TP Récursif
Fractales

Principe

Extrait Wikipedia (en)[1] :

> "A fractal is a mathematical set that has a fractal dimension that usually exceeds its topological dimension and may fall between the integers. Fractals are typically self-similar patterns, where self-similar means they are "the same from near as from far". Fractals may be exactly the same at every scale, or [...], they may be nearly the same at different scales. The definition of fractal goes beyond self-similarity per se to exclude trivial self-similarity and include the idea of a detailed pattern repeating itself."

Les fractales que nous allons voir sont définies par la limite d'un procédé récursif de fabrication : un segment (dans la première partie) est remplacé par un motif composé lui-même de plusieurs segments de tailles inférieures, ensuite chacun de ces segments est à nouveau remplacé par ce motif et ainsi de suite. Chaque itération supplémentaire correspond à une nouvelle courbe, de rang supérieur. La fractale correspond à l'ensemble des points contenus dans l'intersection de ces courbes. Les exercices de la deuxième partie sont basés sur le même principe, mais appliqué à des surfaces.

Bibliothèque graphique

Vous aurez besoin d'utiliser les fonctions de la bibliothèque graphique. [2]

Tout d'abord, il faut charger le module en ajoutant les lignes suivantes au début du fichier .ml :

— avec la version **4.02.3** (installation Windows suggérée sur le gitlab)

```
#load "graphics.cma" ;;      (* Load the library *)
```

— avec les **autres versions**

```
#use "topfind" ;;      (* Loads the findlib library *)
#require "graphics";; (* Loads the graphics library *)
```

Evaluez les instructions suivantes :

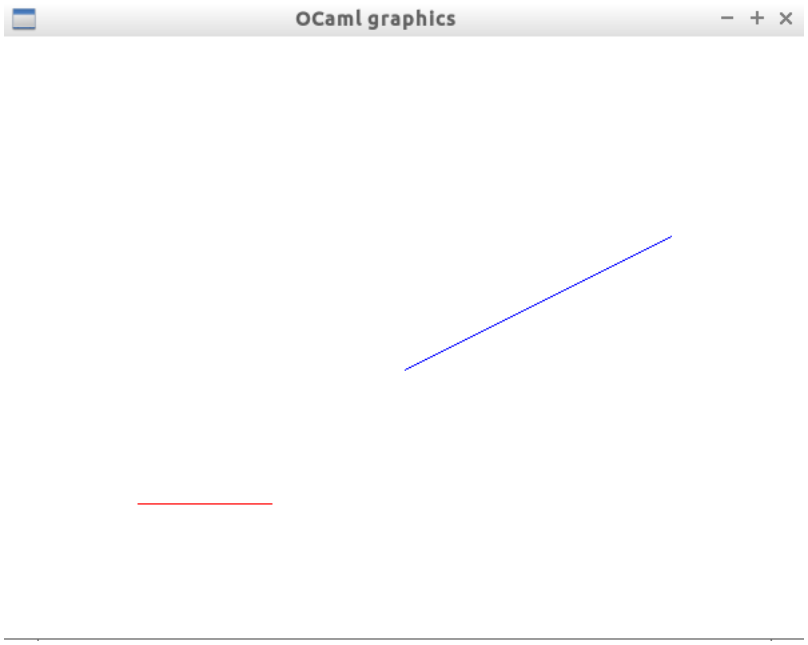
```
open Graphics ;;
open_graph " 1200x800" ;;

let test (x,y) (z,t) =
    clear_graph ();  (* Removes what has been drawn before *)
    set_color red;   (* Defines the current color as red *)
    moveto x y ;     (* Moves the current position to point (x,y) *)
    lineto z t ;     (* Draws a line from the current position to the point (z,t) *)
    set_color blue ; (* Defines the current color as blue *)
    rmoveto x y ;    (* Moves the current position wuth the vector (x,y) *)
    rlineto z t ;;   (* Draws the vector (z,t) from the current position and moves
   the current position to that point*)

test (50,50) (50,150) ;;
test (100,100) (200,100) ;;
```

Si les instructions se sont bien déroulées vous devriez voir :

1. http://en.wikipedia.org/wiki/Fractal
2. https://ocaml.github.io/graphics/graphics/Graphics/index.html



Certaines fonctions du module Graphics utilisent la notion de position courante. Par défaut la position courante est fixée à (0,0). Par exemple la fonction `moveto` déplace la position courante au point (x,y).

Quelques fonctions utiles (extraits du manuel) :

```
val clear_graph : unit -> unit
```

Erase the graphics window.

```
val moveto : x:int -> y:int -> unit
```

Position the current point.

```
val rmoveto : dx:int -> dy:int -> unit
```

`rmoveto dx dy` translates the current point by the given vector.

```
val lineto : x:int -> y:int -> unit
```

Draw a line with endpoints the current point and the given point, and move the current point to the given point.

```
val rlineto : dx:int -> dy:int -> unit
```

Draw a line with endpoints the current point and the current point translated of the given vector, and move the current point to this point.

```
val draw_circle : dx:int -> dy:int -> -> dr:int -> unit
```

Draws a circle with center x,y and radius r. The current point is unchanged.

```
val fill_circle : dx:int -> dy:int -> -> dr:int -> unit
```

Fill a circle with the current color. The parameters are the same as for draw_circle.

```
(* draw a line from point (x, y) to point (z, t) *)

let draw_line (x, y) (z, t) =
   moveto x y ;
   lineto z t ;;
```

1 Les courbes

Conseil pour tous les exercices qui suivent : commencer les tests avec des petites valeurs pour l'ordre de la courbe...

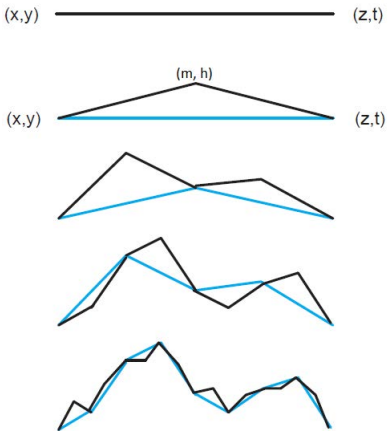
1.1 Mountain

On désire générer aléatoirement une "montagne" selon le principe suivant :

- **étape 0 :** On trace un segment entre 2 points quelconques.
- **étape 1 :** On calcule une nouvelle hauteur pour le milieu du segment (aléatoire [3]).
- **étape n :** On applique le même processus sur chacun des nouveaux segments de droite de l'étape précédente.

Méthode :
La "courbe" d'ordre n entre les points (x, y) et (z, t) est :

- $\boldsymbol{n = 0}$: le segment $[(x, y), (z, t)]$
- $\boldsymbol{n \neq 0}$: la courbe d'ordre $n - 1$ entre (x, y) et (m, h) suivie de la courbe d'ordre $n - 1$ entre $(\mathbf{m}, \mathbf{h})$ et (z, t), où $\mathbf{m}$ est le "milieu" de x et z et $\mathbf{h}$ une hauteur calculée aléatoirement.



Conseil :
Calculer la nouvelle hauteur en fonction des 2 points et éventuellement de n. On peut, par exemple, diminuer la différence de hauteur au fur et à mesure que les points se rapprochent...

Quelques exemples de mise à jour de la hauteur où `int(e)` donne un entier aléatoire entre 0 et $e - 1$:

- $h = (y + t)/2 + int(10 * n)$
- $h = (y + t)/2 + int(abs(z - x)/5 + 20)$

Écrire la fonction `mountain n (x,y) (z,t)` qui prend donc en paramètres l'ordre n et les deux points de départ (x, y) et (z, t). Les deux points de départ sont les bornes de la courbe.

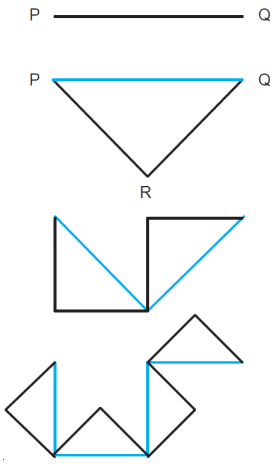
```
val mountain : n:int -> (x,y):int * int -> (z,t):int * int -> unit = <fun>
(* Application example *)
clear_graph ();
mountain 9 (10,200) (800,200) ;;
```

1.2 Dragon

On désire générer aléatoirement un "dragon" selon le principe suivant : partir d'un segment de base ; puis en suivant la courbe, remplacer chaque segment par deux segments à angle droit en effectuant une rotation de 45° alternativement à droite puis à gauche.

Méthode :
La "courbe" d'ordre n entre les points P et Q est :

- $\boldsymbol{n = 0}$: la courbe d'ordre 0 est un segment entre 2 points quelconques P et Q.
- $\boldsymbol{n \neq 0}$: la courbe d'ordre n est la courbe d'ordre $n - 1$ entre P et R suivie de la même courbe d'ordre $n - 1$ entre R et Q (à l'envers), où PRQ est le triangle isocèle rectangle en R, et R est à droite du segment PQ.



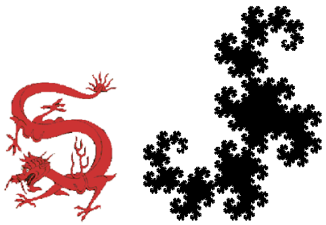
Conseil :
Si P et Q sont les points de coordonnées (x, y) et (z, t), les coordonnées (u, v) de R sont :

- $u = (x + z)/2 + (t - y)/2$
- $v = (y + t)/2 - (z - x)/2$

3. Vous pouvez utiliser les fonctions du "pseudo" générateur de nombres aléatoires : le module `Random` dont vous trouverez une description dans le manuel Caml . N'oubliez pas d'initialiser le moteur !

Écrire la fonction `dragon n (x,y) (z,t)` qui prend donc en paramètres l'ordre n et les deux points de départ (x, y) (P) et (z, t) (Q). Les deux points de départ sont les bornes du premier segment de la courbe.

```
val dragon : n:int -> (x,y):int * int -> (z,t):int *
int -> unit = <fun>
(* Application example to obtain a cute dragon *)
clear_graph ();
dragon 19 (150,150) (350,350) ;;
```



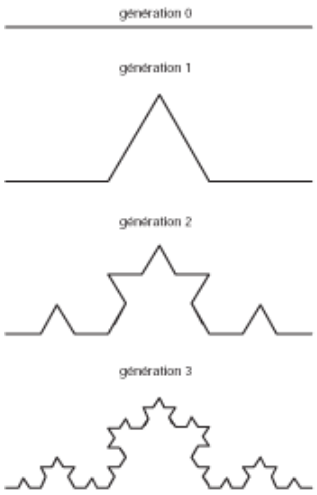
1.3 Koch snowflake

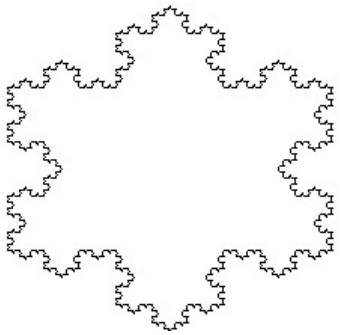
Le flocon de von Koch d'ordre n est défini par :

- $\boldsymbol{n = 0}$: un triangle équilatéral.
- $\boldsymbol{n = 1}$: un triangle équilatéral dont les côtés sont découpés en trois et sur chacun desquels s'appuie un autre triangle équilatéral au milieu.
- $\boldsymbol{n > 1}$: un flocon d'ordre n en appliquant la même opération sur chacun de ses cotés.

En terme de motifs, le flocon peut aussi s'expliquer ainsi : il est constitué de trois courbes de von Koch placées sur les côtés d'un triangle équilatéral.

La courbe de von Koch est un segment de longueur d remplacé par 4 segments de longueur $d/3$, l'angle des segments obliques est 60° (voir figure ci-dessous).





Écrire la fonction `koch_curve n (x,y) (z,t)` qui prend en paramètres l'ordre n et deux positions initiales (x, y) et (z, t). Les positions initiales correspondent au coordonnées du segment de base de la courbe.

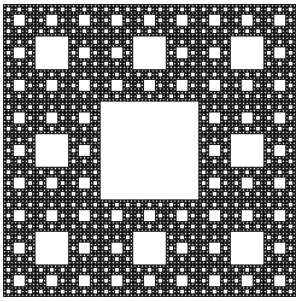
```
val koch_curve : n:int -> (x,y):int * int -> (z,t):int * int -> unit = <fun>
(* Application example *)
clear_graph ();
koch_curve 6 (5,5) (1000,5) ;;
```

Écrire la fonction `koch_snowflake n (x,y) d` qui prend en paramètres l'ordre n, une position initiale (x, y) et une taille d.

```
val koch_snowflake : n:int -> (x,y):int * int -> d:int -> unit = <fun>
(* Application example *)
clear_graph ();
koch_snowflake 6 (400,700) 400;;
```

2 Les surfaces

2.1 Éponge de Sierpinski



Écrire une fonction qui génère une "éponge" á partir d'un point de départ (x, y) et de la taille du carré.
Méthode :
La "courbe" d'ordre n à partir du point courant (x, y) est :
- $\boldsymbol{n = 0}$: un carré.
- $\boldsymbol{n > 0}$: l'éponge de Sierpinski se fabrique en découpant un carré départ en neuf sous carrés égaux (une grille de trois par trois) et la suppression du carré central. On réappliquera récursivement cette opération sur les huit carrés restants, et ainsi de suite.

Les fonctions suivantes vous seront utiles (extrait du manuel) :

```
val fill_rect : int -> int -> int -> int -> unit
```

`fill_rect` x y w h fills the rectangle with lower left corner at (x, y), width w and height h, with the current color and raises `Invalid_argument` if w or h is negative.

```
val current_point : unit -> int * int
```

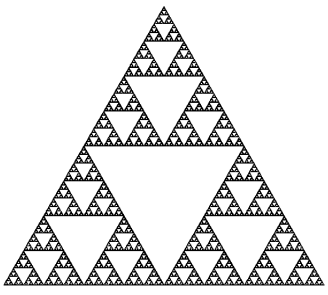
`current_point ()` returns the position of the current point.

Pour plus de précisions sur le dessin, changer le cas d'arrêt !

Écrire la fonction `carpet n (x,y)` qui prend en paramètres l'ordre n et le point de départ (x, y). Le point de départ est le coin bas gauche de l'éponge.

```
val carpet : n:int -> (x,y):int * int -> unit = <fun>
(* Application example *)
clear_graph();
carpet 243 (10,10) ;;
```

2.2 Sierpinski triangle



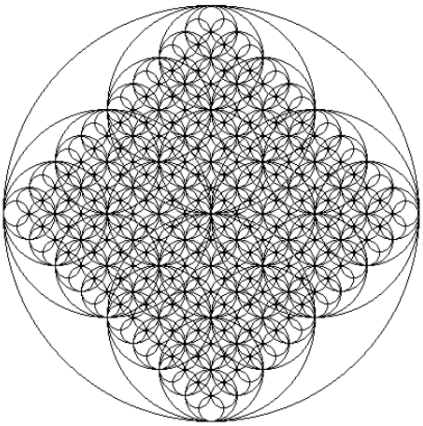
Le triangle de Sierpinski d'ordre n est défini par :
- $\boldsymbol{n = 0}$: un triangle équilatéral.
- $\boldsymbol{n \neq 0}$: le triangle d'ordre n est l'homothétie de rapport 0.5 du triangle d'ordre $n-1$, dupliqué trois fois et positionés de sorte que ceux-ci se touchent deux à deux par un sommet.

Conseil :
Ecrire une fonction auxiliaire qui permet d'afficher un triangle à partir de 3 points.

Écrire la fonction `sierpinski n (x, y) size` qui prend en paramètres l'ordre n, une position de départ (x, y) et la taille du triangle *size*. Le point de départ est le coin bas gauche du triangle.

```
val sierpinski : n:int -> (x,y):int * int -> size:int -> unit = <fun>
(* Application example *)
clear_graph ();
sierpinski 10 (5, 5) 300;;
```

2.3 Cercles



On définit la fractale circulaire représentant un cercle qui contient quatre cercles, qui contiennent chacun quatre cercles et ainsi de suite. Lorsque le cercle devient trop petit, on arrête de le dessiner.

Méthode :
Le cercle de rayon r avec une limite *limit* est défini par :
- $\boldsymbol{r < limit}$: non défini
- $\boldsymbol{r \geq limit}$: le cercle de rayon r puis quatre cercles de rayon $r/2$ sont dessinées récursivement. Ils sont placés au milieu du rayon vertical et horizontal du cercle d rayon r.

Rappel :

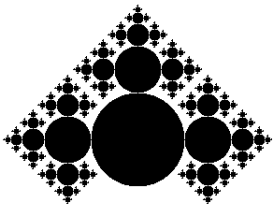
```
val draw_circle : dx:int -> dy:int -> dr:int -> unit
```

Draws a circle with center (dx, dy) and radius (dr). The current point is unchanged.

Écrire la fonction `four_circles r (x,y) limit` qui prend en paramètres le rayon initial r, une position initiale (x, y) et une limite *limit*. La position initiale est le centre de la fractale.

```
val four_circles : r:int -> (x,y):int * int -> limit:int -> unit = <fun>
(* Application example *)
clear_graph ();
four_circles 200 (300,300) 10;;
```

2.4 Flèche



On définit la fractale fléchée représentant une flèche qui pointe vers une direction. Une flèche est composée d'un cercle de rayon r et de trois plus petites flèches de rayon $r/2$ placées vers la direction de la flèche.

Méthode :
La flèche de rayon r et d'orientation o avec une limite *limit* est définie par :
- $\boldsymbol{r \leq limit}$: non défini
- $\boldsymbol{r > limit}$: un cercle de rayon r puis trois flèches de rayon $r/2$ dans la direction o. Par exemple si o est la direction nord alors les trois flèches sont vers le nord, l'est et l'ouest.

Rappel :

```
val fill_circle : dx:int -> dy:int -> dr:int -> unit
```

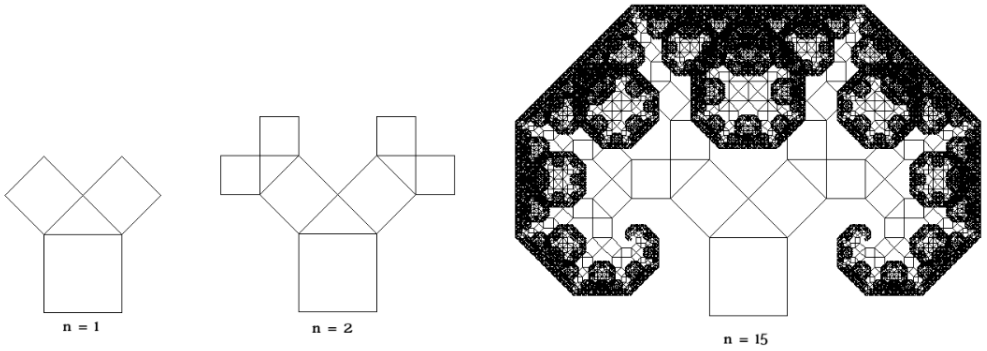
Fill a circle with the current color. The parameters are the same as for draw_circle.

Écrire la fonction `arrow r (x,y) o limit` qui rend en paramètres le rayon initial r, une position initiale (x, y), la direction initiale o ('N', 'E', 'W', 'S') et une limite *limit*. La position initiale est le centre de la fractale.

```
val arrow : r:int -> (x,y):int * int -> o:char -> limit:int -> unit = <fun>
(* Application example *)
clear_graph ();
arrow 100 (300,300) 'N' 0;;
```

2.5 Arbre de Pythagore

L'arbre de Pythagore d'ordre n est défini par :
- $\boldsymbol{n = 0}$: un carré.
- $\boldsymbol{n > 1}$: voir figure ci-dessous.



Conseil :
Ecrire une fonction auxiliaire qui permet d'afficher un carré à partir de 4 points.

Écrire la fonction `pytagora_tree n (x,y) size` qui prend en paramètres l'ordre n, la position initiale (x, y) et la taille initiale *size*. La position initiale correspond au coin bas gauche du premier carré.

```
val pytagora_tree : n:int -> (x,y):int * int -> size:int -> unit = <fun>
(* Application example *)
clear_graph ();
pytagora_tree 15 (450,5) 100;;
```

2.6 Vicsek

La *fractale de Vicsek* (version étoile), similaire à l'éponge de Sierpinski, est définie de la manière suivante :
- $\boldsymbol{n = 0}$: un carré.
- $\boldsymbol{n > 0}$: la fractale de Vicsek se fabrique en découpant un carré départ en neuf sous carrés égaux dont seulement 5 d'entre eux sont conservés. On réappliquera récursivement cette opération sur ces cinq carrés restants, et ainsi de suite.

Écrire la fonction `vicsek_star n (x,y) size` qui prend en paramètres l'ordre n, le point de départ (x, y) et la taille *size*. Le point de départ est le coin bas gauche de l'étoile.

```
val vicsek_star : n:int -> (x,y):int * int -> size:int -> unit = <fun>
(* Application example *)
clear_graph();
vicsek_star 0 (5,5) 500;
```