SE4033 Formal Methods for Software Engineering

Finite State Machine

FSM: Finite State Machine

- A formal process representation
- $M = (S, T, \sum \quad , s, A)$
- S= set of state
- T= Set of transitions
- $\sum$ set of transition labels
- s Starting state
- A accepting/ending states

3

Finite Automata

Another Method for Defining Languages

FAs and Their Languages

EVEN-EVEN

Definition

- A **finite automaton** is a collection of three things:
  - A finite set of states, **one** of which is designated as the initial state, called the **start state**, and **some** (maybe **none**) of which are designated as **final states**.
  - An **alphabet Σ** of possible input letters.
  - A finite set of **transitions** that tell for each state and for each letter of the input alphabet which state to go next.

How Does a Finite Automaton work?

- It works by being presented with an input string of letters that it reads letter by letter starting from the leftmost letter.
- Beginning at the start state, the letters determine a sequence of states.
- This sequence ends when the last input letter has been read
- We will use the term FA for the phrase “finite automaton”.

Example

Consider the following FA:

- The input alphabet has only the two letters a and b. (We usually use this alphabet throughout the chapter.)
- There are only three states, *x*, *y* and *z*, where *x* is the start state and *z* is the final state.
- The transition list for this FA is as follows:
  - Rule 1: From state *x* and input *a*, go to state y.
  - Rule 2: From state *x* and input *b*, go to state *z*.
  - Rule 3: From state *y* and input *a*, go to state *x*.
  - Rule 4: From state *y* and input *b*, go to state *z*.
  - Rule 5: From state *z* and any input, stay at state *z*.

Example Execution

- Let us examine what happens when the input string *aaa* is presented to this FA.
- First input a: state $x \rightarrow y$ by Rule 1.
- Second input *a*: state $y \rightarrow x$ by Rule 3.
- Third input *a*: state $x \rightarrow y$ by Rule 1.
- We did not finish up in the final state z, and therefore have an unsuccessful termination.

Example Analysis

- The set of all strings that lead to a final state is called the **language defined by the finite automaton**.
- Thus, the string *aaa* is **not in the language defined by this FA**.
- We may also say that the string aaa is **not accepted** by this FA, or the string *aaa* is **rejected** by this FA.
- The set of all strings accepted is also called the **language associated with the FA**.
- We also say, “This FA **accepts** the language L”, or “L is the
- **language accepted** by this FA”, or “L is the **language of** the FA”, by which we mean that all the words in L are accepted, and all the inputs accepted are words in L

Abstract definition of FA

A finite set of states $Q = \{q_0, q_1, q_2\ q_3\ \ldots\}$ of which q_0 is start state.

A subset of Q called final state (s).

An alphabet $\sum = \{ x_1, x_2, x_3, \ldots\}$.

A transition function δ associating each pair of state and letter with a state: $\delta(q,x_j) = x_k$

Transition Table

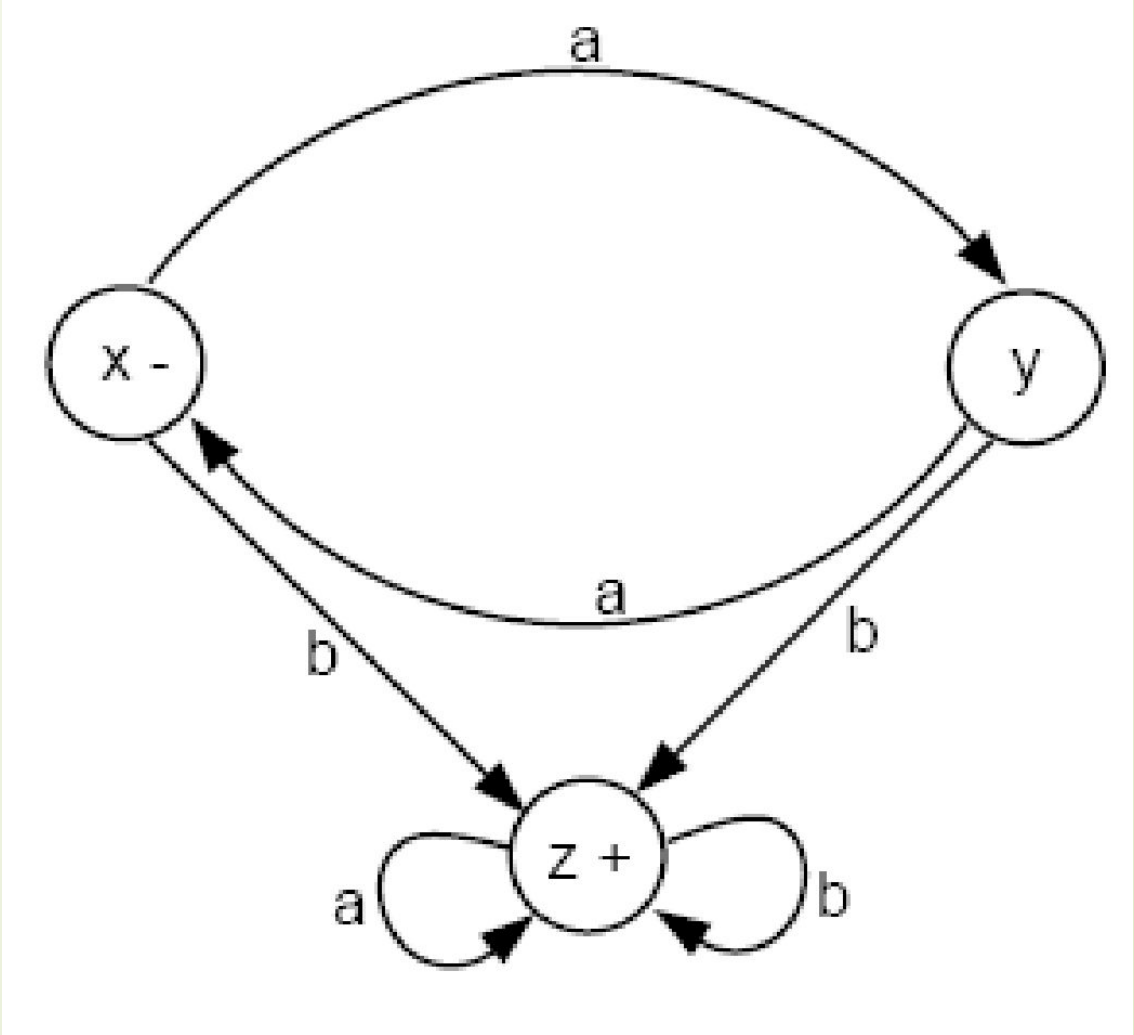
- The transition list can be summarized in a table format in which each row is the name of one of the states, and each column is a letter of the input alphabet.
- For example, the **transition table** for the FA above is

	a	b
Start x	y	z
y	x	z
Final z	z	z

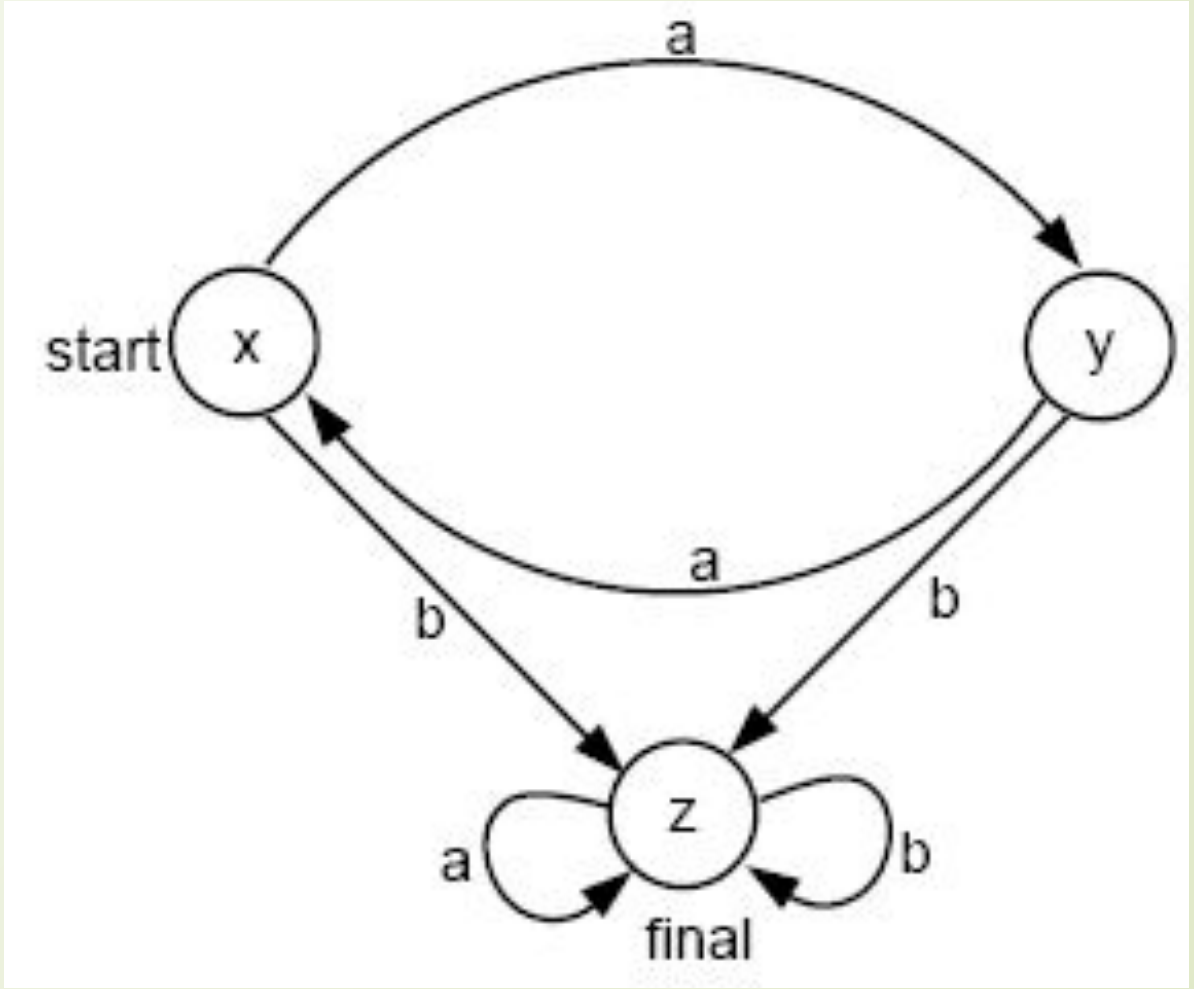
Transition Diagrams

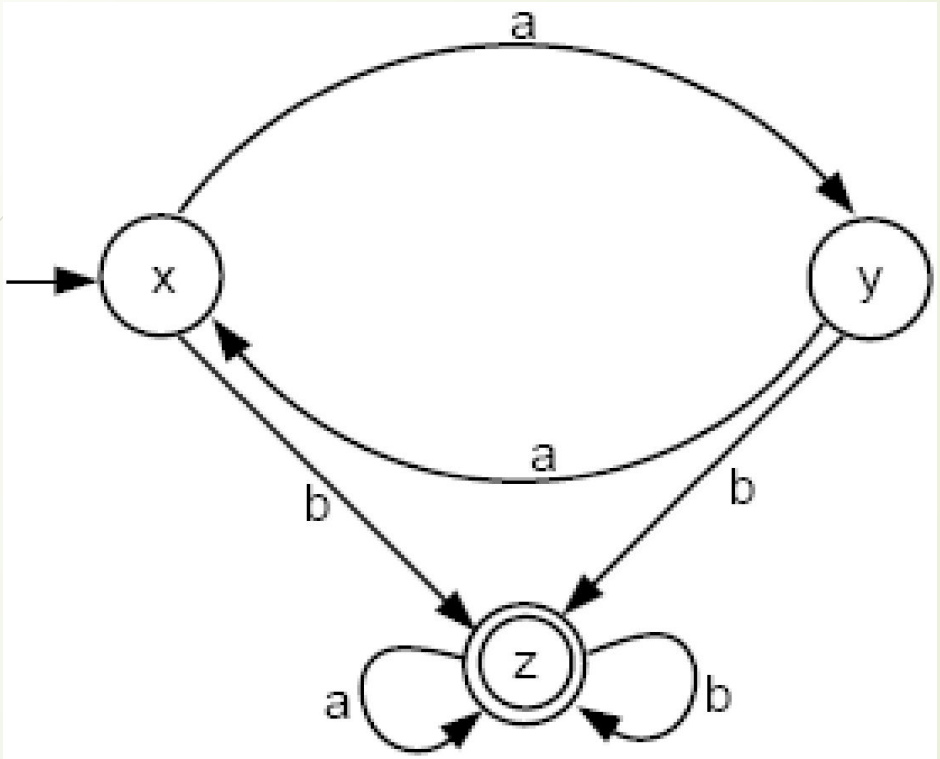
- Pictorial representation of an FA gives us more of a feeling for the motion.
- We represent each state by a small **circle**.
- We draw **arrows** showing to which other states the different **input letters** will lead us. We label these arrows with the corresponding input letters.
- If a certain letter makes a state go back to itself, we indicate this by a **loop**.
- We indicate the start state by a **minus sign**, or by labeling it with the word **start**.
- We indicate the final states by **plus signs**, or by labeling them with the word **final**.
- Sometimes, a start state is indicated by an arrow, and a final state is indicated by drawing another circle around its circle.

Transition Diagram…



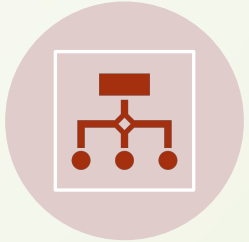
Transition Diagram…



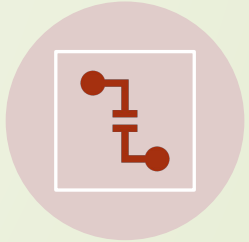


Transition Diagram…

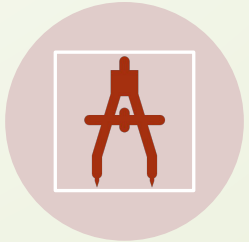
Transition Diagrams contd.



When we depict an FA as circles and arrows, we say that we have drawn a directed graph.



We borrow from Graph Theory the name directed edge, or simply edge, for the arrow between states.

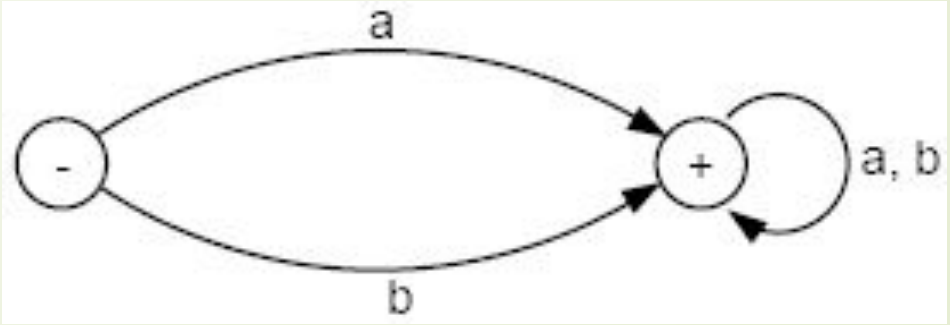


Every state has as many outgoing edges as there are letters in the alphabet.

It is possible for a state to have no incoming edges or to have many.

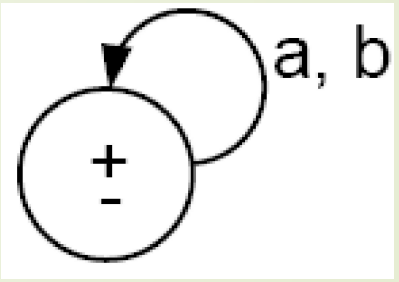
Example

- By convention, we say that the null string starts in the start state and ends also in the start state for all FAs.
- Consider this FA:



Example

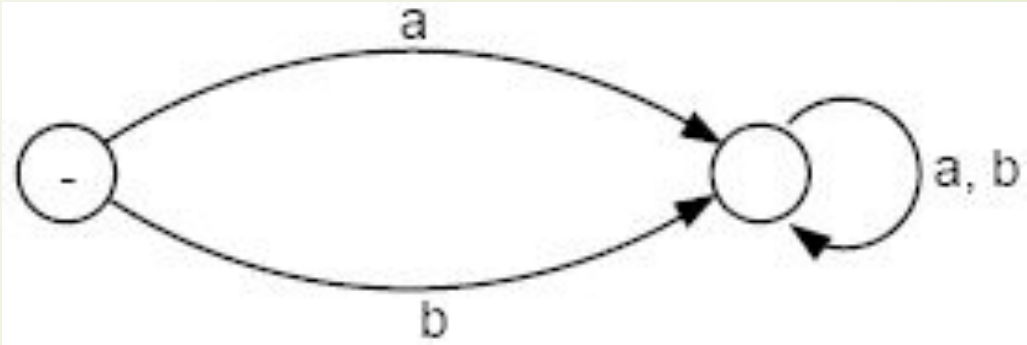
- One of many FAs that accept all words is



- Here, the ± means that the same state is both a start and a final state

Example

- There are FAs that accept no language. These are of two types:
- The first type includes FAs that have no final states, such as



FA and their Languages

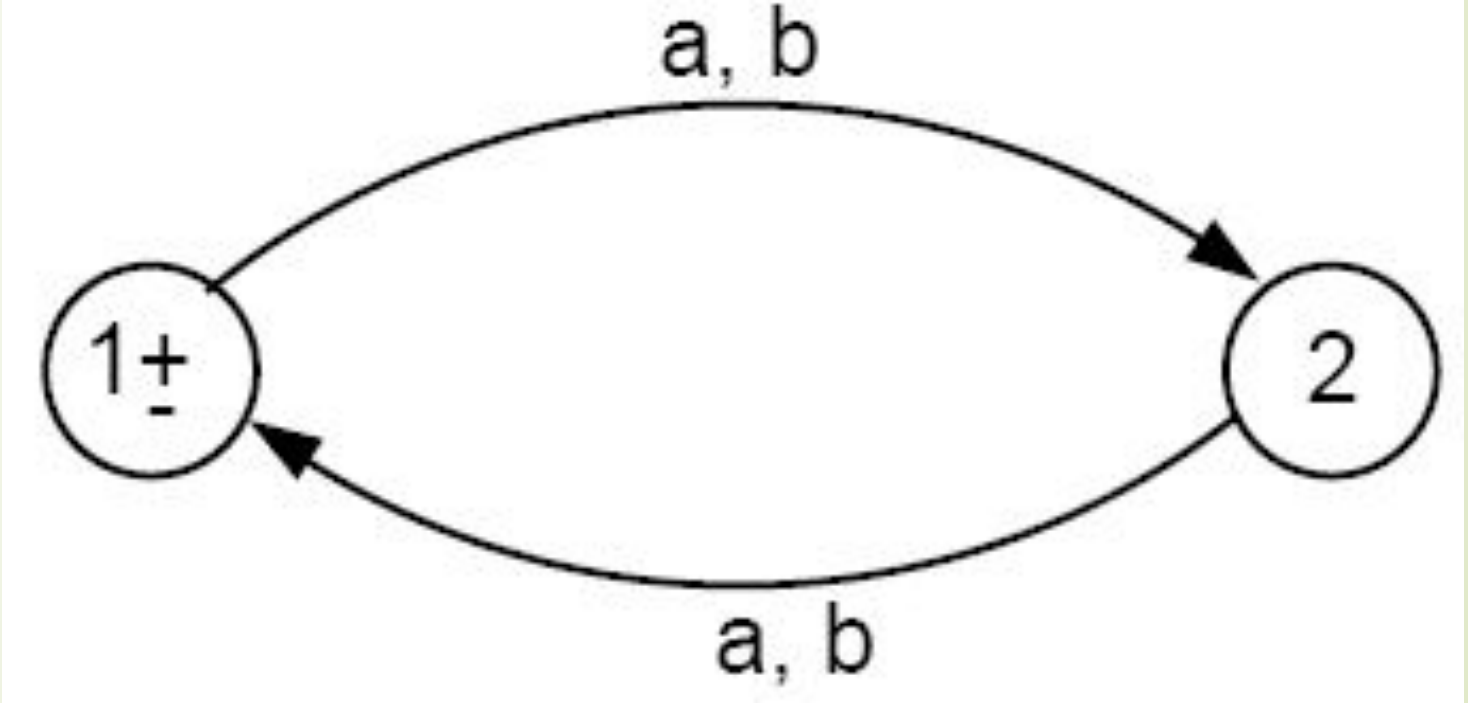
- We will study FA from two different angles:
  - Given a language, can we build a machine for it?
  - Given a machine, can we deduce its language?

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

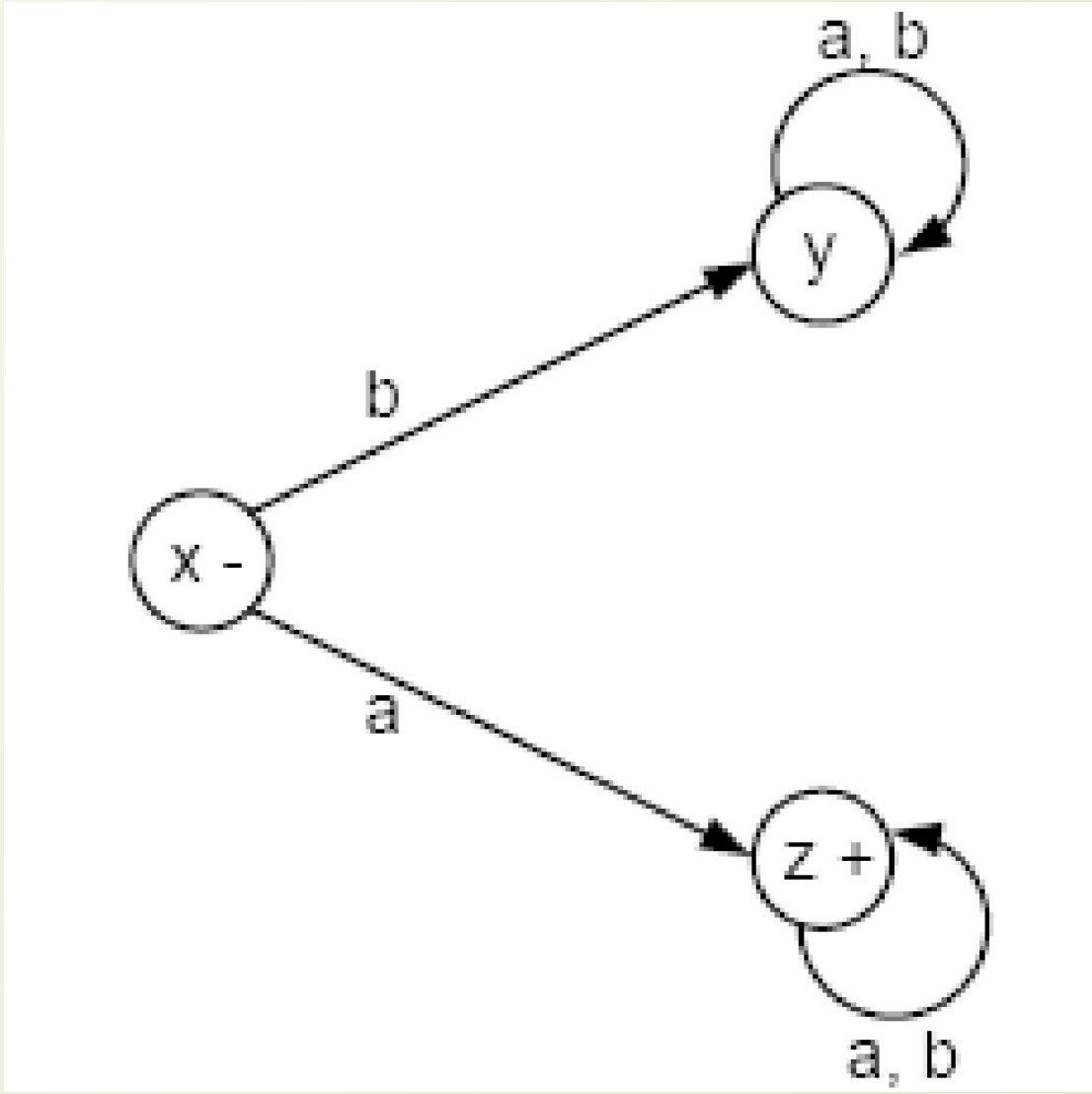
- Let us build a FA that accepts all the words in the language

  $a(a + b)^*$

- 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.

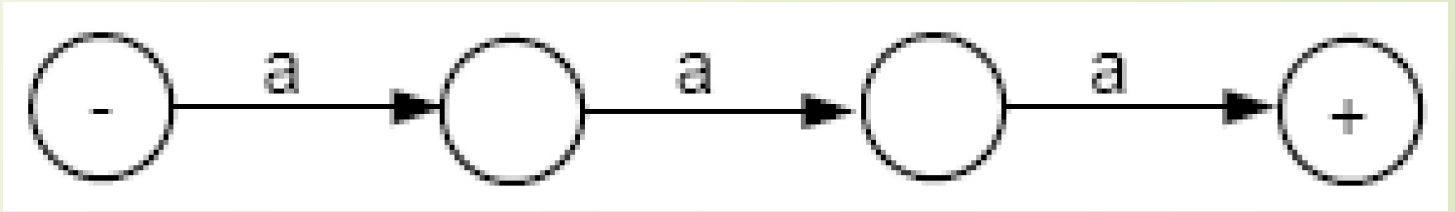
Example

- The machine looks like this:



Example

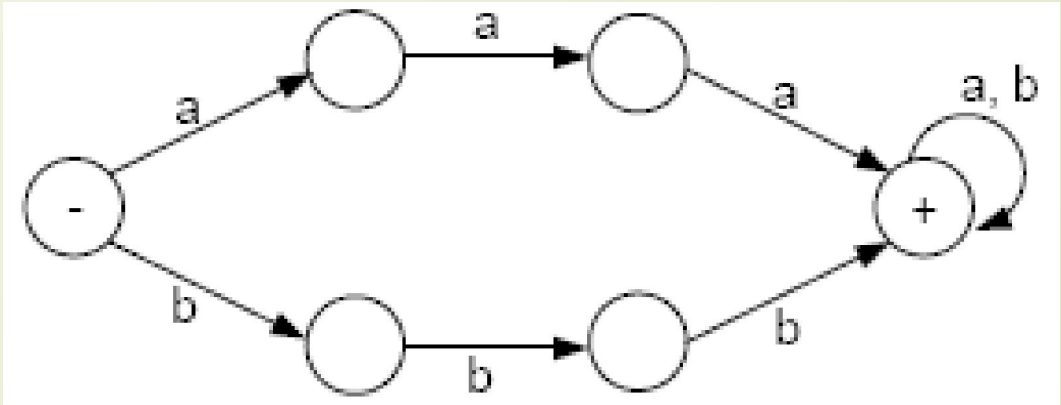
- Let's build a machine that accepts all words **containing a triple letter**, either *aaa* or *bbb*, and only those words.
- From the start state, the FA must have a path of three edges, with no loop, to accept the word *aaa*. So, we begin our FA with the following:



Example Contd.

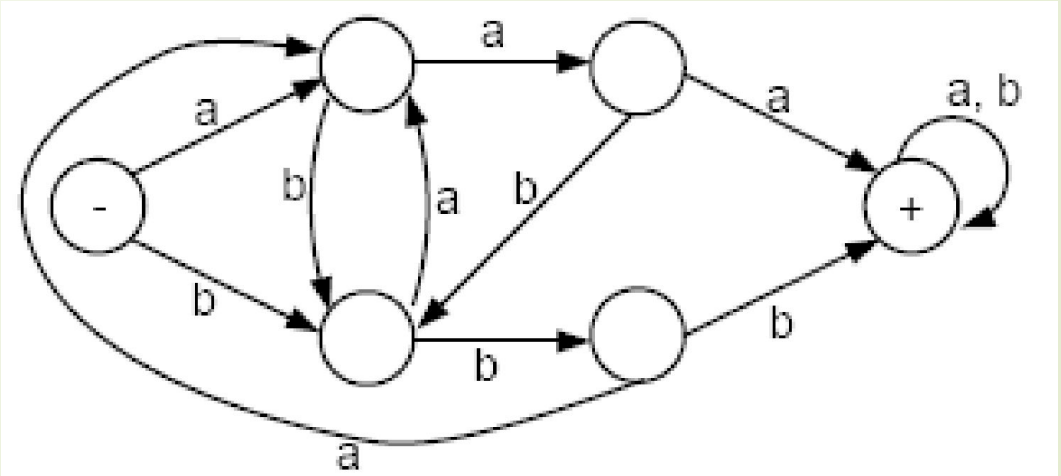
- For similar reason, there must be a path for bbb, that has no loop, and uses entirely differently states. If the b-path shares any states with the a-path, we could mix a's and b's to get to the final state. However, the final state can be shared.

- So, our FA should now look like:



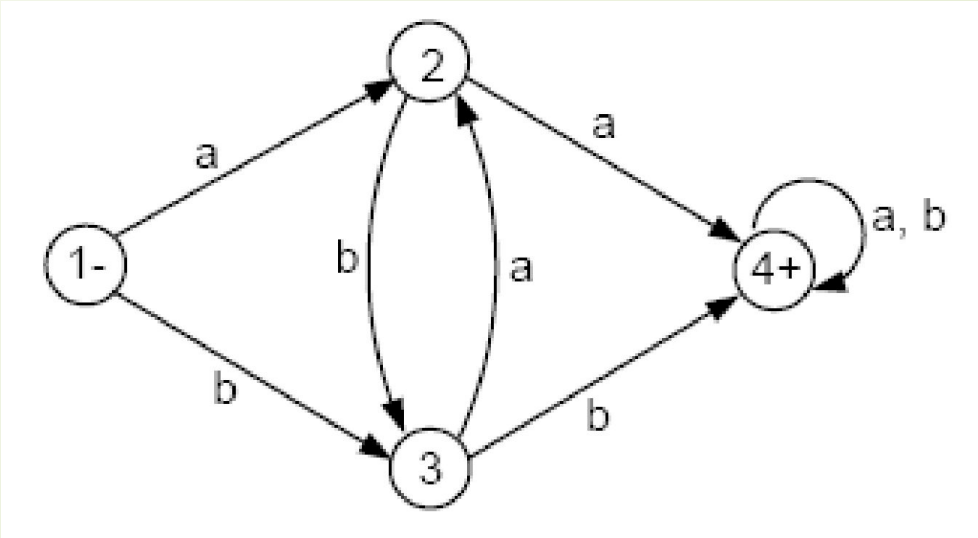
Example Contd.

- 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:



Example

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



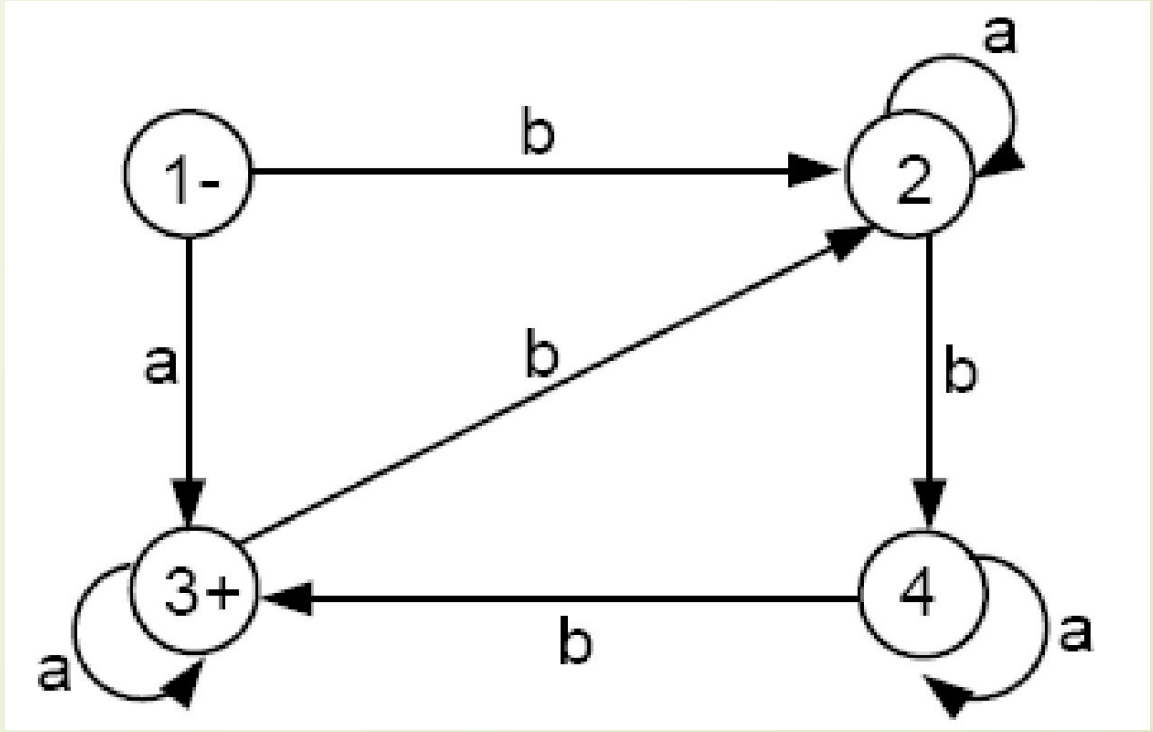
Example

- 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)*

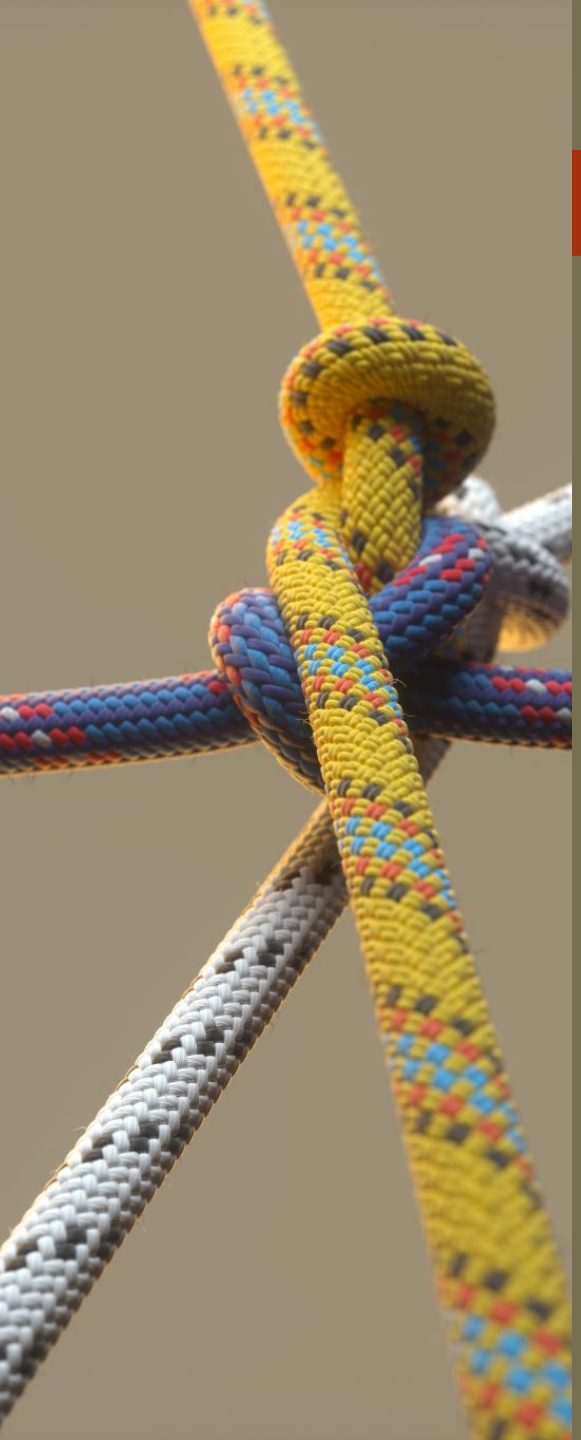
Example

- 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.

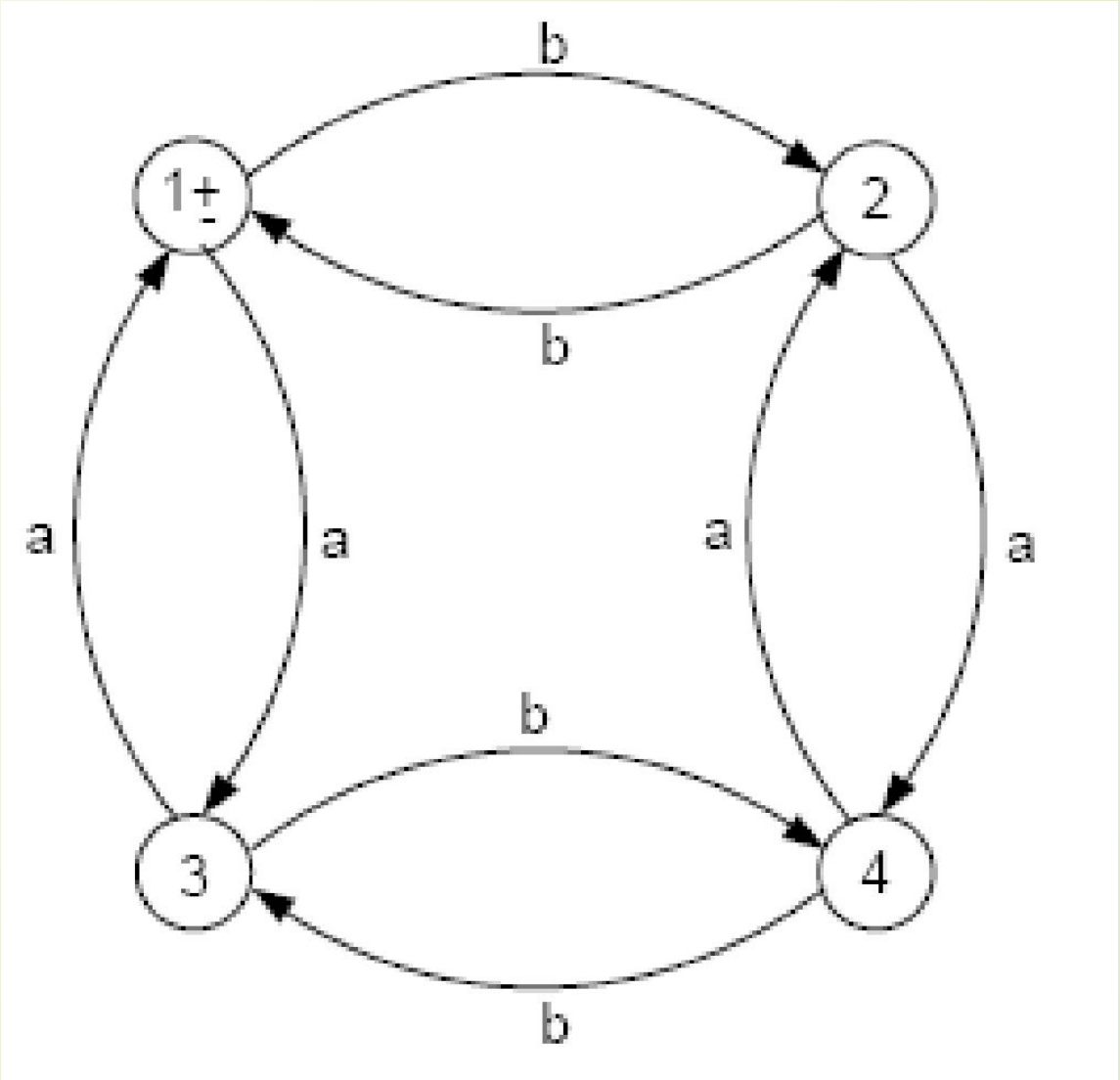
All in all, an input string, whether starting with an a or a b, must have a total number of b's divisible by 3 to be accepted.

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

- Consider the FA below.



Example *EVEN-EVEN* …

- 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* …

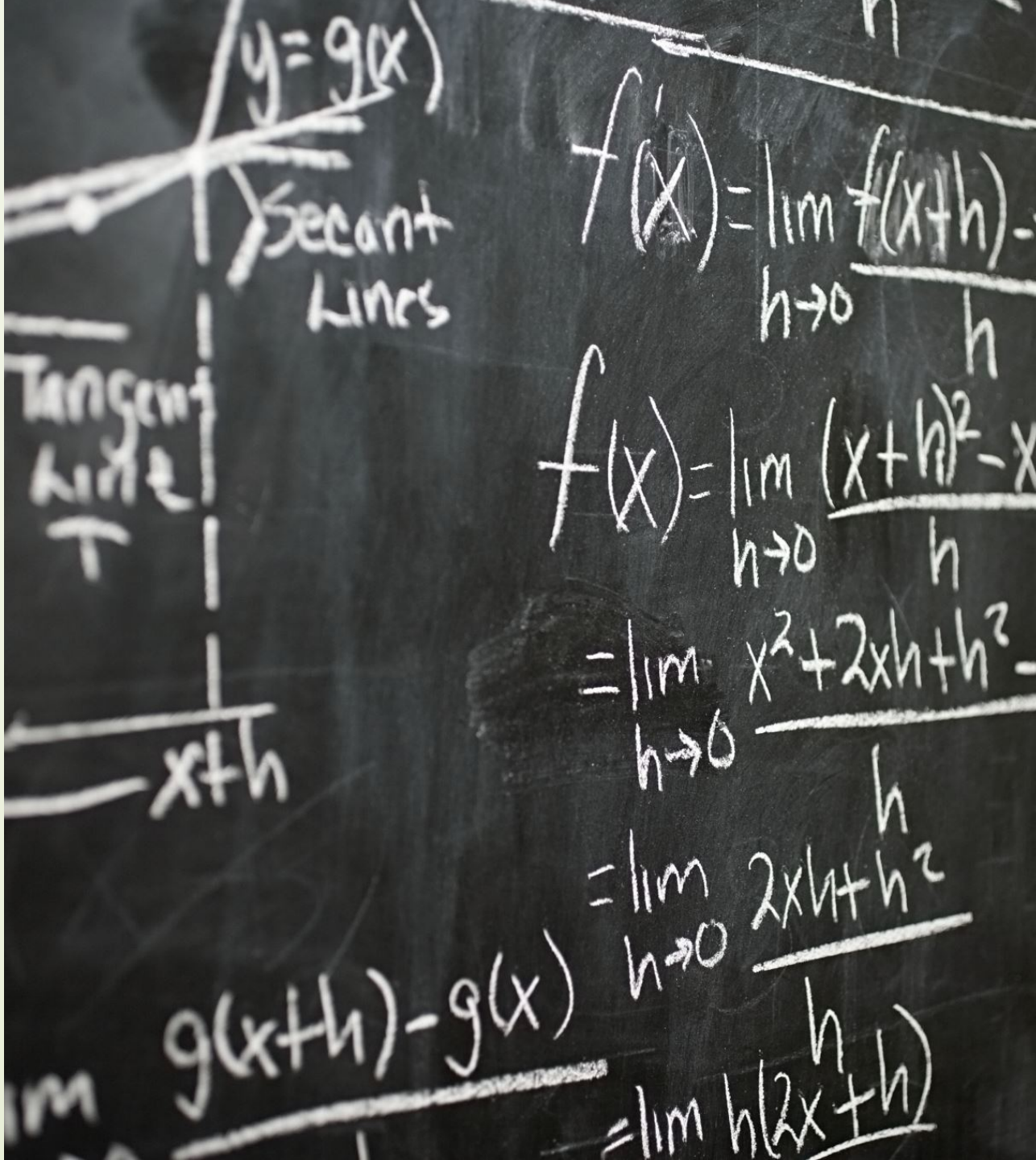
Therefore, all the words in the language accepted by this FA must have an even number of a's and an even number of b's. So, they are in the language *EVEN-EVEN*.

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*.

Examples

Examples on Board



SE4033 Formal Methods for Software Engineering

Regular Expression

38

Regular Expressions

Defining Languages by Another New Method

Formal Definition of Regular Expressions

Languages Associated with Regular Expressions

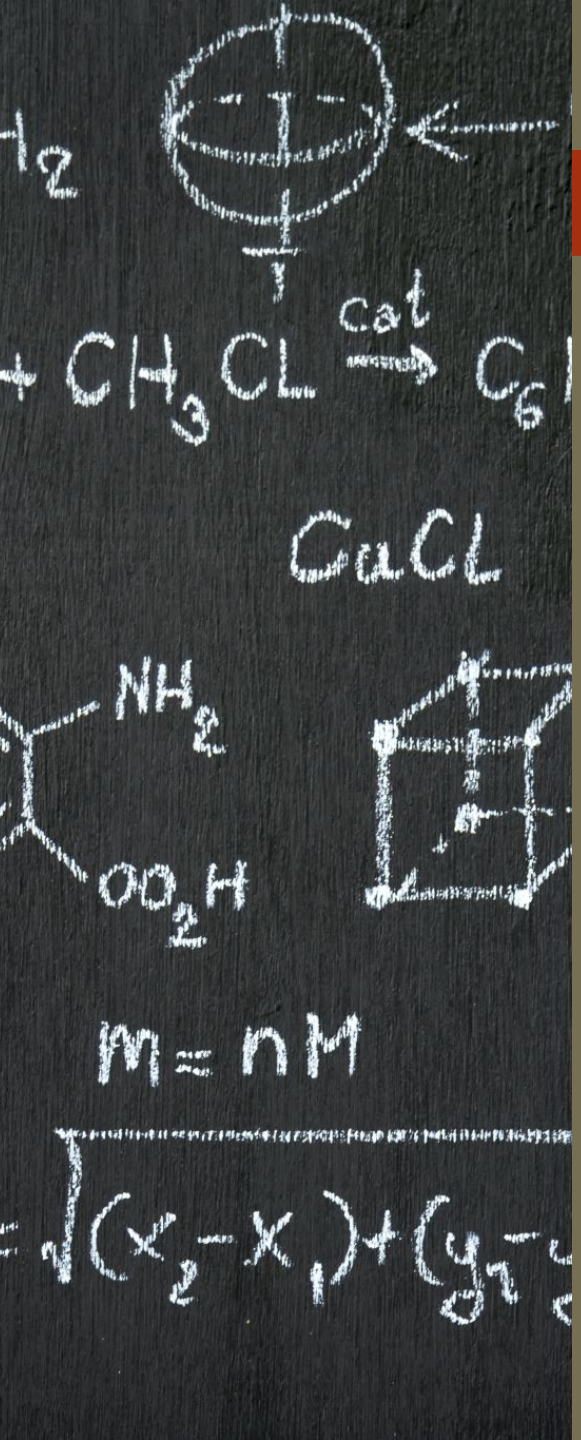
Finite Languages Are Regular

How Hard It Is to Understand a Regular Expression

Introducing EVEN-EVEN

Regular Expression

- A formal way of writing expressions
- Characters
- + OR
- * Repeat multiple times
- + Repeat once or zero times
- R.E.=(a+b)*



Language-Defining Symbols

- We now introduce the use of the Kleene star, applied not to a set, but directly to the letter x and written as a superscript: x*.
- This simple expression indicates some sequence of x's (may be none at all):

  $\mathbf{x}^* = \lambda$ or x or x^2 or x^3…

  $= x^n$ for some n = 0, 1, 2, 3, …
- Letter **x** is intentionally written in boldface type to distinguish it from an alphabet character.
- We can think of the star as an unknown power. That is, **x*** stands for a string of x's, but we do not specify how many, and it may be the null string .

- The notation x* can be used to define languages by writing, say L_4 = language (x*)
- Since x* is any string of x's, L_4 is then the language of all possible strings of x's of any length (including **λ**).
- We should not confuse x* (which is a **language-defining symbol**) with L_4 (which is the **name** we have given to a certain language).

- Given the alphabet = {a, b}, suppose we wish to define the language L that contains all words of the form one *a* followed by some number of *b*'s (maybe no *b*'s at all); that is

  L = {a, ab, abb, abbb, abbbb, …}

- Using the language-defining symbol, we may write

  L = language (ab*)

- This equation obviously means that L is the language in which the words are the concatenation of an initial a with some or no b's.

- From now on, for convenience, we will simply say **some** b**'s** to mean **some or no** b**'s**. When we want to mean **some positive number of** b**'s**, we will explicitly say so.

We can apply the Kleene star to the whole string ab if we want:

(ab)* = or ab or abab or abababab…

Observe that

(ab)* ≠ a*b*

because the language defined by the expression on the left contains the word abab, whereas the language defined by the expression on the right does not.

- If we want to define the language L1 = {x; xx; xxx; …} using the language-defining symbol, we can write

  L1 = language(xx*)

  which means that each word of L1 must start with an x followed by some (or no) x's.

- Note that we can also define L1 using the notation + (as an exponent) introduced in Chapter 2:

  L1 = language(x^+)

- which means that each word of L1 is a string of some positive number of x's.

Plus Sign

- Let us introduce another use of the plus sign. By the expression

  x + y

  where x and y are strings of characters from an alphabet, we mean **either** x **or** y.

- Care should be taken so as not to confuse this notation with the notation + (as an exponent).

Example

Consider the language T over the alphabet $\Sigma = \{a; b; c\}$:

T = {a; c; ab; cb; abb; cbb; abbb; cbbb; abbbb; cbbbb; …}

"In other words, all the words in T begin with either an ***a*** or a ***c*** and then are followed by some number of ***b***'s.

Using the above plus sign notation, we may write this as

T = language((a+ c)b*)

Example

- Consider a finite language L that contains all the strings of a's and b's of length three exactly:

  L = {aaa, aab, aba, abb, baa, bab, bba, bbb}

- Note that the first letter of each word in L is either an a or a b; so are the second letter and third letter of each word in L.
- Thus, we may write

  L = language((a+ b)(a + b)(a + b))

- or for short,

  L = language$((a+ b)^3)$

Example

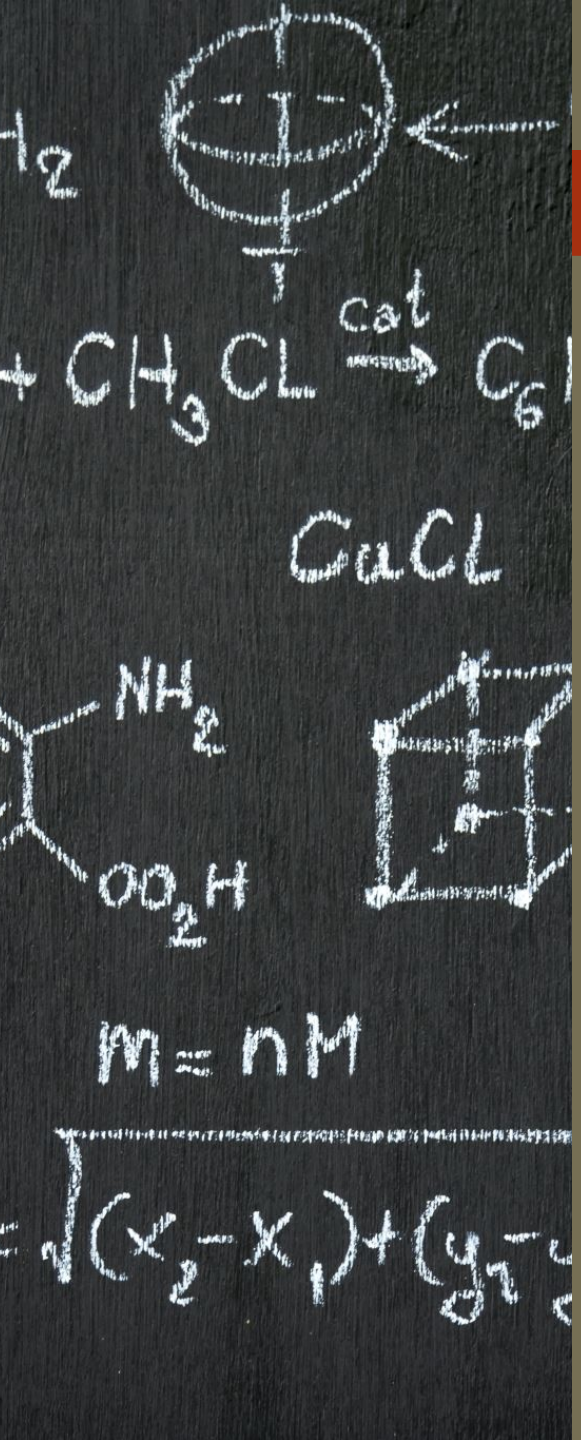
In general, if we want to refer to the set of all possible strings of a's and b's of any length whatsoever, we could write

language((a+ b)*)

This is the set of **all possible strings** of letters from the alphabet Σ = {a, b}, **including the null string**.

This is powerful notation. For instance, we can describe all the words that begin with first an a, followed by anything (i.e., as many choices as we want of either a or b) as

a(a + b)*



Formal Definition of Regular Expressions

- The set of **regular expressions** is defined by the following rules:
- Rule 1: Every letter of the alphabet Σ can be made into a regular expression by writing it in **boldface, λ** itself is a regular expression.
- Rule 2: If r_1 and r_2 are regular expressions, then so are:

  (i) (r_1)

  (ii) $r_1 r_2$

  (iii) $r_1 + r_2$

  (iv) r_1^*
- Rule 3: Nothing else is a regular expression.
- Note: If $r_1 = aa + b$ then when we write r_1^* , we really mean $(r_1)^*$,

  that is $r_1^* = (r_1)^* = (aa + b)^*$

Example

Consider the language defined by the expression $(a + b)^*a(a + b)^*$

At the beginning of any word in this language we have

*(a + b)**, which is any string of *a*'s and *b*'s, then comes an *a*, then another any string.

For example, the word abbaab can be considered to come from this expression by 3 different choices:

(λ)a(bbaab) *or (abb)a(ab)* *or (abba)a(b)*

Example...

- This language is the set of all words over the alphabet Σ = {a, b} that have at least one a.
- The only words left out are those that have only b's and the word λ.

These left out words are exactly the language defined by the expression b*.

- If we combine this language, we should provide a language of all strings over the alphabet Σ = {a, b}. That is,

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

Example

- The language of all words that have at least two a's can be defined by the expression:

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

- Another expression that defines all the words with at least two a's is

  $b^*ab^*a(a + b)^*$

- Hence, we can write

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

  whereby the equal sign we mean that these two expressions are **equivalent** in the sense that they describe the same language.

Example

- The language of all words that have at least one a and at least one b is somewhat trickier. If we write

  (a + b)*a(a + b)*b(a + b)*

then we are requiring that an a must precede a b in the word. Such words as ba and bbaaaa are not included in this language.

- Since we know that either the a comes before the b or the b comes before the a, we can define the language by the expression

  (a + b)a(a + b)b(a + b) + (a + b)b(a + b)a(a + b)

- Note that the only words that are omitted by the first term

  (a + b)*a(a + b)*b(a + b)* are the words of the form some b's followed by some a's. They are defined by the expression bb*aa*

Example

- We can add these specific exceptions. So, the language of all words over the alphabet $\Sigma = \{a, b\}$ that contain at least one a and at least one b is defined by the expression:

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

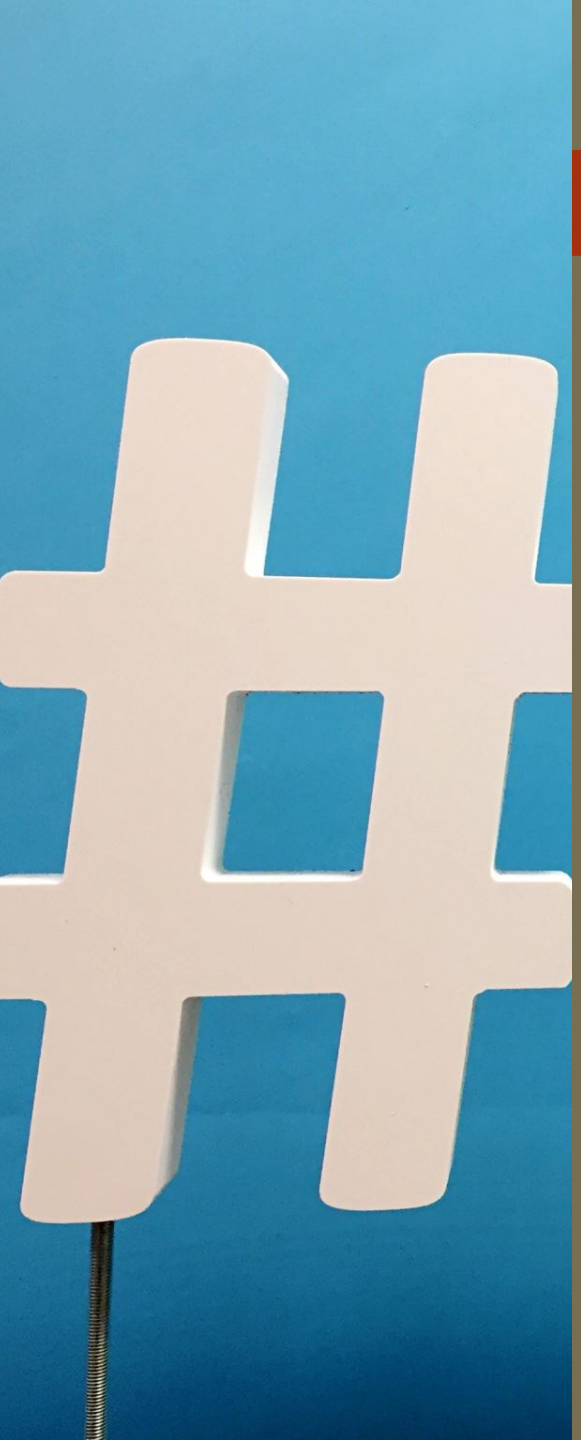
- Thus, we have proved that

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

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

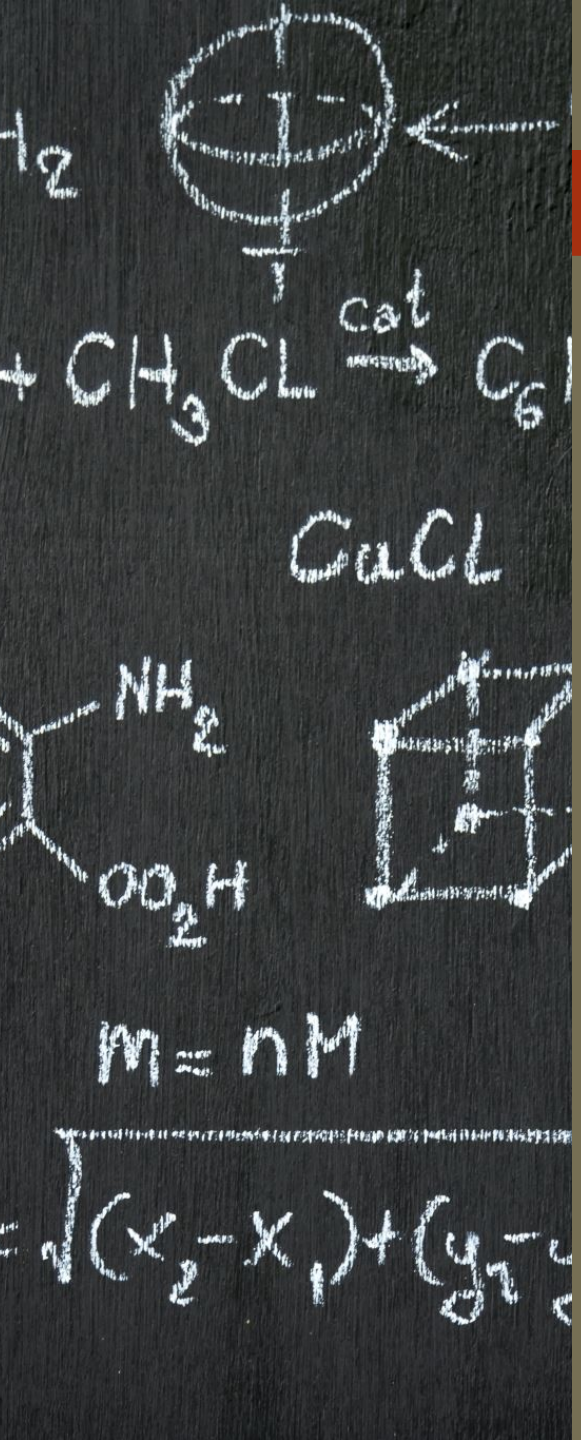
Example

- In the above example, the language of all words that contain both an a and ab is defined by the expression $(a + b)^*a(a + b)^*b(a + b)^* + bb^*aa^*$
- The only words that do not contain both an a and ab are the words of all a's, all b's, or **λ**.
- When these are included, we get everything. Hence, the expression
- $(a + b)^*a(a + b)^*b(a + b)^* + bb^*aa^* + a^* + b^*$
- defines all possible strings of a's and b's, including (accounted for in both a and b).



- Thus

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



Example

- The following equivalences show that we should not treat expressions as algebraic polynomials:

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

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

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

  $(a + b)^* = a(a + b)^* + b(a + b)^* + \lambda$

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

- The last equivalence may need some explanation:
  - The first term in the right-hand side, $(a + b)^*ab(a + b)^*$, describes all the words that contain the substring ab.
  - The second term, b^*a^* describes all the words that do not contain the substring ab (i.e., all a's, all b's, λ, or some b's followed by some a's).

Example

- Let V be the language of all strings of a's and b's in which either the strings are all b's, or else an a followed by some b's. Let V also contain the word λ. Hence,

  V = {λ, a, b, ab, bb, abb, bbb, abbb, bbbb, …}

- We can define V by the expression

  b* + ab*

  where λ is included in b*.

- Alternatively, we could define V by

  (λ + a)b*

which means that in front of the string of some b's, we have either an a or nothing.

Example contd.

- Hence,

  $(\lambda + a)b^* = b^* + ab^*$

- Since $b^* = \lambda\, b^*$, we have

  $(\lambda + a)b^* = b^* + ab^*$

  which appears to be distributive law at work.

- However, we must be extremely careful in applying distributive law. Sometimes, it is difficult to determine if the law is applicable.

Product Set

If S and T are sets of strings of letters (whether they are finite or infinite sets), we define the **product set** of strings of letters to be

ST = {all combinations of a string from S concatenated with a string from T in that order}

Example

- If S = {a, aa, aaa} and T = {bb, bbb} then

ST = {abb, abbb, aabb, aabbb, aaabb, aaabbb}

- Note that the words are not listed in lexicographic order.
- Using regular expression, we can write this example as

(a + aa + aaa)(bb + bbb)

= abb + abbb + aabb + aabbb + aaabb + aaabbb

Example

- If M = {λ, x, xx} and N = {λ, y, yy, yyy, yyyy, …} then
- MN ={λ, y, yy, yyy, yyyy,…x, xy, xyy, xyyy, xyyyy, …xx, xxy, xxyy, xxyyy, xxyyyy, …}
- Using regular expression

  (λ + x + xx)(y*) = y* + xy* + xxy*

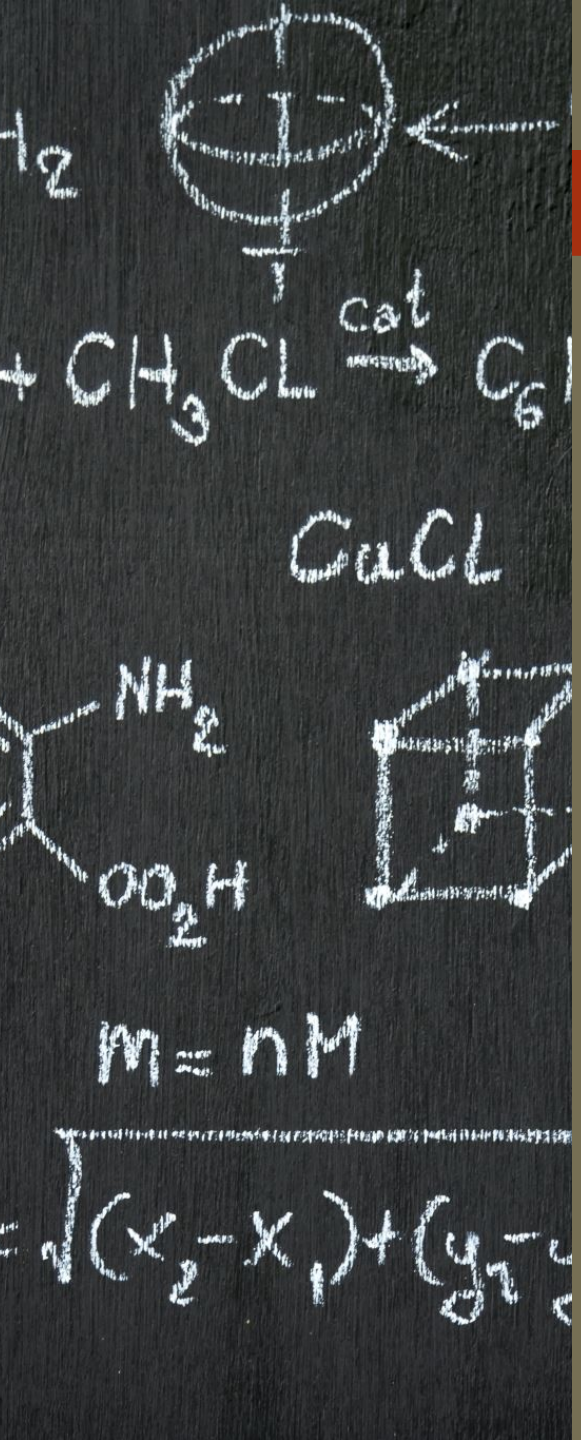

Languages Associated with Regular Expressions

Definition

The following rules define the **language associated** with any regular expression:

Rule 1: The language associated with the regular expression that is just a single letter is that one-letter word alone, and the language associated with Λ is just {Λ}, a one-word language.

Definition contd.



- Rule 2: If r_1 is a regular expression associated with the language L_1 and r_2 is a regular expression associated with the language L_2, then:

  (i) The regular expression $(r_1)(r_2)$ is associated with the product L_1L_2, that is the language L_1 times the language L_2:

  $$\text{language}(r_1r_2) = L_1L_2$$

  (ii) The regular expression r1 + r2 is associated with the language formed by the union of L1 and L2:

  $$\text{language}(r1 + r2) = L1 + L2$$

  (iii) The language associated with the regular expression (r1)* is L1*, the Kleene closure of the set L1 as a set of words:

  $$\text{language}(r1^*) = L1^*$$

How Hard It Is To Understand A Regular Expression

Let us examine some regular expressions and see if we could understand something about the languages they represent.

Example

- Consider the expression

  (a + b)*(aa + bb)(a + b)* =(arbitrary)(double letter)(arbitrary)

- This is the set of strings of a's and b's that at some point contain a double letter.

Let us ask, "What strings do not contain a double letter?" Some examples are

λ; a; b; ab; ba; aba; bab; abab; baba; …

Example...

- The expression (ab)* covers all of these except those that begin with b or end with a. Adding these choices gives us the expression:

  $(\lambda + b)(ab)^*(\lambda + a)$

- Combining the two expressions gives us the one that defines the set of all strings

  $(a + b)^*(aa + bb)(a + b)^* + (\lambda + b)(ab)^*(\lambda + a)$

Examples

- Note that

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

  since the internal * adds nothing to the language. However,

  $(aa + ab^*)^* \neq (aa + ab)^*$

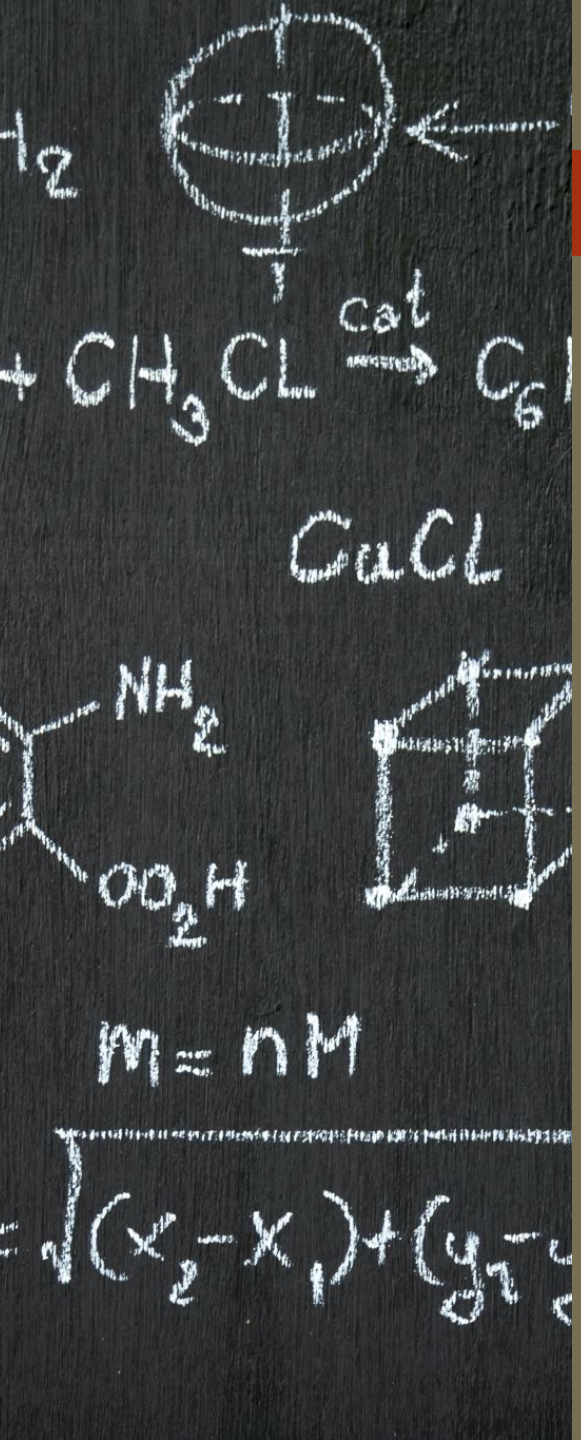
  since the language on the left includes the word *abbabb*, whereas the language on the right does not. (The language on the right cannot contain any word with a double b.)

Example

- Consider the regular expression: (a*b*)*.
- The language defined by this expression is all strings that can be made up of factors of the form a*b*.
- Since both the single letter a and the single letter b are words of the form a*b*, this language contains all strings of a's and b's. That is,

  (a*b*) = (a + b)*

- This equation gives a big doubt on the possibility of finding a set of algebraic rules to reduce one regular expression to another equivalent one.



EVEN-EVEN

- Consider the regular expression

  E = [aa + bb + (ab + ba)(aa + bb)*(ab + ba)]*

- This expression represents all the words that are made up of *syllables* of three types:

  $type_1$ = aa

  $type_2$ = bb

  $type_3$ = (ab + ba)(aa + bb)*(ab + ba)

- Every word of the language defined by E contains an **even number of** a**'s and an even number of** b**'s**.

- All strings with an **even number of** a**'s and an even number of** b**'s** belong to the language defined by E.

Algorithms for EVEN-EVEN

We want to determine whether a long string of a's and b's has the property that the number of a's is even and the number of b's is even.

Algorithm 1: Keep two binary flags, the a-flag and the b-flag. Every time an a is read, the a-flag is reversed (0 to 1, or 1 to 0); and every time a b is read, the b-flag is reversed. We start both flags at 0 and check to be sure they are both 0 at the end.

Algorithm 2: Keep only one binary flag, called the $type_3$-flag. We read letter in two at a time. If they are the same, then we do not touch the $type_3$-flag, since we have a factor of $type_1$ or $type_2$. If, however, the two letters do not match, we reverse the $type_3$-flag. If the flag starts at 0 and if it is also 0 at the end, then the input string contains an even number of a's and an even number of b's.

- If the input string is

(aa)(ab)(bb)(ba)(ab)(bb)(bb)(bb)(ab)(ab)(bb)(ba)(aa) then, by Algorithm 2, the $type_3$-flag is reversed 6 times and ends at 0.

- We give this language the name EVEN-EV EN. so, EVEN-EV EN ={Λ, aa, bb, aaaa, aabb, abab, abba, baab, baba, bbaa, bbbb, aaaaaa, aaaabb, aaabab, …}