Chapter 12
Production Systems

12.1 A brief Introduction

In this chapter we continue our study of language in the specific context of computing. Natural languages, spoken or written, are complex beasts, of course we are not usually aware of this when we communicate in our mother tongue. Only when we learn a foreign language do we need to focus on the *grammar* of that language. Grammar essentially tells us how to *string* words together to produce meaning, and this process of string *production* can go on for a long time and be spread over many sheets of A4[1].

We shall not discuss *natural* language in this chapter, but rather *engineered* languages, designed for a specific purpose, and related to computing. But there is a more fundamental aspect of the material which follows, the *production systems* we shall study capture the essence of computation. By this I mean there is a production system created by Emile Post which is equivalent to the Turing machine and the latter describes totally what is going on in your machine at the moment.

Consider the following system, where we start with a string of one symbols **1** and we apply the *rule* 'replace any occurrence of a **1** with two symbols **11**'. So if we start with **1** we get **11**. Now let's continue and replace these **1**s so we get **1111** and so on. The result is shown in Fig. 12.1 and you can see that, if we take the denary representation of our strings, then each application of the rule is multiplying the 'input' number by 2. Now that's interesting, we can do arithmetic by applying rules to strings! This may remind you of some natural process, cell division or even growth in a bacterial

1	1
11	2
1111	4
11111111	8
1111111111111111	16

Figure 10.1 1Left shows string production, right shows the equivalent arithmetic operation,

[1] We shall not consider 'word limits' as you may encounter on writing an assignment or report, since that is a somewhat un-natural stance, especially if you are writing a novel.

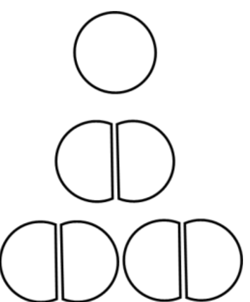


Figure 12.2 Arithmetic by cell division.

colony. Fig 12.2 represents the division of a single cell and subsequent divisions; sure, this is a form of *recursion*.

12.2 Production System Grammar

12.2.1 How to specify a grammar

A given language has a set of symbols called the *alphabet* (the above example just had one **1**) and produces *strings* of these symbols. There has to be a starting string, this is called the *axiom* and also *production rules* which tell you how to generate successive strings. The axiom and rules together is called the *grammar*. So the grammar for the above example is,

Axiom	**1**
Rule	**1 → 11**

Let's look at the following grammar where the alphabet has two characters **F** and +,

Axiom	**F**
Rule	**F → F+F**

The rule says that 'whenever we have an F in any string, we replace it with **F** + **F**. Applying the rule to the axiom, we produce **F** + **F**. Then applying it again we produce **F** + **F** + **F** + **F**. Continuing, we generate the following strings, called *theorems.*

Depth	**Theorems**
0	**F**
1	**F + F**
2	**F + F + F + F**
3	**F + F + F + F + F + F + F + F**

The *depth* of the production is just how many times we apply the production rule(s). Now, theorems are strings produced by the grammar rather than other strings, e.g., **FFF** is not a theorem.

12.2.2 The Turtle Implementation of Strings

The above strings by themselves seem not to have much sense, but this changes dramatically when the characters are interpreted as instructions to a turtle equipped with a pen (turtle graphics). If **F** is interpreted as 'move forward one unit with the pen down' and + is interpreted as 'turn through a specified angle', then the above strings make sense. If we choose an angle of 90 degrees, then the **rule F** + **F** looks like Fig.12.3. It's easy to see that the second production (depth 2) is a square, Fig.12.4.

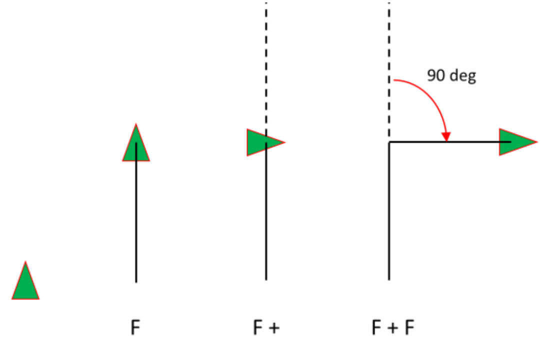


*Figure 12.3 Turtle moves forward **F** then turns 90 degrees clockwise + then moves forwards again.*

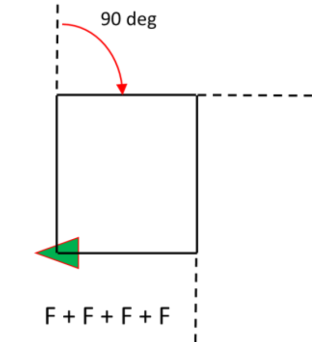
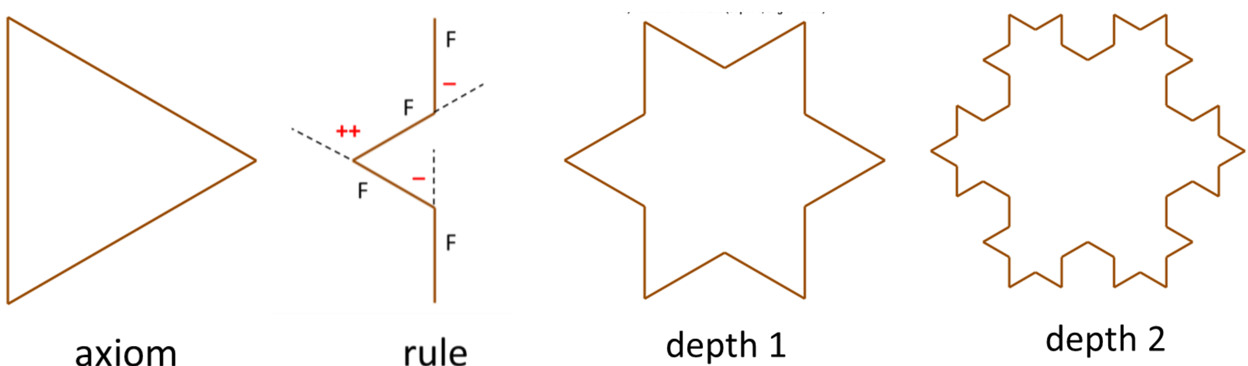


Figure 12.4 Depth-2 production using rule F + F

Let's take another example which reveals another way of thinking about production rules. Here is the grammar,

Axiom	**F + + F + + F**
Rule	**F → F - F + + F - F**

and we take the rotation angle for the + as 60 degrees. The axiom, the rule and the first two productions are shown below. The axiom clearly generates a triangle. The production rule is interesting; note that it has the same total anticlockwise and clockwise turns, so following the application of the rule, the turtle ends up pointing the same way it started. You can also see what the rule does, going from the axiom to the 1st production, each straight line is replaced by the rule shape, and the same happens going from depth 1 to depth 2. Of course, this is what the rule *says*.



The above example is known as the Koch Snowflake first published in 1904. As an aside it's interesting to work out the perimeter length of the snowflake. Looking at the rule, you can see that this divides a line length L into 3 vertical parts of length L/3. But there are 4 parts in the rule, each of length 3, so the rule has length 4/3. Since each application of the rule increases the total length of the curve by 4/3, after n applications the total length becomes,

$$3\left(\frac{4}{3}\right)^n$$

so the length of the snowflake grows quickly, for $n = 50$ the length is around 6 million.

It is interesting to print out the theorems generated by the production system corresponding to the above example curves. We find the following theorems, generated from the axiom by successive application of the rules, clearly they get very long.

Depth	Theorem
1	F-F++F-F++F-F++F-F++F-F++F-F
2	F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F- F-F-F++F-F++F-F++ F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F- F++F-F
3	F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F- F-F++F-F++F-F++F -F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F- F++F-F-F-F++F-F-F-F++F-F++F-F+ +F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F- F++F-F-F-F++F-F-F-F++F-F++F- F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F- F-F++F-F-F-F++F-F-F-F++F-F++ F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F- F-F-F++F-F-F-F++F-F-F-F++F-F ++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F- F++F-F-F-F++F-F-F-F++F-F-F-F++F -F++F-F++F-F-F-F++F-F

Let's take another example, the quadratic Koch island produced by the following grammar where the rotation angle is 90 degrees.

Axiom	**F + F + F + F**
Rule	**F → F + F – F – FF + F + F - F**

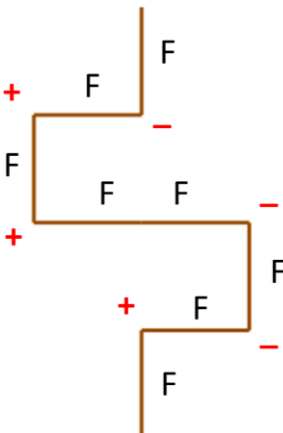
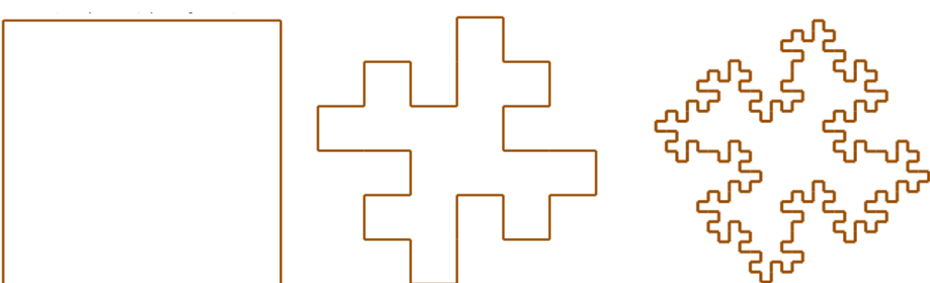


Figure 12.5 Quadratic Koch Island.

The axiom describes the square as usual, and the rule is shown in Fig.12.5. You an easily see the rule in action: Starting from the bottom, go forward F, then turn right + then go forward F, then turn left – then left again – then two steps forward FF then right + then forward F then turn right + then forward F then turn left – then forward. The number of right turns ('+' = 3) equals the number of left turns ('-' = 3) so we end up going in the same direction we started. Some theorems are shown below.



12.3 Extending the Grammar

12.3.1 The 'do not plot' symbol 'X'

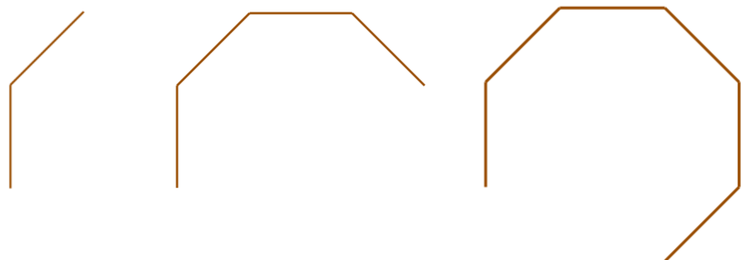
Here we introduce a new character in the alphabet, X. The production system works exactly as before, only the turtle when it sees an X just ignores it and does not plot anything. Here's an example system.

Axiom	**X**
Rule	**X → F + F + X**

To see how this works, let's generate a few theorems.

Axiom	**X**
Depth 1	**F+F+X**
Depth 2	**F+F+F+F+X**
Depth 3	**F+F+F+F+F+F+X**

This is straightforward, you can see that the rule tells us to replace all occurrences of X with F+F+X which is exactly what happens. Also we can give a higher-order interpretation, we see that the strings grow by F+F on each production and the string ends with a + then an X at the end. So the X at the end causes the strings to grow *from the end*. You can see this in the theorems turtleized below.



We can generalize the use of the 'no-plot' symbol to production systems where turtelization is not necessary for a successful interpretation. Consider the following system.

Alphabet	**1**
Axiom	**11**
Rule	**X → X11**

Some of the theorems produced by this system are tabulated below where we have shown the denary representation. Clearly this system generates the even numbers.

Axiom	11	2
Depth 1	1111	4
Depth 2	111111	6
Depth 3	11111111	8

You can also see that if we started with the axiom 1, then we would generate the odd numbers. So we can create any number!

Let's have a look at one more example, the system described by,

Alphabet	**1, +, =**
Axiom	**1 + 1 = 11**
Rule	**X + Y = Z → X1 + Y = Z1**

The rule means, 'if there is a theorem that consists of some string **X** followed by a + followed by a second string **Y** followed by an = followed by a third string **Z**, then we can create a new theorem by appending **1** to **X** and a **1** to the **Z**. Here's some theorems generated.

Axiom	1 + 1 = 11
Depth 1	11 + 1 = 111
Depth 2	111 + 1 = 1111
Depth 3	1111 + 1 = 11111

It's quite easy to see that this system is doing an arithmetic addition. Now that's an interesting result, we now know that production systems can *represent numbers* and it looks as though they can be designed to perform *arithmetic operations* on those numbers, they can compute!

12.3.2 Systems with Multiple Rules

Production systems may be constructed with more than one rule. The question then becomes, say we have two rules, then which one gets executed first? Consider this toy system.

Alphabet	**A, B**
Axiom	**B**
Rule 1	**A → AB**
Rule 2	**B → A**

The first rule tells us to replace an **A** by **AB** and the second rule to replace **B** by **A**. Again which rule fires first. Well production systems are special in that both rules fire

together, so we may simultaneously replace occurrences of both **A** and **B** in a string. Here's a derivation.

Axiom	**B**
Rule 1 (only possibility)	**A**
Rule 2 (only possibility)	**AB**
Rule 2 on A, Rule 1 on B	**ABA**
Rule 2 on A, Rule 1 on B, rule 2 on A	**ABAAB**

But we have to be careful. Look at the last derivation starting with the theorem ABA. We replace A with AB, then we move onto the original B which we replace with A, then we move onto the final A which we replace with AB. What we don't do, is when we replace A with AB, we do not immediately replace the B with an A. In other words, we process each string from left to right. Fig.12.6 shows this explicitly where we traverse a tree, from top to bottom.

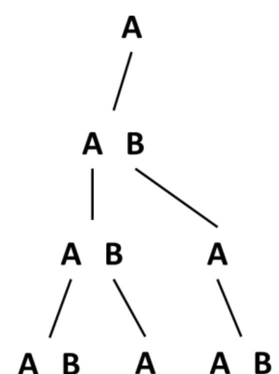


Fig. 12.6 Tree representation of the AB system.

12.4 Lindenmayer Systems

12.4.1 Mechanics of the L-Systems

This is a most beautiful application of production systems, these L-Systems let us grow realistic digital plants, please look ahead to see some examples of complete plants before we set down to study some detail. Let's first see where we are going, Fig. 12.7 shows the depth-1 and depth-2 of the theorems our system will generate, you can see the genesis of a branching structure very suggestive of a plant.

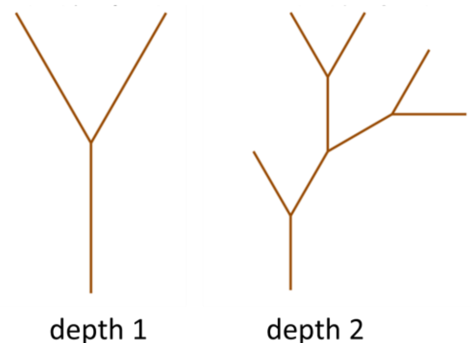
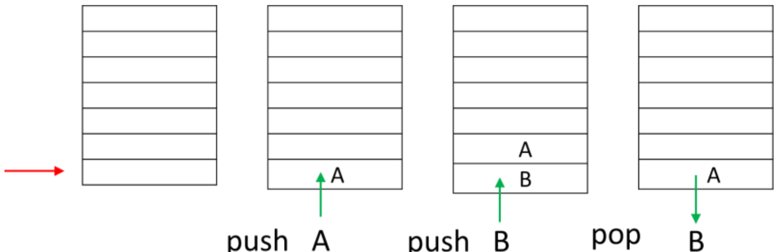


Fig.12.7 First two theorems of our first L-System

Before we look at the production system, we need to understand one concept from computer hardware, the *stack*. Stack are implemented in RAM and can contain many values, but unlike RAM the elements may not be accessed individually. Instead the *stack* has a single port of entry and exit. So store a variable, we *push* it onto the stack through this port, and to read a variable, we *pop* it from the stack. The diagram below shows the stack in operation. Think of each row as holding a 32-bit integer.

Initially the stack is empty and there is a pointer (red arrow) to the bottom of the stack where we save and retrieve our data values. Let's say we are writing some assembler



code and we have variables A and B which hold some number. When we code **push A** then the value of A is pushed into the bottom of the stack, then we code **push B** and the value of B is pushed into the bottom, and A moves up a slot. So both A and B are stored safely. Now we wish to get some data from the stack, so we code **pop**. Note we *cannot* ask for A or B, we get whatever is at the bottom of the stack, at the stack pointer location. So here we get B from the stack, and A tumbles down.

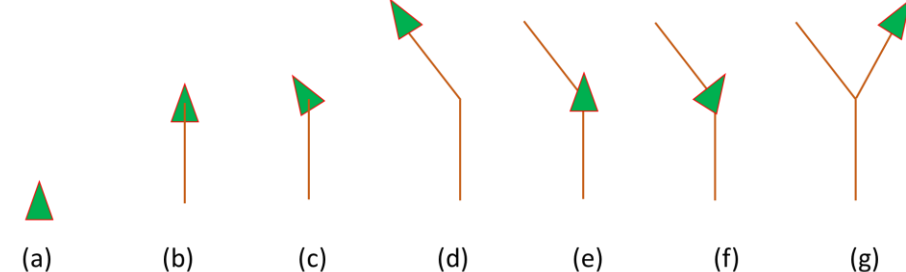
To produce plants we need a push operator, the character **[** and the pop operator, the character **]**. In a well-formed string we must have equal numbers of pushes and pops. So let's see this in action, consider the following grammar.

Alphabet	**F, +, [,]**
Axiom	**F**
Rule	**F → F [- F] + F**

Let's apply the rule to the axiom which will generate the theorem,

F → F [- F] + F

and look at the detailed turtle movements shown in the diagram below.



The turtle starts at (a) and the first string character **F** moves it forward (b). Then comes a push **[** so the turtle remembers its *state* = location and orientation at (b). Then comes a turn left – which here is 30 degrees which is (c). Then another **F** so it moves forwards to (d). Then we hit a pop **]** so its pushed state (b) is restored which is (e). Then it is told to turn right + (f) and finally to move forward **F** to bring it to (g) where it is located with the last rotation intact. So you can see how the plant shown in Fig.12.7 (depth 1) is produced.

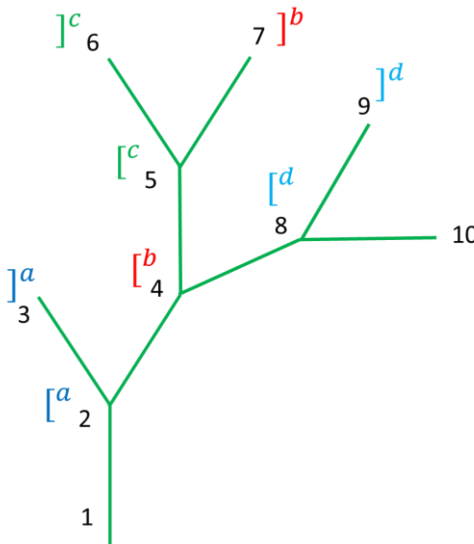
Now let's turn to Fig.12.7 (depth 2) and try to understand how this plant is produced. Hold on to your hat! Here is the theorem produced at depth 2 where we have color coded and applied superscripts to the pushes and pops so you can see how they work. In the case of *b* and *c* they are nested.

$$F[^a - F]^a + F[^b - F[^c - F]^c + F]^b + F[^d - F]^d + F$$

Please consult the diagram below. We start at 1 and move forward **F** to 2. Then we push the turtle state a onto the stack $[^a$ and then turn left – and move forward **F** and arrive at 3. Now we pop the turtle state from the stack $]^a$ and we return to 2. Next we turn right + and move forward **F** which brings us to 4.

Now we are at 4 and we push the state b onto the stack $[^b$ turn left – and move forward **F** which brings us to 5. Now we do another push $[^c$ (so that we have c at the bottom of the stack and b above it) and we turn left – and move forward **F** so we end up at 6.

Now we pop c off the stack]^c, and so we return to 5. We then turn right + and move forward **F** which brings us to 7, so we have completed one of the branches. Next we have to complete the other branch, so we pop b off the stack]^b which returns us to 4 ready to start the other branch, We turn right + and move forward **F** which brings us to 8. Here we push d onto the stack [^{d} then turn left **-** and move forward **F** to 9. To draw the remaining stem, we must return to 8, so we push d off the stack]^d and we turn right + and move forward **F** so we arrive at our destination 10.

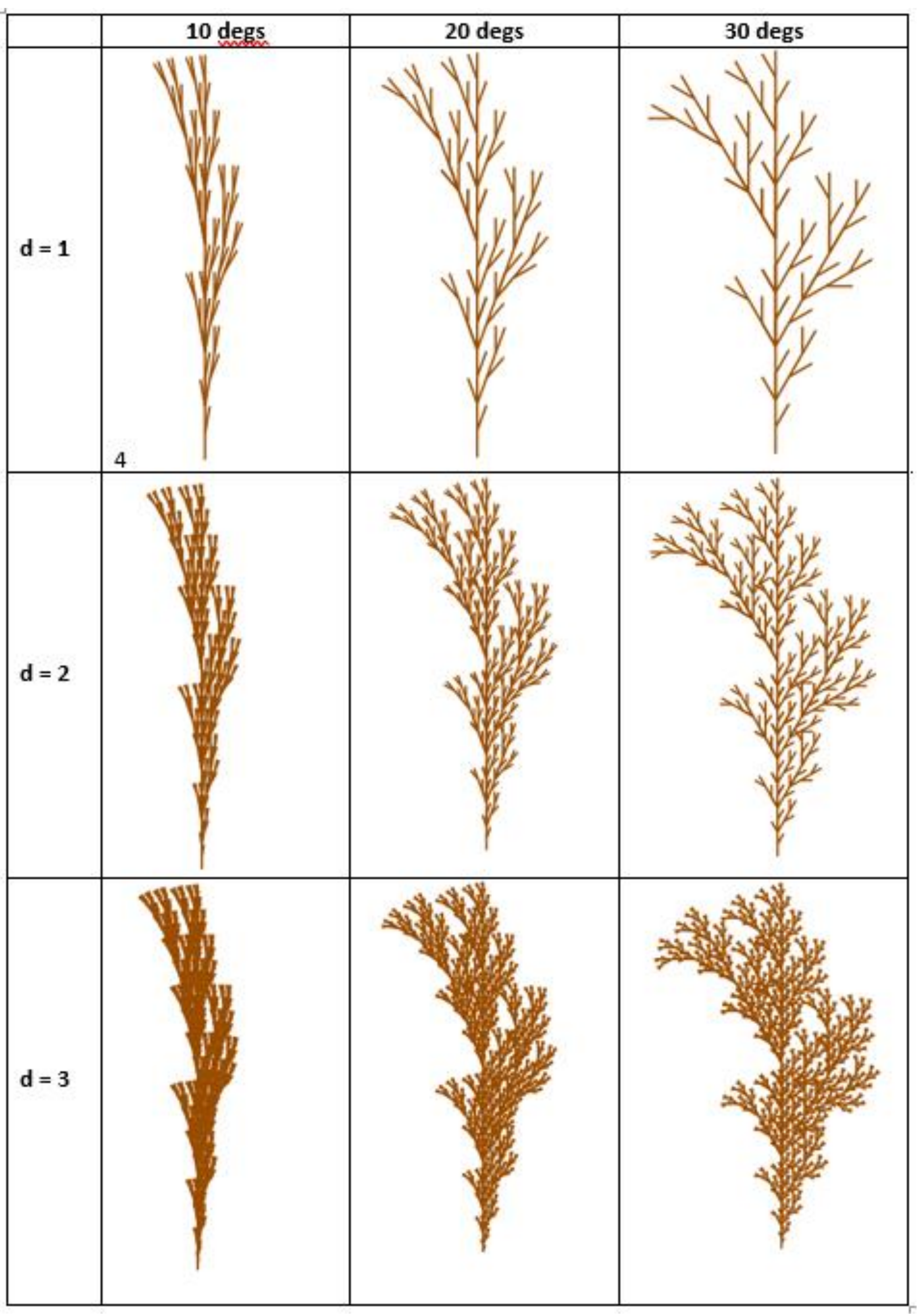


Note in the above diagram how the pushes and pops are *paired*.

12.4.2 An Atlas of Plant Productions.

The diagrams below show plants produced by the following production system for a range of angles and depths. The angle controls the lateral spread of the plant, and the depth controls its 'bushiness'.

Alphabet	**F, +, [,]**
Axiom	**F**
Rule	**F → F [+ F] F [- F] [F]**


10 degs
20 degs
30 degs
d = 1
4
d = 2
d = 3

..Here’s a selection of plants cultured by other systems. Each

Axiom	F
Rule	F → FF-[-F+F+F]+[+F-F-F]

Axiom	F
Rule 1	F → FF-[XY]+[XY]
Rule 2	X → +FY
Rule 3	Y → -FX

Axiom	F
Rule	F → F[+F]F[-F]F

plant in the top row is for depth = 3, angle = 30 degrees, and the bottom row is depth = 4, angle = 30 degrees.

12.5 Designing and Engineering Systems

It is possible to design an L-System to draw a certain type of shape. To do so, we must try to find some basic *atomic* rules for constructing shapes.

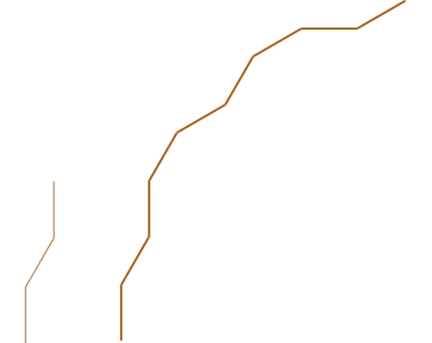
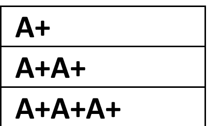


Fig.12.8 Left shows shape element, and right shows concatenation of 3 elements with angle of 30 degrees.

12.5.1 Concatenating Shapes

Let's say we want to create a chain of identical shapes each shape is the same an described by a string **A**. So we will need to produce theorems **A**, then **AA,** then **AAA**. Let's make things a little more complete, and require that each additional shape is rotated with respect to the prior. So we need a list of theorems that look like this



A+
A+A+
A+A+A+

You can see that concatenation can be obtained in general by using the dummy character **X**

Axiom	**X**
Rule	**X → A+ X**

which will produce the following theorems, which is what we want.

A+X
A+A+X
A+A+A+X

Take the example with **A** = **F** + **F** – **F** , this element is shown in Fig.12.8 together with the depth 3 production. Here is the system, with **A** substituted.

Axiom	**X**
Rule	**X → F + F – F + X**

12.5.2 Bicycle Spokes

Now, let's say we wanted to draw a radial pattern of simple spokes as shown in Fig.12.9. How would we approach this? Well when we draw a line using **F** we need to return to the starting point. We know how to do that, we just use the push and pop operators like this **[F]**. But the next spoke is rotated

an angle, so we need something like this **[F]+**. So we need to create theorems like this

[F]+	First spoke, turn ready for second
[F]+ [F]+	Second spoke, turn ready for third
[F]+ [F]+ [F]+	Third spoke, turn ready for fourth

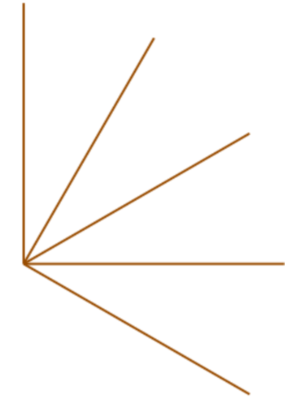


Fig.12.9 Bicycle spokes for depth of 5.

So we have a concatenation which we already have solved. The production system therefore becomes,

Axiom	**X**
Rule	**X → [F] + X**

Of course we can replace **F** in the above rule with **A** which describes the above shape element. In that case we get result in Fig.12.10.

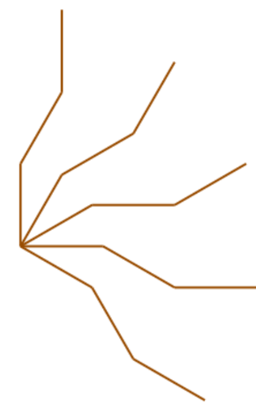


Fig.12.10 Depth-5 radial arrangement of shape elements.

12.5.3 Moving without Drawing: The Cantor Set

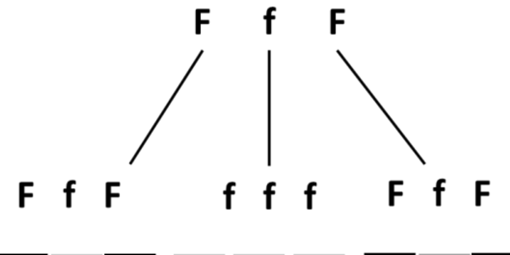
The Cantor Set is a very interesting mathematical object. You can make such a set by drawing a line (depth 0) then erasing the middle third (depth 1) and continuing to do this recursively. This is shown in the diagram below.

We introduce a new character **f** into our alphabet which means move forward but do not draw, the turtle's pen is well and truly raised up. Here is our production system.

Axiom	**+F**
Rule 1	**F → F f F**
Rule 2	**f → f f f**

First note the + in the axiom. This is to turn the turtle so it draws in a horizontal direction. Rule 1 clearly splits a line into 3 line segments, the first and third are drawn, the second is blank. Why do we need the second rule? This ensures a constant gap between line segments when they are split.

Consider the depth 2 production string **F f F f f f F f F**, this is produced and expressed as the following tree. Remember



both rules apply simultaneously. The black lines are drawn and the grey lines show spaces. This agrees with the depth-2 diagram drawn above.