Lecture Notes: Finite State Machines and Context Free Grammars

Prof. Ganesh Ramakrishnan

August 4, 2008

Contents

1 Introduction **3**

2 Finite State Machines **9**

2.1 Deterministic Finite State Machines 11
2.2 Closure: A Brief Introduction 15
2.3 Examples of Non Regular Sets 19
2.4 Non Determinism 20
2.5 Computational Comparison of Deterministic and Non-deterministic Machines 23
2.6 Conversion From Non-deterministic to Equivalent Deterministic Machine 24
2.7 Closure 27
2.7.1 Closure under Union 31
2.7.2 Cosure under Concatenation 32
2.7.3 Closure under Complement 33
2.7.4 Closure under Intersection 33
2.7.5 Alternating Finite State Machines 35
2.8 Equivalent Representations of Regular Sets 36
2.9 Non-regular Sets and Introduction to Pumping Lemma 36
2.10 The Pumping Lemma 39
2.11 Linear Grammar 44
2.11.1 From Finite State Machine to Linear Grammar 44
2.11.2 From Left Linear Grammar to Finite State Machine . . . 47
2.12 Traingle of Equivalance for Finite State Machines 48
2.13 Mimization of Finite State Machines 48
2.13.1 Intuition for Minimization 49
2.13.2 An Efficient Algorithm for Minimization 51
2.13.3 Comutational Complexity 58
2.14 Decision Problems about FSMs 59
2.14.1 Membership Question 59
2.14.2 Equivalence of Two Finite State Machines 59
2.14.3 Is the language of an FSM Infinite? 60
2.14.4 Is Regular Set A contained in regular set B? 61
2.15 Moving Forward 61

3 Context Free Languages **63**
3.1 Context Free Grammars and Derivations 64
3.1.1 Example 1 64
3.1.2 Example 1 65
3.1.3 Example 2 68
3.1.4 Example 3 69
3.2 Techniques for Writing Context Free Grammars 70
3.2.1 Example 4 71
3.2.2 Example 5 72
3.2.3 Example 6 75
3.2.4 Example 7 77
3.2.5 Closure under Intersection and Complement 78
3.2.6 Closure under Union 78
3.3 Relationship to Compiling and Programming Languages 79
3.4 Normal Forms 80
3.4.1 Chomsky Normal Form 80
3.5 Does a CFG Generate any String 87
3.6 Pushdown Machines 88
3.6.1 Example 1 89
3.6.2 Example 2 90
3.6.3 Example 3 92
3.6.4 Simulating a stack with a queue 94
3.6.5 Example 4 94
3.7 Conversion from CFG to PDM 95

Chapter 1

Introduction

Theory of computation (TOC) is perhaps one of the most abstract areas in the computer science curriculum, but one of the most fundamental area any computer scientist can know. The really interesting thing in TOC will neither help you write a program nor build a computer per say. TOC will help you understand what people have thought for the past 50 years about computer science as a science. And it is about

1. what kind of things we can compute mechanically,

2. how fast can we do it and

3. how much space in memory does it take us to do it.

And because of that it is called the theory of computation.

Like lots of abstract areas, and areas that have a lot of cool thinking and neat stuff in it. Especially in computer science, there are lots of applications that come out of TOC. Things that are pretty much serendipitous. All of compiler design and theory about building compilers and writing programs to translate languages comes from theory of computation. It turns out that some of the models of computation we discuss happen to correspond to programs that help us write compilers. There are lots of other applications as well. When you talk about computer architecture, you model a particular process with a finite state machine. The idea of finite state machine comes from the theory of computation. The idea of string matching in any word processor or any kind of editor - that is a finite state machine. When you do any kind of compiling or representation of a language like XML, you describe it with a grammar. And that grammar is the next level of model of computation.

This area talks about different models of computations - different kinds of things you can use to compute stuff. WHat does it mean to compute something? It is simply to abstract away from the specifics of the machine and all the specifics of the problem in the following style. You imagine that all you are dealing with is programs or computations that take an input, the program sits

and thinks and calculates and computes and then says a 'yes' or 'no'. Do I accept this input or do not accept this input. Whatever number crunching is involved in only to decide a yes or no.

In order to do that, we think of computing in terms of set or alphabets of elements. Example alphabets are the binary alphabet $\{0, 1\}$ or even the uniary alphabet $\{1\}$ - consisting on just one symbol. And we consider strings along that alphabet. An example of strings you might want to compute is

> A set of binary strongs that end in 0.

This is not a very hard thing to compute. Given a string, you just need to see the last symbol in the string. If the string is a 0, your answer is 'yes', else it is 'no'. And it is very easy to write a program that does this for you. There are other things that are harder. Suppose every piece of java code is converted into a binary file. And you are asked to determine if a given binary file is a legitimate piece of java code. Or is it just a junky binary file that represents something else. So the question is

> Does the given binary file represent a legitimate piece of java code or not?

How easy is to check if the given binary strings represents a legal Java program. It must be possible - that is why we have syntax checkers that give error messages if the java code is not legitimate. The program that checks syntax in fact, does something more that say 'yes' or 'no'. It actually spits out the error in the code. And that is what compiler design is largely about.

Going one step further, we can identify an even more challenging problem:

> Identify the set of strings that represent legitimate Java programs that never go into an infinite loop.

Is there a program that will look at your Java program and tell you 'yes' iff it is legitimate Java code and will never go into an infinite loop and a 'no' otherwise? In fact, there isn't any such program. It is impossible. There is no way to compute this, no matter how hard you try - it is not going to work. Even if someone attempts such a program, it might either give you the wrong answer or it might never give you any answer - it might simply go into an inifite loop. Your program that you write to try and do this does not exist.

That is what we will be talking about in the following two chapters. What can you compute? And what can't you compute? And what kind of models do you need to do this kind of computations? Your own sense of how to do computation now might be a programming language - C++, scheme, java, *etc.*. We are going to abstract away all details and get to the root of what a programming language really is. The most popular way to view a programming language with everything stripped away is called a Turing machine, invented by Alan Turing.

The Turing machine is a very simple machine which we will not delve into right now. In fact, we will build up to the Turing machine. But Alan Turing,

in the 1936's, when he invented the Turing Machine (he died in the 1950's), invented a mathematical abstraction that he thought was a complete representation of how we might do computation. To the extent that, anything you do on a normal computer with a normal programming language could be done on his machine – you could write a program for his machine that does the same thing. The machine is very very abstract and simple. But because of its power, we can prove really interesting things about it. Like the fact that you can never have a program to solve the problem stated above.

When Alan Turing tried to abstract away what we mean by computation, it was the first attempt to describe it rigorously. He showed the difference between thinking and logical computation. The word *computation* got a rigorous meaning only in the 30's and 40's of the 20th century.

What we are going to do in this tutorial is work our way up from a much lower level. One question to be asked when you define something very abstractly with arms and legs and head, *etc.* is: what if you cut off one of its arms or legs. Will it be able to do everything it did before but just taking a bit longer? Or is it actually handicapped and will not be able to somethings it used to be able to do. For example, consider again the problem of constructing a program to answer the question

> Does the given binary file represent a legitimate piece of java code or not?

We know that we can write a program to accept such strings. Therefore, we know that we can write a program for a Turing machine that would accept those. What if we cut off the left arm of the Turing machine (metaphorically)? Will it be that we cannot do the above computation any more? Is it that we have cut off actual power. Or is it that it will take machine much longer to do the computation? How much can I chop off from a Turing machine and still be able to do the computation. Those are the kind of questions people ask in the Theory of computation.

We will start off from the lowest level of abstraction - the Finite state machine and will build our way up to the Turing machine. And then start exploring problems in the twilight zone which cannot be even computed by a Turing machine.

Figure 1.1 gives a blue-print of the area of theory of computation. We will summarize the set of machines we will cover using the picture in Figure 1.1, typically referred to as the *'Bull's eye'*. The Bull's eye is more or less the *Chomsky hierarchy*, which Noam Chomsky presented in the mid 50's. The simplest kind of machine we are going to consider is one that does not have as much power as programming languages - one that is simpler, called the *Finite State Machine* (FSM). That is the smallest and simplest level lying the center of the bull's eye. FSMs can compute can compute certain kinds of sets and decide certain kind of things - like whether there is a zero at the end of a string. But they cannot do things mopre complicated, like figure out whether something is a legal Java program. There are many other interesting things

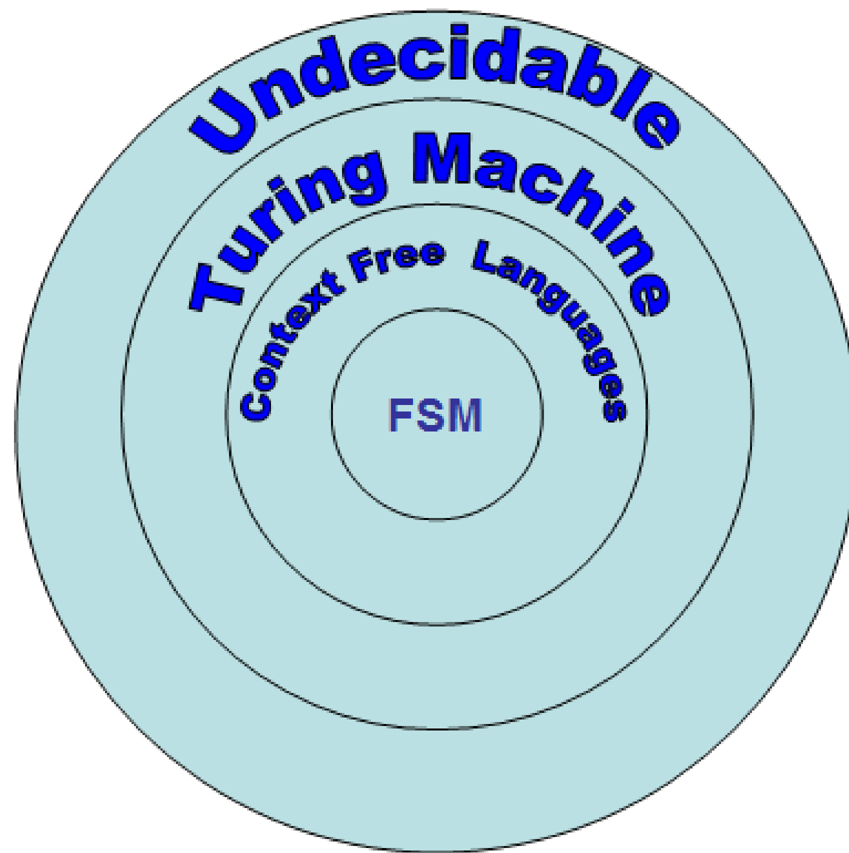

Figure 1.1: The Bull's eye: Blue-print of TOC

they cannot do. If you add some power to FSM, you get a class of sets called *Context Free Language*[1] (CFL) that can be identified only by a more powerful machine. Context free language includes the language corresponding to FSMs[2] The second circle (from inside) that includes the first one, represents CFLs. All the set of strings that correspond to legal Java programs are inside the CFL circle.

Then we bump up another level to *Turing machines.* Turning machines represent full-level computation. Anything that can be really computed by a computer using any normal programming language can be computed by a Turing machine. And problems in the twilight zone (corresponding to the last ring) are called *undecidable.* There is not program anywhere that figure out the answer, mechanically, to *undecidable* problems in general - they are no algorithms to figure out the answers to these problems.

The work on FSMs was primarily done in the 1950's as well as early 1960's, more than a decade after Alan Turing's work on Turing Machines. The work on context free grammars was done primarily in the late 1960's and early 70's. These topics developed more out of mathematical and intellectual interest. And at the same time there was the serendipitous application in the form of compilers and programming languages.

This is a blue-print for the area of theory of computation. As we progress, the blue-print will become more refined. There are sub-circles, overlapping circles, all sorts of relationships between circles. Within the turning machine level itself there are hierarchies. P and NP are both inside the turing machine level - P would be an inner sub-circle while NP would be an outside sub-circle. There is an entire hierarchy of complexity theory that takes place in the turning machine world.

We will also see the there are different ways of describing languages. One way is description using machines with different amounts of power. Another way is by specifying grammars. A third way that works only on some levels is by expressions. So there are different ways of describing these classes of languages.

In particular, the circle of *context free languages* can be split into two. by throwing in extra power. By throwing in the same extra power into the *FSM* and *Turing Machine* circles, though, we do not get a more expressive language.

[1] A set of strings is often called a *language.* This should not be confused with a *programming language.* A language is nothing more or less than a set of strings.

[2] We will later study the language corresponding to FSMs called the *Regular language.*

Chapter 2

Finite State Machines

We will get down to some details of theory of computation by first addressing the innermost circle of the bulls eye. While trying to get the ideas accross, we might compromise on rigour so that ideas do not seem to be really more complicated than they are.

We will see what a finite state machine is by looking at an example and then pointing out in the example what the real traits are. Let us start with an example - the set of binary[1] strings that have an even number of 0's. We will describe finite state machines by actually implementing one for this problem.

Problem 1 *Construct an FSM that accepts only strings that have an even number of 0's.*

An intuitive definition of the class of finite state machines is "Anything you can solve by remembering a finite amount of information". If you use that gut instinct rule, you will be almost able to identify whether or not a given problem can be solved by a finite state machine. An FSM has states or memory. We will construct our first FSM by giving a semantic meaning to each state. Imagine that we also have a tape on which characters get streamed. The machine will have a head that will look at one symbol at a time and strictly move left to right across the tape. The movement of the head from one symbol to the next, corresponds to the movement of the machine from one state to another. And the movement from one state to another is based on a transition function of the observed symbol. This is the simplest possible computation you could have. The following figure shows our first finite state machine.

The state A will capture strings with an even number of '0's. The state B will capture strings with an odd number of '0's. Before we have seen any characters in the string, we have zero (an even number of) '0's. Therefore, we

[1] The alphabets can be very large in general. for example, Java programs are over an alphabet of $A \ldots Z$, $0 \ldots 9$, *punctuations*, *etc.* But for the purpose of understanding the abstractions to be discussed in the class, we will not need to consider alphabets that have more than two symbols. Two symbols give you kind of an exponential jump over an alphabet of one symbol and after that it is kind of convenience.

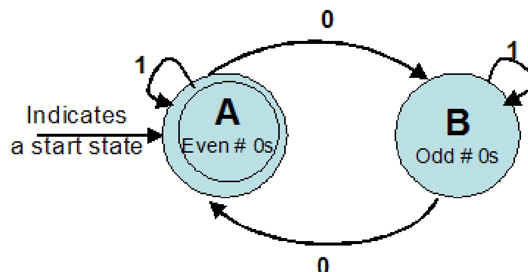

Figure 2.1: Our first finite state machine

will start at the state 'A'. Every finite state machine should have one arrow coming out of each state for each symbol in the alphabet. For binary strings, we should therefore have two arrows coming out of each state. When we see a '0' in state A, we switch over to state B. When we see a '1' at state A, we remain at that state (indicated by a self loop).

In fact, if you write the structure and design of finite state machines in this way by (a) writing the states down first without any arrows, modeling the problem semantically by giving information in the states and (b) then only later throwing in the arrows, you have a much easier time doing it. And if you have trouble deciding what information is to be thrown into the states, that is where you need to stop and think, not after putting in arrows which could perhaps get you on the wrong track.

The state you could begin with is the *start state*. In Figure 2.1, the start state A is marked with an incoming arrow $\longrightarrow$. The subset of states where the machine ends up when it finishes reading the string and tells us that the string has been accepted is termed as the set of *final states*. In the above example, state A is a final state. And if the machine ends up in any non-final state, the string is not accepted by the finite state machine. We will represent each such state by putting a double circle round that state. That gives us a complete finite state machine - a machine that accepts all binary strings with an even number of 0's and rejects every binary string that does not have an even number of 0's. We could run the FSM on any string to determine its membership to the class for which the FSM was designed.

Note that this FSM gives only 'yes' and 'no' answers. It is hard enough a computation to deal with 'yes' and 'no' questions for many problems. At the Turing machine level, we will discuss the problem of producing an output. In computer architecture, you will find versions of FSMs that have outputs, called Mealy machines, Moore machines, *etc.* We will find some such examples in exercises. For the most part, we will not need to discuss such machines in our tutorial.

2.1 Deterministic Finite State Machines

We will now provide a formal definition of deterministic finite state machines.

Deterministic Finite State Machine *A deterministic finite state machine consists of:*

1. *A finite set. of states, often denoted Q.*
2. *A finite set of input symbols, often denoted E.*
3. *A transition function that takes as arguments a state and an input symbol and returns a state. The transition function will commonly be denoted δ. In our informal graph representation of automata, δ was represented by arcs between states and the labels on the arcs. If q is a state, and a is an input symbol, then $\delta(q, a)$ is that state p such that there is an arc labeled a from q to p.*
4. *A start state, one of the states in Q.*
5. *A set, of final or accepting states F. The set F is a subset of Q. A deterministic finite automaton will often be referred to by its acronym: DFA. The most succinct representation of a DFA is a listing of the five components above that is a "five-tuple": $A = (Q, \Sigma, \delta, q_0, F)$ where A is the name of the DFA, Q is its set of states, Σ its input symbols, δ its transition function, q_0 its start state, and F its set of accepting states.*

Let us look at a few more examples of FSMs.

Problem 2 *Construct an FSM that accepts every string with an even number of 0' and even number of 1's.*

Let us design such a machine by first outlining our plan. What should the states of such a machine be? One way to go about it is design four states, each for an even/odd combination of 0's and 1's. We will represent the even (e)/odd (o) nature of 0 and 1 in a string as a tuple $< x, y >$, where $x, y \in \{e, o\}$. Thus, $< e, e >$ represents an even number of both 0's and and 1's. Figure 2.2 shows the corresponding FSM.

When the FSM gets a 0 in state $< e, e >$, it toggles to $< o, e >$. When it receives a 0 in $< o, e >$, it toggles back to $< e, e >$. Similarly, it toggles between $< e, e >$ and $< e, o >$ on receiving the symbol 1, between $< e, o >$ and $< o, o >$ on receiving symbol 0 and between $< o, o >$ and $< o, e >$ on receiving symbol 1. The state $< e, e >$ is the start state as well as final state.

Notice that there need not always be a single final state. Suppose, we had to design an FSM that accepts only strings having an even number of 0's and an even number of 1's or having an odd number of 0's and an odd number of 1's. Such an FSM can be easily designed from the previous one by assigning the status of final state to the $< o, o >$ state in addition to the $< e, e >$ state. This yields the FSM in Figure 2.3.

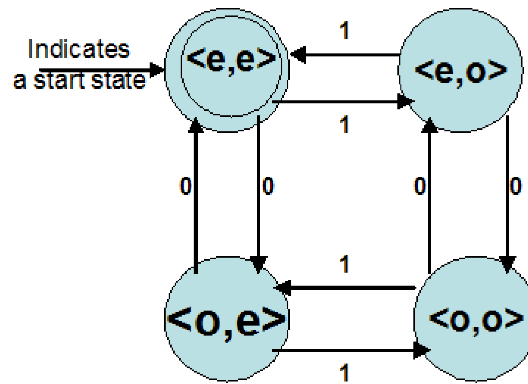

Figure 2.2: Finite state machine that accepts only strings with an even number of 0's and an even number of 1's

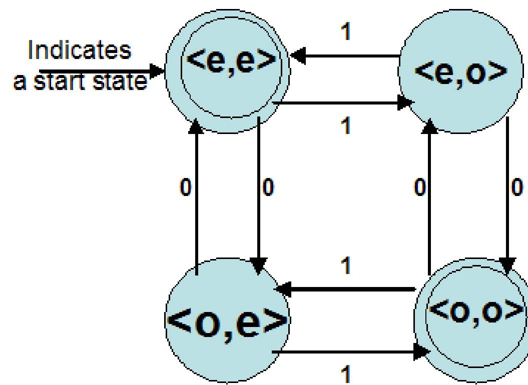

Figure 2.3: Finite state machine that accepts only strings having an even number of 0's and an even number of 1's or having an odd number of 0's and an odd number of 1's

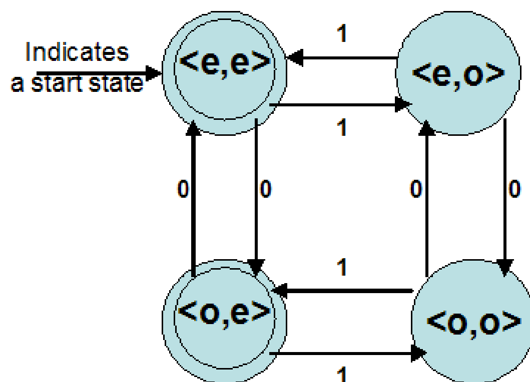

Figure 2.4: Finite state machine that accepts only strings having an even number of 0's and an even number of 1's or having an odd number of 0's and an even number of 1's

It is interesting to note that insisting on a single final state takes away power from a deterministic machine[2] (it is almost akin to cutting off half the brain of a finite state machine). There is no obvious characterization of the subset of the regular language that has finite state machines with the number of final states restricted to 1.

Let us consider another variant of the above problem:

Problem 3 *Design an FSM that accepts only strings having an even number of 0's and an even number of 1's or having an odd number of 0's and an even number of 1's.*

Again, such an FSM can easily leverage the FSM in Figure 2.1 by making state $< o, e >$ a final state in addition to state $< e, e >$. This yields the FSM in Figure 2.3.

Is this the smallest machine? What is the logically equivalent description of the set of strings that have: *An even number of 0's and an even number of 1's or an odd number of 0's and an even number of 1's* ?

It is simply, the set of strings that *have an even number of 1's*! Thus, the FSM in Figure 2.4 is not the smallest FSM. The states $< e, e >$ and $< o, e >$ could be compressed into a single state $< -, e >$ to generate the FSM in Figure 2.5.

This example helps us realize us that given a set of strings, there could be a set of more than one FSMs that accept the strings. If this set had cardinality more than 1, there will be a canonical minimal FSM. And this is a very nice thing to have about a machine. Because there is no canonical minimal program for the set of programs you write in your computer architecture course or your programming language course. However, for finite state machines, you have this

[2] Exercise.

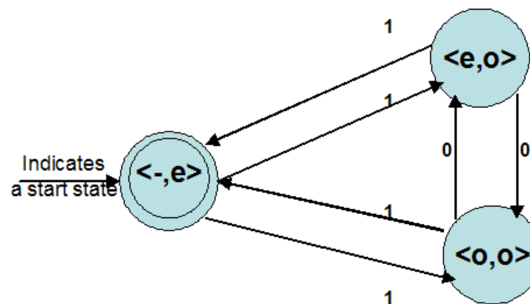

Figure 2.5: Finite state machine that is smaller than the FSM in Figure 2.3 although equivalent.

nice property. And there is a very rigorous dynamic programming algorithm that does that minimization (we will talk about that later in the tutorial).

Next, we will look at another problem:

Problem 4 *Design a finite state machine that accepts only binary strings that are divisible by* 4.

How do we solve this problem in daily practice? One bad way is to convert the binary number to base 10, divide the number in base 10 by 4, observe the remainder and if the remainder happens to be 0, we know that the binary number is divisible by 4. It is like tying one's hands behind and then trying to tie the shoe. It is much easier to look at the binary number directly and decide divisibility by 4 - in fact any division by a power of 2 in binary is obvious. It is as easy as determining if a number in base 10 is divisible by 100 - you just look at the last two digits and if they both happen to be 0's, you know that the number is divisible by 100. Similarly, you have to just look at the last two symbols of the binary number to determine its divisibility by 4 - if the last two symbols are 0's, then you know that the binary number is divisible by 4. Thus, we are looking for an FSM that identifies binary strings ending in 00.

We will try to derive some hint from a slightly different problem; it is easier determining an FSM that accepts only binary strings beginning in 00 (Figure 2.6).

Problem 5 *Design a finite state machine that accepts only binary strings beginning with* 00.

You need four states - one for no zeros (A), another for the first zero (B), the third (accept state) for the second zero (C) and a fourth dead state where you go if you encounter a '1' as one of the first two symbols.

To construct an FSM that accepts only binary strings ending in 00, we will start by retaining states A and B above, while slightly changing their semantics; state A captures all strings that do not end in 0, state B captures strings that

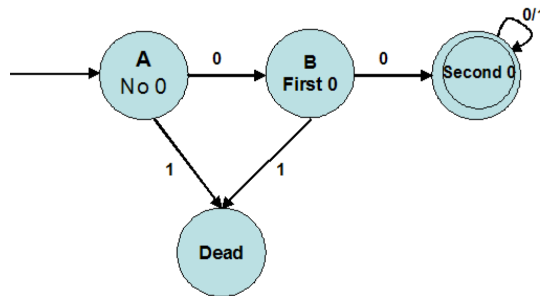

Figure 2.6: Finite state machine that accepts only binary strings beginning in 00.

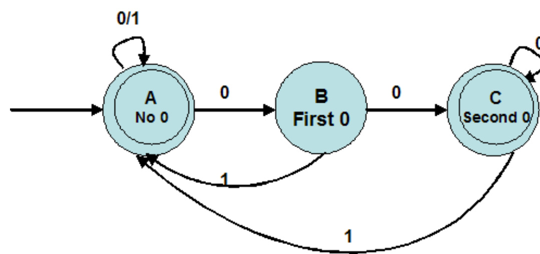

Figure 2.7: Finite state machine that accepts only binary strings divisible by 4. (*i.e.*, ending in 00).

end in one 0 and state C captures strings that end in two 0's. If the FSM encounters a 1 in any of the three states, it goes back to state A. Figure 2.7 shows the FSM that accepts only binary strings that are divisible by 4. Note that state A is an accept state because every empty string (represents 0) is divisible by 4. Also note that since we want the empty string as well as the string with a singleton 0 to be accepted by the FSM, we have made state C the start state[3].

2.2 Closure: A Brief Introduction

We could have done the design of the last FSM faster if we had known something in advance. Let us say you knew that if you could accept a particular language, you knew how to accept its reverse, that is, any time you knew how to design a machine for a language, we had a mechanical process for designing an FSM that

[3]One might think - how will one know to design FSMs for problems harder ones than these? The answer we will give straight up in the beginning. This is a design process. There is no algorithm for developing an FSM for a given description of strings. There is practice involved in designing FSMs! The discovery process could take more time for some problems and less time for some others.

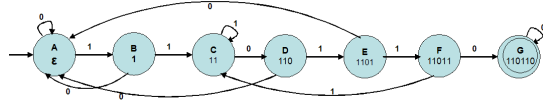

Figure 2.8: Finite state machine that accepts only binary strings containing the pattern 110110.

accepts only strings that are reverse of strings in the first language. For example, we were able to easily design an FSM for the set of strings that start with 00. Is there a mechanical process for designing an FSM for the set of strings that end in 00? The answer is fortunately yes. In fact there a bunch of operations, called *closure operations* such that if there is an FSM for a set of strings, there is an FSM for the set of strings that are obtained by applying the operation on each member of the set. Examples of closure operations are (a) string reversal (b) complement (c) union and (d) intersection. The closure operation is a key property that is studied under models of computation, not just because it is interesting mathematically, but because it gives an entire repertoire of tools to determine if a set of strings will be accepted by any model of computation and how to construct such a model.

For instance, what if toggle all the final and non-final states of the FSM for problem 5? Whatever used to be not accepted will now be accepted and *vice versa*. That actually is a good enough truth - FSMs are closed under complement. This is an informal closure proof pertaining to complementation.

Some of the big applications of FSMs are in string searching tools. Once the FSM is designed, it could be implemented as a program. There are no obvious ways to do the programming - there are clever and bad ways to do it. There are a host of programs to string matching, the first of which was by Knuth, Morris and Pratt. Next, we will consider a problem in this domain.

Problem 6 *Design a finite state machine that accepts only binary strings containing the pattern* 110110*.*

We can have a series of states that represent how much of the string we have seen. When we have seen the entire string, we will put a double circle (representing the accept state). We will use an ϵ to represent an empty string (some books instead use λ), which means we have no string at all. State A represents empty strings, state B represents the fact that the FSM has seen a 1, state C a 11 and so on. Figure 2.8 shows the corresponding FSM.

Given, this how do we construct an FSM that accepts only those strings that do not contain the string 110110?

Problem 7 *Design a finite state machine that accepts only binary strings that do not contain the pattern* 110110*.*

The way to do this is to flip the accept and non-accept states of the FSM for problem 6. It is very hard to explicitly construct an FSM for problem 7 from

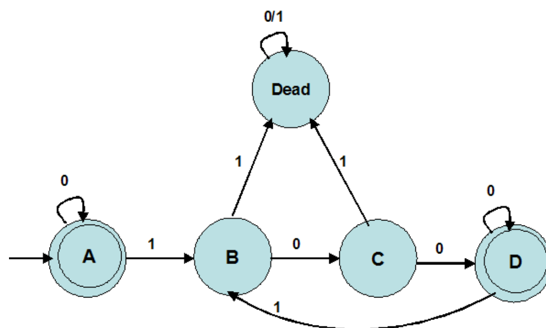

Figure 2.9: FSM that accepts only strings that have at least two 0's following every 1.

scratch. And it actually happens to contain 6 accept states. If we try giving meanings to each of these accept states, it would take a lot more time!

We will keep this example in mind, because when we later add on an extra arm called *non-determinism* to the above machine, we are going to give it power[4]. If we give it power, it turns out not to help it all, but just helps us write easier machines. We will need a proof that a machine with the power of non-determinism can always be converted back to something without the power. There is always a way to write some of the fancier machines to something without the fancy stuff.

Let us do another problem.

Problem 8 *Design a finite state machine that accepts only binary strings such that every 1 is followed by at least two 0's.*

We are doing this example, because in this machine, it is difficult to give semantics to the memory of each state right away. It is much more of a real time, left to right processing machine - it is not a recursive idea.

In Figure 2.9, we give a first cut solution to this problem. Notices that states A and D, are for all purposes, identical; state A embodies the semantics of state D which is: *the machine has not yet seen a 1 that has not been followed by two 0's.* We are pointing it out because that is the idea of minimizing machines (which we will discuss later). Minimization involves noticing groups of states that actually do the same thing, irrespective of where you start and merging them. Thus merging states A and D, we get he minimized machine as in Figure 2.10.

Keep in mind that there is no single finite state machines that alone accepts a set; there are infinite number of machines that will accept a set if at all there is one. There is a unique minimum machine though.

We will do one last example, before moving on to the next topic of non-determinism.

[4]Exercise

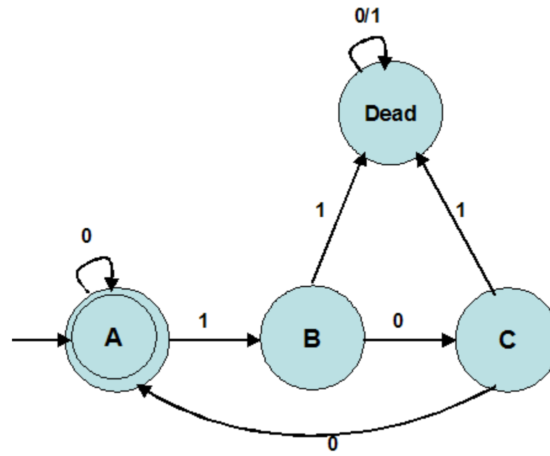

Figure 2.10: Minimized FSM that accepts only strings that have at least two 0's following every 1.

Problem 9 *Design a finite state machine that accepts only binary strings that are not divisible by* 3.

Now you really have to do division in some sense on the binary string. Typically, in division, you have to store a lot of information; you have to remember results that are proportional to the size of the string. And that does not seem like a constant! Real division of a longer string will take a larger amount of space and hence cannot be done with a finite state machine. So how do we go about solving this problem? We really do not have to perform division. We just need to figure out whether or not the string is divisible by 3. In order to do that, we just have to figure out what the remainder is (which is not generally taught). This essentially implies that you need as many states as there are remainders. But do you need to convert the binary number to base ten to get the remainder? It turns out that the answer is 'no'.

This is a much harder problem and you might sit on it for a few hours if you do not get the idea. Is the binary number '1' divisible by 3? So if the machine sees a 1 and the input stopped with that, the remainder will be 1. So that is what we will remember. Continuing from there, let us say we see a 0. So what we have seen now is 10, which is 2 in base 10 and is not divisible by 3. Let us not calculate this way. Rather, let us calculate how the remainder changes when we put a 0 at the end of a string which already leaves a remainder 1 when divisible by 3. Putting a 0 at end of a string, the value doubles and so does it remainder! Thus, the remainder of the new string will be 2, which we will store in another state.

Let us move on to the third spot. Suppose the next symbol we see is a 1. The new remainder (for 101) is obtained by doubling the previous remainder and adding 1. This yields $2 \times 2 + 1 = 5$ which implies the new remainder will

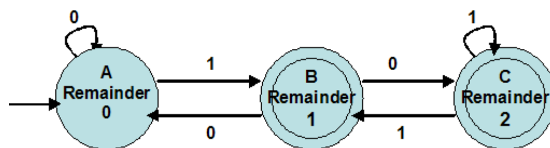

Figure 2.11: FSM that accepts only strings that are not divisible by 3.

be 2 ($2 = 5 \ mod \ 3$). Now let us say the fourth symbol is a 1. The remainder for 1011 can be again determined by only using information about the previous remainder, which happens to be $(2 \times 2 + 1) \ mod \ 3 = 2$. Thus 1011 is also not divisible by 3[5].

In Figure 2.11, we implement the above logic in an FSM that will determine when an input binary string is not divisible by 3. There is beautiful symmetry in this machine.

We did a whole bunch of examples that are accepted by finite state machines. A set of strings that can be accepted by a finite state machine is called *regular*. They are called regular because they tend to grow in regular intervals (we will learn more about the regularity property in the following sections). Regular sets are really well understood. There are machines and grammars that describe them. And there are also regular expressions that describe them. You encounter regular expressions very often in web development tools where you need to search for patterns such as specific urls, etc. Thus there are different ways of describing these machines. We will show equivalence between these different representations and nail down regular sets as best as they can be.

2.3 Examples of Non Regular Sets

Let us try to come with some simple examples that are not *regular*. The reason we are touching upon this topic is because we will not be able to appreciate moving up to *context free grammars* or *turing machines*, if we are under the impression that finite state machines could do everything. Why have unbounded memory if all you need is finite memory? Why make up a turning machine if finite state machines represent a computer? Finite state machine is actually like a computer with half its brain cut out. What kind of strings cannot be accepted by finite state machines that are easier to describe than just Java programs? Let us look at some famous ones.

What we cannot do with finite machines is count! We cannot count to an arbitrary number. In the FSM for problem 9, we counted up to 3. Even in the case of substring matches, we had a finite number of conditions and therefore had a finite number of counts. But determining a machine that accepts only binary strings having an equal number of 0's and 1's requires counting and hence cannot

[5]You could confirm that the number 1011 in base 10 is 11, which leaves a remainder 2 when divided by 3.

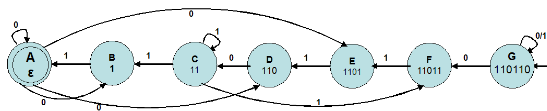

Figure 2.12: The *strange* FSM that accepts only binary strings containing the reverse of pattern 110110.

be modeled by a finite state machine. Even a simpler set of strings represented by 1^n0^n $n \geq 0$ cannot be modeled by a finite state machine[6].

How do we prove that there is no FSM for a given set of strings? Do we keep trying to come with one till we are exhausted? We need to prove this formally. And there is a really wonderful proof that there are sets that are not acceptable by finite state machines. This is proof technique is called *diagonalization* and it is used over and over again at the higher levels. The proof technique comes with a very abstract set. We will get to that in great details some time later in this chapter.

However, for finite state machines, there is an even easier method of proof that even strings as simple as 1^n0^n cannot be accepted by any FSM. This proof will be very constructive[7].

2.4 Non Determinism

What if we reverse all the arrows of the FSM in Figure 2.8? And switch the roles of the start state and accept state? We observe that there are two outgoing arrows on a symbol '0' for state G. We get a machine in Figure 2.12 that does not even make any sense! And how wrong can it be when it does not even make any sense? This nature of a machine can be actually ascribed semantics, and is called *non-determinism*.

Similarly, where would the start state for the FSM in Figure 2.11 be if we reverse the machine? The start state could be either of the two existing accept states B and C! This brings up another difficulty with reversing machines. We could define a dummy start state D and have ϵ transitions from that state to B and C. That is, you could go from the start state to either of states B or C without seeing any symbols. These are called ϵ-transitions (or sometimes λ-transitions). They also make a machine non-deterministic, because you do not know what the machine is going to do when the machine is at such a state.

The above idea of reversing really begs us to ask the question: *What does it mean to have a choice on the same symbol?* If we can make some sense of what

[6]Exercise: Another example set of strings that cannot have an FSM representation is the set of Fibonacci numbers. The nth Fibonacci number is computed as $F(n) = F(n-1) + F(n-2)$. Since determining the nth Fibonacci number entails some computation, these numbers cannot be modeled by any FSM.

[7]In general, the proofs in this tutorial are very constructive. They do not involve a lot of difficult mathematics but instead use very straightforward arguments.

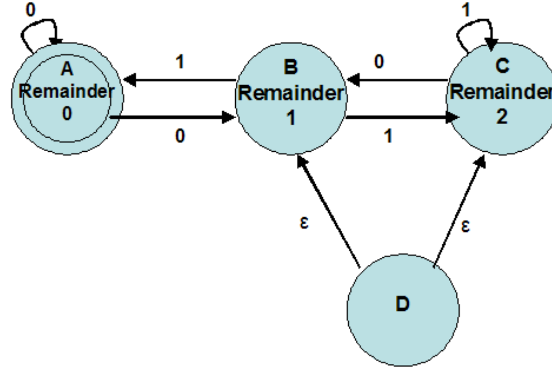

Figure 2.13: The *strange* FSM that accepts only reverse of binary strings that are not divisible by 3.

that means, may be the method actually does work? And it turns out that it does work. We just need to explain what it means to have the same symbols on two outgoing arcs on the same state. This idea again leads to non-determinism.

Non-determinism helps us realize that we can compute things faster than we used to realize without it. Non-determinism however does not give any new power; you could take any non-deterministic machine, and convert it back to a deterministic machine without having these funny duplicates. This is a nice thing to know because it lets us use these facilities without taking us out of the universe of finite state machines. It is the equivalent to learning machine language as your first programming language. And you understand programming through that, to the extent that you can completely program anything. And then somebody teaches you Scheme. And then you realize that you can do recursion much easier using Scheme. But there was not anything you could not have done with machine language that you can do with Scheme, though it is a lot easier to use. They are equivalent in power, though not equivalent in ease of use. That is the same case here. If we tack on non-determinism to a finite state machine, they have equivalent power, but one is much easier to use and can save us a lot of time.

We will now provide a formal definition of non-deterministic finite state machines.

Non-deterministic Finite State Machine *An NFA is represented essentially like a DFA:* $A = (Q, \Sigma, \delta, q_0, F)$ *where:*

1. Q *is a finite set of states.*
2. Σ *is a finite set of input symbols.*
3. q_0*, a member of* Q*, is the start state.*
4. F*, a subset of* Q*, is the set of final (or accepting) states.*

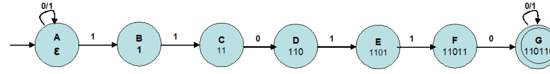

Figure 2.14: The non-deterministic FSM that accepts only binary strings that contain the pattern 110110.

5. *δ, the transition function is a function that takes a state in Q and an input symbol in δ as arguments and returns a subset of Q. Notice that the only difference between an NFA amt a DFA is in the type of value that δ returns: a set of states in the case of an NFA and a single state in the case of a DFA.*

.

As an example, we will design a non-deterministic machine for problem 6 with much ease. We restate problem 6 here for convenience.

Problem 10 *Design a finite state machine that accepts only binary strings that contain the pattern 110110.*

Somewhere in this string, if there is an occurrence of 110110, we should accept it (without explicit knowledge of where in the string this pattern occurs). We we will keep reading in the symbols till we get to the beginning of that pattern. When when we do get to that pattern, we will read it using states B through G and accept that string in state G. That is how simple it can become using non-determinism! Figure 2.14 shows the non-deterministic machine for problem 6.

We need to rigorously define what this strange machine means and what it really accepts. How do we really interpret what strings coming into the machine should be accepted or rejected? And then we will convince ourselves that we can turn such machines into regular deterministic machines.

The definition is: If you look at any string that is given to you as a candidate and figure out some choice through the machine that ends up in a final state, then you accept that state.

There can be lots of choices that do not end up in a final state. But if you can find one set of choices that end up in a final state, then we accept the string. For instance, for an input string, 1111011000. What if you tried the following sequence of state transitions: $AAAAABCD$ - you are dead when you reach the 8th symbol. That was a bad choice and in general, you can potentially hit upon many bad choices. But that does not mean that the string should be rejected; ending up in a rejection state on one choice of moves does not mean you reject. Here is one choice of the sequence of state transitions that will accept this string: $AABCDEFGG$.

The only way you reject a string is if none of the choices get you to the final state; if a choice gets you to the final state, you accept the string. If none of the choices get you to the final state, you reject the string. In the above example,

since there was a transition of states $AAAABCD$ that lead to the final state, the machine can be understood to accept the string 1111011000. In general for any string that contains the pattern 110110, the machine can keep looping around in state A, till the pattern is observed and transition through the sequence of states B through G, when the sequence 110110 is observed. Thereafter, the machine keeps looping in the state G. On the other hand, if any string does not contain the sequence 110110, the machine cannot sneak to the final state G, using any possible sequence of state transitions and hence will not accept the string.

2.5 Computational Comparison of Deterministic and Non-deterministic Machines

Non-determinism takes a little time to get used to. Lot of things that work with determinism, do not get quite analogous with non-determinism. Here is one. What if we toggled all the states in a non-deterministic finite state machine? That is, we change all final states to non-final states and all non-final states to final states? When we do this in deterministic machines, we get a machine that accepts only the complement of the set that was accepted by the original machine. What does a non-deterministic machine such as in Figure 2.14 accept if you toggle the final and non-final states? It will accept every possible string! Therefore complementing in non-deterministic machines does not work by toggling the states. The only way you could get the complement for a non-deterministic machine is to first convert it into a deterministic machine and then complement it. The reason is that the semantics of a non-deterministic machine is not symmetric; you accept something if there exists a way to get to the final state and do not reject it if there is a way that does not lead to the final state. So it is this asymmetry in a non-deterministic definition that makes this complementing not work.

For the example problem 6, we will pictorially compare the computations on the string 1110110 of the deterministic finite state machine in Figure 2.8 with the equivalent non-deterministic finite state machine in Figure 2.14). The comparison is shown in Figure 2.15. Note that the string has 7 symbols. The finite state machine moves sequentially from one state to another through exactly 7 states till it reaches the accept state. The equivalent computation for the non-deterministic machine can be represented as a tree, where a 2-way branching at each node represents the choice of move that the machine has. The computation for the non-deterministic finite state machine looks like a tree that represents all the parallel computations that can be going on at the same time; non-determinism lets you do 'or's in parallel. The string will be accepted if there is a path somewhere down that ends up in a final state. You do not accept the string when all the paths from the root have no accept state at the bottom. In all this non-deterministic machine need to perform 20 computations.

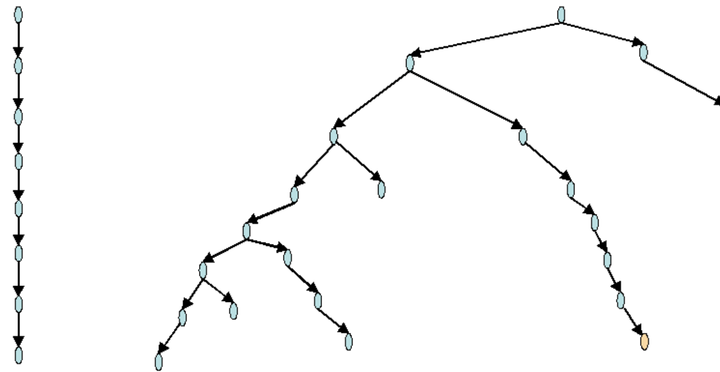

Figure 2.15: Comparison of the number of computations by the deterministic (left) and non-deterministic (right) finite state machines (for problem 6) on the input string 1110110.

2.6 Conversion From Non-deterministic to Equivalent Deterministic Machine

As another example, consider problem 11.

Problem 11 *Design a finite state machine that accepts only binary strings that end in* 00.

We will make a non-deterministic machine for the above problem. And then we will convert it into a deterministic machine by a very mechanical process that we could write a program to do. This will illustrate that any non-deterministic machine can be converted into a deterministic machine and therefore that a non-deterministic machine gives us convenience but no extra power. This machine will be similar to what did for the previous problem; it will have one state (the start state) A that will chew up as many zeros as it wants[8]. And whenever the non-deterministic machine sees a sequence of two ending zeros, it will land in the accept state C. Note that it is often easy to construct a non-deterministic machine that will accept all strings it is supposed to and also accept some strings that it is non supposed to. One has to be very careful about this point while constructing non-deterministic machines. In the machine in Figure 2.16, since there are no outgoing arrows from state C, we can be rest assured that the machine cannot accept any string that does not end in 00[9].

We will now present a mechanical process for converting the above non-deterministic machine (NDA) to an equivalent deterministic machine (FSM).

[8]This is similar to trying them all at the same time and therefore non-determinism is very often referred to as *guessing*.

[9]Leaving out an out-going arrow from a particular state in a non-deterministic machine means that any symbol observed while at that state would lead to a non-accepting state and implies that the sequence of guesses that lead to the state was bad. The symbol corresponds to something like an 'interrupt' or an 'unknown instruction'.

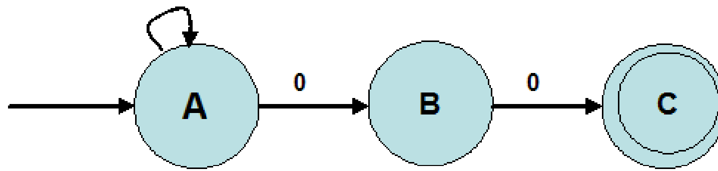

Figure 2.16: Non-deterministic FSM that will accept only strings that end in 00.

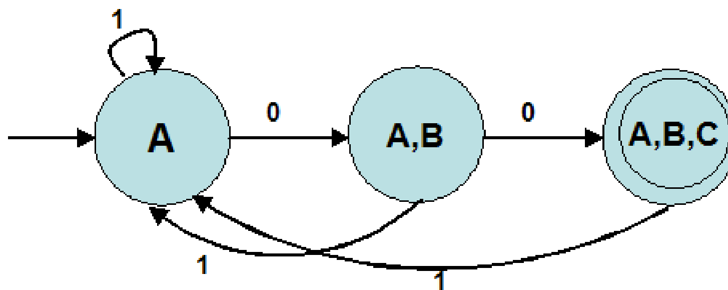

Figure 2.17: Deterministic FSM that will accept only strings that end in 00 by keeping track of the set of states the equivalent NDA in Figure 2.16 might be after reading any number of symbols.

The way to do this is to first label the states so that you have some way to refer to them. The states in the above machine are already labeled.

We will construct an FSM that we keeps track of the set of states the NDA might be in after reading any possible string. Let us say we start at state A in the NDA. Where will the NDA be when it sees a 0? It could be in either A or B. We will remember that in a new state in the FSM, denoted A, B. When the NDA gets a symbol 0 in A, B, it could transition to either A or B or C. We will represent this possibility in a new state A, B, C in the FSM. On observing a symbol 1 on A, B, the NDA could transition only to state A. If in A, B or C, the NDS gets a 0, it will go back to one of A, B or C and therefore, the FSM will have a loop on the state A, B, C for the symbol 0. Similarly, on a 1 while in state A, B, C, the FSM will transition to state A. Figure 2.17 is a deterministic machine for problem 11 by keeping track of where the NDA in Figure 2.16 might be after reading any number of symbols, assuming it does not crash. To make the FSM accept exactly and only those strings that are accepted by the NDA, all those states in the FSM that include at least one accept state from the NDA are designated as accept states. This is because of the semantics of acceptance of a string in a non-deterministic machine: If there is any sequence of state transitions that could lead to an accept state in the non-deterministic machine, the string is accepted by the non-deterministic machine.

Note the subtle difference between this machine and the machine in Figure 2.7 for problem 4. While the FSM in Figure 2.7 accepted even empty strings, this FSM does not.

Let us run through an example string 0110. Running this string through the NDA, the only states you could land up in are A or B. Running the same string through the deterministic machine, the only state the machine will possibly be when it is done is in A, B. We see a correspondence here.

The above process of converting a deterministic finite state machine to an equivalent non-deterministic machine can always be done. When we did this process, the number of states remained the same and therefore there was no trade off. We got the advantage of determinism owing to the conversion. There could be a trade off though and the number of states in the deterministic machine could explode. By how much could the number of states explode? This can be answered by examining, what is represented in the states of the deterministic machine; the states in the deterministic machine represent all subsets of states that the non-deterministic machine could be in. If the non-deterministic machine has n states, the number of possible subsets are 2^n. In the above example, we had only three subsets from a collection of $2^3 = 8$ subsets which was very modest.

In general, the payoff for going from non-determinism to determinism is an exponential growth in the number of states. We do not really care about this explosion if it is finite. But when we talk about machines in the Turing Machine layer, the exponential trade off in states turns into exponential growth in time, though it is the same idea. That is why, we convert non-deterministic algorithms to deterministic algorithms, we end up getting exponential time deterministic algorithms when we had polynomial non-deterministic algorithms to start with. In this chapter, we saw this trade off in its most trivial form.

Let us formally prove the equivalence:

Every NFA has an equivalent DFA: *PROOF: If a language is recognized by an NFA, then we must show the existence of a DFA that also recognizes it. The idea is to convert the NFA into an equivalent DFA that simulates the NFA. Let $N = (Q, E, \delta, q_0, F)$ be the NFA recognizing some language A. We construct a DFA $M = (Q', E, \delta', q_0', F')$ recognizing A. Before doing the full construction, let's first consider the easier case wherein N has no ϵ arrows. Later we take the ϵ arrows into account.*

1. $Q' = \mathcal{P}(Q)$.
   Every state of M is a set of states of N. Recall that $\mathcal{P}(Q)$ is the set of subsets of Q.

2. *For $R \in Q'$ and $a \in \Sigma$ let $\delta'(R, a) = \{q \in Q | q \in \delta(r, a)\ for\ some\ r \in R\}$. If R is a state of M, it is also a set of states of N. When M reads a symbol a in state R, it shows where a takes each state in R. Because each state may go to a set of states, we take the union of all these sets. Another way to write this expression is*

$$\delta'(R,a) = \bigcup_{r \in R} \delta(r,a)$$

.

3. $q_0' = \{q_0\}$.
   M starts in the state corresponding to the collection containing just the start state of N.

4. $F' = \{R \in Q' | R \text{ contains an accept state of } N\}$.
   The machine M accepts if one of the possible states that N could be in at this point is an accept state.

2.7 Closure

Closure *A family of sets is called closed under an operator if the application of the operator on all the elements of a set will yield another set in the same family.*

Regular sets are closed under some operations. The more are the number of closed operations[10], the easier it will be to determine whether or not a set of strings is regular. We touched upon closure under complementation in a previous section. We will delve on it in some more details in this section. We will also address closure under string reversal as well as under union and intersection of regular sets.

Recall problem 8:

Problem 8 *Design a finite state machine that accepts only binary strings such that every 1 is followed by at least two 0's.*

The reason we want to go over this is because it gives another review of an example of a finite state machine. And then, we will try to make up a machine that accepts the reverse of this language. In English, the reverse of this language can be described as:

Reverse Language of Problem 8 *Set of binary strings such that every 1 that shows up has at least two 0's preceding it.*

Instinctively, if we started with the above reverse problem, it would seem a lot harder than the original problem. But that instinct is actually wrong; it is easy to do the reverse problem though one might not be able to figure

[10] Exercise: You will find that for every new exam, the instructor makes up a new operation. Other operations are union, intersection, min, max, prefix, suffix, half, log, square root. At one point in the late sixties or early seventies, there was a paper that basically enumerated a list of operations under which finite state machines are closed under and then tried to characterized them. For your own knowledge, finite state machines are pretty much closed under any operation you can think about except a few. The bottom line is that you are pretty safe with finite state machines.

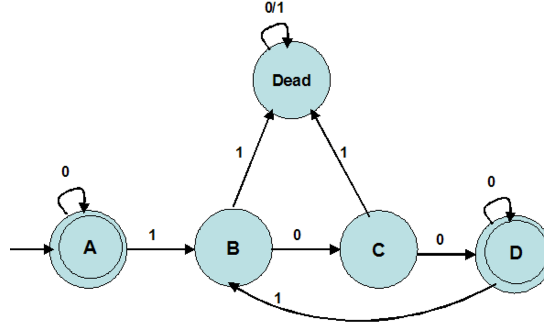

Figure 2.18: Minimized FSM that accepts only strings that have at least two 0's following every 1.

that out at first glance. However, we had found the solution to problem 8 very naturally in Figure 2.10. The fact that knowing the FSM for a set helps you figure out the FSM for its reverse makes the second problem a lot easier and mechanical. This is an advantage of understanding the closure properties. The problem of reversing a machine, is a very general idea. But in doing this, we will come up with a non-deterministic machine again, which is a good motivation for non-determinism.

Below (in Figure 2.18), we reproduce the FSM for problem 8 from Figure 2.10. For the following pedagogical reason, we will refer to the non-minimized bigger machine (Figure 2.9) instead of the equivalent minimized machine (Figure 2.10) - the example reversal procedure we will do next becomes a little more involved and general if the machine is bigger.

We will now go ahead and try to reverse the above machine, using it as a template to show the general procedure for designing a machine that accepts the reverse of a language. It is not straightforward doing this surgery to get a deterministic machine straight up. However,

1. reversing the arcs of the deterministic machine in Figure 2.18 and
2. swapping the start and accept states

is a completely general mechanical process for reversal that yields a non-deterministic machine, as in Figure 2.19.

Does the presence of two start states give any extra power over the deterministic machine? No, and that is what we need to ensure. We will replace the two start states with a single one and enable transitions to these two states from the new start state on empty transitions. Moreover, the vertex *Dead* with its accompanying transition arcs is its own strongly connected component, *i.e.*, it has no incoming edges from any of the remaining vertices and there is no need to draw it. We modify the non-deterministic machine in Figure 2.19 along these lines to the non-deterministic machine in Figure 2.20.

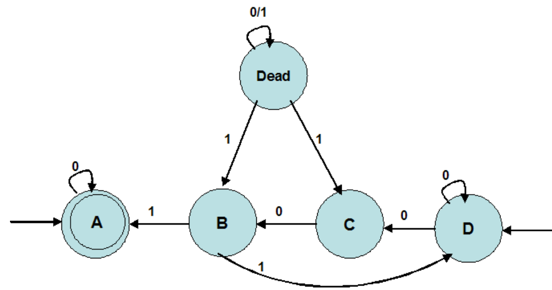

Figure 2.19: Non-deterministic machine that accepts only strings that have at least two 0's preceding every 1.

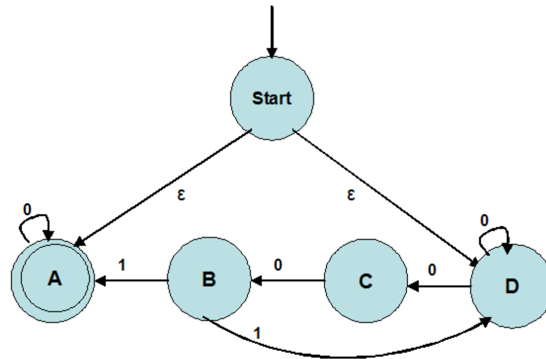

Figure 2.20: Modified non-deterministic machine that accepts only strings that have at least two 0's preceding every 1.

The reversal of a deterministic machine yielded a non-deterministic machine. Mimicking the method from the previous section, we will try to convert the non-deterministic machine in Figure 2.20 to a deterministic machine that is equivalent. We will then try to interpret the deterministic machine. By the mechanics of the reversal process, we will come up with a deterministic machine that actually accepts the reverse language of problem 8. This idea can get very deep when you can actually come up with solutions to problems that you would not have otherwise thought of. There is a really interesting thing about reversing machines and it relates to the fact that some times when you reverse machines, pieces get cut off.

Note that by applying the mechanical procedure for reversal to the minimal finite state machine for this problem (as in Figure 2.10 we could have constructed a non-deterministic machine that has fewer states than in Figure 2.20. We will mention here is that there is no notion of a *minimal non-deterministic* machine as is there with a *deterministic machine* (where you have a unique minimum). Because, in the case of non-deterministic machines, you could have several equivalent machines that have the same number of states.

Next, we will convert the non-deterministic machine in Figure 2.18 into a deterministic machine using the completely mechanical process described in the previous section. The procedure involves keeping track of exactly where the non-deterministic machine might be after reading any sequence of symbols. Where might the machine end up if the first symbol it sees is a 0? Depending on where it chooses to start from (one of states A or D), it will land up in a possible one of any state from the set $\{A, D, C\}$. When it sees a starting symbol of 1, there is none of the states were it might end up. Or in other words, on seeing a starting symbol 1, the machine will end up in one of the states in the empty set ϵ. This empty set is what corresponds to the dead state ϕ. The only place to continue exploration from is $\{A, C, D\}$. Thus, keeping track of the possible states of the non-deterministic machine, we can come up with the equivalent deterministic machine in Figure 2.21.

Since A is an accept state, the state corresponding to the set $\{A, D, C\}$ will also be an accept state. This is very important to realize; the deterministic machine should accept a string whenever the non-deterministic machine could be in one or more final states on seeing the same string. Along similar lines of reasoning, the start state should also be an accept state. One could also associate semantics with each of the states in the deterministic machine in Figure 2.21: (a) State $\{A, C, D\}$ represents that one 0 has been seen, (b) State $\{A, B, C, D\}$ represents that two 0's have been seen.

Here is an interesting folklore about theoretical computer science that deals with the reversal technique[11]. If you want to minimize a minimal finite state machine, reverse the machine, convert the non-deterministic machine to a deterministic machine, reverse it again and finally convert the non-deterministic machine again into a deterministic machine; the final machine is equivalent to the machine we started with, it is deterministic and is always minimal! As far

[11]Exercise

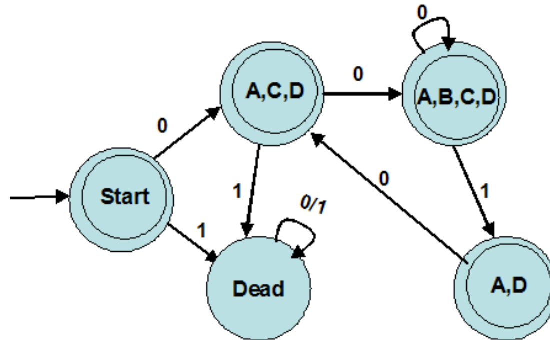

Figure 2.21: Deterministic machine that accepts only strings that have at least two 0's preceding every 1 and is equivalent to the non-deterministic machine in Figure 2.20.

as it is known, this fact is not published anywhere. However, you have to be careful to chop off all the disconnected components at every step of the conversion process. Reversal can introduce the loss of dead states as well as other things. Characterizing exactly what it is that you lose, you end up losing states that are identical. Though this is not a very efficient process for minimization, it always works nevertheless! There are polynomial time algorithms for minimization that we will deal with in the remainder of this chapter and that will give us a more efficient recipe for minimizing deterministic machines.

2.7.1 Closure under Union

The proof for closure under reversal is a tough proof that lead us into all sorts of alleyways. We will now do some that are easier (closure under complementation was already covered and was quite easy).

Suppose we are given two sets of strings

$Set_1$1 *Set of strings having even number of 0's.*

Set_2 *Set of strings containing the sequence 101.*

Both the above sets are regular sets and have finite state machines. You could easily draw them and let us say they are FSM_1 and FSM_2 for Set_1 and Set_2 respectively[12]. FSM_1 should have 2 states and that for the latter should have 3 states.

What if we want a finite state machine FSM_3 for the following set, Set_3?

Set_3 *Set of strings that either have an even number of 0's or contain the sequence 101.*

[12]Exercise

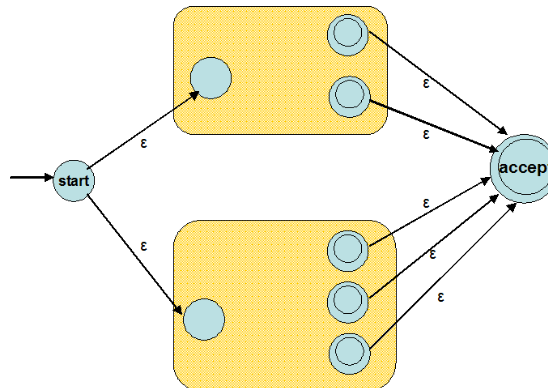

Figure 2.22: Non-deterministic machine that accepts the union (Set_3) of the set of strings (Set_1 and Set_2) by utilizing the finite state machines FSM_1 and FSM_2 as black boxes.

If you try to do this from scratch, you need to keep track of two things simultaneously. And you can get confused. But if you raise your level of abstraction a little bit and think of Set_3 as a union of Set_1 and Set_2, you can conjure up the non-deterministic finite state machine FSM_3 mechanically without caring for what FSM_1 and FSM_2 actually look like as in Figure 2.22. One only needs to know what the initial and final states of FSM_1 and FSM_2 are to accomplish this. When the machine is done with any of the two individual finite state machines, it jumps to the final state of the combined machne (concentric circles situated outside the yellow boxes).

Converting this to deterministic machine might make the resulting machine look a little ugly. If you went through the trouble of converting this non-deterministic machine to a deterministic machine, you will do exactly what is called the product[13]. All that product does is pair states together from the two machines in every possible way. But if you take the simple mechanical route above, you need not bother about the product of two machines!

There is one more power of non-deterministism. We can see that union is a closed operation without dealing with any complicated arguments, such as the fact that *non determinism is equivalent to determinism, etc.*

2.7.2 Cosure under Concatenation

Consider the problem of constructing a finite state machine that accepts only the following set of strings:

Set_4 *Set of strings such that each string is a concatenation of a string from* Set_1 *and a string from* Set_2.

[13]Exercise: Tutorial

That is, we require an FSM that accepts only those strings that have a prefix with an even number of 0's and a suffix containing the sequence 101. The FSM can be constructed easily as follows: Construct the FSMs FSM_1 and FSM_2 for Set_1 and Set_2 respectively. Connect all the accept states of FSM_1 with an ϵ move to the start states of FSM_2. The ϵ move is very powerful; it lets you stay in FSM_1 as long as required and then jump from the accept state of FSM_1, looking for a string in Set_2. The accept state of the 'concatenation' machine will be the accept state of FSM_2. If you get rid of the non-determinism, the machine can become really complicated[14].

2.7.3 Closure under Complement

Complementat is very straightforward, in which you just toggle all the final and non-final states. This was already discussed earlier.

2.7.4 Closure under Intersection

For any sets A and B, let us recollect the DeMorgan's law:

$$A \cap B = \overline{A} \cup \overline{B}$$

The law is all about boolean algebra. The key thing is if we have two machines, one for A and another for B, we can get a machine that does their intersection by a simple application of the DeMorgan's law.

1. We can complement the FSMs for A and B (since we already learnt how to complement deterministic machines)

2. then union the resulting two FSMs to get a non-deterministic machine (we just did union!)

3. convert the non-determinstic machine into a deterministic machine (which we studied earlier)

4. and finally completement the deterministic machine.

Note that step 3 is required because we learnt to complement only determinstic machines. There is more work for intersection, but it can definitely be done. Thus, regular sets are closed under intersection.

We should know that for any language, we cannot have closure under just two out of the three properties of union, intersection and complement, by virtue of the DeMorgan's law. Thus, if the union and intersection of two sets in the language belongs to the same family, so will complement and so on. Either the language is closed under all of them or just one of them. And this is going to happen in the levels above FSMs. As mentioned earlier, FSMs are closed under most operations.

[14]Exercise: construct the non-deterministic and deterministic machines for this problem

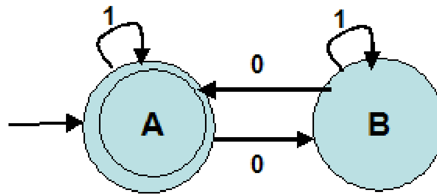

Figure 2.23: Deterministic machine that accepts the set Set_1 of strings.

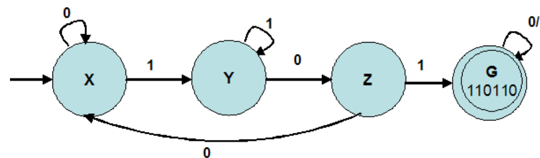

Figure 2.24: Deterministic machine that accepts the set Set_2 of strings.

The above method for intersection is not the nicest way to execute it though. Any time we have to move from non-determinism to determinism, there is an exponential potential. We will see a better and direct way to perform intersection[15] using the notion of product of FSMs[16]. This will reinforce the notion of non-determinism one more time.

Figures 2.23 and 2.24 show two FSMs.

We will keep track of what both these machines are doing as they read symbols. When they start, these machines are in the pair of states, $< A, X >$. Let us forget about final states for now and simulate the condition of the two machines at any point of time. On encountering a 0, the machines transition to the pair $< B, X >$ and on a 1, they transition to $< A, Y >$. From $< B, X >$, on a 1, it will transition back to $< A, X >$ and so on. When you finish this way, you get a deterministic machine.

How do we interpret this to do union or intersection in a way that is very different from how we did before. The above procedure keeps track of where both the machines are. If you want a machine that is a union, that is, it accepts any string that is either accepted by machine 1 or 2, then the final state of the resultant machine should be any state that has either a final state from the first machine or a final state from the second machine (Figure 2.25). But if you wanted the intersection of the two, the final state of the resultant machine should be all the states whose 'representation as a pair' corresponds to a pair of accept states in the two respective machines (Figure 2.26).

How big might this get? Every new state in the new machine will correspond to a pair of states in the old machine. If there are M states in the first (here 2)

[15]Tutorial and Exercise

[16]Pedagogical tools for Theoretical Computer Science at Duke University developed with NSF project. You can download the tools and play around.

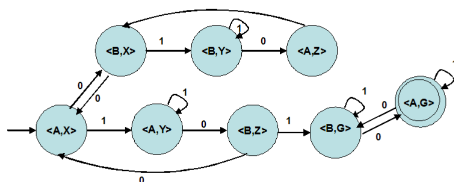

Figure 2.25: Deterministic machine that accepts the union of Set_1 and Set_2 of strings.

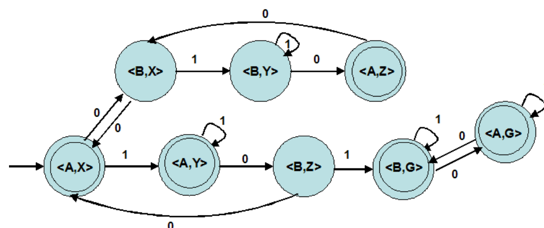

Figure 2.26: Deterministic machine that accepts the intersection of Set_1 and Set_2 of strings.

and N in the second (4 in this case), there will be $M \times N$ (here 8) states in the worst case with the resultant machine. Note that if you build a non-deterministic machine for the union or intersection operators using the technqiues discussed earlier, the number of states will most probably explode exponentially when you convert the resultant (from union or intersection) non-deterministic machine to a deterministic machine. The key is in noticing that while performing intersection or unions, the subsets always pairs.

Figure 2.26 shows the resultant machine.

2.7.5 Alternating Finite State Machines

A salient feature of non-determinism is that it mixes well with ORs (union) and not with ANDs (intersection). And this going to come all the way up in Turning machines. Recall that in non-determinism, you could have two outgoing arcs from a single state on the same symbol. And we defined the semantics as: *you accept if there is some way to get to the final state.* In other words, every state is an OR state. This definition makes non-determinism more ameanable to ORs. If we define it in terms of getting to a final state from all the paths (by defining every state as an AND state), the machine will bear more natural relationship with ANDs.

People do definte finite state machines with "AND" states. Some times people mix the two nodes; some states are labeled OR and some are labeled AND. These machines are called *alternating finite state machines.* And you define whether the machine accept a language in the following terms: for any

string that goes through an OR state, there should atleast be one path that leads to a final state and for any string that goes through an AND state, all the paths passing through that state should lead to the accept state. And it turns out that these alternating machines are no more powerful than the deterministic finite state machines. This alternating idea goes up to Turning machines about which we may talk much later in the tutorial.

2.8 Equivalent Representations of Regular Sets

To summarise the discussion so far, we discovered that non-deterministic finite state machines can be converted into deterministic finite state machines. Either of them are appropriate ways of looking at the set of finite state machines. We will sketch a bigger picture of where we are heading for the rest of this chapter. In web programming and pattern matching, we see *regular expression* very often. Regular expression is another way of defining the set that finite state machines can accept. This is not obvious. We will see how any deterministic finite machine can be converted into a regular expression. But this does not complete the equivalences. We will complete the triangle by showing that any regular expression can be converted into a non-deterministic machine. These are three different windows to the same picture. And in computer science you desparately want different views to the same thing. Depending on the requirements, one of the alternatives will seem easier. And more the number of tools, the easier it will be to cater to a requirement.

But there is a fourth thing to these three, called regular grammars, which are sometimes also called linear grammars. These are also equivalent to all the three other representations. We will fit regular grammars into the picture by showing their equivalence to deterministic machines. Thus, we will be able to look at finite state machines from all the different possible viewpoints. There is no such thing as an analog for regular expressions as we go up the hierarchy - it simply disappears. But there is such a thing as non-determinism and their grammar as well as machine analogs. Grammar and machine analogs rise all the way up the hierarchy up to the Turning machine. And this grammar and machine parallel all the way up to Turning machines is called the Chomsky hierarchy. The interesting thing is that grammars and machines do not look alike but they always come in pairs!

2.9 Non-regular Sets and Introduction to Pumping Lemma

What about characterization of the set of strings that cannot be modeled by a finite state machine? Fibonacci numbers in binary cannot be acceptable by a finite state machine. Fibonacci numbers form a sequence defined by the following recurrence relation:

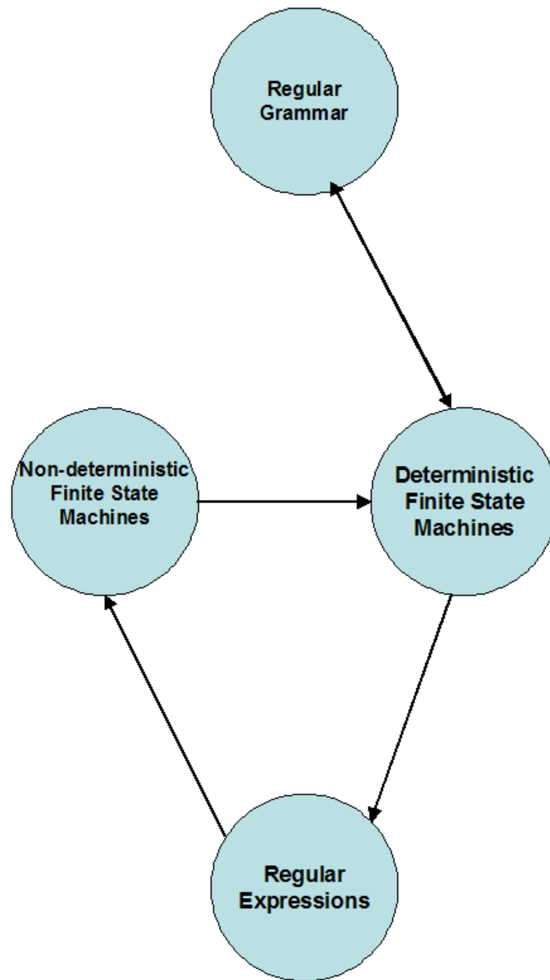

Figure 2.27: Equivalent representations of regular sets.

$$F(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases} \tag{2.1}$$

Similarly, strings containing an equal number of 0's and 1's cannot be accepted by a finite state machine. Anything that requires counting and arithmatic with more than finite storage cannot be captured using FSMs. Then the question is what can be captured by finite state machines? Definitely, problems for which you can write regular expressions can be modeled using finite state machines. It will also be nice to show an example which cannot be accepted by a finite state machine.

Proving you cannot do something is much harder than proving that you can do it. Because potentially we will require an infinite number of trails. Hence, to do this efficiently, you will need a trick.

Consider the set[17] $0^n1^n | n \geq 1$. This set includes the empty string ϵ, 01, 0011, 000111, *etc.* Our claim is that there is no finite state machines that accepts this set. We can try to make one *but keep in mind that the FSM should not include strings it is not supposed to accept.* Remember that in general, the more an FSM includes, the more it is likely to make a mistake. You will find that you will have to make the finite state machine more and more powerful in order to accept this language. And what do you get when you make an FSM more powerful by allowing an infinite number of states (by giving some external data structure to work with)? If the data structure happens to be an array, you get a turing machine. If the data structure is 2 stacks, you again get a turing machine! If you give it a single stack, you get a push down automaton.

Can we systematically convince ourselves that there can be no finite state machine that accepts the set $0^n1^n | n \geq 1$?[18]. Let us say some one (Adversary) does come up with a machine for this problem. Then we will refute this claim through a simple dialogue:

Question: Tell me the number of states in your machine.

Adversary: The machine has 24 (or any other fixed number) of states.

Question: Ok do me a favour. Take the string $0^{24}1^{24}$ and run it through your machine. You would admit that since there are 24 0's here, somewhere in those 24 symbols, you have to encounter the same state twice. This is because when you start from the initial state and keep going to new states, you have only 23 moves before you encounter a state for the second time. Can you tell me how many zeros got read before the loop (path between the first and second encounters of the state).

Adversary: There were 17 zeroes before the loop, 3 zeroes within the loop (note that the loop could be just one 0) and 4 zeroes after the loop.

[17] In this tutorial, exponentiation will almost always mean repetition.

[18] If a student goes home and spends all night trying to build a machine for this problem and informs his friends that he has obtained a machine for this problem, he is probably deluded because of lack of sleep!

Question: So let us split the string as $0^{17}0^{3}0^{4}1^{24}$. There are three parts to the computation of the string; the part before the loop, the loop itself and the part after the loop. Now that you have admitted, try this string on your finite state machine: $0^{17}0^{3}0^{3}0^{4}1^{24}$. What will your machine do? It is going to do the same thing it did before on the first 17 symbols, loop on the next 3 just like it did earlier and then again loop on the next 3 (since it sees the exact string again) and thereafter continue with the earlier path. If the earlier string leads to the final state, the new string should also lead to a final state. What is wrong with this? You were supposed to accept only those strings that have an equal number of 0's and 1's. You started out with $0^{24}1^{24}$. And now I have convinced you without even looking at your machine that the machine has to accept the string $0^{17+3+3+4}1^{24}$ that is $0^{27}1^{24}$. Therefore your machine is bogus - not because it does not accept all strings 0^n1^n but because it accepts more than it is supposed to.

Note that the dialogue is important - the number of states are asked apriori and cannot be changed when faced against the wall. The argument here forces somebody to come up with a hypothetical machine and in a dialogue forces them to admit: Oh! I might have accepted all the strings that you want. But you seem to have convinced me that the machine accepts more than I really want. And this idea is the heart of something called the *puming lemma* which we will discuss next. It is called so because here you pump up the loop. In the same vein that regular sets are termed so because you want to pump up substrings at regular intervals and obtain other members of the set. Regular sets string out at very linear intervals. Therefore anything that grows faster than linear is never regular. Following this logic, we can guess that 0^{n^2} is not regular because n^2 is not linear.

This pumping lemma is also referred to as *diagonlization* and is a jack hammer of a tool that always can spit out a set outside of its collection. We could apply diagonalization on FSMs and discover that there is a set of binary strings that represent FSMs that do not accept themselves. And that set could not possibly be accepted by any FSM[19].

2.10 The Pumping Lemma

Pumping lemma is used to prove that a set is not accepted by any finite state machine; that the set exists outside the horizon of finite state machines. And it works in the following way. It turns out that if a set is accepted by a finite state machine, it should have the property that we should be able to pump it up somehow. In what follows, we will get more specific about this point. Any set that does not have the pumping property, it could not have come from a finite state machine. This lemma is usually written down in the forward way, that is, *regular set⇒pumping property.*

Pumping Lemma: Let L be a regular set, and let FSM be the corrresponding

[19] This idea is akin to the case of a barber who can shave everybody except himself

finite state machine with number of states given by $|FSM|$. The pumping lemma states that $\forall z \in L$ where $|z|$ (the number of symbols in z) is greater than or equal to $|FSM|$, $\exists v, w, x$ such that $|vw| \leq |FSM|$ and $|w| \geq 1$, and $\forall i \geq 0$, $vw^i x$ is also[20] in L.

Note that exponentiation here is simply repeated concatenation. It is much harder to understand the lemma this way than through the dialogue form discussed in the previous section. In words, the lemma states that there exists a way to split z into three parts where the first part v is the sequence of symbols that occur before the first loop, w is the substring that corresponds to the first loop (the first loop cannot get past the first $|FSM|$ symbols)[21] and x is the rest of the string such that repeating w any number of times yields a string in the regular set L.

In practice, the lemma is used in its equivalent contrapositive form: *If you present a set, and if it does not have the pumping property, we can conclude based on the pumping lemma that it cannot be a regular set.* That is, $\neg$*pumping property*$\Rightarrow\neg$*regular set*. The contrapositive can be precisely stated as follows. Recall that pushing a not sign into an existential quantifier yields a universal quantifier.

Contrapositive of Pumping Lemma: $\forall n$ which equals the number of states in any machine for a set L, if the following conditions hold, L can not be a regular set. The conditions are: $\exists z \in L, |z| \geq n$, such that $\forall v, w, x$ with $z = vwx$, and $|vw| \leq n, |w| \geq 1$, $\exists i \geq 0$ such that $vw^i x \notin L$.

In other words, no matter how you split up z into three pieces, the prover does not get to choose where to split, but the adversary gets to choose the split. In the proof, we have to be able to make our argument based on the fact that the loop might appear anywhere, and the adversary has to tell us where the split is. We need to corner the adversary so that he do not have too many cases to choose from[22].

The pumping lemma is used in the above form to prove that a set is not regular; any set that cannot be pumped up at regular intervals is not a regular set. Note that there are non-regular sets that have the pumping property. Thus, one cannot always use the pumping lemma to prove that a set is not regular. It is not enough to examine the pumping property of a set to prove that it is regular. An *if and only if* condition exists only for regular sets that have an alphabet of a single symbol (say 0); there is a linear characterization for all regular sets with a single symbol alphabet[23]. The characterization is that

[20] $i = 0$ means you avoid the loop altogether. There are occassional examples where the pumping number i you use is 0.

[21] There are versions of this lemma statement that do not insist on w corresponding to the first loop

[22] It is like a boxing fight where you want to give the opponent the least maneuverability possible. The opponents represent the universal quantifiers while the boxer in first person represents the existential quantifier.

[23] Exercise: Extra Credit Problem.

each string should look like 0^{mx+b}, where x is a variable and m and b are real constants.

Example 1: Palindromes

We want to show that the set of palindromes, strings that read the same from both left to right and right to left, does not satisfy the pumping property. We will follow the dialogue as illustrated earlier.

Question: Tell me the number of states in your machine.

Adversary: The machine has k states (for any k).

Question: Consider the string 0^k10^k. This choice nails the adversary because the only choice of vw is 0^p where $p \leq k$. For any choice $v = 0^{p-p_1}$, $w = 0^{p_1}$, where w is a loop and $p_1 \geq 1$, $vw^ix = 0^{(p-p_1)}0^{(i\times p_1)}0^{(k-p)}10^k$ is not a palindrome[24].

Thus, by the contrapositive of the pumping lemma, the set of palindromes is not regular.

Example 2: Quadratic Strings

Consider the set of strings over the single symbol alphabet $\{0\}$, given by $L = \{0^{k^2} | k \geq 0\}$[25]. We will show that this set is not regular.

Question: Tell me the number of states in your machine.

Adversary: The machine has n states (for any $n \geq 1$).

Question: Consider the string 0^{n^2}, which is in the language L and which is longer than n symbols ($|0^{n^2}| \geq n$). Let $0^{n^2} = vwx$, where $|vw| \leq n$ and $|w| = m$ such that $n \geq m \geq 1$. The value of m is determined by the substring of vw that contains the loop. We need to pick an i and pump it up and our choice will be 2. $vw^2x = 0^{n^2+m}$. *Claim:* $n^2 + m$ is not a square, or in other words, $0^{n^2+m} \neq 0^{k^2}$ for any k. We will prove this by proving that $n^2 < n^2 + m < (n+1)^2$. That is $n^2 < n^2+m < n^2+2n+1$, or $0 < m < 2n+1$, which is true because $0 < m \leq n$.

Example 3: Strings of composite length

Consider the langauge of strings $L = \{0^n | n \textit{ is a composite number}\}$. The language actually obeys the pumping lemma and therefore we will never be able to prove the contrapositive of the pumping lemma, even though the language is actually not regular. So this is an example of a language that is not regular and yet obeys the pumping lemma. You can however use the pumping property to prove that the language $L' = \{0^p | p \textit{ is prime}\}$, is not regular[26]. You can next note that the L and L' are complements of each other. That is the idea

[24] Note that $0^{\lceil \frac{k}{2} \rceil}10^{\lceil \frac{k}{2} \rceil}$ is a bad choice for trying to prove that the set of palindromes does not satisfy the pumping lemma. Because a choice of $v = 0^{\lceil \frac{k}{2} \rceil}$, $w = 1$ and $x = 0^{\lceil \frac{k}{2} \rceil}$, respects $|vw| \leq k, |w| \geq 1$ and implies that all strings $0^{\lceil \frac{k}{2} \rceil}1^i0^{\lceil \frac{k}{2} \rceil}$ are palindromes (that is they belong to L), $\forall i \geq 0$.

[25] The equivalent of identity in string concatenation operations is the empty string $\epsilon = 0^0$.

[26] Exercise

of closure. We know that the complement of a regular set is regular. Since the complement of L is not regular, we can infer that that set L we are dealing with is also not regular. So we have used proof by contradiction using the closure property. This illustrates another use of the closure property.

Example 4: Strings with equal number of 0's and 1's

Let us consider the language L consisting of strings that have an equal number of 0's and 1's. We note that $\{0^n1^n|0 \leq n\} = L \cap \{0^n1^m|0 \leq m, 0 \leq n\}$. Since regular sets are closed under intersection and since we know that $\{0^n1^m|0 \leq m, 0 \leq n\}$ is regular while $\{0^n1^n|0 \leq n\}$ is not regular (we proved that as our first application of pumping lemma), the set L cannot be regular. For, if the set L were regular, its intersection with another regular set $\{0^n1^m|0 \leq m, 0 \leq n\}$ should have been regular, which is not the case. Thus, we used a combination of pumping lemma and closure under intersection along with the technique of proof by contradiction to prove that L is not regular.

Example 5: Strings that represent legitimate FSMs

We could represent a finite state machine as a binary string as follows:

$$0^n10^{(state[1,0])}10^{(state[1,1])}1\ldots10^{(state[n,0])}0^{(state[n,1])}110^{(acceptState[1])}1\ldots10^{(acceptState[k])}$$

where, n is the number of states in the machine, k is the number of accept states, 1 is the default accept state, $state[i,g]$ returns the id of the state that the FSM goes to on a transition from state i on seeing the symbol g and $acceptState[u]$ returns the id of the uth accept state. For instance, the string representation for the FSM in Figure 2.23 is

$$001001010100110$$

So the above string represents a legal finite state machine. But does

$$0^310101$$

represent a legal finite state machine? No, because it is supposed to have three states, but does not specify what the transitions for the second and third states should be. Nor does it specify the accept states. There are lots of strings that are legal FSMs and many that are not. Let $L_{FSM} = \{M \mid M\ is\ a\ legitimate\ FSM\}$.

Can we figure out using a program whether a given string belongs to L_{FSM}? Can we have an FSM that does the job[27]? In other words, does there exist an FSM that recognizes one of its own kind (another FSM)? As a mundane example, a human being can recognize which of the entities in a room are actually humans[28].

[27] This is different from a previous problem which addressed if an FSM recognizes itself or not. Here we are concerned with the problem of determining whether a FSM can recognize another legal FSM.

[28] Or a sane can identify who in a class room are sane and who are not.

The answer is that the set L_{FSM} is not regular. Can we prove this formally? We will prove it again on the strenght of the contraposition of the pumping lemma in the form of a dialogue.

Question: Tell me the number of states in your machine that accepts L_{FSM}.

Adversary: The machine has n states (for any $n \geq 1$).

Question: Consider the string $0^n \underbrace{101010\ldots10}_{n\ pairs\ of\ 01} 11$. You should split the string into three parts vwx. No matter what you do, the first v will be stuck in the first n symbols, *i.e.*, in 0^n. Suppose $v = 0^p, 0 \leq p < n$, $w = 0^q, 1 \leq q \leq n-p$. Then pumping up w by a factor m, we get the string $vw^m x = 0^p 0^{(mq)} 0^{(n-p-q)} \underbrace{101010\ldots10}_{n\ pairs\ of\ 01} 11$ which is $0^{(n+(m-1)q)} \underbrace{101010\ldots10}_{n\ pairs\ of\ 01} 11$. This string is not a legal FSM because the string is prefixed by $(n+(m-1)q)$ 0's indicating $n+(m-1)q$ states $\forall m \geq 1$ and $\exists q \geq 1$, whereas, it has only n pairs of '10's following.

So this encoding of FSMs is too hard for FSMS to accept. So it is not surprising that it is hard to construct a FSMs that accepts itself. There is however the Turing machine that could do all this. The higher levels can not only understand and recognize FSMs but also simulate them, answer questions about them. The higher levels may not be smart enough to answer questions about their own kind though. Turing machines are powerful enough to recognize their own kind. That is what compilers are. What they are not power enough to do is to recognize any interesting properties about automata of their own kind. They cannot, for example say that '*Here is a Java program that never infinite loops*' or '*Here is another Java program that accepts exactly 5 inputs*' or '*Here is another Java program that does not accept anything*'. The compiler could simulate the program but the program might run forever and we might never know the answer. If the answer is 'yes', we obtain it after some time. But if the answer is supposed to be a 'no', we will wait forever.

Example 6: An alternative encoding of legitimate FSMs

There is an encoding of FSMs that makes them recognizable by themselves. Every FSM has a binary number associated with it. Let us order all FSMs according to size. The first one is the smallest one, the one with a single state. And we will re-label all FSMs such that the smallest one is called 0, the next will be 1, the next will be 10 and so one. Thus, every single FSM has a binary number associated with it. And every single binary number $(0+1)^*$ is taken care of has an associated FSM. In such an event, it is easy to develop an FSM that accepts them all; a single state that loops back on 0 as well as 1 and is also the accept state.

Thus, the discussion in example 5 entirely depends on the encoding scheme. It is normal to construct encodings so that every single encoding means something. As an example, suppose we map every string that does not correspond to an encoding of FSMs discussed on example 5 to an FSM that does not have any accept states (that is it accepts the empty set), then every binary string

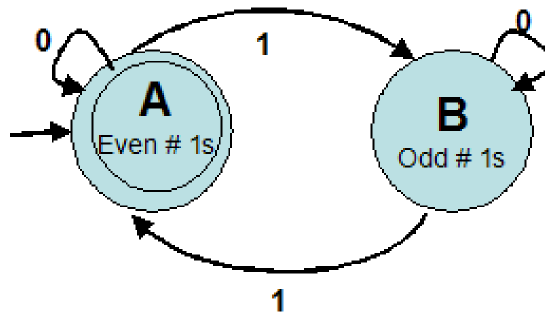

Figure 2.28: Finite state machine for $L_{(odd\ 1's)}$.

will again correspond to a legitimate FSM. That is every single binary string will have some semantics as an FSM.

2.11 Linear Grammar

This is a self-contained, short topic. It connects '*regular expressions*', deterministic FSMs and non-deterministic FSMs to linear grammars. It adds in a new way of looking at sets based on '*grammars*'. As the levels of sets go up, the 'grammar way' of looking at a set becomes at least as important, if not more important than the 'machine way' of looking at a set, which is very different point of view. Finite state machines are not generally expressed at the level of grammar, whereas, compilers and programming languages are mostly represented using grammars. The grammar point of view is very easy and transparent to make sense out of it. The best way to define a linear grammar will be start with an example.

2.11.1 From Finite State Machine to Linear Grammar

Consider the language $L_{(odd\ 1's)}$

$L_{(odd\ 1's)} = \{x|x\ is\ a\ binary\ string\ consisting\ of\ odd\ no.\ of\ 1s\}$

Figure 2.28 gives the FSM for $L_{(odd\ 1's)}$.

We will create a grammar, which is a formalism that neither accepts nor rejects strings; rather it *generates* strings. Any string that is accepted by the FSM can be generated by the grammar while any string that is rejected by the machine cannot be generated by the grammar.

As in the FSM, we will have a start symbol, A. And A can generate strings that the FSM can accept. Hence, A should be able to generate a 0 followed by a string that A can generate again. This production can be succintly represented as $A \rightarrow 0\ A$. Secondly, we should also be able to generate a 1 and then continue with B, that is, $A \rightarrow 1\ B$. Once in B, we should be able to generate a 0 and stay

in B or generate a 1 and go back to A, which are represented by $B \rightarrow 0\ B$ and $B \rightarrow 1\ A$ respectively. The candidate set of production rules are listed below.

$$\begin{array}{lcl} A & \rightarrow & 0\ A \\ A & \rightarrow & 1\ B \\ B & \rightarrow & 0\ B \\ B & \rightarrow & 1\ A \end{array} \tag{2.2}$$

Let us use these productions to generate some strings using a sequence of substitutions in the grammar. A sequence of substitutions in the grammar to create a string, is called a derivation. We will start and when we find that we get stuck up, we will complete this production table. The start symbol is A (though the usual notation for the start symbol is S). Starting from A, we could use the production $A \rightarrow 0\ A$. The A on the right hand side can be further expanded using the production $A \rightarrow 1\ B$. A legitimate next substitution for the B on the right hand side is, $B \rightarrow 0\ B$.

$$A \rightarrow 0\ A \rightarrow 0\ 1\ B \rightarrow 0\ 1\ 0\ B$$

What about termination? We are not generating any string as long we have some capital symbol left over on the right hand side. The capital symbols are called *non-terminals*, and they should not be present in a terminal string. The *terminal symbols* are symbols in the alphabet $\Sigma = \{0, 1\}$ and the final string should consist only of terminal symbols. Note that B is an accept state and therefore, it could disappear in the production sequence. Based on this observation, we add a production rule $B \rightarrow \epsilon$ (remember that ϵ is the empty string. The updated set of production rules is listed below.

$$\begin{array}{lcl} A & \rightarrow & 0\ A \\ A & \rightarrow & 1\ B \\ B & \rightarrow & 0\ B \\ B & \rightarrow & 1\ A \\ B & \rightarrow & \epsilon \end{array} \tag{2.3}$$

This is called a *linear grammar*. Here is a definition of linear grammars.

Linear Grammar *A linear grammar consists of:*

1. *a finite set of symbols Σ that form the strings of the language being defined. The elements of this alphabet are also called the terminals, or terminal symbols.*
2. *a finite set of variables, also called sometimes nonterminals or syntactic categories. Each variable represents a language; i.e., a set of strings. One of the variables represents the language being defined; it is called the start symbol. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol.*

3. *a finite set of productions or rules that represent the recursive definition of the language. Each, production consists of: (a) A variable that is being (partially) defined by the production. This variable is often called the head of the production. (b) The production symbol* $\rightarrow$*. (c) A string of zero or more terminals and variables. This string, called the body of the production, represents one way to form strings in the language of the variable of the head. In so doing, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of that, variable. The string on the right hand side is restricted to have one of two forms:*

   - *it has either a single terminal symbol, such as in* $A \rightarrow 1$ *or*
   - *it has a single terminal and a single non-terminal symbol, such as in* $A \rightarrow 0\ A$.

When the single terminal appears before the non-terminal, the grammar is called left linear. When the single terminal appears after the non-terminal, the grammar is called right linear. Left and right linear grammars are equivalent. Linear grammars generate regular languages. Linear grammars are a special case of context free grammars (to be discussed in chapter 3). Context free grammars allow any sequence of terminals and non-terminals on the right hand side of each production. .

Now we are equipped to continue the derivations above. Using this production, we could get a legitimate string using derivations.

$$A \rightarrow 0\ A \rightarrow 0\ 1\ B \rightarrow 0\ 1\ 0\ B \rightarrow 0\ 1\ 0$$

This is a derivation of the string 010, starting from the start symbol A, going through some intermediary steps. The intermediary steps are called *sentential forms*. In general, we should be prepared to stop further productions when we are on an accept state. Therefore, a production of the form $< Final\ State \rightarrow \epsilon$ will be required.

What we showed above was not example of a grammar. It was an example demonstrating how to take an FSM and convert it to a grammar. We could have done this with any finite state machine $< \Sigma, Q, S, F, \delta >$. The procedure is

1. Each state $q \in Q$ is interpreted as a non-terminal symbol S_q.
2. Each transition arrow $\delta(q_1, a) = q_2$ is interpreted as a production $S_{q_1} \rightarrow a\ S_{q_2}$.
3. Each final state $f \in F$ is interpreted as an empty production $S_f \rightarrow \epsilon$.

What if we had started with a non-deterministic machine? Could we have done this? And the answer is a 'yes'. We could have carried out the same set of steps. Because the 'or's in a non-deterministic machine end up being the

'or's in the interpretation of the grammar. Of course, we could always convert the non-deterministic machine to a deterministic machine and follow the above procedure. But we do not really need to do that.

The above example should convince us that if we had a deterministic machine, we could represent the language corresponding to the machine as a left linear grammar. In what follows, we will discuss how a finite state machine could be induced from a left linear grammar.

2.11.2 From Left Linear Grammar to Finite State Machine

The mapping between left linear grammar to finite state machines is one to one (as might be evident from the discussion in the previous subsection).

1. Each non-terminal symbol S_q is interpreted as a state q.
2. Each production $S_{q_1} \rightarrow a\ S_{q_2}$ is interpreted as a transition arrow $\delta(q_1, a) = q_2$.
3. Each empty production $S_f \rightarrow \epsilon$ is interpreted as a final state f

We will do one example of the reverse procedure and see if there are any pitfalls. Consider the grammar

$$\begin{array}{lcl} S & \rightarrow & 0\ A \\ A & \rightarrow & 1\ B \\ A & \rightarrow & 0\ B \\ A & \rightarrow & 0\ S \\ B & \rightarrow & 1\ S \\ A & \rightarrow & 1 \\ B & \rightarrow & 0 \\ S & \rightarrow & 1 \\ A & \rightarrow & \epsilon \end{array} \tag{2.4}$$

A must be a final state, since it gives an ϵ production. What should we do with single terminal productions such as $S \rightarrow 1$? One legitimate solution is to have a transition from A, S and B to a new final state F on symbols 1, 1 and 0 respectively. But this gives us a non-deterministic machine (Figure 2.29). Actually grammars, are by nature, non-deterministic. For instance, one does not know apriori whether to continue generating 1 B from A using the production $A \rightarrow 0\ B$ or stop using the production $A \rightarrow 1$ to be done! As long as there is a way to generate a string, we should be able to accept it using the corresponding non-determinstic machine. Actually, this is one more motivation for the idea of non-determinism. It fits in naturally with the 'or'ness of grammars.

The above discussion prompts us to define a new set of steps for transforming a linear grammar to a finite state machine.

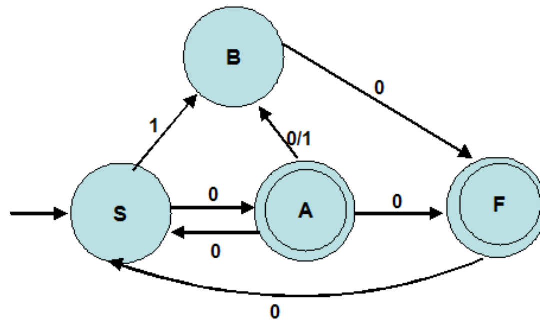

Figure 2.29: Finite state machine for the linear grammar in (2.4).

1. Each non-terminal symbol S_q is interpreted as a state q.
2. Each production $S_{q_1} \to a\ S_{q_2}$ is interpreted as a transition arrow $\delta(q_1, a) = q_2$.
3. Each empty production $S_q \to \epsilon$ is interpreted as a final state q
4. Each singleton production $S_q \to a$ is interpreted as a transition to a final state f from state q on symbol a.

2.12 Traingle of Equivalance for Finite State Machines

We had proved the equivalence between deterministic and non-deterministic finite state machines. In the previous section, we discussed the equivalence between non-deterministic finite state machines and linear grammars. By transitivity of the equivalence relation, we get equivalence of deterministic finite state machines with linear grammars. This gives us the triangle of equivalence as in Figure 2.30.

2.13 Mimization of Finite State Machines

The fundamentally neat thing about finite state machines is that once you create one, there exists a unique finite state machine that is equivalent to the machine while being equivalent to the original. Therefore, if may people develop their own finite state machines for a given problem, all could minimize it to obtain a unique machine. This does not hold for turning machine programs or for pushdown automata that fall somewhere in the middle of the bull's eye. They do not have the notion of a minimum machine. The presence of a minimal machine let one understand finite state machines really one. Minimization is at the core of the so called decision algorithms about finite state machines. To

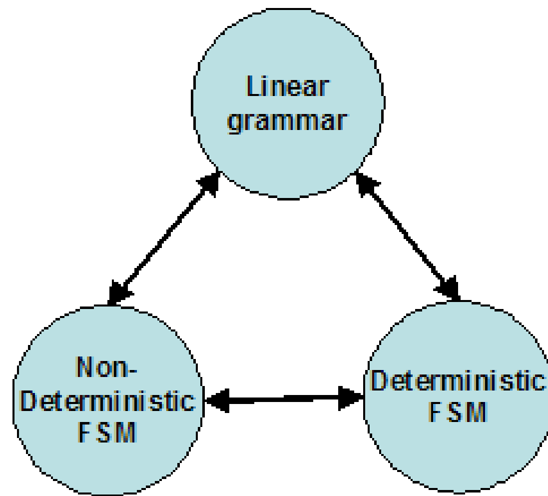

Figure 2.30: The Traingle of Equivalence.

explain this idea, we will go of on a little tangent and hope to get the intuition behind minimization of FSMs. Subsequently, we will get into greater details of finite state machines. We will conclude with a couple of example minimizations.

2.13.1 Intuition for Minimization

Suppose you are stuck in a cave. The cave has rooms that look like the states of an FSM. Each room has little doors out of them, leading to other rooms. Each door is labeled with a 0 or a 1. You do not have any map of this cave; somebody just transports you into one of the rooms. You want to map the cave with a little piece of paper and a pencil in your hands. You want to make a map because it turns out there is water/food in some of the rooms in the cave. You want to be able to figure out which rooms have food/water and be able to back and forth from one to the other without getting lost. This has semblance with adventurous dragon games. In these games, you often find yourself in a maze like this. You basically map up the maze so that later on when you are running through the program, you can figure out where you are, which is very difficult to tell from the description of the program. Suppose you walk from one room to another in the cave. How do you know that a room you walk into a new room or one of the visited rooms? You do not really know of sure. You have no idea where these paths go, no sense of direction. When you are done traversing the map for a long time and notice the pattern of rooms you visited, you suddently realize that the room you visited just now is functionally very similar to another room you visited some time back. And you should perhaps collapse the two rooms together. The map was possibly mistaken and you repeated the same room couple of times. Perhaps you visited the same room with food/water at two

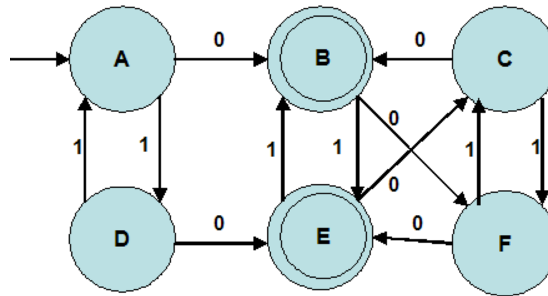

Figure 2.31: A non-minimal finite state machine.

different instances of time and plotted them as different locations on the map and therefore these states should be merged. When you tend to do this instead of creating a huge map, you create a smaller map, that is just as good but much simpler. The complicated map (traversed with no information of direction) is the like a large finite state machine and the real map where every room has just one circle for representation is the minimal finite state machine.

With this intuition in mind, we will do an example. Consider the finite state machine in Figure 2.31. Let us say the FSM in the figure is what we obtain as a result of wandering around the cave. The accept states (states with circles), represent the rooms that store food/water. Can we discover rooms/states in the machine that are functionally identical? What does *functionally identical* mean in the first place? We will point out some states that could not possibly be functionally identical while some others that seem to have common functionality.

1. Two states that can be functionally identical or those that can be confused are the two accept states B and E.

2. By definition, final and non-final states are different. In terms of food/water, one has food/water in it and other does not. Hence they cannot be merged together without disturbing the functionality of the machine. Thus, A is distinguishable from E the empty string ϵ; on an empty string, the FSM is at A and does not accept the string, whereas, E is an accept state, which you cannot reach on an empty string.

3. Consider the states D and C. What if for every sequence of symbols, that lead to D or C, the next symbol 0 leads to an accept state? and whenever we did not accept at D, we did not accept at C. And this is what happens. Then what is the point of having two different states? We might as well merge them into one state.

We can expand a little here. Is it true that D and C are equivalent[29]? Starting at C as well as D, 0 is accepted while the string 00 is not accepted.

[29]If they are, we need to rigorously decide how to go about merging them; simply eyeballing is not sufficient.

	F	E	D	C	B
A		X			X
B	X		X	X	
C		X			
D		X			
E					

Figure 2.32: The table for the minimization procedure for FSM in Figure 2.31 filled up through base cases.

Can we come up with any string that distinguises C from D, from the point of view of accepting/not accepting? It turns out if you can distinguish two states in this FSM using a string, the string must have six symbols or less (where six is the number of states in the FSM). Because after six symbols, you will revisit some states. Therefore, there is no need to go for ever looking for distinguishing strings. This gives a mechanical, brute-force procedure for distinguishing states from each other. We will present a more efficient procedure for this task.

2.13.2 An Efficient Algorithm for Minimization

Our base case will be that accept states can be distinguished from non-accept states. We will create a different picture that represents the depedency as well as the distinguishing states in Figure 2.31. The picture is supposed to represent all combinations of every state with every other state. In all, there are $\binom{6}{2} = 15$ combinations of pairs of states, avoiding repetitions of pairs. We will detect whether to distinguish one from another. For every pair of states that are distinguishable from one another, we will mark a cross '**X**'. We will mark all non-final (non-accept) states A, C, D and F as distinguishable from the final (accept) states B and E. What we would like to do is fill in chart. If we are able to fill in this chart completely, and discern distinguishable states, we could collapse the states that are indistinguishable into single states. So if turns out that there is no cross '**X**' for the pair (B, E), the states B and E could be collapsed. Figure 2.32 shows the table filled up for the base cases.

Let us try to determine whether a cross should go into any of the other blank squares of the table in Figure 2.32. How would you decide that A and F are distinguishable from one another? THe base case does not work for this pair,

since they both are non-final states. Let us try to refine our information by adding in a single symbol that starts in A and F simultaneously and see where it ends up. For the symbol 0, we end up in B and E respectively. Are B and E distinguishable? Not yet. If B and E were distinguishable, what would we have known about A and F? The answer is that they are also distinguishable. That is because we had just tacked on a '0' to the string that landed in A and F to arrive at B and E respectively. When we add the symbol 1 at the states A and F, we land at D and C respectively which are again not distinguishable so far. Later on, however, if we discover that D and C are distinguishable, it will imply that A and F are distinguishable. Similarly, distinguishability of (A, D) depends on distinguishability of (B, E) through symbol 0. Also, the distinguishability of (A, D) depends on distinguishability of (A, D) (itself) through symbol 1. But you do not bother writing this dependency. In fact, such a dependency only means that with respect to the symbol 1, A and D are indistinguishable. (A, C) depends on (B, B) through symbol 0; whenever you have such cases, where the distinguishability of two states depends on the distinguishability of the same state, such as here, the two states are definitely not distinguishable. Therefore, we leave out this redundant dependency. In order to do all this, here is how the algorithm actually functions.

1. Go through all the empty squares one at a time to determine and register all dependencies in corresponding boxes. In order to avoid registering cyclic depedencies, we will start traversing from the top left box, toward the right and then down. An entry will be made in the box corresponding to a pair only if the dependency refers to a pair that occurs before this pair during the traversal. Thus, the entries for (D, F) and (C, F) will not be made in the box for the pair (B, E). For instance,

   - go into the box for (B, E) pair and put an (A, F).
   - go into the box for (C, D) and put in an (A, F) there too.
   - place the entry (A, D) in the box for (B, E).
   - place the entry (A, C) in the box for (D, F).
   - place the entry (B, E) in the box for (C, F).

   This will give us information about states that might be distinguishable based on strings consisting of single symbol each.

2. The semantics of registering such dependencies is that later on, if we happen to put an '**X**' in either of (B, E) or (C, D), we will backtrack and put an '**X**' in (A, F) box as well. And at that time, we will wind up further from (A, F), if there are any pairs registered in its box. So as the second step, go through every box and backtrack if required.

3. After executing step 2, we will go back to step 1. Registering dependencies now using step 1 will gives us information about states that might be distinguishable based on strings of length 2. This can go on. The amount

	F	E	D	C	B
A		X			X
B	X	(A,F) [0] (A,D) [0]	X	X	
C	(B,E) [0]	X	(A,F) [1]		
D	(A,C) [1]	X			
E					

Figure 2.33: The table for the minimization procedure for FSM in Figure 2.31 filled up after one step of the algorithm for minimization.

of the backtracking (limited by the number of states in the machine) will determine the size of the string that distinguishes between pairs of states.

The minimization problem is really a graph searching problem in disguise. The above algorithm is based on the idea of *equivalence relations*. Deciding whether one set is distinguishable from another is an equivalence relation. We do not want states that are equivalent to be represented separately in the finite state machine. Since equivalence relations are transitive, we could iteratively continue the steps (1) and (2) above. Figure 2.33 shows the table updated after the first step of the above algorithm.

We observe that none of the boxes that had dependencies registered have been marked as distinguishable. Which means we cannot go any further in finding distinguishable states. Equivalence relations are transitive and they induce a partioning. In this particular case, the equivalence relation partitions the set of states into two: the indistinguishable set of non-final states and the indistinguishable set of final states. Thus, the FSM in Figure 2.31 is actually equivalent to the minimal FSM in Figure 2.34. And it is surprising to note that all the complicated FSM in Figure 2.31 was doing was accepting only strings that have an odd number of 0's! It is evident from this example that it can be a lot better working with minimum FSMs.

When designing an architecture for a computer, you have a big FSM that represents the micro-code and how it is all going to distributed. And you want to define the micro-code without worrying about minimizing it. But when you want to implement the micro-code and turn it into hard ware, while being small and compact, you would minimize it. The minimization algorithm is very commonly used. You could certainly eye-ball redundancies in a large FSM, but the minimization algorithm we discussed could be mechanically performed by any machine.

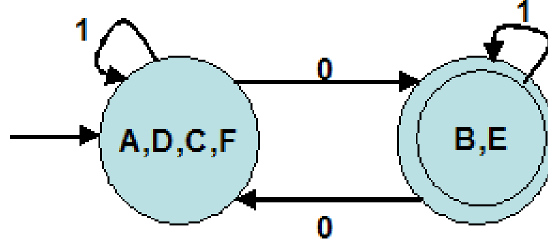

Figure 2.34: The minimal finite state machine that is equivalent to the FSM in Figure 2.31.

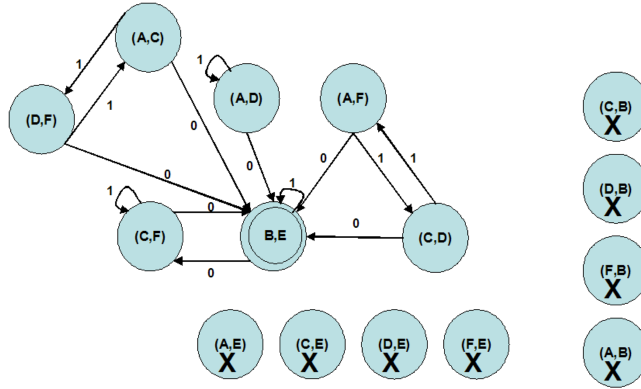

Figure 2.35: The dependency graph corresponding to dependency entries in Table 2.33.

We before we go on to the next example, we will briefly talk about the dependency graph and how it relates to which states are distinguishible. Figure 2.35 shows the dependency graph corresponding to the dependency entries in Figure 2.33.

Second degree dependencies are obtained in the dependency graph by traversing two edges. Thus, (C, D) is dependent on (C, F) by a string of length 2. All you have to do to determine distinguishibility is to take the '**X**'s marked on certain nodes (base cases) and flood them through the graph in the reverse direction of the arrows and see where the '**X**'s can reach. Any node the '**X**'s can reach will be marked with an '**X**' and will be considered to be distinguishible. This can be done using any search algorithm that goes through a graph, such as *depth first search* or *breadth first search*. The algorithm takes time proportional to the number of edges in the graph. Since there are two edges out of every single node, and since the number of nodes in the dependency graph is $\binom{n}{2}$, the time taken for this algorithm is $2\binom{6}{2} = (n-1)n$. Thus the time taken by this

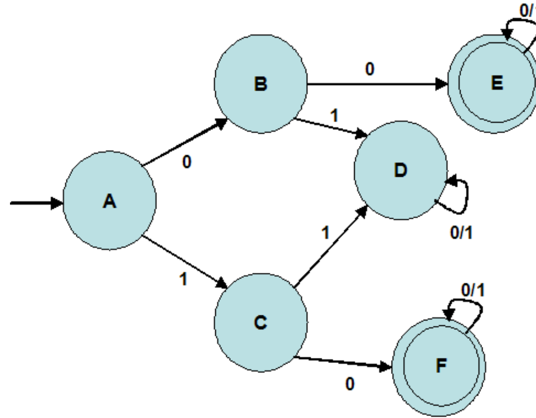

Figure 2.36: Another non-minimal finite state machine.

algorithm should be no more than n^2.

Example

Consider the finite state machine in Figure 2.36 and the corresponding Table in Figure 2.37 marked with base case crosses '**X**'. The base cases correspond to eight isolated nodes.

Next, we traverse the boxes, starting with (A, D). We get dependence (for distinguishability) of (A, D) on (B, D) and (C, D) for the input symbols 0 and 1 respectively. But neither (B, D) nor (C, D) are marked with a '**X**' and hence we can only note down the dependence for possible resolution in the future. However, for the next pair of (A, C), we note its dependence on a distinguishible pair (B, F) for the input symbol 0. We cannot have the A and C be functionally identical and they must be functionally distinguishible. (A, C) will therefore get marked with a '**X**'. Once we get distinguishability on 0, we do not both checking distinguishability on 1 (which was otherwise a dependence on (C, D)). Going further to (A, B), we note its dependence on a 0, on (B, E), which again happens to be distinguishable. Thus (A, B) again gets an '**X**'. (B, D) depends on (D, E) for symbol 0, and note that (D, E) is crossed and is distinguishable. Moving forward to (B, C), it depends on (E, F) for symbol 0 and on (D, D) for symbol 1, both of which are indistinguishable. Finally for (C, D), its dependency on the distinguishable pair (D, F) should fetch it a distinguishability certificate.

Figure 2.38 shows the table after application of the first step of the algorithm. Figure 2.39 shows the same table after application of the second step, which involves propagation of '**X**'s in the reverse direction of arrows. Finally, Figure 2.40 shows the table after application of the first step in the second iteration to mark the pair (A, D) as distinguishable. This leaves two pairs as indistinguishable, *viz.*, (B, C) and (E, F). These two could be collapsed into one state each as in Figure 2.41.

	F	E	D	C	B
A	X	X			
B	X	X			
C	X	X			
D	X	X			
E					

Figure 2.37: The table for the minimization procedure for FSM in Figure 2.36 filled up through base cases, that is final states are distinguishable from non-final states.

	F	E	D	C	B
A	X	X			
B	(A,C) [0] X	(A,B) [0] X	(A,D) [0]		
C	X	X	(A,D) [1]		
D	(C,D) [0] X	(B,D) [0] X			
E	(B,C) [0]				

Figure 2.38: The table in Figure 2.37 after first step of the minimization algorithm.

	F	E	D	C	B
A	X	X		X	X
B	(A,C) [0] X	(A,B) [0] X	(A,D) [0] X		
C	X	X	(A,D) [1] X		
D	(C,D) [0] X	(B,D) [0] X			
E	(B,C) [0]				

Figure 2.39: The table in Figure 2.38 after the second step of the minimization algorithm.

	F	E	D	C	B
A	X	X	X	X	X
B	(A,C) [0] X	(A,B) [0] X	(A,D) [0] X		
C	X	X	(A,D) [1] X		
D	(C,D) [0] X	(B,D) [0] X			
E	(B,C) [0]				

Figure 2.40: The table in Figure 2.39 after the first step in the second iteration of the minimization algorithm.

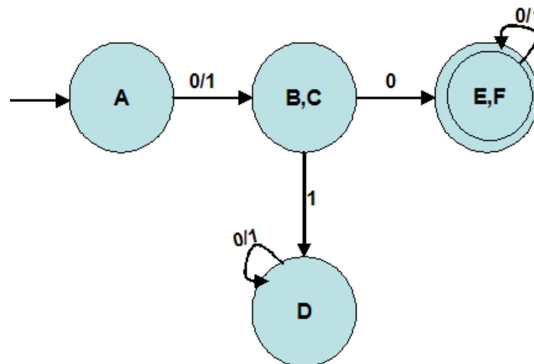

Figure 2.41: The minimal finite state machine that is equivalent to the FSM in Figure 2.36.

What does the minimal in Figure 2.41 do? It is lot easier to analyze now. It accepts every string that does not have a 1 in the second position!

2.13.3 Comutational Complexity

Let us analyze how long this algorithm will take for computing a minimal finite state machine. The dependency graph has $2n^2$ edges and the time taken to travese it is $2n^2$. But that is not exactly what we do in the algorithm. The table in Figure 2.39 has $\binom{n}{2} = \frac{(n-1)n}{2}$ cells We look at each box exactly twice (once corresponding to each of the two symbols) during the forward traversal and this implies a time of $(n-1)n$, which is of the order of n^2. But there is the backtracking and that adds some extra time. The worst thing that could happen is that the algorithm has to backup at every single step. And when it backtracks, it might go through n levels, backtracking up at every single stage, making the time for computation, $n * n^2 = n^3$ in the worst case. But such a case can never happen. The above analysis was quite careless.

How can we never require to backtrack at every single stage for every single value? Everytime, we backtrack, Let us ask ourselves, how many times will some cell get visited to be marked with an '**X**' during backtracking? The answer is that every single cell might get visited twice during backtracking; once in the backward direction of an arrow marked with '1' and once in the backward direction of an arrow marked with '0'. So on the whole, every cell can be visited twice (once during the forward traversal and twice during backtracking). Thus, the overall time for computation will be $3\frac{n^2}{2}$, which is again in the order of n^2. Thus, the algorithm is an n^2 algorithm.

The algorithm is often taught in a simpler to understand manner, which turns out to be n^3 in the worst case. It works as follows: It starts off drawing the graph completey (the $\binom{n}{2}$ states). It marks the final and non-final states.

Next, it propagates the '**X**'s. In doing so, it looks at each of the n^2 vertices, n different times, making it an n^3 algorithm. This is really an inefficient algorithm. There is also an algorithm by Hopcroft and Ullman that runs in $nlog(n)$.

2.14 Decision Problems about FSMs

What questions can we answer about finite state machines, *i.e.*, regular sets? The suite of algorithms that answer yes or no are often called decision algorithms. The inputs are binary strings that represent finite state machines. You might be given an FSM and asked, '*Does this machine accept any string of length 10?*'. Of you could be presented with two FSMs and asked, '*Do they accept the same language?*' '*Or do the two FSMs have anything in common they accept?*' These are questions about FSMs; questions you would write an algorithm or program for. And the input to the program is a finite state machine. The most obvious question you might want to ask about a FSM (one that we already addressed) is, *Here is an FSM. Here is an input string to the FSM. Does the FSM accept this input string?'* Even a beginner in computing class could write a program for this, by setting up an array. There is a built-in utility in Linux called *Lex* that does this. You can specify a FSM, provide it a string and *Lex* can tell you whether or not the FSM accepts the string. In fact, *Lex* can do more that simply answer the decision question.

The reply is that we can answer almost all questions about FSMs. We will do a couple of them.

2.14.1 Membership Question

Given an FSM F for a regular set L, and an input string x, is x generated by F? In other words, is $x \in L$? How do you write an algorithm to answer this question? We will begin with specification of the data structure required. The data structure could be either a graph or a multi-valued array, where the states, paired up with symbols from the alphabet are the indices and the values inside are other states. You seek a row, and traverse the row to set your next state and this way, you could use the data structure to simulate the FSM F on the input string. You could also change the decision problem into a matrix algebra problem which is also computable.

2.14.2 Equivalence of Two Finite State Machines

Given two finite state machines F_1 and F_2 for some unknown languages L_1 and L_2 respectively, can we determine if they accept the same language? That is do they accept exactly the same set of strings and reject the same set? The solution is simple. You could minimize F_1 to get M_1 and minimize F_2 to get a minimal machine M_2. If M_1 and M_2 are the same, we can conclude that F_1 and F_2 accept the same language. To decide if M_1 and M_2 they are the same, we should not just naively compare the two, state by state, since the states

may not be numbered/named in the same order. The problem of determining if two graphs are the same is called the *graph isomorphism* problem. The brute force solution is to relabel the graphs with every possible relabeling and see if any of the relabelings are the same. For a graph with n states, the number of possible relabelings is $n!$. The brute force solution takes exponential time and is definitely not efficient, but it is a solution nevertheless. As far as the decision problem is concerned, the answer is a 'yes', that is, it is possible to determine if two FSMs are equivalent.

Is there any more efficient way to check for graph isomorphism? In this particular case, we could relabel each state using the sequence of symbols on the shortest path from the start state to the state under consideration. And this is an efficient algorithm.

There is another efficient idea to check the equivalence of two FSMs. If the language of two FSMs are equal, their symmetric difference $(L_1 - L_2) \cup (L_2 - L_1) = (L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1})$ should equal the null set ϕ. The finite state machine $F_{(1=2)}$ for the the language $(L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1}$ can be constructed by following the sequence of steps below:

1. Find the complement of the machine F_2 using the procedure in Section 2.7.3. This is linear in the number of states.

2. Find the intersection $F_{(1-2)}$ of the machines F_1 and the complement of F_2 using the procedure discussed in Section 2.7.4. The computation time for this algorithm equals the product of the number of states in the two machines.

3. Similarly, construct the FSM $F_{(2-1)}$ for the language $(L_2 \cap \overline{L_1}$. Construct the FSM $F_{(1=2)}$ as the union of $F_{(1-2)}$ and $F_{(2-1)}$.

4. Minimize $F_{(1=2)}$. If the minimal machine has a single non-accept state, the two machines F_1 and F_2 can be confirmed to be equivalent and the answer is a 'yes'. If the minimal machine has an accept state, the two machines F_1 and F_2 are not equivalent and the answer is a 'no'.

This entire idea of using set theory to construct a new FSM and then minimize it is a very commonly used tool in decision algorithms. And you can know pretty much anything you want about two FSMs using such tools.

2.14.3 Is the language of an FSM Infinite?

Let us say, we are given an FSM F for some regular set L and we want to know if L is infinite? Is it sufficient to see if there is a loop on the machine? Actually no. There shoud be a loop or more generally, a cycle, involving a node that has a path to a final state. This takes us back to the territory of graph algorithms.

There are also other ways of doing this. The FSM could be converted into a regular expression and then we could check for the kleen operators '*' and '+' in the regular expression. If the regular expression involves any one of these

kleen operators, we could conclude that the regular set is infinite. This is a safe procedure.

We should note however, that the problem of determining whether two regular expressions are equivalent is NP-complete in the size of the regular expression. There is no minimum regular expression. The only way to determine equivalence of regular expressions is to convert them into NFAs, convert the NFAs back to DFAs and then follow the procedure in Section 2.14.2. Regular expressions are hard to deal with. Almost all problems involving regular expressions are NP-complete with respect to the size of the regular expression.

2.14.4 Is Regular Set A contained in regular set B?

Say we have an FSM F_A for some regular set A and an FSM F_B for some regular set B. And we have to answer the question: '*Is every string that A accepts also accepted by B?*' Recall from discrete mathematics that $A \subseteq B$ is the same as $x \in A \Rightarrow x \in B$, which is equivalent to $(x \notin A) \vee (x \in B)$ or $\overline{A} \cup B = \Sigma^*$. Thus, $\overline{A} \cup B = \Sigma^*$ represents the logical statement $A \subseteq B$. That is if we build an FSM for $\overline{A} \cup B$, minimize it and find that it is a single state machine, with the single state being an accept state, then we know that $A \subseteq B$.

2.15 Moving Forward

Decision algorithms care about whether a problem can be done or not; is there an algorithm that answers 'yes' or 'no'? We do not usually care so much about the complexity of implementing these algorithms. The reason that we focus more on whether the problem can be done or not is because the moment we are up to the next level of push-down machines and context free grammars, almost everything you want to know about them, you cannot know! If you are given two context free grammars or two compilers and are asked, '*do they generate the same language*', or '*do they accept any language in common*' it is simply undecidable. There is no way to write a program to check that. The only thing we can do at the next levels is the membership problem, which happens to be the compiling problem itself. Given a grammar and a program, to determine if the grammar generates the program is the task of a compiler. That is the only problem you can do at the next level! When you get up to Turing machines, there is essentially nothing that is decidable. There is actually a theorem called the *Rice's Theorem* which states that: *Every non-trivial property of Turing machines is undecidable.* A trivial property of a turning machine is *how many states it has.* That is decidable for instance. On the other hand, there are no undecidable problems that are interesting about finite state machines.

Chapter 3

Context Free Languages

Context free languages form the second layer of the 'Bulls Eye'. Instead of starting with machines, as we did for FSMs, we will describe them in terms of grammar (recall the left and right linear grammar description for FSMs). We will subsequently describe the equivalent machine description for this class of langauges.

We will first draw the big picture of context free languages (CFL). We will refine the CFL layer into two layers;

1. The *deterministic push-down machines* (DPDM) on the inside layer. DPDMs have less power than CFLs and represent a subset of context free languages. The grammar of DPDMs is very difficult to describe easily, but is very important. The grammar of DPDMs is called *LRK* grammar.

2. The *non-deterministic push-down machines* (NPDM) on the outside layer. Context free languages are equivalent to NDPMs. Unlike the case of the FSM where non-determinism gives the same machine in disguise, non-determinism in PDMs adds more power. Note that all deterministic PDMs are contained in the non-deterministic counterpart.

Most compilers are built from LRK grammars; any programming language is almost always described using an LRK grammar. The implication is that if you have an LRK grammar, then the compiler is easy to build around it. If you have just a general CFL, the compiler could be very difficult to build around it. You need determinism to build a compiler.

We will first focus on context free grammar and their terminology. A context free grammar is any grammar, such that the left side of every production has a single non-terminal symbol. The sequence of symbols on the right hand side is completely unrestricted. Grammars with more than one symbol on the left side are much more powerful. On the other hand, linear grammars that were covered earlier are much more restricted than context free grammars. Linear grammars have a single non-terminal on the left, but are restricted on the right to have a

terminal followed by a non-terminal (in the case of left-linear grammars) or a non-terminal followed by a terminal (in the case of right-linear grammars).

3.1 Context Free Grammars and Derivations

It is hard to predict with a grammar, what all it is capable of doing. There are a lot of techniques for designing grammars, for understanding grammars and for making the connection between grammars and parsing, the second important stage of a compiler.

3.1.1 Example 1

Here is our first context free grammar.

$$S \quad \rightarrow \quad 0\ S\ 1 \quad | \quad \epsilon \tag{3.1}$$

The vertical line saves us space; instead of having two different productions with the same left hand side, we have a single production with a single left hand side and a vertical line separating two productions on the right hand side. What kind of strings does the grammar generate? The first production can be

$$S \rightarrow 0\ S\ 1$$

Applying this production recursively, we get

$$S \rightarrow 0\ S\ 1\ \ \rightarrow 0\ 0\ S\ 1\ 1$$

We could stop this at any point and let S go to the empty string and get a full derivation of a string. If we stop the derivation above, we get

$$S \rightarrow 0\ S\ 1\ \ \rightarrow 0\ 0\ S\ 1\ 1\ \ \rightarrow 0\ 0\ 1\ 1$$

This grammar actually generates a string of 0's followed by an equal number of 1's. In other words, the language is

$$L = \{0^n 1^n | n \geq 0\}$$

That was the first language we found to be non-regular! Thus, right off the back, if we had a context free grammar, unrestricted on the RHS (which every CFG can be), we immediatly get a set that is definitely not regular. This illustrates that we have expanded our universe beyond regular sets. Next, we will formally define context free grammars:

Definition of Context Free Grammar: *A context-free grammar G is a quadruple (V, Σ, R, S), where:*

- *V is an alphabet,*
- *Σ (the set of* **terminals***) is a subset of V,*
- *R (the set of* **rules***) is a finite subset of $(V - \Sigma) \times V^*$, and*
- *S (the* **start symbol***) is an element of $V - \Sigma$.*

The members of $V - \Sigma$ are called **nonterminals**. *For any $A \in V - \Sigma$, and $u \in V^*$, we write $A \rightarrow_G u$ whenever $(A, u) \in R$. For any strings $u, v \in V^*$, we write $u \Rightarrow_G v$ if and only if there are strings $x, y \in V^*$ and $A \in V - \Sigma$ such that $u = xAy$, $v = xv'y$, and $A \rightarrow_G v'$. The relation $\Rightarrow_G^*$, is the reflexive, transitive closure of $\Rightarrow_G$. Finally, $L(G)$, the* **language generated** *by G, is $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$; we also say that G* **generates** *each string in $L(G)$. A language L is said to be a* **context-free language** *if $L = L(G)$ for some context-free grammar G.*

3.1.2 Example 1

Here is another grammar[1] that should look familiar again.

$$S \quad \rightarrow \quad 0\,S\,1 \quad | \quad S\,S \quad | \quad \epsilon \tag{3.2}$$

What kind of of strings does this grammar generate? It represents strings that have balanced parentheses, 0 being the opening parenthesis and 1 representing the closing parenthesis. The rule says:

- If you want something to be generated by S, it should either be just the empty string, (which has balanced parenthesis),
- Or if you have already something generated by S, you could put 0 on the left and 1 on the right, which corresponds to putting open and close parenthesis around something that is already balanced,
- Or concatenate two sets of balanced parenthesis.

According to the recursive thinking, this grammar should generate all string with balanced parentheses. We will look at this in some more details.

Pretend that we are writing a compiler. Consider the language described in (3.2) that accepts certain strings and rejects certain strings. Given the string $\sigma = 00100110110011$, can we determine if the grammar in (3.2) can generate this string? And in fact the answer is a '*yes*'. How do we figure this out? One strategy could be to traverse the string from left to right and find the first match with the right hand side of one the productions. This can be done by writing a program that stacks[2] up the symbols one by one and backtracks when it finds a match to the right hand side of some production. How does this program

[1] The first couple of grammars will have only one start symbol.

[2] Recall that a stack is last-in first-out.

connect to the derivation sequence used to derive any string using the grammar (for instance the derivation of σ, if at all it can be derived using the grammar). Can we start with S and end up with the string σ?

For the first production, we have three choices. The third production (production to an empty string) is a bad choice to start with. What if we started with the first rule $S \rightarrow 0\ S\ 1 \rightarrow 0\ 0\ S\ 1\ 1$? We will never be able to generate the string 1001101100 starting from S.

Coming to the second production, are there substrings of σ that by themselves correspond to a balanced set of parentheses? And the answer is yes!

$$\underbrace{0010011011}_{\sigma_1}\underbrace{0011}_{\sigma_2}$$

Both σ_1 and σ_2 can be derived starting with the production $S \rightarrow 0\ S\ 1$ as follows. If we always choose to first substitute the right-most non-terminal on the left hand side at every stage, the derivation is called a *right-most derivation*. Very often, to keep the derivation organized, we fix expansion on either the left-most non-terminal or the right-most non-terminal. This makes comparison of derivations easier; two derivations are the same only if their left-most derivations are the same and two derivations are different only if their left-most derivations are different. The same holds true for right-most derivations. However, mixing left and right derivations do not yield different results; intervening derivations only makes comparisons difficult.

Another canonical way of representing the derivation is as a *parse tree*, in which case, the order of derivations (left or right-most) becomes immaterial. Figure 3.1 shows a parse tree for the string σ using the grammar in (3.2). The parse tree derivation is equivalent to the left-most and right-most derivations.

The strings σ_1 and σ_2 can be derived starting with the same rule $S \rightarrow 0\ S\ 1$. At the next level, for the left subtree, $S \rightarrow S\ S$ is a derivation that is applicable, following which, there are two options for deriving from each S as shown in the derivation trees 3.1 and 3.2. When we are done with a derivation or parse tree, non-terminals end up as internal nodes and all the terminals in the string end up as leaves. How do we print the leaves of the tree in an order that corresponds to the string that was parsed, *i.e.*, σ? You recursively print the left subtree at every node, then the right subtree. A simple *in-order traversal* of the tree will reveal that the string generated by the derivation is the string that was parsed.

Having two parse trees for the same string is a bad feature to have in the derivation; to not know which rule should be used next in the derivation in order to eventually obtain the string under consideration. Also, given a legitimate string in the language, it is not a good feature for compilers if there exists more than one derivation for the string. We say a grammar is *ambiguous*, if any string in the language has two or more parse trees. A grammar is *unambiguous* if and only if each and every string in the language has a unique parse tree. In fact, the problem of determining if a given grammar is ambiguous or not is *undecidable*; it is impossible to obtain an answer to this problem for any given arbitrary grammar. It is a really nice problem to have an answer to. Given an arbitrary

Figure 3.1: A *parse tree* for the string $\sigma = 00100110110011$ using the grammar in (3.2)

Figure 3.2: An alternative *parse tree* for the string 00100110110011 using the grammar in (3.2)

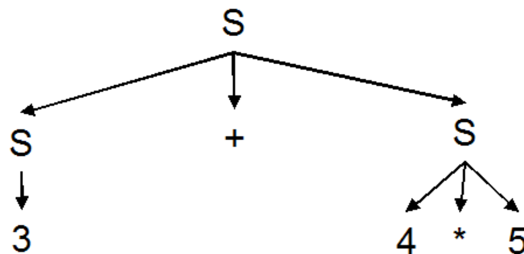

Figure 3.3: A *parse tree* for the string $3 + 4 * 5$ using the grammar in (3.3)

number, we could read it in and try all derivation trees using depth first search on every possible string. Sooner or later, we should be able to deduce if some string can be derived by two different ways. The program will give the right answer, if the answer is that the grammar is ambiguous; it will eventually find a string with two trees that are different. But if the grammar happens to be unambiguous, the program doing simple simulations will run for ever. It cannot ever tell us a 'no' if it happens to be unambiguous. Thus, the simuulation is actually not an algorithm at all.

The tree derivation provides a semantic interpretation for the string as provided by the grammar. In this example, the two interpretations of the string provided by the derivations were not different. In the next example, we will see how different derivations could provide drastically different interpretations for the string based on the grammar.

3.1.3 Example 2

Consider the famous grammar in 3.3. We will continue sticking to a single non-terminal.

$$S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \ldots \mid 9 \tag{3.3}$$

The $+, *, 0, 1, 2 \ldots, 9$ are terminal symbols. This grammar generates strings over the twelve symbol alphabet $\Sigma = \{+, *, 0, 1, 2, \ldots, 9\}$. This grammar generates strings that can appear on the right side of assignment statements in any programming language. Normally, instead of the terminals being $0, 1, 2, \ldots, 9$, we have something called an identifier, which could be either a variable or a number. The variable itself may have been defined elsewhere in the code. Can we determine a parse tree for the string $3+4*5$ using this grammar? Figure 3.3 shows one parse tree which is obtained by starting with the rule $S \rightarrow S + S$.

Can we come up with another parse tree? Instead of starting with $S \rightarrow S + S$, if we start with $S \rightarrow S * S$, we get the alternate parse tree in Figure 3.4.

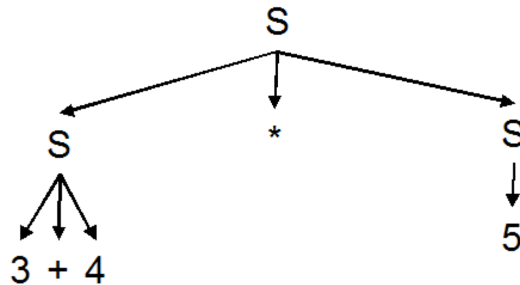

Figure 3.4: An alternative *parse tree* for the string $3 + 4 * 5$ using the grammar in (3.3)

The two parse trees are completely different; you cannot find an isomorphism between them. Thus, the grammar is ambiguous and in fact for almost any string, it will generate more than one parse tree. Why does this matter? It is because the compiler has to use these trees not just to say 'yes' or 'no', but often in the next stage, to ascribe some semantic interpretation or meaning to the string it just read. When the compiler reads the string according to the parse tree in Figure 3.3, it assumes that the expression is $(3+4)*5 = 7*5 = 35$. You could write a nice simple recursive program that evaluates parse trees by recursively going down to a node, doing an operation at the node on its two children and finally propogating the values back up.

What will the interpreation of the string on the lines of the parse tree in Figure 3.4 yield? It will yield $3 + (4 * 5) = 3 + 20 = 23$! Which of the two is correct? What do we normally mean by $3+4*5$? We normally mean $3+(4*5)$, which conforms to the parse tree in Figure 3.4. This is because of the precedence of $*$ over $+$. If the parsing is done as per Figure 3.3, no precedence is assumed for $*$ over $+$. And because of that, the semantic interpretation of the parsing gets lost. This is undesirable. This example emphasises the flip side of ambiguity.

3.1.4 Example 3

How do we fix the ambiguity in example 3? There are many ways to fix it. One way is to add two additional symbols to the alphabet, *viz.*, (and). We will also redefine the grammar with a slight modification in (3.4).

$$S \;\rightarrow\; (\,S + S\,) \;\mid\; (\,S * S\,) \;\mid\; 0 \;\mid\; 1 \;\mid\; 2 \;\mid\; \ldots \;\mid\; 9 \quad (3.4)$$

Now when we generate strings in the alphabet, we have to generate them with the brackets. Thus, the string $3 * 4 + 5$ is no longer a legitimate string in the language. What is then a legitimate string? The strings $(3 * 4) + 5$ and $3 * (4 + 5)$ are certainly legitimate strings. Both the strings have unique parse

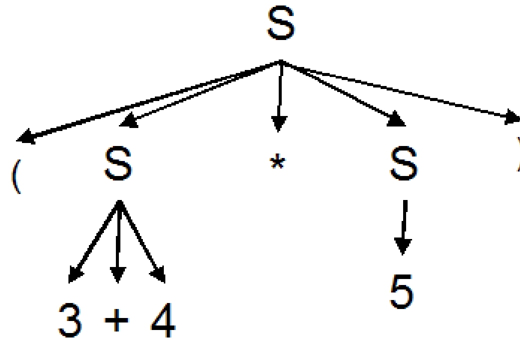

Figure 3.5: A *parse tree* for the string $(3 + 4) * 5$ using the grammar in (3.4)

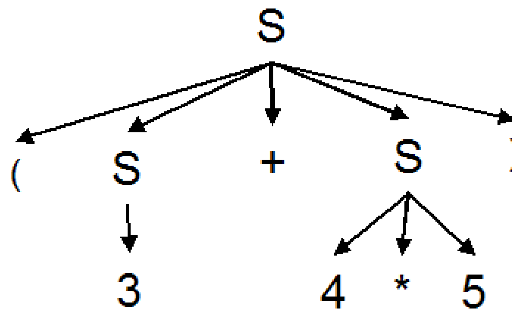

Figure 3.6: A *parse tree* for the string $3 + (4 * 5)$ using the grammar in (3.4)

trees. Figure 3.5 shows the unique derivation tree for the string $(3 * 4) + 5$ while the unique parse tree for $3 * (4 + 5)$ is shown in Figure 3.6.

We got the ambiguity out of the grammar by forcing the programmer to write the program with semantic interpretation of the strings. This is one way of removing ambiguity. There are other ways too. The purpose of this exercise was to illustrate that we could have different grammars generate the same string such that one grammar is ambiguous and another is not ambiguous.

3.2 Techniques for Writing Context Free Grammars

We will take more examples of context free grammars, focussing not so much on ambiguity and derivations but focussing more on how to build context free grammar for a given problem. It is harder than building FSMs. Grammars tend to challenge people more than the challenge of writing machines. Machines are

processes and most people can write programs, figuring out at each step what to do next. The process of writing machines is iterative. But grammars tend to be very elusive in pinning them down. We will mention two common practices adopted for writing context free grammars.

Semantic meaning for non-terminals: The first idea has some commonality with writing machines; assign to each non-terminal a semantic meaning and keep that semantic meaning consistent. This is tougher than writing machines though.

Recursive definition: The second style of writing grammars is to use a recursive idea and define the grammar recursively.

We will take some examples, where we will build grammars using the technique of ascribing semantic meanings to non-terminals.

3.2.1 Example 4

Write a grammar that generates the following language $L^{\#}$.

$$L^{\#} = \{x | x \text{ } is \text{ } made \text{ } up \text{ } of \text{ } an \text{ } equal \text{ } number \text{ } of \text{ } 0's \text{ } and \text{ } 1's\}$$

The 0's need not come before the 1's as in 0^n1^n. We have already proved that this set is not regular. We have to be sure that every string in $L^{\#}$ is generated by the grammar and that every string not in $L^{\#}$ is not generated by the grammar.

We will start by ascribing semantics to the start state. The start state S should generate any string that has an equal number of 0's and 1's. We could start with a 0 or 1 at S. Let us say that if we start with a 0 at S, we enter a new semantic state A and if we start with a 1, we enter a new semantic state B. Alternatively, S could also generate the empty string. The semantic interpretation of A is that it generates all strings that have one more 1 than 0. On the other hand B generates every string that has one more 0 than 1. The first set of productions are therefore:

$$S \rightarrow 0\,A \quad | \quad 1\,B \quad | \quad \epsilon \tag{3.5}$$

How easy it is to describe these new non-terminals in terms of the existing non-terminals[3]? A has two choices; any string it generates could start with a 0 or a 1. When A generates a 1, it should continue with a string that has an equal number of 0's and 1's, which is generated by S! Thus one production from A is $A \rightarrow 1\ S$. But when A generates a 0, it should continue with a string that has two zeros more than 1's. Such a string can always be decomposed into two strings; one string that has an extra 1 followed by another string that has

[3]We do not want to keep creating new non-terminals for ever.

an extra 1. Thus, the second production from A is of the form $A \rightarrow 0\ A\ A$. We will finish off the whole grammar by enumerating productions from B next. On similar lines as for productions from A, the productions from B will be $B \rightarrow 0\ S|1\ B\ B$.

Thus the overall grammar is:

$$\begin{array}{lllllll} S & \rightarrow & 0\ A & \mid & 1\ B & \mid & \epsilon \\ A & \rightarrow & 1\ S & \mid & 0\ A\ A & & \\ B & \rightarrow & 0\ S & \mid & 1\ B\ B & & \end{array} \tag{3.6}$$

This is one way to write a grammar - think about semantics of the non-terminals and keep defining new non-terminals till they wrap around. Is this grammar ambiguous? If one thinks that the grammar is ambiguous, one needs to identify the place of ambiguity. The production $A \rightarrow 0\ A\ A$ is actually ambiguous; the sequence of symbols $A\ A$ represents a string that has two extra 1's. If a string has two extra 1's, the can be many ways the string can be split into two substrings, each having an extra 1. For instance, the string 11001100110011 can be split into four possible pairs as 11001 and 100110011 or 1 and 1001100110011 or 110011001 and 10011 or 1100110011001 and 1. Similarly the sequence $B\ B$ in $B \rightarrow 1\ B\ B$ also represents an ambiguous choice.

3.2.2 Example 5

We will illustrate with an example, the idea of defining grammars recursively. Consider the grammar:

$$\begin{array}{lllll} S & \rightarrow & S\ A\ B & \mid & \epsilon \\ A & \rightarrow & 0\ S\ 1 & \mid & \epsilon \\ B & \rightarrow & 1\ S\ 0 & \mid & \epsilon \end{array} \tag{3.7}$$

What strings does the grammar in (3.7) generate? Every string generated by the grammar definitely has an equal number of 0's and 1's. How do you prove this? It is by induction[4]. Inductive proofs are very useful in the context of grammars because by their very nature they are recursive. Intuitively, every time a 0 is introduced in the string, a 1 is also introduced, thus maintaining an equality between the number of 0's and 1's.

But does the grammar generate all strings that have an equal number of 0's and 1's? For instance, can the string 00011011 be generated by the grammar in (3.7). The answer is actually yes. In fact there are an infinite number of parse trees for the string 00011011 using the grammar in (3.7). Two parse trees are shown in Figures 3.7 and 3.8.

In fact, the grammar in (3.7) generates exactly all strings with equal number of 0's and 1's! Thus, the grammar in (3.7) is actually equivalent to the grammar

[4] Exercise.

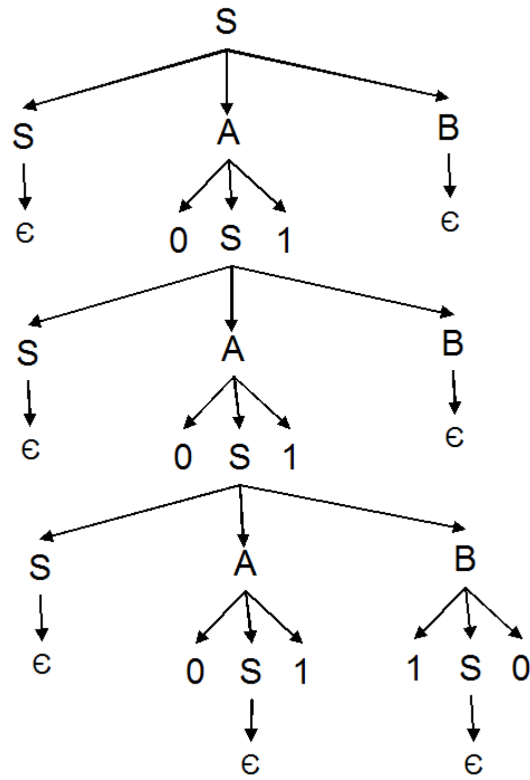

Figure 3.7: A *parse tree* for the string 00011011 using the grammar in (3.7)

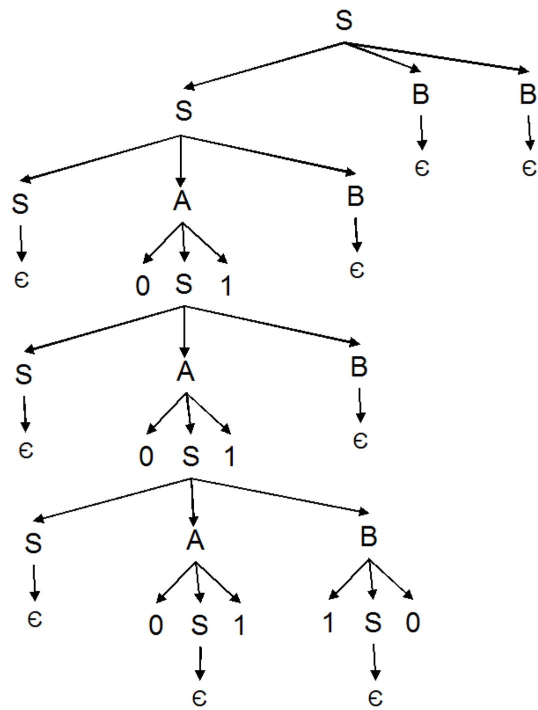

Figure 3.8: A *parse tree* for the string 00011011 using the grammar in (3.7)

in (3.6). And the problem of determining if two grammars are the same is undecidable! The only problem decidable about a grammar is the problem: *Does this grammar accept the empty set?*. Each rule in the grammar has only two choices on the right hand side; one involving a non-terminal and another involving an empty production. But this choice is enough complexity to make every simple question about these kind of grammars undecidable and make them very hard. There are few things that are easier about CFGs than regular grammars. But for a Turing machine, we cannot even decide if the turing machine accepts nothing. But with a context free grammar, we can at least do that - whether it generates an empty set. But the problem of deciding whether a given CFG accepts all strings is undecidable.

How do we interpret the recursive grammar in (3.7)? S can have any strings A's and B's. A and B are also strings of equal 0's and 1's. A has 1's after 0's with a string defined recursively by S in the middle. B has 0's after 1's with a string defined recursively by S in the middle. Taking forward these semantics recursively, we could convince ourselves that the grammar generates exactly all the strings that have an equal number of 0's and 1's.

In general, grammars can be very elusive. There is a pumping lemma for languages generated by context free grammars (somewhat similar to the pumping lemma for regular languages) which can prove that certain languages cannot be context free. An example language that cannot be context free is $L = \{0^n 1^n 0^n | n \geq 0\}$; triple counting is not context free, just as counting in general was outside the purview of finite state machines.

3.2.3 Example 6

Consider the language $L = \{0^n 1^n 0^n\}$. As mentioned in the previous section[5], this language cannot be generated by any context free grammar. We will present one example of a powerful extension to context free grammars, known as *context sensitive grammars*. We will see that context sensitive grammars look very much like machines and are very powerful; machine computations for context sensitive grammars can be somewhat directly simulated with the grammar itself. And this connection does not work quite as well for context free grammars.

The grammar is somewhat long, though not complicated. As a motivation, think of an amusement park; in amusement parks you are enticed into a game which you cannot possibly win. And one of the games in an amusement park is a shooting game, where some object/target is moving by in a zig-zag path. Imagine the grammar moving from the left set of 0's to the right set of 0's in $\{0^n 1^n 0^n\}$ and as it performs its movements, it will generate strings in the language. Let non-terminals L and R serve as the left and right ends of the shooting gallery and let non-terminal T represent the target that is moving back and forth. A, B and C are non-terminals that will respectively eventually turn into and 0's 1's and 0's.

$$S \rightarrow L\ T\ A\ B\ C\ R$$

[5] Will be proved in a following section.

The idea is that as the target moves from the left to the right, and sees an A, it doubles the A. Similarly, as it sees a B and C, it will go ahead, leaving two Bs and Cs behind respectively. When it sees an R, it goes right back in the beginning beside L. The sequence of doubling As, Bs and Cs can continue thereafter. At any point along the way, if it makes it back to the left end to give the pattern, $L\ B$, it could terminate but turning the A's into 0's, the B's into 1's and the C's into 0's. This grammar can only generate a string if it moves all the back to the beginning leaving behind an equal number of terminals produced using the As, Bs and Cs.

The idea of moving the target forward proceeding with the computation appears in the form of the rules $L\ T\ A \rightarrow L\ A\ A\ T$. This rule has more than a single symbol on the left hand side and is definitely not context free. In fact, it is context sensitive. We cannot just substitute $L\ A\ A\ T$ for a single A. The substitution can take place only in the presence of L and T preceding A. Another rule in the grammar should be $A\ T\ A \rightarrow A\ A\ T$ - the target can pass through the A's if no extra context is specified. On similar lines, we introduce the rules $B\ T\ B \rightarrow B\ B\ T$ and $C\ T\ C \rightarrow C\ C\ T$ Another rule is $A\ T\ B \rightarrow A\ B\ B\ T$; when the target finally hits a B in the presence of an A, it moves forward after doubling the B's. Similarly, $B\ T\ C \rightarrow B\ C\ C\ T$ is introduced. We need one production to handle when T hits the R and send back the target to the beginning. We achieve this by painting the target with a non-terminal E to remember that the target has turned around left-wards, *i.e.*, $T\ R \rightarrow E\ R$. What do we want E to do? It should pass by all the non-terminals left-ward and get back to L. This yields the following set of rules: $C\ E \rightarrow E\ C$, $B\ E \rightarrow E\ B$ and $A\ E \rightarrow E\ A$. Finally, when E hits L, it should recover its original identity as T, that is, $L\ E \rightarrow L\ T$. Finally, we have some rules that take non-terminals to terminals, *viz.*, $A \rightarrow 0$, $B \rightarrow 1$, $C \rightarrow 0$. To get rid of the R, L and the target T, we note that R could turn into the empty string at any point of time using the rule $R \rightarrow \epsilon$. Only when T comes back to the beginning, can you shut down the machine as follows: $L\ T \rightarrow \epsilon$. The grammar is presented in (3.8).

$$
\begin{array}{lcl}
S & \rightarrow & L\ T\ A\ B\ C\ R \\
L\ T\ A & \rightarrow & L\ A\ A\ T \\
A\ T\ B & \rightarrow & A\ B\ B\ T \\
B\ T\ C & \rightarrow & B\ C\ C\ T \\
A\ T\ A & \rightarrow & A\ A\ T \\
B\ T\ B & \rightarrow & B\ B\ T \\
C\ T\ C & \rightarrow & C\ C\ T \\
T\ R & \rightarrow & E\ R \\
C\ E & \rightarrow & E\ C \\
B\ E & \rightarrow & E\ B \\
A\ E & \rightarrow & E\ A \\
L\ E & \rightarrow & L\ T \\
R & \rightarrow & \epsilon \\
L\ T & \rightarrow & \epsilon
\end{array}
\tag{3.8}
$$

We will briefly contrast context free grammar against context sensitive grammar. A rule like $L\ T\ A \rightarrow L\ A\ A\ T$ above appears as if involving the reading of a tape; we are in a certain state when we see an A and move the T to the other side leaving an extra A behind. This brings us closer to the computation of a Turing machine that reads inputs on a tape and is capable of inserting and writing new entries on the tape. Turing machines will be covered in Chapter **??**. Context sensitivity gives much more control and lets you define grammars that look like machines. Rules in a context sensitive grammar could always be in a context free form. But a context free grammar cannot contain context sensitive rules.

The context sensitive grammar in (3.8) generates all strings in the language $L = \{0^n 1^n 0^n\}$ and helps us appreciate the connection between grammar and computation.

3.2.4 Example 7

Consider the language $L = \{0^n 1^n 0^p | n \geq 0, p \geq 0$. The context free grammar for this language is as follows:

$$
\begin{array}{lcll}
S & \rightarrow & NM & \\
N & \rightarrow & 0\ N\ 1 & | \quad \epsilon \\
M & \rightarrow & 0\ M & | \quad \epsilon
\end{array}
\tag{3.9}
$$

This example hints that context free grammars could be closed under concatenation. Given a grammar G_1 with start symbol S_1 and a set of rules that include $S_1 \rightarrow \ldots$ and another grammar G_2 with start symbol S_2 and a set of rules that include $S_2 \rightarrow \ldots$, the concatenation of the two grammars can be generated by a grammar G consisting of a start state S with a production $S \rightarrow S_1 S_2$ along with all the productions of G_1 and G_2. This new grammar

precisely generates all strings in the union of the languages of G_1 and G_2. To prove that a grammar G generates precisely all strings in a language L, one needs to prove that for every string $\sigma \in L$, there exists parse for σ using the grammar G and that there is no string $\sigma \notin L$ that can be parsed by G. It might happen that the grammar is not efficient for parsing L, but that does not affect the discussion here.

3.2.5 Closure under Intersection and Complement

Can we construct a context free grammar for the language $L^R = \{0^q 1^n 0^n | n \geq 0, q \geq 0$? Yes, we can.

$$\begin{array}{lclcl} S & \rightarrow & A\ B & & \\ A & \rightarrow & 0\ A & | & \epsilon \\ B & \rightarrow & 1\ B\ 0 & | & \epsilon \end{array} \tag{3.10}$$

Note that the grammar in (3.10) is a concatenation of two grammars, one with a start state M and another with start state N. What is the intersection of $L^R = \{0^q 1^n 0^n | n \geq 0, q \geq 0$ and $L = \{0^n 1^n 0^p | n \geq 0, p \geq 0$? In fact, $L \cap L^R = \{0^m 1^m 0^m | m \geq 0\}$! We have already seen that we could write only a context sensitive grammar for $L \cap L^R$ (in fact the language is not context free as will be proved using the pumping lemma), wheres L and L^R are both context free. Thus, context free languages are not closed under intersection (proof by couter example).

Are context free languages closed under complement? Context free languages are closed under union, but not under intersection. Thus, they are not closed under complement. The complement of some specific context free grammars is context free. For instance, the complements of $L = \{0^n 1^n | n \geq 0\}$ and $L = \{x | x\ \textit{is a palindrome}\}$ are context free. Strangely, while the language $L = \{ww | w\ \textit{is a string}\}$ is not context free, its complement is context free! However, the problem of determining if the complement of a grammar is context free is itself an undecidable problem.

3.2.6 Closure under Union

It is very easy to determine the closure of context free grammars under union. Given a grammar G_1 with start symbol S_1 and a set of rules that include $S_1 \rightarrow \ldots$ and another grammar G_2 with start symbol S_2 and a set of rules that include $S_2 \rightarrow \ldots$, the union of the two grammars can be generated by a grammar G consisting of a start state S with a production $S \rightarrow S_1 | S_2$ along with all the productions of G_1 and G_2. This new grammar precisely generates all strings in the union of the languages of G_1 and G_2.

3.3 Relationship to Compiling and Programming Languages

So far we mainly used the alphabet $\Sigma = \{0, 1\}$ in describing examples of context free languages. Can context free languages be expressive enough to describe programming languages? If so, what is the alphabet. In this section, we will make this connection explicit by relating CFGs to programming language syntax.

Let $< stmt >$ be our starting non-terminal representing a statement in a programming language. String enclosed in angle brackets will indicate non-terminals. A statement in a programming language could mean an (a) assignment statement ($< assgn >$) (b) if-then statement ($< if - then >$) (c) if-then-else statement ($< if - then - else >$) (d) begin-end block ($< begin - end >$). This can be represented using the following rule:

$$< stmt > \rightarrow < assgn > \mid < if{-}then > \mid < if{-}then{-}else > \mid < begin{-}end >$$

The $< if - then >$ statement will have some terminals:

$$< if - then > \rightarrow if \ \ < expression > \ \ then \ \ < stmt >$$

While the $< if{-}then{-}else >$ statement is an extension of the $< if{-}then >$ statement:

$$< if - then - else > \rightarrow if \ \ < expression > \ \ then \ \ < stmt > else < stmt >$$

We can similarly write the other statements:

$$< begin - end > \rightarrow begin \ \ < stmt - list > \ \ end$$

The last statement sounds more like a declaration for the Pascal[6] language (C uses open curly and close curly braces instead of 'begin' and 'end' respectively).

With $< stmt - list >$, we will have to invoke a recursive definition[7]:

$$< stmt - list > \rightarrow begin \ \ < stmt - list > < stmt >$$

$$< assgn > \rightarrow < ID > < expression >$$

The $< expression >$ in the above rule is principally different from the $< expression >$ for the $< if{-}then >$ rule, but we will not focus on this distinction for the time being.

A typical identifier in a programming language looks like a letter followed by a sequence of letters or digits.

$$< ID > \rightarrow < ID > < letter > < temp >$$

$$< temp > \rightarrow < temp > < letter > \ \mid < temp > < digit > \ \mid epsilon$$

$< letter >$ goes to one of $A, B, \ldots, Z$ and $< digit >$ goes to one of $0, 1, 2, ldots, 9$.

[6] The entire grammar specification for Pascal will take only two and half pages - Pascal is that simple.

[7] We could instead write a syntax grammar with a single loop around $< stmt - list >$

- This part of scanning $< ID >$ is typically performed in compilers by the *scanner*, which is a finite state machine implementation. A finite state machine can pull out tokens such as 'if', 'else' as well as IDs. Scanner forms the first component in the compiler pipeline. The tool for scanning is called *LEX*.
- The second component in the pipeline is called the *parser*, which checks the syntax of the program using parsing based on context free grammar rules such as the ones listed above. Tokens are potential strings in the context free language. The tool for parsing is called *YACC* (Yet Another Compiler Compiler).
- The final component in the compiler pipeline generates code that is executable. This component deals with symbol tables and other issues that come up with the different possible ways of writing a program. This will not be addressed.

3.4 Normal Forms

It is very useful to be able to take an arbitrary context free grammar and transform it into a form that has a more specific structure. We already saw that CFGs have a single non-terminal on the left but any arbitrary sequence of terminals and non-terminals on the right. It will be nice if, without any loss of generality, we could specify a form for the right hand side of each rule. An imposition such as '*the right hand side should be a terminal followed by a non-terminal*' does restrict the grammar and takes us back to the finite state machine level. Is there something in between where we could restrict the right hand side without losing any power of context free grammar? Yes, there are many such forms, a prominent one being the *Chomsky Normal Form*. Another normal form is the *Greibach Normal Form*.

3.4.1 Chomsky Normal Form

Chomsky normal form is a very useful form for at least four different reasons. The actual detail of converting between chomsky normal form and the most general form for CFGs is probably the driest part of the whole topic and is relatively starightforward with some subtleties. The motivations for chomsky normal form[8] are:

1. Every string of length n is derivable in $2n - 1$ steps. Given any arbitrary CFG, if you try to derive a string with k symbols, you have no idea of the number of steps it might take; it could take $100 \times k$ or it might take $2 \times k$. You have to keep trying till the string is produced. There is no upper-bound on the number of steps it might take. But if the grammar is represented in Chomsky normal form, every string of length n is derivable

[8] Chomsky normal form was originally motivated by linguistics.

(if at all you can derive it) in $2n-1$ steps. This is a very useful fact to know, because the question of '*whether we can write an algorithm to decide if the grammar really derives a given string*' (for example the parsing algorithm in a compiler) can be immediatly answered. The algorithm for deriving a string using a grammar expressed in chomsky normal form does not look very neat though - the algorithm is to simply try every sequence of $2n-1$ steps and if you do not get the string, you decide that the string cannot be derived. How long does this trial take? Let us say there is an average of p productions from every non-terminal. Then the number of possible trees is $O(p^{2n-1})$. This is a horrible exponential time algorithm and we will show that we can do much better than this. But this at least immediatly implies that there is an algorithm for deriving the string and that the problem is not undecidable.

2. Chomsky normal form enables easy proof of the *pumping lemma* for context free languages. This is much more complicated than the pumping lemma for regular sets. Recall that the proof for the pumping lemma was based on a machine definition for finite state machines (using the idea of a loop). But the analogy does not carry to grammars that are context free. The loop will be in the form of discovering a non-terminal in the parse tree repeating itself repeatedly.

3. Context free grammars are equiavalent to *non-deterministic push down machines*[9] (NPDM) and they can be converted from one to the other. However, their equivalence is nowhere close to obvious like we discovered at the level of finite state machines. However, the fact that '*every CFG has an equivalent NPDM*' can be proved easily if we have the grammar represented in the chomsky normal form. The other part of the equivalence, that is, for any given NPDM, there is an equivalent CFG is subtle and a bit difficult.

4. When the CFG is represented in chomsky normal form, there is a reasobly efficient algorithm to determine membership of strings to the grammar. The algorithm is not very practical but it is better than exponential. It is a dynamic programming algorithm called the *CYK algorithm*. It is very easy to describe if the grammar is in Chomsky normal form. It is an $O(n^3)$ algorithm. Nobody uses this algorithm in real compilers, given that there are linear time algorithms for compilers, that work with LRK grammars, a special subset of context free grammars. LRK grammars correspond to deterministic context free grammars, a subset of context free grammars. These are unambiguous grammars used in practical compliers for programming languages.

[9]A non-deterministic push down machine is a non-deterministic finite state automaton with an additional stack data structure that it can manipulate, such that every time it moves from one state to another, it can push or pop an arbitrary symbol on a stack. If you give an FSM two stacks, you give it the power of a turing machine.

Definition: Every single production in Chomsky Normal Form (CNF) has one of the following two forms

1. $< Non{-}Terminal > \rightarrow < Non{-}Terminal >< Non{-}Terminal >$
2. $< Non - Terminal > \rightarrow < terminal >$
3. $< Start - Non - Terminal > \rightarrow\ \epsilon$. This production will be a part of the grammar for a language only if the empty string ϵ is in the language.

Note that empty string productions are not allowed from any non-terminal except the start symbol, in the chomsky normal form[10]. This is simple, yet offers enough flexibility to capture any context free language. Any given context free grammar can be converted into a grammar with rules that have chomsky normal form. If we replace any one of the non-terminals on the right hand side of the first rule with a terminal, we immediatly restrict the grammar to a left or right linear grammar, which is equivalent to regular sets (as shown in the previous chapter).

How do we show that any grammar can be converted into the chomsky normal form? We will show the precise steps involved in the transformation. Consider the grammar below, which is not completely in CNF.

$$\begin{array}{lllllll} S & \rightarrow & 0\ A\ 1\ B & | & C & | & D\ E\ F \\ A & \rightarrow & \epsilon & | & C\ C & & \end{array} \tag{3.11}$$

What kind of productions are challenging for transformation? An example is $S \rightarrow C$. Another example is the empty production $A \rightarrow \epsilon$. How do we fix the productions that are not in CNF, without losing the semantics of the grammar? To start with, we can introduce non-terminals Z and X that represent 0 and 1 respectively, through the productions

$$\begin{array}{lll} Z & \rightarrow & 0 \\ X & \rightarrow & 1 \end{array} \tag{3.12}$$

This gives us the following set of productions:

$$\begin{array}{lllllll} S & \rightarrow & Z\ A\ X\ B & | & C & | & D\ E\ F \\ A & \rightarrow & \epsilon & | & C\ C & & \\ Z & \rightarrow & 0 & & & & \\ X & \rightarrow & 1 & & & & \end{array} \tag{3.13}$$

[10]This is the reason why we can produce a string in $2n - 1$ steps; at every string, we can only increase the length of the string produced - we are not allowed to erase symbols.

The productions in (3.13) can be divided into two categories, *viz.*, productions that are too long and productions that are too short. The sequence of non-terminals $A\ X\ B$ can be grouped into a new non-terminal M and a new production from M can be introduced. Similarly, the sequence of non-terminals $X\ B$ can be grouped together into a new non-terminal N and a new production from N can be introduced. M and N are simply place-holders; while introducing them we need to ensure that they do not already appear in the grammar. Similarly, the sequence $E\ F$ can be grouped into a new non-terminal P.

$$\begin{array}{lllllll} S & \rightarrow & Z\ M & \mid & C & \mid & D\ P \\ A & \rightarrow & \epsilon & \mid & C\ C & & \\ M & \rightarrow & A\ N & & & & \\ N & \rightarrow & X\ B & & & & \\ P & \rightarrow & E\ F & & & & \end{array} \tag{3.14}$$

The productions that are too short are slighty more tricky to handle. Short productions are of the form

Unit production: $< non-terminal >_1 \rightarrow < non-terminal >_2$

ϵ-production $< non-terminal >_1 \rightarrow \epsilon$

For every rule of the above two forms, we could generate all possible replacements for occurences of $< non-terminal >_1$ on right hand side of other productions with $< non-terminal >_2$ as well as replacements with ϵ. Applying this procedure to (3.14), we obtain,

$$\begin{array}{lllll} S & \rightarrow & Z\ M & \mid & D\ P \\ A & \rightarrow & \epsilon & \mid & S\ S \\ M & \rightarrow & A\ N & & \\ N & \rightarrow & X\ B & & \\ P & \rightarrow & E\ F & & \end{array} \tag{3.15}$$

Consider another set of productions that will help us point out subtleties while handling ϵ productions and *unit* productions for transforming an arbitrary CFG to chomsky normal form.

$$\begin{array}{lllllll} S & \rightarrow & 0 & \mid & X\ O & \mid & Z\ Y\ Z \\ X & \rightarrow & Y & \mid & \epsilon & & \\ Y & \rightarrow & 1 & \mid & X & & \end{array} \tag{3.16}$$

The sequence of substitutions will be as follows:

1. Get rid of useless productions.

2. We will first do substitutions for ϵ productions before the substitutions for the unit productions, because we get unit productions when we substitute for ϵ productions.

3. However, it is not a good idea to substitute for ϵ productions in rules that have a single non-terminal on the right hand side. For example, substituting X with ϵ in $Y \rightarrow 1 \mid X$ does not make sense, since it only yields another ϵ production in the form $Y \rightarrow 1 \mid \epsilon$ and substituting for $Y \rightarrow \epsilon$ in $X \rightarrow Y$ yields $X \rightarrow \epsilon$ again! Instead, a set of non-terminals, called the *nullable set* is maintained which contains non-terminals that can sooner or later disappear (not necessarily in one step). Every non-terminal that can produce an ϵ following some ϵ-substitution is added to the *nullable set.* For example, X is initially nullable, and after an ϵ substitution for X in the rule with Y on the left hand side, Y is also added to the nullable set. The nullable set can be computed by working our way backwards by (a) adding ϵ producing non-terminals to the nullable set and (b) iteratively adding non-terminals that have unit productions to the nullable set.

4. Once the nullable set is determined, all ϵ productions are removed. For every production of the form $< non-terminal >_1 \rightarrow < non-terminal >_2 < non-terminal >_3$, if $< non-terminal >_2$ happens to be nullable, a corresponding new rule $< non-terminal >_1 \rightarrow < non-terminal >_3$ is added. Similarly, if $< non-terminal >_3$ happens to be nullable, a corresponding new rule $< non-terminal >_1 \rightarrow < non-terminal >_2$ is introduced.

$$\begin{array}{lclllllllll} S & \rightarrow & 0 & \mid & X\ O & \mid & Z\ Y\ Z & \mid & O & \mid & Z\ Z \\ X & \rightarrow & Y \\ Y & \rightarrow & 1 & \mid & X \\ Nullable\ set & = & \{X, Y\} \end{array} \tag{3.17}$$

5. To get rid of a unit production, such as $X \rightarrow Y$, we keep track of all unit productions that originate at X. create a production from X for each production from Y, with the right hand side of the production for X set to the right hand side of the corresponding production from Y.

$$\begin{array}{lclllll} S & \rightarrow & 0 & \mid & X\ O & \mid & Z\ Y\ Z \\ X & \rightarrow & 1 & \mid & X \end{array} \tag{3.18}$$

But there are some subtleties with substitutions for unit productions.

Consider the grammar:

$$\begin{array}{lllll} S & \rightarrow & A & | & 1\,1 \\ A & \rightarrow & B & | & 1 \\ B & \rightarrow & S & | & 0 \end{array} \tag{3.19}$$

We do not want to add new unit productions.

$$\begin{array}{lllllll} S & \rightarrow & 1\,1 & | & 1 & & \\ A & \rightarrow & 1 & | & 0 & & \\ B & \rightarrow & S & | & 11 & | & 1 \end{array} \tag{3.20}$$

But there is a mistake in this; S can actually produce 0, whereas our series of substitutions for unit productions has made it impossible for S to generate a 0! This does not happen so often; it happens when you have chains of recursion. The subtlety here is that you need to keep track of non-terminals that can result from a non-terminal X using only unit-productions. This looked ahead for extended unit productions is akin to keeping track of nullable non-terminals where you had to look at extended ϵ productions. You will have a list of single non-terminals that any X can get to, by more than one step. The set of non-terminals, reachable using a series of unit productions from a non-terminal X will be represented as $\mathcal{U}_X$. Thus, $\mathcal{U}_S = \{A, B\}$, $\mathcal{U}_A = \{B, S\}$ and $\mathcal{U}_B = \{S, A\}$. For any non-terminal X, we will place all the right hand sides of the productions from $\mathcal{U}_X$ on the right side of productions from X. Thus, the above grammar, after this modification, should look like:

$$\begin{array}{lllllll} S & \rightarrow & 1\,1 & | & 1 & | & 0 \\ A & \rightarrow & 1 & | & 1\,1 & | & 0 \\ B & \rightarrow & 0 & | & 1\,1 & | & 1 \end{array} \tag{3.21}$$

6. Get rid of long productions, as illustrated with the first example. In (5), there are no long productions and the grammar is already in CNF.

7. Get rid of useless productions. The original grammar in (3.19) was capable of generating three strings, *viz.*, 1 1, 0 and 1. There was no useless productions or non-terminal in (3.19). However, in (5) there are useless symbols in the form of A and B (since you cannot even get to them starting at S) as well as useless productions in the form of the last six productions in (5). Thus, the grammar in (5) is equivalent to the grammar in (7)

$$S \rightarrow 1\,1 \;\mid\; 1 \;\mid\; 0 \qquad (3.22)$$

How do we get rid of useless non-terminals? There are two steps to this process. First of all, when is a symbol useless?

- A symbol is useless, if it cannot be reached from the start symbol S (as in the case of A and B above).
- A symbol is also useless, if does not terminate. For example, if we had a production $S \rightarrow C\ C$ in (7), but had no productions originating with C (that is, we could do nothing useful with C), then this additional production would be useless.

The steps involved in eliminating useless productions are:

(a) Find all productions that can generate strings. Eliminate all other productions. This can be determined in a manner similar to determination of nullable non-terminals; we start with terminals, keep track of the list of all useful non-terminals that produce terminals, then iteratively keep track of non-terminals that produce useful non-terminals. All non-terminals that do not get included in the list are useless non-terminals. Thus productions such as $S \rightarrow C\ C$ and $S \rightarrow B\ C$ will get eliminated. The latter gets eliminated, because B is hitched up with C, which will lead us nowhere.

(b) Find all non-terminals that can be reached from the start symbol S. This can be done using a similar iterative procedure as discussed above. Eliminate all other non-terminals and any production that involves them.

Both the procedures described above, as well as the procedure for determining nullable non-terminals, takes time proportional to the number of non-terminals. On the other hand, the step for substituting information based on nullable non-terminals can take time that is exponential in the number of nullable non-terminals in a production. Thus, if N is a nullable non-terminal and $S \rightarrow A\ N\ B\ N\ C\ N\ D$ is a production, the time taken for substituting for N in this production is 2^3, which is exponential in the number of occurences of the nullable non-terminal on the right hand side of a production.

Consider the simple example grammar below:

$$\begin{array}{lcl} S & \rightarrow & A\ B \;\mid\; 0 \\ A & \rightarrow & 0 \end{array} \qquad (3.23)$$

Following the two steps for elimination of useless productions:

(a) B is obviously a useless non-terminal and therefore the production $S \rightarrow A\ B$ can be eliminated.

(b) With elimination of this production, the non-terminal A and its corresponding production $A \rightarrow 0$ are also rendered useless and can also be eliminated. Therefore, the only production that will remain in an equivalent grammar is

$$S \rightarrow 0 \tag{3.24}$$

The order of the two steps is very important. If we first eliminate all non-reachable non-terminals and their corresponding productions, A and its production $A \rightarrow 0$ will remain; only the production $S \rightarrow A\ B$ will get eliminated in the second step. What will remain is

$$\begin{array}{lcl} S & \rightarrow & 0 \\ A & \rightarrow & 0 \end{array} \tag{3.25}$$

While we pretended to have eliminated all useless productions, the non-terminal A is actually useless and so is the second production. Thus shows the fallacy in reversing the order of the elimination steps.

3.5 Does a CFG Generate any String

One question that is decidable about CFGs is the membership question; *is a given string σ generated by a given CFG G*? Another question that is also decidable about CFGs is *does a given CFG G generate any string at all*? Or in other words, does the given grammar G generate an empty set of strings? The procedure for determining if a CFG G generates any string at all is simple:

1. Convert G to Chomsky Normal Form G_F.

2. If the start symbol of G_F is useless, then G generates no strings. That is, if G_F has no productions, then the set of strings generated by G is empty.

On the other hand, the question '*does a given CFG G generate all strings*?' is undecidable.

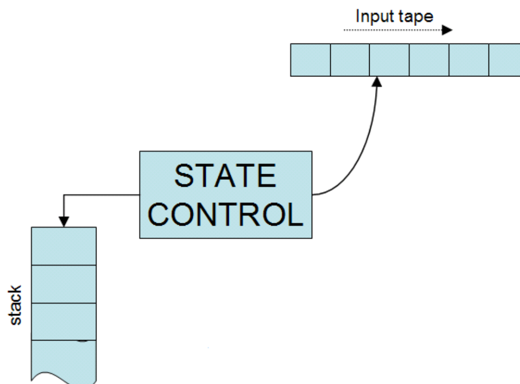

Figure 3.9: Schematic diagram of a pushdown automaton.

3.6 Pushdown Machines

We will switch gears now, while retaining our focus on the broad family of context free languages. We will introduce the machine equivalents for context free grammars, called *non-deterministic machines*. We will also distinguish between these machines and deterministic pushdown machines.

A pushdown machine is just like a finite state machine, except that we put in an extra piece of power in the form of a stack. The machine, while making a move from one state to another state on observing a symbol on the tape, can also manipulate a stack by pushing a symbol on the stack or popping a symbol off the stack. It is pictorially more abstract than the FSM and is shown in Figure 3.9.

The arrow represents a pointer that the FSM uses to keep track of a cell on the input tape. The pointer always moves unidirectionally, left to right on the input tape, step by step. It does not have the power to go back and forth on the tape. A finite state machine that is allowed two way movement of the pointer on the tape will have more power and will be dealt with later in the form of a Turing Machine. The PDM also has a stack on which symbols are stored. The stack is a last-in first-out data structure. When we write a program to simulate a PDM like this, we need to represent the machine using transition tables. To formally define a PDM, we just need to extend the notation used for FSMs by merely adding the stack and the stack alphabet to the existing 5 tuple. Moreover, the transition function takes as an input, the current state, and an input from the top of the stack to return a state and a stack symbol (which again goes into the stack). Formally, we define a push-down automaton as follows

Push Down Automaton: A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

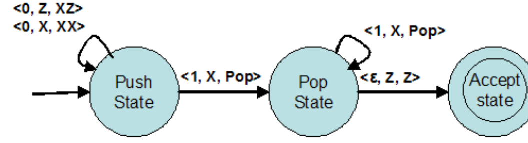

Figure 3.10: Pushdown automaton that accepts the language $L = \{0^n 1^n | n \geq 0\}$.

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

3.6.1 Example 1

Consider the language $L = \{0^n 1^n | n \geq 0\}$. We know that L is not regular. We will construct a PDM that accepts L. We need to manipulate the stack depending on the symbol at the top of the stack as well as the current state. To start with, we read the tape and as long as we see 0's, we keep pushing X's (one X per 0) onto the top of the stack, and when we start seeing 1's, we start popping off X's from the top of the stack. And while popping off X's, if we encounter 0's, we crash (that is, we hit a dead state).

We will have a 'push state' with the semantics that we stay on the state as long as we encounter 0's and keep pushing X's on the stack. The triplet $< 0, Z, XZ >$ (Z represents the bottom of the stack) means a push; it means when the next symbol on the tape is a 0 and the stack is empty, then push the symbol X on top of the stack to yield XZ. Similarly, the triplet $< 0, X, XX >$ also means a push; it means when the next symbol on the tape is a 0 and there is an X on the top of the stack, then push another symbol X onto the top of the stack to yield XX. If we see a 1 in the start state ('push state'), we go into popping mode by transitioning to the 'pop state', while popping an X. The triplet $< 1, X, Pop >$ means that if the next symbol on the tape is 1 and X is on the top of the stack, then pop out the X from the top of the stack. Once in the 'pop state', we keep popping an X to match every input 1. However, if the input is exhausted while on the 'pop state', and there is no X on the top of the stack, then the PDM goes into the accept mode by ϵ transitioning to the 'accept state'.

For FSMs, whenever we had an ϵ move, the machine automatically became non-deterministic. In PDMs, the presence of an ϵ move does not necessarily imply that it is non-deterministic. Intuitively, non-determinism means that at a particular configuration, you do not know what to do, since you have some choices. In the case of FSMs, an ϵ move indicates a choice. But in the case

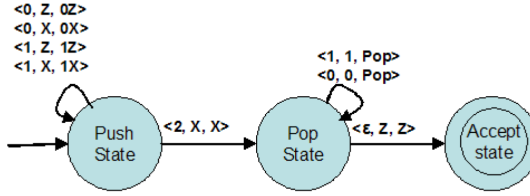

Figure 3.11: Pushdown automaton that accepts a special language of palindromes, $L_2 = \{w2w^R | w \in (0+1)^*\}$.

of PDMs, since the transition is based not only on an ϵ input, but also on the symbol at the top of the stack, you have to look at the combination, before you decide the next move. If we put an $< \epsilon, X, Pop >$ on the loop on the 'pop state', it will yield non-determinism, since there will be a choice for an input symbol 1 with XX at the top of the stack; we could either loop around twice, once for an ϵ and then for a 1 or alternatively, loop around directly for a 1. The summary of the discussion is that if there is no choice of next state for any configuration of the PDM, then the machine is deterministic, else the machine is non-deterministic.

YACC simulates a pushdown machine. Note that if the stack alphabet is restricted to be always of size 1, then we cannot handle context free languages such as palindromes[11].

3.6.2 Example 2

We will do another example that will force us to use more than one symbol on the stack. Consider the language L of even length palindromes $L = \{ww^R | w \in (0+1)^*\}$. We can read the first half w of the string ww^R and as we keep reading in symbols of w, we keep pushing them onto the top of the stack[12]. And then as we read symbols from w^R, we pop symbols from the top of the stack, and when the stack empties we move to the accept state. But how do we know when w ends and w^R starts? Let us say we slightly modify the language to $L_2 = \{w2w^R | w \in (0+1)^*\}$. This is a similar but much simpler problem. The problem of designing a PDM for L_2 is simpler because we can now detect the start of w^R. The corresponding PDM is shown in Figure 3.11.

Here is how we build a PDM that accepts the language $L = \{ww^R | w \in (0+1)^*\}$ without the intervening 2. We do this by introducing non-determinism; when we see a 0 or 1, the symbol could potentially be a part of w or of w^R. So by substituting $< \epsilon, X, X >$ for $< 2, X, X >$, we can ensure that all palindromes will be accepted by the modified PDM. The only subtle issue could be that by making this substitution, other strings (that is non-palindromes), that were not

[11]But for single alphabet languages, it can be the case that a single alphabet stack is powerful enough to capture CFLs.

[12]The stack alphabet, in this case, will be the same as the alphabet of the language.

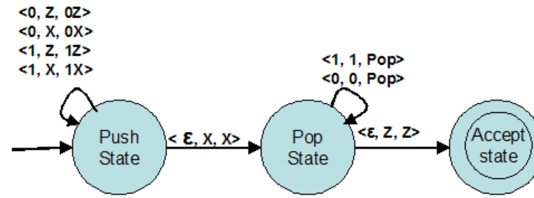

Figure 3.12: Pushdown automaton that accepts even length palindromes, $L = \{ww^R | w \in (0+1)^*\}$.

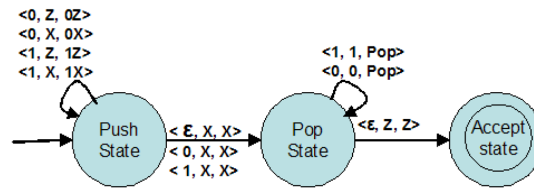

Figure 3.13: Pushdown automaton that accepts even as well as odd length palindromes.

accepted by 3.11 might get accepted. But by inspection, we can ward off this danger. Figure 3.12 shows a PDM that accepts only even length palindromes.

By introducing $< 0, X, X >$ and $< 1, X, X >$ along with $< \epsilon, X, X >$, we get a PDM that accepts even as well as odd length palindromes.

There is no PDM that accepts $L = \{ww | w \in (0+1)^*\}$. If we try inserting the symbols of the first half, w and then try popping them out based on the second half, we simply will not be able to, because the data structure is last-in first-out. Could we do this if we had two stacks? Yes we could. We could also do this with a 2-way machine (a machine that can read backward as well as forward on the tape). However a 2-way PDM machine is not as powerful as a Turing machine. Some things that 2-way deterministic PDMs can do, NPDMs cannot do and vice-versa. Figure 3.14 shows a high level overview of the power of different PDMs.

What about a queue? Can a queue simulate one or more stacks? Even with a single queue, a queue automata is equivalent in computational power to a Turing machine. Hence a queue automata cannot be simulated by a one stack automata (the pushdown automata). Although one stack is too little to simulate a queue, with two stacks, the simulation is quite efficient. A deque automata can be efficiently simulated by a three stack automata. The converse question, simulating a (one) stack machine by a queue machine is not satisfactorily resolved.

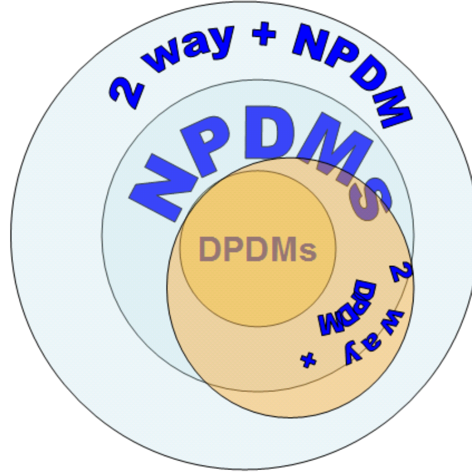

Figure 3.14: High level overview of the power of different PDMs.

3.6.3 Example 3

Can we design a PDM for the language $L = \{\overline{ww^R} | w \in (0 + 1)^*\}$ - the complement of even length palindromes. We saw that CFLs are not closed under complementation. We cannot just toggle the final and non-final states of the PDM in Figure 3.13. Why is that so? If it were a deterministic machine, we could do so. In general, deterministic machines (including deterministic PDMs) are closed under complement. Thus, we if take the complement of the states of a DPDM for a CFL, you get the DPDM of the complement of the CFL. But this does not hold for the NPDM for a CFL. Recall that the trick of toggling states did not work for a NFA either.

Nevetheless, this discussion does not mean that $L = \{\overline{ww^R} | w \in (0+1)^*\}$ has no PDM - it only means that we are not automatically in the ballpark. Recall that context free languages are closed under union. On the other hand, the language defined by deterministic PDMs (DPDMs) is not closed under union; to define union, you need a language with a machine equivalent that supports non-determinism. Here is a classic case $L^+ = \{0^n 1^n | n \geq 0\} \cup \{0^n 1^{2n} | n \geq 0\}$. L^+ can be accepted only by an NPDM[13], whereas both $\{0^n 1^n | n \geq 0\}$ and $\{0^n 1^{2n} | n \geq 0\}$ can be accepted by DPDMs. And this is a counter-example to the hypothesis that the subset of CFL defined by DPDMs is closed under union.

To start, all odd length strings are elements of L. To detect odd length strings, we do not even require the stack and can make a non-deterministic choice to the upper half of the machine. In the lower half of the machine, we look for mismatches, and transition to the accept state the moment we find one. However, we cannot transition to the accept state until we confirm that

[13]Exercise.

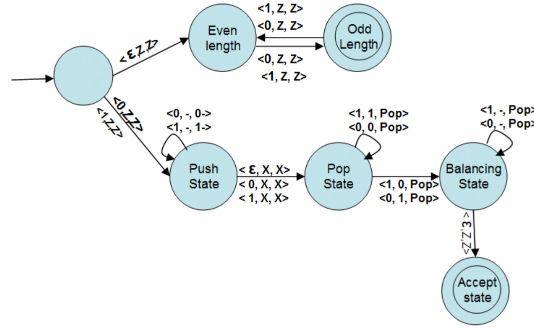

Figure 3.15: PDM that accepts the complement of the set of even length palindromes.

the mismatch was legitimate or in other words, that the mismatch was found exactly after moving to the 'Pop state' half way down the string. This can be done by transitioning to a 'balancing state' before moving to the accept state and looping around in the 'balancing state' till the end, making sure that the number of incoming symbols equal the number of symbols on the stack. Figure 3.15 shows the PDM for this problem.

This is not the only way of doing this problem; we will solve this problem in a different but powerful way, that will extend the repertoire of techniques we know for constructing PDMs. This alternative will make more efficient use of non-determinism. The odd length handling (upper) part of Figure 3.15 will remain the same. To determine a palindrome-mismatch in the string, all we need to find out is a symbol β_1 on the left half of the string that does not match a symbol β_2 in the right half of the string. And we note that the number of symbols to the right of β_2 should equal the number of symbols to the left of β_1. We will count the symbols (in the 'Seek Beta_1' state) by pushing a count symbol X on the stack, corresponding to every input symbol on the tape, as we go from left to right. And at some point we are non-deterministically going to guess a symbol that is β_1. We will remember β_1 on the top of the stack and move to a 'Seek Beta_2' state. And then we will move across the tape, ignoring everything we see until we non-deterministically guess in the 'Seek Beta_2' state that we got a match with β_2. If $\beta_2 \neq \beta_1$ (where β_1 is on the top of the stack), then we verify that we made the right guess by checking if the number of slots after β_2 is the same as the number of symbols on the stack (besides) β_1. This avoids many of the inner loops we had in Figure 3.15. The new equivalent PDM is shown in Figure 3.16.

Note that while we make the guesses non-deterministically, a machine implementation will require deterministic guessing, which will take us to the world of NP-complete problems later in the sequel.

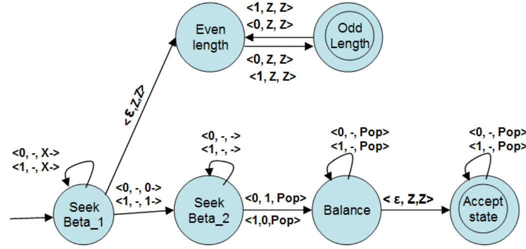

Figure 3.16: An alternate PDM, richer in non-determinism, that accepts the complement of the set of even length palindromes.

3.6.4 Simulating a stack with a queue

You can simulate a stack with queue, by keeping a pointer to the last 'pushed' symbol. And when you need to 'pop' it out, you deque all symbols appearing before the pointer and enqueue them from the back. A single push and pop is a linear operation with the stack, whereas the push or pop when simulated through a queue will require time proportional to number of symbols currently in the queue. You can simulate a stack using a queue, but you have to pay for it in terms of time. Can a queue simulate two stacks? Yes, it can even simulate two stacks.

3.6.5 Example 4

This example will be a really hard problem. It is kind of a puzzle. We will sketch the machine for this, without going into the details. We discussed that the language $L = \{ww | w \in (0+1)^*\}$ is not context free, that is, it has no PDM implementation. We did not prove it though we will prove it soon using the pumping lemma (you can do it with a queue FSM though).

What is really cool is that you can build a PDM for the complement of $L = \{ww | w \in (0+1)^*\}$, *i.e.*, L^C. But we cannot build the PDM for L^C building on a PDM for L, because L is not context free! We need a completely different way for building the PDM for L.

Mimicking the second solution for example 5, we can make guesses for the two halves of the string. We require that the if the string is σ and has length n and σ_i is the ith symbol in the string, then some property should be satisfied by σ_k and $\sigma_{\frac{n}{2}+k}$. We will push symbols on the stack, as before, remembering how many symbols it took to get at any particular position. And then we randomly move along, guessing where the middle is. Subsequently, we will start popping symbols from the middle. But how do we count till the middle and balance it using the symbols appearing after the middle? In fact, fundamentally, there is no way to do the matching for L. Because, *they are not nested!*.

An important thing to observe is that the length of the string between σ_k and $\sigma_{\frac{n}{2}+k}$ is equal to the sum of the length of the strings between σ_1 and σ_k

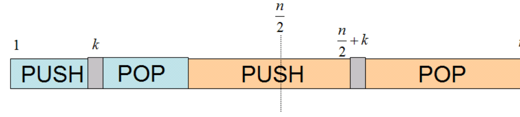

Figure 3.17: The higher level logic for example 4.

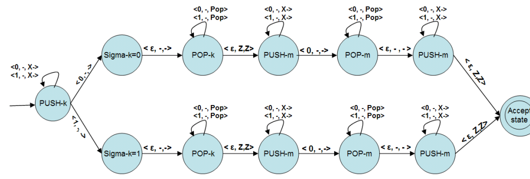

Figure 3.18: The PDM for the example 4.

and between $\sigma_{\frac{n}{2}+k}$ and σ_n. So we make our first guess at σ_k and the next guess at $\sigma_{\frac{n}{2}+k}$. We start pushing some k symbols (for some arbitrary k) into the stack starting at σ_1 and pop them out, starting at σ_k, till the stack is emptied. We keep track of the symbol σ_k using a state. And then we immediatly start pushing some m symbols (for some arbitrary m) into the stack and pop them out starting at a symbol $\sigma_{\frac{n}{2}+k} \neq \sigma_k$, for the next m symbols till the end of the tape is met and the stack is empty. Thus, $2k+2m+2 = n$. Also, $\frac{n}{2}+k = m+2k+1$. If the end of the tape and empty stack are encountered at the end simultaneously, then the string is accepted.

Figure 3.17 depicts the high level logic of the machine, whereas Figure 3.18 depicts the exact PDM for this problem.

3.7 Conversion from CFG to PDM

Deterministic and non-deterministic PDMs are not equivalent. The only equivalence you can show here is that CFGs are same as non-deterministic PDMs. The expressibility of CFGs and PDMs is the same. If you have a NDPDM, you can come with an equivalent CFG and vice verca. The CFG to PDM conversion is very logical and relates to parsing using Yacc. The conversion is interesting not only from the theoretical point of view but also from the application viewpoint. The other direction, from PDMs to CFGs is much more rusty and can be read up in some standard text book.

While the CFG to PDM conversion is logical, to make it easier to describe, we will use the CNF. This is one of the first of three main ideas that make use of the CNF.

Consider a typical simple CFG in CNF:

$$\begin{array}{lllllll} A & \rightarrow & B\ C & | & A\ B & | & 1 \\ B & \rightarrow & A\ A & | & 0 & & \\ C & \rightarrow & C\ B & | & 1 & | & 0 \end{array} \tag{3.26}$$

We will do an exercise that will help us understand the equivalence. We will turn the above grammar into a machine That will accept only the strings generated by this grammar, and no other strings - a machine equivalent to this grammar. We can understand how to do this by understanding how to generate a string from the grammar; to do what the Yacc tool does - take as input a string and a grammar and determine whether the grammar generates the string. This grammar is not deterministic and the machine that will result from the conversion is a non-deterministic pushdown machine. The subset of CFGs called LRK grammars are equivalent to deterministic pushdown machines.

Consider the string 10110 and a left-most derivation using the CNF.

$$A \rightarrow AB \rightarrow 1B \rightarrow 1\,AA \rightarrow 1BCA \rightarrow 10CA \rightarrow 101A \rightarrow 101AB \rightarrow 1011B \rightarrow 10110$$

One nice thing about left-most derivations from CNFs is that the terminals always sit on the left-most half while the right half contains non-terminals (if at all).

There is another left-most derivation for the same string (illustrating that the grammar is ambiguous):

$$A \rightarrow BC \rightarrow AAC \rightarrow 1\,ABAC \rightarrow 1BAC \rightarrow 10AC \rightarrow 101C \rightarrow 101CB \rightarrow 1011B \rightarrow 10110$$

As discussed earlier, the number of steps for deriving a string of length n using CNF is $2n-1$. We will derive a NDPDM that has the power to do either of the above left-most derivations. The mechanism is quite straightforward.

1. Everytime the grammar generates a symbol, the machine will read the symbol. These suite of actions are called the *popping actions*.

2. The non-terminals that appear in the derivation, waiting to be looked at and substituted for, will be stored by the machine on the stack. The non-terminal are pushed into the stack from the right to the left. The left-most non-terminal (the last thing to be pushed into the stack) is the first to be popped out of the stack and substituted for. The correponding actions are called the set of *processing actions*.

To start, we read no symbol off the tape and have an empty stack. A is the start state and gets pushed into the stack on the ϵ string. We will give the machine the choice of pushing A off the stack and popping in two symbols corresponding to any of the productions from A, as given by the CNF. We do all this without reading any symbol on the tape. This takes us into the set of *processing states*. The same procedure is followed for any non-terminal on top of the stack - the symbol on top of the stack is popped and the non-terminals

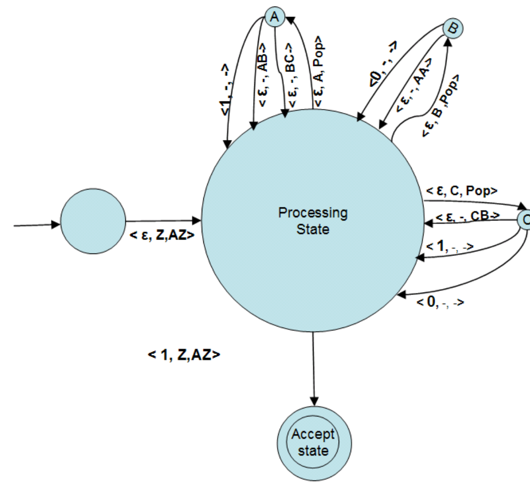

Figure 3.19: PDM for the grammar in 3.26

on the right hand side of its productions are pushed (in right to left order) into the stack.

To turn the non-terminals on the stack into non-terminals, we invoke the productions of the form $< terminal > \rightarrow < non - terminal >$; for each such production, corresponding to an input symbol $< terminal >$ on the tape, we pop the $< non - terminal >$ from the top of the stack. We move to the accept state when we reach the end of the input tape and the stack is empty. Figure 3.19 shows the PDM for the CFG under consideration in Table 3.26.