Generative Art with Context Free Grammars

Leon Rische

[2019-07-11 Thu 12:00]

Contents

The Chomsky hierarchy defines four levels of formal grammars:

1. Unrestricted (most powerful)
2. Context Sensitive
3. Context Free
4. Regular (simplest)

Usually, the problem is checking whether a word belongs to a grammar.

Here, we only care about generating random expansions of a grammar. For this CFGs are a good choice because random expansion is easy to implement and they are not as limited as regular grammars.

If you're not familiar with the concept of context free grammars, read Context Free Grammars first.

Terminals of a grammar don't have to be characters, they can be (interpreted as) anything at all, for example functions that draw geometric shapes.

1 Turtle Grammars

One easy way to visualize the output of the expansion of a grammar is to interpret it as instructions for a turtle-graphics program.

For the following examples, I'm using only three commands:

1. move forward
2. turn left / right
3. scale (reduce the size of each forward step)

The expansions of grammars can be infinitely long, so we need a mechanism to prevent this from happening. I'm simply limiting the depth of the expansion to some value, after that all non-terminals are ignored.

1.1 Iteration

Let's start with a simple turtle rule system, to see how iteration can be implemented:

language=Python,label= ,caption= ,captionpos=b,numbers=none grammar.add$_n on_t erminal('S', [l$
$turtle.forward(), lambdaturtle : turtle.turn_r ight(5), lambdaturtle : turtle.scale_b y(0.99),' S'])$

grammar.expand('S', turtle, 300)

For the examples, I'll be using a small python framework I've created instead of mathematical notation. If you want to follow along, download the code from `https://github.com/l3kn/generative_cfg`.

The main element of this framework is a `Grammar`. Non-terminals are just lists of either strings to reference another non-terminal or (anonymous) functions as terminals.

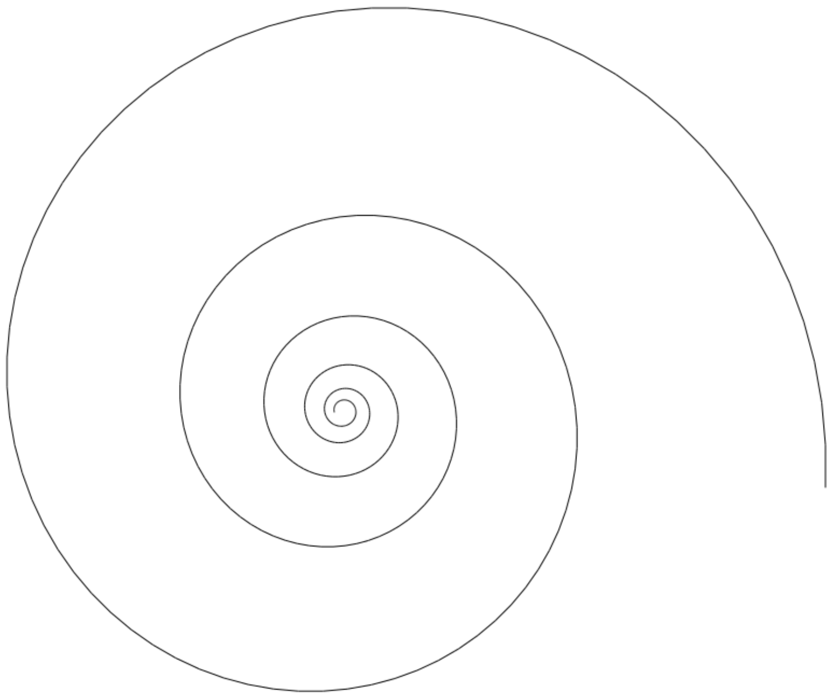
These terminal functions can act on some kind of state, in this case a turtle graphics object `turtle`.

Using `grammar.add_non_terminal(name, body)` new non-terminals can be added to the grammar.

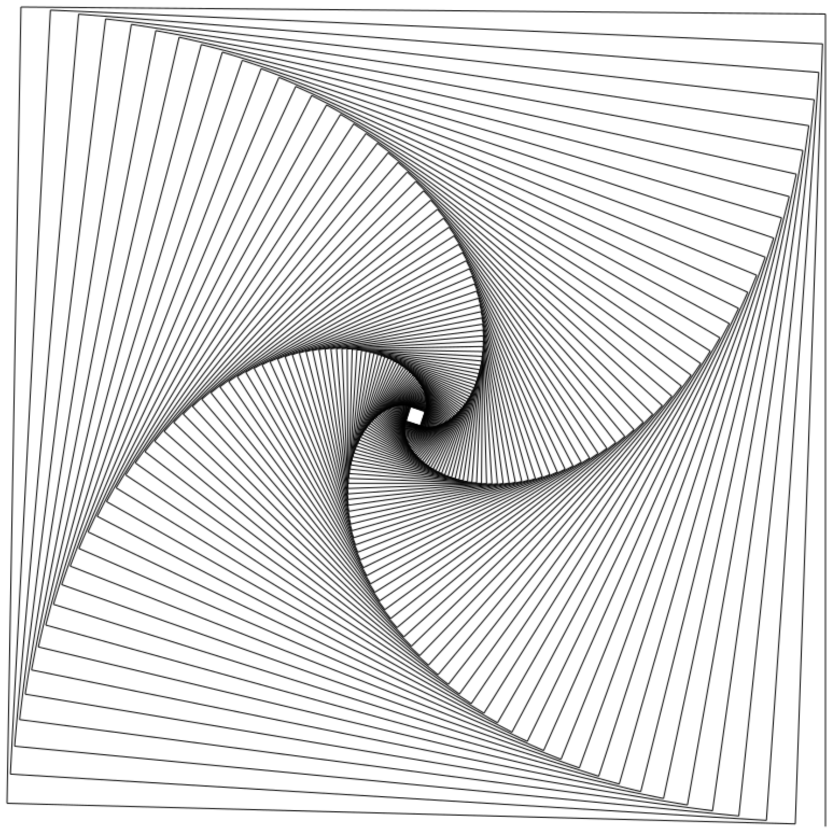
A grammar can be expanded using the method `grammar.expand(start_symbol, state, depth)`.

Now it should be clear how the code above produces the image below, the non-terminal S makes the turtle move forward, turns it to the right by 5 degrees, and then reduces its scale by a factor of 0.99.

Then the grammar is expanded to a depth of 300 using S as start symbol.



Changing the rotation angle from 5 to 89.5 degrees leads to a very different result



1.2 Branching 1

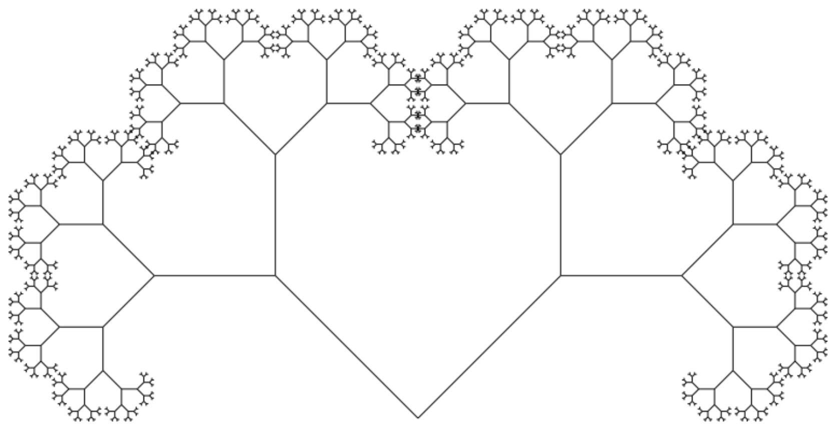
language=Python,label= ,caption= ,captionpos=b,numbers=none from generative$_{c}fgimport*$

backend = SVGBackend() turtle = Turtle(backend) grammar = Grammar() scale = 0.6

grammar.add$_{n}on_{t}erminal('Tree', [lambdat : t.store(), lambdat : t.turn_{r}ight(45), lambdat : t.forward(), lambdat : t.scale(scale), 'Tree', lambdat : t.restore(), lambdat : t.turn_{l}eft(45), lambdat : t.forward(), lambdat : t.scale(scale), 'Tree',])$

grammar.expand('Tree', turtle, 10) backend.write('out.svg')

"Tree" draws a left “branch” and then another smaller "Tree", restores the turtles state and then draws a right “branch” and a second smaller "Tree".



By adding two or more references to the rule into its RHS, we can generate all kinds of branching patterns.

Because we want the turtle to draw the left and right branches starting form the same position, we need a way to store and restore its state (instead of walking it back each time).

In the python framework, I've implemented this by pushing and popping the current position, direction and scale from a stack stored in the turtle object.

1.3 Branching 2

The tree above doesn't look very "natural", real trees don't grow at straight angles.

By making a small step forward, then turning the turtle multiple times, we can draw smooth arcs.

Python's "array multiplications" (`[1,2] * 3 # => [1,2,1,2,1,2]`) can be used to save some work here.

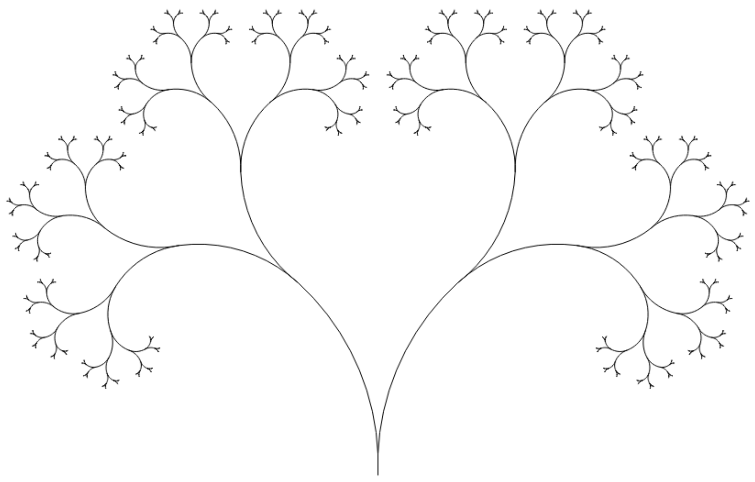
language=Python,label= ,caption= ,captionpos=b,numbers=none arc$_{r}ight_{s}egment = [lambdaturtle : turtle.forward(0.1), lambdaturtle : turtle.turn_{r}ight(5)]arc_{l}eft_{s}egment = [lambdaturtle : turtle.forward(0.1), lambdaturtle : turtle.turn_{l}eft(5)]$

grammar.add$_{n}on_{t}erminal('ArcLeft', arc_{l}eft_{s}egment*10)grammar.add_{n}on_{t}erminal('ArcRight', a$
10)

scale = 0.6 grammar.add$_{n}on_{t}erminal('Tree', [lambdaturtle : turtle.store(),' ArcLeft', lambdaturtle$
$turtle.scale_{b}y(scale),' Tree', lambdaturtle : turtle.restore(),' ArcRight', lambdaturtle :$
$turtle.scale_{b}y(scale),' Tree',])$

grammar.expand('Tree', turtle, 10)

There is no need to wrap the second part in `turtle.store()` and `turtle.restore()` because we don't care where the turtle ends up after expanding the last part. (This is similar to tail-call-optimization for recursive functions).

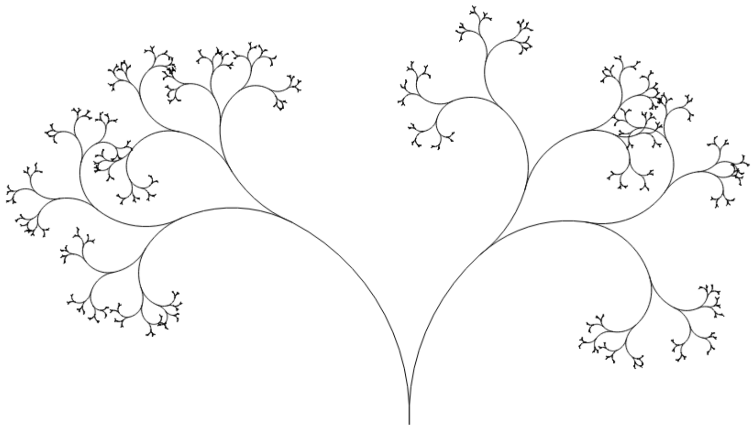


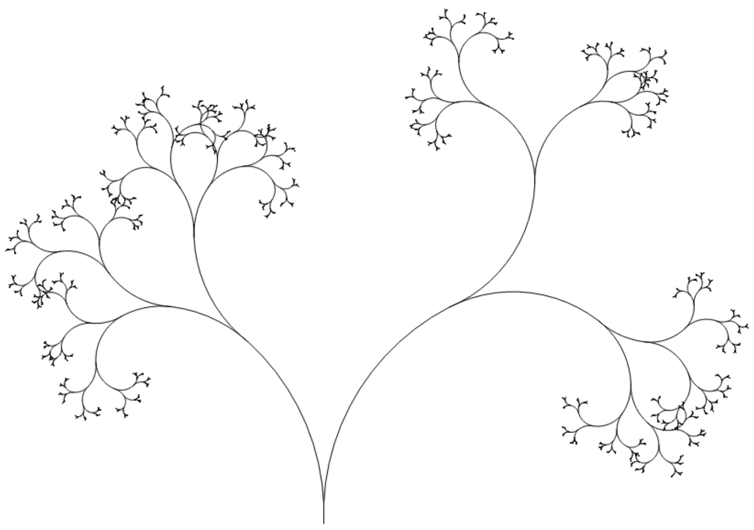
1.4 Randomness

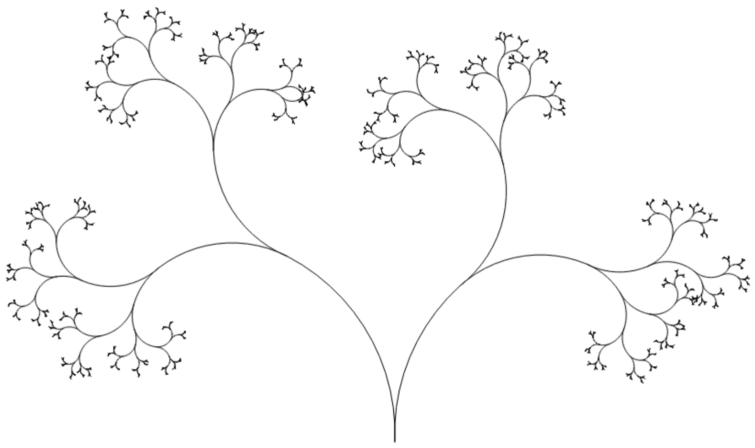
In the introduction, we saw that grammars can have multiple non-terminals with the same name. We can use this to introduce an element of randomness into the generated images.

language=Python,label= ,caption= ,captionpos=b,numbers=none grammar.add$_n on_t erminal('Arc$
$7) grammar.add_n on_t erminal('ArcLeft', arc_l eft_s egment*10) grammar.add_n on_t erminal('ArcLeft', arc$
$13) grammar.add_n on_t erminal('ArcRight', arc_r ight_s egment*7) grammar.add_n on_t erminal('ArcRight',$
$10) grammar.add_n on_t erminal('ArcRight', arc_r ight_s egment * 13)$

By randomly using shorter or longer arcs, a different tree is generated each time the script is run.







2 Evolving Rule Sets

Time to explore a different direction from the same starting point, to see how complex grammars can be built with small steps.

2.1 Starting Point

Make the branches straight at an 60deg angle, and grow three "trees" starting from a root in the center.

language=Python,label= ,caption= ,captionpos=b,numbers=none grammar.add$_n on_t erminal('Bra$
$turtle.forward(1.0), lambdaturtle : turtle.turn_r ight(60)])grammar.add_n on_t erminal('BranchLeft', [la$
$turtle.forward(1.0), lambdaturtle : turtle.turn_l eft(60)])$

grammar.add$_n on_t erminal('Root', ['Tree', lambdaturtle : turtle.turn_l eft(120),' Tree', lambdaturtle$
$turtle.turn_l eft(120),' Tree',])$

scale = 0.5 grammar.add$_n on_t erminal('Tree', [lambdaturtle : turtle.store(),' BranchLeft', lambdatu$
$turtle.scale(scale),' Tree', lambdaturtle : turtle.restore(), We can' tTCOanymorelambdaturtle :$
$turtle.store(),' BranchRight', lambdaturtle : turtle.scale(scale),' Tree', lambdaturtle :$
$turtle.restore(),])$

[width=.9]images/generative$_a rt_w ith_c fgs/flower$

2.2 Randomly Terminating Branches

... by adding an empty "Tree" non-terminal
language=Python,label= ,caption= ,captionpos=b,numbers=none grammar.add$_n on_t erminal('Tre$
$turtle.store(),' BranchLeft', lambdaturtle : turtle.scale(scale),' Tree', lambdaturtle :$
$turtle.restore(), We can' tTCOanymorelambdaturtle : turtle.store(),' BranchRight', lambdaturtle :$
$turtle.scale(scale),' Tree', lambdaturtle : turtle.restore(),], 8) grammar.add_n on_t erminal('Tree', [])$

[width=.9]images/generative$_a rt_w ith_c fgs/flower1_0 0$

[width=.9]images/generative$_a rt_w ith_c fgs/flower1_0 1$

2.3 Randomized Scaling

Now extract a "Scale" non-terminal that scales the turtle only 50% of the time.
language=Python,label= ,caption= ,captionpos=b,numbers=none scale
= 0.5 grammar.add$_n on_t erminal('Scale', [lambdaturtle : turtle.scale(scale),]) grammar.add_n on_t ermin$

[width=.9]images/generative$_a rt_w ith_c fgs/flower2_0 0$

[width=.9]images/generative$_a rt_w ith_c fgs/flower2_0 1$

[width=.9]images/generative$_a rt_w ith_c fgs/flower2_0 2$

[width=.9]images/generative$_a rt_w ith_c fgs/flower2_0 3$

2.4 Thick Lines

Using a `ThickTurtle`, we can draw lines with different widths
language=Python,label= ,caption= ,captionpos=b,numbers=none from generative$_c fgimport*$
...
backend = SVGBackend() turtle = ThickTurtle(backend) turtle.thickness = 0.05
grammar.expand('Root', turtle, 12)

and modify scale to adjust the turtles scale, too.

language=Python,label= ,caption= ,captionpos=b,numbers=none scale = 0.5 grammar.add$_n on_t erminal('Scale', [lambdaturtle : turtle.scale(scale), lambdaturtle : turtle.scale_t hickness(0.6),])grammar.add_n on_t erminal('Scale', [])$

[width=.9]images/generative$_a rt_w ith_c fgs/flower3_0 0$

[width=.9]images/generative$_a rt_w ith_c fgs/flower3_0 1$

[width=.9]images/generative$_a rt_w ith_c fgs/flower3_0 2$

[width=.9]images/generative$_a rt_w ith_c fgs/flower3_0 3$

2.5 Decoration

To make the result more visually interesting, we can add random "decoration" elements to the grammar.

language=Python,label= ,caption= ,captionpos=b,numbers=none grammar.add$_n on_t erminal('Circ$
$turtle.draw_f illed_c ircle(0.2),])grammar.add_n on_t erminal('Circle', [lambdaturtle : turtle.draw_c ircle(0.2),], 2)grammar.add_n on_t erminal('Circle', [], 8)$

language=Python,label= ,caption= ,captionpos=b,numbers=none grammar.add$_n on_t erminal('Tree$
$turtle.store(), 'BranchLeft', 'Scale', 'Circle', 'Tree', lambdaturtle : turtle.restore(), lambdaturtle : turtle.store(), 'BranchRight', 'Scale', 'Circle', 'Tree', lambdaturtle : turtle.restore(),], 8)$

[width=.9]images/generative$_a rt_w ith_c fgs/flower4_0 0$

[width=.9]images/generative$_a rt_w ith_c fgs/flower4_0 1$

[width=.9]images/generative$_a rt_w ith_c fgs/flower4_0 2$

[width=.9]images/generative$_a rt_w ith_c fgs/flower4_0 3$

[width=.9]images/generative$_a rt_w ith_c fgs/flower4_0 4$

3 Further Work

There are a bunch of things that can be done to extend the ideas presented here.

A simple change would be to add a “finally” function to rules that is executed once the expansion depth has reached some limit. As far as I can tell, this would be equivalent to context-free L-Systems and can be used to draw all kinds of fractals and plants.

The turtle graphics implementation used here is very limited and it would be easy to extend it to support drawing circles, polygons or thick lines.

Another direction could be to implement different kinds of states (besides turtle graphics) that the grammar can operate on. (Hint: Quadtrees and Octrees are one possibility).

If you have any questions about how this works, how to implement it the programming language of your choice, or if you want to share an image you generated, feel free to contact me at `cfg <at> leonrische.me`

4 References

Turtle Graphics are a great way to teach programming to children, if you are interested in that angle, I recommend the book "Mindstorms: Children, Computers and Powerful Ideas" by Seymour Papert.

For some background on the LOGO language and its history, visit `https://el.media.mit.edu/logo-foundation/what_is_logo/index.html`

With a few small changes, the context free grammars described in this article can be expanded to context-free L-Systems which can be used to model the growth of different kinds of plants. The book "The Algorithmic Beauty of Plants" is a great introduction into this topic and can be downloaded for free at `http://algorithmicbotany.org/papers/`.

Context Free Art is a program generates (colored) images from CFGs, the website contains a large collection of example programs / images.