Fractals and the Beauty of Nature
DM550 - Fall Project 2017

Chanthosh Sivanandam
Erik Andersen
Henrik Flindt

October 27, 2017

Contents

1 Introduction

While the description of a tree, triangle or a leaf is easily communicated among humans, as most humans share a common cultural understanding challenges arise when trying to convey the same idea to a computer. The task assigned in DM500 requests that a script be written which can draw different figures based upon the Fractal Description Language (FDL) in the programming language, Python, by using the Turtle module. The assignment is separated into several obligatory tasks, divided into two sections; "Fractals and the Beauty of Nature" and "Fractal Description Language". The enterity of the source code, project files and figures can be found at https://github.com/ErikAndersen81/DM550-FractalProject

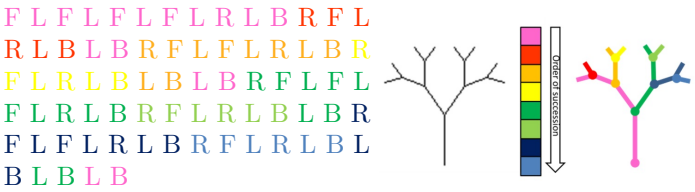
2 Understanding recursion

As an illustrative explanation of recursions, a binary-tree of a depth of four, as seen in figure 1. The rule for the binary-tree is written as FLXRXLB, with F indicating a forward movement, B for a backward movement, and R and L as a right or left turn respectively. Each X indicates where each iteration (i.e. depth) would insert the rule. Table 1 shows the four iterations from depth one to four.

Table 1: Rules for the binary tree

Depth	**Rule**
One	F L R L B
Two	F L F L R L B R F L R L B L B
Three	F L F L F L R L B R F L R L B
	L R F B L F L R L B R F L R L
	B L B L B
Four	F L F L F L F L R L B R F L R
	L B L B,R F L F L R L B R F L
	R L B L B L B R F L F L F L R
	L B R F L R L B L B R F L F L
	R L B R F L R L B L B L B L B

It should be noted that in the assignment, specifications for scaling of the length are included, but they have been removed from this example for a simple and smooth explanation. The colors in figure 1 indicate the succession of when a line is drawn. It must be stressed that the colors do not show iterations, but first passed line in a given direction, so if two or more iterations results in a line being drawn twice, only the first passing line is drawn in the figure.



(a) Color coded FDL-rule (b) Color code applied to the binary tree

Figure 1: Binary tree with depth 4

3 Utilizing FDL in the assignment

The difference between one binary-tree of depth four and one binary-tree of depth 100 is only the length of the FDL and subsequent size of the tree, but the main mechanism remains unchanged. The same mechanism would by modifying the rule, the amount of turns available, and the angle of the turns would be able to draw any figure which is shown with the Sierpinski Triangle and fern. In this assignment additional parts have been added to the FDL mechanism that are not related to drawing lines in the Python-Turtle program, but are designed to reduce the time drawing the figure. These extra lines would in the context of the binary-tree in figure 1 as an example enable the FDL reader to jump from the end of the red line directly to the start of the orange, the end of the yellow to jump to the green line and so forth. These jumps would either be by moving the Turtle-drawer directly or by cloning the drawing instance of Turtle to the needed point.

4 Trial and error

While the final product shows a fully functional FDL readers with extra functions, minor errors in the FDL rule written would sometimes result in rather interesting figures. Figure 2 show a first attempt at a fern. While this is not a ideal fern, there are several signs that the underling FDL rule are on track since there are recursions in the structure and the bend is present.

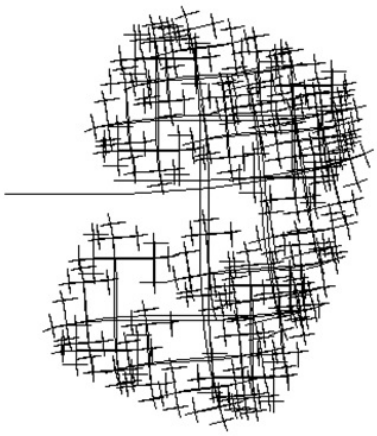


Figure 2: Unsuccessful attempt at drawing the fern

The first attempt took depth as a value that was transmuted in to a length that the tree and branches should cover. (In context of figure 1 this would be the length of the pink line.) While this in theory would work, the math needed to ensure that the scaling would result in a tree with the correct length (len) at any given depth (x) were not found. (See figure 3 for an impression of one of the failed implementations.) The main problem were a continuous mismatch between requested depth and the printed depth of the binary-tree.

```
x = (input('Enter depth of tree:'))
if x.isalpha():
    exit('That is not a number!')
x = int(round(float(x)))
```

Figure 3: A simple safeguard

The problem was overcome by adding depth as the secondary input factor in the definition of the object tree. (See figure 4) Length were calculated based on the request in the assignment that a tree of the depth 10 should have the length 100. The depth value x now functioned as the iteration counter while length were scaled accordingly to the branch.

```
import math
r = 0.7
s = 5
len = 10 ** (math.log10(s) / x - math.log10(r)) * 5
```

Figure 4: Hard math

The initial attempt at drawing a fern ran in to some problems in regards to hardware limitations. While the correctness and cleanliness of the code was unquestionable, the algorithm consumed incredible amounts of ram due to several factors. One of the factors was the tremendous amounts of recursive calls. An explanation hereof is, that every time a function is called in python, the interpreter allocates memory in order to store the variables in that scope. Calling a function recursively prohibits the interpreter from freeing the allocated memory from the stack until the final function has returned its value. Furthermore, and perhaps even more malignant in terms of eating ram, was the fact that the implementation made a clone of the instance of the turtle class every time there was a new branch. While this made some clean and beautiful code, an instance of the turtle class takes up a lot of memory. Some tests showed that running the algorithm for about five to ten minutes would easily take up 30 GB of ram. On the other hand, if depth and segments where kept to a minimum, the algorithm would present a nice looking fern, yet somewhat scrawny.

5 Binary tree

A simple binary tree algorithm were written[1], which were developed to have an option to select depth. Each time a value was calculated print statements were added for that value to ensure that troubleshooting were more easily done. A safeguard were added to the input value x to ensure that depth would be an integer, which then would be passed down through the code. The safeguard checked for strings, and force exits if one is found. Subsequently, it rounds floats and converts them to integers. (Fig. 3) Following Turtle is imported and a few definitions of Turtle functions are written. Minor if statements with a print statements were added, and some modifications form the Turtle package were added for aesthetics reasons and to ensure a proper user experience. A safety question was implemented if a depth of above 12 were requested by the user. This is due to the time it takes for the tree to print, but it can be overwritten by the uses if needed. Finally the scaffolding was removed and the script was tested for a range of depth. A binary tree of depth 10 can be seen in fig. 6.

For this version of the binary-tree no limit for the size of the branches were implemented, since the uses might want all the branches generated, and no knowledge of the users resolution can be know in advanced. Testing the binary tree was done to ensure a proper output. Different values for angels in the turns, and scales were tested, and depths from 1-50 were input. See figure 5 for four trees of depth 10, with l(25)r(60), l(45)r(80), scale 0.2 and scale 0.9. It becomes quite obvious that the predetermined values given in the assignment are optimal to render pretty binary tree as seen in figure 6, when comparing this to figure 5.

[1]The source code for this part of the project can be found in the file binary-tree.py or Appendix A.1

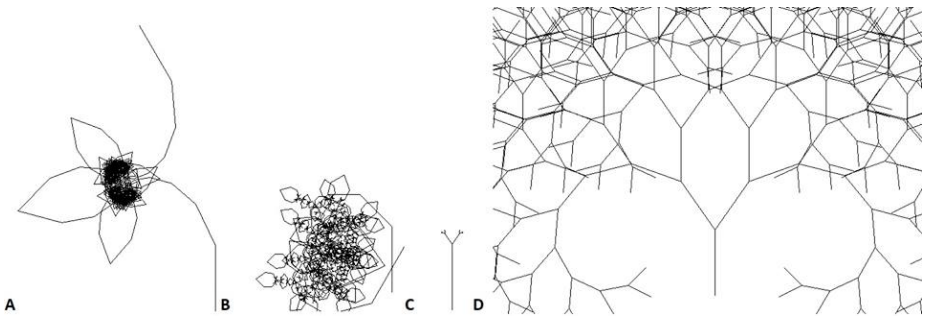


Figure 5: Weird trees

At the level of 12 the rendering time began to increase to a point where it became inconveniently to render. A second problem arose for depth of 20 plus since the crown of the tree would tend to leave the screen. A quick fix to this was implemented with some if-statements asking for a depth cap at 12. The cap is still at the user's prerogative, so a tree with a large depth can still be generated if so required. An alternative fix would have been to change the length of the tree when the depth passed a certain value, but this will first be implemented in a later patch. ”

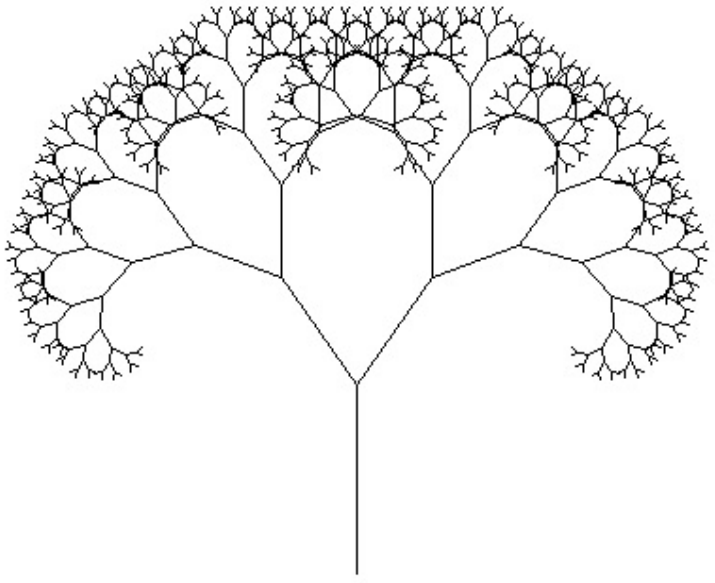


Figure 6: A beautiful binary tree

6 Sierpinski Triangle

Initially, implementation of the Sierpinski triangle was build upon the idea of placing inverted triangles inside other triangles. It turned out to be a tad complicated, but it yielded some rather interesting results, which can be seen in 8a and 8b

(a) An unsuccessful attempt (b) Another unsuccessful attempt

Figure 7: Not Sierpinski's triangles

After meddling around for some time, a decision was made to try the approach suggested in the project description. A successful algorithm that gave the expected result was then rapidly developed.

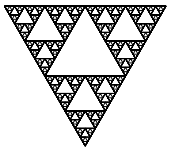


Figure 8: Sierpinski triangle with 5 subdivisions.

Revisiting the initial code, the faultiness of its algorithm became apparent

and hence could be corrected. The effectiveness of this new code compared to the approach recommended in the project description is due to the lack of redundant drawing.

We used an iterative development process, meaning we started out by making a small piece of the code work in one iteration. Through testing, trial and error the goal of the iterative step was reached and we then carried on with the next step, where more code was implemented and tested. After several steps, we realized that our approach was overly complicated, so we started again from scratch. This time we did not care for optimization of the algorithm, and we solely focused on correctness.[2]

7 Fern

The idea is to create a program that illustrate a fern graphically. Eventually, we were able to come with some ground principle illustrating the fern. The program we made is supposed to draw the first part of the stalk and then turn left and draw the leaves and sub-leaves. When the first side of the leaf the program heads back to the position. It draws the right side of the leaf and sub-leaves and every time we draw a leaf or sub-leaf the angle gets 0,3 smaller, then it returns to the position and goes forward and repeats itself, until the steps eventually goes down to zero.

First of all, we divided the fern into smaller pieces and named them, while we got a better knowledge on how to define a program. eventually we came up with a function with some necessary parameters for creating the fern. The parameters in the function are the turtle module, the width (thickness of each line), the segments, the curve (angle every time we draw the next leaf) and steps is the length of the fern frond (kildehenvisning). iWe had a lot of trials making the program recursive, because we did not have a position to return to the stalk.

Generally, we defined a function for the stem on the fern with several important parameters (length, width, segments curve, steps). To illustrate the segments the for-loop was a necessary method. The way for-loops work is for statement iterates over the segments of a sequence in order, executing the block each time. The pen size is the width of each line, in other words it is the thickness of the leaves and sub-leaves. And every time it goes to the next step the width downsizes with 0,3. the curve is the angle each time we go to the next sub-leaves and the stalk is also turn a bit to the left. At this point we had some difficulty because we did not have any position and return when the sub-leaves were drawn. So we used a two methods called position and heading. The first method is a way too return the current position (x,y) which is very useful when we want to return back to the stalk and go forward. The other method is the heading which returns the turtle direction.

[2]The source code for this part of the project can be found in the file sierpinsky-triangle.py or appendix A.2

At the beginning we had a lot of trials because we had some difficulty making it recursive. Then we came up with the idea that to divide the fern into smaller pieces and make the program for each piece and put them together. But at the end it made it too hard to assemble the pieces together, so we tried another approach. At the end we finally made a function with all the parameters that worked with recursion. Then we realized that the run time was way too long to execute, so we began to create another fern with a different method called clone. We made it work, however run time was fast but because the clone method uses a lot of memory the execute time takes way too much time. [3]

```
def recursive_stem(t, length, width, segments, curve, steps):
    if steps == 0:
        return
    if length < 0.5:
        return
    for i in range(segments):
        t.pensize(width)
        t.lt(curve)
        t.fd(length)
        pos=t.pos()
        head=t.heading()
        t.rt(90)
        recursive_stem(t, length*0.3, width*0.3, segments, curve, steps-1)
```

8 FDL-Parser

Since the fdl-file is to be read by an instance of the turtle module's Turtle class, it was decided to make a subclass of it called SmartTurtle. This class should be able to read a fdl-file, and execute he commands, giving a reasonable result. Through analysis of several fdl-files, a decision as to which attributes the SmartTurtle class should implement, could be made. Thus, an instance of the SmartTurtle should have a *start* attribute containing a command, a *length*, a *depth*, a dictionary of *rules*, and one of commands (*cmd*) which could be called with appropriate arguments. The rules should be unfolded according to the depth of the SmartTurtle, which seemed to indicate that an *unfolded* attribute would be convenient. This attribute would initially be an empty string, but when the user of the SmartTurtle class would want to execute the commands in the fdl-file loaded, a function to unfold the rules list and store them as a string in the unfolded attribute, will be required. This functionality is implemented in SmartTurtle's *step* function. The parser basically just reads through each line of the file, recognizing keywords, like *start*, *rule* and the like. The keywords are then used as look-ups in the dictionary yielding a string containing arguments, list of rules, or a float or int, depending on the type of keyword.

[3] The source code for this part of the project can be found in the file fern.py or appendix A.3

9 Conclusion

Trial and error seemed to yield the expected results, and in the process a fully functional fdl-parser was developed. There could be several optimizations in regards to a more user friendly interface and faster rendering of the fractals. Most of these optimizations should probably not be implemented using the turtle module, but perhaps with the help of the matplotlib module. Furthermore, in regards to increasing the speed of execution, some of the code could presumably be written in Cython or plain c/c++ and then utilized as an extension to python.

A Appendix

A.1 Binary tree

```
import turtle

def binTree(t, size, step):
    t.pensize(step//2)
    if step == 0:
        return
    if step == 1:
        t.pencolor("#00ff00")
        t.pensize(3)
    t.fd(size)
    a=t.clone()
    a.rt(35)
    t.lt(35)
    binTree(a,size*(2/3),step-1)
    binTree(t,size*(2/3),step-1)

def main():
    bob = turtle.Turtle()
    bob.hideturtle()
    bob.lt(90)
    turtle.tracer(0)
    bob.pencolor("#555555")
    binTree(bob,100,10)
    screendump(bob)

def screendump(t):
    """ Saves the canvas of t as a file.
    """
    img=t.getscreen()
    img.getcanvas().postscript(file="bintree.eps")
```

```
if __name__=="__main__":
    main()
```

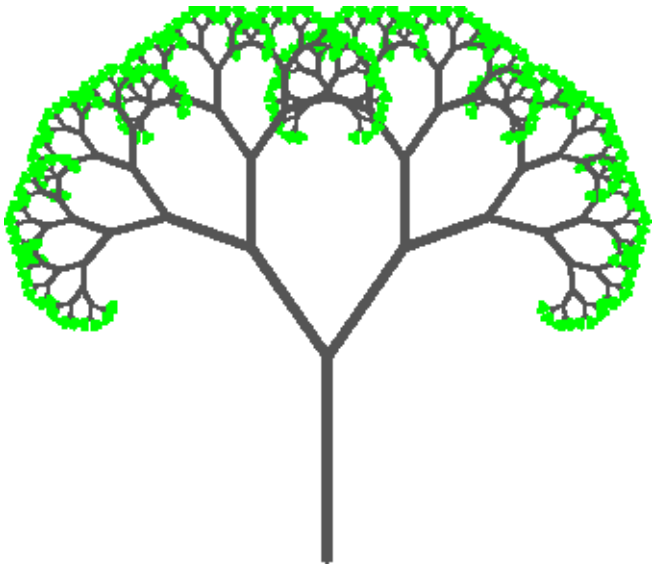


Figure 9: The output of binary-tree.py

A.2 Sierpinski's Triangle

```
import turtle
import tkinter
from math import sqrt, pi

def screendump(t):
    """ Saves the canvas of t as a file.
    """
    img=t.getscreen()
    img.getcanvas().postscript(file="duck.eps")

def triangle(my_turtle, center, side_len, height, points_up=False):
    """ Draws a triangle that points either up or down and
    has a given side length and center.
    """
    my_turtle.up()
    x=center[0]
    if points_up:
        y=center[1]+(2/3)*height
        my_turtle.rt(60)
```

```
    else:
        y = center[1]-(2/3)*height
        my_turtle.lt(120)
    my_turtle.goto(x,y)
    my_turtle.down()
    for i in range(3):
        my_turtle.fd(side_len)
        my_turtle.rt(120)
    t.setheading(0)

def init_sierpinsky_triangle(my_turtle, center, side_len, steps):
    """ This function is called only once, and basically just draws
    a triangle pointing up and afterwards calls
    sierpinsky_triangle() which then calls itself recursively
    """
    height=sqrt(side_len**2-(side_len/2)**2)
    triangle(my_turtle, center, side_len, height, True)
    sierpinsky_triangle(my_turtle, center, side_len, height, steps-1)

def sierpinsky_triangle(my_turtle, center, side_len, height, step):
    """ Draws a triangle pointing down and
    calls itself three times if step is gt 0.
    """
    triangle(my_turtle, center, side_len/2, height/2)
    if step >0:
        x=center[0];y=center[1]
        triangle_centers = [
            (x+side_len/4,y-height/6),
            (x-side_len/4,y-height/6),
            (x,y+height/3)]
        for c in triangle_centers:
            sierpinsky_triangle(my_turtle,
                                c,
                                side_len/2,
                                height/2,
                                step-1)

if __name__=="__main__":
    t=turtle.Turtle()
    turtle.tracer(1)
    t.hideturtle()
    t.speed(0)
    init_sierpinsky_triangle(t,(0,0),400,6)
```

A.3 Fern

```
import turtle

def lazy_stem(t, length, width, segments, curve, steps):
    if steps == 0:
        return
    for i in range(segments):
        t.pensize(width)
        t.lt(curve)
        t.fd(length)
        a=t.clone()
        a.rt(90)
        lazy_stem(a, length*0.3, width*0.3, segments, curve, steps-1)
        b=t.clone()
        b.lt(90)
        lazy_stem(b, length*0.3, width*0.3, segments, curve, steps-1)
        width *= 0.87
        length *= 0.87


def recursive_stem(t, length, width, segments, curve, steps):
    if steps == 0:
        return
    if length < 0.5:
        return
    for i in range(segments):
        t.pensize(width)
        t.lt(curve)
        t.fd(length)
        pos=t.pos()
        head=t.heading()
        t.rt(90)
        recursive_stem(t, length*0.3, width*0.3, segments, curve, steps-1)
        t.up()
        t.goto(pos)
        t.setheading(head)
        t.down()
        t.lt(90)
        recursive_stem(t, length*0.3, width*0.3, segments, curve, steps-1)
        t.up()
        t.goto(pos)
        t.setheading(head)
        t.down()
        width *= 0.87
        length *= 0.87
```

```
def main():
    t=turtle.Turtle()
    t.ht()
    turtle.tracer(9000)
    t.up()
    t.goto((-600,-100))
    t.down()
    #Stem(t, length, width, segments, curve, steps)
    recursive_stem(t,200,6, 5, 3, 5)
    turtle.update()
    turtle.mainloop()

if __name__=="__main__":
    main()
```

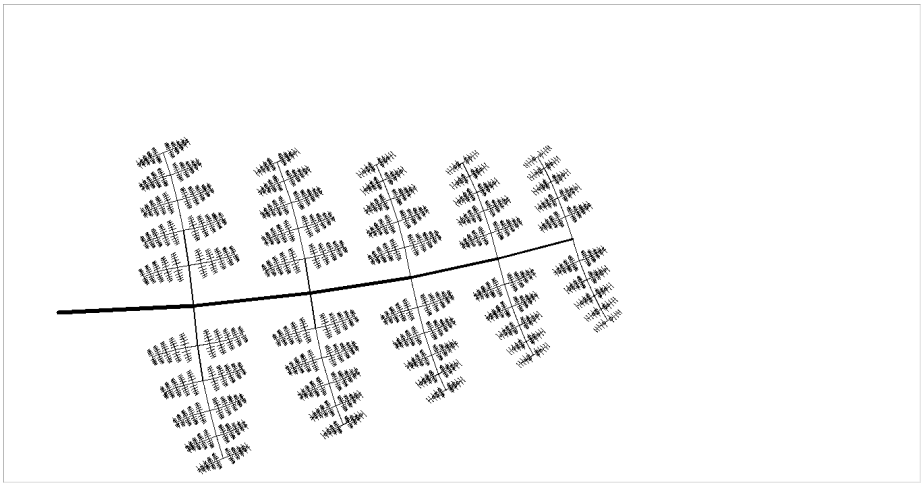


Figure 10: The output of fern.py

A.4 FDL-Parser

```
""" When this module is loaded as main, it will prompt
the user for a fdl-file (Fractal Descriptive Language).
The file must be present in the current working directory,
and the full filename must be specified.
"""

import turtle
import sys
```

```
class SmartTurtle(turtle.Turtle):
    """ This smart turtle can be instantiated
    with a fdl file, which can be interpreted o
    using SmartTurtle's draw method eg.:

    t=SmartTurtle(fdl='my.fdl')
    t.draw()

    """
    def __init__(self, **kwargs):
        self.rule={}
        self.cmd={}
        self.unfolded=''
        try:
            filename=kwargs.pop('fdl')
            self.load(filename)
        except FileNotFoundError:
            raise FileNotFoundError("Please instantiate \
            SmartTurtle with a valid filename.")
        except KeyError:
            print("Please load an fdl-file using load() \
            method in order to use SmartTurlte's draw method")
        super().__init__(**kwargs)
        self.ht()

    def load(self, filename):
        """ ### TASK 12 ###
        Loads fdl file and reads its elements into
        appropriate data structures.
        """
        f=open(filename,'r')
        for i in f.readlines():
            i=i.strip()
            i=i.split()
            if len(i)==0:
                continue
            if i[0]=='start':
                self.start=''.join(i[1:])
            if i[0]=='rule':
                self.rule[i[1]]=''.join(i[3:])
            if i[0]=='length':
                self.length=float(i[1])
            if i[0]=='depth':
                self.depth=int(i[1])
            if i[0]=='cmd':
                self.cmd[i[1]]=i[2:]
```

```
def step(self):
    """ ### TASK 8 ###
    Step computes the list of commands to be executed
    according to the depth given by the fdl. To save
    memory the list is saved as a string, where each
    command is represented by a letter.
    """
    state=self.start
    rules = list(self.rule.keys())
    rules.sort()
    m=len(rules)
    if m==1:
        for i in range(self.depth):
            state=state.replace(rules[0],self.rule[rules[0]])
    if m==2:
        for i in range(self.depth):
            state=state.replace(rules[0],"temp")
            state=state.replace(rules[1],self.rule[rules[1]])
            state=state.replace("temp",self.rule[rules[0]])
    self.unfolded=state
    for k in list(self.cmd.keys()):
        #Clean out 'commands' that are not executing a process.
        if self.cmd[k]==['nop']:
            self.unfolded=self.unfolded.replace(k,'')
        # Convert command strings to python functions
        else:
            self.cmd[k][0]=eval('self.' + self.cmd[k][0])
            if len(self.cmd[k])>1:
                # Convert argument to float
                self.cmd[k][1]=float(self.cmd[k][1])

def draw(self):
    """ ### TASK 11 ###
    Execute the commands in self.unfolded
    """
    if self.unfolded =='':
        self.step()
    for r in self.unfolded:
        cmd = self.cmd[r]
        if len(cmd)==1:
            arg=self.length
        else:
            arg=cmd[1]
        cmd=cmd[0]
```

```
            cmd(arg)
    def scale(self,val):
        self.length *= val


if __name__=="__main__":
    fdl_file=input("Enter fdl file:")
    s=SmartTurtle(fdl=fdl_file)
    turtle.tracer(0)
    s.draw()
    turtle.update()
    turtle.mainloop()
```