Maggie Johnson Handout #28
CS109

Finite Automata

Key Topics

* Some Regular Expression Practice
* Introduction
* Finite Automata

Some Regular Expression Practice

Define a regular expression for {∂, aa, ba, aaaa, aaba, baba, ...} ____________________________
*Give a set of characteristics for strings generated by a(ba + a)*b* ____________________________
____
What is the shortest string of a's and b's not in the language corresponding to the regular expression b(abb*)*a*?* ____
*Describe as simply as possible the language defined by a*b(a*ba*b)*a** ______________________
(b + ab)(a + ab)** ____

Introduction

Several children's games fit the following description: Pieces are set up on a playing board; dice are thrown (or a wheel is spun) and a number is generated at random. Based on the generated number, the pieces on the board are rearranged specified by the rules of the game. Then, another child throws or spins and rearranges the pieces again. There is no skill or choice involved - the entire game is based on the values of the random numbers.

Consider all possible positions of the pieces on the board and call them **states**. We begin with the **initial state** of the starting positions of the pieces on the board. The game then changes from one state to another based on the value of the random number. For each possible number, there is one and only one resulting state given the input of the number, and the prior state. This continues until one player wins and the game is over. This is called a **final state**.

Now consider a very simple computer with an input device, a processor, some memory and an output device. We want to calculate 3 + 4, so we write a simple list of instructions and feed them into the machine one at a time (e.g., STORE 3 TO X; STORE 4 TO Y; LOAD X; ADD Y; WRITE TO OUTPUT). Each instruction is executed as it is read. If all goes well, the machine outputs '7' and terminates execution. This process is similar to the board game. The state of the machine changes after each instruction is executed, and each state is completely determined by the prior state and the input instruction (thus this machine is defined as **deterministic**). No choice or skill is involved; no knowledge of the state of the machine 4 instructions ago is needed. The machine simply starts at an initial state, changes from state to state based on the instruction and the prior state, and reaches the final state of writing '7'.

Finite Automata

A general model (of which the previous two examples are instances) of this type of machine is called a **Finite Automaton**; 'finite' because the number of states and the alphabet of input symbols is finite; automaton because the structure or machine (as it is more commonly called) is deterministic, i.e., the change of state is completely governed by the input.

A **finite automata** has:

1) A finite set of states, one of which is designated the initial state or **start state**, and some (maybe none) of which are designated as **final states.**

2) An alphabet ∑ of possible input symbols.

3) A finite set of **transitions** that tell for *each* state and for *each* symbol of the input alphabet, which state to go to next.

In the simple computer example, the start state is the original state of the machine before program execution and the final state is '7' written on the output device. The input alphabet is the set of strings representing the instructions.

Example 1

Suppose ∑ = {a,b}, the set of states = {x, y, z} with x the start state and z the final state, and we have the following rules of transition:

1) from state x and with input a, goto state y.
2) from state x and with input b, goto state z.
3) from state y and with input a, goto state x.
4) from state y and with input b, goto state z.
5) from state z and with any input, stay at state z.

This is a perfectly defined 'FA' (short for finite automaton) according to the definition above. We now need to examine what happens to various input strings when presented to this FA. Consider 'aaa': we begin in state x and goto state y. The next symbol is also an 'a', so from state y we go to state x. Then on the last 'a', we go from state x back to state y. That's all the input we have - since we do not end up in state z, the final state, we have an unsuccessful end.

This exercise illustrates how an FA can be used to *recognize or accept* strings of a particular language. The set of all strings that leave us in a final state of an FA is called **the language accepted or defined by the FA**. 'aaa' is not a word in the language defined by the FA given above. This is why FA's are also called **language recognizers.**

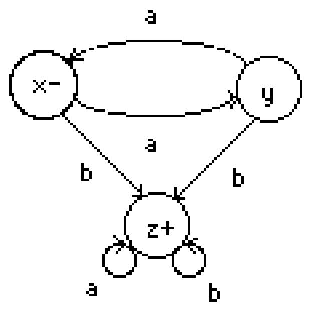
Now, try 'abba': from state x to state y, from state y to state z, from state z to state z, from state z to state z. This word is a part of the language defined by this FA. In fact, any input string that has just one'b' (or several) will be accepted by this FA. We can define this by the regular expression:

$$(\mathbf{a} + \mathbf{b})^* \, \mathbf{b} \, (\mathbf{a} + \mathbf{b})^*$$

This is a very simple FA. Typically, the list of transition rules can be quite long, so an alternative representation is frequently used. One method is to use a **transition table**:

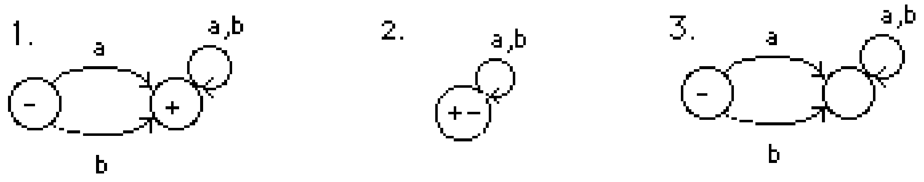
		a	b
start	x:	y	z
	y:	x	z
final	z:	z	z

This table has all the information required to define an FA. Even though it is no more than a table of symbols or a list of rules, we consider an FA to be a machine, i.e., we understand that an FA has dynamic capabilities. It moves. It processes input. Something goes from state to state as the input is read. One way to represent an FA that feels more like a machine is a **transition diagram:**



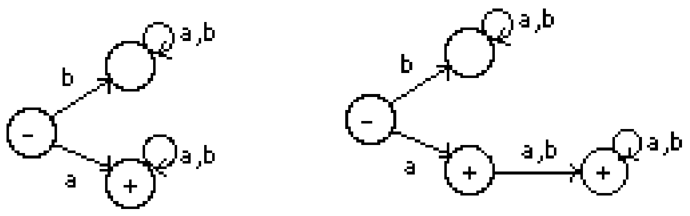
The vertices are the states, and the edges indicate the input into each state. A '-' is a start state, and a '+' is a final state. This representation makes it much easier to see that this FA accepts only strings with at least one 'b' in them. Notice also that we do not have to name the states when we represent an FA using a transition diagram.

What language over ∑ = {a,b} do the following FA's accept? ______________________________



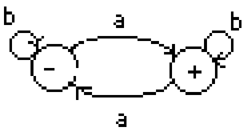
(1) appears to accept any combination of a's and b's. The first letter of the input takes us over to the final state, and once there we are trapped forever. When the input string runs out, there we are in the correct final state. This ignores the possibility of the input being ∂, which would leave us in the start state. This is true of any FA: ∂ will always leave us in the start state. (2) handles this problem by having the start and final states the same; this FA accepts any combination of a's and b's, but the first accepts the language represented by the following regular expression: (**a** + **b**)(**a** + **b**)* or (**a** + **b**)$^+$. What about (3)? This FA accepts no language since there is no final state.

Suppose we want to build an FA that accepts all the words in a particular language, like: **a** (**a** + **b**)*. The first step is to analyze the regular expression to see what words are in this language. Listing some of the words is a good start; then look for a pattern (a, aa, ab, aaa, aba, abb, ...) Every word in this language must start with 'a', and then we can have any combination of a's or b's (maybe none). This is what will take us to a final state. There are other considerations, however, as the definition of an FA says: A finite set of **transitions** that tell for *each* state and for *each* symbol of the input alphabet, which state to go to next. If an input string begins with 'b', the FA must handle it as well, although we won't want to end up in a final state. Two possible FA's that accept this language are given below:



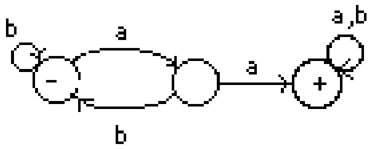
Notice from this example that there is more than one unique machine for a given language. Also, notice that a given FA can have more than one final state.

Draw an FA for the following regular expression: **b***a** (**b*** **ab*** **ab***)*b***. List some of the words of this language: a, aaa, babab, bba, abbbabbbbbbabbbbabab.... This regular expression defines words with an odd number of a's. We either have one 'a', or we add two more a's for each concatenation. The FA for this regular expression will have us move to a final state for 1 'a', to an "unfinal" state for 2 a's, back to the final state for 3 a's, etc. The following FA does the trick:



It's not always easy to come up with an FA for a given regular expression. We will see later that there is an algorithm for creating an FA from any regular expression, and vice versa.

What regular expression is represented by the following FA? ______________________________



Define an FA that accepts the language of all strings that end in b and does not contain the substring aa. What is a regular expression for this language? ____

Define an FA that accepts the following language: (aa)(bb)*.*

Notice that we have been informally showing the equivalence of the set of languages defined by regular expressions, and the set of languages defined by FA's. Later, we will formally prove this.

Bibliography

* The classic texts:

M. Davis, E. Weyuker, *Computability, Complexity and Languages: Fundamentals of Theoretical Computer Science*, New York: Academic Press, 1983.

J. Hopcroft, J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading, MA: Addison-Wesley, 1979.

H. Lewis, C. Papadimitriou, *Elements of the Theory of Computation*, Englewood Cliffs, NJ: Prentice Hall, 1981.

M. Minsky, *Computation: Finite and Infinite Machines*, Englewood Cliffs, NJ: Prentice Hall, 1967.

* The "more readable" references in the previous handout on "Languages and Regular Expressions" apply here too.

* Of historical interest:

W. McCulloch, W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, 5 (1943), 115-133.

G.H. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System Technical Journal* 34, 5 (1955), 1045-1079.

E.F. Moore, "Gedanken-Experiments on Sequential Machines," in C. Shannon and J. McCarthy (eds), *Automata Studies*, Princeton, NJ: Princeton University Press, 1956.

M.O. Rabin, D. Scott, "Finite Automata and Their Decision Problems," *IBM Journal of Research and Development*, 3 (1959), 114-125.

Historical Notes

The idea of a finite automata first appeared in McCulloch and Pitts as a way of modeling neural nets. Properties of FA's are developed more extensively in Mealy and Moore. A classic paper on FA's is Rabin and Scott.