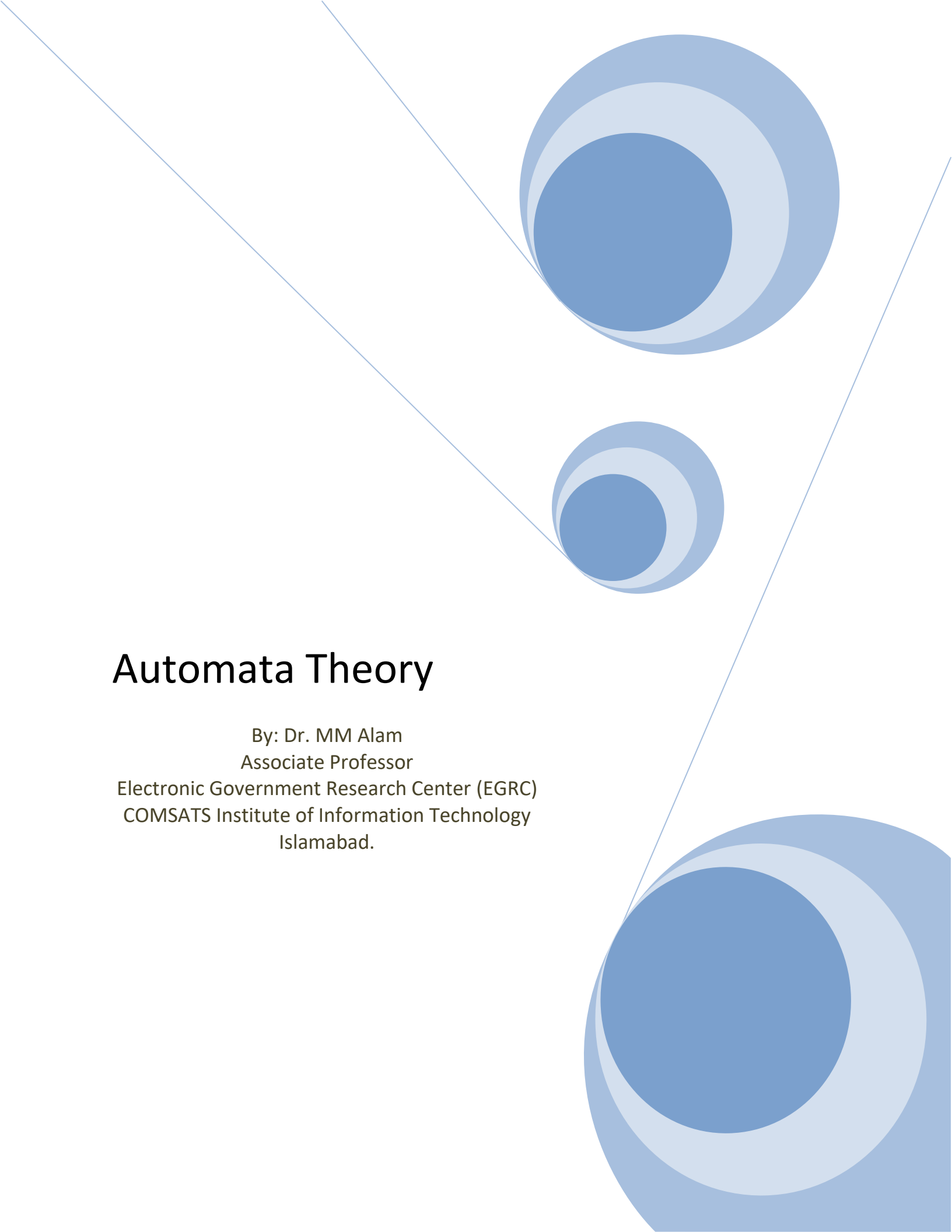


Automata Theory

By: Dr. MM Alam
Associate Professor
Electronic Government Research Center (EGRC)
COMSATS Institute of Information Technology
Islamabad.

A Word of thanks…

First of All, thanks to **Allah** Mighty, who has helped me shape this course in a nice way. Afterwards, I would like to say a bundle of thanks to **Daniel I.A. Cohen** for his wonderful book **Automata Theory**. The beauty of this book is that as the time passes, this book is getting more and more organized and simple for the new comers. I consider it as one of the best books for newbies in computer theory. I have taken material from this book for making slides, including text and examples.

JFLAP Ah! what a wonderful tool made available by **Prof. Susan H. Rodger**. I am impressed by such a dedicated effort. Without JFLAP, just making one diagram in Power point would waste plenty of time. Considering its intellectual value such as its abilities to simulate various aspects of theory of automata is remarkable and cannot be narrated in words. I have taken examples from JFLAP Tutorials and sometimes shown the tool practically in the lectures.

Further, Theory of Automata taught by **Dr. Shahid Siddqui sb** from Virtual University Pakistan has done a lot of efforts in simplifying the course for Pakistani students. I consider myself not very capable of doing the same type of efforts, but I have taken help from his slides too in many places.

Finally, **Mr. Shahid Islam** a colleague of mine and MS computer science student, who has done a lot of help to me in making these slides and hands out.

Instructions

1. The examples are simulated in JFLAP that can be downloaded from http://JFLAP.org for free by giving some basic information.
2. Examples taken from the JFLAP Tutorial accessible at jflap.org/tutorial are not explained in these hands outs. For that, please follow the video lectures.
3. Your system shall be equipped with the latest version of JAVA for running JFLAP software.

Table of Contents

Lecture # 1

Text and Reference Material

Introduction to Computer Theory, by Daniel I. Cohen, John Wiley and Sons, Inc., 1991, Second Edition (as a Text Book)

Introduction to Languages and Theory of Computation, by J. C. Martin, McGraw Hill Book Co., 1997, Second Edition (for Additional Reading)

What does Theory of automata mean?

The word "Theory" means that this subject is a more mathematical subject and less practical.

It is not like your other courses such as programming. However, this subject is the foundation for many other practical subjects.

Automata is the plural of the word Automaton which means "self-acting"

In general, this subject focuses on the theoretical aspects of computer science.

Theory of Automa Applications

This subject plays a major role in:

Theory of Computation

Compiler Construction

Parsing

Formal Verification

Defining computer languages

Types of languages

There are two types of languages

Formal Languages are used as a basis for defining computer languages

A predefined set of symbols and string

Formal language theory studies purely syntactical aspects of a language (e.g., word **abcd**)

Informal Languages such as English has many different versions.

Basic Element of a Formal Language – Alphabets

Definition:

A finite non-empty set of symbols (letters), is called an alphabet. It is denoted by Greek letter sigma Σ.

Example:

Σ={1,2,3}

Σ={0,1} //Binary digits

Σ={i,j,k}

Example Computer Languages

- C Language
- Java Language
- C++
- Java
- Visual C++

What are Strings

A String is formed by combining various symbols from an alphabet.

Example:

If Σ= {1,0} then

0, 1, 110011, …..

Similarly, If Σ= {a,b} then

a, b, abbbbbb, aaaabbbbb, …..

What is an EMPTY or NULL String

A string with no symbol is denoted by (Small Greek letter Lambda) λ or (Capital Greek letter Lambda) Λ. It is called an empty string or null string.

We will prefer Λ in this course. Please don't confuse it with logical operator 'and'.

One important thing to note is that we never allow Λ to be part of alphabet of a language

What are Words

Words are strings that belong to some specific language.

Thus, all words are strings, but vice versa is not true.

Example:

If Σ= {a} then a language L can be defined as

L={a,aa,aaa,….} where L is a set of words of the language. Also a,aa,… are the words of L but not ab.

Defining Alphabets – Guidelines

The following are three important rules for defining Alphabets for a language:

Should not contain empty symbol Λ

Should be finite. Thus, the number of symbols are finite

Should not be ambiguous

Example: an alphabet may contain letters consisting of group of symbols for example Σ_1= {A, aA, bab, d}.

Now consider an alphabet

Σ_2= {A, Aa, bab, d} and a string AababA.

This string can be factored in two different ways

(Aa), (bab), (A)

(A), (abab), (A)

Which shows that the second group cannot be identified as a string, defined over Σ = {a, b}.

This is due to ambiguity in the defined alphabet Σ_2

Why Ambiguity comes: A computer program first scans A as a letter belonging to Σ_2, while for the second letter, the computer program would not be able to identify the symbols correctly.

Ambiguity Rule:- The Alphabets should be defined in a way that letters consisting of more than one symbols should not start with a letter, already being used by some other letter.

Ambiguity Examples

Σ_1= {A, aA, bab, d}

Σ_2= {A, Aa, bab, d}

Σ_1 is a valid alphabet while Σ_2 is an in-valid alphabet.

Similarly,

Σ_1= {a, ab, ac}

Σ_2= {a, ba, ca}

In this case, Σ_1 is a invalid alphabet while Σ_2 is a valid alphabet.

Length of Strings

Definition:

The length of string s, denoted by |s|, is the number of letters/symbols in the string.

Example:

Σ={a,b}

s=aaabb

|s|=5

Word Length Example

Example:

Σ= {A, aA, bab, d}

s=AaAbabAd

Factoring=(A), (aA), (bab), (A), (d)

|s|=5

One important point to note here is that aA has a length 1 and not 2.

Length of strings over n alphabets

Formula: Number of strings of length 'm' defined over alphabet of 'n' letters is $\mathbf{n^m}$

Examples:

The language of strings of length 2, defined over Σ={a,b}is L={aa, ab, ba, bb} *i.e.* number of strings = 2^2

The language of strings of length 3, defined over Σ={a,b} is L={aaa, aab, aba, baa, abb, bab, bba, bbb} *i.e.* number of strings = 2^3

Reverse of a String

Definition:

The reverse of a string s denoted by Rev(s) or s^r, is obtained by writing the letters of s in reverse order.

Example:

If s=abc is a string defined over Σ={a,b,c}

then Rev(s) or s^r= cba

Σ= {A, aA, bab, d}

s=AaAbabAd

Rev(s)=dAbabaAA or

Rev(s)=dAbabAaA

Which one is correct?

PALINDROME:

The language consisting of Λ and the strings s defined over Σ such that Rev(s)=s.

It is to be denoted that the words of PALINDROME are called palindromes.

Example:ForΣ={a,b},

PALINDROME={Λ , a, b, aa, bb, aaa, aba, bab, bbb, ...}

How to form a Palindrome String?

For strings of length =1:

Take rev(s) where s is a string

Example: if s = a then Rev(a) = a

If s=b then Rev(b) = b

Similarly, if Σ= {A, aA, bab, d} and

s=aA then rev(aA) = aA

For strings of length >1:

Take s Rev(s) where s is string

Example: if s = ab then s Rev(s) = abba is a palindrome

Similarly, if Σ= {A, abab, d}

s=Aabab then s Rev(s) = A abababab A which is a palindrome

If the length of Palindrome is even then:

s Rev(s) means length of string s is n and thus, there will be 2^n strings.

As we know that string is of length n and number of symbols in the alphabet is 2, which shows that there are as many palindromes of length 2n as there are the strings of length n *i.e.* the required number of palindromes are 2^n.

If the length of Palindrome is odd then:

How an odd length palindrome looks like:

abc a cba

Formula: s (symbol from alphabet) Rev(s)

For odd length palindrome, string is of length n and symbol from the alphabet which appears in the middle and has a length 1 which is subtracted, plus the Rev(s) which is again of length n.

Thus it is **2n-1**. For example: we have a palindrome abcacba. So it is formed from string: **s = abc a Rev(abc)**
thus: **n-1 +n = 2n -1**

The no of palindromes in the odd case depends on the alphabets/symbols in the Sigma. Suppose if number is 2 that is Σ= {a, b} then number of palindromes of length 2n-1 with 'a' as a middle letter, will be 2^{n-1}.

Similarly the number of palindromes of length 2n-1, with ' b ' as middle letter, will be 2^{n-1} as well. Hence the total number of palindromes of length 2n-1 will be $2^{n-1} + 2^{n-1} = 2\ (2^{n-1}) = 2^n$.

Lecture # 2

Definition of the word Automata

Types of languages, empty/Null String, Alphabets, words, length of strings, Palindromes

How to form palindromes of even and odd length.

Question

Q) Prove that there are as many palindromes of length 2n, defined over Σ = {a,b,c}, as there are of length 2n-1, n = 1,2,3… . Determine the number of palindromes of length 2n defined over the same alphabet as well.

Example:- L={aa,ab,ac,bb,ba,bc,cc,ca,cb} = 9 words.

Solution

Since the number of symbols are 3 in the alphabet, therefore, the length of 2n palindromes will be 3^n. As 2n means that palindromes are of even length.

Similarly, for palindromes of length 2n-1, we need to calculate their number for each respective symbol in the alphabet, that is, the required number of palindromes are 3^{n-1} for symbol a

Similarly the number of palindromes of length 2n-1, with b or c as middle letter, will be 3^{n-1} as well. Hence the total number of palindromes of length 2n-1 will be :

$3^{n-1} + 3^{n-1} + 3^{n-1} = 3\,(3^{n-1}) = 3^n$.

Consider the language of Palindrome over the alphabet {a,b}. Prove that if x is in PALINDROME then so is x^n for any n.

Example: suppose x = aba and is a palindrome and $x^r = (aba)^r$ = aba

Then: $(aba)^5 = (aba\ abaabaabaaba\ aba)^r$

=aba abaabaabaaba

which is again a palindrome

Proof. It is true for n = 0 by assumption $\mathbf{x^n}$ is palindrome.

Assume that it is true for n-1 namely $\mathbf{x^{n-1}}$ is in palindrome and $(\mathbf{x^{n-1}})^r = \mathbf{x^{n-1}}$. Now as we know that $(xy)^r = y^r x^r$.

$\mathbf{(x^n)^r} = (\mathbf{x^{n-1}x})^r = x^r(\mathbf{x^{n-1}})^r = x\,\mathbf{x^{n-1}} = \mathbf{x^n}$. Thus, $\mathbf{x^n}$ is in palindrome

Kleene Star closure

In order to further generalize the notation of any combination of strings the famous logician kleene and founder of the 'Theory of Automata' subject has introduced a notation called kleene closure.

It is denoted by Σ^* and represent all collection of strings defined over Σ including Null string.

The language produced by Kleene closure is infinite. It contains infinite words, however each word has finite length.

Examples

If Σ = {x}

Then Σ* = {Λ, x, xx, xxx, xxxx, ….}

If Σ = {0,1}

Then Σ* = {Λ, 0, 1, 00, 01, 10, 11, ….}

If Σ = {aaB, c}

Then Σ* = {Λ ,aaB, c, aaBaaB, aaBc, caaB, cc, ….}

Q) Consider the language S*, where S = {a b}. How many words does this language have of length 2? of length 3? of length n?

A) Number of words = n^m

Length 2: $2^2 = 4$

Length 3: $2^3 = 8$

Length n: 2^n

Consider the language S*, where S = {aa b}. How many words does this language have of length 4? of length 5? of length 6? What can be said in general?

Words of length 0: Λ = 1 word

Words of length 1: b = 1 word

Words of length 2: (Add aa to all words of length 0 --> 0 + 2 = 2)

Add b to all words of length 1 --> 1 + 1 = 2)

aa bb = 2 words

Words of length 3:

(Add aa to all words of length 1 --> 1 + 2 = 3)

(Add b to all words of length 2 --> 2 + 1 = 3)

aab baa bbb = 3 words

Words of length 4:

(Add b to all words of length 3)

(Add aa to all words of length 2)

baabbbaabbbbaaaaaabb = 5 words

Words of length 5:

(Add aa to the 3 words of length 3)

(Add b to the 5 words of length 4) = 8 words

Words of length 6:

(Add aa to the 5 words of length 4)

(Add b to the 8 words of length 5) = 13 words

This is the Fibonacci sequence: 1,2,3,5,8,13,…

Consider the language S*, where S = {abba}. Write out all the words in S* that have seven or fewer letters. Can any word in this language contain the substrings aaa or bbb? What is the smallest word that is not in this language?

Words of length 0: Λ

Words of length 2: abba

Words of length 4: abababbabaab baba

Words of length 6: abababababbaabbaabababbababaabababbaabbababaabbababa

No words can contain aaa or bbb because the first a in string ab and the a in ba never allow to make aaa or bbb.

Consider the language S*, where S = {xx xxx}. In how many ways can x^{19} be written as the product of words in S? This means: How many different factorizations are there of x^{19} into xx and xxx?

Example: (xx) (xx) (xx) (xx) (xx) (xx) (xx) (xx) + (xxx)

= $x^{16}+x^{3}=x^{19}$

x^{19} can consist of 8 doubles (xx) and 1 triple (xxx) 8*2 + 1*3 = 19

x^{19} can consist of 5 doubles (xx) and 3 triples (xxx) 5*2 + 3*3 = 19

x^{19} can consist of 2 doubles (xx) and 5 triples (xxx) 2*2 + 5*3 = 19

3 doubles can be replaced by 2 triples: (xx)(xx)(xx) = (xxx)(xxx)

Let xx = d and xxx = t

The number of ways of factoring 19 x's into d's and t's is equal to:

The number of ways of arranging 8d's and 1t

+ the number of ways of arranging 5 d's and 3 t's

+ the number of ways of arranging 2 d's and 5 t's.

(i) Let S = {ab bb} and let T = {ab, bb, bbbb}. Show that S* = T*.

(ii) Let S = {ab, bb} and let T = {ab, bb, bbb}. Show that S* ≠ T*, but that S* ⊂ T*

(i) S* = {Λ and all possible concatenations of ab and bb}

T* = {Λ and all possible concatenations of ab, bb, and bbb}, but bbb is just bb concatenated with itself, so

T* = {Λ and all concatenations of ab and bb} = S*.

(ii) Let S = {ab, bb} and let T = {ab, bb, bbb}. Show that S* ≠ T*, but that S* ⊂ T*

The reason is that T contains triple b, and S contains double b. S* will always add double b, thus resulting string will be even only. However, S* ⊂ T*. Please evaluate S* and T* to prove above statement.

Plus Operation

With Plus Operation, combination of different letters are formed. However, Null String is not part of the generated language.

Examples

If Σ = {x}

Then Σ^+ = { x, xx, xxx, xxxx, ….}

If Σ = {0,1}

Then Σ^+ = { 0, 1, 00, 01, 10, 11, ….}

If Σ = {aaB, c}

Then Σ^+ = {aaB, c, aaBaaB, aaBc, caaB, cc, ….}

Lecture # 3

Prove that for all sets S,

- (S+)* = (S*)*
- (S+)+ = S+
- Is (S*)+ = (S+)* for all sets S?

S^+ = {all concatenations of words in S, excluding Λ}.

$(S^+)^*$ = {Λ and all concatenations of words in S^+}

= {Λ and all concatenations of (concatenations of words in S, excluding Λ)}

= {Λ and all concatenations of words in S}

= S*

S* = {Λ and all concatenations of words in S}

(S*)* = {Λ and all concatenations of words in S}

= {Λ and all concatenations of (Λ and all concatenations of words in S)}

= {Λ and all concatenations of words in S}

= S*

Therefore $(S^+)^* = (S^*)^* = S^*$

(ii) $(S^+)^+ = S^+$

L.H.S

S^+ = {all concatenations of words in S, excluding Λ}

$(S^+)^+$ = {all concatenations of words in S^+, excluding Λ}

= {all concatenations of (all concatenations of words in S, excluding Λ) excluding Λ)}

= {all concatenations of words in S, excluding Λ}

= S^+

(iii) Is $(S^*)^+ = (S^+)^*$ for all sets S?

R.H.S:

S^* = {Λ and all concatenations of words in S}

$(S^*)^+$ = {all concatenations of words in S^*, but not Λ}

S^* already contains Λ, so $(S^*)^+$ contains Λ too, since it's part of the language.

No new words are added with the + operator, so $(S^*)^+ = S^*$.

S^+ = {all concatenations of words in S, without Λ}

$(S^+)^*$ = {Λ and all concatenations of words in S^+}

= {Λ and all concatenations of (all concatenations of words in S, not Λ)}

= {Λ and all concatenations of words in S}

= S^*

The external * operator only added Λ to the language.

Therefore $(S^*)^+ = (S^+)^* = S^*$.

Let S = {a, bb, bab, abaab}. Is abbabaabab in S*? Is abaabbabbaab? Does any word in S* have an odd total number of b's?

(a)(bb)(abaab)ab can't be factored into substrings from S, so it is not in the language. (abaab)(bab)b

(a)(a)(bb) can't be factored into substrings from S, so it's not in the language.

It is not possible to have an odd no of b's. The reason is that even b's +even b's = even b's

Suppose that for some language L we can always concatenate two words in L and get another word in L if and only if the words are not the same. That is, for any words w1 and w2 in L where w1 ≠ w2, the word w1w2 is in L but the word w1w1 is not in L. Prove that this cannot happen.

w1 ∈ L and w2 ∈ L (2 different words)

therefore (w1)(w2) ∈ L

w1w2 ∈ L and w1 ∈ L (2 different words)

therefore (w1w2)(w1) ∈ L

w1w2w1 ∈ L and w2 ∈ L (2 different words)

therefore (w1w2w1)(w2) ∈ L

But (w1w2w1)(w2) can also be factored as: (w1w2)(w1w2), which are 2 equal factors / words, so this language can't exist.

Let us define (S)* = S*** Is this set bigger than S*? Is it bigger than S?**

S* is no bigger than S*. Both sets contain an infinite number of elements. S*** is bigger than S (if S is not {Λ}) because it is made up of all concatenations of elements in S.**

(i) If S = {a b} and T* = S*, prove that T must contain S. (ii) Find another pair of sets S and T such that if T* = S*, then S ⊂ T

(i)

S* = {Λ and all concatenations of a's and b's} = T*.

This means that a ∈ T and b ∈ T. (The smallest words in S* and T*).

T may contain other strings (only concatenations of a & b) and still have the same closure as

S*, but it must have at least the elements of S (which are the smallest factors in the definition of S*), so T contains S.

(ii) Find another pair of sets S and T such that if T* = S*, then S ⊂ T

T = {a, b, aa, abb}

S ⊂ T

S = {a b aa}

Still S* = T* but S ⊂ T

Four different ways, in which a language can be defined

So far, we have studied many languages like INTEGER, PRIME, PALINDROME etc.,

Now, we will take a detailed look at how a language can be defined in different ways.

There are four ways that we will study in this course:

Descriptive way

Recursive way

Regular Expression

Finite Automata

The language and its associated conditions are defined in plain English.

This way is semi-formal way with chances of ambiguity.

Descriptive method is used in early phases of requirement engineering and resultant equations are then formally transformed using other methods.

Example:

The language L of strings of even length, defined over Σ={b}, can be written as

L={bb, bbbb, …..}

Example:

The language L of strings that does not start with a, defined over Σ={a,b,c}, can be written as

L={b, c, ba, bb, bc, ca, cb, cc, …}

Example:

The language L of strings of length 3, defined over Σ={0,1,2}, can be written as

L={000, 012, 022,101, 101,120,…}

Example:

The language L of strings ending in 1, defined over Σ ={0,1}, can be written as

L={1,001,101,0001,0101,1001,1101,…}

Example:

The language **EQUAL**, of strings with number of a's equal to number of b's, defined over Σ={a,b}, can be written as

{Λ ,ab,aabb,abab,baba,abba,…}

Example:

The language **EVEN-EVEN**, of strings with even number of a's and even number of b's, defined over Σ={a,b}, can be written as

{Λ, aa, bb, aaaa,aabb,abab, abba, baab, baba, bbaa, bbbb,…}

Example:

The language **INTEGER**, of strings defined over Σ={-,0,1,2,3,4,5,6,7,8,9}, can be written as

INTEGER = {…,-2,-1,0,1,2,…}

Example:

The language **EVEN**, ofstings defined over Σ={-,0,1,2,3,4,5,6,7,8,9}, can be written as

EVEN = { …,-4,-2,0,2,4,…}

Example:

The language $\{a^n b^n\}$, of strings defined over Σ={a,b}, as

$\{a^n b^n: n=1,2,3,\ldots\}$, can be written as

{ab, aabb, aaabbb,aaaabbbb,…}

Example:

The language $\{a^n b^n c^n\}$, of strings defined over Σ={a,b,c}, as

$\{a^n b^n c^n: n=1,2,3,\ldots\}$, can be written as

{abc, aabbcc, aaabbbccc,aaaabbbbcccc,…}

Lecture # 4

Recursive way

A recursive definition is fundamentally a three-step process:

We specify some basic objects in the set. The number of basic objects specified must be finite. This means that we write some basic facts about the set

Second, we give a finite number of basic rules for constructing more objects in the set from the ones we already know.

Finally, we provide declaration that no objects except those constructed in this way are allowed in the set.

Language of Integer

Defining language of INTEGER

Rule 1: 1 is in **INTEGER**.

Rule 2: If x is in **INTEGER** then x+1 and x-1 are also in **INTEGER**.

Rule 3: No strings except those constructed in above, are allowed to be in **INTEGER**.

Defining the language L, of strings beginning and ending in different letters , defined over Σ={a, b}

Rule 1: a and b are in L

Rule 2: (a)s(b) and (b)s(a) are also in **L,** where s belongs to Σ^*

Rule 3: No strings except those constructed in above, are allowed to be in **L**

Factorial Example

Lets consider the language of factorial:

Rule 1: We know that 0!=1, so 1 is in **factorial**.

Rule 2: Also n!=n*(n-1)! is in **factorial**.

Rule 3: Thus, no strings except those constructed in above, are allowed to be in **factorial**.

Consider the set P-EVEN, which is the set of positive even numbers.

Method#1:

We can define P-EVEN to be the set of all positive integers that are evenly divisible by 2. OR

P-EVEN is the set of all 2n, where n = 1, 2, ...

P-EVEN is defined by these three rules:

Rule 1: 2 is in P-EVEN.

Rule 2: If x is in P-EVEN, then so is x + 2.

Rule 3: The only elements in the set P-EVEN are those that can be produced from the two rules above.

How to apply a recursive definition

In particular, to show that 12 is in P-EVEN using the last definition, we would have to do the following:

2 is in P-EVEN by Rule 1.

2 + 2 = 4 is in P-EVEN by Rule 2.

4 + 2 = 6 is in P-EVEN by Rule 2.

6 + 2 = 8 is in P-EVEN by Rule 2.

8 + 2 = 10 is in P-EVEN by Rule 2.

10 + 2 = 12 is in P-EVEN by Rule 2.

Method#2

We can make another definition for P-EVEN as follows:

Rule 1: 2 is in P-EVEN.

Rule 2: If x and y are both in P-EVEN, then x + y is in P-EVEN.

Rule 3: No number is in P-EVEN unless it can be produced by rules 1 and 2.

How to show that 12 is in P-EVEN:

2 is in P-EVEN by Rule 1.

2 + 2 = 4 is in P-EVEN by Rule 2.

4 + 4 = 8 is in P-EVEN by Rule 2.

4 + 8 = 12 is in P-EVEN by Rule 2.

Example: Let PALINDROME be the set of all strings over the alphabet = {a, b} PALINDROME = {w : w = reverse(w)} = {Λ, a, b, aa, bb, aaa, aba, bab, bbb, aaaa, abba, . . .}.

A recursive definition for PALINDROME is as follows:

Rule 1: , a, and b are in PALINDROME.

Rule 2: If w 2 PALINDROME, then so areawa and bwb.

Rule 3: No other string is in PALINDROME unless it can be produced by rules 1 and 2.

Regular Expressions

Regular expressions is a common means of defining a language. It actually provide concise matching of string.

The concept is associated with the Kleenes formalism of regular sets introduced in 1950

Useful operators used in the regular expression are as follows:

* = 0 or more

+ = 1 or more

a|b = a or b

(a|b)* = (a or b) 0 or more times

$(a|b)^+$ = (a or b) 1 or more times

a|b* = a or b (only b 0 or more times)

(a|b)*(c|Λ) = (a or b) 0 or more times and c or Null string

Construct a regular expression for all words in which a appears tripled, if at all. This means that every clump of a's contains 3 or 6 or 9 or 12... a's

(aaa + b)* or (b + aaa)* or ((aaa)*b*)* or (b*(aaa)*)*

Construct a regular expression for all words that contain at least one of the strings s1, s2, s3, or s4

(s1 + s2 + s3 + s4)(s1 + s2 + s3 + s4)*

Construct a regular expression for all words that contain exactly two b's or exactly three b's, not more.

a*ba*ba* + a*ba*ba*ba* or

a*(b + Λ)a*ba*ba*

Construct a regular expression for:

(i) all strings that end in a double letter.

(ii) all strings that do not end in a double letter

(i)

(a + b)*(aa + bb)

(ii)

(a + b)*(ab + ba) + a + b + Λ

Construct a regular expression for all strings that have exactly one double letter in them

(b + Λ)(ab)*aa(ba)*(b + Λ) + (a + Λ)(ba)*bb(ab)*(a + Λ)

Construct a regular expression for all strings in which the letter b is never tripled. This means that no word contains the substring bbb

(Λ + b + bb)(a + ab + abb)*

Words can be empty and start and end with a or b. A compulsory 'a' is inserted between all repetitions of b's.

Construct a regular expression for all words in which a is tripled or b is tripled, but not both. This means each word contains the substring aaa or the substring bbb but not both.

(Λ + b + bb)(a + ab + abb)*aaa(Λ + b + bb)(a + ab + abb)* +

(Λ + a + aa)(b + ba + baa)*bbb(Λ + a + aa)(b + ba + baa)*

Let r1, r2, and r3 be three regular expressions. Show that the language associated with (r1 + r2)r3 is the same as the language associated with r1r3 + r2r3. Show that r1(r2 + r3) is equivalent to r1r2 + r1r3. This will be the same as providing a 'distributive law' for regular expressions.

(r1+ r2)r3:

The first expression can be either r1 or r2.

The second expression is always r3.

There are two possibilities for this language: r1r3 or r2r3.

r1(r2 + r3):

The first expression is always r1.

It is followed by either r2 or r3.

There are two possibilities for this language: r1r2 or r1r3.

Question

Can a language be expressed by more than one regular expressions, while given that a unique language is generated by that regular expression?

Consider the language, defined over

Σ={a, b}, of words starting with double a and ending in double b then its regular expression may be $aa(a+b)^*bb$

Consider the language, defined over

Σ={a, b} of words starting with a and ending in b OR starting with b and ending in a, then its regular expression may be $a(a+b)^*b+b(a+b)^*a$

Lecture # 5

Deterministic Finite Automata

Also known as Finite state machine, Finite state automata

It represents an abstract machine which is used to represent a regular language

A regular expression can also be represented using Finite Automata

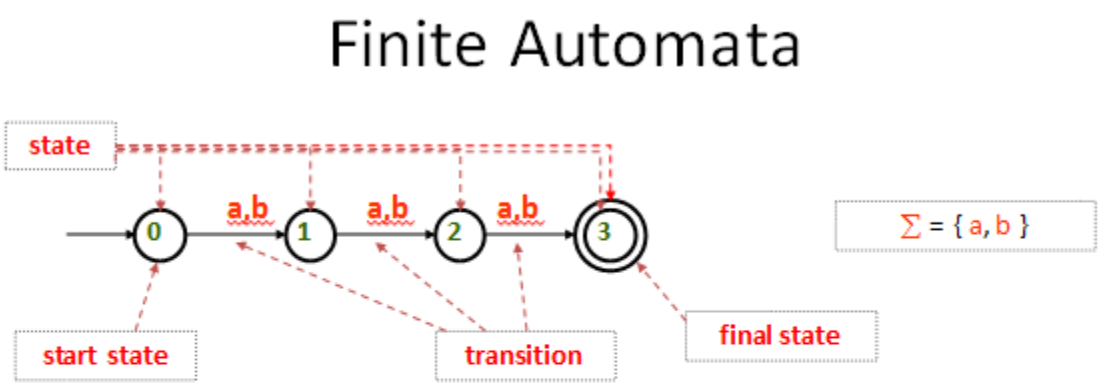
There are two ways to specify an FA: Transition Tables and Directed Graphs.

Graph Representation

Each node (or vertex) represents a state, and the edges (or arcs) connecting the nodes represent the corresponding transitions.

Each state can be labeled.

Finite Automata



- Table Representation
  (continued)
  - An FSA may also be represented with a state-transition table. The table for the above FSA:

State	Input a	Input b
0	1	1
1	2	2
2	3	3

Finite Automata Definition

An FA is defined as follows:-

Finite no of states in which one state must be initial state and more than one or may be none can be the final states.

Sigma Σ provides the input letters from which input strings can be formed.

FA Distinguishing Rule: For each state, there must be an out going transition for each input letter in Sigma Σ.

Σ = {a,b} and states = 0,1,2 where 0 is an initial state and 1 is the final state.

Transitions:

At state 0 reading a,b go to state 1,

At state 1 reading a, b go to state 2

At state 2 reading a, b go to state 2

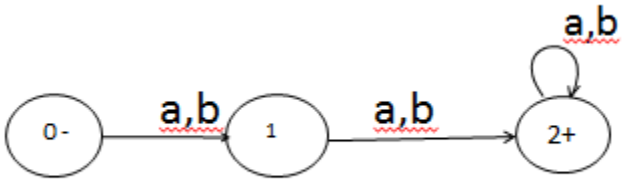
Transition Table Representation

Old state	Input	Next State
0	a	1
0	B	1
1	a	2
1	b	2
2	a	2
2	b	2

Another way of representation…

Old States	New States	
	Reading a	Reading b
0 -	1	1
1	2	2
2 +	2	2

FA directed graph



Example

Σ = {a,b} and states = 0,1,2 where 0 is an initial state and 1 is the final state.

Transitions:

At state 0 reading a go to state 1,

At state 0 reading b go to state 2,

At state 1 reading a,b go to state 2

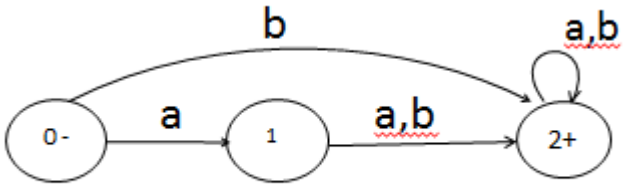
At state 2 reading a, b go to state 2

Transition Table Representation

Old state	Input	Next State
0	a	1
0	b	2
1	a	2
1	b	2
2	a	2
2	b	2

Another way of representation…

Old States	New States	
	Reading a	Reading b
0 -	1	2
1	2	2
2 +	2	2



Σ = {a,b} and states = 0,1,2 where 0 is an initial state and 0 is the final state.

Transitions:

At state 0 reading a go to state 0,

At state 0 reading b go to state 1,

At state 1 reading a go to state 1

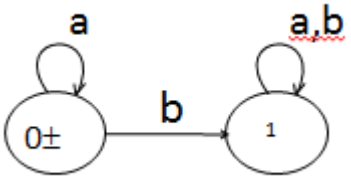
At state 1 reading b go to state 1

Transition Table Representation

Old state	Input	Next State
0	a	0
0	b	1
1	a	1
1	b	1

Another way of representation…

Old States	New States	
	Reading a	Reading b
0 -	0	1
1	1	1



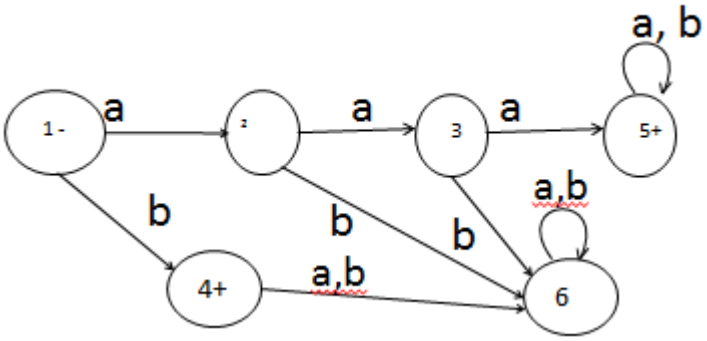
Given an input string, an FA will either accept or reject the input based on the following:

If final state is reached after reading the string, the FA will accept the string

If the final state is not reachable after reading the individual symbols of a string, then FA will reject the string.

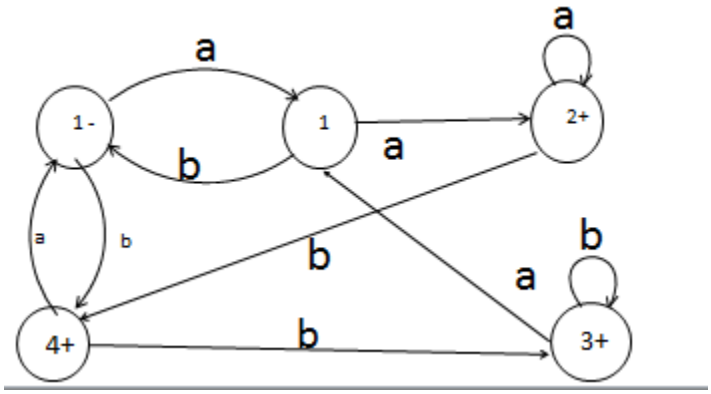
Construct a regular expression and correspondingly an FA for all words in which a appears tripled, if at all.

- **The regular expression is as follows:-**
  - **(aaa+b)***

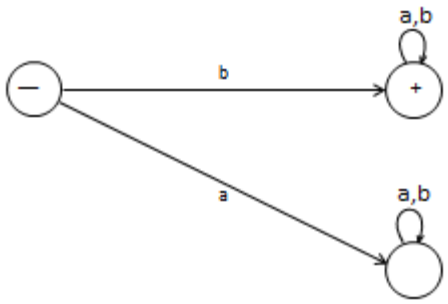


Construct a regular expression and correspondingly an FA for all strings that end in a double letter.

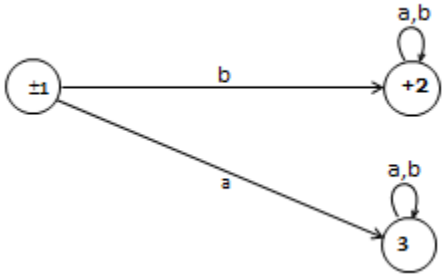
- The regular expression is as follows:-
  - (a+b)*(aa+bb)



L_1 = The language of strings, defined over Σ={a, b}, **beginning with b**



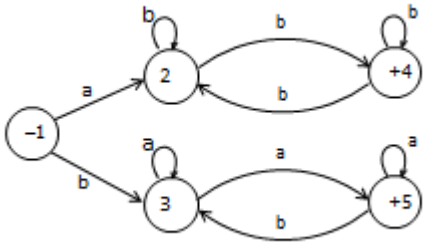
The language of strings, defined over Σ={a, b}, not beginning with a



Language of Strings of length two or more, defined over Σ = {a, b}, beginning with and ending in different letters.

The language L may be expressed by the following regular expression

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



Lecture # 6

JLFAP provides a Hands-on Approach to Formal Languages and Automata.

JLFAP = Java Formal Languages and Automata Package

It is an Instructional tool to learn concepts of Formal Languages and Automata Theory

Topics:

Regular Languages (Finite Automata, Regular Expressions etc.,)

Context-Free Languages and many more…

Lecture # 7

An FA is defined as follows:-

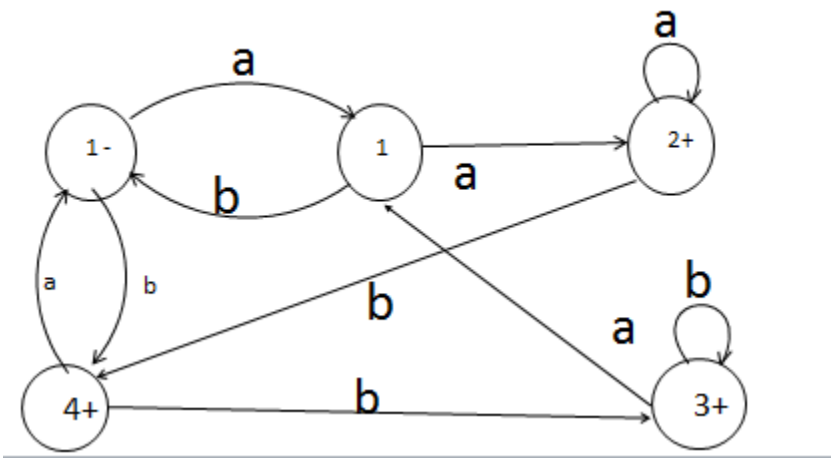
Finite no of states in which one state must be initial state and more than one or may be none can be the final states.

Sigma Σ provides the input letters from which input strings can be formed.

FA Distinguishing Rule: For each state, there must be an out going transition for each input letter in Sigma Σ.

Construct a regular expression and correspondingly an FA for all strings that end in a double letter.

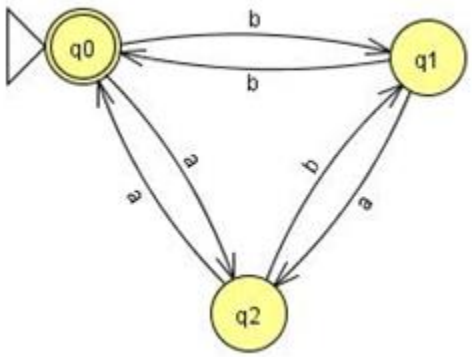
- The regular expression is as follows:-
  - (a+b)*(aa+bb)



Beginning and ending in different letters

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

FA Optionality Behavior



Can and Cannot represent Even-Even Language

Dead or Trap States

Dead states are used to implement the FA **Distinguishing Rule.**

How to use JFLAP to draw dead states

JFLAP Tour (Again)

Even-Even Language in JFLAP

Multiple Input Examples running using JFLAP

Lecture # 8

How to avoid Dead States in FA

Martin method:

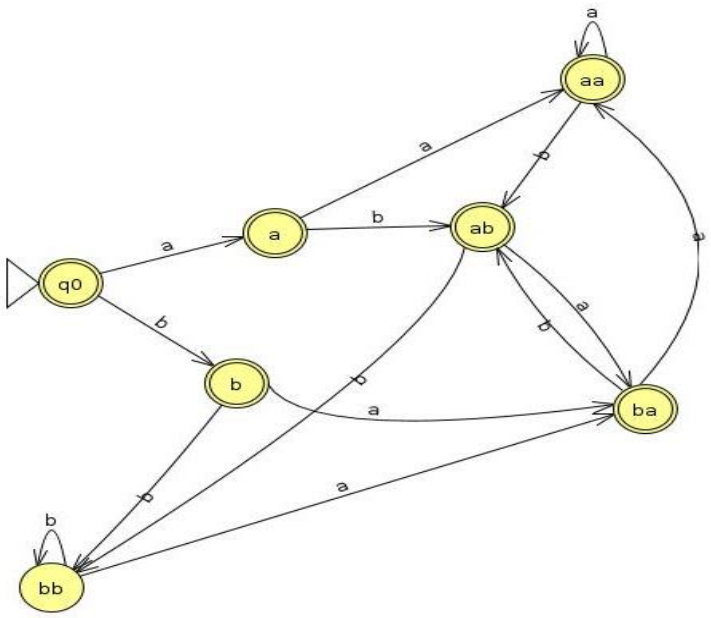
Make each state label, as it progresses, based on the input strings.

Based on the conditions of the Regular expressions or FA, only required states are marked final.

Not every FA can be modeled in this way!

Example for FA that does not end at **bb** only.

RE will be as :-Λ+a+b+(a+b)*(ab+ba+aa)

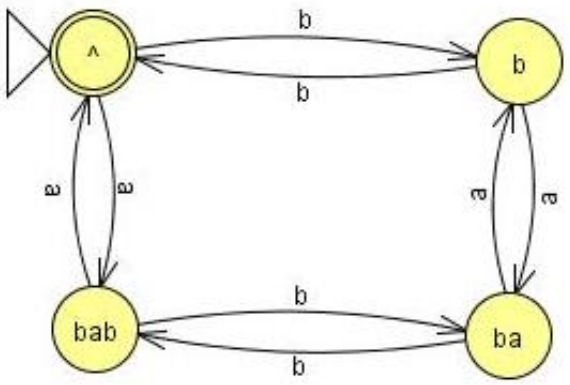


Example for FA that does not end at **aba and abb**. Also the length of each word >= 3

- RE will be as follows:-
- aab+ aaa+ bab+ baa+bbb+bba

Even-Even Example

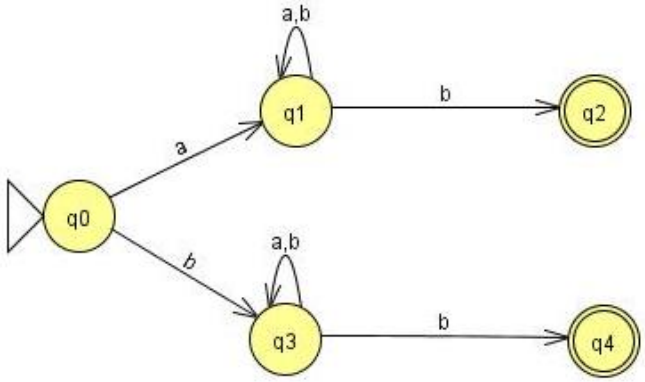
Even-Even Example cannot be modeled using Martin's method.



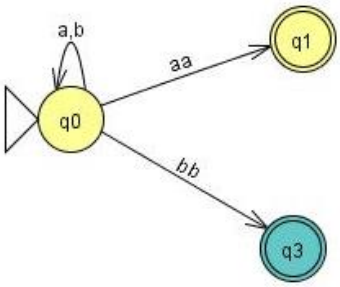
Transition Graphs (TGs) and Generalized Transition Graphs (GTGs)

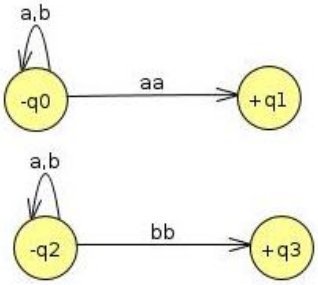
Transition Graphs	Generalized Transition Graphs
Finite number of states	same
Finite set of input strings	same
Finite set of transitions including NULL string	Finite set of transitions including NULL string and transitions can represent Regular expressions

Starting and ending in different letters

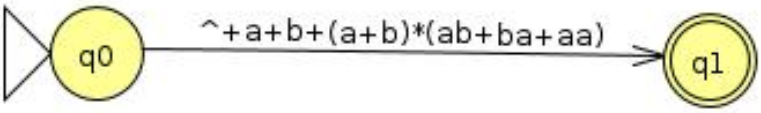


Ends at a double letter





GTG Example



Kleene Theorem

Daniel I Cohen has divided Kleene Theorem in to three parts:

Part I: Every FA is a TG

Part II: Every TG has a regular expression

Every Regular expression can be represented by a Finite Automata

Kleene Theorem Part I

Every FA is a TG as well.

Please refer to Previous Slides.

FA	TG
Single Start State and multiple end states	Multiple State States and multiple end states
Finite set of input symbols	Same

Finite set of transitions	Same
Deterministic	Non-Deterministic
Distinguishing Rule	No such rule

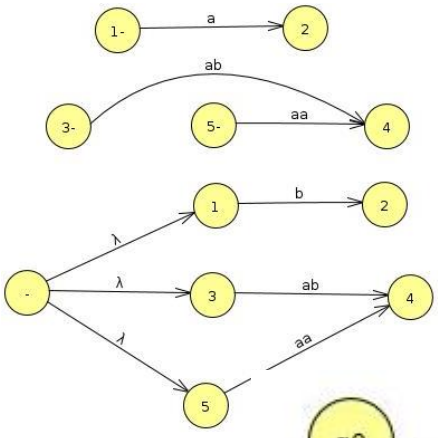
Kleene Theorem Part II

Every TG has a regular expression.

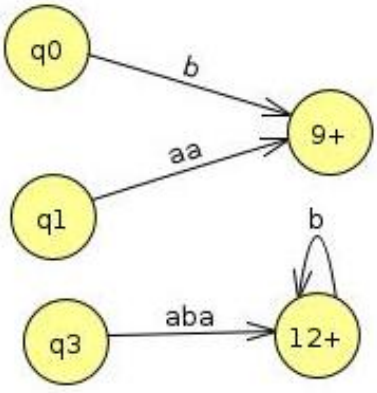
The prove of this Part requires a systematic algorithm through which a TG can be converted to a GTG, in which all transitions are actually regular expressions. Thus, we need to transform a TG to a GTG and eliminate its various states and convert it to a single state or single transition GTG, so that we can get the regular expression associated with the TG.

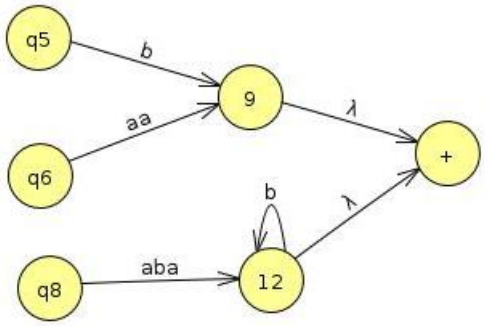
Steps involved in TG to GTG conversion
Examples taken from Daniel I Cohen Book

Step I: More than one Initial states conversion:

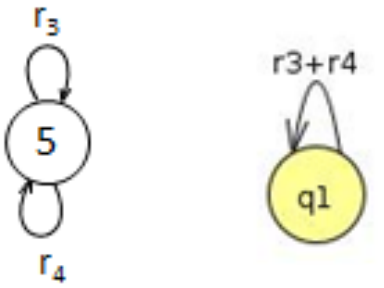


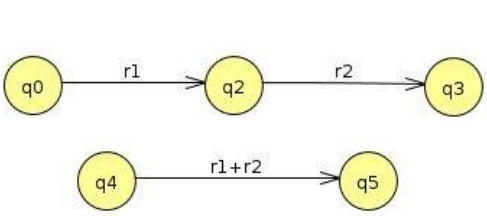
Step II: More than one Final states conversion:

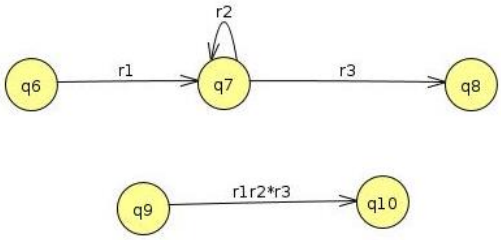


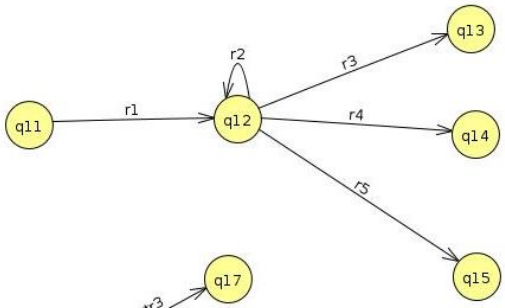


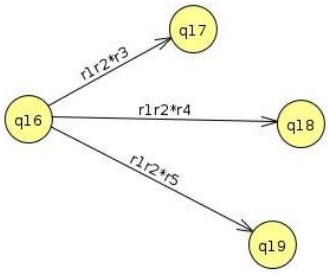
Step III: Multiple Loops conversion:





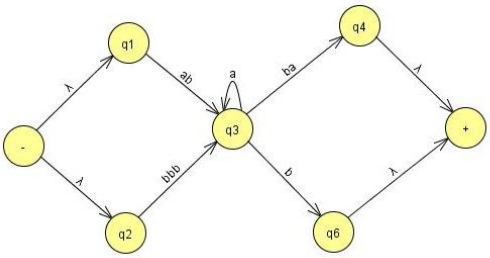


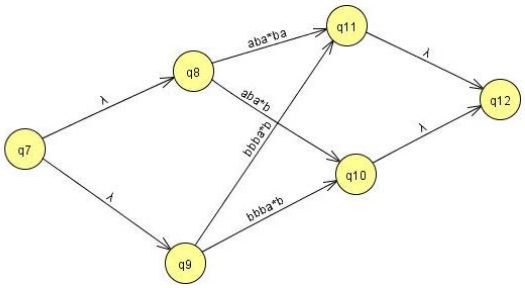


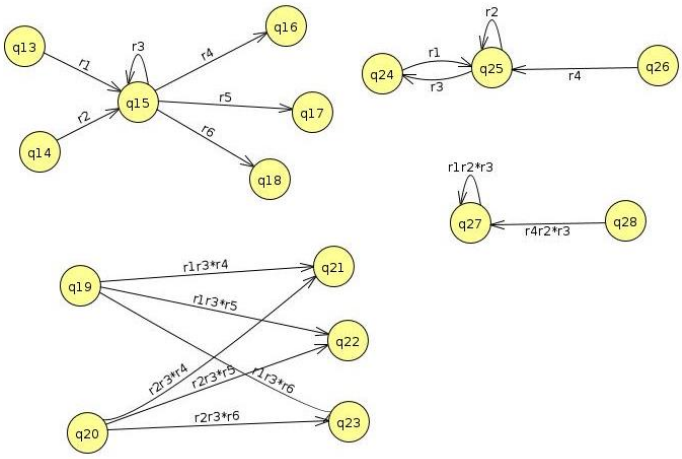


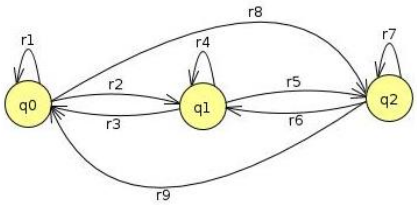
Lecture # 9

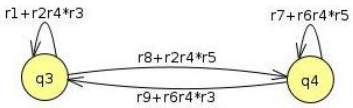
State elimination

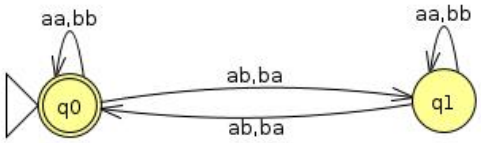












Kleene Part III

Every Regular Expression can be represented by an FA

We already know that a regular expression has a corresponding FA. However, the difficult part is that a regular expression can be combined with other regular expression through union (sum), concatenation and closure of FA. Thus, we need to devise methods for FA1+FA2, FA1FA2, FA1 Closure.

Kleene Theorem Part III (Union)

If r1+r2 represents a regular expression r3, then FA+FA2 represents an FA3 that correspond to r3.

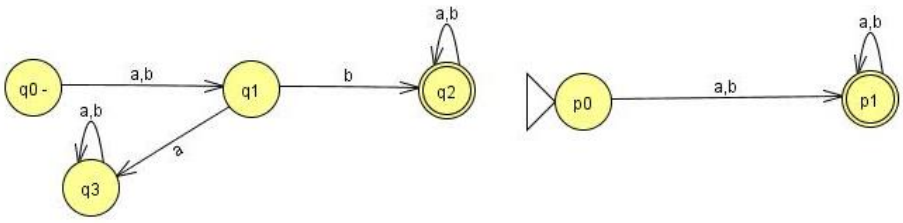
Start by taking both FA's initial state and traversing on each input symbol in the respective FA

Since one initial state is allowed in FA, therefore, only one state can be marked as initial state

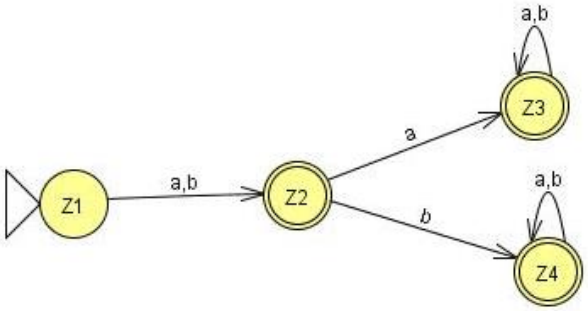
During the process, any state encountered final, the resultant state will be final. This is due to the fact that multiple final states are allowed in FA.

Question

Find FA1 U FA2 for the following:

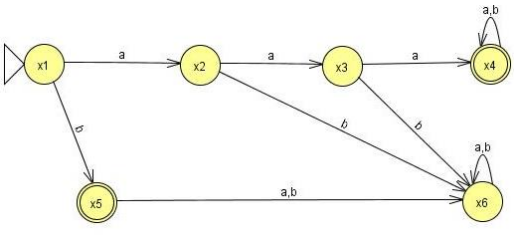


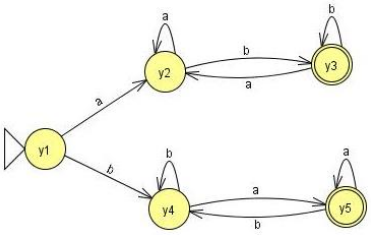
Old States	Reading at a	Reading at b
z1-≡(q0,p0)	(q1,p1)≡z2	(q1,p1)≡z2
z2+≡(q1,p1)	(q3,p1)≡z3	(q2,p1)≡z4
z3+≡(q3,p1)	(q3,p1)≡z3	(q3,p1)≡z3
z4+≡(q2,p1)	(q2,p1)≡z4	(q2,p1)≡z4



Question

Find FA1 U FA2 for the following:





Lecture # 10

Repeat – Kleene Part III

Every Regular Expression can be represented by an FA

We already know that a regular expression has a corresponding FA. However, the difficult part is that a regular expression can be combined with other regular expression through union (sum), concatenation and closure of FA. Thus, we need to devise methods for FA1+FA2, FA1FA2, FA1 Closure.

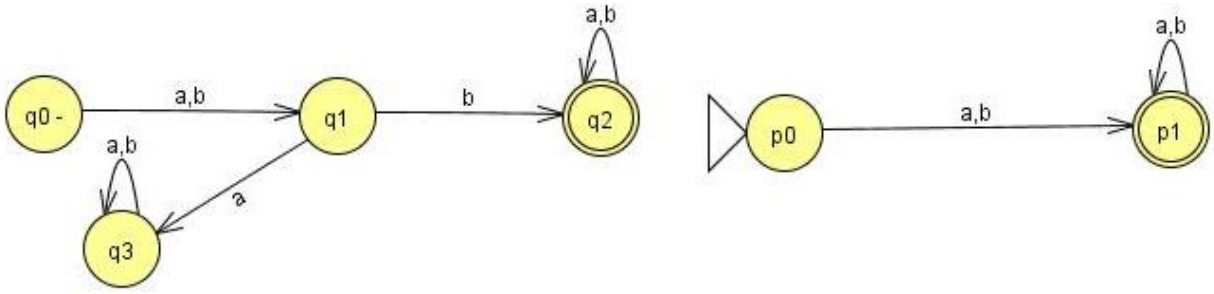
Repeat – Kleene Theorem Part III (Union)

If r1+r2 represents a regular expression r3, then FA+FA2 represents an FA3 that correspond to r3.

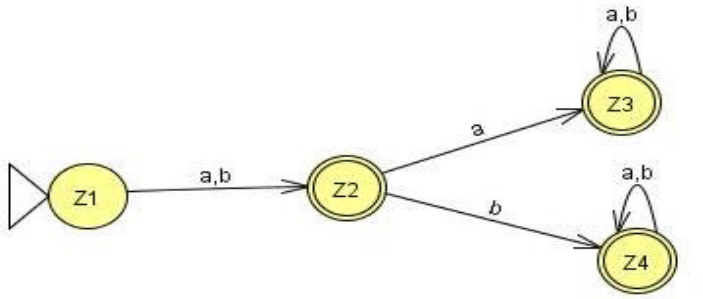
Start by taking both FA's initial state and traversing on each input symbol in the respective FA

Since one initial state is allowed in FA, therefore, only one state can be marked as initial state

During the process, any state encountered final, the resultant state will be final. This is due to the fact that multiple final states are allowed in FA.



Old States	Reading at a	Reading at b
z1-≡(q0,p0)	(q1,p1)≡z2	(q1,p1)≡z2
z2+≡(q1,p1)	(q3,p1)≡z3	(q2,p1)≡z4
z3+≡(q3,p1)	(q3,p1)≡z3	(q3,p1)≡z3
z4+≡(q2,p1)	(q2,p1)≡z4	(q2,p1)≡z4



Kleene Theorem Part III (Concatenation)

If r1r2 represents a regular expression r3, then FA1FA2 represents an FA3 that should correspond to r3.

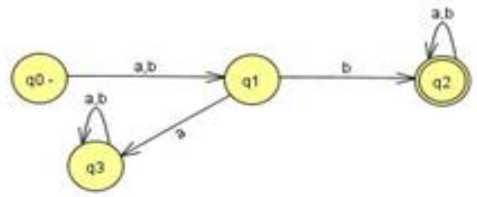
Start by taking the first FA's initial state and traversing on each input symbol in the respective FA.

Since one initial state is allowed in FA, therefore, only one state can be marked as initial state

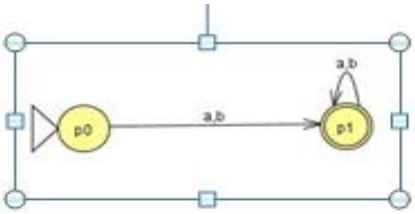
During the process, any state encountered final of the second FA only, the resultant state will be final. Further, the second FA will be concatenated through first FA's initial state.

However, if the final state of the second FA is encountered, it will not be combined with the first FA.

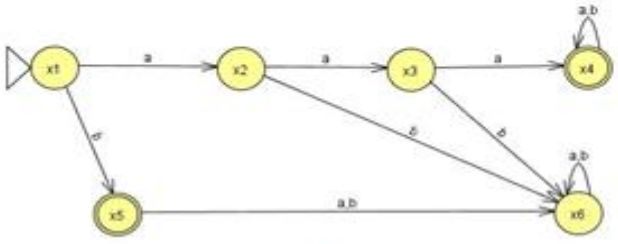
3 Questions for Concatenation



FA1: (a+b)b(a+b)*



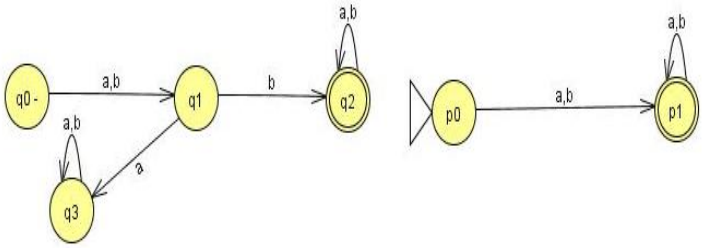
FA2: (a+b)+



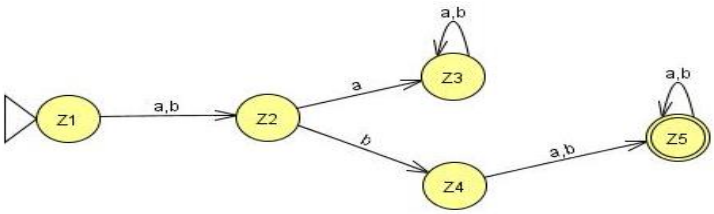
FA3: (aaa+b)*

Question (concatenation)

Find FA1FA2 for the following:



Old States	Reading at a	Reading at b
z1-≡q0	q1≡z2	q1≡z2
z2≡q1	q3≡z3	(q2,p0)≡z4
z3≡q3	q3≡z3	q3≡z3
z4 ≡ (q2,p0)	(q2,p0,p1)≡z5	(q2,p0,p1)≡z5
z5+≡(q2,p0,p1)	(q2,p0,p1,p1)= (q2,p0,p1)≡z5	(q2,p0,p1,p1)= (q2,p0,p1)≡z5

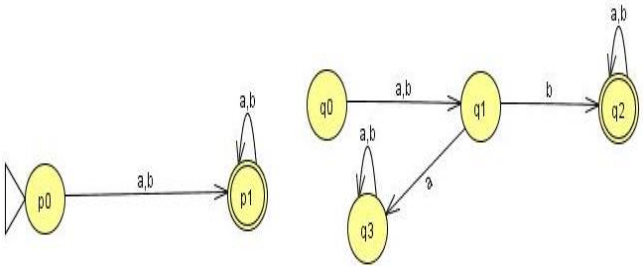


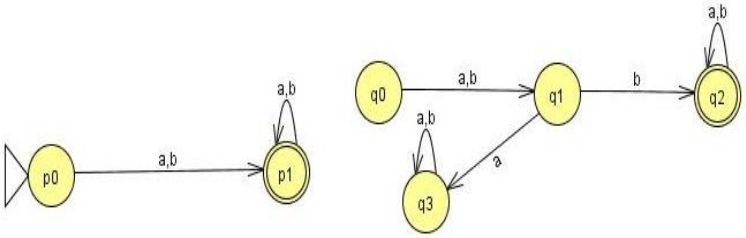
Verification: $(a+b)b(a+b)^*(a+b)^+$

bba,

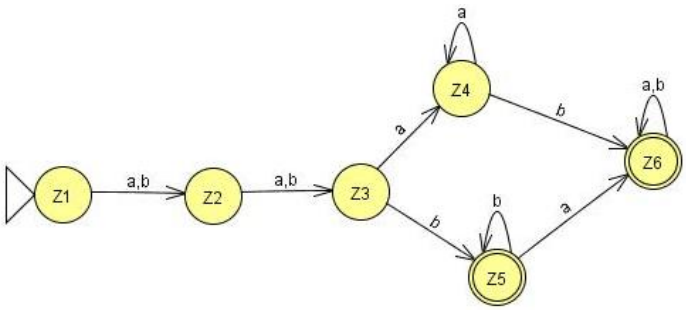
Question (concatenation)

Find FA2FA1 for the following:





Old States	Reading at a	Reading at b
z1-≡p0	(p1, q0)≡ z2	(p1, q0)≡ z2
z2≡(p1, q0)	(p1,q0,q1)≡ z3	(p1,q0,q1)≡ z3
z3 ≡ (p1,q0,q1)	(p1,q0,q1,q3)≡ z4	(p1,q0,q1,q2)≡z5
z4≡ (p1,q0,q1,q3)	(p1,q0,q1,q3,q3)= (p1,q0,q1,q3)≡z4	(p1,q0,q1,q2,q3)≡z6
z5+≡(p1,q0,q1,q2)	(p1,q0,q1,q2,q3)≡z6	(p1,q0,q1,q2,q2)= (p1,q0,q1,q2)≡z5
z6+≡(p1,q0,q1,q2,q3)	(p1,q0,q1,q3,q2,q3)= (p1,q0,q1,q2,q3)z6	(p1,q0,q1,q2,q2,q3)= (p1,q0,q1,q2,q3)≡z6

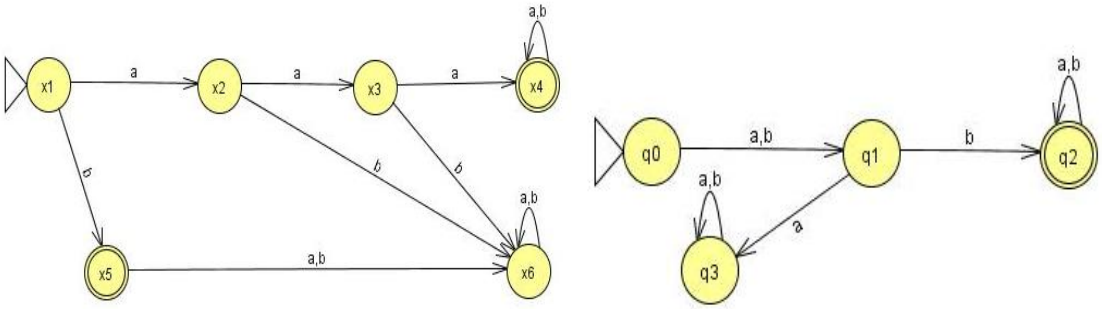


Verification: $(a+b)^{+}(a+b)b(a+b)^{*}$

aabaaa

Question (concatenation)

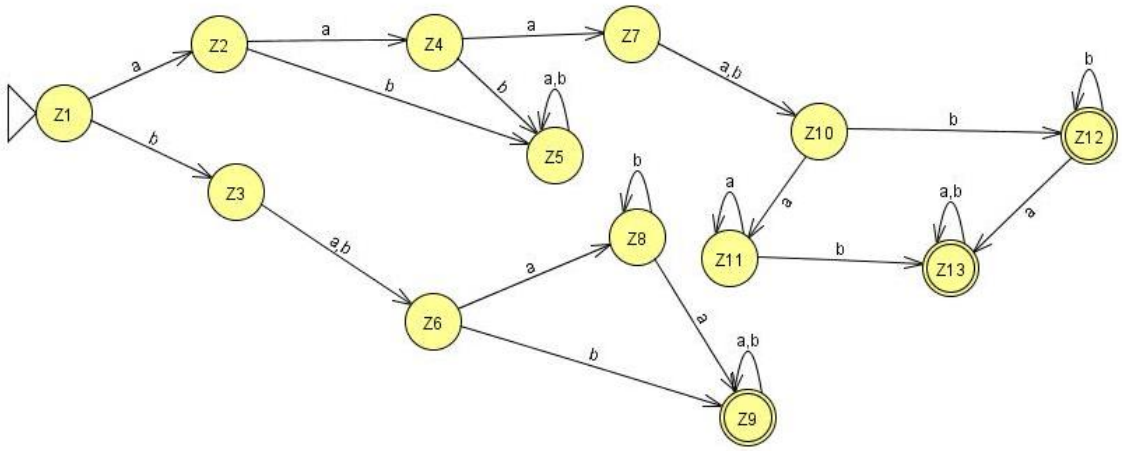
Find FA3FA1 for the following:



Old States	Reading at a	Reading at b
z1- ≡x1	x2≡ z2	(x5,q0)≡z3
z2 ≡x2	x3≡ z4	x6≡ z5
z3 ≡(x5,q0)	(x6,q1)≡z6	(x6,q1)≡z6
z4 ≡x3	(x4,q0)≡z7	x6≡ z5
z5 ≡x6	x6≡z5	x6≡z5
z6 ≡(x6,q1)	(x6,q3)≡ z8	(x6,q2)≡ z9

z7≡(x4,q0)	(x4,q0,q1)≡ z10	(x4,q0,q1)≡ z10
z8 ≡(x6,q3)	(x6,q3)≡ z8	(x6,q3)≡ z8

z9+≡(x6,q2)	**(x6,q2)≡ z9**	**(x6,q2)≡ z9**
z10 ≡(x4,q0,q1)	(x4,q0,q1,q3)≡ z11	(x4,q0,q1,q2)≡ z12
z11≡(x4,q0,q1,q3)	(x4,q0,q1,q3,q3)= (x4,q0,q1,q3)≡ z11	(x4,q0,q1,q2,q3)≡ z13
z12+≡(x4,q0,q1,q2)	(x4,q0,q1,q2,q3)≡ z13	(x4,q0,q1,q2,q2)= (x4,q0,q1,q2)≡z12
z13+≡(x4,q0,q1,q2,q3)	(x4,q0,q1,q3,q2,q3)= (x4,q0,q1,q2,q3)≡z13	(x4,q0,q1,q2,q2,q3)= (x4,q0,q1,q2,q3)≡z13

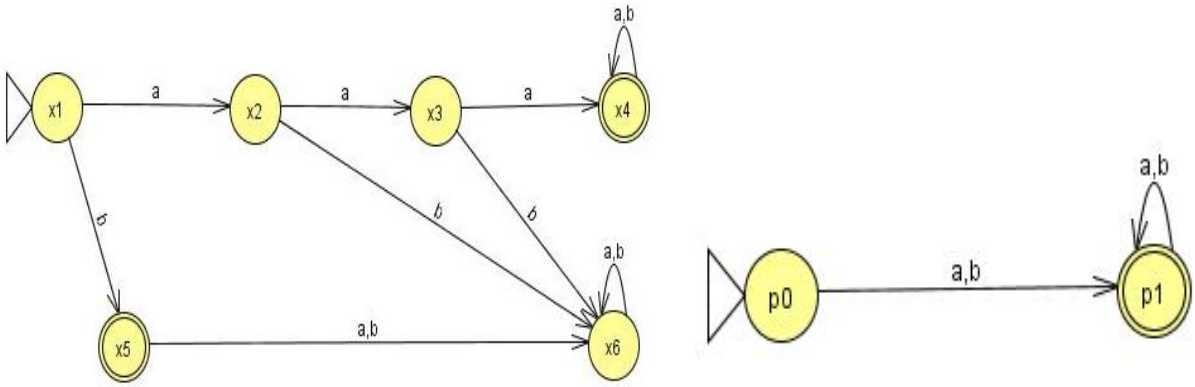


Verification: $(aaa+b)^{+}(a+b)b(a+b)^{*}$

bab

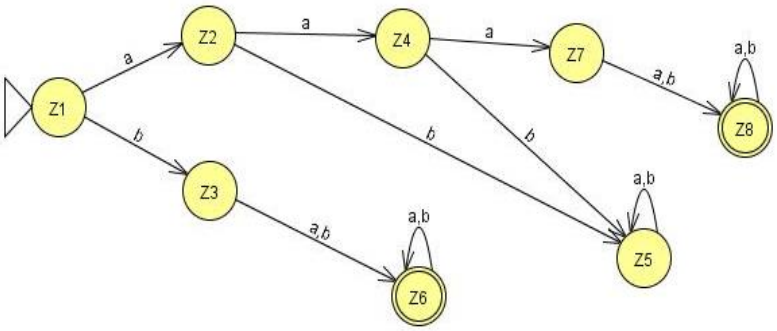
Question (concatenation)

Find FA3FA2 for the following:



Old States	Reading at a	Reading at b
z1- ≡x1	x2≡ z2	(x5,p0)≡ z3
z2 ≡x2	x3≡ z4	x6≡ z5
z3≡(x5,p0)	(x6,p1)≡z6	(x6,p1)≡z6
z4 ≡x3	(x4,p0)≡z7	x6≡z5
z5 ≡x6	x6≡z5	x6≡z5
z6+≡(x6,p1)	(x6,p1)≡z6	(x6,p1)≡z6

z7≡(x4,p0)	(x4,p0,p1)≡ z8	(x4,p0,p1)≡ z8
z8+≡(x4,p0,p1)	(x4,p0,p1,p1)= (x4,p0,p1)≡z8	(x4,p0,p1,p1)= (x4,p0,p1)≡z8



Verification: $(aaa+b)^{+}(a+b)^{+}$

Aaabab

Lecture # 11

Kleene Theorem Part III (Concatenation) – Repeat

If r1r2 represents a regular expression r3, then FA1FA2 represents an FA3 that should correspond to r3.

Start by taking the first FA's initial state and traversing on each input symbol in the respective FA.

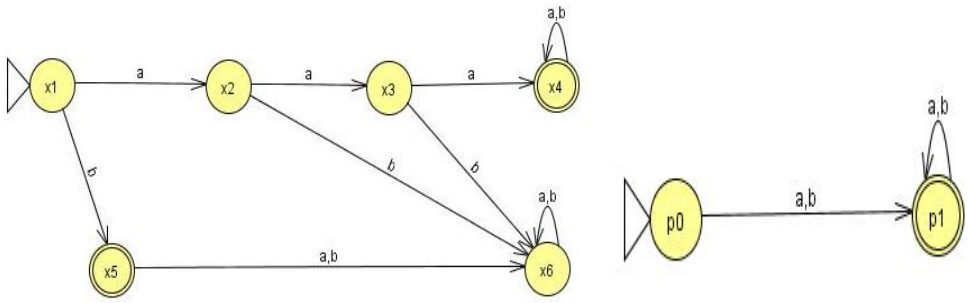
Since one initial state is allowed in FA, therefore, only one state can be marked as initial state

During the process, any state encountered final of the second FA only, the resultant state will be final. Further, the second FA will be concatenated through first FA's initial state.

However, if the final state of the second FA is encountered, it will not be combined with the first FA.

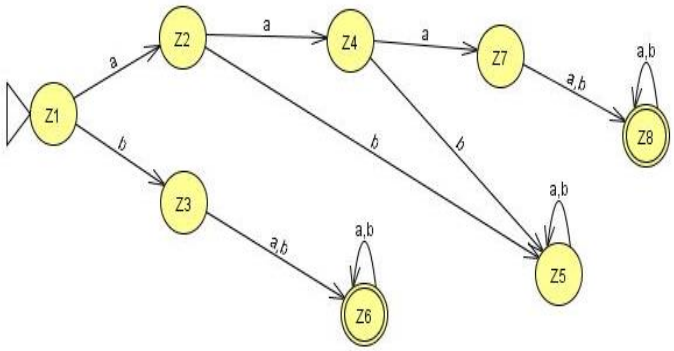
Question (concatenation)

Find FA3FA2 for the following:



Old States	Reading at a	Reading at b
z1- ≡x1	x2≡ z2	(x5,p0)≡ z3
z2 ≡x2	x3≡ z4	x6≡ z5
z3≡(x5,p0)	(x6,p1)≡z6	(x6,p1)≡z6
z4 ≡x3	(x4,p0)≡z7	x6≡z5
z5 ≡x6	x6≡z5	x6≡z5
z6+≡(x6,p1)	(x6,p1)≡z6	(x6,p1)≡z6

z7≡(x4,p0)	(x4,p0,p1)≡ z8	(x4,p0,p1)≡ z8
z8+≡(x4,p0,p1)	(x4,p0,p1,p1)= (x4,p0,p1)≡z8	(x4,p0,p1,p1)= (x4,p0,p1)≡z8



Kleene Theorem Part III (Closure)

If r1 represents a regular expression and r1* the closure of the r1. Similarly, if FA1 corresponds to r1 then FA1* should correspond to closure of r1.

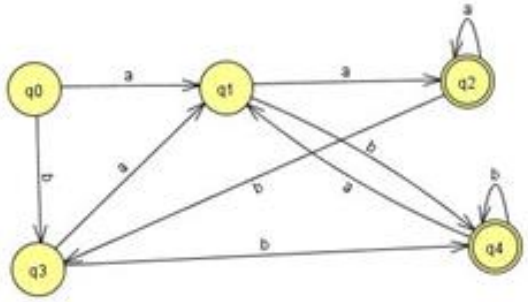
Start by taking the first FA's and traversing on each input symbol in the respective FA.

Since one initial state is allowed in FA, therefore, only one state can be marked as initial state. Further, the initial state is marked final by default, in order to accommodate NULL string.

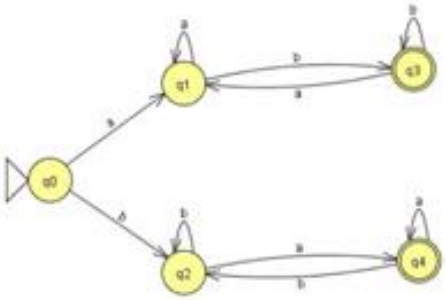
During the process, any state found final, the resultant state will be final. Further, the FA's initial state will be concatenated through its final state. The new state will be marked as final.

Note that, the final state of the FA will always be combined with the initial state of the FA.

2 Questions for Closure

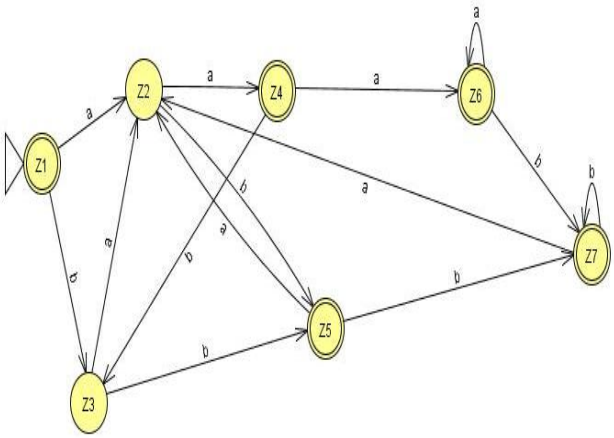


FA1: (a+b)*(aa+bb)



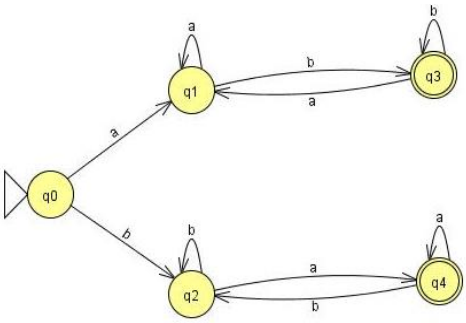
FA2: a(a+b)*b + b(a+b)*a

Old States	Reading at a	Reading at b
z1±≡q0	q1≡z2	q3≡z3
z2 ≡q1	(q2, q0)≡z4	(q4, q0)≡z5
z3 ≡q3	q1≡z2	(q4, q0)≡z5
z4+≡ (q2, q0)	(q2, q0,q1)≡z6	(q3,q3)≡ q3≡z3
z5+≡ (q4, q0)	(q1,q1)≡ q1≡z2	(q4,q0,q3)≡z7
z6+≡ (q2, q0,q1)	(q2, q0,q1,q2, q0)≡(q2, q0,q1)≡z6	(q3,q3,q4,q0)≡(q4,q0,q3)≡z7
z7+≡ (q4,q0,q3)	(q1,q1q1)≡q1≡z2	(q4,q0,q3,q4,q0,)≡(q4,q0,q3)≡z7



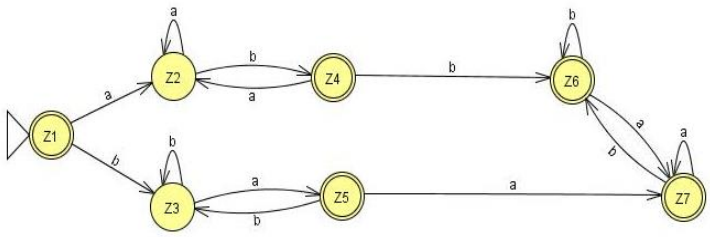
Verification: ((a+b)*(aa+bb))*

Aaaa



Old States	Reading at a	Reading at b
z1±≡q0	q1≡z2	(q3, q0)≡z3
z2 ≡q1	(q2, q0)≡z4	(q4, q0)≡z5
z3 ≡(q3, q0)	(q1,q1)≡q1≡z2	(q4, q0,q1)≡z6
z4+≡ (q2, q0)	(q2, q0,q1)≡z6	(q3,q3,q0)= (q3,q0)≡z3
z5+≡ (q4, q0)	(q1,q4,q0)≡z7	(q2,q2)≡q2≡z3
z6+≡ (q3, q0,q2)	(q1,q1,q4,q0)≡(q1,q4,q0)≡z7	(q3, q0,q2,q2)≡ (q3, q0,q2)≡z6

z7+≡ (q1,q4,q0)	(q1,q4,q0,q1)≡(q1,q4,q0)≡z7	(q3, q0,q2,q2)≡ (q3, q0,q2)≡z6

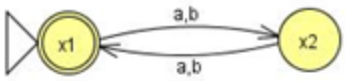


Verification: (a(a+b)*b + b(a+b)*a)*

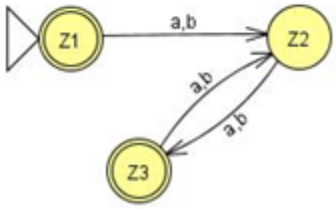
abba,baab,aaaabbbbba,

FA Closure Special Note

One special requirement for FA's having an initial state which has a loop or an in coming transition in the closure results in two different states: One non-final state and other one final. See the Next Example…



Final Z1≡ x1	x2≡Z2	x2≡Z2
Z2≡x2	Non final x1≡Z3	Non final x1≡Z3
Z3≡x1	x2≡Z2	x2≡Z2

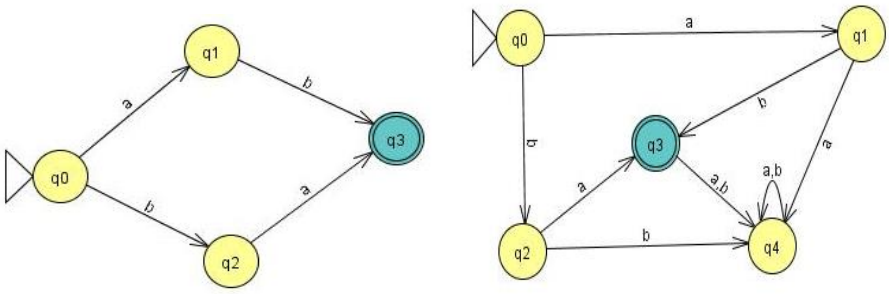


NFA & FA at a glance..

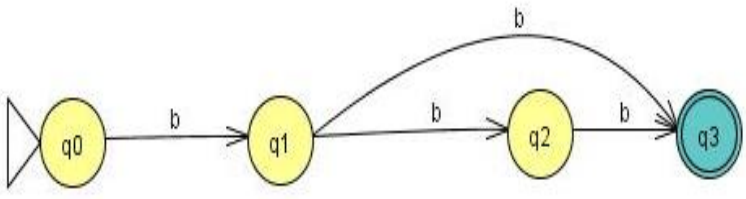
FA	NFA
Single Start State and multiple end states (may be none)	Single Start State and multiple end states (may be none)
Finite set of input symbols	Same
Finite set of transitions	Same
Deterministic	Non-Deterministic
Distinguishing Rule	No such rule

Example

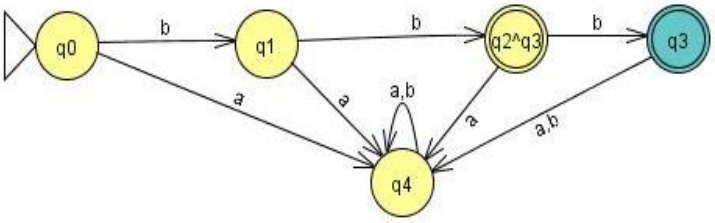
Consider the following NFA

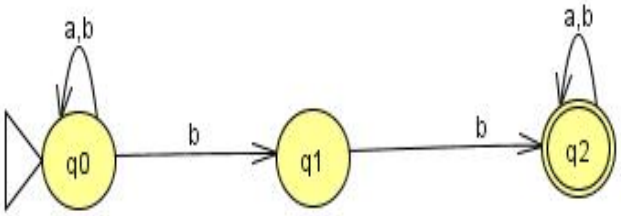


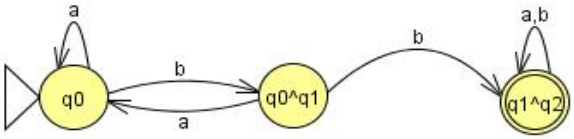
A simple NFA that accepts the language of strings defined over □={a,b}, **consists of bb and bbb**



The above NFA can be converted to the following FA

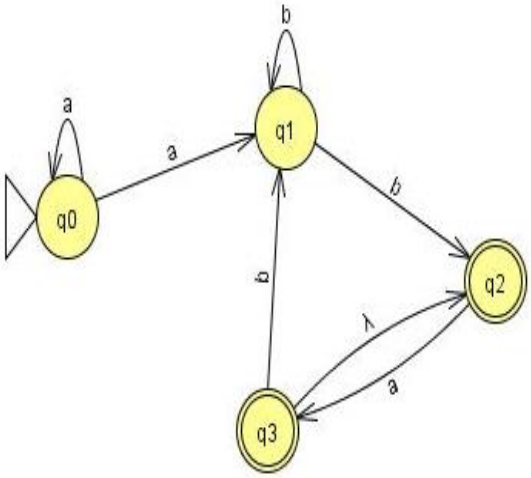

q0
b
q1
b
q2^q3
b
q3
a
a
a,b
a
a,b
q4

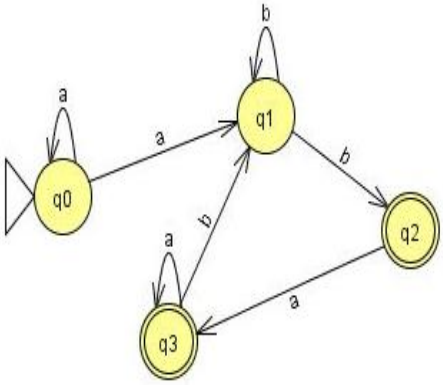

a,b
a,b
q0
b
q1
b
q2


a
a,b
b
b
q0
q0^q1
q1^q2
a

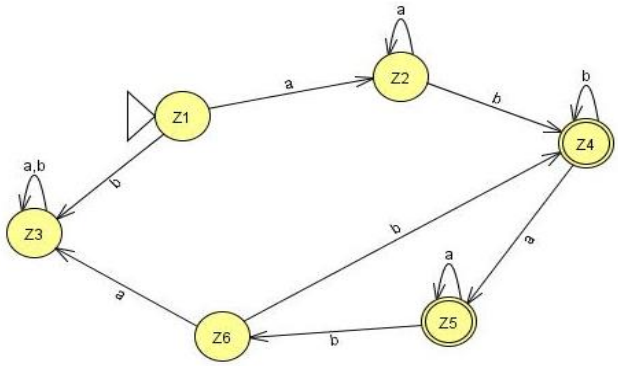
Lecture # 12

NFA with Null Transition

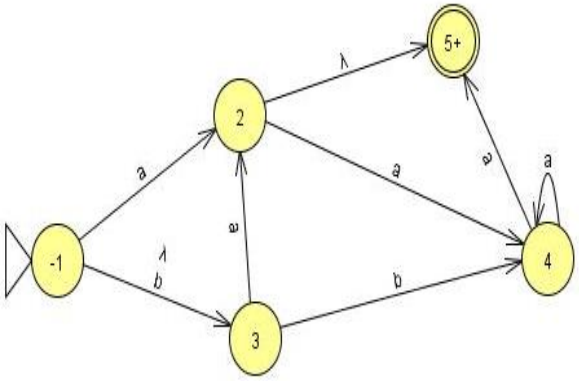


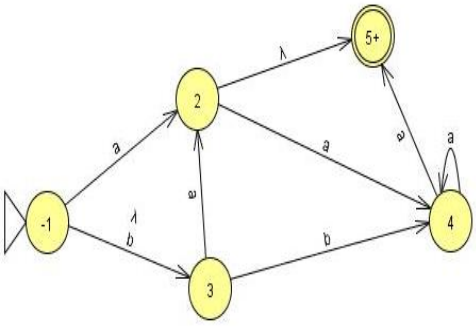


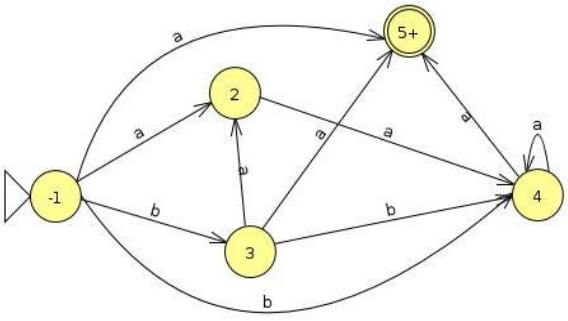
	a	b
$z_1 \equiv q_o$	$(q_o, q_1) \equiv z_2$	$\phi \equiv z_3$
$z_2 \equiv (q_o, q_1)$	$(q_o, q_1) \equiv z_2$	$(q_1, q_2) \equiv z_4$
$z_4^+ \equiv (q_1, q_2)$	$q_3 \equiv z_5$	$(q_1, q_2) \equiv z_4$
$z_5^+ \equiv q_3$	$q_3 \equiv z_5$	$q_1 \equiv z_6$
$z_6 \equiv q_1$	$\phi \equiv z_3$	$(q_1, q_2) \equiv z_4$

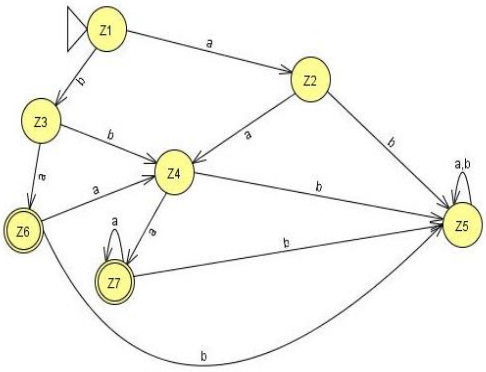


Convert the following NFA with NULL transition to FA.



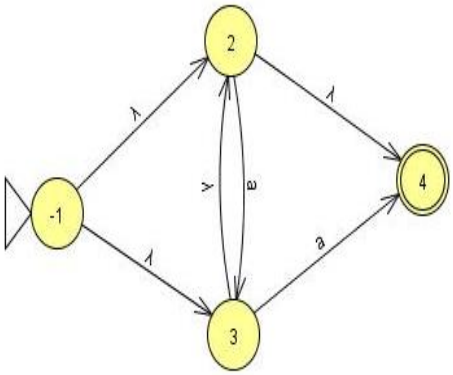


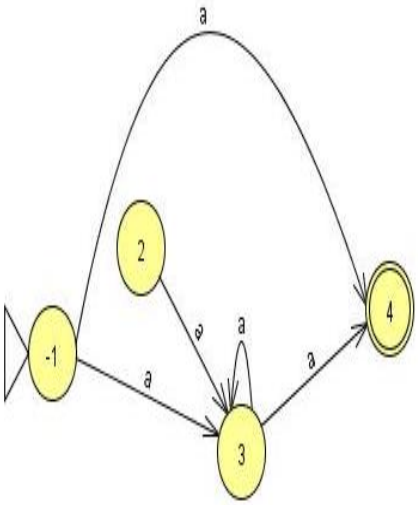


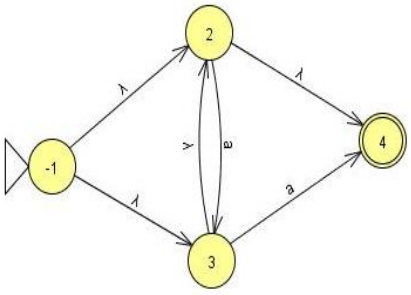


	a	b
$z_1^- \equiv 1$	(2,5) ≡ z_2	3,4 ≡ z_3
$Z_2^+ \equiv (2,5)$	4 ≡ z	$\phi \equiv Z_5$
$Z_3 \equiv (3,4)$	(2,5,4) ≡ z_6	4 ≡ z_4
$Z_4 \equiv 4$	(4,5) ≡ z_7	$\phi \equiv Z_5$
$Z_5 \equiv \psi$	Z_5	Z_5
$Z_6^+ \equiv (2,5,4)$	(4,4) ≡ z_4	$\phi \equiv Z_5$
$Z_7^+ \equiv (4,5)$	(4,5) ≡ z_7	$\phi \equiv Z_5$

Convert the following NFA with NULL transition to FA.

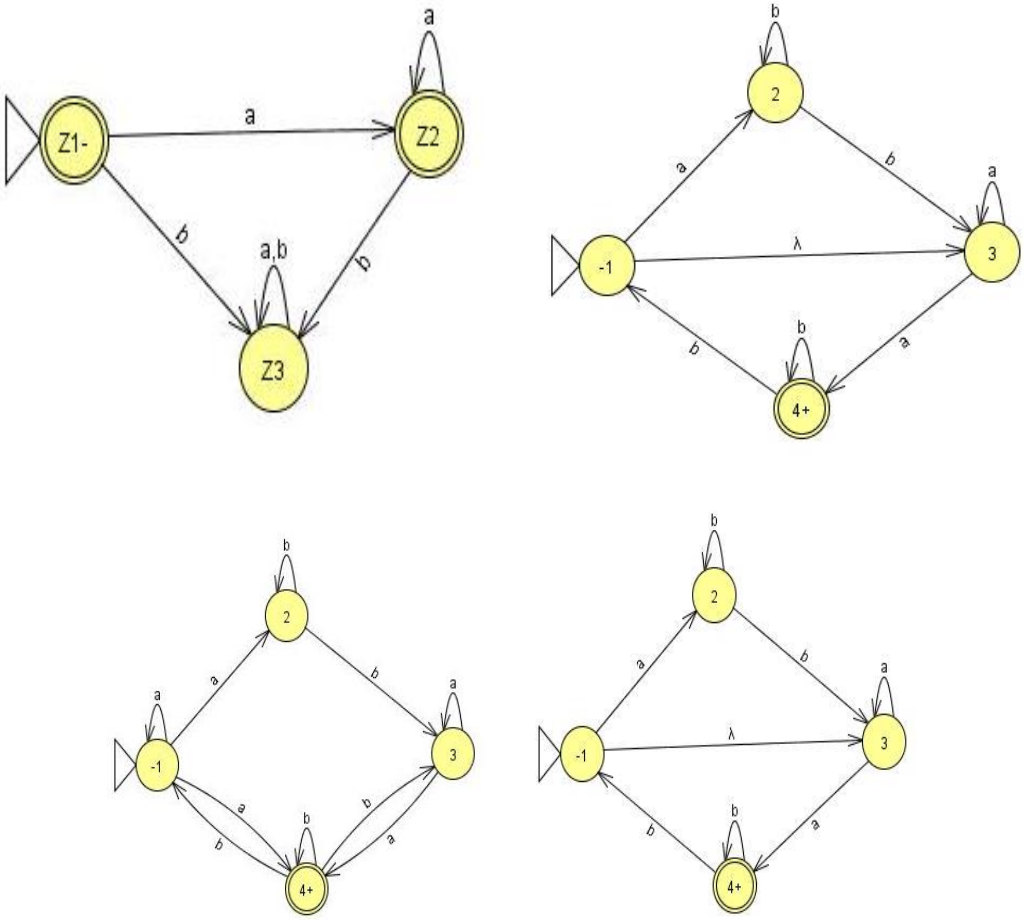




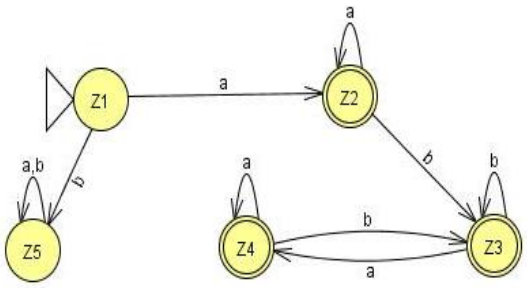


	a	b
$z_1^+ \equiv 1$	(3,4) ≡ z_2	$\phi \equiv Z_3$
$Z_2^+ \equiv (3,4)$	(3,4) ≡ z_2	$\phi \equiv Z_3$
$Z_3 \equiv \psi$	Z_3	Z_3

Convert the following NFA with NULL transition to FA.



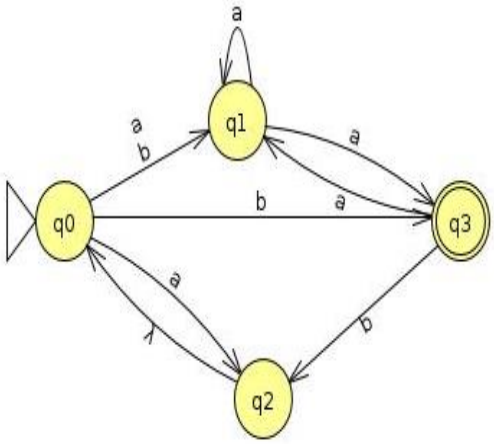
	a	b
$z_1 \equiv 1$	$(1,2,4) \equiv z_2$	$\phi \equiv Z_5$
$Z_2^+ \equiv (1,2,4)$	$(1,2,4) \equiv z_2$	$(1,3,4) \equiv z_3$
$Z_3^+ \equiv (1,3,4)$	$(1,2,4,3) \equiv z_4$	$(1,3,4) \equiv z_3$
$Z_4^+ \equiv (1,2,3,4)$	$(1,2,4,3) \equiv z_4$	$(1,3,4) \equiv z_3$



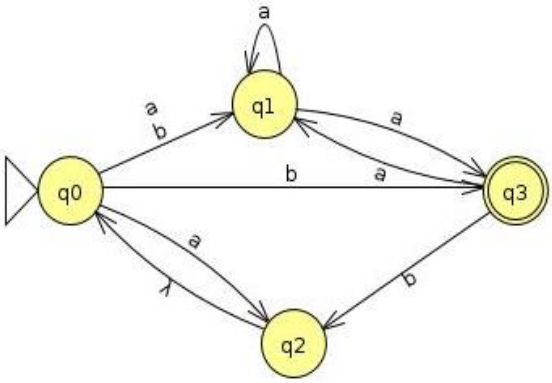
	a	b
$z_1 \equiv 1$	$(1,2,4) \equiv z_2$	$\phi \equiv Z_5$
$Z_2^+ \equiv (1,2,4)$	$(1,2,4) \equiv z_2$	$(1,3,4) \equiv z_3$
$Z_3^+ \equiv (1,3,4)$	$(1,2,4,3) \equiv z_4$	$(1,3,4) \equiv z_3$
$Z_4^+ \equiv (1,2,3,4)$	$(1,2,4,3) \equiv z_4$	$(1,3,4) \equiv z_3$

Lecture # 13

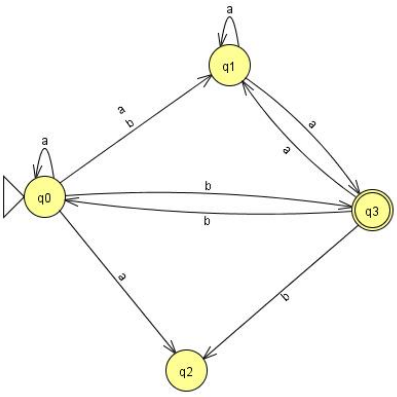
Convert the following NFA with NULL transition to FA.

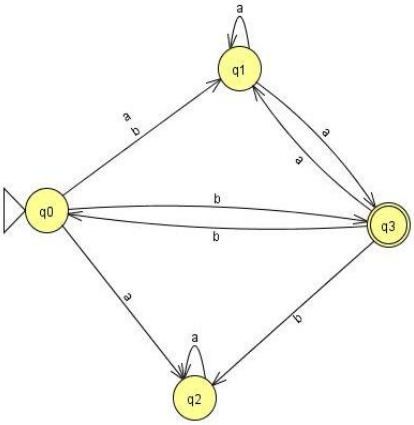


Question :



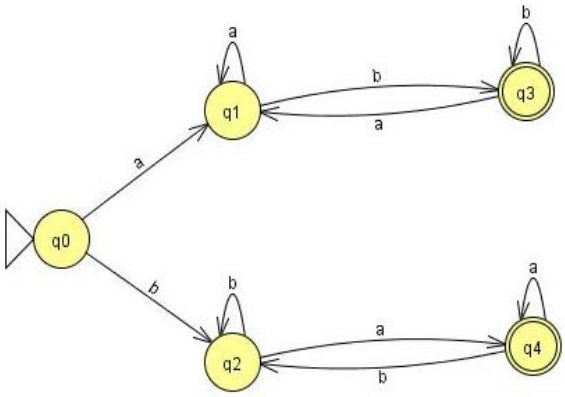
Which conversion from NFA-with NULL to NFA is OK?



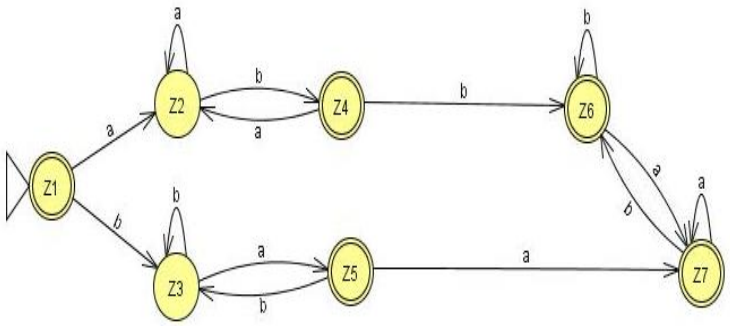


	a	b

$q_o = z_1$	$(q_o, q_1, q_2) = Z_2$	$(q_1, q_3) = Z_3$
$Z_2 = (q_o, q_1, q_2)$	$(q_o, q_1, q_2, q_3) = Z_4$	$(q_1, q_3) = Z_3$
$Z_3^+ = (q_1, q_3)$	$(q_1, q_3) = Z_3$	$(q_0, q_2) = Z_5$
$Z_4^+ = (q_o, q_1, q_2, q_3)$	$(q_o, q_1, q_2, q_3) = Z_4$	$(q_1, q_3, q_2, q_0) = Z_4$
$Z_5 = (q_o, q_2)$	$(q_o, q_1, q_2) = Z_2$	$(q_1, q_3) = Z_3$



Old States	Reading at a	Reading at b
z1±≡q0	q1≡z2	q2≡z3
z2 ≡q1	q1≡z2	(q3, q0)≡z4
z3 ≡q2	(q4, q0)≡z5	q2≡z3
z4+≡ (q3, q0)	(q1,q1)≡q1≡z2	(q3, q0,q2)≡z6
z5+≡ (q4, q0)	(q1,q4,q0)≡z7	(q2,q2)≡q2≡z3
z6+≡ (q3, q0,q2)	(q1,q1,q4,q0)≡(q1,q4,q0)≡z7	(q3, q0,q2,q2)≡ (q3, q0,q2)≡z6
z7+≡ (q1,q4,q0)	(q1,q4,q0,q1)≡(q1,q4,q0)≡z7	(q3, q0,q2,q2)≡ (q3, q0,q2)≡z6



Verification: (a(a+b)*b + b(a+b)*a)*

abba,baab,aaaabbbbba,

Finite Automata with output

In Finite Automata, the input string represents the input data to a computer program. Reading the input letters is very much similar to how a computer program performs various instructions.

The concept of states tell us that what we are printing and what we have printed so far.

Our goal in this chapter is to prove that the mathematical models that we have studied so far can be represented as machines.

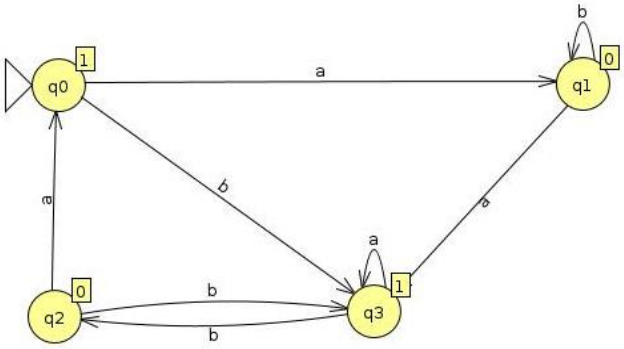
We will study two machines created by G.H Mealy (1955) and by E.F Moore (1956).

Initially both models were intended for sequential circuit design.

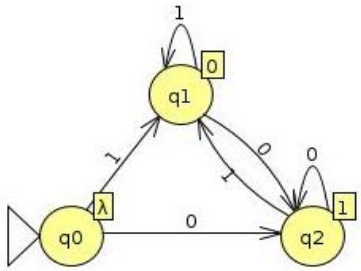
Moore machine Definition

- **A Moore machine consists of the following**
- **A finite set of states q_0, q_1, q_2, … where q_0 is the initial state.**
- **An alphabet of letters Σ = {a,b,c,…} from which the input strings are formed.**
- **An alphabet Γ={x,y,z,…} of output characters from which output strings are generated.**
- **A transition table that shows for each state and each input letter what state is entered the next.**
- **No final state, so no question of acceptance of input strings.**

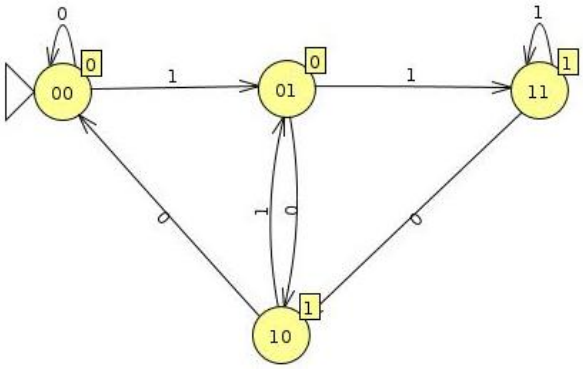
Moore Machine with JFLAP



Example: Moore NOT machine



Example: Divide an Input in to half



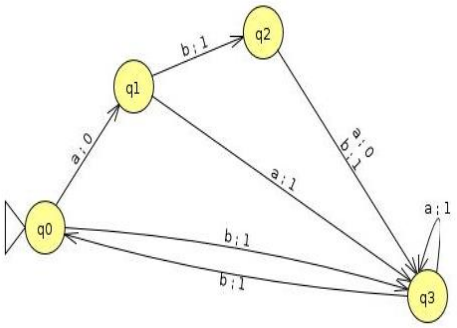
Mealy machine

A Mealy machine consists of the following

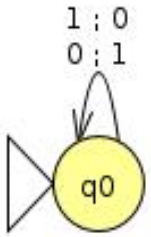
- A finite set of states q_0, q_1, q_2, … where q_0 is the initial state.

- An alphabet of letters Σ = {a,b,c,…} from which the input strings are formed.
- An alphabet Γ={x,y,z,…} of output characters from which output strings are generated.

Example Mealy machine



Complementing Mealy Machine



Lecture # 14

Incrementing Mealy Machine

Consider the following additions

a) 100101100 + 1 = 100101101

b) 1001111111 + 1 = 1010000000

If the right most bit is 0 then…

If the right most bit is 1 then…

Incrementing Mealy Machine

The machine will have three states: start, owe-carry (OC), and no-carry(NC). Owe-carry state will output 0 when two 1 bits are added.

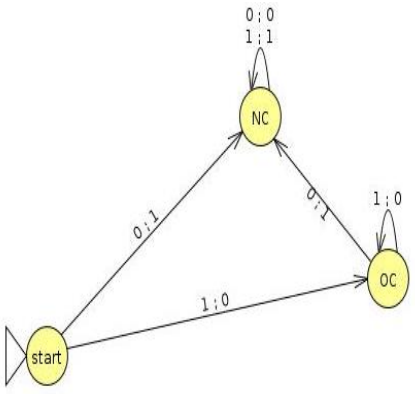
The concept of states tell us that what we are printing and what we have printed so far.

The Mealy machine have the states

q_0, q_1, q_2 , where q_0 is the start state and

$\Sigma = \{0,1\}$,

$\Gamma=\{0,1\}$



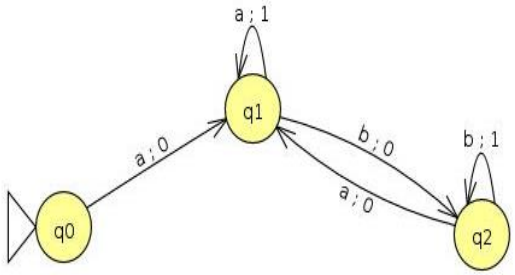
Overflow state

Due to typical property of Mealy machines of having input bits equals to outputs, if string 1111 is run as input, the output will be 0000 and not 10000. This is called overflow state.

Relationship between input and output

Generally, a Mealy machine does not accept or reject an input string, there is an implicit relationship between the input and output string.

Consider the following example Mealy machine taken from Daniel A Cohen book, in which whenever a double letter such as aa or bb appears, the output string places 1 as indication.

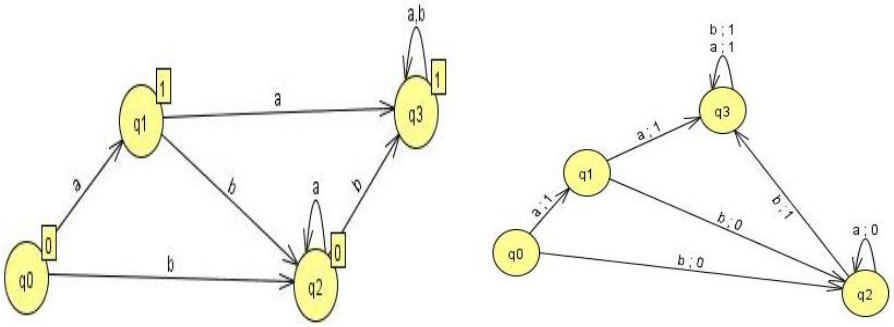


Theorem

Statement:

For every Moore machine there is a Mealy machine that is equivalent to it (ignoring the extra character printed the Moore machine).

Proof: see the following example…



Theorem

Statement:

For every Mealy machine there is a Moore machine that is equivalent to it (ignoring the extra character printed the Moore machine).

Proof:

Let M be a Mealy machine. At each state there are two possibilities for incoming transitions

All incoming transitions have the same output.

All incoming transitions have different output.

If all the transitions have same output characters, handling it is very easy.

If all the transitions have different output characters, then the corresponding state will be divided in to the number of outputs generated by incoming transitions.

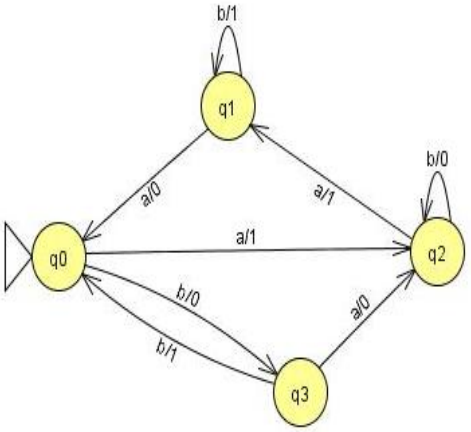
Please note that the divided state shall behave like the undivided one, that is the output should be same.

Initial state conversion….

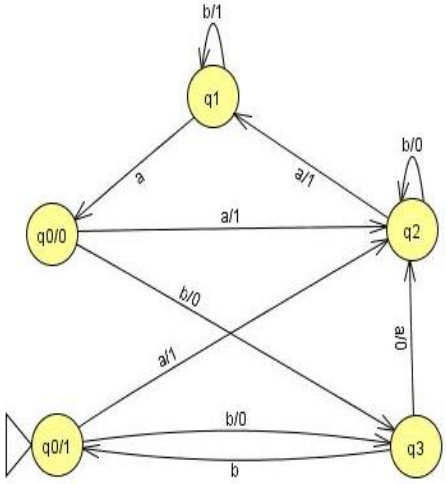
No output on a set of transitions….

Example

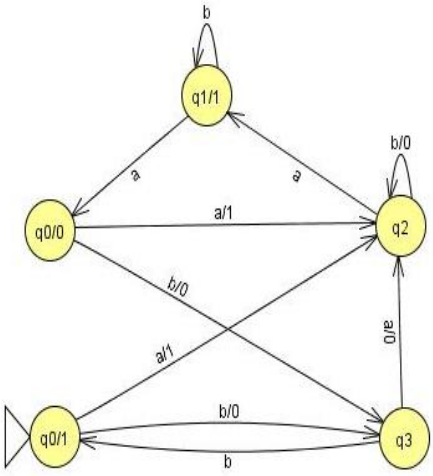
Consider the following Mealy machine



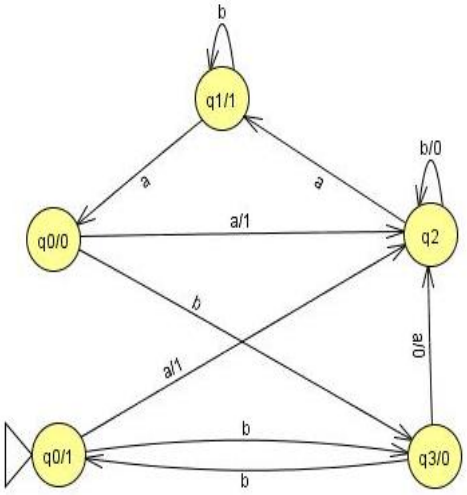
Dividing the state **q0into q0/0 and q0/1**



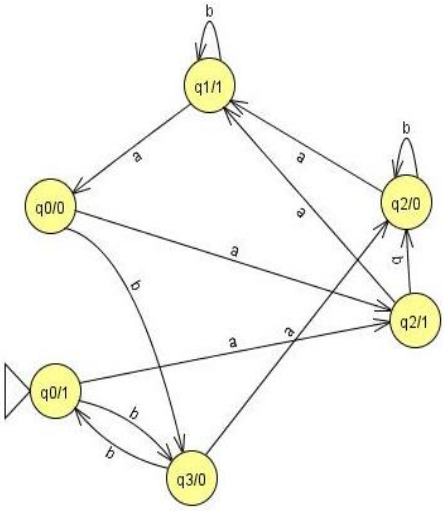
Shifting the output character 1 of transition a to $\mathbf{q_1}$Example continued ...



Shifting the output character 0 of transition b to $\mathbf{q_3}$



Dividing the state **q2into q2/0 and q2/1**



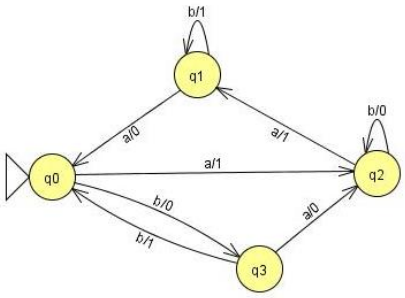
Summing Up

Applications of complementing and incrementing machines, Equivalent machines, Moore equivalent to Mealy, proof, example, Mealy equivalent to Moore, proof, example

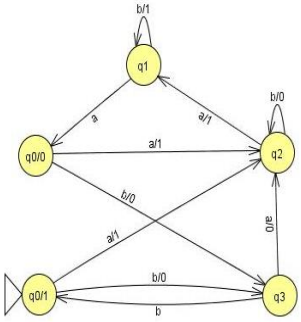
Lecture # 15

Example

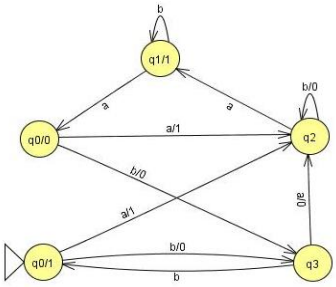
Consider the following Mealy machine



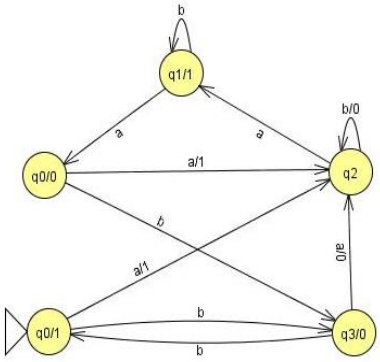
Dividing the state q0 into q0/0 and q0/1



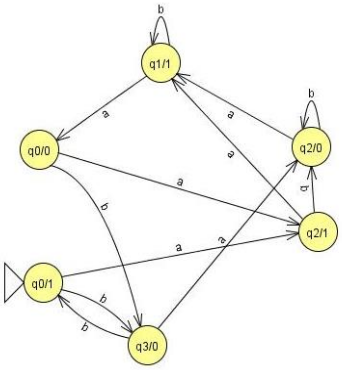
Shifting the output character 1 of transition a to q_1



Shifting the output character 0 of transition b to q_3



Dividing the state q2into q2/0 and q2/1



Finite Automata as Sequential Circuits

How the state machines that we have studied so far, can be used as sequential circuits.

We have already seen these systems working in our computer logic and architecture course.

Automata that we have studied so far, is called transducers because of their similarity to digital circuits.

Example Taken from Daniel I Cohen Book

Let us consider an example of a simple sequential circuit. The box labeled NAND means "not and.“. Its output wire carries the complement of the Boolean AND of its input wires.

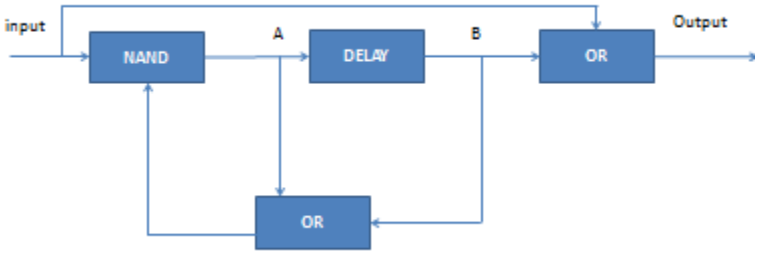
The output of the box labeled DELAY is the same as its previous input. It delays transmission of the signal along the wire by

Example1 ..

one step (clock pulse).

The DELAY is sometimes called a D flip-flop. The AND and OR are as usual. Current in a wire is denoted by the value 1, no current by 0.

Example1 ..



We identify four states based on whether or not there is current at pointsA and B in the circuit.

- q_o is A = 0 B = 0
- q_1, is A = 0 B = 1
- q_2 is A = 1 B = 0
- q_3 is A = 1 B = 1

The operation of this circuit is such that after an input of 0 or 1 the state changes according to the following rules:

new B = old A
new A = (input) NAND (old A OR old B)
output = (input) OR (old B)

At various discrete pulses of a time clock input is received, the state changes, and output is generated.

Suppose we are in state q_0 and we receive the input 0:

new B = old A = 0
new A = (input) NAND (old A OR old B)
= (0) NAND (0 OR 0)
= 0 NAND 0
= 1
output = 0 OR 0 = 0

The new state is q_2 (since new A = 1,
new B = 0).

If we are in state q_0 and we receive the input 1:

new B = old A = 0
new A = 1 NAND (0 OR 0) = I
output = 1 OR0 = 1

The new state is q_2 (since the new A = 1 and the new B = 0).

If we are in q_1, and we receive the input 0:

new B = old A = 0
new A = 0 NAND (0 OR 1) = I
output = 0 OR 1 = I

The new state is q_2.

If we are in q_1, and we receive the input 1:

new B = old A = 0
New A = 1 NAND (0 OR 1) = 0
output = I OR 1 = 1

The new state is q_o

If we are in state q_2 and we receive the input 0:

new B = old A = 1
new A = 0 NAND (1 OR0) = 1
output = 0OR0 = 0

The new state is q_3 (since new A = 1,new B = 1).

If we are in q_2 and we receive the input 1:

New B = old A = 1
new A = 1 NAND (1 OR 0) = 0
output = I OR0 = I

The new state is q_1 .

If we are in q_3 and we receive the input 0:

newB = oldA = 1
new A = 0 NAND (1 OR 1) =
output = 0 OR 1 = 1

The new state is q_3.

If we are in q_3 and we receive the input 1:

new B = old A = I
new A = 1 NAND (1 OR 1) =
output = I OR 1 = I

The new state is q_1 .

Example1 ..

If we are in q_3 and we receive the input 0:

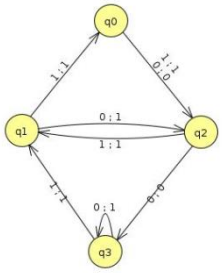
newB = oldA = 1
new A = 0 NAND (1 OR 1) =
output = 0 OR 1 = 1

The new state is q_3.

If we are in q_3 and we receive the input 1:

new B = old A = I
new A = 1 NAND (1 OR 1) =
output = I OR 1 = I

The new state is q_1 .



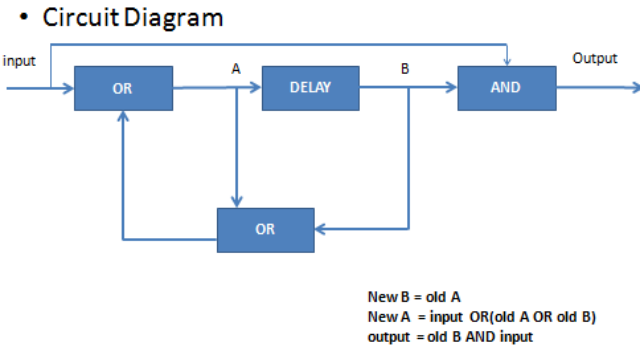
	After input 0		After input 1	
Old State	**New State**	**Output**	**New State**	**Output**
q_0	q_2	0	q_2	1

q_1	q_2	1	q_0	1
q_2	q_3	0	q_1	1
q_3	q_3	1	q_1	1

Lecture # 16

Example2 Taken from the Book Exercises

Circuit Diagram



Example2…

A = 0, B = 0 q_0

A = 0, B = 1 q_1

A = 1, B = 0 q_2

A = 1, B = 1 q_3

New B = old A

New A = input OR(old A OR old B)

output = old B AND input

State q_0 /input = 0 / A = 0, B = 0

New B = old A = 0

New A = input OR (old A OR old B)

= 0 OR (0 OR 0)

= 0 OR 0 = 0

Output = input AND old B

= 0 AND 0 = 0

State q_o / input = 1 / A =0, B = 0

new B = old A = 0

new A = input OR (old A OR old B)

= 1 OR (0 OR 0)

= 1 OR 0 = 1

output = input AND old B

= 1 AND 0 = 0

state q_1 / input = 0 /A = 0, B= 1

new B = old A = 0

new A = input OR (old A OR old B)

= 0 OR (0 OR 1)

= 0 OR 1 = 1

output = old B AND input = 1 AND 0 = 0

state q_1/ input = 1 /A = 0 , B = 1

new B = old A = 0

new A = input OR (old A OR old B)

= 1 OR (0 OR 1)

= 1 OR 1 = 1

output = old B AND input = 1 AND 1 = 1

state q_2/ input = 0/ A = 1, B= 0

new B = old A = 1

new A = input OR (old A OR old B)

= 0 OR (1 OR 0)

= 0 OR 1 =1

output = old B AND input = 0 AND 0 = 0

state q_2 / input = 1 / A= 1/ B= 0

new B = old A = 1

new A = input OR (old A OR old B)

= 1 OR (1 OR 1)

= 1 OR 1= 1

output = old B AND input = 0 AND 1 = 0

state q_3 / input = 0 / A= 1/ B= 1

new B = old A = 1

new A = input OR (old A OR old B)

= 0 OR (1 OR 1)

= 0 OR 1= 1

output = old B AND input = 1 AND 0 = 0

state q3 / input = 1 / A= 1/ B= 1

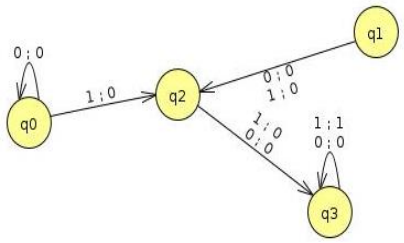
new B = old A = 1

new A = input OR (old A OR old B)

= 1 OR (1 OR 1)

= 1 OR 1= 1

output = old B AND input = 1 AND 1 = 1



	After input 0		After input 1	
Old State	**New State**	**Output**	**New State**	**Output**
q_0(A=0,B=0)	q_0	0	q_2	0
q_1(A=0,B=1)	q_2	0	q_2	0
q_2(A=1,B=0)	q_3	0	q_3	0
q_3(A=1,B=1)	q_3	0	q_3	1

Regular Languages

Defined by Regular Expression

Properties:

- Closure Properties
- Complements and intersection

See chapter 9 of the Daniel Book. (All theorems are referred from the book)

Theorem 10

Statement

If L_1 and L_2 are regular languages then $L_1 + L_2$, L_1L_2, L_1^* are also regular languages

PROOF: By Regular Expressions

This proof uses the fact that if L_1 and L_2 are regular languages, they must be definable by some regular expressions.

if L_1 and L_2 are regular languages then there are regular expressions r_1, r_2 that define these languages.

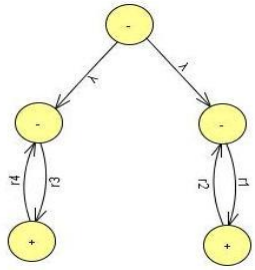
- $(r_1 + r_2)$ defines $L_1 + L_2$
- $r_1 r_2$ defines $L_1 L_2$
- $(r_1)^*$ defines L_1^*

PROOF: By Machines

Let there are two TGs that accept L_1 and L_2

- TG_1 accepts L_1
- TG_2 accepts L_2

Each TG has a unique start state and unique separate final state .

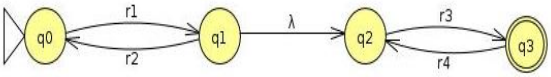


TG Accepts $L_1 + L_2$

- Starting at the start state of TG_1 , can only follow a path on TG_1
- Starting at the start state of TG_2, can only follow a path on TG2

Theorem 10

PROOF: By Machines



TG Accepts $L_1 L_2$

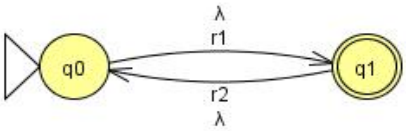
- Can stop and accept the string or jump back at no cost, and run another segment of input string back down to +

Theorem 10

Proof Example

TG Accepts L_1*

- Can stop and accept the string or jump back at no cost
- and run another segment of input string back down to +



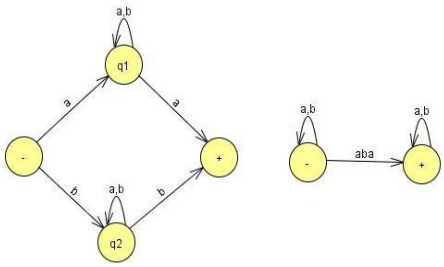
Proof Example

- Let the alphabet be Σ = {a,b}
    - L_1 = all words of two ore more letters that begin and end with the same letter
    - L_2 = all words contain the substring aba
- The regular Expressions are:
    - a(a + b)*a + b(a + b)*b
    - (a + b)*aba(a + b)*

Σ = {a,b} $L_1 + L_2$

L_1 = all words of two ore more letters that begin and endwith the same letter.

L_2 = all words contain the substring aba



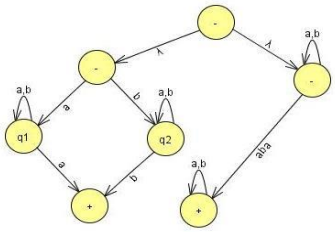
Example Σ = {a,b}

Σ = {a,b} , L_1L_2

L_1 = all words of two ore more letters that begin and endwith the same letter.

L_2 = all words contain thesubstring aba.

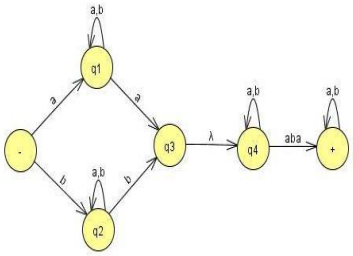
The language $L_1 + L_2$ is regular because it can be defined by regular expression [a(a + b)*a + b(a + b)*b] + [(a + b)*aba(a + b)*]



Example Σ = {a,b}

Σ = {a,b} , L_1*

L_1 = all words of two ore more letters that begin and endwith the same letter.

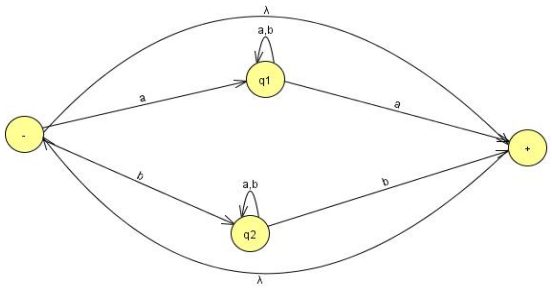


L_2 = all words contain the substring aba.

Example Σ = {a,b}

Σ = {a,b} , L_1*

L_1 = all words of two ore more letters that begin and end with the same letter.



The language L_1* is regular because it can be defined by regular expression [a(a + b)*a + b(a + b)*b]*

Accepted by TG

Complements and Intersections

If L is a language over Σ , its complement is defined L', the language of all strings of letters from Σ that are not words in L

Example

If L is a language over = (a,b) of all words that have a double a in them, then L' is the language of all words that donot have a double a.

The complement of a language L' is the language Li.e. (L')' = L

Theorem 11

Statement

The set of regular languages is closed under complement

If L is a regular language then L' is also a regular language

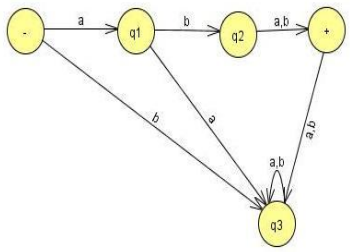
PROOF:

- If L is a regular language then there is some FA that accepts L. (Kleen's theorem)
- Make final state nonfinal state.
- Make nonfinal state final state.
- The input string now ends in nonfinal state and vice versa.
- Make Initial state both initial and final.
- Accepts words of L'.
- Rejects words in L.
- This machine accepts exactly words in L'.
- So L' is regular. (Kleen's theorem)

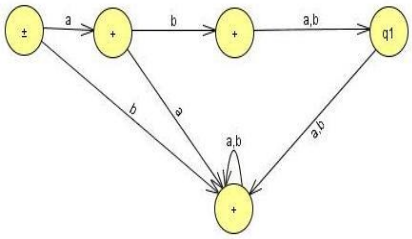
•

Example:

A FA that accepts only the string aba and abb



A FA that accepts all strings but not aba and abb



Theorem 12

Statement

The set of regular languages is closed under intersection.

If L1 and L2 are regular languages then $L_1 \cap L_2$ is also a regular language.

Theorem 12 (Regular Languages Intersection Theorem)

PROOF: By Demorgan's Law

For sets of any kind (regular or not)

$$L_1 \cap L_2 = (L_1' + L_2')'$$

$L_1 \cap L_2$ consists of all words that are not in L1' or L2'.

Because L1 is regular then L1’ is also regular (using Theorem 11) and so as L2’ is regular

Also L_1' + L_2' is regular, therefore (L_1' + L_2')‘ is also regular (using Theorem 11)

Example

Two languages over Σ = {a,b}

L_1 = all strings with a double a

L_2 = all strings with an even number of a's.

L_1 and L_2 are not the same, since aaa is in L_1 but not in L_2 and aba in in L_2 but not in L_1

Example

L_1 and L_2 are regular languages defined by the regular expressions below.

r_1 = (a + b)*aa(a + b)*

r_2 = b*(ab*ab*)*

A word in L_2 can have some b’s in the front

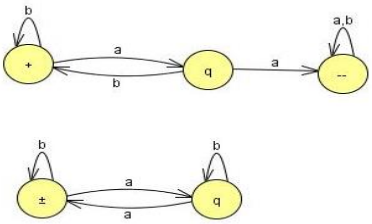
But whenever there is an a, it balanced by an other a (after some b’s).

Gives the factor of the form (ab*ab*)

The words can have as many factors of this from as it wants. It can end an a or ab.

Example

These two languages can also be defined by FA (Kleen’s theorem)



- In the first machine we stay in the start state until we read our first a.
- Then we move to the middle state.
- Here we can find a double a.
- If we read another a from the input string while in the middle state, we move to the final state where we remain.

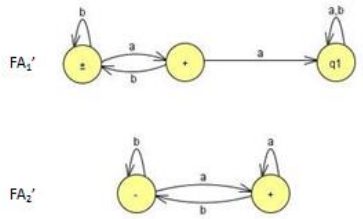
If we read a b, we go back to -. If we never get past the middle state, the word has no double a and is rejectedTheorem 12

- The second machine switches from left state to right state or from right state to left state every time it reads an a.
- It ignores all b's. If the string begins on the left and ends on the left, it must have made an even number of left/right switches.
- Therefore, the strings this machine accepts are exactly those in L_2.
- The first step in building the machine (and regular expression) for $L_1 \cap L_2$ is to find the machines that accept the complementary languages L_1' and L_2'.
- The English description of these languages is:

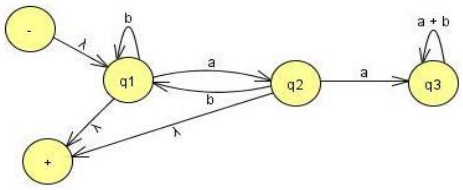
  L_1' = all strings that do not contain the substring a

  L_2' = all strings having an odd number of a's
- In the proof of the theorem that the complement of a regular language is regular an algorithm is given for building the machines that accept these languages.
- Now reverse what is a final state and what is not a final state.
- The machines for these languages are then

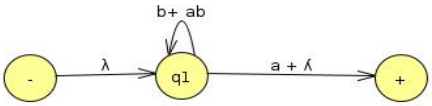
- The machines for these languages are



- Recall that how we go through stages of transition graphs with edges labeled by regular expressions. Thus, FA_1' becomes:



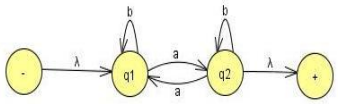
- State q_3 is part of no path from - to +, so it can be dropped.
- We need to join incoming a edge with both outgoing edges(b to q1 and q2 to +).
- When we add the two loops, we get b + ab and the sum of the two edges from q1 to + is a + Λ so the machine looks like



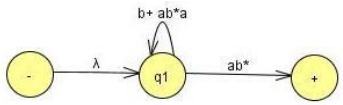
- The last step is to bypass state q_1.
- Concentrate the incoming Λ label with the loop label starred (b + ab)*

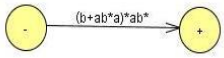
- Concentrated with outgoing (a + Λ) to produce one edge from – to + with the regular expression for L_1' , r_1' = (b +ab)*(a + Λ)
  - Lets do the same thing for the language L_2

    FA2 becomes



  - When state q_2 is bypassed and adding the two loop labels we get,





  - It gives the regular expression

    r_2' = (b + ab*a)*ab*

  - This is one of the regular expression that define the language of all words with an odd number of a's.Another expression that define the same language
  - Add b*a in the front of regular expression for L1

We getb*ab*(ab*ab*)*Theorem 12

- This works because words with an odd number of a's can be interpreted as b*a in front of words with an even number of a's.
- The fact that these two different regular expressions define the same language is not obvious.
- As we have regular expression for L_1' and L_2'
- We can write the regular expression for $L_1'+L_2'$

  $r_1' + r_2' = (b + ab)*(a + \Lambda) + (b + ab*a)*ab*$
- Make this regular expression in to FA
- so that we can take its complement to get the FA that defines $L_1 \cap L_2$.

Lecture # 17

Theorem 12 – Repeat

Statement

The set of regular languages is closed under intersection.

If L1 and L2 are regular languages then $L_1 \cap L_2$ is also a regular language.

Theorem 12 – Repeat

PROOF: By Demorgan's Law

- For sets of any kind (regular or not)

$$L_1 \cap L_2 = (L_1' + L_2')'$$

- $L_1 \cap L_2$ consists of all words that are not in L1' or L2'.
- Because L1 is regular then L1' is also regular (using Theorem 11) and so as L2' is regular
- Also $L_1' + L_2'$ is regular, therefore $(L_1' + L_2')'$ is also regular (using Theorem 11)

Example

- Two languages over $\Sigma = \{a,b\}$

L_1 = all strings with a double a

L_2 = all strings with an even number of a's.

- L_1 and L_2 are not the same, since aaa is in L_1 but not in L_2 and aba in in L_2 but not in L_1

Example

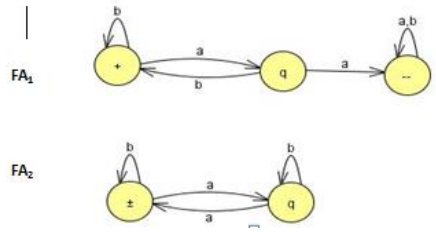
- L_1 and L_2 are regular languages defined by the regular expressions below.

$r_1 = (a + b)^*aa(a + b)^*$

$r_2 = b^*(ab^*ab^*)^*$

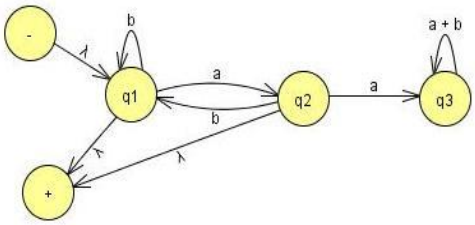
- A word in L_2 can have some b's in the front
- But whenever there is an a, it balanced by an other a (after some b's).

- Gives the factor of the form (ab*ab*)
- The words can have as many factors of this from as it wants. It can end an a or ab.
- These two languages can also be defined by FA (Kleen's theorem)

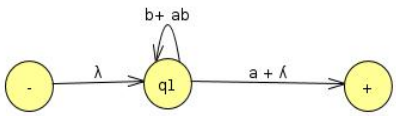


Example

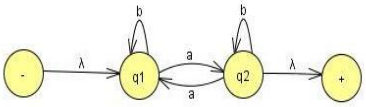
- In the proof of the theorem that the complement of a regular language is regular an algorithm is given for building the machines that accept these languages.
- Now reverse what is a final state and what is not a final state.
- The machines for these languages are then



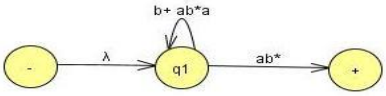
- State q_3 is part of no path from - to +, so it can be dropped.
- We need to join incoming a edge with both outgoing edges(b to q1 and q2 to +).
- When we add the two loops, we get b + ab and the sum of the two edges from q1 to + is a + Λ so the machine looks like



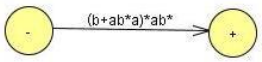
-
- The same thing for the language L_2FA2 becomes



- Simplify:
- Eliminate state q_2
- There is one incoming edge a loop, and two outgoing edges,
- Now to replace them with only two edges:
- The path $q_1 – q_2 – q_2 – q_1$ becomes a loop at q_1
- The path $q_1 – q_2 – q_2 –$ +becomes an edge from q1 to +.
- When state q_2 is bypassed and adding the two loop labels we get,



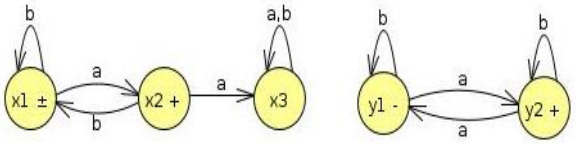
We can eliminate state q_1 we get



- As we have regular expression for L_1' and L_2'
- We can write the regular expression for L_1'+L_2'

  $r_1' + r_2' = (b + ab)^*(a + \Lambda) + (b + ab^*a)^*ab^*$

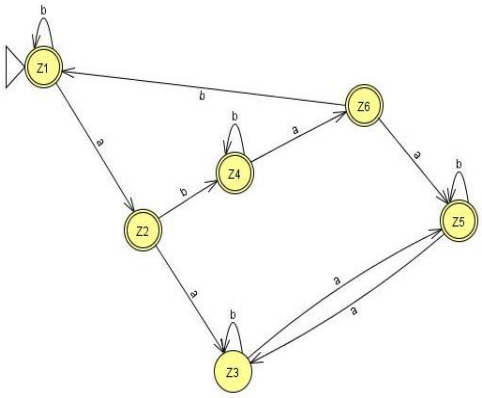
- Make this regular expression in to FA so that we can take its complement to get the FA that defines $L_1 \cap L_2$.
- To build the FA that corresponds to a complicated regular expression is not easy job, as (from the proof of Kleene's Theorem).
- Alternatively:
  - Make the machine for $L_1' + L_2'$ directly from the machines for L_1' and L_2' without resorting to regular expressions.
  - The method of building a machine that is the sum of two FA's is already developed .
  - Also in the proof of Kleene's Theorem.
- Let us label the states in the two machines for FA_1' and FA_2'.



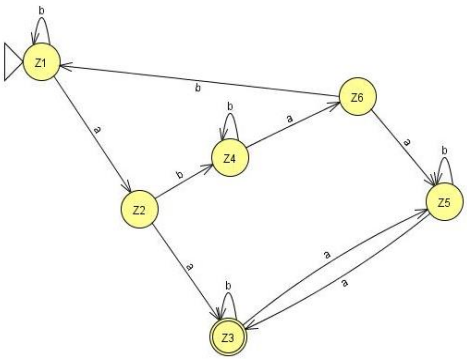
Old States	Reading at a	Reading at b
z1-+≡(x1,y1)	(x2,y2)≡z2	(x1,y1)≡z1
Z2+≡(x2,y2)	(x3,y1)≡z3	(x1,y2)≡z4
z3≡(x3,y1)	(x3,y2)≡z5	(x3,y1)≡z3
Z4+≡(x1,y2)	(x2,y1)≡z6	(x1,y2)≡z4
Z5+≡(x3,y2)	(x3,y1)≡z3	(x3,y2)≡z5
Z6+≡(x2,y1)	(x3,y2)≡z5	(x1,y1)≡z1

- The start states are x_1, and y_1 and the final states are x_1, x_2, and y_2.

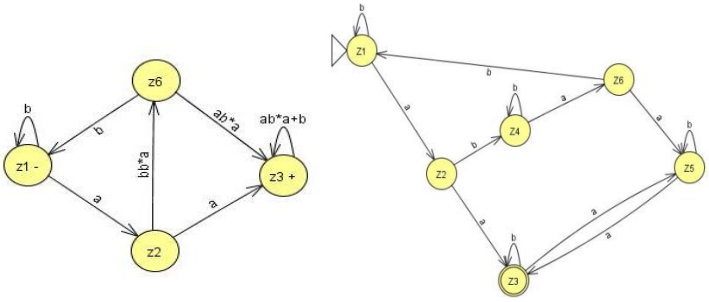
The Union Machine



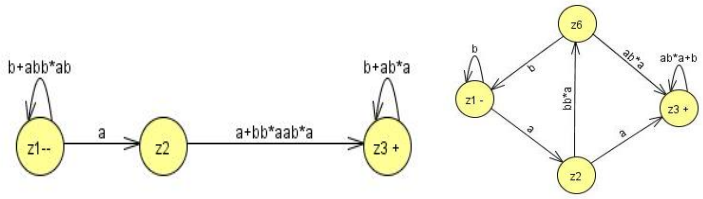
- This FA can accept the language $L_1'+L_2'$
- Reverse the status of each state from final to non-final and vice versa, an FA is going to be produced for the language $L_1 \cap L_2$ after taking the complement again.



- **Bypassing z_4 and z_5 gives**



- Then bypassing z_3 gives



-

- So the whole machine reduces to regular expression

  (b + abb*ab)*a(a + bb*aab*a)(b + ab*a)*

- As it stands, there are four factors (the second is just an a and the first and fourth are starred).

- Every time we use one of the options from the two end factors we incorporate an even number of a's into the word (either none or two).

- The second factor gives us an odd number of a's (exactly one).

- The third factor gives us the option of taking either one or three a's. In total, the number of a's must be even.

- So all the words in this language are in L_2.

- The second factor gives us an a, and then we must immediately concatenate this with one of the choices from the third factor.
- If we choose the other expression, bb*aab*a,
- then we have formed a double a in a different way.
- By either choice the words in this language all have a double a and are therefore in L_1.
- This means that all the words in the language of this regular expression are contained in the language $L_1 \cap L_2$.
- Are all the words in $L_1 \cap L_2$ included in the language of this expression?YES
- Look at any word that is in $L_1 \cap L_2$
- It has an even number a's and a double a somewhere
- Two possibilities, to consider separately
- Before the first double a there are an even number of a's.
- Before the first double a there are an odd number of a's.
- Words of type 1 come from expression
- (even number of a's but not doubled) (first aa)
- (even number of a's may be doubled)

  = (b + abb*ab)* (aa)(b + ab*a)*

  = type 1
- Notice that
- The third factor defines the language L, and is a shorter expression than the r_1, used in previous slide.
- Words of type 2 come from the expression:
- (odd number of not doubled a's) (first aa)
- (odd number of a's may be doubled)
- Notice that
- The first factor must end in b, since none of its a's is part of a double a.
- = [(b + abb*ab)*abb*] aa [b*a(b + ab*a)*]

- = (b + abb*ab)*(a)(bb* aab*a)(b + ab*a)*
- = type 2
- Adding type 1 and type 2 together (and factoring out like terms using the distributive law), we obtain the same expression we got from the algorithm.
- We now have two proofs that this is indeed a regular expression for the language $L_1 \cap L_2$
-

Non-regular languages

Many Languages can be defined using FA's.

These languages have had many different structures, they took only a few basic forms: languages with required substrings, languages that forbid some substrings, languages that begin or end with certain strings,languages with certain even/odd properties, and so on.

Lets take a look to some new forms, such as the language PALINDROME or the language PRIME of all words a^P where p is a prime number.

Neither of these is a regular language.

They can be described in English, but they cannot be defined by an FA.

Conclusion: More powerful machines are needed to define them

Definition

A language that cannot be defined by a regular expression is called a non-regular language.

Non-regular languages

Cannot be accepted by any FA or TG.
(Kleene's theorem,)

All languages are either regular or nonregular, but not both of them.

Non-regular languages

Consider a simple case. Let us define the language L.

L = {a, ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb . ..}

We could also define this language by the formula

$$L = \{a^n b^n \text{ for } n = 0\ 1\ 2\ 3\ 4\ 5\ \ldots\}$$

Or short for

$$L = \{a^n b^n\}$$

Please not that when the range of the abstract exponent n is unspecified we mean to imply that it is 0,1,2,3, ...

Lets show that this language is nonregular.

It is a subset of many regular languages, such as a*b*, which, however, also includes such strings as aab and bb that $\{a^n b^n\}$ does not.

Suppose this language is regular and some FA must be developed that accepts it.

Let us picture one of these FA's .This FA might have many states.

Say that it has 95 states, just for the sake of argument.

Yet we know it accepts the word $a^{96}b^{96}$. The first 96 letters of this input string are all a's and they trace a path through this hypothetical machine.

Because there are only 95 states ,The path cannot visit a new state with each input letter read.

The path returns to a state that it has already visited at some point.

The first time it was in that state it left by the a-road.

The second time it is in that state it leaves by the a-road again.

Even if it only returns once we say that the path contains a circuit in it.

(A circuit is a loop that can be made of several edges.)

The path goes up to the circuit and then it starts to loop around the circuit, maybe 0 or more times.

It cannot leave the circuit until, b is read in.

Then the path can take a different turn. In this hypothetical example the path could make 30 loops around a three-state circuit before the first b is read.

After the first b is read, the path goes off and does some other stuff following b edges and eventually winds up at a final state where the word $a^{96}b^{96}$ is accepted.

Let us, say that the circuit that the a-edge path loops 'around has seven states in it. The path enters the circuit, loops around it continuously and then goes off on the b-line to a final state.

What will happen to the input string $a^{96+7}\,b^{96}$? Just as in the case of the input string $a^{96}\,b^{96}$,

This string would produce a path through the machine that would walk up to the same circuit (reading in only a's) and begin to loop around it in exactly the same way.

However, the path for $a^{96+7}\,b^{96}$ loops around this circuit one more time than the path for $a^{96}\,b^{96}$ --precisely one extra time.

Both paths, at exactly the same state in the circuit, begin to branch off on the b-road.

Once on the b-road, they both go the same 96 b-steps and arrive at the same final state.

But this would mean that the input string $a^{103}\,b^{96}$ is accepted by this machine.

But that string is not in the language $L = \{a^n b^n\}$.

This is a contradiction. We assumed that we were talking about an FA that accepts exactly the words in L and then we were able to prove that the same machine accepts some word that is not in L.

This contradiction means that the machine that accepts exactly the words in L does not exist. In other words, L is nonregular.

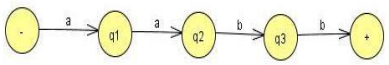
Let us review what happened.

We chose a word in L that was so large (had so many letters) that its path through the FA had to contain a circuit.

Once we found that some path with a circuit could reach a final state, we asked ourselves what happens to a path that is just like the first one,

But that loops around the circuit one extra time and then proceeds identically through the machine.

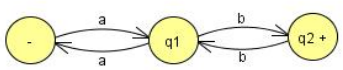
Suppose the following FA for a^2b^2



Only those used in the path of a^2b^2

Consider the following FA with Circuit….

However, the path for $a^{13}\,b^9$ still has four more a-steps to take, which is one more time around the circuit, and then it follows the nine b-steps.



Let us return to.

With our first assumptions we made above (that there were 95 states and that the circuit was 7 states long), we could also say that $a^{110}b^{96}$, $a^{117}b^{96}$, $a^{124}b^{96}$ ... are also accepted by this machine.

All can be written the form

$$\mathbf{a^{96}(a^{7})^{m}b^{96}}$$

where m is any integer 0,1,2,3 ..... If m is 0, the path through this machine is the path for the word $a^{96}b^{96}$.

If m is (1, that looks the same, but it loops the circuit one more time.

If m 2, the path loops the circuit two more times.

In general, $\mathbf{a^{96}(a^{7})^{m}b^{96}}$ loops the circuit exactly m more times.

After doing this looping it gets off the circuit at exactly the same place $a^{96}b^{96}$ does and proceeds along exactly the same route to the final state.

All these words, though not in L, must be accepted.

Suppose that we had considered a different machine to accept the language L, a machine that has 732 states.

When we input the word $a^{733}\,b^{733}$,

the path that the a's take must contain a circuit.

The word $a^{9999}b^{9999}$ also must loop around a circuit in its a-part of the path.

Suppose the circuit that the a-part follows has 101 states.

Then $a^{733+101}\,b^{733}$ would also have to be accepted by this machine, because

its path is the same in every detail except that it loops the circuit one more time.

This second machine must also accept some strings, that are not in L:

$a^{834}\,b^{733}$ $a^{935}\,b^{733}$ $a^{1036}\,b^{733}$ ……

$= a^{733}\,(a^{101})^{m}\,b^{733}$

PUMPING LEMMA (Version I)

Let L be any regular language that has infinitely many words. Then there exist some three strings x, y, and z (where y is not the null string) such that all the strings of the form xy^nz for n = 1 2 3 ...

Theorem 13

PROOF

If L is a regular language, then there is an FA that accepts exactly the words in L.

Let us focus on one such machine.

Like all FA's, this machine has only finitely many states. But L has infinitely many words in it.

This means that there are arbitrarily long' words in L. (If there were some maximum on the length of all the words in L, then L could have only finitely many words in total.)

Let w be some word in L that has more letters in it than there are states in the machine we are considering.

When this word generates a path through the machine, the path cannot visit a new state for each letter because there are more letters than states.

Therefore, it must at some point revisit a state that it has been to before.

Part 1

All the letters of w starting at the beginning that lead up to the first state that is revisited. Call this part x. Notice that x may be the null string if the path for w revisits the start state as its first revisit.

Part 2

Starting at the letter after the substring x, let y denote the substring of w that travels around the circuit coming back to the same state the circuit began with.

Since there must be a circuit, y cannot be the null string. y contains the letters of w for exactly one loop around this circuit.

Part 3

Let z be the rest of w starting with the letter after the substring y and going to the end of the string w. This z could be null.

The path for z could also possibly loop around the y circuit or any other. What z does is arbitrary.

From the definition of these three substrings

w = xyz

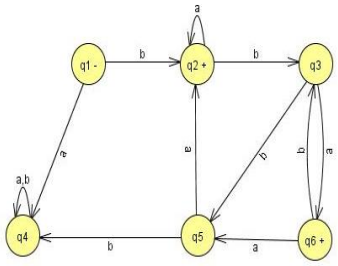
w is accepted by this machine.

Theorem 13

Example

Let us illustrate the action of the Pumping Lemma on a concrete example of a regular language.

The machine below accepts an infinite language and has only six states



Example

Any word with six or more letters must correspond to a path that includes a circuit.

Some words with fewer than six letters correspond to paths with circuits, such as baaa.

The word we will consider in detail is

w = bbbababa

Which has more than six letters and therefore includes a circuit.

The path that this word generates through the FA can be decomposed into three stages:

The first part, the x-part, goes from the - state up to the first circuit.

This is only one edge and corresponds to the letter b alone.

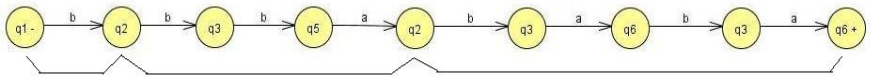
The second stage is the circuit around states q2, q3, and q5.

This corresponds to edges labeled b, b, and a.

We therefore say that the substring bba is the y-part of the word w.

After going around the circuit, the path proceeds to states q3, q6, q3, and q6.

This corresponds to the substring baba of w, which constitutes the z-part.



w = b bba baba

- what would happen to the input string xyyz.

  x y y z = b bba bba baba

- Clearly, the x-part (the letter b) would take this path from -- to the beginning of the circuit in the path of w.

- Then the y-part would circle the circuit in the same way that the path for w does when it begins xy.

Theorem 13

At this point, we are back at the beginning of the circuit. We mark off the circuit starting at the first repeated state, which in this case is state q2.

Consider it to be made up of exactly as many edges as it takes to get back there (in this case 3 edges).

In the original path for w we proceed again from state q2 to state q3 as in the circuit, this is not part of the first simple- looping and so not part of y.

so, we can string two y-parts together since the second y begins in the state in which the first leaves us.

The second y-part circles the circuit again and leaves us in the correct state to resume the path of the word w.

We can continue after xyy with the z path exactly as we continued w with the z path after its y-part.

Lecture 18

Theorem 12 – Repeat

Statement

The set of regular languages is closed under intersection.

If L1 and L2 are regular languages then $L_1 \cap L_2$ is also a regular language.

PROOF: By Demorgan's Law

- For sets of any kind (regular or not)

$$L_1 \cap L_2 = (L_1' + L_2')'$$

- $L_1 \cap L_2$ consists of all words that are not in L1' or L2'.
- Because L1 is regular then L1' is also regular (using Theorem 11) and so as L2' is regular
- Also $L_1' + L_2'$ is regular, therefore $(L_1' + L_2')'$ is also regular (using Theorem 11)

Example

Two languages over $\Sigma = \{a,b\}$

L_1 = all strings with a double a

L_2 = all strings with an even number of a's.

- L_1 and L_2 are not the same, since aaa is in L_1 but not in L_2 and aba in in L_2 but not in L_1

Example

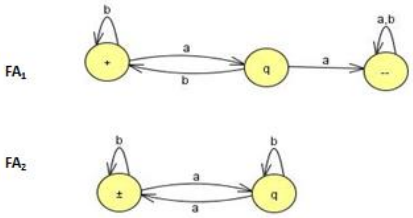
- L_1 and L_2 are regular languages defined by the regular expressions below.

$r_1 = (a + b)^*aa(a + b)^*$

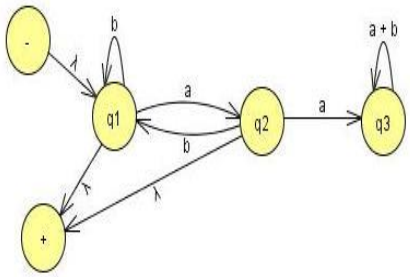
$r_2 = b^*(ab^*ab^*)^*$

- A word in L_2 can have some b's in the front
- But whenever there is an a, it balanced by an other a (after some b's).
- Gives the factor of the form (ab*ab*)
- The words can have as many factors of this from as it wants. It can end an a or ab.

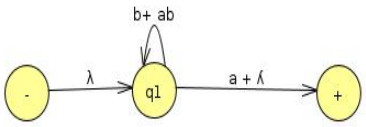
- These two languages can also be defined by FA (Kleen's theorem)



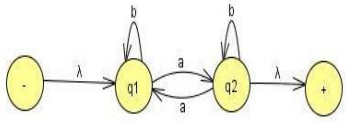
- In the proof of the theorem that the complement of a regular language is regular an algorithm is given for building the machines that accept these languages.
- Now reverse what is a final state and what is not a final state.
- The machines for these languages are then
- Recall that how we go through stages of transition graphs with edges labeled by regular expressions. Thus, FA_1' becomes:



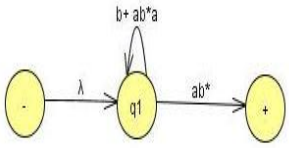
- State q_3 is part of no path from - to +, so it can be dropped.
- We need to join incoming a edge with both outgoing edges(b to q1 and q2 to +).
- When we add the two loops, we get b + ab and the sum of the two edges from q1 to + is a + ʌ so the machine looks like



- The same thing for the language L_2
- FA2 becomes



- Simplify:
- Eliminate state q_2
- There is one incoming edge a loop, and two outgoing edges,
- Now to replace them with only two edges:
- The path $q_1 – q_2 – q_2 – q_1$ becomes a loop at q_1
- The path $q_1 – q_2 – q_2 –$ +becomes an edge from q1 to +.
- When state q_2 is bypassed and adding the two loop labels we get,

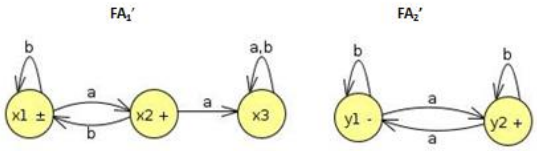


- We can eliminate state q_1 we getTheorem 12
- As we have regular expression for L_1' and L_2'
- We can write the regular expression for L_1'+L_2'

$$r_1' + r_2' = (b + ab)^*(a + \Lambda) + (b + ab^*a)^*ab^*$$

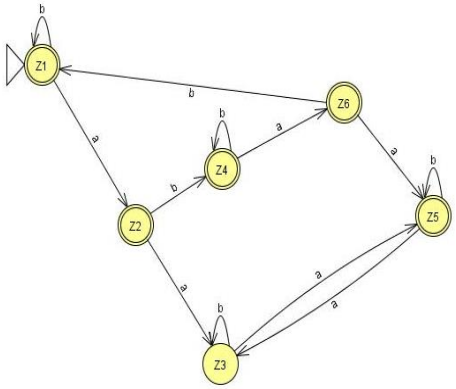
- Make this regular expression in to FA
- so that we can take its complement to get the FA that defines $L_1 \cap L_2$.
- To build the FA that corresponds to a complicated regular expression is not easy job, as (from the proof of Kleene's Theorem).
- Alternatively:
- Make the machine for $L_1' + L_2'$ directly from the machines for L_1' and L_2' without resorting to regular expressions.
- The method of building a machine that is the sum of two FA's is already developed .
- Also in the proof of Kleene's Theorem

- Let us label the states in the two machines for FA_1' and FA_2'.

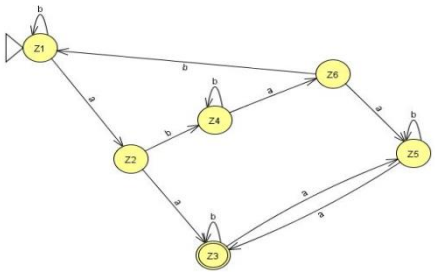


Old States	Reading at a	Reading at b
z1-+≡(x1,y1)	(x2,y2)≡z2	(x1,y1)≡z1
Z2+≡(x2,y2)	(x3,y1)≡z3	(x1,y2)≡z4
z3≡(x3,y1)	(x3,y2)≡z5	(x3,y1)≡z3
Z4+≡(x1,y2)	(x2,y1)≡z6	(x1,y2)≡z4
Z5+≡(x3,y2)	(x3,y1)≡z3	(x3,y2)≡z5
Z6+≡(x2,y1)	(x3,y2)≡z5	(x1,y1)≡z1

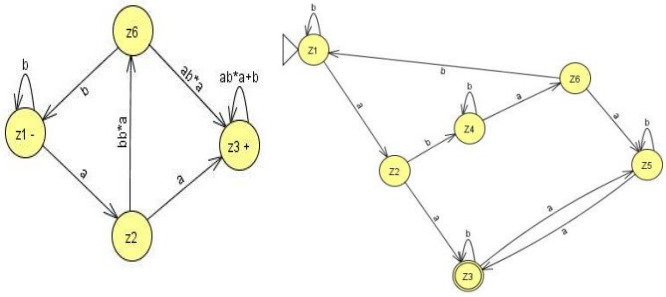
- The Union Machine



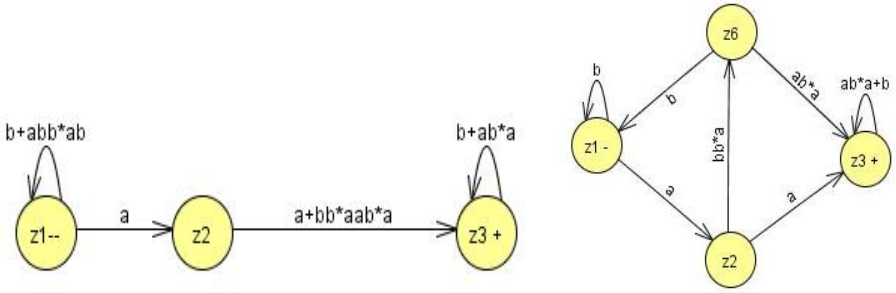
- This FA can accept the language L_1'+L_2'
- Reverse the status of each state from final to non-final and vice versa, an FA is going to be produced for the language $L_1 \cap L_2$ after taking the complement again.



- Bypassing z_4 and z_5 gives



- Then bypassing z_3 gives



-
- So the whole machine reduces to regular expression

  (b + abb*ab)*a(a + bb*aab*a)(b + ab*a)*

- As it stands, there are four factors (the second is just an a and the first and fourth are starred).

- Every time we use one of the options from the two end factors we incorporate an even number of a's into the word (either none or two).

- The second factor gives us an odd number of a's (exactly one).

- The third factor gives us the option of taking either one or three a's. In total, the number of a's must be even.

- So all the words in this language are in L_2.

- The second factor gives us an a, and then we must immediately concatenate this with one of the choices from the third factor.
- If we choose the other expression, bb*aab*a,
- then we have formed a double a in a different way.
- By either choice the words in this language all have a double a and are therefore in L_1.
- This means that all the words in the language of this regular expression are contained in the language $L_1 \cap L_2$.
- Are all the words in $L_1 \cap L_2$ included in the language of this expression?YES
- Look at any word that is in $L_1 \cap L_2$
- It has an even number a's and a double a somewhere
- Two possibilities, to consider separately
- Before the first double a there are an even number of a's.
- Before the first double a there are an odd number of a's.
- Words of type 1 come from expression
- (even number of a's but not doubled) (first aa)
- (even number of a's may be doubled)

  = (b + abb*ab)* (aa)(b + ab*a)*

  = type 1
- Notice that
- The third factor defines the language L, and is a shorter expression than the r_1, used in previous slide.
- Words of type 2 come from the expression:
- (odd number of not doubled a's) (first aa)
- (odd number of a's may be doubled)
- The first factor must end in b, since none of its a's is part of a double a.
- = [(b + abb*ab)*abb*] aa [b*a(b + ab*a)*]
- = (b + abb*ab)*(a)(bb* aab*a)(b + ab*a)*

- = type 2
-

Non-regular languages

- Many Languages can be defined using FA's.
- These languages have had many different structures, they took only a few basic forms: languages with required substrings, languages that forbid some substrings, languages that begin or end with certain strings,
- languages with certain even/odd properties, and so on.
- Lets take a look to some new forms, such as the language PALINDROME or the language PRIME of all words a^P where p is a prime number.
- Neither of these is a regular language.
- They can be described in English, but they cannot be defined by an FA.
- Conclusion: More powerful machines are needed to define them

Definition

A language that cannot be defined by a regular expression is called a non-regular language.

- Cannot be accepted by any FA or TG.
  (Kleene's theorem,)
- All languages are either regular or nonregular, but not both of them.

- Consider a simple case. Let us define the language L.
- L = {a, ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb . ..}
- We could also define this language by the formula

$$L = \{a^n b^n \text{ for } n = 0\ 1\ 2\ 3\ 4\ 5\ \ldots\}$$

Or short for

$$L = \{a^n b^n\}$$

- Please not that when the range of the abstract exponent n is unspecified we mean to imply that it is 0,1,2,3, ...
- Lets show that this language is nonregular.

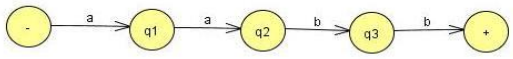
- It is a subset of many regular languages, such as a*b*, which, however, also includes such strings as aab and bb that $\{a^n b^n\}$ does not.
- Suppose this language is regular and some FA must be developed that accepts it.
- Let us picture one of these FA's .This FA might have many states.
- Say that it has 95 states, just for the sake of argument.
- Yet we know it accepts the word $a^{96}b^{96}$. The first 96 letters of this input string are all a's and they trace a path through this hypothetical machine.
- Because there are only 95 states ,The path cannot visit a new state with each input letter read.
- The path returns to a state that it has already visited at some point.
- The first time it was in that state it left by the a-road.
- The second time it is in that state it leaves by the a-road again.
- Even if it only returns once we say that the path contains a circuit in it.

(A circuit is a loop that can be made of several edges.)

- The path goes up to the circuit and then it starts to loop around the circuit, maybe 0 or more times.
- It cannot leave the circuit until, b is read in.
- Then the path can take a different turn. In this hypothetical example the path could make 30 loops around a three-state circuit before the first b is read.
- After the first b is read, the path goes off and does some other stuff following b edges and eventually winds up at a final state where the word $a^{96}b^{96}$ is accepted.
- Let us, say that the circuit that the a-edge path loops 'around has seven states in it. The path enters the circuit, loops around it continuously and then goes off on the b-line to a final state.
- What will happen to the input string $a^{96+7}b^{96}$? Just as in the case of the input string $a^{96}b^{96}$,
- This string would produce a path through the machine that would walk up to the same circuit (reading in only a's) and begin to loop around it in exactly the same way.

- However, the path for $a^{96+7}\, b^{96}$ loops around this circuit one more time than the path for $a^{96}\, b^{96}$ --precisely one extra time.
- Both paths, at exactly the same state in the circuit, begin to branch off on the b-road.
- Once on the b-road, they both go the same 96 b-steps and arrive at the same final state.
- But this would mean that the input string $a^{103}\, b^{96}$ is accepted by this machine.
- But that string is not in the language $L = \{a^n b^n\}$.
- This is a contradiction. We assumed that we were talking about an FA that accepts exactly the words in L and then we were able to prove that the same machine accepts some word that is not in L.

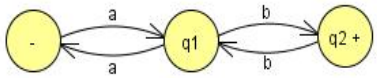
- This contradiction means that the machine that accepts exactly the words in L does not exist.
- In other words, L is nonregular.
- Let us review what happened.
- We chose a word in L that was so large (had so many letters) that its path through the FA had to contain a circuit.
- Once we found that some path with a circuit could reach a final state, we asked ourselves what happens to a path that is just like the first one,
- But that loops around the circuit one extra time and then proceeds identically through the machine.
- Suppose the following FA for a^2b^2



- Only those used in the path of a^2b^2.

Consider the following FA with Circuit….

- However, the path for $a^{13}\, b^9$ still has four more a-steps to take, which is one more time around the circuit, and then it follows the nine b-steps.



- Let us return to.
- With our first assumptions we made above (that there were 95 states and that the circuit was 7 states long), we could also say that $a^{110}b^{96}$, $a^{117}b^{96}$, $a^{124}b^{96}$ ... are also accepted by this machine.
- All can be written the form

$$a^{96}(a^{7})^{m}b^{96}$$

- where m is any integer 0,1,2,3 ..... If m is 0, the path through this machine is the path for the word $a^{96}b^{96}$.
- If m is (1, that looks the same, but it loops the circuit one more time.
- If m 2, the path loops the circuit two more times.
- In general, $a^{96}(a^{7})^{m}b^{96}$ loops the circuit exactly m more times.
- After doing this looping it gets off the circuit at exactly the same place $a^{96}b^{96}$ does and proceeds along exactly the same route to the final state.
- All these words, though not in L, must be accepted.
- Suppose that we had considered a different machine to accept the language L, a machine that has 732 states.
- When we input the word $a^{733}\,b^{733}$,
- the path that the a's take must contain a circuit.
- The word $a^{9999}b^{9999}$ also must loop around a circuit in its a-part of the path.
- Suppose the circuit that the a-part follows has 101 states.
- Then $a^{733+101}\ b^{733}$ would also have to be accepted by this machine, because
- its path is the same in every detail except that it loops the circuit one more time.
- This second machine must also accept some strings, that are not in L:

  $a^{834}\ b^{733}$ $a^{935}\ b^{733}$ $a^{1036}\ b^{733}$ ……

  $= a^{733}\ (a^{101})^{m}\ b^{733}$

PUMPING LEMMA (Version I)

Let L be any regular language that has infinitely many words. Then there exist some three strings x, y, and z (where y is not the null string) such that all the strings of the form xy^nz for n = 1 2 3 ...

Theorem 13

PROOF

- If L is a regular language, then there is an FA that accepts exactly the words in L.
- Let us focus on one such machine.
- Like all FA's, this machine has only finitely many states. But L has infinitely many words in it.
- This means that there are arbitrarily long' words in L. (If there were some maximum on the length of all the words in L, then L could have only finitely many words in total.)

Let w be some word in L that has more letters in it than there are states in the machine we are considering.

- When this word generates a path through the machine, the path cannot visit a new state for each letter because there are more letters than states.
- Therefore, it must at some point revisit a state that it has been to before.
- Theorem 13
- Part 1

All the letters of w starting at the beginning that lead up to the first state that is revisited. Call this part x. Notice that x may be the null string if the path for w revisits the start state as its first revisit.

- Part 2

Starting at the letter after the substring x, let y denote the substring of w that travels around the circuit coming back to the same state the circuit began with.

Since there must be a circuit, y cannot be the null string. y contains the letters of w for exactly one loop around this circuit.

- Part 3

Let z be the rest of w starting with the letter after the substring y and going to the end of the string w. This z could be null.

The path for z could also possibly loop around the y circuit or any other. What z does is arbitrary.

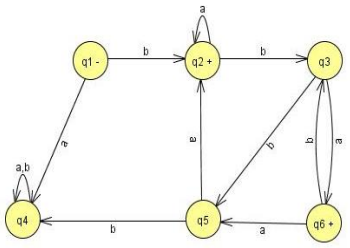
- From the definition of these three substrings

  w = xyz

- w is accepted by this machine.

Example

- Let us illustrate the action of the Pumping Lemma on a concrete example of a regular language.
- The machine below accepts an infinite language and has only six states



Example

- Any word with six or more letters must correspond to a path that includes a circuit.
- Some words with fewer than six letters correspond to paths with circuits, such as baaa.
- The word we will consider in detail is

  w = bbbababa

- Which has more than six letters and therefore includes a circuit.

  Example

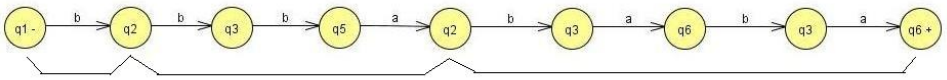
- The path that this word generates through the FA can be decomposed into three stages:
- The first part, the x-part, goes from the - state up to the first circuit.
- This is only one edge and corresponds to the letter b alone.
- The second stage is the circuit around states q2, q3, and q5.

  Example

- This corresponds to edges labeled b, b, and a.

- We therefore say that the substring bba is the y-part of the word w.
- After going around the circuit, the path proceeds to states q3, q6, q3, and q6.
- This corresponds to the substring baba of w, which constitutes the z-part.

w = b bba baba



what would happen to the input string xyyz.

x y y z = b bba bba baba

- Clearly, the x-part (the letter b) would take this path from -- to the beginning of the circuit in the path of w.
- Then the y-part would circle the circuit in the same way that the path for w does when it begins xy.

At this point, we are back at the beginning of the circuit. We mark off the circuit starting at the first repeated state, which in this case is state q2.

- Consider it to be made up of exactly as many edges as it takes to get back there (in this case 3 edges).
- In the original path for w we proceed again from state q2 to state q3 as in the circuit, this is not part of the first simple- looping and so not part of y.
- so, we can string two y-parts together since the second y begins in the state in which the first leaves us.
- The second y-part circles the circuit again and leaves us in the correct state to resume the path of the word w.
- We can continue after xyy with the z path exactly as we continued w with the z path after its y-part.

Lecture # 19

PUMPING LEMMA Example1

- Consider the language

$$a^n b a^n = \{b, aba, aabaa\}$$

- If these languages are regular, then there exists three strings x,y,z such that xyz and xyyz are both words in the language, we can show that this is impossible.
- **Observation 1**

  If the y string contained the b, then xyyz will contain double b's which word in this language can have.
- **Observation 2**

  If y string is all a's, then the b in the middle of the word xyz is in the x-side or z-side.
- In either case xyz is has increased the number a's either in front of the b or after the b.
- But not both.
- **Conclusion 1:**
  - xyyz donot have its b in the middle and not its form $a^n b a^n a$
- **Conclusion 2:**
  - The language cannot be pumped and therfore not regular.
- Example

  Consider the language **$a^n b^n a b^{n+1}$ for n = 1,2,3. . .**
- The first two words of infinite language are ababb aabbabbb, we will show that this language is too not a regular language.
- We will show that if xyz in this language for any three strings x,y,z then xyz is not in this language.

  Observation 1:
- For every word in this language if we know that total number of a's, we can calculate the exact number of b's. and conversely.
- If we know the total number of b's then we can calculate the unique number of a's
- So no two different words have the same number of a's and b's.

Observation 2:

- All words in this language have exactly two substrings equal to ab and one equal to ba.
- **Observation 3:**
- If xyz and xyyz both are in this language, then y cannot contain either the substring ab or the substring ba.
- **Conclusion 1:**
- Because y cannot b ʌ, it must be solid clump of a's and solid clump of b's.

Example

Conclusion 2:

- If y is solid a's then xyz xyyz are different word with the same total b's violating observation 1.
- if y is solid b's then xyz xyyz are different word with the same number of a's violating **observation 1.**
- **Conclusion 3:**
- It is impossible for both xyz xyyz to be in this language for any string xyz.
- Therefore the language is un-pumpable and not regular.

Pumping Lemma Version II

- Let L be an infinite language accepted by a finite automaton with N states.

  Then for all words w in L that have more than N letters there are strings x, y, and z

  where y is not null and where length(x) + length(y) does not exceed N

  such that

  W =xyz

  and all strings of the form

  xy^nz (for n =1 2 3 ...)

  are in L.

Why Pumping Lemma Version II?

Example

- We shall show that the language PALINDROME is nonregular. We cannot use the first version of the Pumping Lemma to do this because the strings

  x=a y=b z=a

- satisfy the lemma and do not contradict the language. All words of the form

  $xy^nz = ab^na$

  are in PALINDROME.

- However, let us consider one of the FA's that might accept this language.

- Let us say that the machine we have in mind has 77 states. Now the palindrome

  $w = a^{80}ba^{80}$

  must be accepted by this machine and it has more letters than the machine has states.

- This means that we can break w into the three parts, x, y, and z. But since the length of x and y must be in total 77 or less, they must both be made of solid a's, since the first 77 letters of w are all a's.

- It means when we form the word xyyz we are adding more a's on to the front of w.

- But we are not adding more a's on to the back of w since all the rear a's are in the z part,

- Which stays fixed at 80 a's. This means that the string xyyz is not a palindrome since it will be of the form $\mathbf{a^{more\ than\ 80}ba^{80}}$

- Which stays fixed at 80 a's. This means that the string xyyz is not a palindrome since it will be of the form

- Why Pumping Lemma Version II?

- It means when we form the word xyyz we are adding more a's on to the front of w.

- But we are not adding more a's on to the back of w since all the rear a's are in the z part,

- Which stays fixed at 80 a's. This means that the string xyyz is not a palindrome since it will be of the form

  $\mathbf{a^{more\ than\ 80}ba^{80}}$

- But the second version of the Pumping Lemma said that PALINDROME has to include. this string.
- Therefore, the second version does not apply to the language PALINDROME, which means that PALINDROME is nonregular.

Example

- Let us consider the language

  PRIME = $\{a^P$ where p is a prime$\}$

  = {aa, aaa, aaaaa, aaaaaaa, ...

- Is PRIME a regular language? If it is, then there is some some FA that accepts exactly these words.
- Let us suppose, that it has 345 states. Let us choose a prime number bigger than 345, for example 347.
- Then a^{347} can be broken into parts x, y, and z such that xy^nz is in PRIME for any value of n.
- The parts x, y, and z are all just strings of a's. Let us take the value of n = 348.
- By the Pumping Lemma, the word $xy^{348}z$ must be in PRIME. Now

  $xy^{348}z$ = X y Z y^{347}

- We can write this because the factors x, y, and z are all solid clumps of a's, and it does not matter in what order we concatenate them.
- All that matters is how many a's we end up with.
- Let us write

  X y Z $y^{347} = a^{347}\, y^{347}$

- This is because x, y, and z came originally from breaking up a^{347} into three parts.
- We also know that y is some (nonempty) string of a's.
- Let us say that

  $y = a^m$ for some integer m that we do not know

  $a^{347}\, y^{347} = a^{347}(a^m)^{347}$

  $= a^{347 + 347m}$

$$= a^{347(m + 1)}$$

- these operations are all standard algebraic manipulations.

- What we have arrived at is that there is an element in PRIME that is of the form a to the power 347(m + 1).
- Now since m is not equal to 0, we know that 347(m+ 1)
- is not a prime number. But this is a contradiction, since all the strings in PRIME are of the form a^p where the exponent is a prime number.
- This contradiction arose from the assumption that PRIME was a regular language.
- Therefore PRIME is nonregular.

Theorem 15

- Given a language L, we shall say that any two strings x and y are in the same class if for all possible strings z either both xz and yz are in L or both are not.

1. The language L divides the set of all possible strings into separate classes.
2. If L is regular the number of classes L creates is finite.
3. If the number of classes L creates , is finite then L is regular.

Theorem 15

Example 1

- Let us consider a language of all words that end with an a , at first it may seem that there is only one class here because for all x and y , both xz and yz end an a or not, depending on z alone.
- But this overlooks the fact that if z is Λ.
- Then xz and yz are in the same class depending on whether x and y end an a themselves.
- There are therefore two classes.
- C_1 = all strings that end an a, a final state.
- C_2 = all strings that don't, the start state.

Example 2

- Let L be a language of all string that contain double a, there are three classes

1. C_1 = strings without aa that end in a.
2. C_2 = strings without aa that end in b or ʎ.
3. C_3 = strings with aa, the final state.
4. Theorem 15

Example 3

- Working the algorithm of theorem 15 on the language EVEN – EVEN creates 4 obvious states.
- C_1 = EVEN-EVEN
- C_2 = even a's , odd b's
- C_3 = odd a's , even b's
- C_4 = odd a's, odd b's
- Clearly if x and y are in any one class, then both xz and yz are in L or not, depending on how many a's and b's z alone has.

Example 4

- To show that the language a^nb^n is not regular, we need to observe that the string a, aa, aaa, aaaa, …… are all in different classes.
- Because for each m only am is turned into a word in L by $z = b^m$
- Theorem 15

Example 5

- To show that $a^nba^{n \text{ is a}}$ nonregular
- We note that the string ab, aab, aaab, . . . All are in different classes because for each of them one value of $z = a^m$ will produce a word in L and leaves the others out of L.

Lecture 20

Non-regular languages

- Many Languages can be defined using FA's.
- These languages have had many different structures, they took only a few basic forms: languages with required substrings, languages that forbid some substrings, languages that begin or end with certain strings,
- languages with certain even/odd properties, and so on.
- Lets take a look to some new forms, such as the language PALINDROME or the language PRIME of all words a^P where p is a prime number.
- Neither of these is a regular language.
- They can be described in English, but they cannot be defined by an FA.
- Conclusion: More powerful machines are needed to define them

Definition

A language that cannot be defined by a regular expression is called a non-regular language.

- Cannot be accepted by any FA or TG.
  (Kleene's theorem,)
- All languages are either regular or nonregular, but not both of them.
- Consider a simple case. Let us define the language L.
- L = {a, ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb . ..}
- We could also define this language by the formula

$$L = \{a^n b^n \text{ for } n = 0\ 1\ 2\ 3\ 4\ 5\ \ldots\}$$

Or short for

$$L = \{a^n b^n\}$$

PUMPING LEMMA (Version I)

Let L be any regular language that has infinitely many words. Then there exist some three strings x, y, and z (where y is not the null string) such that all the strings of the form xy^nz for n = 1 2 3 ...

Theorem 13

PROOF

- If L is a regular language, then there is an FA that accepts exactly the words in L.
- Let us focus on one such machine.
- Like all FA's, this machine has only finitely many states. But L has infinitely many words in it.
- This means that there are arbitrarily long' words in L. (If there were some maximum on the length of all the words in L, then L could have only finitely many words in total.)
- Let w be some word in L that has more letters in it than there are states in the machine we are considering.
- When this word generates a path through the machine, the path cannot visit a new state for each letter because there are more letters than states.
- Therefore, it must at some point revisit a state that it has been to before.

Part 1

All the letters of w starting at the beginning that lead up to the first state that is revisited. Call this part x. Notice that x may be the null string if the path for w revisits the start state as its first revisit.

Part 2

Starting at the letter after the substring x, let y denote the substring of w that travels around the circuit coming back to the same state the circuit began with.

Since there must be a circuit, y cannot be the null string. y contains the letters of w for exactly one loop around this circuit.

Part 3

Let z be the rest of w starting with the letter after the substring y and going to the end of the string w. This z could be null.

The path for z could also possibly loop around the y circuit or any other. What z does is arbitrary.

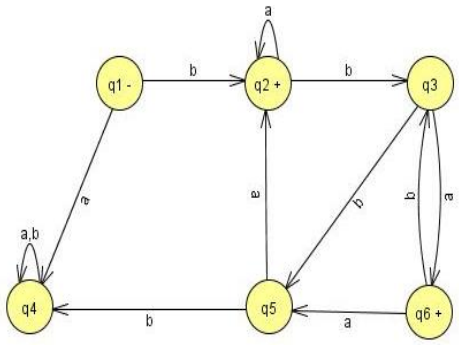
- From the definition of these three substrings

  w = xyz

- w is accepted by this machine.

Example

- Let us illustrate the action of the Pumping Lemma on a concrete example of a regular language.
- The machine below accepts an infinite language and has only six states



- Any word with six or more letters must correspond to a path that includes a circuit.
- Some words with fewer than six letters correspond to paths with circuits, such as baaa.
- The word we will consider in detail is

  w = bbbababa

- Which has more than six letters and therefore includes a circuit.
- The path that this word generates through the FA can be decomposed into three stages:
- The first part, the x-part, goes from the - state up to the first circuit.
- This is only one edge and corresponds to the letter b alone.
- The second stage is the circuit around states q2, q3, and q5.
- This corresponds to edges labeled b, b, and a.

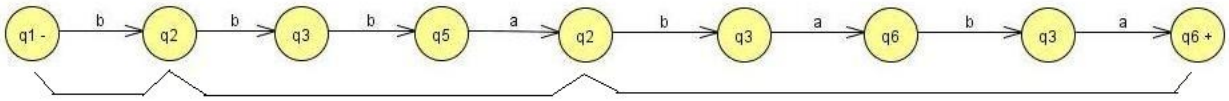
- We therefore say that the substring bba is the y-part of the word w.
- After going around the circuit, the path proceeds to states q3, q6, q3, and q6.
- This corresponds to the substring baba of w, which constitutes the z-part.

w = b bba baba

- what would happen to the input string xyyz.

x y y z = b bba bba baba

- Clearly, the x-part (the letter b) would take this path from -- to the beginning of the circuit in the path of w.
- Then the y-part would circle the circuit in the same way that the path for w does when it begins xy.



PUMPING LEMMA Example2

Example

Consider the language $a^n b^n a b^{n+1}$ **for n = 1,2,3. . .**

- The first two words of infinite language are ababb aabbabbb, we will show that this language is too not a regular language.
- We will show that if xyz in this language for any three strings x,y,z then xyz is not in this language.

Observation 1:

- For every word in this language if we know that total number of a's, we can calculate the exact number of b's. and conversely.
- If we know the total number of b's then we can calculate the unique number of a's
- So no two different words have the same number of a's and b's.

Observation 2:

- All words in this language have exactly two substrings equal to ab and one equal to ba.

- **Observation 3:**
- If xyz and xyyz both are in this language, then y cannot contain either the substring ab or the substring ba.
- **Conclusion 1:**
- Because y cannot b ʌ, it must be solid clump of a's and solid cump of b's.

  Example
- If y is solid a's then xyz xyyz are different word with the same total b's violating observation 1.
- if y is solid b's then xyz xyyz are different word with the same number of a's violating observation 1.
- Conclusion 3:
- It is impossible for both xyz xyyz to be in this language for any string xyz.
- Therefore the language is unpumpable and not regular.
- Let L be an infinite language accepted by a finite automaton with N states.

  Then for all words w in L that have more than N letters there are strings x, y, and z

  where y is not null and where length(x) + length(y) does not exceed N

  such that

  W =xyz

  and all strings of the form

  xy^nz (for n =1 2 3 ...)

  are in L.

Example

- We shall show that the language PALINDROME is nonregular. We cannot use the first version of the Pumping Lemma to do this because the strings

  x=a y=b z=a
- satisfy the lemma and do not contradict the language. All words of the form

  $xy^nz = ab^na$

are in PALINDROME.

- However, let us consider one of the FA's that might accept this language.
- Let us say that the machine we have in mind has 77 states. Now the palindrome

$$w = a^{80}ba^{80}$$

must be accepted by this machine and it has more letters than the machine has states.

- This means that we can break w into the three parts, x, y, and z. But since the length of x and y must be in total 77 or less, they must both be made of solid a's, since the first 77 letters of w are all a's.
- It means when we form the word xyyz we are adding more a's on to the front of w.
- But we are not adding more a's on to the back of w since all the rear a's are in the z part,
- Which stays fixed at 80 a's. This means that the string xyyz is not a palindrome since it will be of the form

$$a^{\text{more than } 80}ba^{80}$$

- Which stays fixed at 80 a's. This means that the string xyyz is not a palindrome since it will be of the form
- But the second version of the Pumping Lemma said that PALINDROME has to include. this string.
- Therefore, the second version does not apply to the language PALINDROME, which means that PALINDROME is nonregular.
- PUMPING Lemma Version II using JFLAP

Lecture # 21

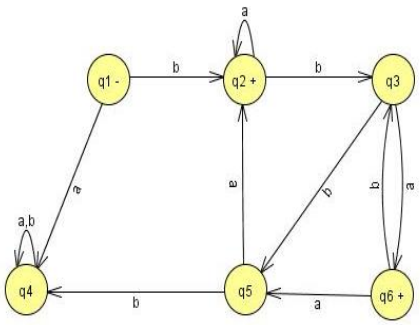
Pumping Lemma in JFLAP and in Daniel I. Cohen Book

- M is not taken as a length of x and y but it other words, it is shown as xy<= m
- Within the string decomposition, y cannot be null, but pumping 0 times, y cannot be empty. This fact is not very much obvious in the book.
- However, the following example, make a clear note on it.

- Pumping Lemma Example in Daniel I cohen Book

Example

- Let us illustrate the action of the Pumping Lemma on a concrete example of a regular language.
- The machine below accepts an infinite language and has only six states



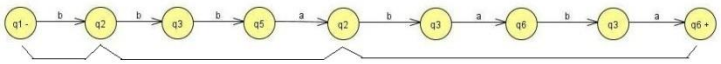
- Any word with six or more letters must correspond to a path that includes a circuit.
- Some words with fewer than six letters correspond to paths with circuits, such as baaa.
- The word we will consider in detail is

w = bbbababa

- Which has more than six letters and therefore includes a circuit.
- The path that this word generates through the FA can be decomposed into three stages:
- The first part, the x-part, goes from the - state up to the first circuit.
- This is only one edge and corresponds to the letter b alone.

- The second stage is the circuit around states q2, q3, and q5.
- This corresponds to edges labeled b, b, and a.
- We therefore say that the substring bba is the y-part of the word w.
- After going around the circuit, the path proceeds to states q3, q6, q3, and q6.
- This corresponds to the substring baba of w, which constitutes the z-part.

w = b bba baba



- what would happen to the input string xyyz.

x y y z = b bba bba baba

- Clearly, the x-part (the letter b) would take this path from -- to the beginning of the circuit in the path of w.
- Then the y-part would circle the circuit in the same way that the path for w does when it begins xy.

Pumping Lemma Version II from book

- Let L be an infinite language accepted by a finite automaton with N states.

  Then for all words w in L that have more than N letters there are strings x, y, and z

  where y is not null and where length(x) + length(y) does not exceed N

  such that

  W =xyz

  and all strings of the form

  xy^nz (for n =1 2 3 ...) are in L.

Why we needed Pumping Lemma Version II?

- Example
- We shall show that the language PALINDROME is nonregular. We cannot use the first version of the Pumping Lemma to do this because the strings

x=a y=b z=a

- satisfy the lemma and do not contradict the language. All words of the form

  $xy^nz = ab^na$

  are in PALINDROME.

Lecture # 22

- Theory Of Automata

JFLAP for Pumping Lemma Version II

- Practical Demonstrations

• Context Free Grammar

- For earliest computers . Every procedure, no matter how complicated, had to be spelled out in the crudest set of instructions:
- They accepted no instructions except those in their own machine language.
- It could take dozens of these primitive instructions to do anything useful.
- It could take quite a few to evaluate one complicated arithmetic expression.

• For example to calculate

$$S = \frac{\sqrt{(7-5)^2 + (4-8)^2 + (9-5)^2}}{2}$$

- This clearly required that some "higher-level" language be invented-a language in which one mathematical step, such as evaluating the formula above, could be converted into one single computer instruction.
- A super program called the compiler does the conversion from a high-level language into assembler language code.
- It cannot just look at the expression and understand it.
- Rules must be given by which this string can be processed.
- Also we want our machine to be able to reject strings of symbols that make no sense as arithmetic expressions.

  Like "((9)+".
- This input string should not take us to a final state in the machine.
- However, we cannot know that this is a bad input string until we have reached the last letter.
- If the + were changed to a), the formula would be valid.

- An FA that translated expressions into instructions as it scanned left to right would already be turning out code before it realized that the whole expression is nonsense.

- Rules for valid arithmetic expressions.

  Rule 1 Any number is in the set AE.

  Rule 2 If x and y are in AE then so are:

  (i) -(x) (x + y) (x - y) (x*y) (x/y) (x**y)

- For example, the input string((3 + 4) * (6 + 7))
- The way this can be produced from the rules is by the sequence

  3 is in AE

  4 is in AE

  (3 + 4) is in AE

  6 is in AE

  7 is in AE

  (6 + 7) is in AE

  ((3 + 4) * (6 + 7)) is in AE

- Convert this into

  LOAD 3 in Register 1

  LOAD 4 in Register 2

  ADD the contents of Register 2 into Register 1

  LOAD 6 in Register 3

  LOAD 7 in Register 4

  ADD the contents of Register 3 into Register 4

  MULTIPLY Register 1 by Register 4

- It was realized by earlier high-level languages designers that this problem is analogous to the problem humans have to face hundreds of times every day when they must decipher the sentences that they hear or read in English.

- "grammar" -- Study of grammar as well as the set of rules themselves, we sometimes refer to the set of rules as forming a "generative grammar."
- According to the rules in English grammar that allows us to form a sentence by juxtaposing a noun and a verb.
- We might produce:Birds sing.
- However, using the same rule might also produce:

  Wednesday sings.

  or Coal mines sing.
- Semantics :
  - Rules that involve the meaning of words.
- Syntax
  - Rules that do not involve the meaning of words.
- In English the meaning of words can be relevant but in arithmetic the meaning of numbers is easy to understand.
- Context Free Grammar
- In the high-level computer languages, one number is as good as another. If

  X= B + 9

  is a valid formulation then so are

  X = B + 8 X = B + 473 X = B + 9999
- So long as the constants do not become so large that they are out of range,
- We do not try to divide by 0, to take the square root of a negative number, and to mix fixed-point numbers with floating- point numbers in bad ways

  In general, the rules of computer language grammar are all syntactic and not semantic.
- In reality a law of grammar is a suggestion for possible substitutions.
- Started out with the initial symbol Sentence.
- Then the rules for producing sentences listed in the generative grammar are applied.
- In most cases one have some choice in selecting which rule she wanted to apply.

- Terminals :
  - The words that cannot be replaced by anything .
- Nonterminal:
  - Words that must be replaced by other things we call nonterminals.
- Through the production procedure we developed the sentence into as many nonterminals as it was going to become.
- Context Free Grammar
- For defining arithmetic expressions We follow the same model.
- We can write the whole system of rules of formation as the list of possible substitutions

  Start → (AE)

  AE → (AE + AE)

  AE → (AE - AE)

  AE → (AE *AE)

  AE → (AE AE)

  AE → (AE ** AE)

  AE → (AE)

  AE → (AE)

  AE → ANY-NUMBER
- Context Free Grammar
- The word "Start" is used to begin the process, as the word "Sentence“ was sued in the example of English.
- Aside from Start, the only other nonterminal is AE.
- The terminals are the phrase "any number" and the symbols

  + - * /** ( )
- We are satisfied by knowing what is meant by the words "any number“.
- OR else we could define this phrase by a set of rules, thus converting it from a terminal into a nonterminal.

Rule 1: ANY-NUMBER ---> FIRST-DIGIT

Rule 2: FIRST-DIGIT ---> FIRST-DIGIT OTHER-DIGIT

Rule 3: FIRST-DIGIT--* 123456789

Rule 4: OTHER-DIGIT-->0 123456789

- Rules 3 and 4 offer choices of terminals.
- Spaces between them have been placed to indicate "choose one,".
- We can produce the number 1066 as follows:

ANY-NUMBER → FIRST-DIGIT (Rule 1)

FIRST-DIGIT OTHER-DIGIT (Rule 2)

FIRST-DIGIT OTHER-DIGIT OTHER-DIGIT (Rule 2)

FIRST-DIGIT OTHER-DIGIT (Rule 2)

OTHER-DIGIT OTHER-DIGIT1066 (Rules 3 and 4)

- The sequence of applications of the rules that produces the finished string of terminals from the starting symbol is called a derivation.
- The grammatical rules are often called productions.
- They all indicate possible substitutions.
- The derivation may or may not be unique, i.e by applying productions to the start symbol in two different ways we may still produce the same finished product.

• CFG Definition:

A context-free grammar, is a collection of three things:

1 An alphabet Σ of letters called terminals from which we are going to make strings that will be the words of a language.

2 A set of symbols called nonterminals, one of which is the symbol S, standing for "start here.“

3 A finite set of productions of the form one nonterminal → finite string of terminals and/or nonterminals

Context Free Grammar

- The strings of terminals and nonterminals can consist of only terminals (designated by lowercase letters and special symbols) or of only nonterminals(designated by capital letters), or any mixture of terminals and nonterminals or even the empty string.
- At least one production has the nonterminal S as its left side.

- Context Free Language

Definition

- A language generated by a CFG is called a context-free language, abbreviated CFL.
- CFL is the set of all strings of terminals that can be produced from the start symbol S using the productions as substitutions.

Example 1

- Let the only terminal be a.
- Let the productions be:

PROD 1 $S \rightarrow aS$

PROD 2 $S \rightarrow \Lambda$

- If we apply Production 1 six times and then apply Production 2 (see Next Slide)
- Context Free Language
- we generate the following:

S => aS

=> aaS

=> aaaS

=> aaaaS

=> aaaaaS

=> aaaaaaS

=> aaaaaa Λ

= aaaaaa

This is a derivation of a^6 in this CFG. The string a^n comes from n applications of Production 1 followed by one application of Production 2.

- If Production 2 is applied without Production 1, we find that the null string is itself in the language of this CFG.
- Context Free Language
- Since the only terminal is a it is clear that no words outside of a* can possibly be generated.
- The language generated by this CFG is exactly a*.
- The symbol → means we employ in the statement of the productions. It means "can be replaced by" as in S → aS
- Context Free Language
- Since the only terminal is a it is clear that no words outside of a* can possibly be generated.
- The language generated by this CFG is exactly a*.
- The symbol → means we employ in the statement of the productions.
- The other arrow symbol => we employ between the unfinished stages in the generation of our word.
- It means "can develop into" as in aaS → aaaS. These "unfinished stages" are strings of terminals and non-terminals that we shall call working strings.
- We have both S → aS as a production in
- the abstract and S => aS as the first step in a particular derivation.

Example 2

- Let the only terminal be a.
- Let the productions be:

  PROD 1 S → SS

  PROD 2 S → a

  PROD 3 S → ʌ

- In this language we can have the following derivation.

S => SS

=> SSS

=> SaS

=> SaSS

=> ʌ aSS

=> ʌaaS

=> ʌaaʌ

= aa

- It is also just the language a*, but here the string aa can be obtained in many ways.
- In previous example there was a unique way to produce every word in the language. This also illustrates that the same language can have more than one CFG generating it.
- Above that there are two ways to go from SS to SSS-either of the two S's can be doubled.
- In the previous example the only terminal is a and the only nonterminal is S.
- What then is ʌ? It is not a nonterminal since there is no production of the form

  ʌ → something
- As always, ʌ is a very special symbol and has its own status.

Lecture # 23

Context Free Language

Example 3

- Let the terminals be a and b, the only nonterminal be S, and
- Productions

PROD 1 S → aS

PROD 2 S → bS

PROD 3 S → a

PROD 4 S →b

- The word baab can be produced as follows:

S => bS (by PROD 2)

=> baS (by PROD 1)

=> baaS (by PROD 1)

=> baab (by PROD 4)

- Productions 3 and 4 can be used only once and only one of them can be used.
- E.g, to generate babb we apply in order Prods 2, 1, 2, 4, as below:

S => bS => baS => babS => babb

- Example 4

- Let the terminals be a and b, nonterminals be S, X, and Y.
- The productions are:

S → X

S → y

X → ʌ

Y→aY

$Y \rightarrow bY$

$Y \rightarrow a$

$Y \rightarrow b$

$S \rightarrow X$

$S \rightarrow y$

$X \rightarrow \Lambda$

$Y \rightarrow aY$

$Y \rightarrow bY$

$Y \rightarrow a$

$Y \rightarrow b$

- All the words in this language are either of type X, if the first production in their derivation is $S \rightarrow X$
- or of type Y, if the first production in their derivation is $S \rightarrow Y$
- The only possible continuation for words of type X is the production $X \rightarrow \Lambda$
- Therefore Λ is the only word of type X.

$S \rightarrow X$

$S \rightarrow y$

$X \rightarrow \Lambda$

$Y \rightarrow aY$

$Y \rightarrow bY$

$Y \rightarrow a$

$Y \rightarrow b$

- The productions whose left side is Y form a collection identical to the productions in the previous example except that the start symbol S has been replaced by the symbol Y.
- We can carry on from Y the same way we carried on from S before.
- This does not change the language generated, which contains only strings of terminals.

- Example 5

- Let the terminals be a and b. Let the only nonterminal be S.
- Let the productions be

S → aS

S → bS

S → a

S → b

S → ʌ

S → aS

S → bS

S → a

S → b

S → ʌ

- The word ab can be generated by the derivation

S => aS

=> abS

=> ab ʌ

= ab

- or by the derivation

S => aS

=> ab

- The language of this CFG is also (a + b)*, but the sequence of productions that is used to generate a specific word is not unique.

Example 6

- Let the terminals be a and b, nonterminals be S and X, and the productions be

$S \rightarrow XaaX$

$X \rightarrow aX$

$X \rightarrow bX$

$X \rightarrow a$

$X \rightarrow b$

$S \rightarrow XaaX$

$X \rightarrow aX$

$X \rightarrow bX$

$X \rightarrow a$

$X \rightarrow b$

- If the nonterminal X appears in any working string we can apply productions to turn it into any word we want.
- Therefore, the words generated from S have the form

  anything aa anything

  or

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

Example 7

- Let the terminals be a and b, nonterminals be S, X, and Y the productions be

$S \rightarrow XY$

$X \rightarrow aX$

$X \rightarrow bX$

$X \rightarrow a$

$Y \rightarrow Ya$

$Y \rightarrow Yb$

$Y \rightarrow a$

- What can be derived from X? Let us look at the X productions alone.

  X → aX

  X → bX

  X → a

- X => bX => baX => babX => babbX => babba
- Context Free Language
- Similarly, the words that can be derived from Y are exactly those that begin with an a.
- To derive abbab,

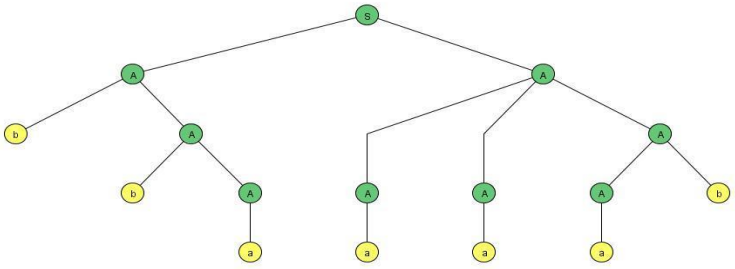
  Y => Yb => Yab => Ybab => Ybbab => abbab

Tree format for CFG

- In old-fashioned English grammar courses students were often asked to diagram a sentence.
- They were to draw a parse tree, which is a picture with the base line divided into subject and predicate.
- Start with the symbol S.
- Every time a production is used to replace a nonterminal by a string, draw downward lines from the nonterminal to each character in the string.
- The CFG

  S → AA

  A → AAA I bA I Ab I a



- We begin with S and apply the production

  S → AA.

- To the left-hand A the production A → bA is applied.
- To the right-hand A apply A → AAA is applied.
- The b that we have on the bottom line is a terminal, so it does not descend further.
- In the terminology of trees it is called a terminal node.

Ambiguity

DEFINITION

- A CFG is called ambiguous if for at least one word in the language that it generates there are two possible derivations of the word that correspond to different syntax trees.

CFG

Example 1

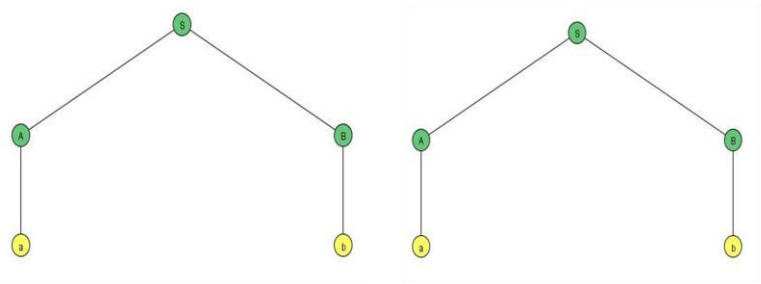
- Consider the language generated by the CFG:

  PROD I S→ AB

  PROD 2 A → a

  PROD 3 B → b

- There are two different sequences of applications of the productions that generate the word ab. One is PROD 1, PROD 2, PROD 3.
- The other is PROD 1, PROD 3, PROD 2.
- S => AB => aB => ab or
- S => AB => Ab => ab
- When a corresponding syntax trees is drawn we see that the two derivations are essentially the same:



23

Example 2

- Consider the language PALINDROME, define by
- the CFG below:

  S → aSa I bSb a I b| ʌ

- At every stage in the generation of a word by this grammar the working string contains only the one nonterminal S smack dab in the middle.
- The word grows like a tree from the center out.

• Example 2

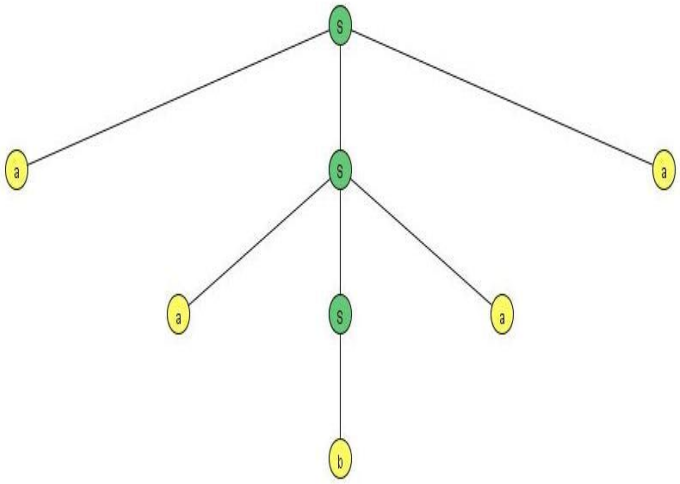
... baSab => babSbab== babbSbbab => babbaSabbab ...

- When we finally replace S by a center letter (or A if the word has no center letter) we have completed the production of a palindrome.
- The word aabaa has only one possible generation:

  S => aSa

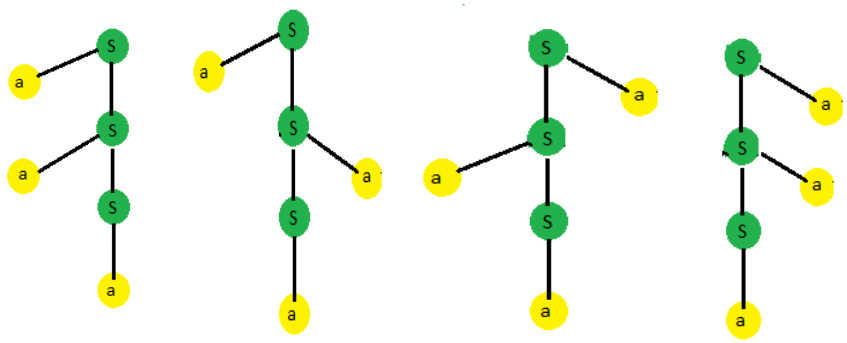
  => aaSaa

  => aabaa



- EXAMPLE 3
- The language of all not-null strings of a's can be defined by a CFG as follows:

  S → aS I Sa I a
- In this case the word a^3 can be generated by four different trees Shown on next slide
- CFG Ambiguity



- This CFG is therefore ambiguous
- However the same language can also be defined by the CFG:

  S → aS I a
- Introduction to CFG in JFLAP
- JFLAP GRAMMAR Menu

Lecture # 24

Unrestricted Grammars

- An unrestricted grammar is similar to a context-free grammar (CFG), except that the left side of a production may contain any nonempty string of terminals and variables, rather than just a single variable.
- In a CFG, the single variable on the left side of a production could be expanded in a derivation step. In an unrestricted grammar, multiple variables and/or terminals that match the left-side of a production can be expanded by the right-side of that production.
- Unrestricted Grammar Example

S	→	AX
Db	→	bD
A	→	aAbc
DX	→	EXc
A	→	aBbc
BX	→	ʌ
Bb	→	bB
cE	→	Ec

CFG with JFLAP

- Unrestricted Grammar with JFLAP
- User Controlled Parsing with JFLAP

Semi words

Definition

- For a given CFG a semiword is a string of terminals (maybe none) concatenated with exactly one nonterminal (on the right), for example,

  (terminal) (terminal) . . . (terminal) (Nonterminal)

Relationship between regular languages and context-free grammars?

1. All regular languages can be generated by CFG's, and so can some nonregular languages but not all possible languages.
2. Some regular languages can be generated by CFG's and some regular languages cannot be generated by CFG's. Some nonregular languages can be generated by CFG's and some nonregular languages cannot.

- DEFINITION
- A CFG is called a regular grammar if each of its productions is of one of the two forms

  Nonterminal → semiword

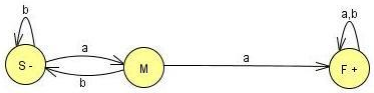
  or

  Nonterminal → word

- According to previous theorems all regular languages can be generated by regular grammars and all regular grammars generate regular languages.
- Despite both theorems it is still possible that a CFG that is not in the form of a regular grammar can generate a regular language.

FA Conversion to Regular grammar

- Accepts the language of all words with a double a:



S →aM

S →bS

M →aF

M →bS

F →aF

F →bF

F →ʌ

Example 1

	S
S→bS	bS
S→aM	baM
M→bS	babS
S→bS	babbS
S→aM	babbaM
M→aF	babbaaF
F→bF	babbaabF
F→aF	babbaabaF
F→λ	babbaaba

The word babbaaba is Accepted by FA Through sequence of this semi paths.

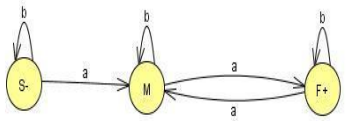
Example 1

- The language of all words with an even number of a's is accepted by the FA:

  S → aM I bS

  M → bS | aF

  F → aF I bF | λ



EXAMPLE 2

- Consider the CFG:

  S → aaS I bbS I λ

- There is only one nonterminal, S, so there will be only two states in the TG, - and the mandated +.

- The only production of the form

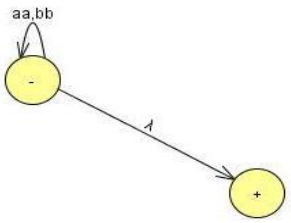
  $N_p \rightarrow w_q$ (Nonterminal to word)

  is S → ʎ ,

- So there is only one edge into + and that is labeled A.

- The productions S - aaS and S → bbS are of the form $N_1 \rightarrow wN_2$ where the N's are both S.

- Since these are supposed to be made into paths from N, to N2 they become loops from S back to S.



- These two productions will become two loops at one labeled aa and one labeled bb.

EXAMPLE 3

- Consider the CFG:

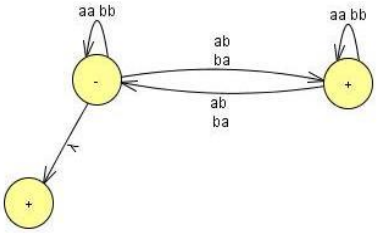
$$S \rightarrow aaS \mid bbS \mid abX \mid baX \mid \lambda$$

$$X \rightarrow aaX \mid bbX \mid abS \mid baS$$

- The algorithm tells us that there will be three states: -, X, +.
- Since there is only one production of the form

$$N_p \rightarrow w_q$$

- TG is



EXAMPLE 4

- Consider the CFG:

$$S \rightarrow aA \mid bB$$

$$A \rightarrow aS \mid a$$

$$B \rightarrow bS \mid b$$

- This language can be defined by the regular expression $(aa + bb)^+$.
- It does not have any productions of the form

$$N_x \rightarrow \Lambda$$

Important Point

For a CFG to accept the word Λ, it must have at least one production of this form, called a Λ-production.

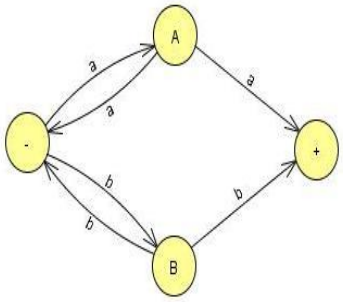
- A Λ-production need not imply that Λ is in the language, as with

S → aX

X → ʌ

Regular Grammar

- The corresponding TG:



S → aA | bB

A → aS | a

B → bS | b

- Regular Grammar Conversion to FA using JFLAP

Lecture 25

Elimination of null production Theorem

Statement:

If L is a context-free language generated by a CFG that includes Λ -productions then there is a different context-free grammar that has no Λ -productions that generates either the whole language L (if L does not include the word Λ) or else generates the language of all the words in L that are not Λ.

Proof (by Example)

- Consider CFG for EVENPALINDROME (the
- palindromes with an even number of letters):

  S → aSa I bSb I Λ

- Following possible derivation:

  S => aSa

  => aaSaa

  => aabSbaa

  => aabbaa

- When we apply this replacement rule to the following CFG.

  S → aSa I bSb I Λ

- We remove the production S → Λ and replace it with S → aa and S → bb,
- These are the first two productions with the right-side S deleted.
- The CFG is now:

  S → aSa I bSb I aa I bb

- Which also generates EVENPALINDROME, except for the word Λ, which can no longer be derived.
- For example, the derivation is generated in the old CFG:
- OLD CFG:

Derivation	Production Used
S => aSa	S → aSa
=> aaSaa	S → aSa
=> aabSbaa	S → bSb
=> aabbaa	S →Λ

- New CFG : We can combine the last two steps .

Derivation	Production Used
S => aSa	S → aSa
=> aaSaa	S → aSa
=> aabbaa	S → bb

Null Production Elimination

EXAMPLE

- Consider the CFG for the language defined by (a + b)*a

S → Xa

X → aX I bX I Λ

- The only nullable nonterminal here is X,
- The productions that have right sides including X are:

Productions with Nullables

S → Xa

X → aX

X → bX

New Productions Formed by the Rule

S → a

X → a

X → b

- The full new CFG is:

S → Xa I a

X → aX | bX | a I b

- To produce the word baa we formerly used the derivation shown In the table.

Derivation	Production Used
S => Xa	S → Xa
=> bXa	X → bX
=> baXa	X → aX
=> baa	X →Λ

- Combine the last two steps, and the new derivation in the new CFG is:

Derivation	Production Used
S => Xa	S → Xa
=> bXa	X → bX
=> baa	X →a

- Consider this CFG for the language defined by (a + b)*bb(a + b)*

S → XY

X → Zb

Y → bW

Z → AB

W → Z

A → aA I bA I ʌ

B → Ba | Bb| ʌ

The modified replacement algorithm tells us to generate new productions to replace the ʌ -productions as follows:

Old	New Productions
X → Zb	X → b
Y → bW	Y → b
Z → AB	Z → A and Z → B
W → Z	Nothing new

A → aA A → a

A → bA A → b

B → Ba B → a

B → Bb B → b

- We do not eliminate all of the old productions, only the old A-productions.
- The fully modified new CFG is:

S → XY

X → Zb | b

Y → bW | b

Z → AB| A I B

W → Z

A → aA | bA | a | b

B → Ba | Bb | a | b

Eliminate Unit Productions

DEFINITION

A production of the formone Nonterminal → one Nonterminal

is called a unit production.

Bar-Hillel, Perles, and Shamir tell us how to get rid of these too.

EXAMPLE

- Consider

S → A Ibb

A → B Ib

B → S | a

- Separate the units from the nonunits:.

Unit Production	Other ones
S → A	S → bb
A → B	A → b
B → S	B → a

Killing Unit Productions

EXAMPLE

- We create the new productions that allow the first nonterminal to be replaced by any of the strings that could replace the last nonterminal in the sequence.

S → A	gives	S → b
S → A → B	gives	S → a
A → B	gives	A → a
A → B → S	gives	A → bb
B → S	gives	B → bb
B → S → A	gives	B → b

Chomsky Normal form

If L is a language generated by some CFG, then there is another CFG that generates all the non-Λ words of L, all of whose productions are of one of two basic forms:

Nonterminal.-- string of only Nonterminals

or

Nonterminal - one terminal

Example 1

- Let start with the CFG:

$S \rightarrow X \mid X_2aX_2 \mid aSb \mid b$

$X_1 \rightarrow X_2X_2 \mid b$

$X_2 \rightarrow aX_2 \mid aaX_1$

- After the conversion we have:

$S \rightarrow X_1$ $\quad\quad$ $X_1 \rightarrow X_2X_2$

$S \rightarrow X_2AX_2$ $\quad\quad$ $X_1 \rightarrow B$

$S \rightarrow ASB$ $\quad\quad$ $X_2 \rightarrow AX_2$

$S \rightarrow B$ $\quad\quad$ $X_2 \rightarrow AAX_1$

$A \rightarrow a$

$B \rightarrow b$

- We have not employed the disjunction slash I but instead have written out all the productions separately so that we may observe eight of the form:

Nonterminal → string of Nonterminals

- and two of the form:

Nonterminal → one terminal

EXAMPLE 2

- Why not simply replace all a's in long strings by this Nonterminal? For instance, why cannot

$S \rightarrow Na$

$N \rightarrow alb$

become

$S \rightarrow NN$

$N \rightarrow a \mid b$

- The answer is that bb is not generated by the first grammar but it is by the second.
- The correct modified form is

$S \rightarrow NA$

$N \rightarrow alb$

$A \rightarrow a$

EXAMPLE 3

- The CFG

$S \rightarrow XY$

$X \rightarrow XX$

$Y \rightarrow YY$

$Y \rightarrow a$

$Y \rightarrow b$

- (which generates aa*bb*),
- If we mindlessly attacked it with our algorithm, become:

$S \rightarrow XY$

$X \rightarrow XX$

$y \rightarrow yy$

$X \rightarrow A$

$Y \rightarrow B$

$A \rightarrow a$

$B \rightarrow b$

- With our algorithm, become:

$S \rightarrow XY$

$X \rightarrow XX$

$Y \rightarrow yy$

$X \rightarrow A$

$Y \rightarrow B$

$A \rightarrow a$ Needless Unit Productions

$B \rightarrow b$

Chomsky Normal form

Definition

- If a CFG has only production of the form

  Nonterminal → strings of exactly two nonterminals

- Or of the form

  Noneterminal → one terminal

- It is used to be Chomsky Normal Form

Example 4

- Let us convert

  S---> aSa I bSb I a | b I aa I bb

- First separate the terminals from the nonterminal as in T

  S → ASA

  S → >BSB

  S → AA

  S → BB

  S → a

  S → b

  A → a

  B → b

- Chomsky Normal form Conversion

- We are careful not to introduce the needless unit productions

  S → A and S → B.

- Now we introduce the R's:

  S → AR $_1$ S → AA

  R $_1$ → SA S → BB

$S \rightarrow BR_2$

$R_2 \rightarrow SB$

$S \rightarrow a$

$S \rightarrow b$

$A \rightarrow a$

$B \rightarrow b$

Example 5

- Convert the CFG into CNF.

$S \rightarrow bA \ I \ aB$

$A \rightarrow bAA \mid aS \mid a$

$B \rightarrow aBB \mid bS \mid b$

- The grammar becomes:

$S \rightarrow YA$

$S \rightarrow XB$

$A \rightarrow YAA$

$A \rightarrow XS$

$A \rightarrow a$

$B \rightarrow XBB$

$B \rightarrow YS$

$B \rightarrow b$

$X \rightarrow a$

$Y \rightarrow b$

- We have left well enough alone in two instances:

$A \rightarrow a$ and $B \rightarrow b$

- We need to simplify only two productions:

$A \rightarrow YAA$ becomes $\{A \rightarrow YR_{1,}\ R_1 \rightarrow AA\}$

- and

$B \rightarrow XBB$ becomes $\{ B \rightarrow XR_2, R_2 \rightarrow BB\}$

- The CFG has now become:

$S \rightarrow YA \ I \ XB$

$A \rightarrow YR_1 \mid XS \mid a$

$B \rightarrow XR_2 \mid YS \mid b$

$X \rightarrow a$

$Y \rightarrow b$

$R_1 \rightarrow AA$

$R_2 \rightarrow BB$

Example 6

- Consider the CFG

$S \rightarrow aaaaS \mid aaaa$

- Which generates the language

a^{4n} for n = 1 2 3....= $\{a^4, a^8, a^{12} ... \}$

- Convert this to CNF as follows:

$S \rightarrow AAAAS$

$S \rightarrow AAAA$

$A \rightarrow a$

Which in turn becomes

$S \rightarrow AR_1$

$R_1 \rightarrow AR_2$

$R_2 \rightarrow AR_3$

$R_3 \rightarrow AS$

$S \rightarrow AR_4$

$R_4 \rightarrow AR_5$

$R_5 \rightarrow AA$

A → a

Left Most Derivation

If a word w is generated by a CFG by a certain derivation and at each step in the derivation a rule of production is applied to the leftmost nonterminal in the working string, then this derivation is called a leftmost derivation.

Example

- Consider the CFG:

  S → aSX I b

  X → Xb Ia

- Left most derivation is

  S => aSX

  =>aaSXX

  =>aabXX

  => aabXbX

  => aababX

  => aababa

Lecture # 26

A new Format for FAs

- We will start with our old FA's and throw in some new diagrams that will augment them and make them more powerful.
- In this chapter complete new designs will be created for modeling Fas.

New terminologies

- We call it INPUT TAPE to the part of the FA where the input string lives while it is being run.
- The INPUT TAPE must be long enough for any possible input, and since any word in a* is a possible input, the TAPE must be infinitely long.
- The TAPE has a first location for the first letter of the input, then a second location, and so on.
- Therefore, we say that the TAPE is infinite in one direction only.

A new Format for FAs

- The locations into which put the input letters are called cells. (See table below)

a	**a**	**b**	**Δ**	**Δ**	

- Name the cells with lowercase Roman numerals.
- The Δ used to indicate the blank
- Input string is aab

Input tape parsing

- As TAPE is processed, on the machine we read one letter at a time and eliminate each as it is used.
- When we reach the first blank cell we stop.
- We always presume that once the first blank is encountered the rest of the TAPE is also blank.
- We read from left to right and never go back to a cell that was read before.

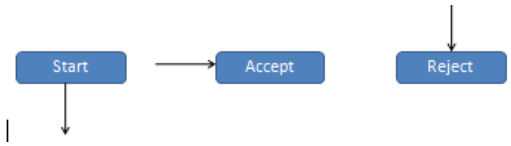
New symbols for FA

- To streamline the design of the machine, some symbols are used.

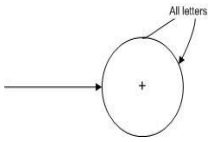
- The arrows (directed edges) into or out of these states can be drawn at any angle. The START state is like a state connected to another state in a TG by a Λ edge.
- We begin the process there, but we read no input letter. We just proceed immediately to the next state.

A new Symbol for FAs

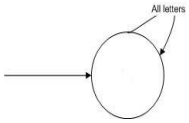
- A start state has no arrows coming into it.



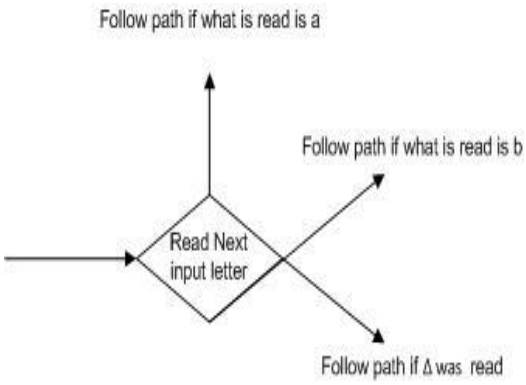
- An ACCEPT state is a shorthand notation for a dead-end final state-once entered, it cannot be left, shown:

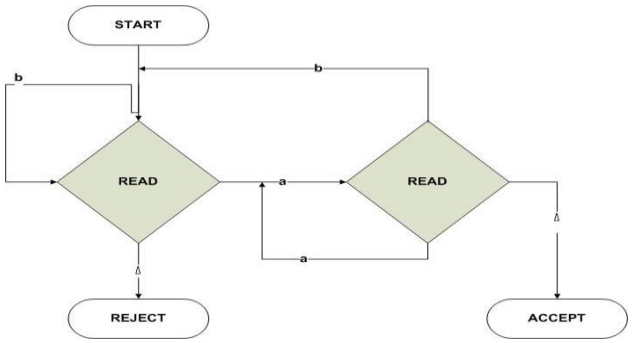


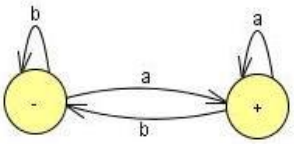
- A REJECT state is a dead-end state that is not final



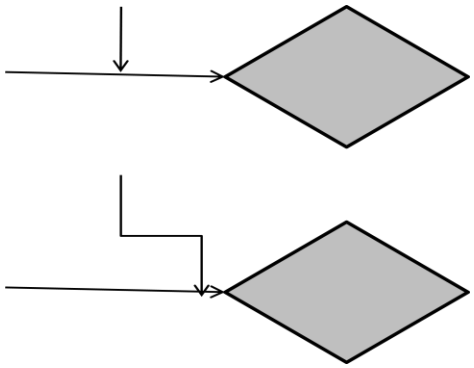
- READ states are introduced.
- These are depicted as diamond shaped boxes



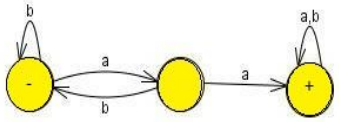




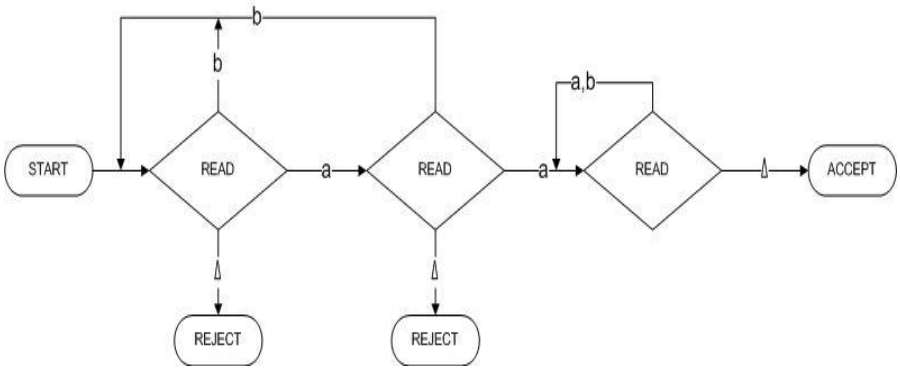
- We have also used the electronic diagram notation for wires flowing into each other.



- We have also used the electronic diagram notation for wires flowing into each other.

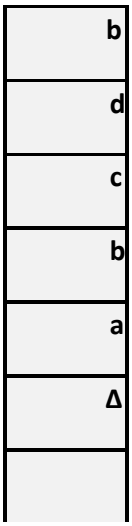


Becomes

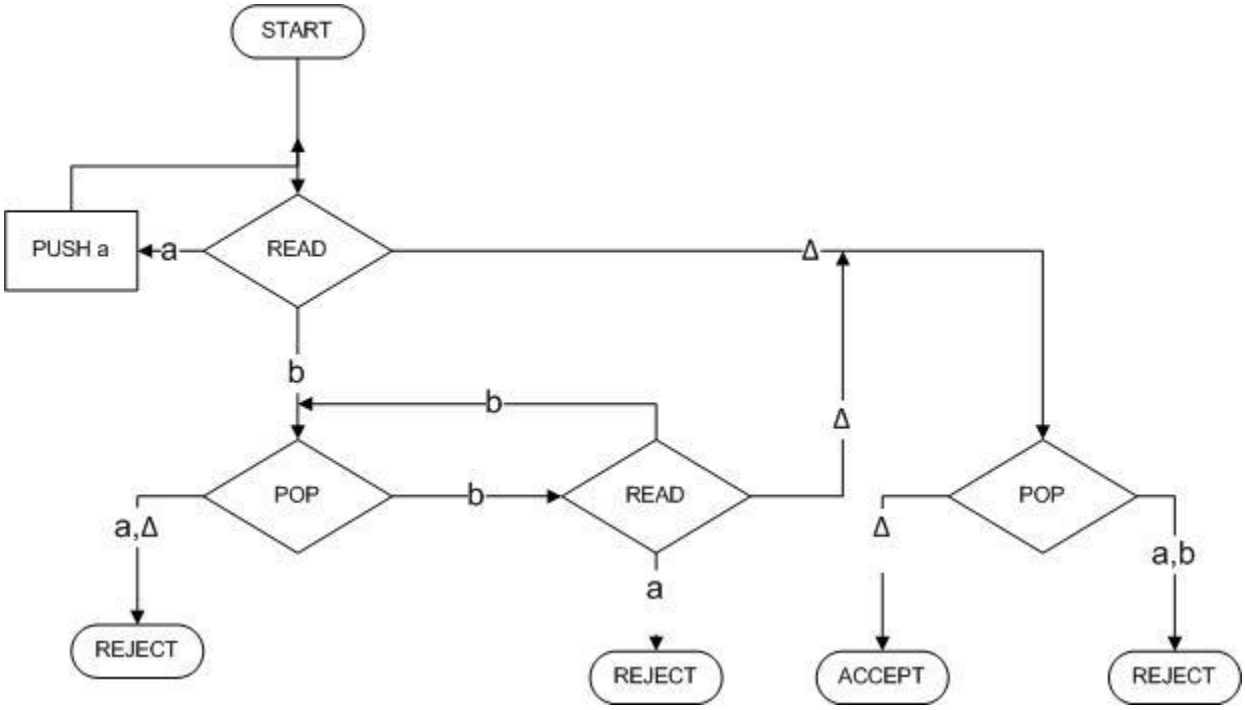


Adding A Pushdown Stack

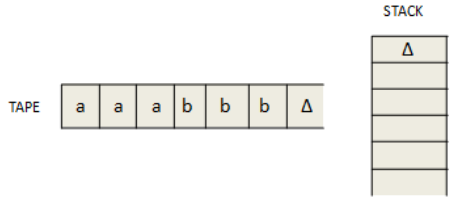
- A PUSHDOWN STACK is a place where input letters can be stored until we want to refer to them again.
- It holds the letters it has been fed in a long line. The operation PUSH adds a new letter to the line.
- The new letter is placed on top of the STACK, and all the other letters are pushed back (or down) accordingly.
- Before the machine begins to process an input string the STACK is presumed to be empty, which means that every storage location in it initially contains a blank.
- If the STACK is then fed the letters a, b, c, d by this sequence of instructions:
- PUSH a
- PUSH b
- PUSH c
- PUSH d
- Then top letter in the STACK is d, the second is c, the third is b, and the fourth is a.
- If we now execute the instruction:
- PUSH b the letter b will be added to the STACK on the top. The d will be pushed down to position 2, the c to position 3, the other b to position 4, and the bottom a to position 5.
- One pictorial representation of a STACK with these letters in it is shown below.
- Beneath the bottom a we presume that the rest of the STACK, which, like the INPUT TAPE, has infinitely many storage locations, holds only blanks.

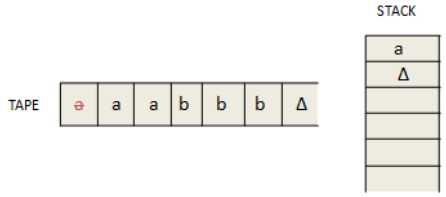


- How following PDA is working:

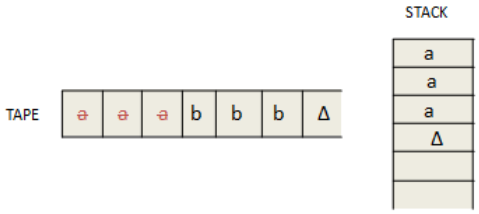


- Its operation on the input string aaabbb.
- We begin by assuming that this string has been put on the TAPE.
-

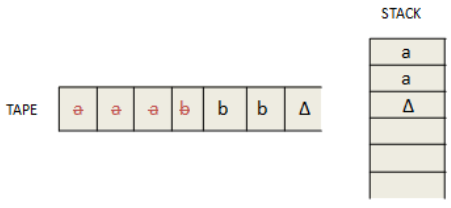




- We now read another a and proceed as before along the a edge to push it into the STACK.
- Again we are returned to the READ box.
- Again we read an a (our third), and again this a is pushed onto the STACK.

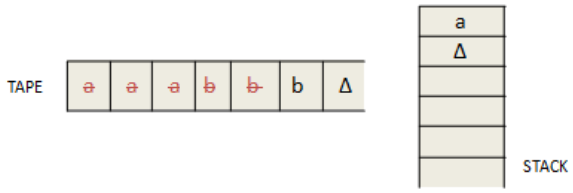


- After the third PUSH a, we are routed back to the same READ state again.
- This time, we read the letter b.
- This means that we take the b edge out of this state down to the lower left POP.

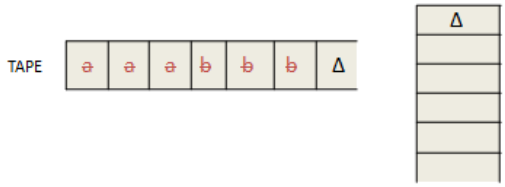


- The b road from the second READ state now takes us back to the edge feeding into the POP state.
- So we pop the STACK again and get another a.

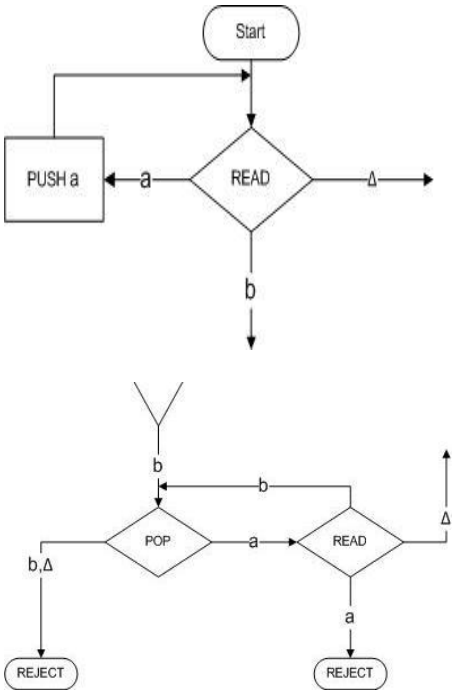
- The STACK is now down to only one a.
- The a line from POP takes us again to this same READ.
- There is only one letter left on the input TAPE, a b



- We read it and leave the TAPE empty, hat is, all blanks. However, the machine does not yet know that the TAPEis empty.
- It will discover this only when it next tries to read the TAPE and finds Δ.



- Let



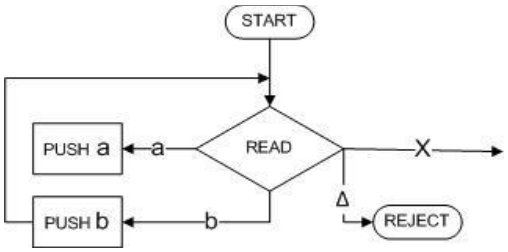
Example

- The PALINDROMEX, language of all words of the form s X reverse(s) where s is any string in (a + b)*.
- The words in this language are

  { X aXa bXb aaXaa abXba baXab bbXbb aaaXaaa aabXbaa . . . }

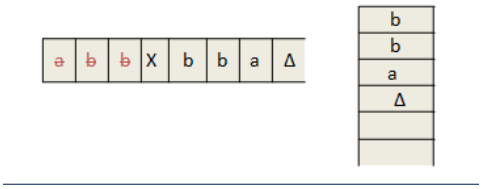
- They all contain exactly one X, and this X marks the middle ofthe word.
- We can build a deterministic PDA that accepts the language PALINDROMEX.
- It has the same basic structure as the PDA we had for the language $\{a^n b^n\}$.

- In the first part of the machine the STACK is loaded with the letters from the input string just as the initial a's from $a^n b^n$ were pushed onto the STACK.
- The letters go into the STACK first letter on the bottom, second letter on top of it, and so on till the last letter pushed in ends up on top.
- When we read the X we know we have reached the middle of the input.
- We can then begin to compare the front half of the word (which is reversed in the STACK) with the back half (still on the TAPE) to see that they match.
- We begin by storing the front half of the input string in the STACK with this part of the machine

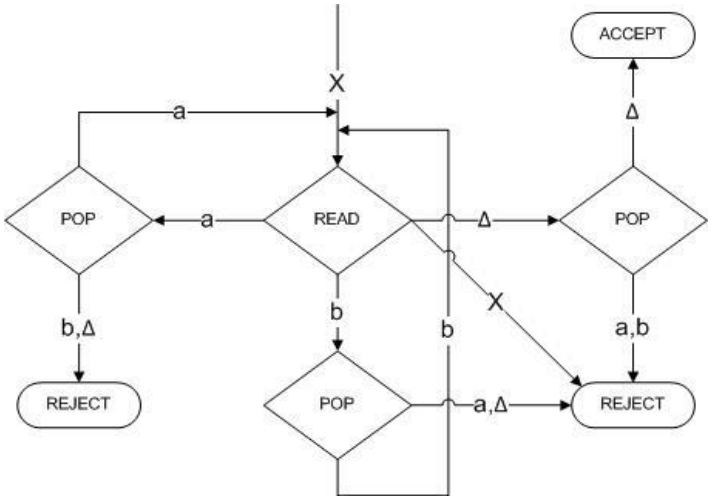


- If we READ an a, we PUSH an a. If we READ a b, we PUSH a b, and on and on until we encounter the X on the TAPE.
- After we take the first half of the word and stick it into the STACK, we have reversed the order of the letters and it looks exactly like the second half of the word.
- For example, if we begin with the input string
- abbXbbaAdding

- At the moment we are just about to read the X we have:

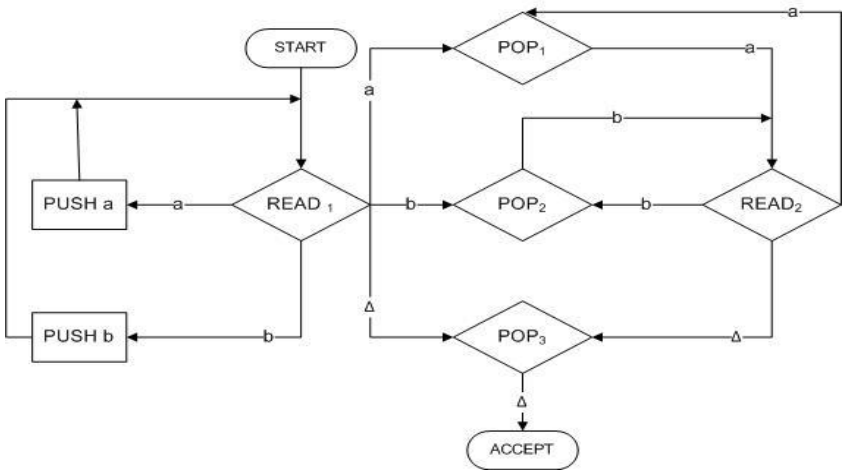


- When we read the X we do not put it into the STACK. It is used up the process of transferring us to phase two.
- In order to reach ACCEP these two should be the same letter for letter, down to the blanks.:

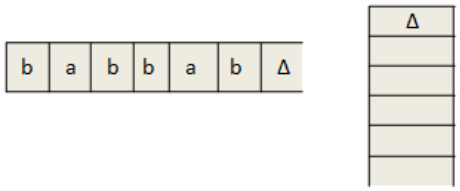


Example 2

- Consider the language:
- EVENPALINDROME = {s reverse(s), where s is in (a + b)*}
- = { A aa bb aaaa abba baab bbbbaaaaaa... }
- This is the language of all palindromes with an even number of letters.
- One machine to accept this language is shown:



- The first letter of the second half of the word is read in $READ_1$,
- Then we immediately go to the POP that compares the character read with what is on top of the STACK. After this we cycle $READ_2$-->POP-$READ_2$-->POP---> ....



- We can trace the path by which this input can be accepted by the successive rows in the table below:

STATE	STACK	TAPE
START	Δ . . .	babbabΔ ...
$READ_1$	Δ . . .	~~b~~abbabΔ ...
$PUSH_b$	bΔ . . .	~~b~~abbabΔ ...
$READ_1$	bΔ . . .	~~b~~abbabΔ ...
$PUSH_a$	abΔ . . .	~~b~~abbabΔ ...
$READ_1$	abΔ . .	~~ba~~bbabΔ ...
$PUSH_b$	babΔ . .	~~ba~~bbabΔ ...
$READ_1$	babΔ	~~bab~~babΔ ...

- If we are going to accept this input string this is where we must make the jump out of the left circuit into the right circuit. The trace continues:

STATE	STACK	TAPE
POP_2	abΔ . . .	~~babbab~~Δ …
$READ_2$	abΔ . . .	~~babbab~~Δ …
POP_1	bΔ . . .	~~babbab~~Δ …
$READ_2$	bΔ . . .	~~babbab~~Δ …
POP_2	Δ . . .	~~babbab~~Δ …
$READ_2$	Δ . . .	~~babbab~~Δ …

- We have just read the first of the infinitely many blanks on the TAPE.):

STATE	STACK	TAPE
POP_3	Δ . . . (Popping a blank from an empty stack still leaves blanks)	babbabΔ … (Reading a blank from from an empty an empty tape still stack still leaves leaves blanks)
ACCEPT	Δ . . .	~~babbab~~Δ …

Pushdown Automata

A pushdown automaton, PDA, is a collection of eight things

1. An alphabet I of input letters.
2. An input TAPE (infinite in one direction). Initially the letters is placed on the TAPE starting in cell i. The rest is blank.
3. An alphabet F of STACK characters.
4. A pushdown STACK (infinite in one direction). Initially empty (contains all blanks).
5. One START state that has only out-edges, no in-edges.

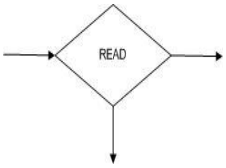
6.Halt states of two kinds: some ACCEPT and some REJECT.

They have in-edges and no out-edge

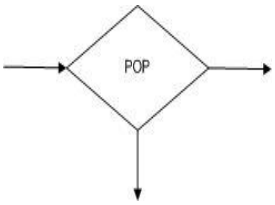
7. Finitely many nonbranching PUSH states that introduce characters onto the top of the STACK. They are of the form

8. Finitely many branching states of two kinds:

(i) States that read the next unused letter from the TAPE



(ii) States that read the top character of the STACK

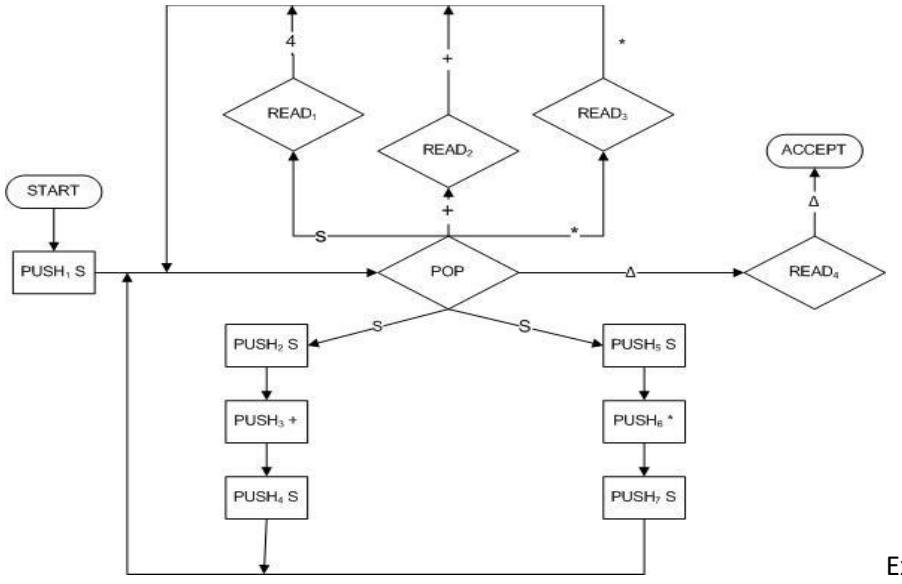


Example

- Consider the language generated by the CFG:

$$S \rightarrow S + S \mid S * S \mid 4$$

- The terminals are +, *, and 4 and the only nonterminal is S.
- The following PDA accepts this language:



Example

Example

Trace the acceptance of 4 + 4 * 4

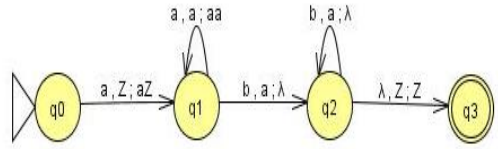
STATE	STACK	TAPE
START	Δ	4 +4 * 4
$PUSH_1$ S	S	4 +4 * 4
POP	Δ	4 +4 * 4
$PUSH_2$ S	S	4 +4 * 4
$PUSH_3$ +	+ S	4 +4 * 4
$PUSH_4$ S	S + S	4 +4 * 4
STATE	STACK	TAPE
POP	+S	4 +4 * 4
READ1	+S	4 * 4
POP	S	4 * 4
READ2	S	4 * 4
POP	Δ	4 * 4
PUSH5 S	S	4 * 4
PUSH6 *	*S	4 * 4
PUSH7 S	S*S	4 * 4
POP	*S	4 * 4

READ1	*S	* 4
POP	S	* 4
READ3	S	4
POP	Δ	4
READ1	Δ	Δ
POP	Δ	Δ
READ4	Δ	Δ
ACCEPT	Δ	Δ

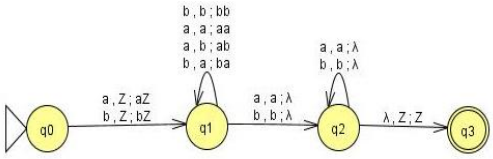
Lecture 27

PDA Examples in JFLAP

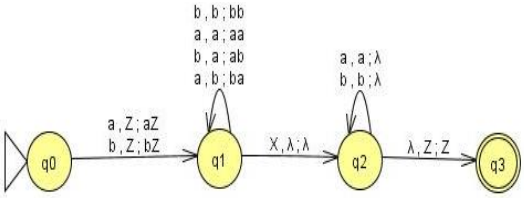
- How to simulate PDA in JFLAP
- Deterministic PDA Example
- Non Deterministic PDA Example
- PDA Example



- Even Palindrome

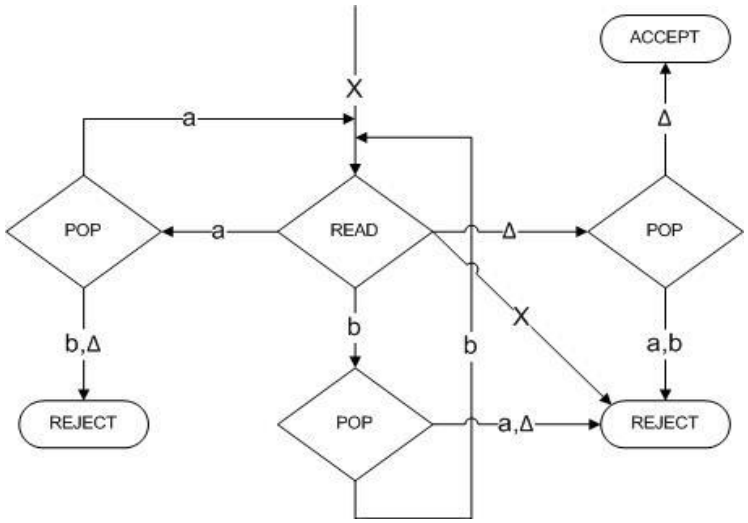


- Palindrome with X

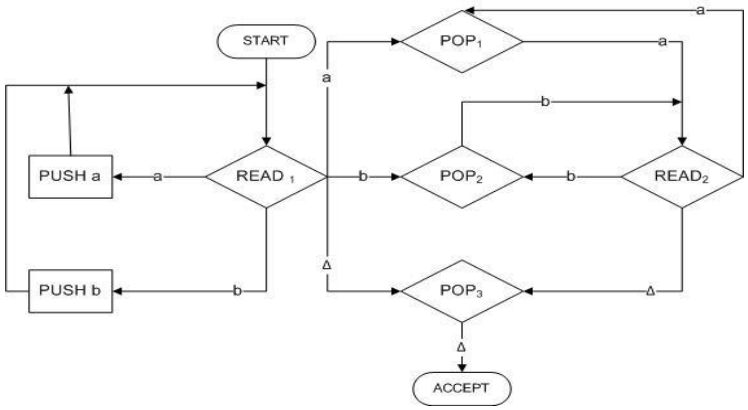


Lecture # 28

Conversion of PDA to JFLAP Format



- Conversion of PDA to JFLAP Format



Lecture # 29

PDA Conversion to CFG

- Theorem

  Given a language L generated by a particular CFG, there is a PDA that accepts exactly L.

- Let us consider the following CFG in CNF

  S → SB

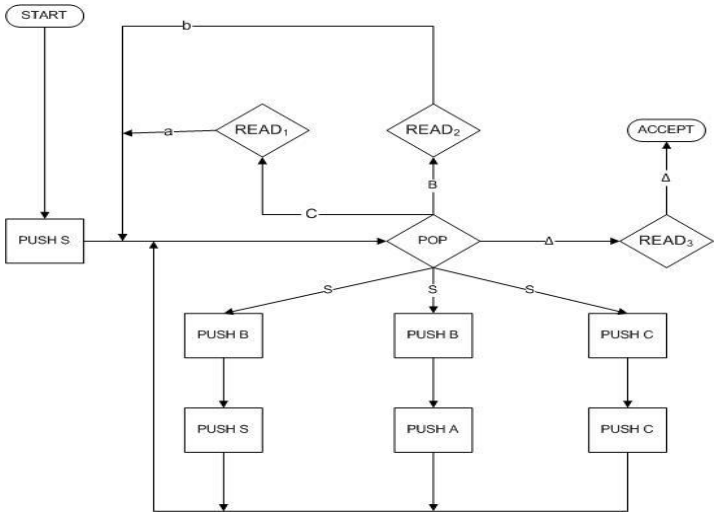
  S → AB

  A → CC

  B → b

  C → a.

- Theorem
- STACK alphabet : Γ = {S, A, B, C}
- TAPE alphabet: S = {a, b}



- The word aab can be generated by most derivation in this grammar as follows:

Working-String Generation	Production Used	
S => AB	S → AB	Step I
=>CCB	A → CC	Step 2
=>aCB	C → a	Step 3
=>aaB	C → a	Step 4
=> aab	B → b	Step 5

- We start with

STACK	TAPE
Δ	aab

- Immediately we push the symbol S onto the STACK.

STACK	TAPE
S	aab

- We pop the S and then we PUSH B, PUSH

STACK	TAPE
AB	aab

- We again feed back into the central POP. The production we must now simulate is a A → CC. This is done by popping the A and following the path PUSH C, PUSH C.

STACK	TAPE
CCB	aab

- Again we feed back into the central POP. This time we must simulate the reduction C → a. POP C and then read the a from the TAPE.

STACK	TAPE
CB	aab

- Theorem
- We again feed back into the central POP. The production we must now simulate is a A → CC. This is done by popping the A and following the path PUSH C, PUSH C.

STACK	TAPE
CCB	aab

- Again we feed back into the central POP. This time we must simulate the reduction C → a. POP C and then read the a from the TAPE.

STACK	TAPE
CB	aab

- The next production we must simulate is another C - a. Again we POP C and READ a.

STACK	TAPE
B	aab

- This time when we enter the central POP we simulate the last production in the derivation, B → b. We pop the B and read the b.

STACK	TAPE
Δ	aab

Theorem

- We now reenter the POP, and we must make sure that both STACK and TAPE are empty.

  POP Δ → $READ_3$ → ACCEPT

- To accept a word we must follow its left- most derivation from the CFG. If the word is ababbbaab
- and at some point in its left-most Chomsky derivation we have the working string ababbZWV.
- Theorem
- Then at this point in the corresponding PDA-processing the status of the STACK and TAPE should be

STACK	TAPE
ZWV	ababbbaab

- This process continues until we have derived the entire word. We then have

STACK	TAPE
Δ	ababbbaab

- Theorem
- Example

S → AB

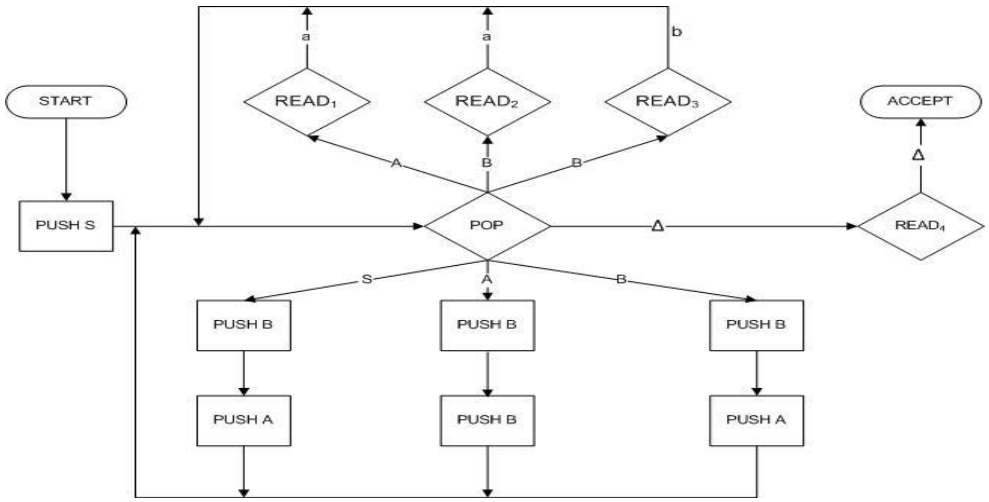
B → AB

B → a

A → BB

A → a

B → b



Left-most derivation	State	STACK	TAPE
	START	Δ	baaab
S	PUSH S	S	baaab
	POP	Δ	baaab
	PUSH B	B	baaab
=> AB	PUSHA	AB	baaab
	POP	B	baaab
	PUSH B	BB	baaab
⇒ BBB	PUSH B	BBB	baaab
	POP	BB	baaab
=>bBB	$READ_3$	BB	baaab

	POP	B	baaab
	PUSH B	BB	baaab
Babb	READ3	BB	baaab
	POP	B	baaab
	PUSH B	BB	baaab
BaBB	Push A	ABB	baaab
	POP	BB	baaab
BaBB	READ1	B	baaab
	POP	B	baaab
⇒ bAAB	READ2	B	baaab
	POP	Δ	baaab
	PUSH B	B	baaab
baaAB	PUSH A	AB	baaab
	Pop	B	baaab
baaAB	READ1	B	baaab
	POP	Δ	baaab
baaab	READ3	Δ	baaab

	POP	Δ	baaab
	READ4	Δ	baaab
	ACCEPT	Δ	baaab

- At every stage we have the following equivalence:

  Working string = (letters cancelled from TAPE) (string of nonterminals from STACK

- At the beginning this means:

  working string = S

  letters cancelled = none

  string of nonterminals in STACK = Δ

- At the end this means:

working string = the whole word

letters cancelled = all

STACK= Δ

- Now that we understand this example, we can give the rules for the general case.

Lecture # 30

What are Non-Context-Free languages

- All languages are not CFL.
- Languages which are not Context-Free, are called Non-CFL.
- This chapter is about proving that all languages are not Context-Free
- We will also use the Pumping Lemma with and without JFLAP for the proof of the aforementioned fact.

Live Production VS Dead Production

Definition

- A production of the form

  nonterminal → string of two nonterminals

  is called a live production.

- A production of the form nonterminal → terminal

  is called a dead production.

- Please keep in mind that every CFG in CNF has only aforementioned types of productions.
  - Guess what, are the rules of CNF?
- Live Productions Example

$S \Rightarrow b$	or	$S \Rightarrow XY$ $\Rightarrow aY$ $\Rightarrow aa$	or	$S \Rightarrow AB$ $\Rightarrow XYB$ $\Rightarrow bXB$ $\Rightarrow bSXB$ $\Rightarrow baXB$ $\Rightarrow baaB$ $\Rightarrow baab$
0 live 1 dead		1 live 2 dead		3 live 4 dead

- Finite many words can be generated if a CFG is in CNF and if there is restriction to use the live production at most once each, then only the.

- It may be noted that every time a live production is applied during the derivation of a word it increases the number of non-terminals by one, similarly, dead productions are applied, it decreases the number of non-terminals by one.

Theorem (33)

Statement:

If a CFG is in CNF with p live and q dead productions

and if w is a word generated by the CFG, having more than 2^p letters then any derivation tree for w has a nonterminal z which is used twice, where the second z is the descended from the first z.

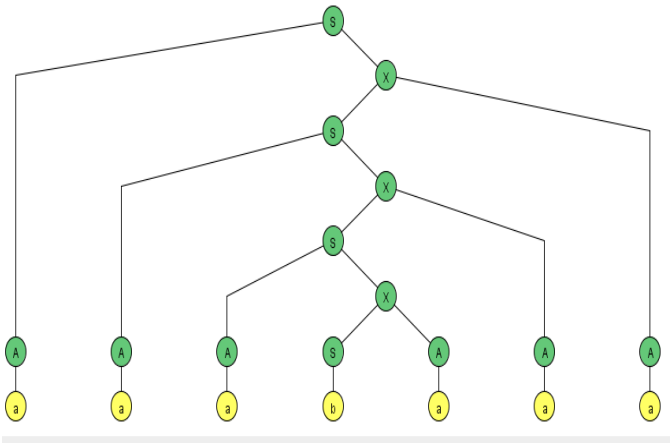
Self Embedded Nonterminal

A nonterminal is said to be self-embedded, if in

a given derivation of a word, it ever occurs as a

tree descendant of itself

The nonterminal X is self-embedded



Example Grammer

S	→	AX
X	→	SA
S	→	BY
Y	→	SB

S	→	a
S	→	b
S	→	AA
S	→	BB
A	→	a
B	→	b

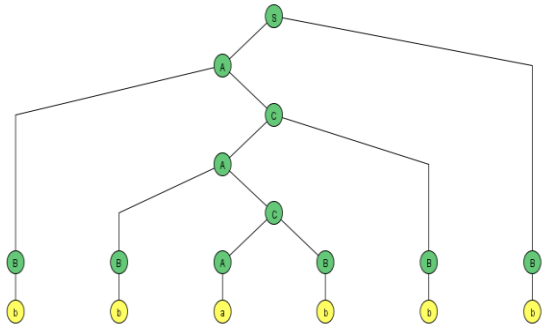
	S
S→AX	**AX**
X→SA	**ASA**
S→AX	**AAXA**
X→SA	**AASAA**
S→AX	**AAAXAA**
X→SA	**AAASAAA**
A→a	**aAASAAA**
A→a	**aaASAAA**
A→a	**aaaSAAA**
S→b	**aaabAAA**
A→a	**aaabaAA**
A→a	**aaabaaA**

A→a	aaabaaa

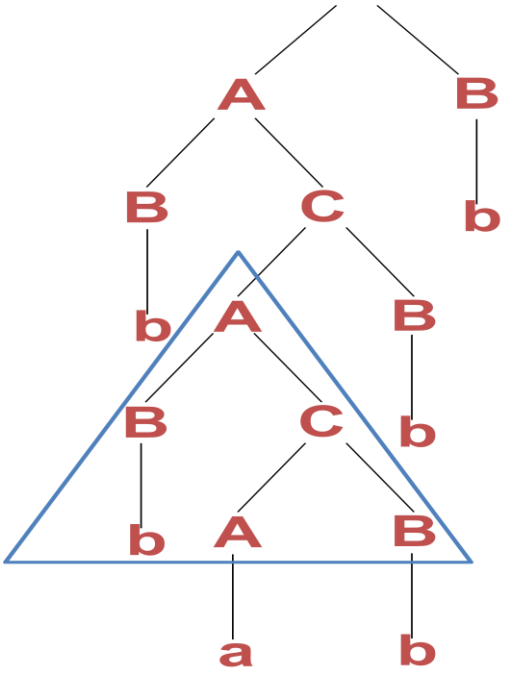
Consider the following CFG in CNF

and the derivation tree of the word bbabbb

S	→	AB
A	→	BC
C	→	AB
A	→	a
B	→	b



- Non-terminal A is self embedded
- Look at the triangle – upper one and lower one identified in the lower.
- Guess what – Pumping Lemma for CFG in CNF?



Pumping lemma for CFL in CNF

If G is any CFG in CNF with p live productions, then every word w of length more than 2^p c, then we can break substrings as w=uvxyz, where x is not null string and v and y are both not null string.

Thus, all the words of the form uv^nxy^nz, n=1,2,3,… can also be generated by G.

Consider the following CFG in CNF:

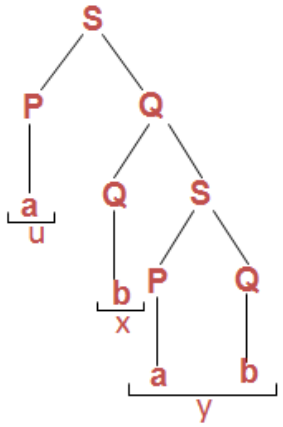
S → PQ

Q →QS|b

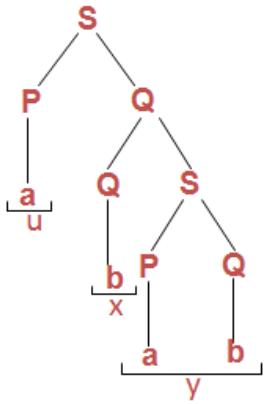
P →a

we derive the word **abab** as follows:

- w=uvxyz where u=a, v= ʌ, x=b, y=ab, z= ʌ
- Left most is u, within the triangle, the left most is x and remaining part is y, since the v part can also be null.



- uvvxyyz=a ʌ ʌ babab ʌ =ababab belongs to the language generated by the given CFG.
- uv^nxy^nz, n=1,2,3,… belong to the language generated by the given CFG.



- Consider the language

  $L=\{a^nb^nc^n : n=1,2,3,\ldots\}$, let the language L be Context Free language

- Let the word $w=a^{200}b^{200}c^{200}$ of length more than 2^p, where p is the number of live productions of its CFG in CNF.
- Please follow the observations given:
- **Observations:**
- **Substrings u,v,x,y and z. uv^2xy^2z can't belong to L, as all the words in $a^nb^nc^n$ have**
-

Only one substring ab

Only one substring bc

No substring ac

No substring ba

No substring ca

No substring cb

For any n=1,2,3,…

It may be noted that the pumping lemma is satisfied by all CFLs and the languages which don't hold this pumping lemma, can't be Context Free languages. Such languages are non-CFLs. Following are the examples of non-CFL.

"**If w is a large enough word (2^P) in a CF: then, w can be decomposed into w=uvxyz such that all words of the form uv^nxy^nz belong to L**"

- JFLAP For Non Context Free Languages
- Pumping Lemma in JFLAP
- How to list cases
- Playing game
  - You go first
  - Computer go first

Lecture # 31

Decidability

In this chapter, we will focus on three problems

1. Whether or not the given CFG generates any word? Problem of emptiness of CFL.
2. To determine whether the nonterminal X is ever used in the derivation of word from the given CFG? Problem of usefulness.
3. Whether or not the given CFG generates the finite language? Problem of finiteness.

Whether or not the given CFG generates any word? Problem of emptiness of CFL.

Consider the following Example:

S→AB,

A →BSB,

B→CC

C→SS

A→a|b

C →b|bb

A→a, C →b, it can be written as

S→aB

A→BSB

A→bb

B→aaS

B→bb

C→SS

B→bb and A→bb, it can be written as:

S→abb

A→bbSbb

B→aaS

C→SS

Since S→abb has been obtained so, abb is a word in the corresponding CFL.

- To determine whether the nonterminal X is ever used in the derivation of word from the given CFG? Problem of usefulness.

Consider the following CFG

S→Aba | bAZ | b

A→Xb | bZa

B→bAA

X→aZa|aaa

Z→ZAbA

To determine whether X is ever used to generate some words, unproductive nonterminals are determined. Z is unproductive nonterminal, so eliminating the productions involving Z.

S→Aba|b

A→Xb

B→bAA

X→aaa

X points to non terminal, so A also. Thus B and S also point to non terminal.

So only useful productions shall be used.

Whether or not the given CFG generates the finite language? Problem of finiteness.

Example:

Consider the CFG

S→ABa|bAZ|b

A→Xb|bZa

B→bAA

X→aZa|bA|aaa

Z→ZAbA

Here the nonterminal Z is useless, so by eliminating we get the following grammar:

S→ABa|b

A→Xb

B→bAA

X→bA|aaa

Starting with nonterminal X. It is self-embedded, therefore, the grammar creates an infinite language.

S→ABa|b

A→Xb

B→bAA

X →bA|aaa

Example:

Consider the following CFG in CNF

S→AA

A→AA

A→a

Let x=aaa. To determine whether x can be generated from the given CFG let

$x=x_1x_2x_3$ where $x_1=x_2=x_3=a$

According to CYK algorithm, the list of nonterminals producing single letter double letter substrings of x and the string x itself, can be determined as follows

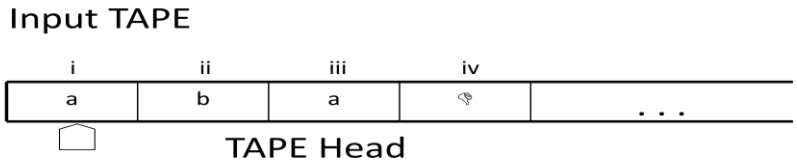
Since S is in the list of producing nonterminals, so aaa can be generated by the given CFG.

Lecture # 32

Turing machine Components

A Turing machine (TM) consists of the following

1. An alphabet Σ of input letters.
2. An input TAPE partitioned into cells, having infinite many locations in one direction. The input string is placed on the TAPE starting its first letter on the cell i, the rest of the TAPE is initially filled with blanks (’s).



3. A tape Head can read the contents of cell on the TAPE in one step. It can replace the character at any cell and can reposition itself to the next cell to the right or to the left of that it has just read.

   Initially the TAPE Head is at the cell i. The TAPE Head can’t move to the left of cell i. the location of the TAPE Head is denoted by

   .

4. An alphabet G of characters that can be printed on the TAPE by the TAPE Head. G may include the letters of Σ. Even the TAPE Head can print blank ( ), which means to erase some character from the TAPE.
5. Finite set of states -- exactly one START state and some (may be none) HALT states. Halt is similar to Accept.
6. A **Function** which is the set of rules, which show that which state is to be entered when a letter is read form the TAPE and what character is to be printed. This function has the format:

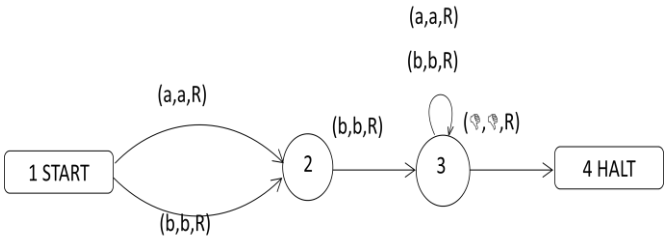
   (letter, letter, direction)

   It may be noted that the first letter is the character the TAPE Head reads from the cell to which it is pointing. The second letter is what the TAPE Head prints the cell before it leaves. The direction tells the TAPE Head whether to move one cell to the right, R, or one cell to the left, L. Following is a note

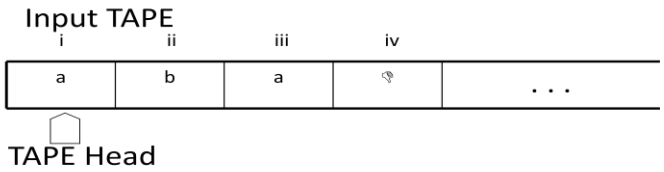
- Note

Example

Consider the following Turing machine

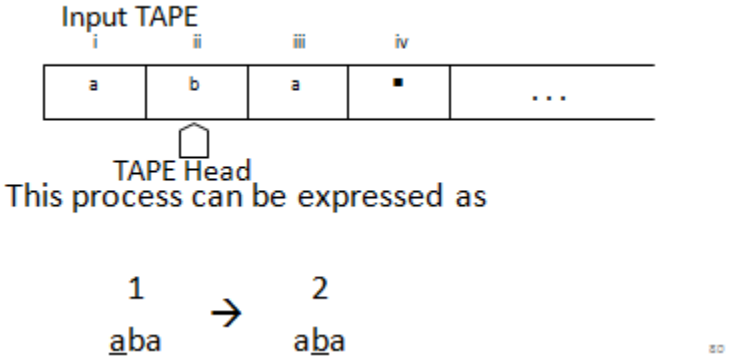


Let the input string aba be run over this TM



Starting from the START state, reading a form the TAPE and according to the TM program, a will be printed *i.e.* a will be replaced by a and the TAPE Head will be moved one cell to the right.

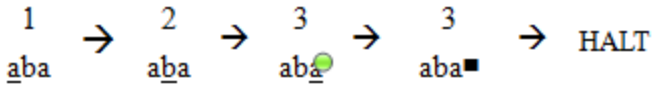
Which can be seen as



This process can be expressed as

At state 2 reading b, state 3 is entered and the letter b is replaced by b, *i.e.*

At state 3 reading a, will keep the state of the TM unchanged. Lastly, the blank ( ) is read and ( ) is replaced by D and the HALT state is entered. Which can be expressed as



Which shows that the string aba is accepted by this machine. It can be observed, from the program of the TM, that the machine accepts the language expressed by $(a+b)b(a+b)^*$.

Turning Machines using JFLAP.