Theory of Automata
Finite Automata

Hafiz Tayyeb Javed

[Section B (Week-04-Lecture-02)]
and
[Section A (Week-04-Lecture-01)]

Contents

- FAs & their languages
- Discrete Finite Automata

FA and their Languages

- We will study FA from two different angles:

1. Given a language, can we build a machine for it?

2. Given a machine, can we deduce its language?

Example

- Let us build a machine that accepts the language of all words over the alphabet Σ = {a, b} with an **even number of letters**.

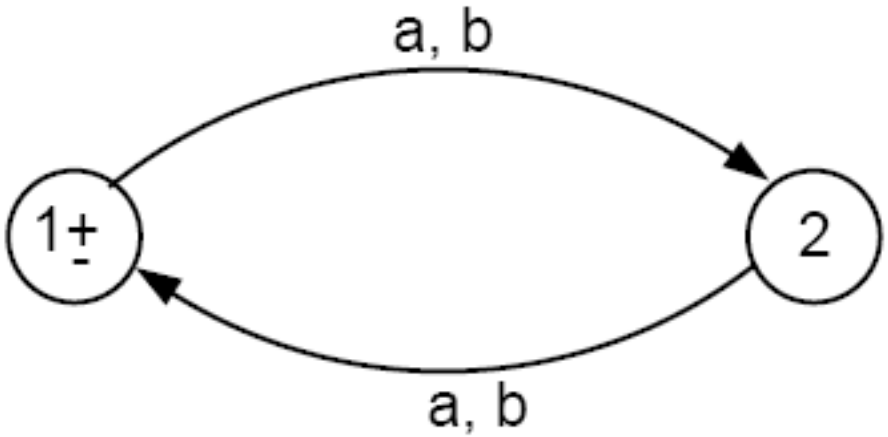
- A mathematician could approach this problem by counting the total number of letters from left to right. A computer scientist would solve the problem differently since it is not necessary to do all the counting:

- Use a Boolean flag, named E, initialized with the value TRUE. Every time we read a letter, we reverse the value of E until we have exhausted the input string. We then check the value of E. If it is TRUE, then the input string is in the language; if FALSE, it is not.

- The FA for this language should require only 2 states:
  - State 1: E is TRUE. This is the start and also final state.
  - State 2: E is FALSE.

Example Contd.

- So the FA is pictured as follows:



Example

- Let us build a FA that accepts all the words in the language

  $a(a + b)^*$

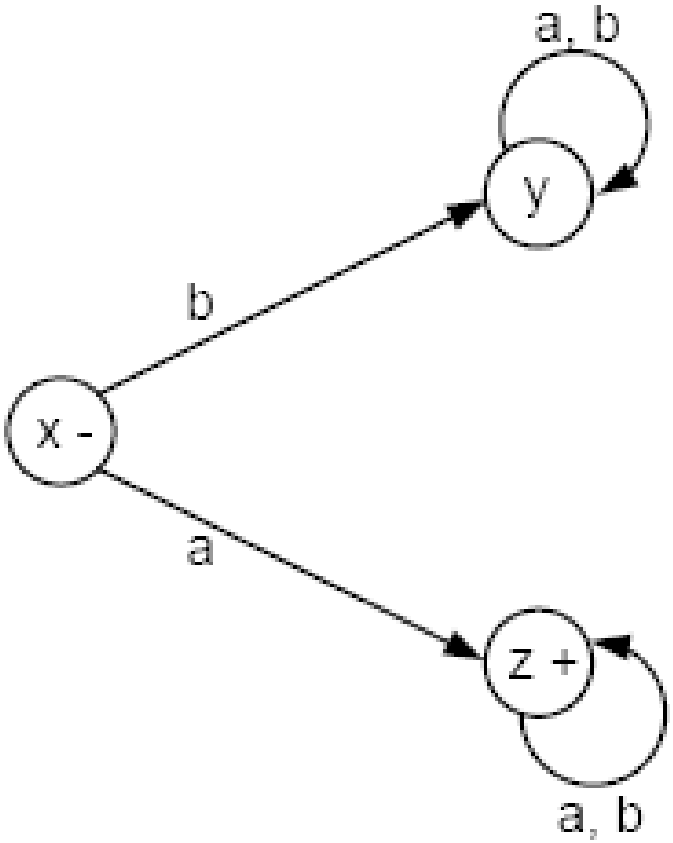
- This is the language of all strings that begin with the letter a.

- Starting at state x, if we read a letter b, we go to a **dead-end** state y. *A dead-end state is one that no string can leave once it has entered.*

- If the first letter we read is an a, then we go to the dead-end state z, which is also a final state.

Example

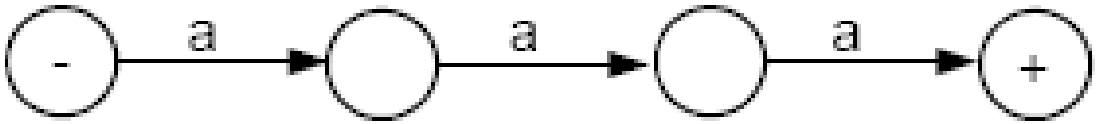
- The machine looks like this:



Example

- Let's build a machine that accepts all words **containing a triple letter**, either *aaa* or *bbb*, and only those words.

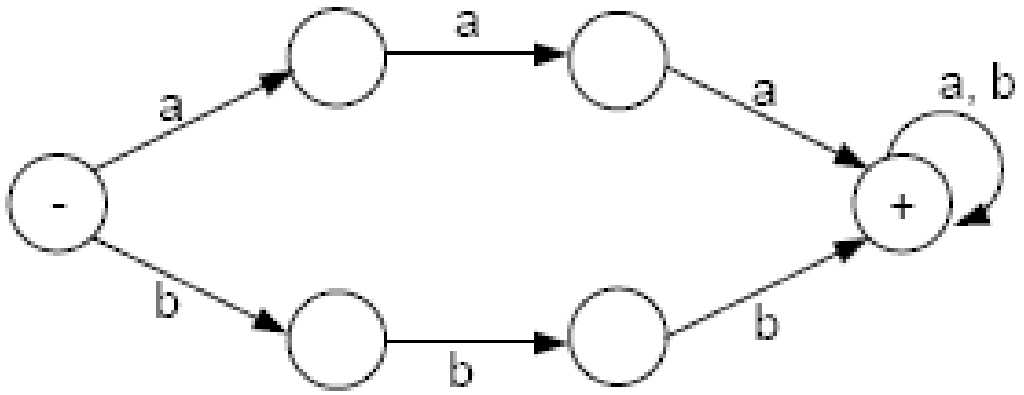
- From the start state, the FA must have a path of three edges, with no loop, to accept the word *aaa*. So, we begin our FA with the following:



Example Contd.

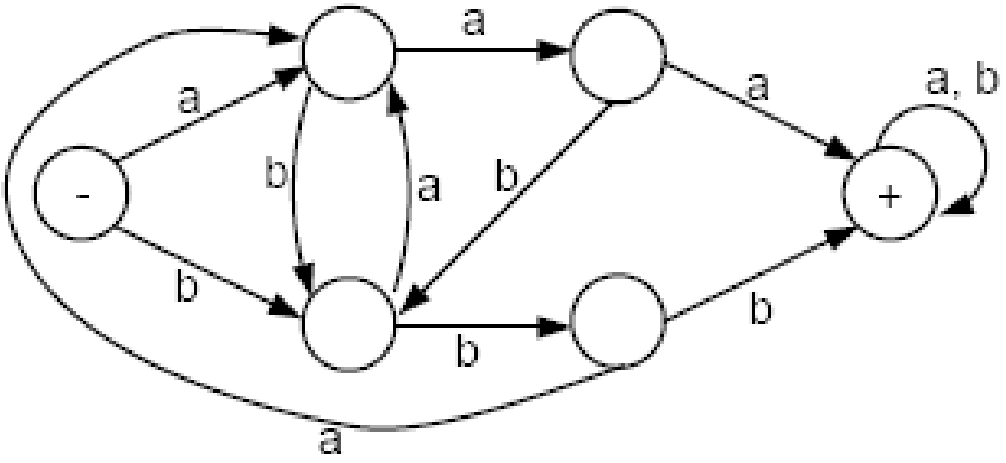
- For similar reason, there must be a path for bbb, that has no loop, and uses entirely differently states. If the b-path shares any states with the a-path, we could mix a's and b's to get to the final state. However, the final state can be shared.

- So, our FA should now look like:



Example Contd.

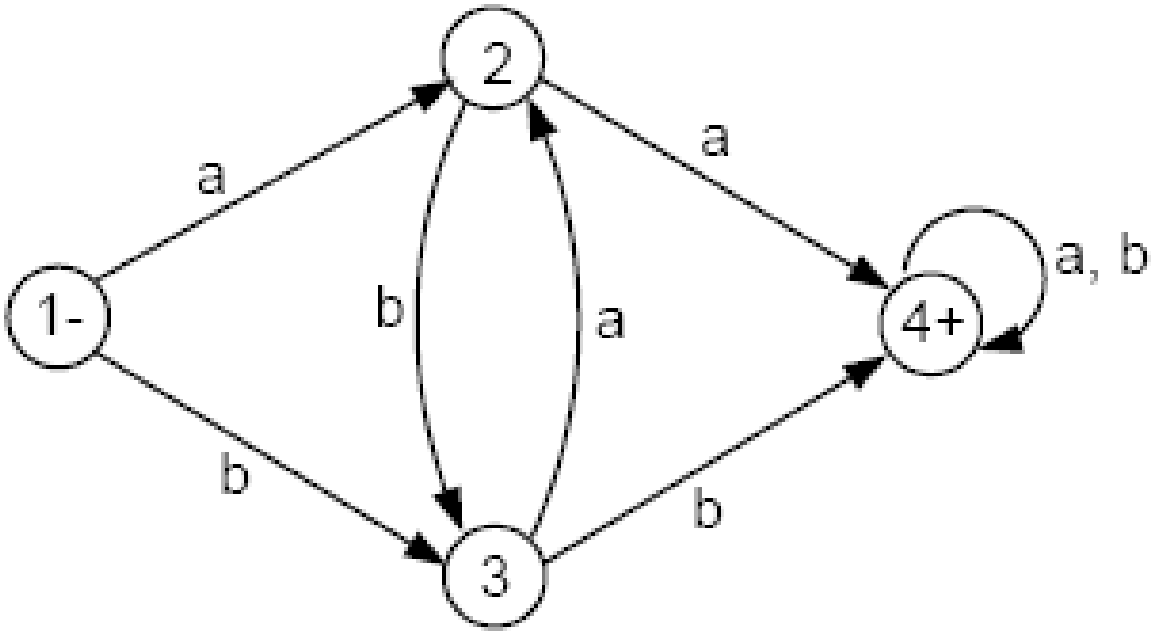
- If we are moving along the a-path and we read a *b* before the third a, we need to jump to the *b*-path in progress and vice versa. The final FA then looks like this:



How Hard is to Understand FA

Example

- Consider the FA below. We would like to examine what language this machine accepts.



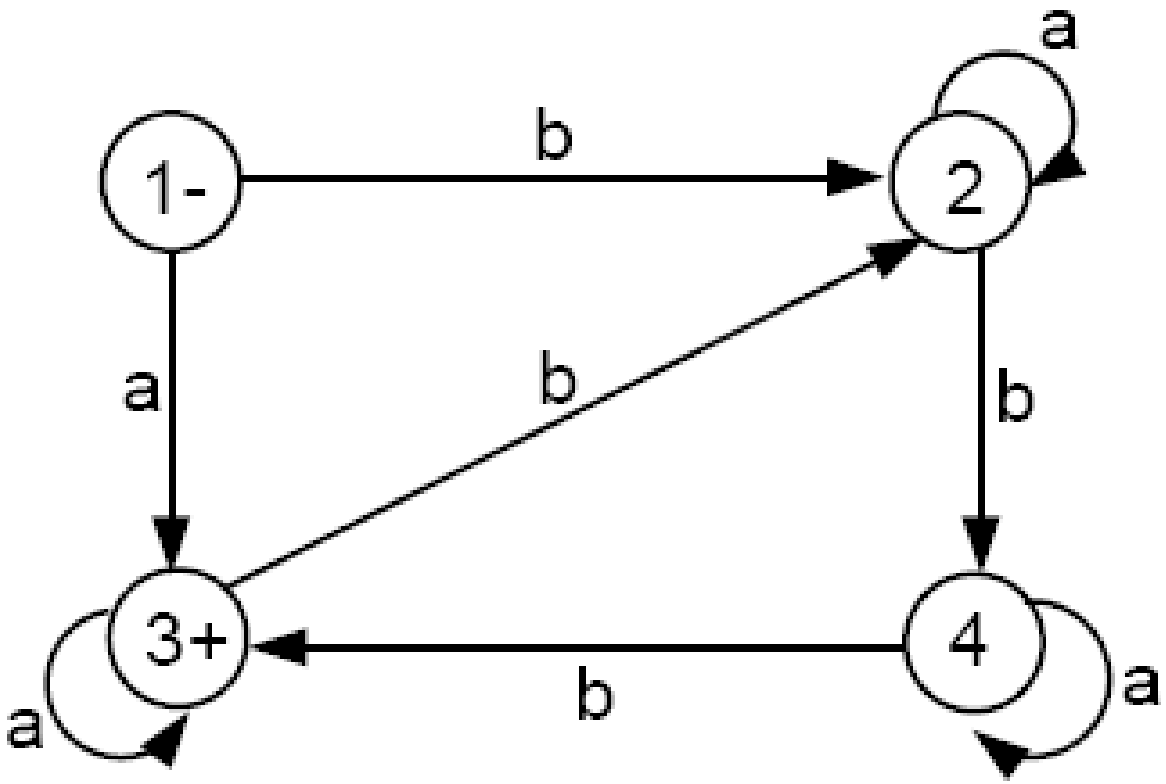
Example

- There are only two ways to get to the final state 4 in this FA: One is from state 2 and the other is from state 3.
- The only way to get to state 2 is by reading an **a** while in either state 1 or state 3. If we read another **a** we will go to the final state 4.
- Similarly, to get to state 3, we need to read the input letter **b** while in either state 1 or state 2. Once in state 3, if we read another **b**, we will go to the final state 4.
- Thus, the words accepted by this machine are exactly those strings that have a double letter *aa* or *bb* in them. This language is defined by the regular expression

$$(a + b)^*(aa + bb)(a + b)^*$$

Example

- Consider the FA below. What is the language accepted by this machine?



Example

- Starting at state 1, if we read a word beginning with an **a,** we will go straight to the final state 3. We will stay in state 3 as long as we continue to read only **a**'s. Hence, all words of the form **aa** are accepted by this FA.

- What if we began with some a's that take us to state 3 and then we read a b? This will bring us to state 2. To get back to the final state 3, we must proceed to state 4 and then state 3. This trip requires two more b's.

- Notice that in states 2, 3, and 4, all a's that are read are ignored; and only b's cause a change of state.

- Summarizing what we know: If an input string starts with an a followed by some b's, then it must have 3 b's to return to the final state 3, or 6 b's to make the trip twice, or 9 b's, or 12 b's and so on.

- In other words, an input string starting with an a and having a total number of b's **divisible by 3** will be accepted. If an input string starts with an a but has a total number of b's not divisible by 3, then it is rejected because its path will end at either state 2 or 4.

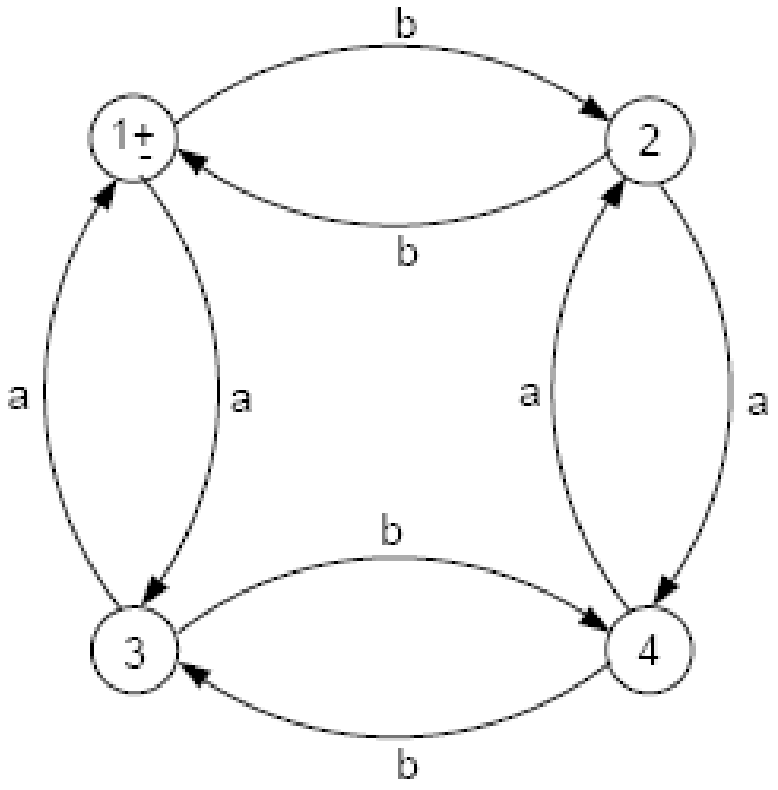
Example Contd.

- What happens to an input string that begins with a b?
- Such an input string will lead us to state 2. It then needs two more b's to get to the final state 3. These b's can be separated by any number of a's. Once in state 3, it needs no more b's, or 3 more b's, or 6 more b's and so on.
- All in all, an input string, whether starting with an **a** or a **b**, must have a total number of b's divisible by 3 to be accepted.
- The language accepted by this machine therefore can be defined by the regular expression

$$(a + ba^*ba^*b)^+ = (a + ba^*ba^*b)(a + ba^*ba^*b)^*$$

Example *EVEN-EVEN* revisited

- Consider the FA below.



Example *EVEN-EVEN* revisited Contd.

- There are 4 edges labeled a. All the a-edges go either from one of the upper two states (states 1 and 2) to one of the lower two states (states 3 and 4), or else form one of the lower two states to one of the upper two states.
- Thus, if we are north and we read an a, we go south. If we are south and we read an a, we go north.

- If a string gets accepted by this FA, we can say that the string must have had an even number of a's in it. Every a that took us south was balanced by some a that took us back north.
- So, every word in the language of this FA has an even number of a's in it. Also, we can say that every input string with an even number of a will finish its path in the north (ie., state 1 or state 2).

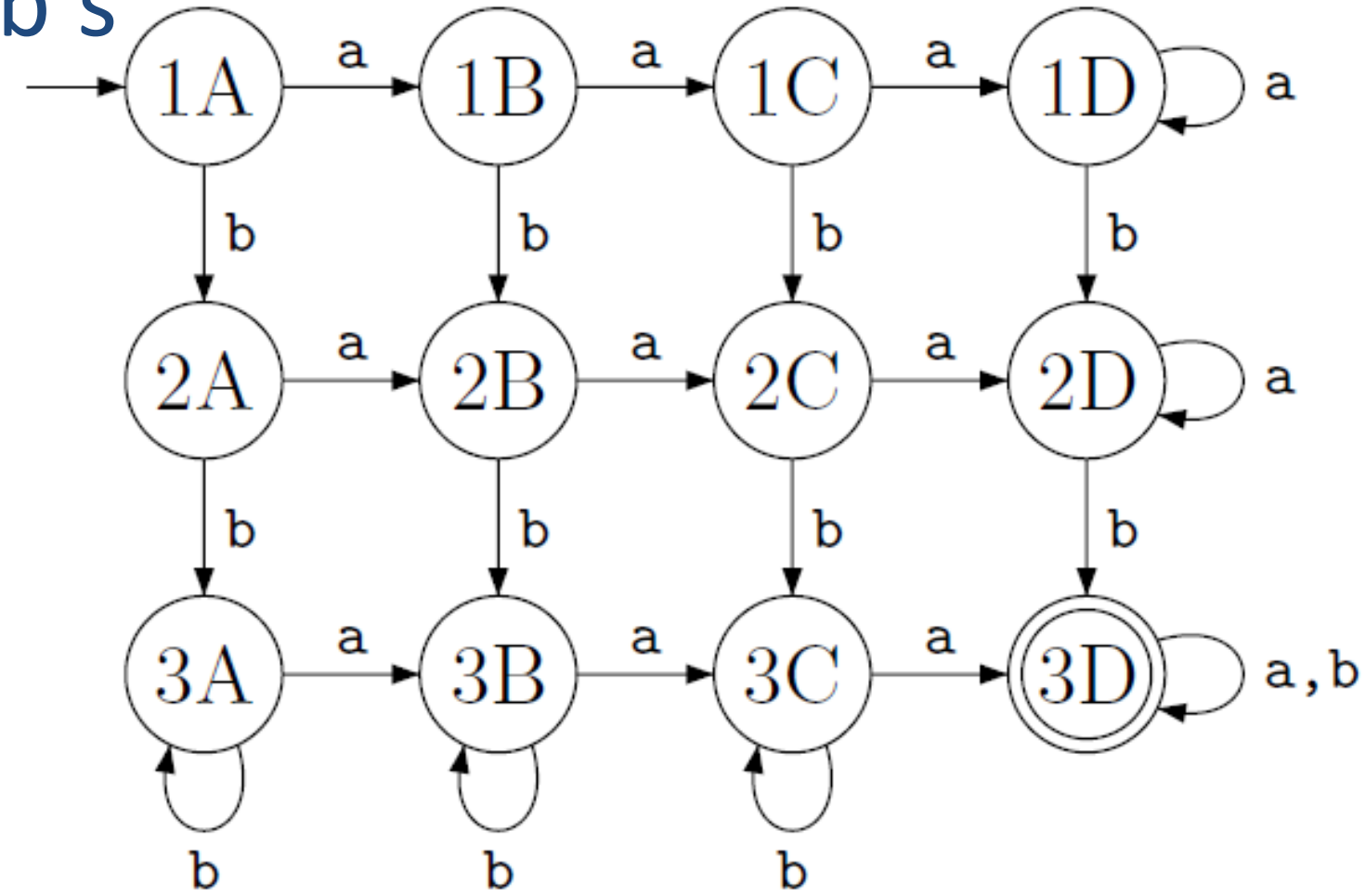
Example *EVEN-EVEN* revisited Contd.

- Therefore, all the words in the language accepted by this FA must have an even number of a's and an even number of b's. So, they are in the language *EVEN-EVEN.*

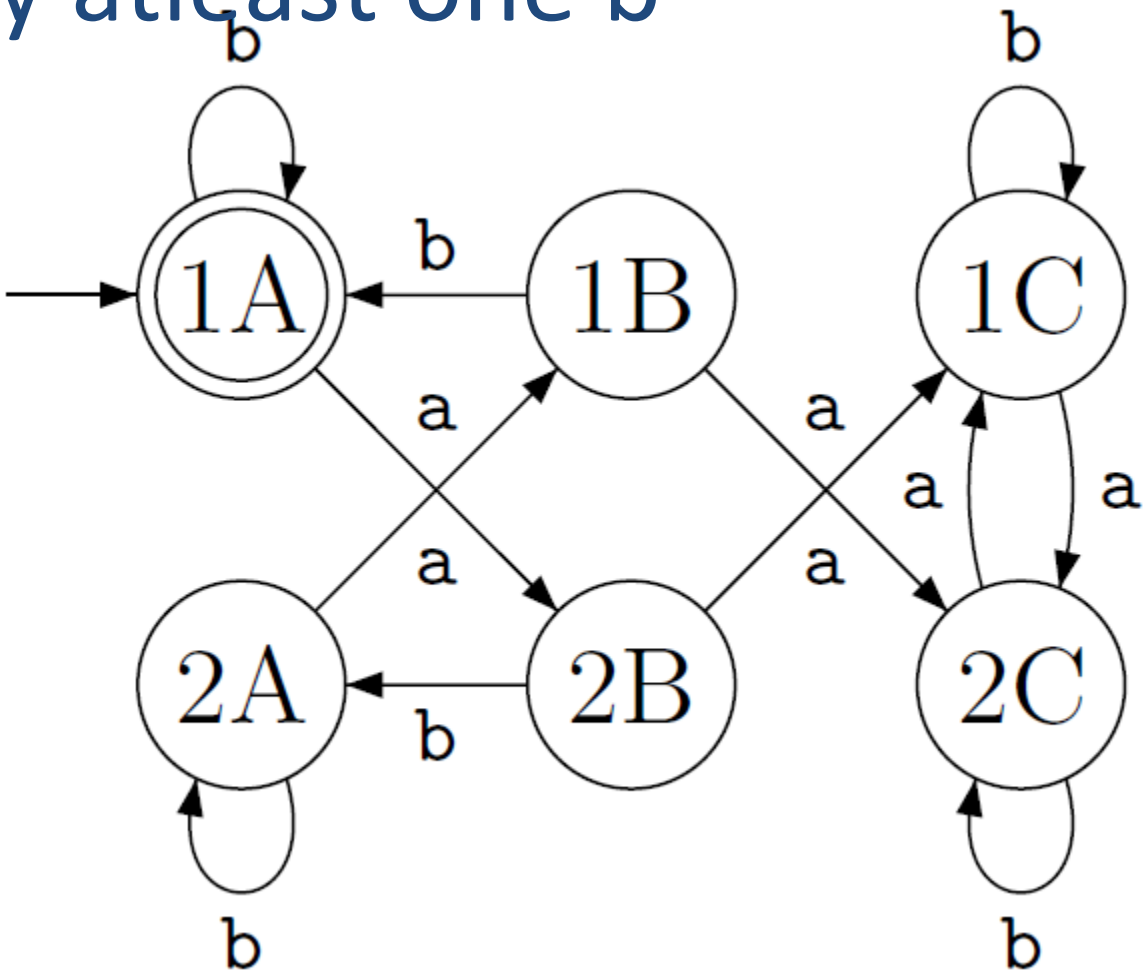
- Notice that all input strings that end in state 2 have an even number of a's but an odd number of b's. All strings that end in state 3 have an even number of b's but an odd number of a's. All strings that end in state 4 have an odd number of a's and an odd number of b's. Thus, every word in the language *EVEN - EVEN* must end in state 1 and therefore be accepted.

- Hence, the language accepted by this FA is *EVEN-EVEN.*

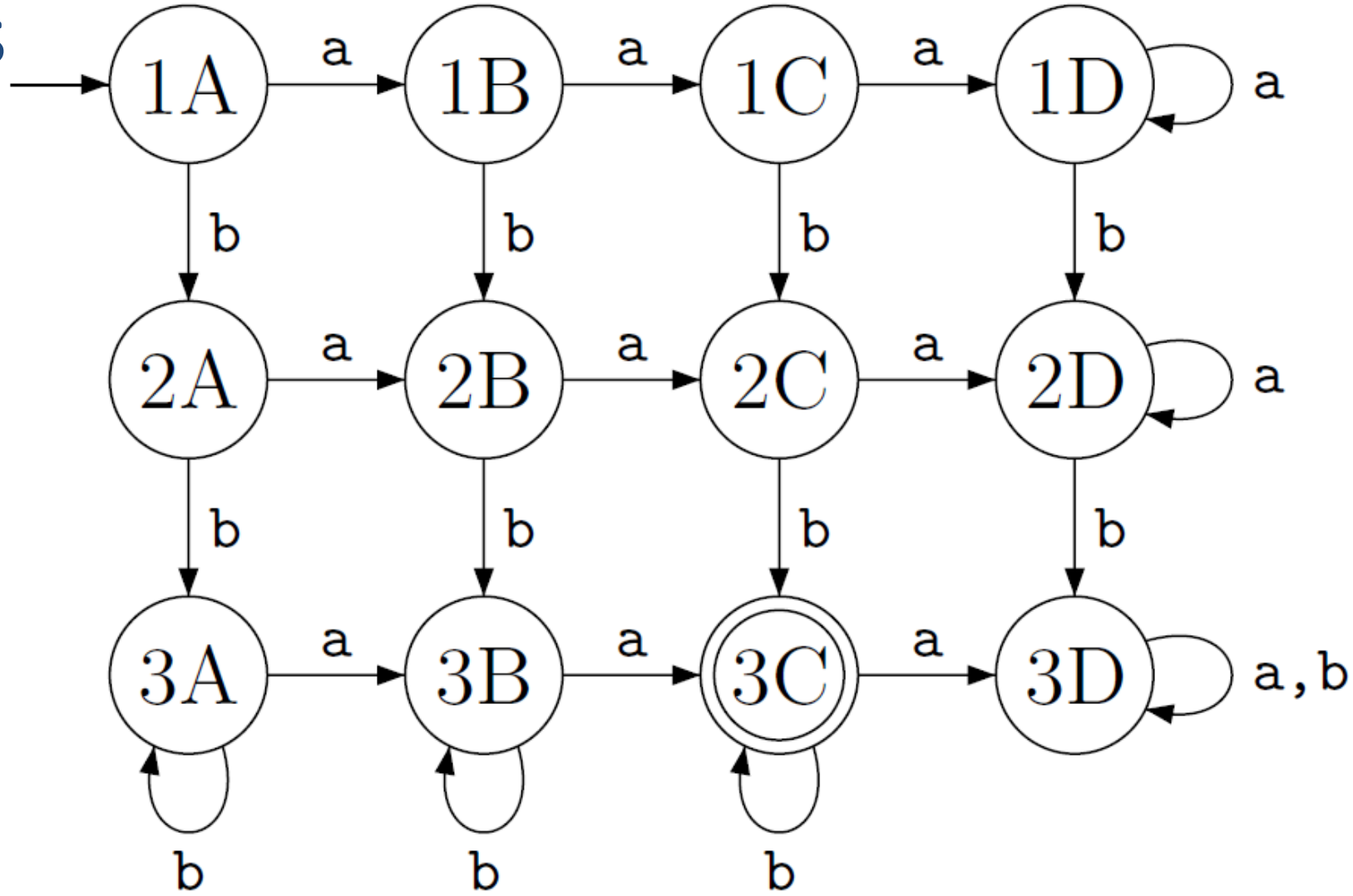
FA – at least three a's and at least two b's



FA – even number of a's and each a is followed by atleast one b



FA – exactly two a's and at least two b's



DFA

- DFA is the FA we have covered so far.

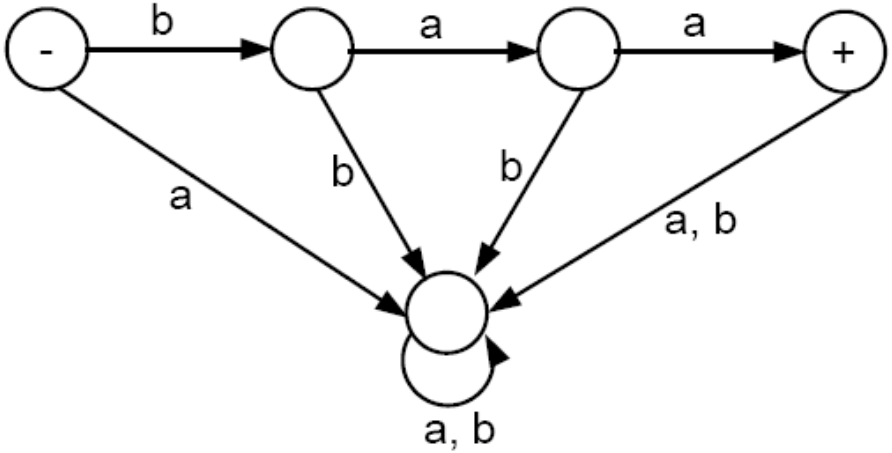
Contents

- Relaxing the Restriction on Inputs
- Looking at TGs

Relaxing the Restriction on Inputs

Relaxing the restrictions on inputs

- Let's consider a very specialized FA that accepts only the word baa:

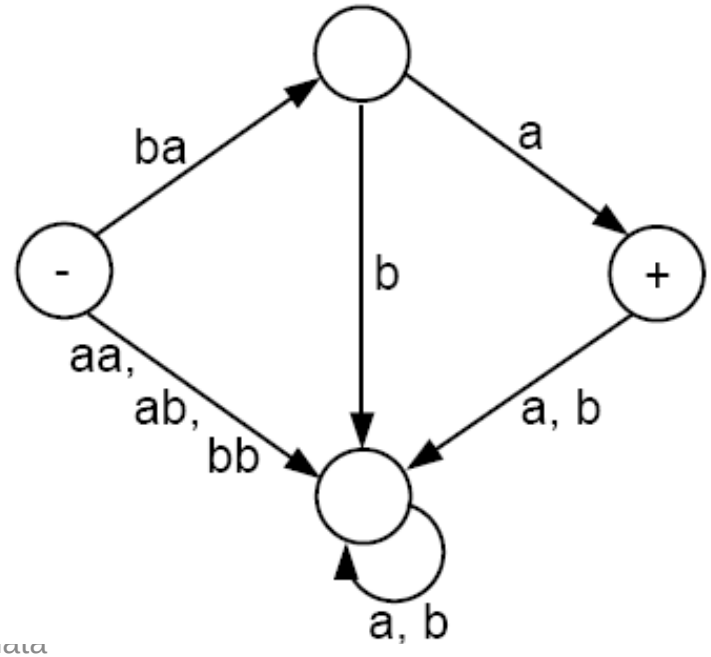


- Beginning at the start state, anything but the string *baa* will drop down into the garbage collecting state and never be seen again. Even the string *baabb* will fail.
- Hence the language accepted by this FA is L = {baa}

Relaxing restrictions on input

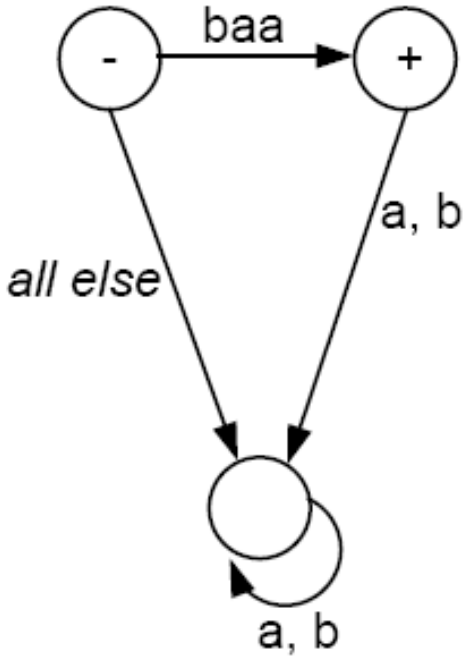
- Let us relax the restriction that *an FA can only read one letter from the input string at a time*

- Suppose we now allow a machine to read *either one or two letters* of the input string at a time. Then we may design a machine that accepts only the word *baa* with fewer states as the one below:

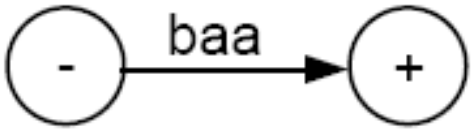


Relax restrictions on input

- If we go further to allow a machine to read *up to three letters* of the input string at a time, then we may design the machine accepting only the word baa with even fewer states as follows:



or even



Relaxing restrictions on input

- The picture on the last page tells us that when the input fails to be of the desired form, we must go to the garbage collection state and read through the rest of the input, knowing that we can never leave there.

- The picture on the last page introduces some problems:
  - If we begin in the start (-) state and the first letter of the input is an **a**, we have no direction as to what to do.

  - We also have problem even with the input baaa. The first three letters take us to the accept state, but then the picture does not tell us where to go when the last a is read. (*Remember that according to the rules of FAs, one cannot stop reading input letters until the input string completely runs out*.)

- We may assume, as a convention, that there is some garbage collection state associated with the picture on the right, but we do not draw it; and that we must go and stay there when the input string fails to be of the desired form.

- With this assumption, we can consider the two pictures above to be equivalent, in the sense that they accept the exact same language.

Relaxing restrictions on input

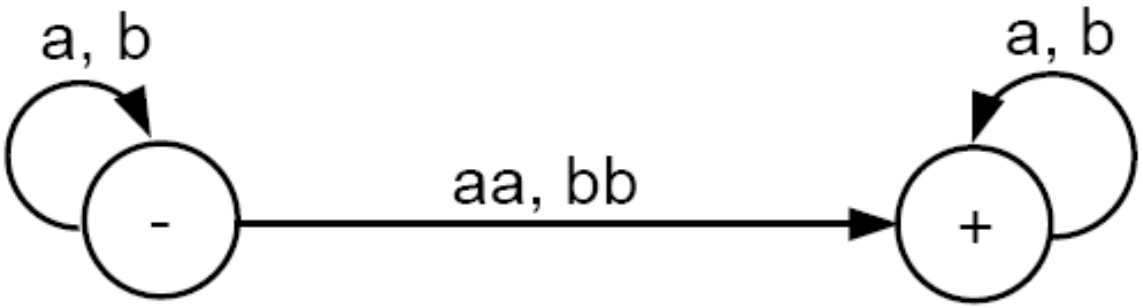
- Rather than trying to imagine a garbage collection state as described above, it is more standard to introduce a new term to describe what happens when an input is running on a machine and gets into a state from which it cannot escape, even though it has not yet been fully read.

- ***Definition*: When an input string that has NOT been completely read reaches a state (final or otherwise) that it cannot leave because there is no outgoing edge that it may follow, we say that the input (or the machine) *crashes* at that state. Execution then terminates and the input must be rejected.**

- *Note that on an FA it is not possible for any input to crash because there are always an outgoing a-edge and an outgoing b-edge from each state.* As long as there remains input letters unread, progress is possible.

- There are now two different ways that an input can be rejected: (i) It could peacefully trace a path ending in a non-final state, or (ii) it could crash while being processed.

- If we hypothesize that a machine can read *one or two* letters at a time, then one may build a machine using only two states that can recognize all words containing a double letter (aa or bb) as follows:



- We must now realize that we have changed something more fundamental than just the way the edges are labeled or the number of letters read at a time: *This machine makes us **exercise some choice,*** i.e., we must **decide how many letters to read** from the input string each time.

- As an example for the problems of making choices, let us say that the input is baa.
- If we first read b and then read aa we will go to the final state. Hence, the string is accepted.
- If we first read b, then read a, and then read a, we will loop back and be stuck at the start state. Hence, the string is rejected in this case.
- If we first read two letters ba at once, then there is no edge to tell us where to go. So, the machine crashes and the input string is rejected.
- What shall we say? Is this input string a word in the language of this machine or not?

- *The above problems tell us that if we change the definition of our machine to allow for more than one letter to be read at a time, we must also change the definition of acceptance.*

- ***We shall say that a string is accepted if there is some way it could be processed so as to arrive at a final state.***

- Due to many difficulties inherent in expanding our definition of machine to reading more than one letter of input at a time, we shall leave the definition of finite automaton alone and call these new machines ***transition graphs***.

- Transition graphs were invented by John Myhill in 1957 for reasons that will be revealed in the next chapter.

Definition of A Transition Graph

- A **transition graph**, abbreviated **TG**, is a collection of three things:

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

2. An alphabet ∑ of possible input letters from which input strings are formed.

3. *A finite set of transitions (edge labels) that show how to go from some states to some others, based on reading specified substrings of input letters (possibly even the null string **Λ**).*

Transition Graph Continued…