# Algorithmes récursifs

## 1 Raisonnement par récurrence et par induction structurelle

### 1.1 Principe de récurrence

Soit $P(n)$ une propriété dépendant de l'entier naturel $n$ prenant pour chaque $n$ la valeur V ou F.
Le principe de récurrence est le suivant :

$$P(0) \text{ et } (\forall n \in \mathbb{N},\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n \in \mathbb{N},\ P(n)\,.$$

On peut appliquer ce principe à partir d'un entier $n_0$ au lieu de 0.
Exemples d'utilisation :

- On veut montrer que : $\forall n \geqslant 4,\ 2^n \geqslant n^2$.
  C'est vrai pour $n = 4$.
  Si $2^n \geqslant n^2$ avec $n \geqslant 4$, alors $2^{n+1} \geqslant 2n^2 \geqslant (n+1)^2$ car $n^2 - 2n = n(n-2) \geqslant 1$.
- On veut montrer que $\forall n \in \mathbb{N},\ \displaystyle\sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2$.
  C'est vrai pour $n = 0$ (et $n = 1$).
  Si c'est vrai au rang $n$, alors $\displaystyle\sum_{k=1}^{n+1} k^3 = \left(\frac{n(n+1)}{2}\right)^2 + (n+1)^3 = \left(\frac{(n+1)(n+2)}{2}\right)^2$.

### 1.2 Induction dans un ensemble bien ordonné

**Définition.** Une relation d'ordre est une relation binaire réflexive, transitive et antisymétrique.

Si $\preceq$ est une relation d'ordre, on note $\prec$ l'ordre strict associé.

**Définition.** Soit $E$ un ensemble muni d'une relation d'ordre $\preceq$. Un élément $a \in E$ est un minimum si et seulement si $\forall x \in E,\ a \preceq x$.

**Proposition.** *Le minimum, lorsqu'il existe, est unique.*

**Définition.** Une relation d'ordre sur un ensemble $E$ est totale si et seulement si deux éléments de $E$ sont toujours comparables.

**Définition.** Une relation d'ordre sur un ensemble $E$ est un bon ordre si et seulement si toute partie non vide de $E$ possède un élément minimum. On dit alors que $E$ est bien ordonné.

**Remarque.** Un bon ordre est un ordre total. (prendre une partie à deux éléments)

**Exemples.** La relation $\leqslant$ est un bon ordre sur $\mathbb{N}$, mais pas sur $\mathbb{Z}$ (pas de minimum).
$\mathbb{N}^2$ muni de l'ordre lexicographique $(a,b) \preceq (c,d) \iff (a < c)$ ou $(a = c$ et $b \leqslant d)$ est bien ordonné.
Dans $\mathbb{Z}$, la relation définie par $x \preceq y \iff |x| < |y|$ ou $(x = -y$ et $x \leqslant 0)$ ou $x = y$ est un bon ordre.

**Principe d'induction structurelle.**
Soit $E$ un ensemble bien ordonné, soit $a = \min E$. Soit $P$ un prédicat sur $E$ (c'est-à-dire une application de $E$ dans $\{V, F\}$). Le principe est le suivant :
Si $P(a)$ et $\forall\, x \in E,\ (\forall y \prec x,\ P(y)) \Rightarrow P(x)$, alors $\forall x \in E,\ P(x)$.

*preuve.* Soit $A$ l'ensemble des $x$ ne vérifiant pas $P$ (i.e tels que $P(x)$ est fausse). Si $A$ est non vide, il admet un plus petit élément $b$. Alors pour tout $x \prec b$, $P(x)$ est vraie, donc par hypothèse, $P(b)$ est vraie, ce qui est contradictoire. •

**Un peu de logique**.

Le théorème de Zermelo affirme que tout ensemble peut être bien ordonné.
Un ensemble est dit inductif lorsque toute partie non vide totalement ordonnée possède un majorant.
Le théorème de Zorn affirme que tout ensemble inductif possède un élément maximal.
L'axiome du choix affirme que le produit d'une famille d'ensembles non vides est non vide.
Le théorème de Zermelo est équivalent au théorème de Zorn ou à l'axiome du choix, mais aucun de ces résultats ne peut être démontré. On doit en ajouter un à l'axiomatique pour obtenir les autres.

$\mathbb{Q}$ peut être facilement muni d'un bon ordre (basé sur l'ordre lexicographique de $\mathbb{N}^2$). On peut munir $\mathbb{R}$ d'un bon ordre mais on ne peut pas l'expliciter.

# 2 Récursivité

## 2.1 Principe

Un algorithme récursif est un algorithme qui s'appelle lui-même. Il comporte deux parties : la description des cas de base et le (ou les) appels récursifs.
L'intérêt de la programmation récursive est de produire des programmes rapides et élégants.

### 2.1.1 Exemple de la factorielle

$$\texttt{facto}(n) = \begin{cases} 1 \text{ si } n = 0 \\ n \times \texttt{facto}(n-1) \text{ si } n \geqslant 1 \end{cases}$$

**Spécification :** $n$ est un entier naturel.
**Fonctionnement de l'algorithme :** le calcul de `facto`$(n)$ se ramène à celui de `facto`$(n-1)$, ce qui permet de descendre ainsi jusqu'au calcul de `facto(0)`, qui est donné par la fonction, puis de remonter en faisant les calculs jusqu'à la valeur de `facto`$(n)$.
**Syntaxe Caml** : `let rec facto n = if n = 0 then 1 else n * facto (n-1) ;;`
Si on enlevait le mot `rec`, le code ne marcherait pas car tout objet Caml doit être défini avant d'être utilisé. Le mot-clé `rec` permet de lever l'ambiguïté.

**Comment suivre en Caml les appels récursifs :**
La fonction `trace` permet de tracer la fonction, c'est-à-dire d'imprimer la succession des appels récursifs.
La syntaxe pour tracer une fonction $f$ est `# trace f ;;`
La syntaxe pour supprimer le traçage est `# untrace f ;;`

```
#facto 3 ;;
facto <-- 3
facto <-- 2
facto <-- 1
facto <-- 0
facto --> 1
facto --> 1
facto --> 2
facto --> 6
- : int = 6
```

### 2.1.2 Ecriture en pseudo-langage

Un algorithme récursif (à un seul appel récursif) se présente sous la forme suivante :

$$P(x) = \begin{cases} I_1 \\ \text{si } C(x) \quad \text{alors } I_2 \\ \text{sinon} \quad\quad I_3 \ ; \ P(\phi(x)) \\ \text{fin si} \end{cases}$$

où $I_1$, $I_2$, $I_3$ sont des blocs d'instructions.
A chaque appel de l'algorithme, $I_1$ est exécuté. Le bloc $I_2$ est exécuté lorsque $x$ se trouve dans un cas de base (déterminé par la condition $C(x)$). Sinon, le bloc $I_3$ est exécuté avant de passer à l'appel récursif, dans lequel on applique l'algorithme à la valeur $\phi(x)$.

**Fonctionnement d'un programme récursif :**

A tout programme récursif est associé une pile, c'est-à-dire une zone mémoire réservée (dont la taille peut être paramétrée par le programmeur). Son rôle est de stocker l'environnement lors de chaque appel récursif, c'est-à-dire les variables locales, les paramètres et les adresses de retour des fonctions en cas d'exécution.
A chaque appel récursif, on empile le nouvel environnement.
A chaque retour de fonction, on dépile le dernier environnement, ce qui permet de revenir au point d'appel.
Un programme récursif comporte donc une phase de descente (empilement des appels récursifs) et une phase de remontée pour renvoyer le résultat.
La taille de la pile est bornée au départ, donc un programme nécessitant trop d'appels récursifs échouera.

Exemple : `let rec f x = f(x+1) ;;` Le calcul de `f 0` tourne indéfiniment et fait planter Caml.

## 2.2 Description d'une fonction récursive comportant un seul appel récursif

**Définition.** Une fonction $f : E \to F$ est dite récursive lorsqu'il existe une partie $A \subset E$ (ensemble des cas de base), une fonction $f_0 : A \to F$ (valeur de $f$ sur les cas de base), une fonction $\varphi : E \to E$ et une fonction $h : E \times F \to F$ telles que : $\begin{cases} \forall x \in A,\ f(x) = f_0(x) \\ \forall x \in E \setminus A,\ f(x) = h(x, f(\varphi(x))) \end{cases}$

On dit que la fonction récursive **termine** lorsque, pour tout $x \in E$, le calcul de $f(x)$ aboutit au bout d'un nombre fini d'opérations, c'est-à-dire d'un nombre fini d'appels récursifs.
Dans l'étude d'une fonction récursive, le problème principal est sa terminaison.
On devra parfois réduire l'ensemble $E$ pour que la fonction termine (spécification de la fonction).

**Exemples** :

- La fonction `facto` termine dans $\mathbb{N}$ mais pas dans $\mathbb{Z}$.

- Soit $f$ la fonction suivante : $f(x) = \begin{cases} 0 & \text{si } x = 10, \\ 1 + f(x+1) & \text{si } x \neq 10. \end{cases}$

Le calcul de $f(x)$ termine uniquement pour $x \leqslant 10$.

## 2.3 Terminaison d'un algorithme récursif

**Théorème.** *Soit $E$ un ensemble bien ordonné.*
*Soit $f$ une fonction récursive définie sur $E$ par* $f(x) = \begin{cases} f_0(x) & \text{si } x \in A, \\ h(x, f(\varphi(x))) & \text{sinon.} \end{cases}$
*On suppose que $\forall x \in E \setminus A,\ \varphi(x) \prec x$. Alors la fonction $f$ termine toujours.*

*preuve.* Notons $B = \{x \in E \mid \text{le calcul de } f(x) \text{ ne termine pas}\}$ et supposons $B$ non vide.
Soit $x_0 = \min B$. Alors $x_0 \notin A$, d'où $f(x_0) = h(x, f(\varphi(x_0)))$ avec $\varphi(x_0) \prec x_0$. On en déduit que $\varphi(x_0) \notin B$, donc le calcul de $f(\varphi(x_0))$ termine et donc celui de $f(x_0)$ aussi : contradiction, donc $B$ est l'ensemble vide.
*Remarque : cela revient à montrer la terminaison par induction structurelle.* •

**Exemples** :

- Factorielle : le calcul de $n!$ se termine au bout de $n$ étapes car le calcul de $n!$ renvoie à celui de $(n-1)!$ et le cas de base est $n = 0$. La preuve est immédiate par récurrence.

- Algorithme de Collatz :
– $f(1) = 1$.
– $f(x) = \begin{cases} f(\frac{x}{2}) \text{ si } x \text{ est pair}, \\ f(3x+1) \text{ si } x \text{ est impair} \end{cases}$

On ne connait aucune valeur de $x$ pour laquelle le calcul de $f(x)$ ne termine pas, mais on ne peut pas démontrer que l'algorithme termine.

## 2.4 Correction d'un algorithme récursif

Un algorithme doit toujours se prouver, à plus forte raison s'il s'agit d'un algorithme récursif. Classiquement, si l'algorithme prend son argument $x$ dans un ensemble bien ordonné $E$, on prouve par induction structurelle que $P(x)$ renvoie la valeur désirée. Souvent, la preuve s'effectuera par récurrence.

**Exemples.**

**Factorielle** : on suppose que `facto(k)` renvoie $k!$ pour $k \leqslant n-1$.
Alors `facto(n) = n` $\times$ `facto(n-1)` renvoie $n!$
Comme `facto(0)=1`, l'algorithme est prouvé par récurrence sur $n$.

**Maximum d'une partie de tableau** :

- **Un par un**

```
let rec maxi t i j =
 if i = j then t.(i)
 else max (t.(i)) (maxi t (i+1) j) ;;
```

- **Par dichotomie**

```
let rec maxi t i j =
 if i=j then t.(i)
 else max (maxi t i ((i+j)/2)) (maxi t ((i+j)/2+1) j) ;;
```

Dessiner l'arbre des appels récursifs. Tracer les appels récursifs.

La terminaison et la correction se prouvent par récurrence sur $j-i$.

Les deux algorithmes sont de performance équivalente car chaque élément du tableau est lu une et une seule fois, mais pas dans le même ordre.

## 2.5 Récursivité multiple

Il s'agit d'une fonction ou d'un algorithme comportant plusieurs appels récursifs.
Avec deux appels récursifs, la fonction s'écrira $f(x) = \begin{cases} f_0(x) & \text{si } x \in A, \\ h(x, f(\varphi(x)), f(\psi(x))) & \text{sinon.} \end{cases}$
Comme précédemment, on démontre que si $E$ est bien ordonné et si $\varphi(x) \prec x$ et $\psi(x) \prec x$ pour tout $x \in E \setminus A$, alors la fonction $f$ termine.
**Remarque** : il faudra toujours réfléchir à l'efficacité d'un algorithme comportant plusieurs appels récursifs (voir plus loin).

## 2.6 Récursivité terminale

Une fonction est dite récursive terminale si l'appel récursif est la dernière instruction à être évaluée.
Une fonction récursive terminale $f$ se présente sous la forme suivante :
$f(x) = \begin{cases} f_0(x) & \text{si } x \in A \\ f(\varphi(x)) & \text{sinon} \end{cases}$ où $\varphi(x) \prec x$ pour tout $x \in E \setminus A$.

**Exemple.** Factorielle récursive terminale :

```
let facto n =
 let rec facto_aux n k = if n = 0 then k else facto_aux (n-1) (n*k)
 in facto_aux n 1 ;;
```

Dans un algorithme récursif terminal, tous les calculs se font à la descente, la phase de remontée est inutile. Il n'y a rien à retenir dans la pile d'exécution, il suffit juste de conserver en mémoire le dernier appel récursif à chaque étape de l'exécution.
Un tel algorithme peut être facilement transformé en algorithme itératif.

Algorithme récursif terminal :

```
let rec f cond f0 phi x = if cond x then f0 x else f cond f0 phi (phi x) ;;
```

Typage de la fonction `f : ('a -> bool) -> ('a -> 'b) -> ('a -> 'a) -> 'a -> 'b = <fun>`

Algorithme itératif associé :

```
let iter_f cond f0 phi x =
 let y = ref x in
 while not (cond !y) do y := phi !y done;
 f0 !y ;;
```

Généralement, un algorithme récursif peut être transformé en récursif terminal en utilisant une fonction auxiliaire récursive terminale prenant davantage de paramètres. Voir l'exemple de la factorielle.

# 3 Algorithmes récursifs classiques

## 3.1 Calcul du PGCD

Le filtrage permet d'écrire un code élégant. Spécification : `a` et `b` sont des entiers naturels.

```
let rec pgcd a b = match (a,b) with
 | (_,0) -> a
 | (0,_) -> b
 | _ when a < b -> pgcd b a
 | _ -> pgcd (a-b) b ;;
```

La terminaison et la correction se démontrent par récurrence sur $|b-a|$ ou par induction suivant l'ordre lexicographique.

## 3.2 Exponentiation rapide

On écrit une fonction `puiss` telle que `puiss(x,n)` calcule $x^n$.

Méthode naïve :

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

La terminaison et la correction se démontrent par récurrence sur $n$.
Le nombre d'appels récursifs pour calculer $x^n$ est égal à $n$.

Exponentiation rapide : l'algorithme se base sur la fonction récursive suivante : $x^0 = 1$ et si $n$ est pair, $x^n = x^{n/2} \times x^{n/2}$ ; sinon, $x^n = x \times x^{(n-1)/2} \times x^{(n-1)/2}$.

```
let rec puiss_rap x = function
   0 -> 1
 | n -> let y = puiss_rap x (n/2) and z = (if n mod 2 = 0 then 1 else x) in z * y * y ;;
```

La terminaison et la correction se démontrent par récurrence sur $n$.
Le nombre d'appels récursifs pour calculer $x^n$ est égal à $\lfloor \lg n \rfloor$.

**Remarque importante :** dans le code, il faut veiller à demander un seul appel récursif pour ne pas accroitre inutilement la complexité. Le bout de code `puiss_rap x (n/2) * puiss_rap x (n/2)` est à proscrire.

## 3.3 Tours de Hanoï

Ce jeu a été inventé par Edouard Lucas en 1883. On dispose de 3 tiges de bois sur lesquelles on peut placer des disques percés en leur milieu. Initialement, on place $n$ disques de taille strictement décroissante sur la tige numéro 1, les deux autres étant vides. Le jeu consiste à faire passer ces $n$ disques sur la tige numéro 3, en déplaçant à chaque étape un disque d'une tige sur une autre, avec la règle qu'on ne peut placer sur un disque qu'un disque de diamètre plus petit. On souhaite écrire une fonction décrivant la séquence des mouvements permettant de déplacer $n$ disques de la tige numéro 1 à la tige numéro 3.
On utilise une fonction `deplace : string -> string -> unit` affichant à l'écran un déplacement et une fonction récursive `hanoi : int -> string -> string -> string -> unit` affichant à l'écran la succession des déplacements.
Pour déplacer les $n$ disques de $A$ vers $C$, on déplace récursivement les $n-1$ premiers de $A$ vers $B$ et le dernier vers $C$, puis récursivement les $n-1$ situés sur $B$ vers $C$.

```
let deplace p q =
 print_newline ();
 print_string "le disque au sommet de ";
 print_string p;
 print_string " est placé sur ";
 print_string q ;;
let rec hanoi n a b c =
 if n > 0 then begin
   hanoi (n-1) a c b;
   deplace a c;
   hanoi (n-1) b a c end ;;
```

La terminaison et la correction s'obtiennent par récurrence sur $n$. Plus précisément, on démontre par récurrence que le nombre de déplacements pour $n$ disques est égal à $2^n - 1$.

# 4 Récursivité mutuelle

On s'intéresse à un algorithme comportant deux fonctions $P$ et $Q$ telles que $P(x)$ fasse appel à $Q(y)$ et $Q(x)$ fasse appel à $P(z)$. Les deux fonctions $P$ et $Q$ sont dites mutuellement récursives.

La terminaison d'un tel algorithme suppose qu'on dispose des cas de base et que lors des appels récursifs, on ait toujours $y \prec x$ et $z \prec x$.

Exemple élémentaire :

$$\text{pair}\,(x) = \begin{cases} V & \text{si } x = 0 \\ \text{impair}\,(x-1) & \text{sinon} \end{cases} \quad \text{et impair}\,(x) = \begin{cases} F & \text{si } x = 0 \\ \text{pair}\,(x-1) & \text{sinon} \end{cases}$$

Syntaxe Caml : `let rec f = ...`
`and g = ...`

# 5 Structure de liste en OCaml

## 5.1 Définition

La notion de liste s'est développée au moment de la création du langage LISP (list processing) - l'ancêtre des langages fonctionnels - crée par John MacCarthy en 1960.

**Définition informelle.** Une liste en Caml est une suite finie ordonnée de valeurs du même type. Contrairement aux tableaux, on a accès dans une liste à son premier élément, puis séquentiellement aux éléments suivants. Pour accéder au $k$-ième élément, il faut passer par les $k-1$ précédents.

**Définition informatique.** Une liste est une structure de données récursive : une liste est soit vide, soit donnée par un couple $(a, l)$ où $a$ est sa tête et $l$ sa queue, qui est une liste d'éléments du même type que $a$.

On pourrait la définir en OCaml à l'aide du type récursif suivant :

```
type 'a list = Null | Cons of 'a * 'a list ;;
```

## 5.2 Module List

La structure de liste est implantée en OCaml au sein du module `List`. Il s'agit d'une structure polymorphe. Une liste d'éléments de type `'a` aura pour typage `'a list`.

La syntaxe Caml d'une liste est `[a_1; a_2; ...; a_n]`

**Une liste OCaml est non mutable.**

Le module `List` donne accès à un certain nombre de fonctions dédiées.

La liste vide est notée `[ ]`

La tête de la liste `l` est `List.hd l`

La queue de la liste `l` est `List.tl l`

L'ajout d'un élément se fait en tête de liste : `a::l`

La concaténation de deux listes `l1` et `l2` est `l1 @ l2` ou `List.append l1 l2`

## 5.3 Fonctions classiques

La structure de liste étant récursive, on s'efforcera d'écrire des fonctions récursives lorsqu'on travaille sur des listes.
On favorisera la programmation par filtrage :

```
let rec f = function
     [] -> ...
 | t::q -> ...

let rec f = fun l -> match l with
     [] -> ...
 | (t::q) -> ...

let rec f l = match l with
      [] -> ...
  | t::q -> ...
```

**Appartenance.**
`appartient : 'a -> 'a list -> bool`

```
let rec appartient x = function  [] -> false
 | (t::q) -> (x = t) || (appartient x q) ;;
```

**n-ième élément.**
`nth : int -> 'a list -> 'a`

```
let rec nth n = function
   [] -> failwith "liste trop courte"
 | t::q when n<1 -> failwith "numéro invalide"
 | t::q when n=1 -> t
 | t::q -> nth (n-1) q ;;
```

**Intervalle d'entiers.**
`intervalle : int -> int -> int list`

```
let rec intervalle a b =
 if a > b then [] else a::(intervalle (a+1) b) ;;
```

**Appliquer une fonction à tous les termes d'une liste.**
`map : ('a -> 'b) -> 'a list -> 'b list`

```
let rec map f = function [] -> []
 | t::q -> (f t)::(map f q) ;;
```

## 5.4 Fonctions du module List

| | | |
|---|---|---|
| `length` | `List.length l` | retourne la longueur de la liste. |
| `mem` | `List.mem x l` | teste l'appartenance de `x` à la liste `l`. |
| `rev` | `List.rev l` | renverse la liste `l`. |
| `nth` | `List.nth l n` | renvoie le $n$-ième élément de la liste `l`. |
| `map` | `List.map f [a1;...;an]` | renvoie la liste `[f a1;...;f an]` |
| `iter` | `List.iter f [a1;...;an]` | applique $f$ (type unit) aux éléments de la liste. |
| `fold_left` | `List.fold_left f a [b1;...,bn]` | renvoie `f (... (f (f a b1) b2) ...) bn` |
| `fold_right` | `List.fold_right f [a1; ...; an] b` | renvoie `f a1 (f a2 ( ... (f an b) ...))` |
| `for_all` | `List.for_all p [a1;...;an]` | renvoie `p a1 && p a2 &&` ... `&& p an` |
| `exists` | `List.exists p [a1;...;an]` | renvoie `p a1 \|\| p a2 \|\| ... \|\| p an` |
| `find` | `List.find p l` | retourne le premier élément de `l` satisfaisant la condition `p`. |
| `filter` | `List.filter p l` | retourne la liste des éléments de `l` satisfaisant la condition `p`. |
| `assoc` | `List.assoc a l` | retourne le premier élément `b` tel que `(a,b)` appartient à `l`. |
| `split` | `List.split [(a1,b1);...;(an,bn)]` | retourne le couple `[a1;...;an],[b1;...;bn]` |
| `sort` | `List.sort ord l` | trie la liste selon l'ordre passé en paramètre. |
| `merge` | `List.merge ord l1 l2` | fusionne `l1` et `l2` au départ triées par ordre croissant. |

**Exemples** : somme des éléments d'une liste, maximum des éléments d'une liste.

# Annexe 1 : graphisme

Le module `Graphics` permet d'accéder à la fenêtre graphique et d'utiliser des fonctions dédiées au graphisme.
Ce module n'est pas chargé au lancement de OCaml donc doit être chargé par l'instruction `# load "graphics.cma"` puis ouvert par l'instruction `open Graphics`
La fenêtre graphique est repérée par des entiers selon les axes de coordonnées `x` et `y`.
Un type `color` est défini (codé par un entier à partir des couleurs primaires rgb).
Certaines couleurs sont prédéfinies (`black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`).

Les fonctions principales du module `Graphics` sont les suivantes :

| | | |
|---|---|---|
| `open_graph` | `Graphics.open_graph ""` | ouvre la fenêtre graphique. |
| `size_x` | `Graphics.size_ x ()` | retourne la dimension horizontale. |
| `size_y` | `Graphics.size_ y ()` | retourne la dimension verticale. |
| `close_graph` | `Graphics.close_graph ()` | ferme la fenêtre graphique. |
| `clear_graph` | `Graphics.clear_graph ()` | efface la fenêtre graphique. |
| `current_point` | `Graphics.current_point ()` | retourne les coordonnées du point courant. |
| `plot` | `Graphics.plot a b` | trace le point de coordonnées `(a,b)`. |
| `moveto` | `Graphics.moveto a b` | déplace le point courant vers le point `(a,b)`. |
| `lineto` | `Graphics.lineto a b` | trace un trait depuis le point courant vers le point `(a,b)`. |
| `set_color` | `Graphics.set_color c` | spécifie la couleur. |
| `rgb` | `Graphics.rgb r g b` | crée une couleur à partir de ses composantes rgb. |
| `draw_rect` | `Graphics.draw_rect x0 y0 larg haut` | trace un rectangle de largeur larg de hauteur haut dont le point inférieur gauche a pour coordonnées `(x0,y0)`. |
| `draw_circle` | `Graphics.draw_circle x0 y0 r` | trace un cercle de centre `(x0,y0)`de rayon `r`. |
| `fill_rect` | `Graphics.fill_rect x0 y0 larg haut` | trace un rectangle et remplit l'intérieur. |
| `fill_circle` | `Graphics.fill_circle x0 y0 r` | trace un cercle et remplit l'intérieur. |

Les 4 dernières fonctions n'affectent pas le point courant.

### Tracé d'un polygone

La fonction `trace` prend en argument la liste des points et trace le polygone.

```
# load "graphics.cma" ;;
open Graphics;;
Graphics.open_graph "";;
size_x();;
size_y();;
let trace polygone =
  let rec aux poly = match poly with
     [] -> ()
    | (a,b)::q -> lineto a b; aux q
  in let (a,b) = List.hd polygone in
  moveto a b; aux (List.tl polygone @ [(a,b)])
;;

let (ox,oy) = (200,200) ;;
let p = [(ox,oy);(ox+60,oy+60);(ox+120,oy+40);(ox+140,oy+30);(ox+160,oy-20);(ox+100,oy-50);(ox+40,oy-60
trace p ;;
```

### Utilisation de la ligne de commandes

Si on exécute le fichier en ligne de commandes, il faut enlever `#load "graphics.cma"` et ajouter à la fin une instruction permettant de conserver la fenêtre graphique ouverte dans l'attente de presser une touche quelconque. On pourra ajouter par exemple `ignore (read_key ())` ou `read_line ()`
Pour lancer le fichier : `ocaml graphics.cma polygone2.ml`
On peut aussi compiler le fichier pour en faire un exécutable :
`ocamlc graphics.cma polygone2.ml -o polygone`
On l'exécute par le script : `ocamlrun polygone`

**Exemple de programme graphique récursif**

On programme le flocon de Von Koch, exemple simple de courbe fractale.

```
# load "graphics.cma" ;;
open Graphics ;;

let pi = 4. *. atan 1. ;;

let flocon n =
  Graphics.clear_graph ();
  let (ox,oy) = (ref 200., ref 200.) in
  let rec cote angle longueur n =
    if n = 0 then
      begin
      Graphics.moveto (int_of_float !ox) (int_of_float !oy);
      ox := !ox +. longueur *. cos angle ;
      oy := !oy +. longueur *. sin angle ;
      Graphics.lineto (int_of_float !ox) (int_of_float !oy)
      end
    else begin
(* on subdivise le côté en 4 segments de longueur l/3 et on trace récursivement *)
      cote angle (longueur /. 3.) (n-1);
         cote (angle -. (pi/.3.)) (longueur /. 3.) (n-1);
      cote (angle +. (pi/.3.)) (longueur /. 3.) (n-1);
      cote angle (longueur /. 3.) (n-1)
      end
  in
  cote 0.0 200. n ;
  cote ((2.*. pi)/. 3.) 200. n ;
  cote (-.(2.*. pi)/. 3.) 200. n
   ;;
Graphics.open_graph "" ;;
Graphics.size_x(), Graphics.size_y() ;;
flocon 5 ;;
flocon 8 ;;
```

# Annexe 2 : quelques fonctions récursives connues

**Fonction 91 de Mac Carthy**

$$f(n) = \begin{cases} n-10 \text{ si } n > 100, \\ f(f(n+11)) \text{ sinon.} \end{cases}$$

Pour $n \leqslant 100$, $f(n) = 91$.

**Suite Q de D.Hofstadter**

$Q(0) = Q(1) = 1. \qquad \forall n \geqslant 2,\ Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2))$.

Cette suite est une déformation de la suite de Fibonacci. La terminaison de cette fonction est un problème ouvert. Elle revient à montrer que $Q(n) \leqslant n$ pour tout $n$.

Le calcul de $Q(n)$ devient très lent à partir de $Q(38)$.

**Fonction de Morris**

$M(0,p) = 1. \qquad M(n,p) = M(n-1, M(n,p))$ si $n \geqslant 1$.

Cette fonction ne termine pas.

**Fonction de Ackermann**

On pose $A(m,n) = \begin{cases} n+1 \text{ si } m = 0, \\ A(m-1,1) \text{ si } n = 0 \text{ et } m \geqslant 1, \\ A(m-1, A(m, n-1)) \text{ sinon.} \end{cases}$

On constate expérimentalement qu'il est impossible de calculer $A(m,n)$ pour $m \geqslant 4$ dès que $n \geqslant 2$. $A(1,n) = n+2$ par récurrence sur $n$. $A(2,n) = 2n+3$ par récurrence sur $n$. $A(3,n) = 2^{n+3} - 3$ par récurrence sur $n$. $A(4,n) = \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3 \text{ chiffres}} - 3$.

La fonction d'Ackermann croît extrêmement rapidement : $A(4,2)$ a déjà 19729 chiffres, et représente bien plus que le nombre d'atomes estimé dans l'univers. Cette extrême croissance peut être exploitée pour montrer que la fonction $f$ définie par $f(n) = A(n,n)$ croît plus rapidement que n'importe quelle fonction récursive primitive et ainsi que $A$ n'en est pas une.

**Fonction de Takeuchi**

$$\tau(x,y,z) = \begin{cases} \tau(\tau(x-1,y,z), \tau(y-1,z,x), \tau(z-1,x,y)) \text{ si } y < x, \\ y \text{ sinon.} \end{cases}$$

Cette fonction récursive termine toujours, mais le temps de calcul peut être très long si le compilateur n'est pas optimisé.

Elle admet une expression directe très simple.