

Le langage Scheme : Niveau Intermédiaire III, dessin

D'après les cours de J.-P. Roy (Nice) et Jerzy Karczmarczuk (Caen)

I. Le principe

On va faire du graphisme en utilisant un objet nommé la Tortue (c.f. Logo), défini par J.-P. Roy¹.

Rappelons au préalable qu'il y a deux types de graphisme :

- Polaire. Il est local, utilisant un repère mobile lié à l'objet virtuel que l'on déplace (la tortue). Le principe du repère mobile est utilisé en géométrie différentielle. Tout déplacement se résume ici à avancer et tourner.
- Cartésien. Il est global, utilisant un repère fixe lié à la fenêtre graphique. On sait dessiner un segment joignant deux points ; des problèmes de paliers vont se poser (algorithmique graphique).

La direction dans laquelle regarde la tortue est le 'cap'. Son état est <position, cap, crayon ?>, avec une variable de position utile pour le graphisme cartésien, et crayon? pour savoir s'il est baissé ou levé.

(forward d) ; avance de d pixels

(back d) ; recul de d pixels

(right a) ; changement de cap : tourne de a degrés sur sa droite

(left a) ; changement de cap : tourne de a degrés sur sa gauche

(pen-up) ; lève le crayon

(pen-down) ; baisse le crayon

(clear) ; efface la fenêtre et retourne au centre, crayon baissé, cap Nord

(tortue-rouge) (tortue-verte) (tortue-bleu) (tortue-neutre) ; change la couleur dominante

(tortue-couleur r g b) ; modifie la couleur de la tortue (on ne prend pas en compte les dominantes)

(tortue-aleaCouleur) ; donne une couleur aléatoire à la tortue en prenant en compte les dominantes

(turtle-position) (t-xcor) (t-ycor) ; position courante (x, y). Utile pour le graphisme cartésien

(turtle-heading) ; la direction (cap) dans laquelle la tortue regarde

(turtle-set-position L) (turtle-set-heading a) (toward point) ; bouger la tortue, modifier son cap

Les procédures graphiques « ne renvoient aucun résultat ». Implicitement, comme toute fonction renvoie malgré tout un résultat, cela signifie qu'elles renvoient (`void`), qui est une valeur de rien, non affichable.

Notons que `void` est la primitive, et donc (`void`) son application : (`void`) ne renvoie pas, `void` est une λ !

On veut faire du graphisme en 'séquentialité' (instructions successives). On définit des formes spéciales :

- (`begin e1 ... en`). Évalue en séquence ($\neq \lambda$!) chacun des arguments (gauche vers droite) et renvoie le résultat du dernier. La forme `begin` est implicite partout, sauf dans un `if`.

- (`do ((x1 i1 s1) ... (xn in sn)) (test? e1 ... en) c1 ... cn`) *Par des macros, ces textes sont équivalents* (`define (iter x1 ... xn) (if test? (begin e1 ... en) (begin c1 ... cn (iter s1 ... sn))) (iter i1 ... in)`)

On pose les variables x_i initialisées à i_i et on décrit leur évolution s_i . Si le test est vérifié, alors on fait les instructions e_i associées et on s'arrête. Sinon à chaque tour on fait les instructions c_i .

- (`repeat n e1 ... ek`). On répète n fois la série d'instructions, et on ne rend aucun résultat.

Prenons l'exemple du dessin d'un polygone régulier à n côtés pour mettre en avant ces fonctions.

(`define (poly n c)`

(`let ((angle (/ 360.0 n)))`

(`define (iter n)`

(`if (zero? n)`

(`void`

(`begin (forward c) (left angle)`

(`iter (- n 1))))`

(`define (poly n c)`

(`let ((angle (/ 360.0 n)))`

(`do ((n n (- n 1)))`

((`zero? n`) (`void`))

(`forward c`)

(`left angle`))))

*do impératif
il y a un corps ici*

(`define (poly n c)`

(`let ((angle (/ 360.0 n)))`

(`repeat n`

(`forward c`)

(`left angle`))))

II. Entraînement à l'utilisation des nouvelles formes spéciales

On définit la factorielle fonctionnelle itérative et la longueur fonctionnelle itérative d'une liste :

```
(define (fac n)
  (do ((n n (- n 1)) (f 1 (* f n)))
      ((zero? n) f)))
(define ($length L)
  (do ((L L (cdr L)) (n 0 (+ n 1)))
      ((null? L) n)))
```

On veut n nombres à partir de x à partir de s (la fonction iota). Si on construit la liste au fur et à mesure, elle sera à l'envers ; on se donne donc la fin de liste pour commencer : (+ (* (- n 1) s) x). Comme on commence de la fin, on prend l'inverse du pas pour progresser.

```
(define ($iota n x s)
  (do ((n n (- n 1)) (L '()) (cons (+ (* (- n 1) s) x) L)))
      ((zero? n) L)))
```

On définit l'intégrale de f sur [a b] avec un pas dx. On se donne une variable de parcours de l'intervalle avec le pas, et un accumulateur. Quand on est arrivé à la borne b, il n'y a qu'à rendre l'accumulateur.

```
(define (integrale f a b dx)
  (do ((x a (+ x dx)) (accu 0 (+ (* (f x) dx) accu)))
      ((< b x) accu)))
```

Calculons la somme des entiers impairs de [1, 100] :

```
(printf "La somme 12 + 32 + ... 992 vaut : ~a\n" ; soit (apply + (map sqr (filter odd? (iota 100 1))))
  (do ((courant 1 (+ courant 2)) (total 0 (+ total (* courant courant)))) ((= 101 courant) total) ))
```

Faisons un do fonctionnel pour inverser une liste (en impérative, il y a du let, un do (), et des set !...):

```
(define (inverser L)
  (do ((courant L (cdr courant)) (inverse '()) (cons (car courant) inverse)))
      ((null? courant) inverse)))
```

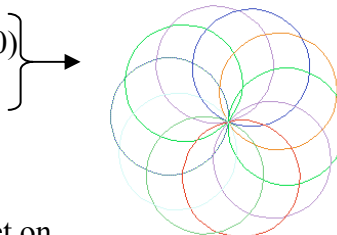
III. Graphisme de base

On a défini précédemment la fonction permettant de tracer un polygone à n côtés de longueur c.

Pour faire un cercle, on utilise un polygone régulier convexe ayant beaucoup de sommets. L'approximation est visuellement correcte à partir de 36 sommets. On peut faire une rosace :

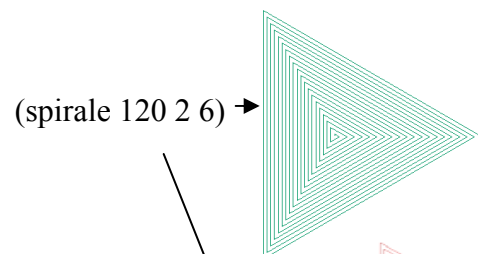
```
(define (rosace)
  (repeat 10
    (tortue-aleaCouleur)
    (poly 36 10)
    (right 36)))
```

- (init '(0 0) 0)
- (rosace)



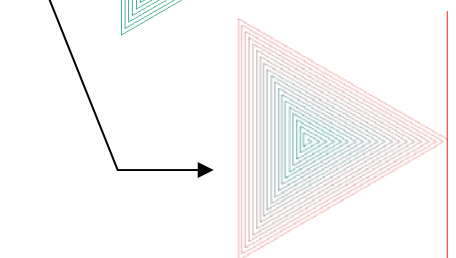
Pour faire une spirale, on part du centre, on tourne et on avance d'un pas un peu plus grand à chaque tour. On s'arrête quand on touche le côté de la fenêtre. D'où :

```
(define (spirale angle c dc)
  (define (iter c)
    (if (and (<= (abs (t-X)) 200) (<= (abs (t-Y)) 200))
        (begin (forward c) (right angle) (iter (+ c dc))))
        (iter c)))
```



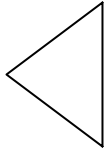
Faisons maintenant un dégradé dans la fonction :

```
(define (spirale angle c dc)
  (define (iter c r) ; on fait varier un des paramètres de la couleur
    (tortue-couleur r 0.5 0.5) ; et on en fixe 0 ou 2
    (if (and (<= (abs (t-X)) 200) (<= (abs (t-Y)) 200))
        (begin (forward c) (right angle) (iter (+ c dc) (+ r 0.015))))
        (iter c)))
```



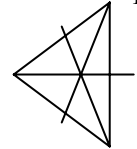
Un grand classique et de construire un triangle équilatéral (de côté c) avec ses bissectrices.

Le raisonnement est graphique : on doit voir comment faire les mouvements et trouver le motif à répéter.



Pour un triangle équilatéral, on avance, on tourne, on avance, on tourne, on avance. Le motif est évident :

(define (equi c) (repeat 3 (forward c) (left 120))) ; chacun angle fait 120°



Pour tracer en plus les bissectrices, on tourne dans la position de la bissectrice, on avance, on recule, et on se remet dans la position pour faire le triangle normal. Il s'agit donc de modifier un peu le code précédent.

(define (equibis c) (repeat 3 (left 30) (forward c) (back c) (right 30) (forward c) (left 120)))

(define (star n c) ; étoile à n branches de côté c

(let ((angle (/ 360.0 n)))

(repeat n

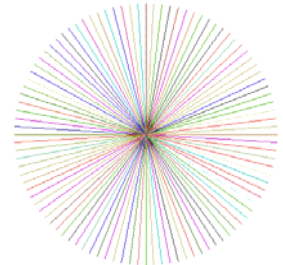
(tortue-aleaCouleur)

(forward c)

(back c)

(left angle))))

(star 100 175) →



IV. Traceur de courbes paramétriques

Une courbe paramétrique $x=f(t)$, $y=g(t)$ modélise un objet qui se déplace avec le temps. Sa position horizontale comme verticale est donc défini par deux fonctions dépendant d'un temps t dans $[a, b]$.

Si on considère que f et g sont continues sur $[a, b]$ sans variations trop brutales, on propose un traceur :

(define (paramplot f g a b zx zy h) ; f g : défini la courbe ; t varie dans [a b] ; zx zy zoome ; h précision

(define (iter x y t Cr Cg Cb) ; je suis sur le point (x,y) au temps t

(if (> t b)

(void)

(let* ((nextTime (+ t h)) (nextX (* (f nextTime) zx)) (nextY (* (g nextTime) zy))) ; h : pas du temps

(tortue-couleur Cr Cg Cb)

(turtle-set-position (list nextX nextY))

(cond ((< Cr 0.99) (iter (+ x h) (+ y h) nextTime (+ Cr 0.01) Cg Cb))

((< Cg 0.99) (iter (+ x h) (+ y h) nextTime Cr (+ Cg 0.01) Cb))

((< Cb 0.99) (iter (+ x h) (+ y h) nextTime Cr Cg (+ Cb 0.01)))

(else (iter (+ x h) (+ y h) nextTime 0.0 0.0 0.0))))))

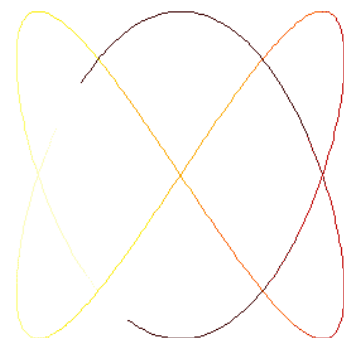
(let* ((x0 (* (f a) zx)) (y0 (* (g a) zy))) ; je pars du point (x0,y0) au temps t0=a

(pen-up)

(turtle-set-position (list x0 y0))

(pen-down)

(iter x0 y0 a 0.0 0.0 0.0)))



Faisons tracer la courbe paramétrique de Lissajou, définie par :

$x = \sin(2t)$

$y = \sin(3t)$

(paramplot (lambda (t) (sin (* 2 t))) (lambda (t) (cos (* 3 t)))

0 (* 2 pi) 130 130 0.01) ; intervalle, zoom et précision

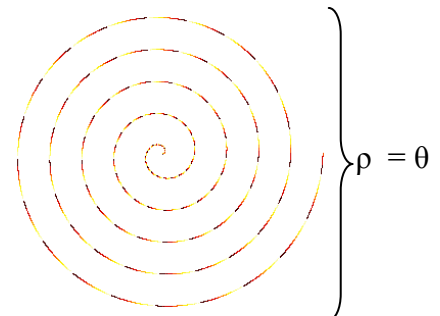
On peut aussi faire des courbes en polaire : $\rho = \rho(\theta)$

$x = \rho \cdot \cos(\theta)$

$y = \rho \cdot \sin(\theta)$

(paramplot (lambda (t) (* t (cos t))) (lambda (t) (* t (sin t)))

0 (* 10 pi) 5 5 0.001)



Enfin, on peut faire tracer de simples courbes de fonction $y = f(x)$.

$x = t$

$y = f(t)$

V. Intégration d'un élément d'arc le long d'une courbe

On veut calculer la longueur d'une courbe paramétrique $x=f(t)$, $y = g(t)$.

On intègre l'élément d'arc ds le long de la courbe T :

$$L = \oint_T ds \quad \text{avec} \quad ds = \sqrt{dx^2 + dy^2}$$

Or $dx = f'(t)dt$ et $dy = g'(t)dt$ d'où $ds = \sqrt{f'(t)^2 + g'(t)^2} dt$

On se ramène donc à un calcul d'intégrale :

$$L = \int_a^b \sqrt{f'(t)^2 + g'(t)^2} dt \quad (\text{pour un cercle, pas pour une ellipse !})$$

La théorie des intégrales elliptiques permet de calculer la longueur d'une ellipse.

On va effectuer un calcul discret en remplaçant l'intégrale par une sommation, en découpant le temps en tranches de largeur dt . On rappelle au passage la formule pour calculer la dérivée :

(define (derivee f x h) (/ (- (f (+ x h)) (f x)) h)) ; la formule est $f(x+h) - f(x)$, divisé par le pas h

(define (longueur f g a b h) ; rectification d'un arc paramétrique

(let* ((fp (lambda (t) (derivee f t 0.01))) ; derivation numérique !

(gp (lambda (t) (derivee g t 0.01))) ; derivation numérique !

(phi (lambda (t) (sqrt (+ (sqr (fp t)) (sqr (gp t)))))) ; la fonction à intégrer

(integrale phi a b h)))

(printf "La longueur du cercle de rayon 1 est (* 2 pi) : ~a\n" (longueur cos sin 0 (* 2 pi) 0.01))

→ 6.289973791699388 (calcul de π par intégration)

VI. Une introduction aux fractales

Une courbe fractale est une courbe invariante d'échelle (isomorphe à chaque morceau). Si l'on zoome sur le voisinage de l'un de ses points, on retrouve la courbe dans son ensemble.

Un exemple simple est la courbe de Von Koch (1904), qui est un exemple de courbe continue dérivable en aucun point et obtenue par une construction géométrique élémentaire.

La programmation se fait par récurrence sur le niveau $n \geq 1$:

- au niveau 1, un segment
- une courbe de niveau $n > 1$ est constituée de 4 courbes de niveau $n - 1$

(define (VK taille niveau) ; taille = longueur de la base

(if (= niveau 1)

(forward taille)

(begin (VK (/ taille 3) (- niveau 1))

(left 60)

(VK (/ taille 3) (- niveau 1))

(right 120)

(VK (/ taille 3) (- niveau 1))

(left 60)

(VK (/ taille 3) (- niveau 1))))))

(define (FVK taille niveau)

(do ((n 3 (- n 1)))

((zero? n) (void))

(VK taille niveau)

(right 120)))

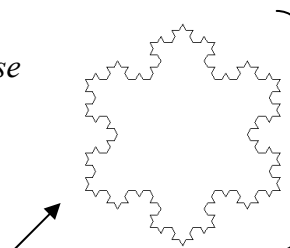
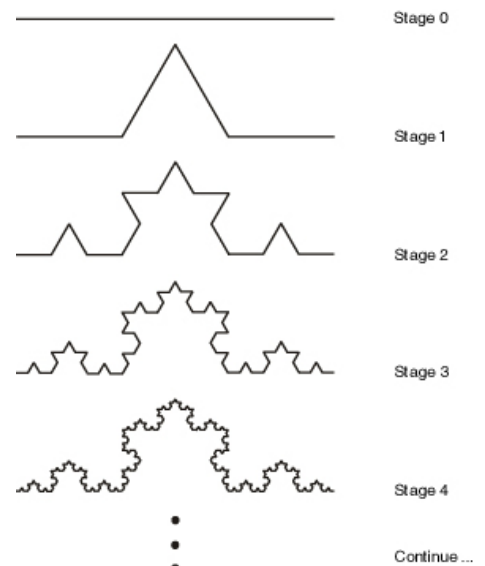
Le flocon fractal de Von Koch s'obtient en greffant une courbe de Von Koch sur chaque côté d'un triangle équilatéral.

(define (FVK taille niveau)

(repeat 3

(VK taille niveau)

(right 120)))



Mandelbrot ('70) a proposé la formule ci-contre pour la dimension fractale : $D = (\log N) / (\log 1/k)$, où :

- chaque segment est remplacé par N sous-segments
- le rapport des mesures entre un segment et le segment précédent est égal à k (facteur de zoom)

Pour la courbe de Von Koch, $N = 4$ et $k = 1/3$ donc $D = (\log 4) / (\log 3) = 1.26\dots$

Pour un segment usuel, $N = 2$ et $k = 1/2$ donc $D = 1$. Un segment est bien une courbe de dimension 1.

Une courbe est fractale lorsque sa dimension D est $1 < D \leq 2$! (c.f. topologie)

La courbe limite (niveau infini de la courbe de Von Koch) est telle que sa longueur est infinie, mais elle entoure une surface d'aire finie. En effet $4/3 > 1$ donc la suite géométrique de raison > 1 diverge.

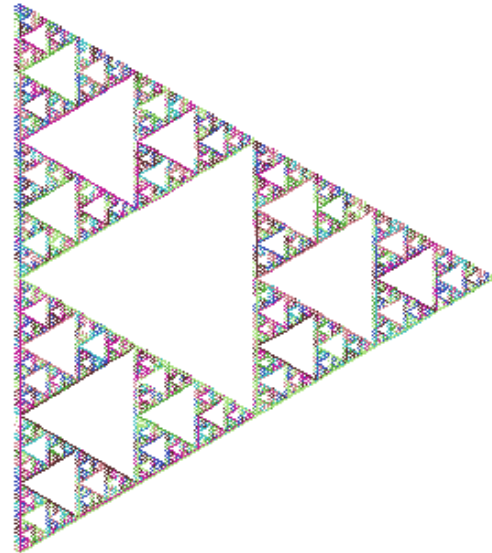
`(define (trif n s) ; triangle de sierpinski`

```

(if (zero? n)
    (poly 3 s)
    (begin
      (trif (- n 1) (/ s 2))
      (forward (/ s 2))
      (trif (- n 1) (/ s 2))
      (right 120)
      (forward (/ s 2))
      (left 120)
      (trif (- n 1) (/ s 2))
      (right 60)
      (back (/ s 2))
      (left 60))))

```

`(init '(-150 -150) 0)`
`(trif 8 400) →`



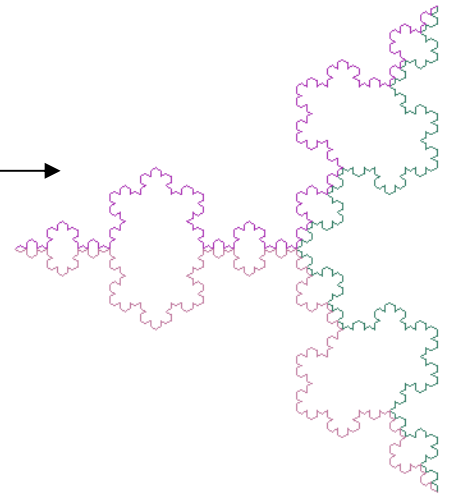
L'anti-flocon de Von Koch s'obtient en faisant rentrer la courbe à l'intérieur du triangle.

```

(define (AFVK taille niveau)
  (do ((n 3 (- n 1)))
      ((zero? n) (void))
      (VK taille niveau)
      (left 120)))

```

`(init '(160 -190) 0)`
`(AFVK 380 5) →`



La fractale du dragon est telle que :

- le premier niveau est un segment AB de longueur T
- le second niveau est la lignée brisée ACB avec $AB = T$, $AC = AB$ et il y a un angle droit en A.
- le processus est itéré sur chacun des sous-segments AC et CB
- le niveau n comporte donc 2^{n-1} segments, dont certains sont confondus

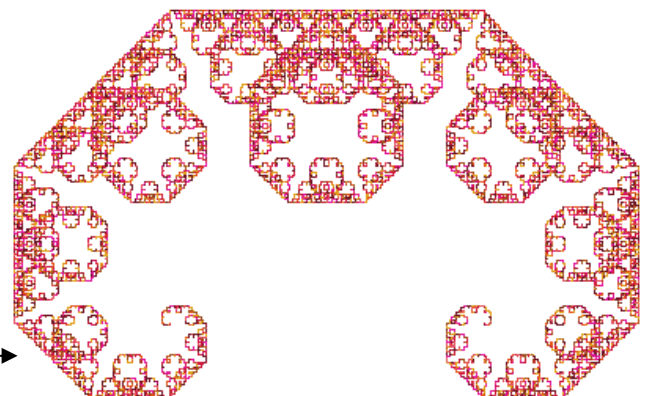
`(define (fractale taille n) ; taille : distance en ligne droite du point de départ A au point d'arrivée B`

```

(if (= n 1)
    (forward taille)
    (begin
      (tortue-aleaCouleur)
      (left 45)
      (fractale (sqrt (/ (sqr taille) 2)) (- n 1))
      (right 90)
      (fractale (sqrt (/ (sqr taille) 2)) (- n 1))
      (left 45))))

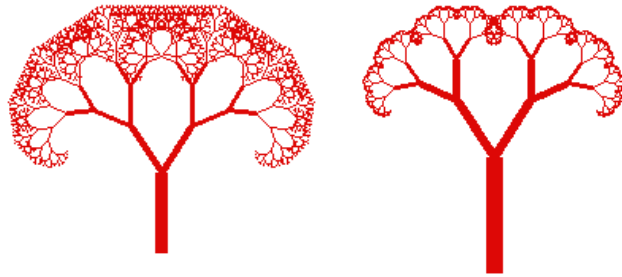
```

`(init '(-100 -50) 90) (fractale 200 15) →`



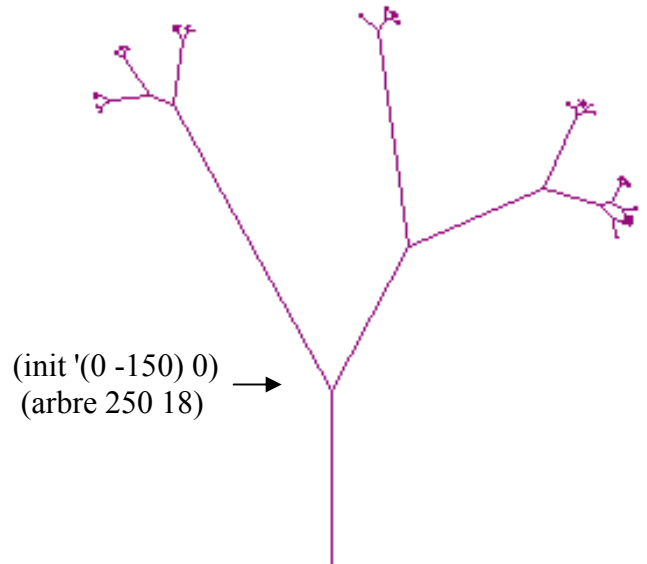
VII. Arbres fractals

On considère qu'un arbre est au départ un tronc, soit une ligne droite. Au niveau suivant, son tiers se divise en deux, etc. etc. On a quelque variante selon l'angle.



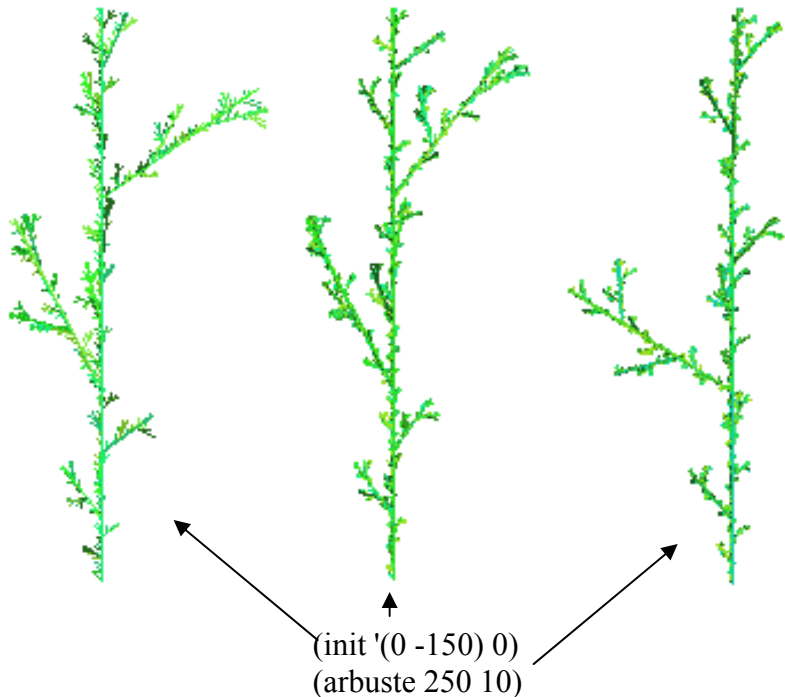
Pour obtenir un arbre plus réaliste, il faut insuffler « du vent » : de l'aléatoire.

```
(define (arbre taille n) ; n : niveau
  (if (= 1 n)
      (begin
        (forward taille)
        (back taille))
      (begin
        (let ((vent (+ 25 (srandom 25))))
          (tronc (/ (+ 3 (srandom 7)) 10)))
          (forward (* taille tronc))
          (left vent)
          (arbre (* taille (- 1 tronc)) (- n 1))
          (right (* vent 2))
          (arbre (* taille (- 1 tronc)) (- n 1))
          (left vent)
          (back (* taille tronc)))))))
```



Il est plus réaliste d'expérimenter des feuillages moins symétriques, comme l'arbuste fractal :

```
(define (arbuste taille n)
  (let ((mouvement (random 60)) (aleaSup (random 60)))
    (if (= n 1)
        (begin
          (left mouvement)
          (forward taille)
          (back taille)
          (right mouvement))
        (begin
          (tortue-aleaCouleur)
          (arbuste (/ taille 3) (- n 1))
          (forward (/ taille 3))
          (left mouvement)
          (arbuste (/ taille 3) (- n 1))
          (right mouvement)
          (tortue-aleaCouleur)
          (arbuste (/ taille 3) (- n 1))
          (forward (/ taille 3))
          (right aleaSup)
          (arbuste (/ taille 3) (- n 1))
          (left aleaSup)
          (arbuste (/ taille 3) (- n 1))
          (forward (/ taille 3))
          (back taille))))))
```



i

¹ Turtlegraphics pour DrScheme v209 par Jean-Paul Roy, Janvier 2005. Amélioré avec l'ajout de la couleur et des dominantes par Philippe Giabbanelli et Thomas Strangi, version Greatturtle, Mars 2006.