Programmation fonctionnelle

Cours n°7 & 8 : Dessins et fractales

Licence 1 Informatique, Université Paris 8

14 & 21 Novembre 2022

Formes simples

- On peut dessiner des formes simples en Racket en utilisant le package `2htdp/image`, avec la commande

```
(require 2htdp/image)
```

Formes simples

- On peut dessiner des formes simples en Racket en utilisant le package `2htdp/image`, avec la commande

```
(require 2htdp/image)
```

- Les fonctions de base de ce package sont **circle**, **ellipse**, **line**, **triangle**, **rectangle**, **square**, **add-line**, **add-curve**, **add-solid-curve**, **text**, **text/font** et **empty-image**.

- Les fonctions **circle**, **ellipse**, *line*, etc., permettent de dessiner des formes du même nom.

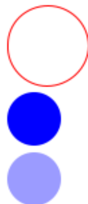
Formes simples

- On peut dessiner des formes simples en Racket en utilisant le package `2htdp/image`, avec la commande

```
(require 2htdp/image)
```

- Les fonctions de base de ce package sont **circle**, **ellipse**, **line**, **triangle**, **rectangle**, **square**, **add-line**, **add-curve**, **add-solid-curve**, **text**, **text/font** et **empty-image**.
- Les fonctions **circle**, **ellipse**, *line*, etc., permettent de dessiner des formes du même nom.
- La fonction **circle** prend en paramètres:
  - un rayon, qui est une valeur numérique;
  - un mode de tracé, qui peut être une valeur de transparence variant de 0 à 255. Le mode `solid` est par défaut la valeur 255. Pour une valeur de transparence de 0, le tracé est invisible. Le mode `outline` ne trace que le contour du cercle;
  - une couleur, sous la forme d'une chaîne de caractères, en anglais.
- **Exemples :**

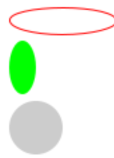
```
(circle 30 "outline" "red")
(circle 20 "solid" "blue")
(circle 20 100 "blue")
```



Formes simples II

- La fonction **ellipse** prend en paramètres:
  - une largeur, qui est une valeur numérique (longueur horizontale);
  - une hauteur, qui est une valeur numérique (longueur verticale);
  - un mode de tracé, comme pour un cercle;
  - une couleur;
- **Exemples :**

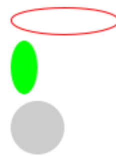
```
(ellipse 80 20 "outline" "red")
(ellipse 20 40 "solid" "green")
(ellipse 40 40 200 "gray")
```



Formes simples II

- La fonction **ellipse** prend en paramètres:
  - une largeur, qui est une valeur numérique (longueur horizontale);
  - une hauteur, qui est une valeur numérique (longueur verticale);
  - un mode de tracé, comme pour un cercle;
  - une couleur;
- **Exemples :**

```
(ellipse 80 20 "outline" "red")
(ellipse 20 40 "solid" "green")
(ellipse 40 40 200 "gray")
```



- La fonction **line** prend en paramètres:
  - une valeur en x (longueur horizontale). Si $x = 0$, la ligne sera verticale;
  - une valeur en y (longueur verticale). Si $y = 0$, la ligne sera horizontale. Par défaut, les lignes dessinées avec des valeurs en x et y positives sont orientées vers le bas à droite.
  - une couleur;
- **Exemples :**

```
(line 20 40 "red")
(line -40 20 "blue")
(line -60 -60 "green")
```

Formes simples III

- La fonction **square** (resp. **rectangle**) prend en paramètres:
  - une largeur, qui est une valeur numérique correspondant à un côté; (resp. une largeur et une longueur);
  - un mode de tracé, comme pour un cercle;
  - une couleur;
- **Exemples :**

```
(square 40 "outline" "red")
(square 50 100 "yellow")
(square 30 "solid" "blue")
```

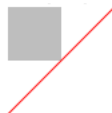
Formes simples III

- La fonction **square** (resp. **rectangle**) prend en paramètres:
  - une largeur, qui est une valeur numérique correspondant à un côté; (resp. une largeur et une longueur);
  - un mode de tracé, comme pour un cercle;
  - une couleur;
- **Exemples :**

```
(square 40 "outline" "red")
(square 50 100 "yellow")
(square 30 "solid" "blue")
```

- La fonction **add-line** permet de superposer des lignes sur un dessin en cours: l'image sur laquelle est superposée la ligne est passée en premier paramètre; et les paramètres suivants concernant l'ajout de la ligne en question.
- Cette fois, il faut donner les coordonnées de départ (en x et en y) et d'arrivée de la ligne à tracer: il y a donc 5 paramètres.

```
(add-line (square 40 "solid" "gray")
          0 80 80 0 "red")
```



Formes simples IV

- On peut aussi ajouter plusieurs lignes à un dessin en cumulant l'utilisation de *addline*, par exemple:

```
(add-line
          (add-line (square 40 "solid" "gray")
           0 80 80 0 "red")
80 80 0 0 "red")
```

Formes simples IV

- On peut aussi ajouter plusieurs lignes à un dessin en cumulant l'utilisation de *addline*, par exemple:

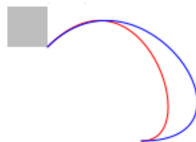
```
(add-line
        (add-line (square 40 "solid" "gray")
         0 80 80 0 "red")
80 80 0 0 "red")
```

- La fonction **add-curve** permet de tracer des courbes en les ajoutant sur une figure existante: elle prend 10 paramètres:
  - le premier est l'image concernée par l'ajout du tracé de la courbe;
  - 4 d'entre eux sont les coordonnées de départ, incluant la valeur en x, la valeur en y, l'angle en degré et la durée de conservation de cet angle;
  - les 4 suivants sont les coordonnées d'arrivée, selon le même principe;
  - le dernier est la couleur de tracé.

```
(add-curve (square 40 "solid" "blue") 0 40 135 2 0 0 -45 2 "red")
```

- On peut aussi cumuler les tracés comme précédemment:

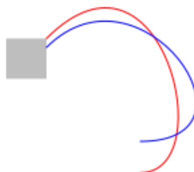
```
(add-curve
 (add-curve (square 30 "solid" "gray") 30 30 45 1 100 100 180 0.5 "red")
 30 30 45 1 100 100 180 1 "blue")
```



Formes simples V

- **Attention avec add-curve :** les coordonnées de départ placent les tracés à partir des dimensions du dessin en cours; ainsi l'ajout d'un tracé peut modifier les dimensions du sessin en cours et placer à des positions différentes des tracés ayant les mêmes coordonnées de départ.
- **Exemples :**
  - Dans le carré bleu de la slide précédente, tracé la courbe rouge symétrique de l'autre côté du carré, il faut modifier les coordonnées: ce ne sera pas $(40, 40)$ et $(40, 0)$.
  - Ci-dessous, la courbe rouge agrandit la taille du dessin et modifie la position de départ de la courbe bleue:

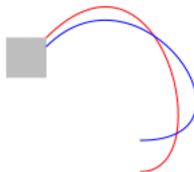
```
(add-curve
 (add-curve (square 30 "solid" "gray")
  30 0 45 1 100 100 180 0.5 "red")
30 30 45 1 100 100 180 1 "blue")
```



Formes simples V

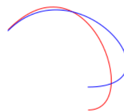
- **Attention avec add-curve :** les coordonnées de départ placent les tracés à partir des dimensions du dessin en cours; ainsi l'ajout d'un tracé peut modifier les dimensions du sessin en cours et placer à des positions différentes des tracés ayant les mêmes coordonnées de départ.
- **Exemples :**
  - Dans le carré bleu de la slide précédente, tracé la courbe rouge symétrique de l'autre côté du carré, il faut modifier les coordonnées: ce ne sera pas $(40, 40)$ et $(40, 0)$.
  - Ci-dessous, la courbe rouge agrandit la taille du dessin et modifie la position de départ de la courbe bleue:

```
(add-curve
 (add-curve (square 30 "solid" "gray")
  30 0 45 1 100 100 180 0.5 "red")
30 30 45 1 100 100 180 1 "blue")
```



- La fonction **empty-image** correspond au tracé d'un carré blanc de côté 0: ainsi, superposer les deux courbes précédentes avec coordonnées de départ en $(0, 0)$ équivaut à placer l'ordonnée de départ de la deuxième courbe au point le plus haut de la première.

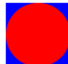
```
(add-curve
 (add-curve empty-image
  0 0 45 1 100 100 180 0.5 "red")
0 30 45 1 100 100 180 1 "blue")
```



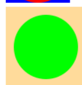
Superposer des dessins

- Les fonctions **overlay** et **underlay** permettent de superposer des dessins de toutes formes. Les deux dessins sont superposés centrés; avec **underlay**: le premier paramètre est une sous-couche du second; avec **overlay**: le premier est une sur-couche du second.

```
(underlay (square 80 "solid" "blue")
          (circle 40 "solid" "red"))
```



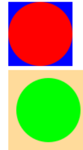
```
(overlay (circle 40 "solid" "green")
         (square 100 100 "orange"))
```



Superposer des dessins

- Les fonctions **overlay** et **underlay** permettent de superposer des dessins de toutes formes. Les deux dessins sont superposés centrés; avec **underlay**: le premier paramètre est une sous-couche du second; avec **overlay**: le premier est une sur-couche du second.

```
(underlay (square 80 "solid" "blue")
          (circle 40 "solid" "red"))

(overlay (circle 40 "solid" "green")
         (square 100 100 "orange"))
```



- Les fonctions **overlay/xy** et **underlay/xy** permettent de préciser des décalages entre les dessins superposés:

```
(underlay/xy (square 80 "solid" "gray")
             60 0
             (square 40 "solid" "blue"))

(overlay/xy (square 40 "solid" "red")
            60 0
            (square 80 "solid" "gray"))

(overlay/xy (square 80 "solid" "gray")
            60 0
            (square 40 "solid" "red"))
```

- Les valeurs x et y en deuxième ligne définissent le décalage du second par rapport au premier. Avec **underlay/xy**, le premier est une sous-couche, donc le second peut cacher partiellement le premier ; avec **overlay/xy**, le premier est une sur-couche, donc le premier peut cacher partiellement le second.

Première fonction de dessin

- On va définir une fonction qui génère un nombre n de tracés aléatoires:

```
(define (rline n)
  (if (= n 0)
      empty-image
      (underlay
       (line (random 100) (- (random 200) 100) "black")
       (rline (- n 1)))))

(rline 10)
```

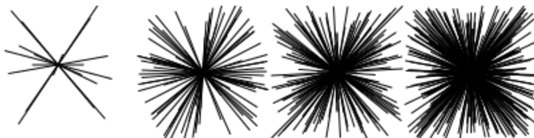
Première fonction de dessin

- On va définir une fonction qui génère un nombre n de tracés aléatoires:

```
(define (rline n)
  (if (= n 0)
      empty-image
      (underlay
       (line (random 100) (- (random 200) 100) "black")
       (rline (- n 1)))))

(rline 10)
```

- On peut ensuite afficher différents résultats de cette fonction en utilisant **underlay/xy** ou **overlay/xy**:

```
(underlay/xy
 (underlay/xy
  (underlay/xy
   (rline 10)
   100 0 (rline 50))
  200 0 (rline 100))
 300 0 (rline 200))
```

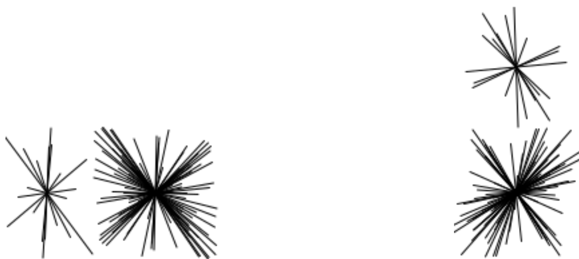


- Ici, les deux fonctions fournissent le même résultat puisqu'avec le décalage de 100 en x imposé, les figures obtenues ne s'intersectent pas.

Placement de dessins

- Il est également possible de placer les images les unes par rapport aux autres à l'aide des fonctions:
  - **above** – verticalement vers le bas;
  - **above/align** – verticalement vers le bas avec un alignement à gauche ou à droite;
  - **beside** – horizontalement vers la gauche;
  - **beside/align** – idem avec alignement à gauche ou à droite. Pour les fonctions avec align, il faut ajouter une string qui précise l'alignement `"right"`, `"left"`, `"top"` ou `"bottom"`.

- **Exemples :**



`(beside (rline 10) (rline 50))` `(above (rline 10) (rline 50))`

- **Remarque :** En fait, ces fonctions de placement acceptent des listes de dessins comme arguments: par exemple, `(above '(A B C))` indiquerait de placer A au dessus de B au dessus de C.

Alignement

- Voici un exemple d'utilité des fonctions d'alignement:

```
(beside (above (square 40 "solid" "blue")
               (square 80 "solid" "gray"))
        (square 60 "solid" "red"))
```

Alignement

- Voici un exemple d'utilité des fonctions d'alignement:

```
(beside (above (square 40 "solid" "blue")
               (square 80 "solid" "gray"))
        (square 60 "solid" "red"))
```

- Ici, on va d'abord superposer verticalement le carré bleu et le gris, puis placer le carré rouge à côté de cette nouvelle figure. Mais on remarque que ce carré est centré par rapport à la figure composée des deux carrés précédents. Mieux:

```
(beside/align "bottom"
              (above/align "right"
                           (square 40 "solid" "blue")
                           (square 80 "solid" "gray"))
              (square 60 "solid" "red"))
```

Alignement

- Voici un exemple d'utilité des fonctions d'alignement:

```
(beside (above (square 40 "solid" "blue")
               (square 80 "solid" "gray"))
        (square 60 "solid" "red"))
```

- Ici, on va d'abord superposer verticalement le carré bleu et le gris, puis placer le carré rouge à côté de cette nouvelle figure. Mais on remarque que ce carré est centré par rapport à la figure composée des deux carrés précédents. Mieux:

```
(beside/align "bottom"
              (above/align "right"
                           (square 40 "solid" "blue")
                           (square 80 "solid" "gray"))
              (square 60 "solid" "red"))
```

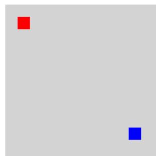
- Les placements sont définis par rapport au rectangle englobant une figure: par exemple, si on reprend les placements précédents avec des triangles:

```
(beside
 (beside
  (above (triangle 40 "solid" "blue")
         (triangle 80 "solid" "gray"))
  (triangle 60 "solid" "red"))
 (beside/align "bottom"
               (above/align "right"
                            (triangle 40 "solid" "blue")
                            (triangle 80 "solid" "gray"))
               (triangle 60 "solid" "red")))
```

Ajout de texte

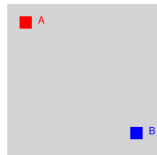
- La fonction **text** permet d'ajouter du texte dans un dessin. Elle prend en paramètres une chaîne de caractères qui est le texte à ajouter, une valeur numérique qui représente la taille du texte, et une couleur.
- **Exemple :** Voici une instruction permettant d'ajouter un petit carré rouge et un petit carré bleu en surcouche d'un grand carré gris:

```
(underlay/xy (underlay/xy
  (square 250 "solid" "lightgray")
  20 20 (square 20 "solid" "red"))
200 200 (square 20 "solid" "blue"))
```



- On peut ajouter deux textes en ajoutant deux fonctions **underlay/xy** et les deux commandes suivantes:

```
(underlay/xy (underlay/xy (underlay/xy (underlay/xy
(square 250 "solid" "lightgray")
20 20 (square 20 "solid" "red"))
50 20 (text "A" 16 "red"))
200 200 (square 20 "solid" "blue"))
230 200 (text "B" 16 "blue"))
```



- La fonction **text/font** permet de préciser taill de font, famille et style, par exemple:

```
(text/font "A" 16 "red" "times" 'italic 'bold #t)
```

Triangles, polygones

- La fonction **triangle** définit un triangle équilatéral avec pour paramètres une taille de côté, le mode de tracé et la couleur.

- On peut aussi tracer différents types de triangles, notamment des isocèles ou des triangles rectangles.

- **Exemples :**

```
(right-triangle 40 50 "solid" "Cyan")
(isosceles-triangle 60 50 "solid" "Purple")
```

Triangles, polygones

- La fonction **triangle** définit un triangle équilatéral avec pour paramètres une taille de côté, le mode de tracé et la couleur.
- On peut aussi tracer différents types de triangles, notamment des isocèles ou des triangles rectangles.
- **Exemples :**

```
(right-triangle 40 50 "solid" "Cyan")
(isosceles-triangle 60 50 "solid" "Purple")
```

- Liste des principaux polygones traçables, et des paramètres requis:

`(triangle side-length mode color)`	pour un triangle équilatéral
`(right-triangle side-length1 side-length2 mode color)`	pour un triangle rectangle
`(isosceles-triangle side-length angle mode color)`	pour un triangle isocèle
`(square side-len mode color)`	pour un carré
`(rectangle width height mode color)`	pour un rectangle

Couleurs

- Voici la liste des principaux mots-clés de couleurs utilisables en Racket:

Tomato, Red, Pink, Brown, Orange, Gold, Yellow, Green, Turquoise, Cyan, Blue, Indigo, Purple, White, LightGray, Silver, Gray, DarkGray, Black

Couleurs

- Voici la liste des principaux mots-clés de couleurs utilisables en Racket:

Tomato, Red, Pink, Brown, Orange, Gold, Yellow, Green, Turquoise, Cyan, Blue, Indigo, Purple, White, LightGray, Silver, Gray, DarkGray, Black

- Il est également possible de définir une couleur à partir de 3 ou 4 paramètres:

  `(color r g b)` ou `(color r g b a)`

  avec `r`,`g`, `b` représentant respectivement les teintes de rouge, vert et bleu de la couleur, et `a` représentant la transparence, allant de 0 (complètement transparent) à 255 (complètement opaque). La valeur par défaut de `a` est 255.

Couleurs

- Voici la liste des principaux mots-clés de couleurs utilisables en Racket:

Tomato, Red, Pink, Brown, Orange, Gold, Yellow, Green, Turquoise, Cyan, Blue, Indigo, Purple, White, LightGray, Silver, Gray, DarkGray, Black

- Il est également possible de définir une couleur à partir de 3 ou 4 paramètres:

  `(color r g b)` ou `(color r g b a)`

  avec `r`,g, `b` représentant respectivement les teintes de rouge, vert et bleu de la couleur, et `a` représentant la transparence, allant de 0 (complètement transparent) à 255 (complètement opaque). La valeur par défaut de `a` est 255.

- **Exemple :**

```
(define (A alpha) (square 100 "solid" (color 200 0 0 alpha)))
(define B (circle 50 "solid" (color 0 200 0)))
(beside (underlay B (A 255))
(underlay B (A 200))
(underlay B (A 100))
(underlay B (A 50))
(underlay B (A 0)))
```

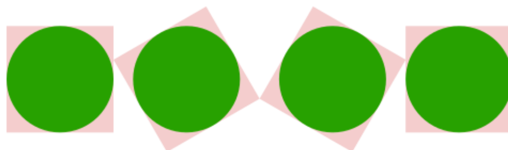
Rotation d'un dessin

- La fonction **rotate** permet d'appliquer une rotation sur un dessin; elle prend deux paramètres: un angle `a` de rotation et une figure `f`.

```
(rotate a f)
```

- **Exemple :**

```
(define (C a) (rotate a (square 100 "solid" (color 200 0 0 50))))
(define D (circle 50 "solid" (color 0 200 0)))
(beside (underlay D (C 0)) (underlay D (C 30))
(underlay D (C 60)) (underlay B (C 90)))
```



Sauvegarder un dessin dans un fichier

- La fonction save-image permet de sauvegarder un dessin dans un fichier au format `.png`.

```
(save-image
 (underlay/xy
 (underlay/xy
  (underlay/xy
   (rline 10)
   100 0 (rline 50))
  200 0 (rline 100))
 300 0 (rline 200))
 "image1.png")
```

a pour effet d'enregistrer la figure dessinée dans le répertoire courant du projet Racket, sous le nom `image1.png`.

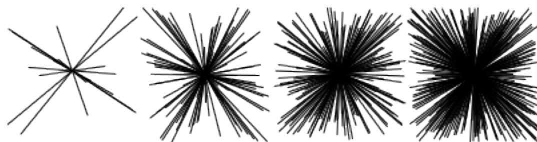
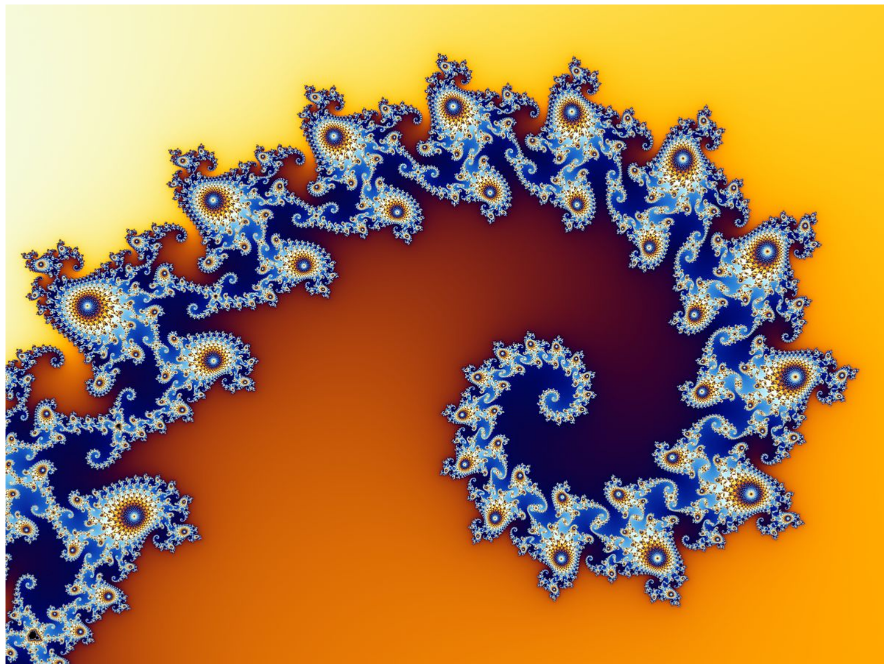


Figure: Image `image1.png` enregistrée.

- Notez que un appel à la fonction **save-image** retourne #t en fin d'exécution, pour indiquer que l'image s'est bien enregistrée.

Fractales

- Une **fractale** est (Wikipedia) un objet géométrique «infiniment morcelé» dont des détails sont observables à une échelle arbitrairement choisie. En zoomant sur une partie de la figure, on peut retrouver toute la figure, on dit qu'elle est auto similaire.



Fractales

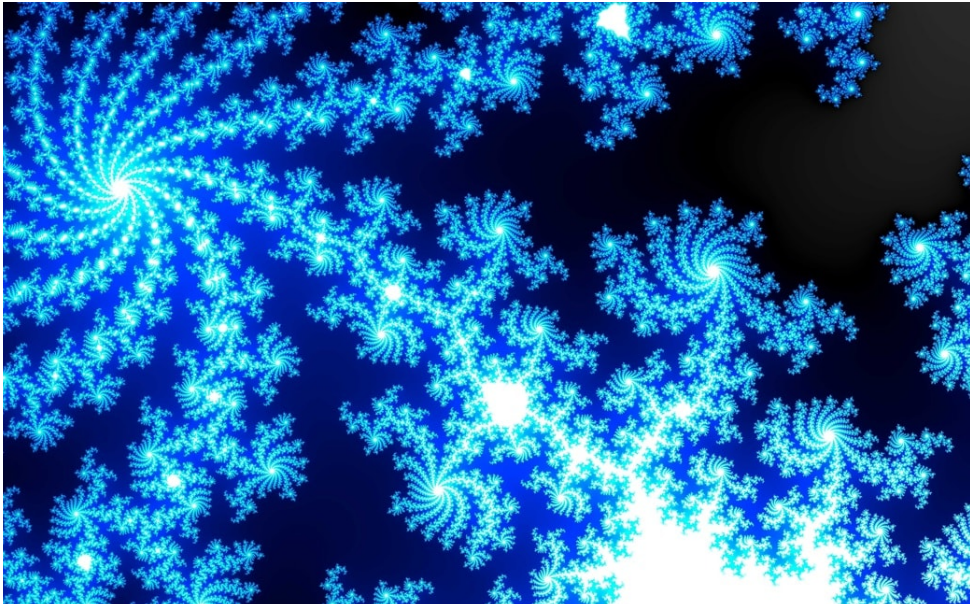
- Une **fractale** est (Wikipedia) un objet géométrique «infiniment morcelé» dont des détails sont observables à une échelle arbitrairement choisie. En zoomant sur une partie de la figure, on peut retrouver toute la figure, on dit qu'elle est auto similaire.

Fractales

- Une **fractale** est (Wikipedia) un objet géométrique «infiniment morcelé» dont des détails sont observables à une échelle arbitrairement choisie. En zoomant sur une partie de la figure, on peut retrouver toute la figure, on dit qu'elle est auto similaire.

Fractales

- Une **fractale** est (Wikipedia) un objet géométrique «infiniment morcelé» dont des détails sont observables à une échelle arbitrairement choisie. En zoomant sur une partie de la figure, on peut retrouver toute la figure, on dit qu'elle est auto similaire.

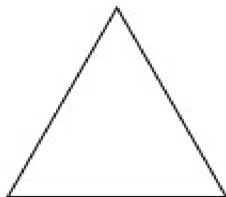
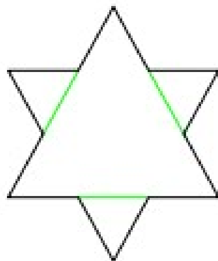
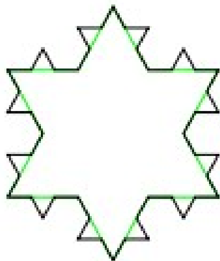
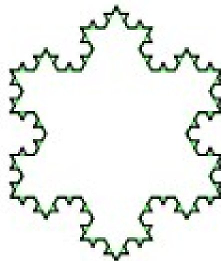


Fractales

- Une **fractale** est (Wikipedia) un objet géométrique «infiniment morcelé» dont des détails sont observables à une échelle arbitrairement choisie. En zoomant sur une partie de la figure, on peut retrouver toute la figure, on dit qu'elle est auto similaire.

Fractales

- Une **fractale** est (Wikipedia) un objet géométrique «infiniment morcelé» dont des détails sont observables à une échelle arbitrairement choisie. En zoomant sur une partie de la figure, on peut retrouver toute la figure, on dit qu'elle est auto similaire.

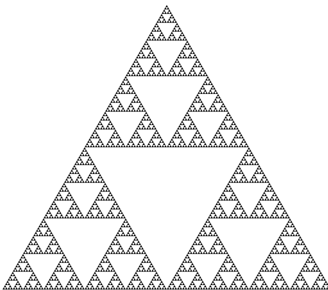





Triangle de Sierpiñski

- Le **triangle de Sierpiñski** est une fractale construite à partir d'un triangle équilatéral.
- **Idée :** Considérer les milieux des 3 côtés du triangle, puis supprimer le triangle interne constitué des 3 milieux, et répéter ce processus pour tous les triangles restants.

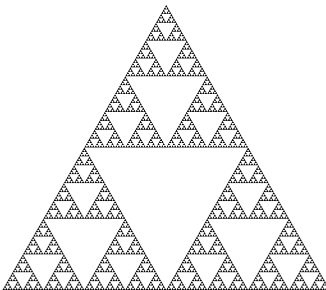
Triangle de Sierpiñski

- Le **triangle de Sierpiñski** est une fractale construite à partir d'un triangle équilatéral.
- **Idée :** Considérer les milieux des 3 côtés du triangle, puis supprimer le triangle interne constitué des 3 milieux, et répéter ce processus pour tous les triangles restants.



Triangle de Sierpiñski

- Le **triangle de Sierpiñski** est une fractale construite à partir d'un triangle équilatéral.
- **Idée :** Considérer les milieux des 3 côtés du triangle, puis supprimer le triangle interne constitué des 3 milieux, et répéter ce processus pour tous les triangles restants.



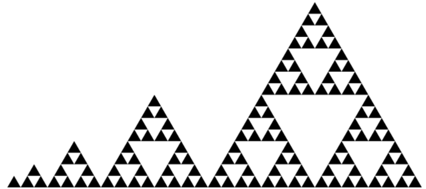
- La construction de ce triangle est **récursive**: pour construire le triangle suivant, il suffit de répéter trois fois le précédent: deux alignés horizontalement, puis le troisième juxtaposé verticalement.

```
(define (sierpinski0 d) (triangle d "solid" "black"))

 (define (sierpinski n d)
   (if (= n 0)
       (sierpinski0 d)
       (above (sierpinski (- n 1) d)
              (beside (sierpinski (- n 1) d) (sierpinski (- n 1) d)))))
```

Triangle de Sierpiñski II

- **Exemple 1:** Tests en gardant la même longueur de triangle au fil des appels récursifs: la taille du dessin est multipliée par 2 à chaque étape.

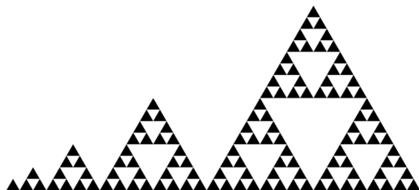
```
(beside/align "bottom"(sierpinski 0 20)
                       (sierpinski 1 20)
                       (sierpinski 2 20)
                       (sierpinski 3 20)
                       (sierpinski 4 20))
```



Triangle de Sierpiñski II

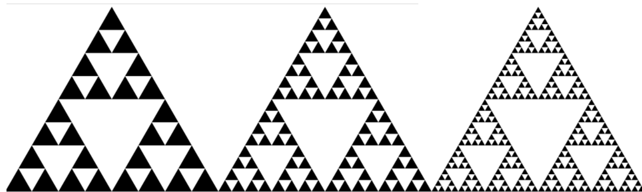
- **Exemple 1:** Tests en gardant la même longueur de triangle au fil des appels récursifs: la taille du dessin est multipliée par 2 à chaque étape.

```
(beside/align "bottom"(sierpinski 0 20)
              (sierpinski 1 20)
              (sierpinski 2 20)
              (sierpinski 3 20)
              (sierpinski 4 20))
```



- **Exemple 2:** Tests en divisant par 2 la longueur des triangles au fil des appels récursifs: la taille du dessin est conservée, et les triangles sont de plus en plus petits.

```
(beside (sierpinski 3 40)
        (sierpinski 4 20)
        (sierpinski 5 10))
```



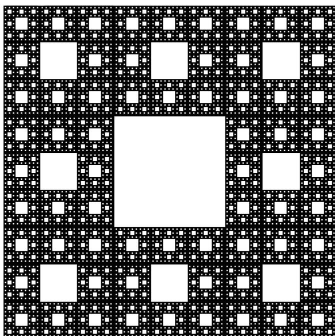
Tapis de Sierpiñski

- Le **tapis de Sierpiñski** est une fractale construite à partir d'un carré.
- **Idée :** Voir le carré comme une grille de 3×3 (séparée en 9 sous-carrés, puis retirer le carré central, et répéter ce processus.

Tapis de Sierpiñski

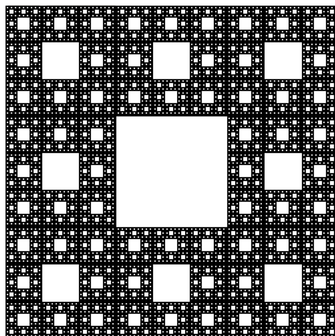
- Le **tapis de Sierpiñski** est une fractale construite à partir d'un carré.

- **Idée :** Voir le carré comme une grille de 3×3 (séparée en 9 sous-carrés, puis retirer le carré central, et répéter ce processus.



Tapis de Sierpiñski

- Le **tapis de Sierpiñski** est une fractale construite à partir d'un carré.
- **Idée :** Voir le carré comme une grille de 3×3 (séparée en 9 sous-carrés, puis retirer le carré central, et répéter ce processus.



- La construction de ce tapis est **récursive**: pour construire le tapis suivant, il suffit prendre le tapis précédent, noté C, et de le répéter selon le schéma

$$\begin{array}{ccc} C & C & C \\ C & & C \\ C & C & C \end{array}$$

Tapis de Sierpiñski II

- On définir d'abord le carré noir de base, avec un paramètre de longueur, et on peut même rajouter un paramètre de couleurs pour intégrer les variantes noir/blanc et blanc/noir.

```
(define (sierpinski0 d col)
  (square d "solid" col))
```

- On définit ensuite la procédure récursive, en dessinant ligne par ligne avec un **above**.

```
(define (sierpinski n d col0 col1)
(if (= n 0)
    (sierpinski0 d col0)
    (above (sierp-line0 (- n 1) d col0 col1)
           (sierp-line1 (- n 1) d col0 col1)
           (sierp-line0 (- n 1) d col0 col1))))
```

- Notez que les fonctions **sierp-line0** et **sierp-line1** définissent les deux types de ligne: la première trace une ligne de trois carrés précédents avec l'alternance de couleur; la seconde trace une ligne de trois carrés mais le second garde la même couleur, autrement dit il restera soit noir, soit blanc.

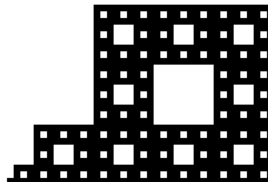
```
(define (sierp-line0 n d col0 col1)
  (beside (sierpinski n d col0 col1)
          (sierpinski n d col0 col1)
          (sierpinski n d col0 col1)))
```

```
(define (sierp-line1 n d col0 col1)
  (beside (sierpinski n d col0 col1)
          (sierpinski n d col1 col1)
          (sierpinski n d col0 col1)))
```

Tapis de Sierpiñski III

- **Exemple 1:** Tests en gardant la même longueur de carré au fil des appels récursifs: la taille du dessin est multipliée par 3 à chaque étape. Le choix de couleur est noir, blanc.

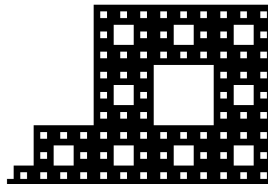
```
(beside/align "bottom"
              (sierpinski 0 10 "black" "white")
              (sierpinski 1 10 "black" "white")
              (sierpinski 2 10 "black" "white")
              (sierpinski 3 10 "black" "white"))
```



Tapis de Sierpiñski III

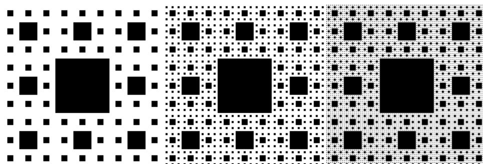
- **Exemple 1:** Tests en gardant la même longueur de carré au fil des appels récursifs: la taille du dessin est multipliée par 3 à chaque étape. Le choix de couleur est noir, blanc.

```
(beside/align "bottom"
              (sierpinski 0 10 "black" "white")
              (sierpinski 1 10 "black" "white")
              (sierpinski 2 10 "black" "white")
              (sierpinski 3 10 "black" "white"))
```



- **Exemple 2:** Tests en divisant par 3 la longueur des carrés au fil des appels récursifs: la taille du dessin est conservée, et les carrés sont de plus en plus petits. Le choix de couleur est blanc, noir.

```
(beside
 (sierpinski 3 9 "white" "black")
 (sierpinski 4 3 "white" "black")
 (sierpinski 5 1 "white" "black"))
```



Manipulation d'images pixel par pixel

- L'utilisation du package `2htdp/image` permet de manipuler des images: une image est une matrice/tableau à double entrées de pixels. **Pour chaque pixel, on a 4 composantes `r g b a`.**
- La fonction **bitmap** permet de charger une image `.png` ou `.jpg` dans Racket. Il faut lui indiquer en paramètre le chemin d'accès à l'image.
- **Exemple :** Déposer l'image `nuages.png` dans le dossier courant.

```
(bitmap "nuages.png")
```

Manipulation d'images pixel par pixel

- L'utilisation du package `2htdp/image` permet de manipuler des images: une image est une matrice/tableau à double entrées de pixels. **Pour chaque pixel, on a 4 composantes `r g b a`.**
- La fonction **bitmap** permet de charger une image `.png` ou `.jpg` dans Racket. Il faut lui indiquer en paramètre le chemin d'accès à l'image.
- **Exemple :** Déposer l'image `nuages.png` dans le dossier courant.

```
(bitmap "nuages.png")
```

- Voici ci-dessous un exemple de fonction permettant de modifier l'image. On va mettre l'image en gris en stockant dans chaque pixel la même couleur correspondant à la moyenne des composantes `r g b`.

```
(define (image-color->gray img)
  (define (f L)
    (if (null? L)
        L
        (let* ((pix (car L))
               (sum-pix (+ (color-red pix)
                           (color-green pix)
                           (color-blue pix)))
               (gpix (round (/ sum-pix 3))))
          (cons (color gpix gpix gpix (color-alpha pix)) (f (cdr L))))))
  (color-list->bitmap (f (image->color-list img))
                      (image-width img)
                      (image-height img)))
```

```
(image-color->gray (bitmap "nuages.png"))
```

Manipulation d'images pixel par pixel II

- Pour manipuler des images, on utiliser principalement les fonctions:
  - **image->color-list** qui permet de récupérer la liste des couleurs contenues dans une image; par exemple,

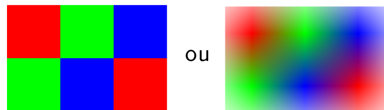
    `(image->color-list (bitmap "nuages.png"))`

    donne la liste de toutes les couleurs différentes contenues dans `nuages.png`, et il y en a beaucoup !
  - **color-list->bitmap** qui permet de créer une image à partir d'une liste de couleurs. Elle prend en paramètres cette liste, ainsi que la taille horizontale et verticale de l'image.

- **Exemples :**

```
(scale 50
(color-list->bitmap (list "red" "green" "blue") 3 1))
```

```
(scale 100
(color-list->bitmap
(list "red" "green" "blue" "green" "blue" "red") 3 2))
```



- **Remarques :**
  - Il est également possible de charger une image à partir d'une url avec la fonction **bitmap/url**.
  - On pourra utiliser le prédicat **image?** pour savoir si un élément est une image, et **image-color?** pour savoir si un élément est un pixel.

Manipulation d'images pixel par pixel III

- On va définir une fonction de construction d'une image conformément à une fonction `f` dépendent des coordonnées `x` et `y`:

```
(define (mk-image-line y fun width)
  (define (f x)
    (if (= x width)
        empty
        (cons (fun x y) (f (add1 x)))))
  (f 0))

(define (mk-image fun width height)
  (define (f y L)
    (if (= y height)
        (color-list->bitmap L width height)
        (f (add1 y) (append L (mk-image-line y fun width)))))
  (f 0 empty))
```

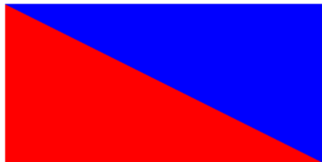
Manipulation d'images pixel par pixel III

- On va définir une fonction de construction d'une image conformément à une fonction `f` dépendent des coordonnées `x` et `y`:

```
(define (mk-image-line y fun width)
  (define (f x)
    (if (= x width)
        empty
        (cons (fun x y) (f (add1 x)))))
  (f 0))

(define (mk-image fun width height)
  (define (f y L)
    (if (= y height)
        (color-list->bitmap L width height)
        (f (add1 y) (append L (mk-image-line y fun width)))))
  (f 0 empty))
```

- On peut ensuite l'appliquer à une fonction qui va colorier le triangle inférieur gauche en rouge, et le supérieur droit en bleu:

```
(define (rgb-fun x y)
  (if (= y 0)
      (color 0 0 255)
      (if (< (/ x y) 2)
          (color 255 0 0)
          (color 0 0 255))))
(mk-image rgb-fun 400 200)
```



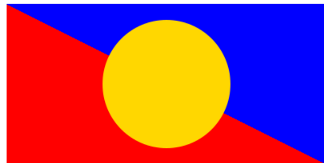
Manipulation d'images pixel par pixel III

- On va définir une fonction de construction d'une image conformément à une fonction `f` dépendent des coordonnées `x` et `y`:

```
(define (mk-image-line y fun width)
  (define (f x)
    (if (= x width)
        empty
        (cons (fun x y) (f (add1 x)))))
  (f 0))

(define (mk-image fun width height)
  (define (f y L)
    (if (= y height)
        (color-list->bitmap L width height)
        (f (add1 y) (append L (mk-image-line y fun width)))))
(f 0 empty))
```

- On peut ensuite l'appliquer à une fonction qui va colorier le triangle inférieur gauche en rouge, et le supérieur droit en bleu:

```
(define (rgb-fun x y)
  (if (= y 0)
      (color 0 0 255)
      (if (< (/ x y) 2)
          (color 255 0 0)
          (color 0 0 255))))
(mk-image rgb-fun 400 200)
```



- On peut superposer une image vectorielle avec les fonctions de dessin précédentes:

```
(overlay (circle 100 "solid" "gold") (mk-image rgb-fun 400 200))
```

Dessiner des fractales I

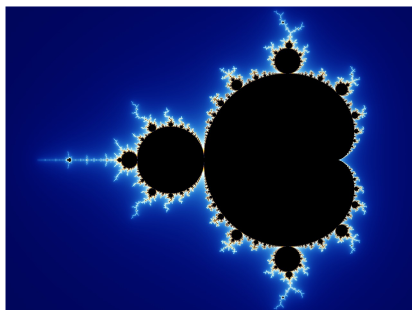
- On va pouvoir dessiner des fractales avec la fonction **mk-image**, en utilisant des **nombres complexes**.
- Un nombre complexe est un nombre de la forme $a + ib$ où a et b sont des nombres réels et i est un nombre appelé imaginaire pur, vérifiant $i^2 = -1$.
- On peut se servir d'un tel nombre pour représenter un point dans le plan de coordonnées (a, b).
  - On dit que a est la partie réelle de $a + ib$, et b est sa partie imaginaire.
- Un nombre complexe peut aussi être écrit sous la forme $z = \rho e^{i\theta}$ où ρ est un nombre réel positif mesurant la distance entre $(0, 0)$ et le point associé à z, et θ est l'angle entre l'axe horizontal et la droite reliant $(0, 0)$ au point associé à z.
- Si $z = a + ib = \rho e^{i\theta}$, alors

$$a = \rho \cos(\theta), b = \rho \sin(\theta), \qquad \rho = \sqrt{a^2 + b^2},\ \theta = 2 \arctan\left(\frac{b}{a + \sqrt{a^2 + b^2}}\right)$$

- La fonction **(make-rectangular a b)** permet de définir le nombre complexe $a + ib$. La fonction **real-part** permet de récupérer la partie réelle d'un complexe; la fonction **imag-part** permet de récupérer la partie imaginaire.
- La fonction **(make-polar rho theta)** permet de définir un complexe de forme polaire $\rho e^{i\theta}$. La fonction **magnitude** permet de récupérer la valeur de ρ; la fonction **angle** permet de récupérer la valeur de θ.

Dessiner des fractales II

- L'ensemble de Mandelbrot est un fractale qui ressemble à:



- Cet ensemble est défini par une suite récurrente: on part d'un complexe z_0, et ensuite on itère avec

$$z_n = z_{n-1}^2 + c$$

où c est un nombre complexe donné.

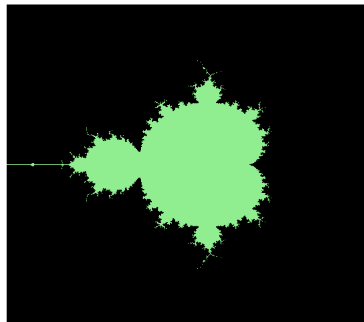
- On fixe une distance maximale m. Après n itérations, on regarde la distance la distance à l'origine des points obtenus: les positions de départ z_0 ayant un z_n qui est à distance supérieur à m sont coloriés d'une couleur; les z_0 dont le z_n sont situés à distance inférieure à m sont coloriés d'une autre couleur.

```
(define maxite 20)
(define maxdist 2.0)
```

Dessiner des fractales III

- Selon le point de départ z_0, la suite z_n sera bornée (quand la distance ne dépasse pas la valeur limite) ou non bornée (quand la distance croît avec n).
- On va regarder cette distance après 20 itérations:
  - la fonction `f1` va calculer le terme suivant de la suite (z_n);
  - la fonction `ite-in` retourne le nombre k pour lequel z_k est à distance inférieure à `maxdist`;
  - le calcul est limité à `maxite` itérations: si après `maxite` itérations on a une distance inférieure à `maxdist`, on retourne `maxite`.

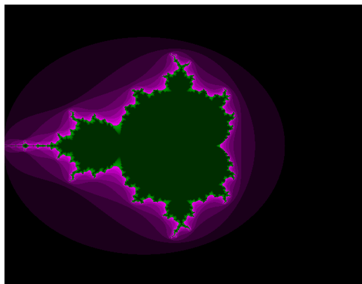
```
(define (ite-in x fun)
  (define (f xi ite)
    (if (> ite maxite)
        ite
        (let ([nx (fun x xi)])
          (if (> (magnitude nx) maxdist)
              ite
              (f nx (+ 1 ite))))))
  (f x 0))

(define (f x y)
  (let ([nx (/ (- x 320) 160.0)]
        [ny (/ (- y 240) 140.0)])
    (if (< (ite-in (make-rectangular nx ny) f1) maxite)
        "black"
        "lightgreen")))
```



Dessiner des fractales IV

- Pour ajouter un dégradé sur les valeurs inférieures à 10 et un dégradé sur les valeurs supérieures à 10, on modifie la fonction `f` en testant le nombre d'itérations pour être au delà de la distance `maxdist`.

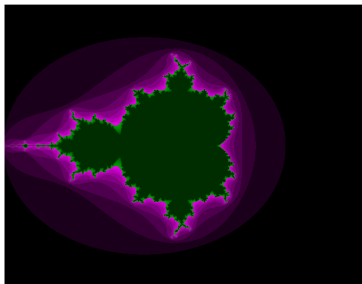
```
(define (f x y)
(let ([nx (/ (- x 320) 160.0)]
      [ny (/ (- y 240) 140.0)]
      [nite (ite-in (make-rectangular nx ny) f1)])
  (if (< nite 10)
      (color (* nite 26) 0 (* nite 26))
      (color 0 (- 255 (* nite 10)) 0))))
(mk-image f 640 480)
```



Dessiner des fractales IV

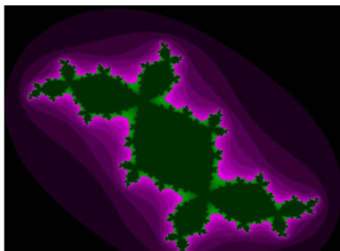
- Pour ajouter un dégradé sur les valeurs inférieures à 10 et un dégradé sur les valeurs supérieures à 10, on modifie la fonction `f` en testant le nombre d'itérations pour être au delà de la distance `maxdist`.

```
(define (f x y)
(let ([nx (/ (- x 320) 160.0)]
      [ny (/ (- y 240) 140.0)]
      [nite (ite-in (make-rectangular nx ny) f1)])
  (if (< nite 10)
      (color (* nite 26) 0 (* nite 26))
      (color 0 (- 255 (* nite 10)) 0))))
(mk-image f 640 480)
```



- En modifiant la fonction `f1`, c'est-à-dire en itérant avec d'autres nombres complexes c la fonction $z^2 + c$, on obtient d'autres fractales.
- Le **lapin de Douady**:

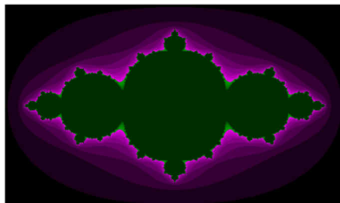
```
(define (f1 z z0)
  (+ (* z0 z0) -0.124-0.758i))
```

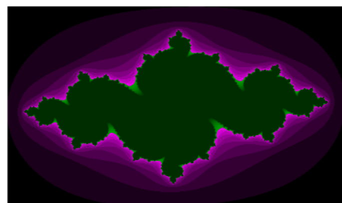


Dessiner des fractales V

- Deux **ensembles de Julia** avec

```
(define (f1 z z0)
  (+ (* z0 z0) -0.75 + 0.0i))

(define (f2 z z0)
  (+ (* z0 z0) -0.75+0.1i))
```

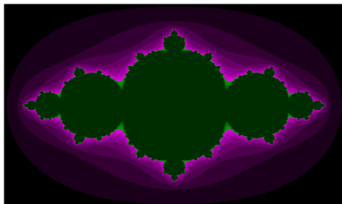


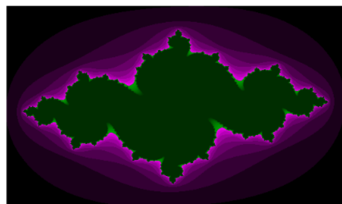


Dessiner des fractales V

- Deux **ensembles de Julia** avec

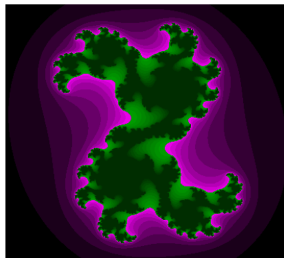
```
(define (f1 z z0)
  (+ (* z0 z0) -0.75 + 0.0i))

(define (f2 z z0)
  (+ (* z0 z0) -0.75+0.1i))
```





- Un **dragon** avec

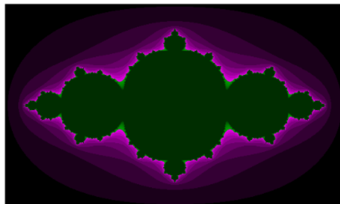
```
(define (f1 z z0)
  (+ (* z0 z0) 0.4-0.192i))
```

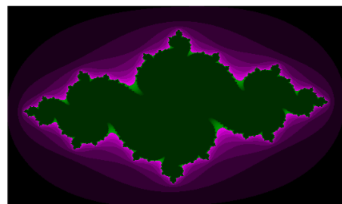


Dessiner des fractales V

- Deux **ensembles de Julia** avec

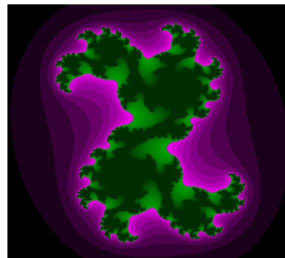
```
(define (f1 z z0)
  (+ (* z0 z0) -0.75 + 0.0i))

(define (f2 z z0)
  (+ (* z0 z0) -0.75+0.1i))
```





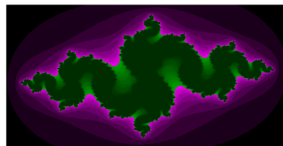
- Un **dragon** avec

```
(define (f1 z z0)
  (+ (* z0 z0) 0.4-0.192i))
```



- Un autre **dragon** avec

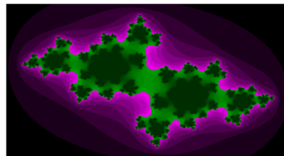
```
(define (f1 z z0)
  (+ (* z0 z0) 0.8+0.168i))
```



Dessiner des fractales VI

- Des **iles** avec

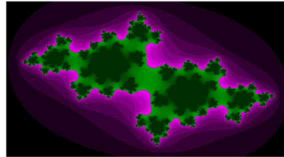
```
(define (f1 z z0)
  (+ (* z0 z0) -0.683-0.408i))
```



Dessiner des fractales VI

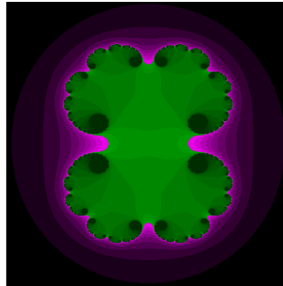
- Des **iles** avec

```
(define (f1 z z0)
  (+ (* z0 z0) -0.683-0.408i))
```



- Un **trèfle** avec

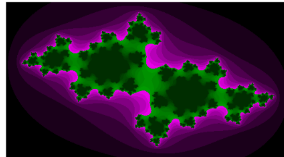
```
(define (f1 z z0)
  (+ (* z0 z0) 0.3))
```



Dessiner des fractales VI

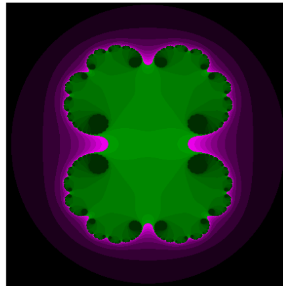
- Des **iles** avec

```
(define (f1 z z0)
  (+ (* z0 z0) -0.683-0.408i))
```



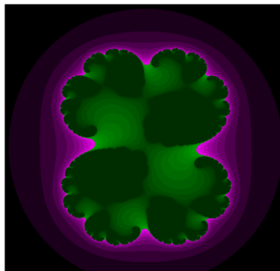
- Un **trèfle** avec

```
(define (f1 z z0)
  (+ (* z0 z0) 0.3))
```



- Un autre **trèfle** avec

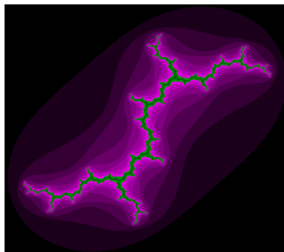
```
(define (f1 z z0)
  (+ (* z0 z0) 0.285+0.01i))
```



Dessiner des fractales VI

- Une **dendrite** avec

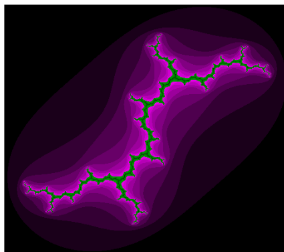
```
(define (f1 z z0)
  (+ (* z0 z0) 0.0+i))
```



Dessiner des fractales VI

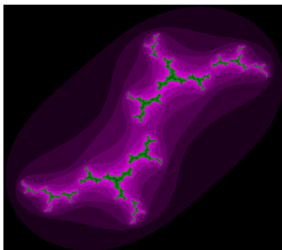
- Une **dendrite** avec

```
(define (f1 z z0)
  (+ (* z0 z0) 0.0+i))
```



- Une autre **dendrite** avec

```
(define (f1 z z0)
  (+ (* z0 z0) 0.01+i))
```



Dessiner avec des tortues I

- Les tortues (package **turtle**, aussi disponible en Python) présentent une interface ludique de dessin au travers d'un animal que l'on va faire se déplacer. Pour utiliser les tortues, il faut utiliser la commande

```
(require graphics/turtles)
```

Dessiner avec des tortues I

- Les tortues (package **turtle**, aussi disponible en Python) présentent une interface ludique de dessin au travers d'un animal que l'on va faire se déplacer. Pour utiliser les tortues, il faut utiliser la commande

```
(require graphics/turtles)
```

- **Fonctions principales :**
  - **(turtles #t)** pour activer et désactiver les tortues;
  - **(draw n)** pour tracer un trait de n pixels dans la direction de la tortue;
  - **(erase n)** pour effacer n pixels;
  - **(move n)** pour déplacer la tortue de n pixels sans tracé;
  - **(turn theta)** pour tourner la tortue d'un angle de θ degrés;
  - **(turn/radians theta)** pour tourner la tortue d'un angle de θ radians;
  - **(clear)** pour tout effacer;
  - **(home)** pour replacer la tortue à sa position initiale;
  - **(save-turtle-bitmap "img.png" "png")** pour sauvegarder le dessin dans le fichier `img.png`.

Dessiner avec des tortues I

- Les tortues (package **turtle**, aussi disponible en Python) présentent une interface ludique de dessin au travers d'un animal que l'on va faire se déplacer. Pour utiliser les tortues, il faut utiliser la commande

```
(require graphics/turtles)
```

- **Fonctions principales :**
  - **(turtles #t)** pour activer et désactiver les tortues;
  - **(draw n)** pour tracer un trait de n pixels dans la direction de la tortue;
  - **(erase n)** pour effacer n pixels;
  - **(move n)** pour déplacer la tortue de n pixels sans tracé;
  - **(turn theta)** pour tourner la tortue d'un angle de θ degrés;
  - **(turn/radians theta)** pour tourner la tortue d'un angle de θ radians;
  - **(clear)** pour tout effacer;
  - **(home)** pour replacer la tortue à sa position initiale;
  - **(save-turtle-bitmap "img.png" "png")** pour sauvegarder le dessin dans le fichier `img.png`.
- Pour simplifier l'écriture, on va définir les fonctions `D`, `M` et `T` correspondant respectivement à `draw`, `move` et `turn`.

```
(require graphics/turtles)
(turtles #t)
(define M move)
(define T turn)
(define D draw)
```

Dessiner avec des tortues II

- La fonction **begin** permet de définir une séquence de fonctions à appeler, permettant ainsi de définir une suite de déplacements.
- **Exemple :**

```
(define (draw-sqr side); side est la longueur
                       ; de tracé souhaitée
                       ; à chaque étape
  (begin (D side)
         (T 90)
         (D side)
         (T 90)
         (D side)
         (T 90)
         (D side)))
```

Ici, la tortue avance de *side*, se tourne de 90°, avance de `side`, se tourne de 90°, avance de `side`, se tourne de 90°, avance de `side`; autrement dit, elle trace un carré de taille `side` à partir de sa position initiale.

Dessiner avec des tortues II

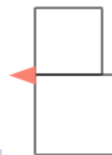
- La fonction **begin** permet de définir une séquence de fonctions à appeler, permettant ainsi de définir une suite de déplacements.
- **Exemple :**

```
(define (draw-sqr side); side est la longueur
                       ; de tracé souhaitée
                       ; à chaque étape
  (begin (D side)
         (T 90)
         (D side)
         (T 90)
         (D side)
         (T 90)
         (D side)))
```

  Ici, la tortue avance de *side*, se tourne de 90°, avance de `side`, se tourne de 90°, avance de `side`, se tourne de 90°, avance de `side`; autrement dit, elle trace un carré de taille `side` à partir de sa position initiale.

- S'ouvre alors une fenêtre dans laquelle apparaît le tracé. On remarque que la tortue est revenue à sa position initiale, mais elle est maintenant dirigée vers le bas, puisque le dernier côté tracé est celui à gauche. Si on trace une nouvelle figure, la tortue démarrera dans cette direction.

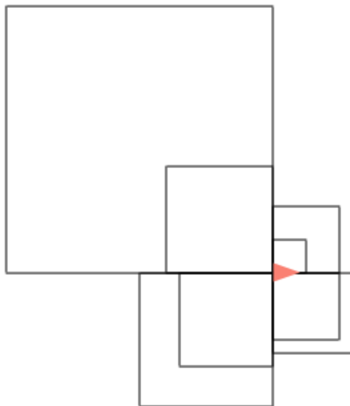
```
(draw-sqr 50)
(draw-sqr 60)
```



Dessiner avec des tortues III

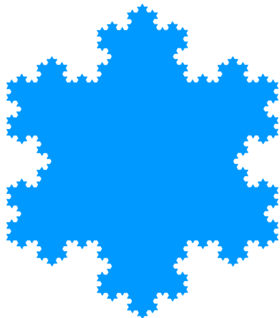
- Si on poursuit l'exemple précédent:

```
(draw-sqr 50)
(draw-sqr 60)
(draw-sqr 70)
(draw-sqr 80)
(draw-sqr 25)
(draw-sqr 50)
(draw-sqr 100)
(draw-sqr 200)
```



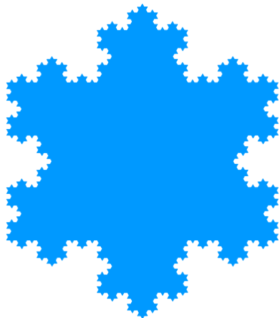
Flocon de Koch

- Le **flocon de Koch** est une fractale qui se construit à partir d'un triangle, en subdivisant les segments en trois morceaux, et en réalisant un pic sur le morceau du milieu.



Flocon de Koch

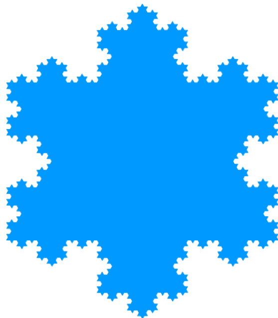
- Le **flocon de Koch** est une fractale qui se construit à partir d'un triangle, en subdivisant les segments en trois morceaux, et en réalisant un pic sur le morceau du milieu.



- Pour tracer le flocon, il faut ainsi dessiner un morceau du segment, tourner de 60°, dessiner un autre morceau, tourner de -120°, dessiner un autre morceau, tourner de 60, et dessiner un dernier morceau.

Flocon de Koch

- Le **flocon de Koch** est une fractale qui se construit à partir d'un triangle, en subdivisant les segments en trois morceaux, et en réalisant un pic sur le morceau du milieu.



- Pour tracer le flocon, il faut ainsi dessiner un morceau du segment, tourner de $60°$, dessiner un autre morceau, tourner de $-120°$, dessiner un autre morceau, tourner de 60, et dessiner un dernier morceau.
- Voici ci-dessous le programme permettant de réaliser cette subdivision en pics:

```
(define (draw-koch side)
  (begin (D side)
         (T 60)
         (D side)
         (T -120)
         (D side)
         (T 60)
         (D side)))
```

Flocon de Koch II

- Puis la fonction permettant d'itérer ce procédé n fois:

```
(define (draw-koch2 side n)
  (local ([define (sub-draw) (draw-koch2 (/ side 2) (- n 1))])
    (if (= n 0)
        (draw-koch side)
        (begin (sub-draw) (T 60)
               (sub-draw) (T -120)
               (sub-draw) (T 60)
               (sub-draw)))))
```

Cas $n = 0$

Cas $n = 1$

Cas $n = 2$ (avec `side` $= 20$)

Flocon de Koch II

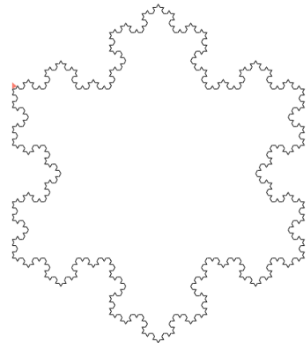
- Puis la fonction permettant d'itérer ce procédé n fois:

```
(define (draw-koch2 side n)
  (local ([define (sub-draw) (draw-koch2 (/ side 2) (- n 1))])
    (if (= n 0)
        (draw-koch side)
        (begin (sub-draw) (T 60)
               (sub-draw) (T -120)
               (sub-draw) (T 60)
               (sub-draw)))))
```

Cas $n = 0$ Cas $n = 1$ Cas $n = 2$ (avec `side` $= 20$)

- Pour dessiner le **flocon de Koch complet**, il faut intégrer le dessin d'un triangle en réutilisant la fonction **draw-koch2**:

```
(define (koch side n)
  (begin (draw-koch2 side n)
         (T -120)
         (draw-koch2 side n)
         (T -120)
         (draw-koch2 side n)))

(M (* -1 (/ turtle-window-size 2)))
(koch 40 3)
```



Interprètes et L-systèmes I

- Les fractales et les L-systèmes (pour *systèmes de Lindenmayer*, modélisant le développement des plantes et des bactéries) ont donné lieu à un système particulier d'écriture de suites de symboles interprétables par des déplacements de tortue.
- Liste des déplacements couramment utilisés:
  - `F` pour avancer d'un pas;
  - `+` pour tourner dans le sens horaire d'un angle prédéfini;
  - `-` pour tourner dans le sens anti-horaire;
  - `|` pour faire demi-tour;
  - `[` pour sauvegarder sa position courante;
  - `]` pour restaurer la dernière position sauvegardée.

Interprètes et L-systèmes I

- Les fractales et les L-systèmes (pour *systèmes de Lindenmayer*, modélisant le développement des plantes et des bactéries) ont donné lieu à un système particulier d'écriture de suites de symboles interprétables par des déplacements de tortue.
- Liste des déplacements couramment utilisés:
  - `F` pour avancer d'un pas;
  - `+` pour tourner dans le sens horaire d'un angle prédéfini;
  - `-` pour tourner dans le sens anti-horaire;
  - `|` pour faire demi-tour;
  - `[` pour sauvegarder sa position courante;
  - `]` pour restaurer la dernière position sauvegardée.
- A partir de ces symboles, deux ensembles définissent les L-systèmes:
  - Une suite initiale de symboles, notée ω;
  - Des règles de réécriture permettant de faire évoluer la suite initiale de génération en génération, notées p.
  - **Exemple :** Le flocon de Koch peut se définir par le L-système suivant:

ω: `F;` p: `F` $\rightarrow$ `F + F - - F + F`.

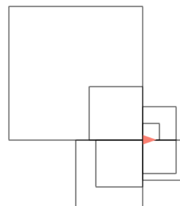
Interprètes et L-systèmes II

- Après génération de la suite de symboles au cran désiré, il faut **interpréter** cette suite en tracés réalisés par des mouvements de tortue.
- On va considérer les éléments à tracer comme des listes d'éléments, définir une fonction **interprete** qui réalise le tracé correspondant à un élément, et une fonction **process** qui appelle la fonction **interprete** pour chacun des éléments de la liste.

Interprètes et L-systèmes II

- Après génération de la suite de symboles au cran désiré, il faut **interpréter** cette suite en tracés réalisés par des mouvements de tortue.
- On va considérer les éléments à tracer comme des listes d'éléments, définir une fonction **interprete** qui réalise le tracé correspondant à un élément, et une fonction **process** qui appelle la fonction **interprete** pour chacun des éléments de la liste.
- Reprenons le premier exemple des carrés successifs: il s'agit d'avoir des listes de taille de côté de carré dont on réalise le tracé en séquence.
  - La fonction **process** prend les éléments de cette liste un à un et réalise les dessins correspondant en appelant la fonction **interprete** à chaque étape.
  - Pour une valeur passée en paramètre, la fonction **interprete** dessine un carré avec cette valeur pour taille de côté.

```
(turtles #t)
(define (interprete d)
  (draw d) (turn 90) (draw d) (turn 90) (draw d) (turn 90) (draw d))

(define (process L)
  (if (= (length L) 1)
      (interprete (car L))
      (begin (interprete (car L)) (process (cdr L)))))

(define L '(50 60 70 80 25 50 100 200))
(process L)
```



Interprètes et L-systèmes III

- Reprenons l'exemple du flocon de Koch, dont le L-système est donné par

$$\omega\text{: } \texttt{F}; \qquad p\text{: } \texttt{F} \rightarrow \texttt{F + F - - F + F}.$$

- On va modifier la fonction **interperete** en passant en paramètre un symbole et une traille de tracé; si le symbole est inconnu (différent de F, + et -), la tortue ne se déplace pas.

```
(define (interprete e d)
  (cond
    ((equal? e #\F) (draw d))
    ((equal? e #\+) (turn 60))
    ((equal? e #\-) (turn -60))
    (else (turn 0))))
```

Interprètes et L-systèmes III

- Reprenons l'exemple du flocon de Koch, dont le L-système est donné par

$$\omega\colon \texttt{F}; \qquad p\colon \texttt{F} \to \texttt{F + F - - F + F}.$$

- On va modifier la fonction **interperete** en passant en paramètre un symbole et une traille de tracé; si le symbole est inconnu (différent de F, + et -), la tortue ne se déplace pas.

```
(define (interprete e d)
  (cond
    ((equal? e #\F) (draw d))
    ((equal? e #\+) (turn 60))
    ((equal? e #\-) (turn -60))
    (else (turn 0))))
```

- On va créer 3 fonctions génératrices **develop**, **develop-elt** et **generate**:
  - **develop-elt** permet de remplacer un caractère F de la suite par la séquence F + F - - F + F;
  - **develop** remplace chaque F de la liste en appelant **develop-elt**;
  - **generate** produit les générations d'indice n, dans une liste que l'on passera en argument de **process**.

```
(define (develop-elt e)
  (cond
    ((equal? e #\F) (list #\F #\+ #\F #\- #\- #\F #\+ #\F))
    (else (list e))))

(define (develop L)
  (if (empty? L)
      L
      (append (develop-elt (car L)) (develop (cdr L)))))
```

Interprètes et L-systèmes III

- Fonctions **generate** et **process**:

```
(define (generate L n)
  (if (= n 0)
      (develop L)
      (develop (generate L (- n 1)))))
```

```
(define (process L d)
  (if (= (length L) 1)
      (interprete (car L) d)
      (begin (interprete (car L) d) (process (cdr L) d))))
```

Interprètes et L-systèmes III

- Fonctions **generate** et **process**:

```
(define (generate L n)
  (if (= n 0)
      (develop L)
      (develop (generate L (- n 1)))))
```

```
(define (process L d)
  (if (= (length L) 1)
      (interprete (car L) d)
      (begin (interprete (car L) d) (process (cdr L) d))))
```

- On démarre alors d'une liste à un élément contenant le symbole `F`, puis on peut générer les listes suivantes avec la fonction **generate**. À chaque génération, on divisera la longueur de tracé par 2 pour avoir un dessin de dimension constante.

```
(move (* -1 (/ turtle-window-size 2)))
(define start (list #\F))
(process (generate start 0) 20)
(process (generate start 1) 10)
(process (generate start 2) 5)
(process (generate start 3) 2)
```