Recursive Pratical
Fractals

Principle

Wikipedia extract[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."

Here, the fractals are based on a recursive process of building: a segment (in the first part) is replaced by a pattern composed of several lower length segments. Then, each segment is in turn replaced by the same pattern and so on. Each additional iteration is a new curve: the superior rank curve. The fractal is the set of points at the intersections of these curves. The exercises in second part are based on the same principle, but here applied to surfaces.

Graphics library

You will need several functions from the graphics library. [2]
First you will need to load the module by adding the following lines at the beginning of the .ml file:

- with the **4.02.3** version (Windows installation suggested on the gitlab)

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

- with the **other versions**

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

Evaluate the following instructions:

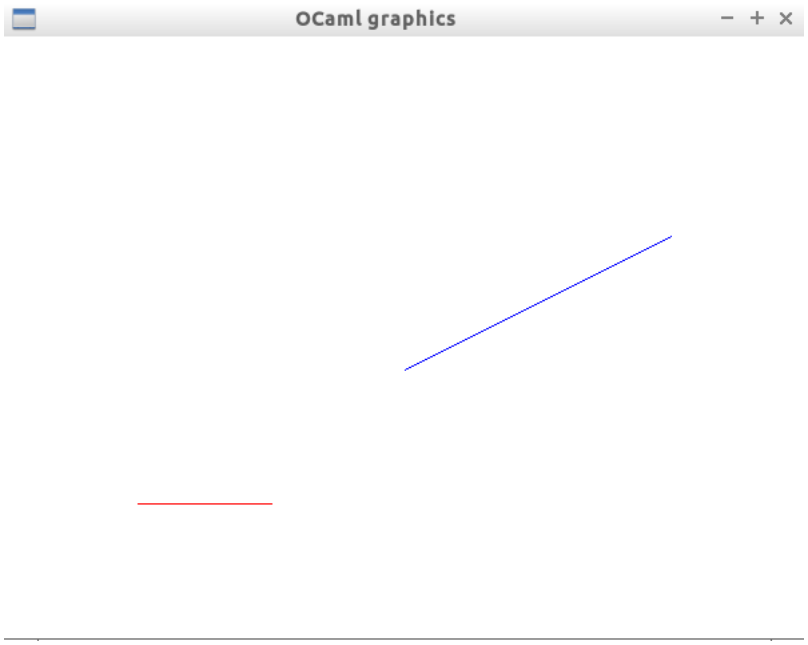
```
open Graphics ;;
open_graph " 1200x800" ;;

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) ;;
```

If the instructions worked, you should see:

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



Some functions of the Graphics module use the idea of having a current position. By default the currentposition is fixed to (0,0). For example the function `moveto` moves the current position to the (x,y) point.

Some useful functions (extract from 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 Curves

Advice for the following exercises: start with small values for the curve order...

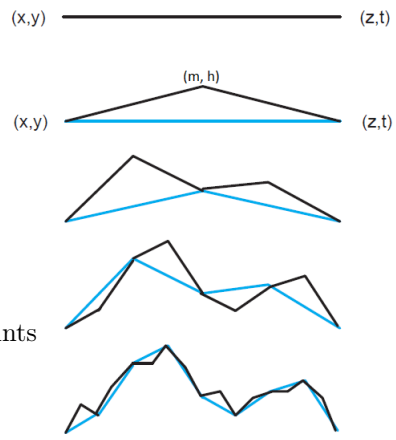
1.1 Mountain

The goal of the exercise is to generate a "mountain" randomly according to the following principle:



- **step 0:** Draw a segment between any two points.
- **step 1:** A new height for the center of the segment is computed (random[3]).
- **step n:** The same process is applied on each new segment of the previous step.

Method:
The n-order "curve" between points (x, y) and (z, t) is:



- $\boldsymbol{n = 0}$: the segment $[(x, y), (z, t)]$
- $\boldsymbol{n \neq 0}$: the $(n-1)$-order "curve" between points (x, y) and (m, h) followed by the $(n-1)$-order "curve" between points (m, h) and (z, t), where m is the "center" of x and z, and h is a new height randomly calculated.

Advice:
Calculate the new height depending on the two points and possibly on n.
As an example, we might decrease the height difference as and when the two points get closer...
Some examples of height updates where `int(e)` gives a random integer between 0 and $e - 1$:

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

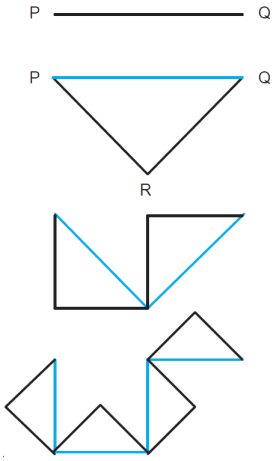
Write the function `mountain n (x,y) (z,t)` with the order n and the two starting points (x, y) and (z, t) as parameters. The two starting points are the borders of the curve.

```
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

The goal of the exercise is to generate a "dragon" randomly according to the following principle: starting from a base segment, replace each segment by 2 segments with a right angle and with a rotation of 45° alternatively to the right and to the left.

Method:
The n-order "curve" between the points P and Q is:



- $\boldsymbol{n = 0}$: the segment between the $[P, Q]$.
- $\boldsymbol{n \neq 0}$: the n-order curve is the $(n-1)$-order curve between P and R followed by the same $(n-1)$-order curve between R and Q (reversed), where PRQ is an isocel triangle rectangle in R, and R is on the right of segment PQ.

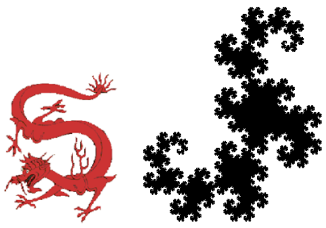
Advice:
If P and Q have the coordinates (x, y) and (z, t), the R coordinates (u, v) are:

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

[3] Functions from the pseudo-random number generator can be used: the module `Random` that has a description in the Caml manual. Do not forget to initialize the generator!

Write the function `dragon n (x,y) (z,t)` with the order n and the two starting points (x, y) (P) and (z, t) (Q) as parameters. The two starting points are the borders of the first segment of the curve.

```
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) ;;
```

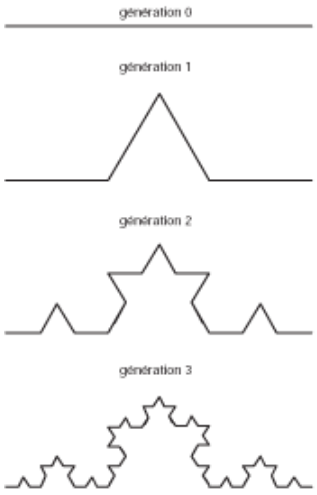


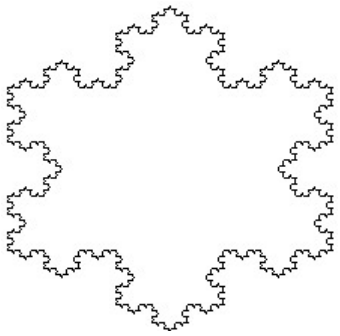
1.3 Koch snowflake

The n-order Koch snowflake is defined as follows:

- $\boldsymbol{n = 0}$: an equilateral triangle.
- $\boldsymbol{n = 1}$: an equilateral triangle which sides are cut into three parts and on each one, another equilateral triangle leans (in the middle).
- $\boldsymbol{n > 1}$: an n-order snowflake on which the same operation is applied to each of its sides.

The snowflake can also be seen as follows: three Koch curves placed on the sides of an equilateral triangle.
A Koch curve is a segment of length d replaced by 4 segments of length $d/3$. The angle of oblique segments is 60° (see the figure below).





Write the function `koch_curve n (x,y) (z,t)` with the order n and two inital positions (x, y) and (z, t)) as parameters. The initial positions correspond to the coordinates of the base segment of the curve.

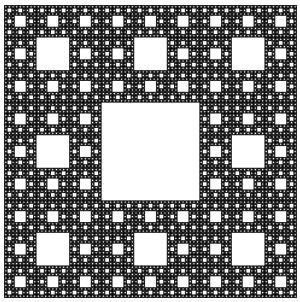
```
val koch_curve : n:int -> (x,y):int * int -> (z,t):int * int -> unit = <fun>
(* Application example *)
clear_graph ();
koch_curve 6 (5,5) (1000,5) ;;
```

Write the function `koch_curve n (x,y) d` with the order n, the inital position (x, y) and a size d as parameters.

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

2 Surfaces

2.1 Sierpinski carpet



Write a function that creates a "carpet" starting from a given point (x, y) and with a given size.
Method:
The n-order "curve" from the current point (x, y) is:

- $\boldsymbol{n = 0}$: a square.
- $\boldsymbol{n > 0}$: the construction of the Sierpinski carpet begins with a square. This square is cut into 9 congruent subsquares in a 3-by-3 grid, and the central subsquare is removed. The same procedure is then applied recursively to the remaining 8 subsquares.

These functions, from the manual, will be useful:

```
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
```

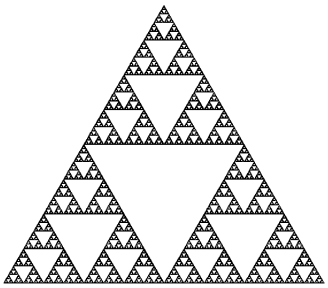
`current_point ()` returns the position of the current point.

For more details on the drawing, change the stopping criteria!

Write the function `carpet n (x,y)` with the order n and the starting point (x, y). The starting point if the bottom left corner of the carpet.

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

2.2 Sierpinski triangle



The Sierpinski triangle of order n is defined as follows:

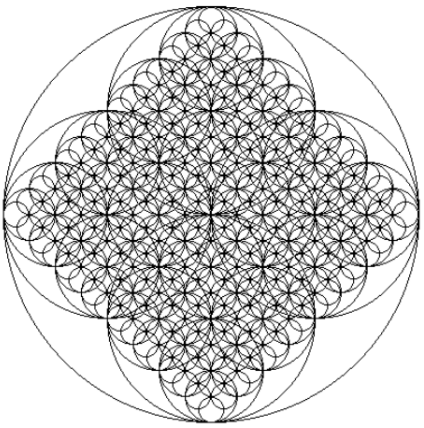
- $\boldsymbol{n = 0}$: an equilateral triangle.
- $\boldsymbol{n \neq 0}$: the (n)-order triangle is the $n-1$-order triangle homothety of scale factor 0.5, three times duplicated and placed in such a way that these are two by two adjacent by one vertex.

Advice:
Write an auxiliary function that displays a triangle given 3 points.

Write the function `sierpinski n (x, y) size` with the order n, a starting position (x, y) and the size of the triangle *size*. The starting point if the bottom left corner of the triangle.

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

2.3 Circles



A circular fractal can be defined as a circle containing four circles, each containing four circles and so on. When the circle gets too small, it is not displayed.

Method:
The circle of radius r with a limit *limit* is defined by:

- $\boldsymbol{r < limit}$: not defined
- $\boldsymbol{r \geq limit}$: the circle of radius r and four circles of radius $r/2$ are recursively displayed. They are positionned on the middle of the vertical and horizontal radiuses of the circle of radius r.

Reminder:

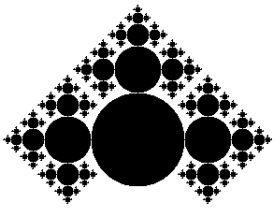
```
val draw_circle : dx:int -> dy:int -> dr:int -> unit
```

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

Write a function `four_circles r (x,y) limit` with the initial radius r, the inital position (x, y) and a limit *limit* as parameters. The initial position is the center of the fractal.

```
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 Arrow



The arrow fractal represents an arrow pointing to a direction. An arrow is composed of a circle of a radius r and three smaller arrows of radius $r/2$ positionned in the direction of the arrow.

Method:
The arrow of radius r and orientation o with a limit *limit* is defined by:

- $\boldsymbol{r \leq limit}$: not defined
- $\boldsymbol{r > limit}$: a circle of radius r and three arrows of radius $r/2$ in the direction o. For example if o is the north then the three arrows are towards the north, the east and the west.

Reminder:

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

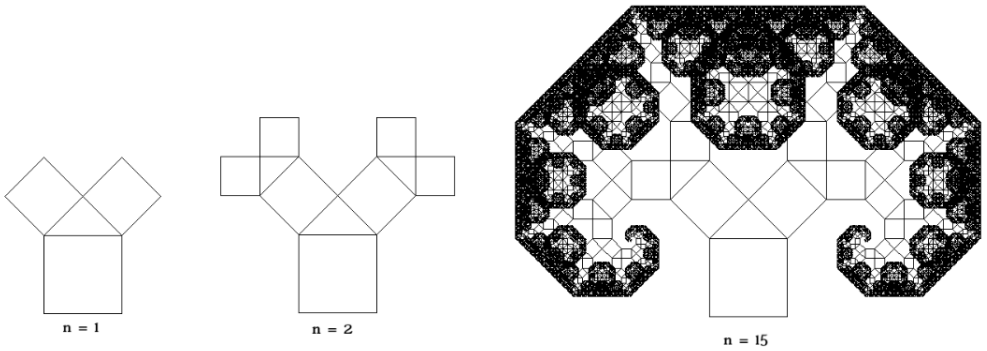
Fill a circle with the current color. The parameters are the same as for draw_circle.
Write a function `arrow r (x,y) o limit` with the initial radius r, the inital position (x, y), the initial direction o ('N', 'E', 'W', 'S') and a limit *limit* as parameters. The initial position is the center of the fractal.

```
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 Pytagora tree

The n-order Pytagora tree is defined as follows:

- $\boldsymbol{n = 0}$: a square.
- $\boldsymbol{n > 1}$: see figure below.



Advice:
Write an auxiliary function that displays a square given 4 points.
Write the function `pytagora_tree n (x,y) size` with the order n, the inital positions (x, y) and the initial size *size* as parameters. The initial position correspond to the bottom left corner of the first square.

```
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

The *Vicsek fractal* (star version), similar to the Sierpinski carpet, is defined as follows:

- $\boldsymbol{n = 0}$: a square.
- $\boldsymbol{n > 0}$: the Vicsek fractal begins with a square. This square is cut into 9 congruent subsquares in a 3-by-3 grid, and only five are conserved. The same procedure is then applied recursively to the remaining 5 subsquares.

Write the function `vicsek_star n (x,y) size` with the order n, the starting point (x, y) and the size *size*. The starting point if the bottom left corner of the star.

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

