A  ;   (* évalue si x est dans A        *)
   jz etiq_1         ;   (* saut si résultat "faux"       *)
   call fonc_g       ;   (* évalue g(x)                   *)
                     ;
etiq_1 :
   dup               ;   (* premier argument de h         *)
   call fonc_phi     ;   (* évalue phi(x)                 *)
   call fonc_f       ;   (* appel récursif de f           *)
   call fonc_h       ;   (* appel de h                    *)
   ret
```

Observons maintenant l'évolution de la pile d'exécution lors d'un appel récursif :



On constate qu'après un appel récursif, la pile contient un élément de plus. La taille de celle-ci va donc croître linéairement avec le nombres d'appels imbriqués, ce qui peut éventuellement occasionner un débordement de la capacité de la pile, cette dernière ayant une taille majorée. En Caml, ceci se traduira par le déclenchement de l'exception **`Out_of_memory`**. C'est un phénomène qu'on peut aisément expérimenter :

```
# let n = ref 0 ;;
n : int ref = ref 0
# let rec f () =
    n := !n + 1 ; 1 + f () ;;
f : unit -> int = <fun>
# try f () with Out_of_memory -> !n ;;
- : int = 131027
```

On observe qu'au bout de 131 027 appels à la fonction f l'interprète de commande est à cours de mémoire.

Il est aussi possible de visualiser l'évolution de la pile d'exécution à l'aide de l'instruction **`trace`**. Cette fonction, de type *string -> unit*, prend en argument le nom d'une fonction. Dès lors qu'une fonction **`f`** est tracée, la liste des appels à celle-ci sera affichée avec les conventions suivantes :

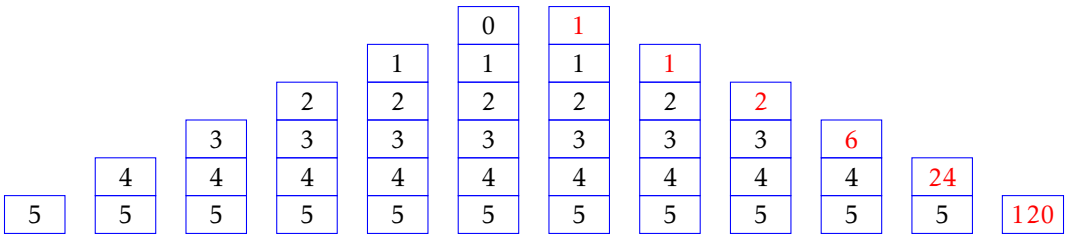
- **`f <- x`** : la fonction **`f`** est appelée avec l'argument **`x`** ;
- **`f -> y`** : la fonction **`f`** renvoie la valeur **`y`**.

Expérimentons ceci en traçant la fonction factorielle définie par :

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n-1) ;;
```

```
# trace "fact" ;;
La fonction fact est dorénavant tracée.
- : unit = ()
# fact 5 ;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
fact --> 120
- : int = 120
```

La pile d'exécution de la fonction **fact** évolue comme suit :



3.2 Récursivité terminale

Considérons de nouveau un ensemble bien ordonné $(E, \preccurlyeq)$, A une partie de E et $\varphi : E \setminus A \to E$ tel que : $\forall x \in E \setminus A,\ \varphi(x) \prec x$. Si $g : A \to F$ est une fonction quelconque, on définit une unique fonction $f : E \to F$ en posant :

$$\forall a \in A, \quad f(a) = g(a)$$
$$\forall x \in E \setminus A, \quad f(x) = f\big(\varphi(x)\big)$$

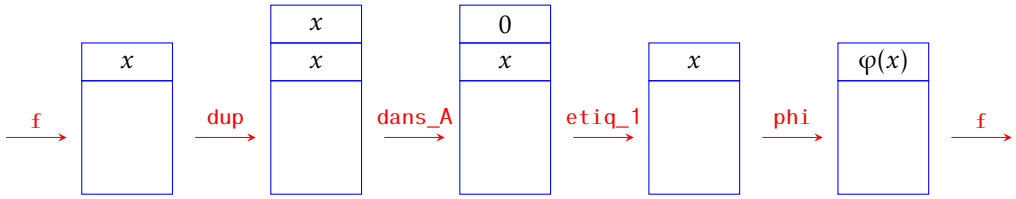
Il s'agit d'un cas particulier de fonction inductive. Une telle définition est qualifiée de *récursive terminale* car l'appel récursif est la dernière opération que l'on effectue pour calculer $f(x)$ lorsque $x \in E \setminus A$.
Cette définition se traduira en Caml de la manière suivante :

```
let rec f = function
  | x when dans_A x -> g x
  | x               -> f (phi x) ;;
```

et le programme pseudo-compilé devient :

```
fonc_f :
   dup              ;   (* duplique le sommet de la pile *)
   call fonc_dans_A ;   (* évalue si x est dans A        *)
   jz etiq_1        ;   (* saut si résultat "faux"       *)
   jmp fonc_g       ;   (* va évaluer g(x)               *)
                    ;
etiq_1 :
   call fonc_phi    ;   (* évalue phi(x)                 *)
   call fonc_f      ;   (* appel récursif de f           *)
   ret
```

Cette fois, la pile d'exécution ne va pas croitre, il n'y aura donc aucun risque de dépassement de capacité :



Par exemple, le calcul du pgcd de deux entiers à l'aide de l'algorithme d'Euclide est récursif terminal :

```
let rec pgcd = function
  | (0, q) -> q
  | (p, q) -> pgcd (q mod p, p)  ;;
```

```
# trace "pgcd" ;;
La fonction pgcd est dorénavant tracée.
- : unit = ()
# pgcd (95,115) ;;
pgcd <-- 95, 115
pgcd <-- 20, 95
pgcd <-- 15, 20
pgcd <-- 5, 15
pgcd <-- 0, 5
pgcd --> 5
pgcd --> 5
pgcd --> 5
pgcd --> 5
pgcd --> 5
- : int = 5
```

On constate que le retour des différents appels récursifs se fait par simple transmission, sans calcul supplémentaire.

4. Exercices

4.1 Fonctions récursives

Exercice 1 Montrer qu'un ensemble ordonné $(E, \leqslant)$ est bien fondé si et seulement s'il n'existe pas de suite strictement décroissante dans E.

Exercice 2 Déterminer ce que calcule la fonction suivante (et le démontrer) :

```
let rec f n =
   if n > 100 then n-10 else f (f (n+11)) ;;
```

On notera qu'il n'est pas évident qu'une telle fonction donne un résultat.

Exercice 3 La fonction d'Ackermann est définie sur $\mathbb{N} \times \mathbb{N}$ par :

```
let rec Ack = fun
  | 0 p -> p + 1
  | n 0 -> Ack (n-1) 1
  | n p -> Ack (n-1) (Ack n (p-1)) ;;
```

Prouver qu'elle se termine toujours, puis donner une forme close (c'est à dire sans appel récursif) pour $A(1,p)$, $A(2,p)$, $A(3,p)$ et $A(4,p)$.

Exercice 4 On dispose d'un stock illimité de pièces et de billets de c_1, c_2, …, c_p euros, et on souhaite dénombrer le nombre de manières possibles d'obtenir n euros avec ces pièces. L'objectif de cet exercice est de définir une fonction **compte**, de type *int –> int list –> int*, effectuant ce calcul.
Par exemple, pour connaitre le nombre de décompositions de 50€ à l'aide de pièces et de billets de 1€, 2€, 5€, 10€ et 20€, on écrira :

```
# compte 50 [1; 2; 5; 10; 20] ;;
- : int = 450
```

On dispose donc de 450 manières différentes de le faire.
Commencez par répondre aux deux questions suivantes : combien y-a-t-il de décompositions de n n'utilisant pas la pièce c_1 ? et au moins une fois ? et en déduire un algorithme récursif répondant au problème.
Adaptez ensuite votre réponse pour produire l'affichage de toutes les décompositions possibles.

4.2 Puzzles récursifs

Exercice 5 Les tours de Hanoï est un puzzle inventé par le mathématicien français Édouard Lucas : il est constitué de trois tiges sur lesquelles peuvent être enfilés n disques de diamètres différents. Au début du jeu, ces disques sont tous enfilés sur la même tige, du plus grand au plus petit, et le but du jeu et de déplacer tous ces disques sur une autre tige en respectant les règles suivantes :

- un seul disque peut être déplacé à la fois ;
- on ne peut jamais poser un disque sur un disque de diamètre inférieur.

Le puzzle dans sa configuration initiale

a) Écrire une fonction récursive donnant une solution de ce puzzle. On numérotera les tiges de 1 à 3 et on supposera que la tige 1 est la tige de départ et la tige 3, la tige d'arrivée. Par exemple, la résolution du problème à 3 disques affichera :

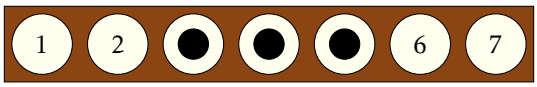
```
# hanoi 3 ;;
déplacer un disque de 1 vers 3
déplacer un disque de 1 vers 2
déplacer un disque de 3 vers 2
déplacer un disque de 1 vers 3
déplacer un disque de 2 vers 1
déplacer un disque de 2 vers 3
déplacer un disque de 1 vers 3
- : unit = ()
```

b) Déterminer le nombre minimal de déplacements nécessaires pour résoudre ce problème.

c) Résoudre le problème en s'interdisant tout mouvement entre les tiges 1 et 3 (tous les mouvements doivent donc aboutir ou débuter par la tige 2), et déterminer le nombre de mouvements nécessaires dans ce cas.

Exercice 6 Le jeu du baguenaudier est un puzzle constitué d'anneaux enchevêtrés dans une navette, le but étant de libérer celle-ci du système des anneaux. Il peut être modélisé de la manière suivante : on dispose d'une réglette contenant n cases numérotées de 1 à n et sur chacune desquelles est disposé un pion. Le but du jeu est d'ôter ces n pions en respectant les règles suivantes :

- il n'y a jamais plus d'un pion par case ;
- on peut à tout moment poser (s'il n'y en a pas) ou enlever (s'il y en a un) un pion sur la case 1 ;
- on peut poser ou enlever un pion sur la case $j \in [\![2, n]\!]$ s'il y a un pion sur la case $j-1$ et aucun sur les cases précédentes.



Dans cette configuration, on peut poser un pion dans la case 1 ou ôter celui de la case 4.

a) Écrire une solution récursive donnant une solution de ce puzzle.

b) Combien de mouvements nécessite la résolution de ce problème ?

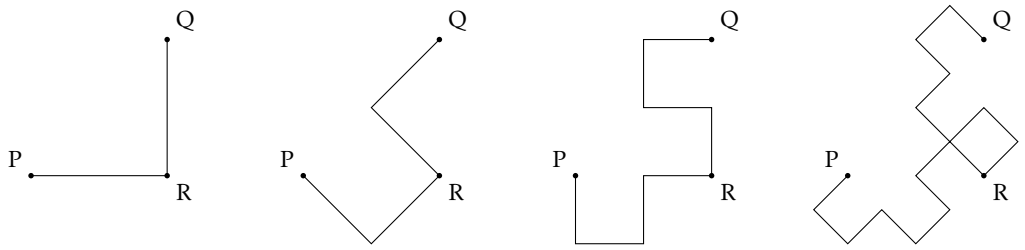
4.3 Figure géométriques récursives

Les procédures graphiques ne sont pas accessibles directement lorsqu'on utilise Caml, elles se trouvent dans le module **graphics** de la bibliothèque standard. Avant d'utiliser les fonctions dont la description suit, il faudra donc appliquer l'instruction : `#open "graphics"`.

On ouvre une nouvelle fenêtre graphique à l'aide de l'instruction `open_graph ""`.
Les points de cette fenêtre sont repérés par des coordonnées entières (x, y) où $0 \leqslant x \leqslant x_{\max}$ et $0 \leqslant y \leqslant y_{\max}$, le point de coordonnées $(0, 0)$ se trouvant en bas à gauche de l'écran. Les primitives `size_x ( )` et `size_y ( )` donnent respectivement les valeurs de $x_{\max} + 1$ et $y_{\max} + 1$ (ces valeurs dépendent de votre ordinateur).

À l'ouverture de la fenêtre graphique, le point courant se trouve en $(0, 0)$. la primitive `moveto x y` lève le crayon et le déplace au point de coordonnées (x, y) ; la primitive `lineto x y` déplace le crayon au point de coordonnées (x, y) en traçant un segment de droite.
Enfin, `clear_graph ( )` permet d'effacer le contenu de la fenêtre graphique.

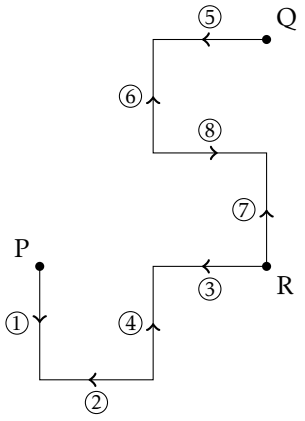
Exercice 7 On considère deux points distincts P et Q du plan. La courbe du dragon d'ordre 0 entre P et Q est le segment [P, Q]. Si $n \geqslant 1$, la courbe du dragon d'ordre n entre P et Q est obtenue en déterminant le point R tel que PRQ soit un triangle isocèle direct rectangle en R, et en traçant les courbes du dragon d'ordre $(n-1)$ entre P et R et entre Q et R.



Les courbes du dragon d'ordres 1, 2, 3, 4

a) Définir une procédure récursive traçant la courbe du dragon d'ordre n entre P et Q.

Le défaut de la fonction précédente est de provoquer un tracé fort discontinu car les segments ne sont pas tracés dans un ordre naturel :



Le sens et l'ordre dans lesquels sont tracés les segments pour $n = 3$

b) Réécrire cette fonction en s'imposant en plus de ne jamais lever le crayon lors du tracé, c'est à dire en n'utilisant plus la fonction `moveto`.

Exercice 8 Apparu en 1967, le langage de programmation Logo est surtout connu pour sa « tortue » graphique, vision imagée d'un robot qui se déplacerait sur l'écran en laissant derrière lui une trace de son passage. Nous allons en donner une version simplifiée : notre tortue sera définie par sa position (x,y) et sa direction (l'un des quatre points cardinaux). Ceci nous conduit à définir les types :

```
type direction = N | W | S | E ;;
type état = {mutable X : int; mutable Y : int; mutable Dir : direction} ;;
```

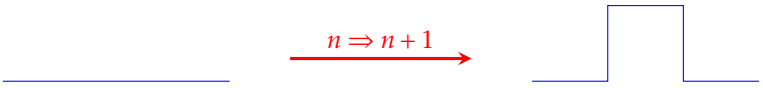
et à définir une tortue graphique :

```
let tortue = {X = 0; Y = 0; Dir = N} ;;
```

Nous n'allons lui autoriser que deux types de mouvements : tourner à gauche ou à droite d'un quart de tour, et avancer d'une certaine distance. On définit donc :

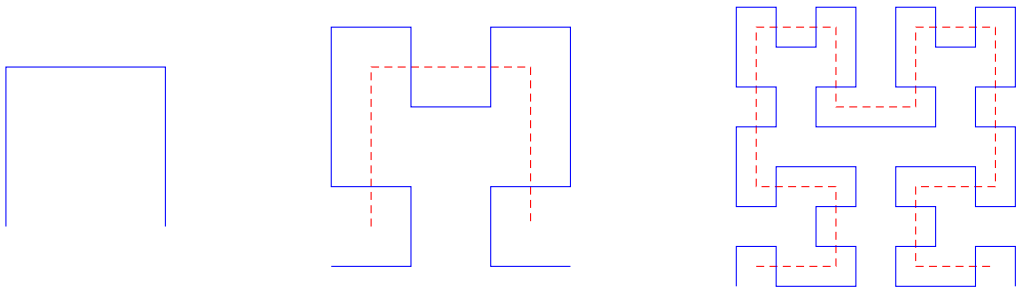
```
let gauche () = match tortue.Dir with
  | N -> tortue.Dir <- W  | S -> tortue.Dir <- E
  | W -> tortue.Dir <- S  | E -> tortue.Dir <- N ;;

let droite () = match tortue.Dir with
  | N -> tortue.Dir <- E  | S -> tortue.Dir <- W
  | W -> tortue.Dir <- N  | E -> tortue.Dir <- S ;;

let avance p = let (x,y) = match tortue.Dir with
        | N -> (tortue.X,tortue.Y+p) | S -> (tortue.X,tortue.Y-p)
        | W -> (tortue.X-p,tortue.Y) | E -> (tortue.X+p,tortue.Y)
  in lineto x y ; tortue.X <- x ; tortue.Y <- y ;;
```

a) La courbe quadratique de von Koch à l'ordre n peut être définie récursivement de la façon suivante :
   - la courbe d'ordre 0 est un segment de droite ;
   - pour obtenir la courbe d'ordre $n+1$, chaque segment de la courbe d'ordre n subit la transformation :



   Définir une fonction récursive permettant le tracé de la n^e courbe de von Koch par la tortue.

b) Voici les trois premières courbes de Hilbert. Devinez-en les règles de construction, puis écrire une fonction permettant le tracé de la n^e courbe de Hilbert par la tortue.



   On a représenté en pointillés la courbe d'ordre précédent.