

## Table of contents:

Lecture NO. 1 .....	3
What does automata mean? .....	3
Introduction to languages.....	3
Alphabets.....	3
Strings.....	3
Defining Languages.....	4
Lecture NO. 2.....	7
Kleene Star Closure.....	7
Recursive definition of languages.....	7
Lecture NO. 3.....	9
Regular Expression .....	9
Recursive definition of Regular Expression(RE) .....	9
Method 3 (Regular Expressions).....	9
Lecture NO. 4.....	10
Equivalent Regular Expressions .....	10
Method 4 (Finite Automaton) .....	11
Lecture NO. 5.....	13
Lecture NO. 6.....	15
Equivalent FAs.....	15
Lecture NO. 7.....	17
FA corresponding to finite languages .....	17
Method 5 (Transition Graph).....	18
Lecture NO. 8.....	19
Examples of TGs: accepting all strings, accepting none, starting with b, not ending in b, containing aa, containing aa or bb.....	19
Lecture NO. 9.....	21
Generalized Transition Graphs .....	23
Lecture NO. 10.....	24
Nondeterminism.....	25
Kleene's Theorem .....	25
Lecture NO. 11.....	26
Proof(Kleene's Theorem Part II) .....	26
Lecture NO. 12.....	30
Kleene's Theorem Part III.....	31
Lecture NO. 13.....	34
Lecture NO. 14.....	37
Lecture NO. 15.....	40
Nondeterministic Finite Automaton (NFA) .....	40
Converting an FA to an equivalent NFA.....	41
Lecture NO. 16.....	42
NFA with Null String .....	42
Lecture NO. 17.....	45
NFA and Kleene's Theorem .....	45
Lecture NO. 18.....	48
NFA corresponding to Concatenation of FAs.....	48
NFA corresponding to the Closure of an FA.....	50
Lecture NO. 19.....	52
Memory required to recognize a language.....	52
Distinguishable strings and Indistinguishable strings .....	53
Lecture NO. 20.....	55
Finite Automaton with output.....	55
Moore machine .....	55
Lecture NO. 21.....	57
Mealy machine.....	57
Lecture NO. 22.....	60
Equivalent machines.....	60
Lecture NO. 23.....	63
Lecture NO. 24.....	65

Regular languages.....	65
Complement of a language .....	66
Lecture NO. 25 .....	68
Nonregular languages.....	71
Lecture NO. 26 .....	72
Pumping Lemma .....	72
Lecture NO. 27 .....	75
Pumping Lemma version II .....	75
Lecture NO. 28 .....	77
Pseudo theorem .....	78
Lecture NO. 29 .....	79
Decidability.....	79
Determining whether the two languages are equivalent or not ? .....	80
Lecture NO. 30 .....	83
Lecture NO. 31 .....	87
Context Free Grammar (CFG) .....	87
CFG terminologies.....	87
Lecture NO. 32 .....	90
Trees.....	91
Lecture NO. 33 .....	93
Polish Notation (o-o-o) .....	94
Lecture NO. 34 .....	96
Total language tree.....	96
Regular Grammar .....	97
Lecture NO. 35 .....	99
Null Production.....	99
Lecture NO. 36 .....	102
Chomsky Normal Form (CNF) .....	102
Lecture NO. 37 .....	105
A new format for FAs.....	105
Lecture NO. 38 .....	109
Nondeterministic PDA .....	111
Lecture NO. 39 .....	114
PDA corresponding to CFG .....	114
Lecture NO. 40 .....	118
Conversion form of PDA.....	119
Lecture NO. 41 .....	121
Lecture NO. 42 .....	124
Lecture NO. 43 .....	127
Non-Context-Free language .....	127
Pumping lemma for CFLs .....	128
Lecture NO. 44 .....	133
Decidability.....	133
Parsing Techniques .....	136
Lecture NO. 45 .....	140
Turing machine.....	140

**Last Updated: January 24, 2013**

## Theory of Automata

### Lecture N0. 1

#### Reading Material

#### Introduction to Computer Theory

#### Chapter 2

#### **Summary**

Introduction to the course title, Formal and In-formal languages, Alphabets, Strings, Null string, Words, Valid and In-valid alphabets, length of a string, Reverse of a string, Defining languages, Descriptive definition of languages, EQUAL, EVEN-EVEN, INTEGER, EVEN,  $\{a^n b^n\}$ ,  $\{a^n b^n a^n\}$ , factorial, FACTORIAL, DOUBLEFACTORIAL, SQUARE, DOUBLESQUARE, PRIME, PALINDROME.

#### What does automata mean?

It is the plural of automaton, and it means “something that works automatically”

#### Introduction to languages

There are two types of languages

- Formal Languages (Syntactic languages)
- Informal Languages (Semantic languages)

#### Alphabets

##### Definition

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

##### Example

$$\begin{aligned}\Sigma &= \{a,b\} \\ \Sigma &= \{0,1\} \text{ (important as this is the language which the computer understands.)} \\ \Sigma &= \{i,j,k\}\end{aligned}$$

Note Certain version of language ALGOL has 113 letters.

$\Sigma$  (alphabet) includes letters, digits and a variety of operators including sequential operators such as GOTO and IF

#### Strings

##### Definition

Concatenation of finite number of letters from the alphabet is called a string.

##### Example

If  $\Sigma = \{a,b\}$  then  
a, abab, aaabb, abababababababab

#### Note

##### Empty string or null string

Sometimes a string with no symbol at all is used, denoted by (Small Greek letter Lambda)  $\lambda$  or (Capital Greek letter Lambda)  $\Lambda$ , is called an empty string or null string.

The capital lambda will mostly be used to denote the empty string, in further discussion.

#### Words

##### Definition

Words are strings belonging to some language.

##### Example

If  $\Sigma = \{x\}$  then a language L can be defined as

$L = \{x^n : n=1,2,3,\dots\}$  or  $L = \{x,xx,xxx,\dots\}$

Here  $x,xx,\dots$  are the words of L

##### Note

All words are strings, but not all strings are words.

#### Valid/In-valid alphabets

While defining an alphabet, an alphabet may contain letters consisting of group of symbols for example  $\Sigma_1 = \{B, aB, bab, d\}$ .

Now consider an alphabet

$\Sigma_2 = \{B, Ba, bab, d\}$  and a string BababB.

This string can be tokenized in two different ways

(Ba), (bab), (B)

(B), (abab), (B)

Which shows that the second group cannot be identified as a string, defined over

$\Sigma = \{a, b\}$ .

As when this string is scanned by the compiler (Lexical Analyzer), first symbol B is identified as a letter belonging to  $\Sigma$ , while for the second letter the lexical analyzer would not be able to identify, so while defining an alphabet it should be kept in mind that ambiguity should not be created.

#### Remarks

While defining an alphabet of letters consisting of more than one symbols, no letter should be started with the letter of the same alphabet *i.e.* one letter should not be the prefix of another. However, a letter may be ended in a letter of same alphabet.

#### Conclusion

$\Sigma_1 = \{B, aB, bab, d\}$

$\Sigma_2 = \{B, Ba, bab, d\}$

$\Sigma_1$  is a valid alphabet while  $\Sigma_2$  is an in-valid alphabet.

#### Length of Strings

##### Definition

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

##### Example

$\Sigma = \{a, b\}$

$s = ababa$

$|s| = 5$

##### Example

$\Sigma = \{B, aB, bab, d\}$

$s = BaBbabBd$

Tokenizing = (B), (aB), (bab), (B), (d)

$|s| = 5$

#### Reverse of a String

##### Definition

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

##### Example

If  $s = abc$  is a string defined over  $\Sigma = \{a, b, c\}$

then  $\text{Rev}(s)$  or  $s^r = cba$

##### Example

$\Sigma = \{B, aB, bab, d\}$

$s = BaBbabBd$

$\text{Rev}(s) = dBbabaBB$

#### Defining Languages

The languages can be defined in different ways, such as Descriptive definition, Recursive definition, using Regular Expressions(RE) and using Finite Automaton(FA) etc.

#### Descriptive definition of language

The language is defined, describing the conditions imposed on its words.

Example

The language  $L$  of strings of odd length, defined over  $\Sigma = \{a\}$ , can be written as  
 $L = \{a, aaa, aaaaa, \dots\}$

Example

The language  $L$  of strings that does not start with  $a$ , defined over  $\Sigma = \{a, b, c\}$ , can be written as  
 $L = \{\Lambda, b, c, ba, bb, bc, ca, cb, cc, \dots\}$

Example

The language  $L$  of strings of length 2, defined over  $\Sigma = \{0, 1, 2\}$ , can be written as  
 $L = \{00, 01, 02, 10, 11, 12, 20, 21, 22\}$

Example

The language  $L$  of strings ending in 0, defined over  $\Sigma = \{0, 1\}$ , can be written as  
 $L = \{0, 00, 10, 000, 010, 100, 110, \dots\}$

Example

The language **EQUAL**, of strings with number of  $a$ 's equal to number of  $b$ 's, defined over  $\Sigma = \{a, b\}$ , can be written as  
 $\{\Lambda, ab, ba, aabb, abab, baba, abba, \dots\}$

Example

The language **EVEN-EVEN**, of strings with even number of  $a$ 's and even number of  $b$ 's, defined over  $\Sigma = \{a, b\}$ , can be written as  
 $\{\Lambda, aa, bb, aaaa, aabb, abab, abba, baab, baba, bbaa, bbbb, \dots\}$

Example

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

Example

The language **EVEN**, of strings defined over  $\Sigma = \{-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ , can be written as  
 $\text{EVEN} = \{\dots, -4, -2, 0, 2, 4, \dots\}$

Example

The language  $\{a^n b^n\}$ , of strings defined over  $\Sigma = \{a, b\}$ , as  
 $\{a^n b^n : n=1, 2, 3, \dots\}$ , can be written as  
 $\{ab, aabb, aaabbb, aaaabbbb, \dots\}$

Example

The language  $\{a^n b^n a^n\}$ , of strings defined over  $\Sigma = \{a, b\}$ , as  
 $\{a^n b^n a^n : n=1, 2, 3, \dots\}$ , can be written as  
 $\{aba, aabbaa, aaabbbaaa, aaaabbbbbaaa, \dots\}$

Example

The language **factorial**, of strings defined over  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$  *i.e.*  
 $\{1, 2, 6, 24, 120, \dots\}$

Example

The language **FACTORIAL**, of strings defined over  $\Sigma = \{a\}$ , as  
 $\{a^n! : n=1, 2, 3, \dots\}$ , can be written as  
 $\{a, aa, aaaaa, \dots\}$ . It is to be noted that the language **FACTORIAL** can be defined over any single letter alphabet.

Example

The language **DOUBLEFACTORIAL**, of strings defined over  $\Sigma = \{a, b\}$ , as  
 $\{a^n! b^n! : n=1, 2, 3, \dots\}$ , can be written as  
 $\{ab, aabb, aaaaaabbbbb, \dots\}$

Example

The language **SQUARE**, of strings defined over  $\Sigma = \{a\}$ , as  $\{a^{n^2} : n=1,2,3,\dots\}$ , can be written as  $\{a, aaaa, aaaaaaaaa, \dots\}$

#### Example

The language **DOUBLESQUARE**, of strings defined over  $\Sigma = \{a,b\}$ , as  $\{a^{n^2} b^{n^2} : n=1,2,3,\dots\}$ , can be written as  $\{ab, aaaabbbb, aaaaaaaabbbbbbbb, \dots\}$

#### Example

The language **PRIME**, of strings defined over  $\Sigma = \{a\}$ , as  $\{a^p : p \text{ is prime}\}$ , can be written as  $\{aa, aaa, aaaaa, aaaaaaa, aaaaaaaaa, \dots\}$

### An Important language

#### PALINDROME

The language consisting of  $\Lambda$  and the strings  $s$  defined over  $\Sigma$  such that  $\text{Rev}(s)=s$ . It is to be denoted that the words of PALINDROME are called palindromes.

#### Example

For  $\Sigma = \{a,b\}$ ,  
PALINDROME =  $\{\Lambda, a, b, aa, bb, aaa, aba, bab, bbb, \dots\}$

#### Remark

There are as many palindromes of length  $2n$  as there are of length  $2n-1$ .  
To prove the above remark, the following is to be noted:

#### Note

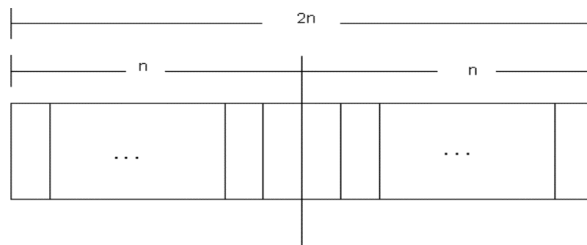
Number of strings of length 'm' defined over alphabet of 'n' letters is  $n^m$ .

#### Examples

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

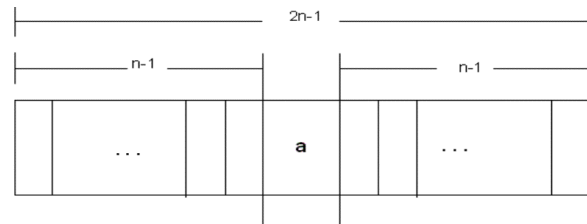
The language of strings of length 3, defined over  $\Sigma = \{a,b\}$  is  $L = \{aaa, aab, aba, baa, abb, bab, bba, bbb\}$  i.e. number of strings =  $2^3$

To calculate the number of palindromes of length  $(2n)$ , consider the following diagram,



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  $2^n$ .

To calculate the number of palindromes of length  $(2n-1)$  with 'a' as the middle letter, consider the following diagram,



which shows that there are as many palindromes of length  $2n-1$  as there are the strings of length  $n-1$  i.e. the required number of palindromes are  $2^{n-1}$ .

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  $2^{n-1} + 2^{n-1} = 2(2^{n-1}) = 2^n$ .

## Theory of Automata

### Lecture N0. 2

#### Reading Material

Introduction to Computer Theory

Chapter 3

#### Summary

Kleene Star Closure, Plus operation, recursive definition of languages, INTEGER, EVEN, factorial, PALINDROME,  $\{a^n b^n\}$ , languages of strings (i) ending in a, (ii) beginning and ending in same letters, (iii) containing aa or bb (iv) containing exactly one a

#### Kleene Star Closure

Given  $\Sigma$ , then the Kleene Star Closure of the alphabet  $\Sigma$ , denoted by  $\Sigma^*$ , is the collection of all strings defined over  $\Sigma$ , including  $\Lambda$ .

It is to be noted that Kleene Star Closure can be defined over any set of strings.

#### Examples

If  $\Sigma = \{x\}$

Then  $\Sigma^* = \{\Lambda, x, xx, xxx, xxxx, \dots\}$

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

Then  $\Sigma^* = \{\Lambda, 0, 1, 00, 01, 10, 11, \dots\}$

If  $\Sigma = \{aaB, c\}$

Then  $\Sigma^* = \{\Lambda, aaB, c, aaBaaB, aaBc, caaB, cc, \dots\}$

#### Note

Languages generated by Kleene Star Closure of set of strings, are infinite languages. (By infinite language, it is supposed that the language contains infinite many words, each of finite length).

#### PLUS Operation ( $^+$ )

Plus Operation is same as Kleene Star Closure except that it does not generate  $\Lambda$  (null string), automatically.

#### Example

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

Then  $\Sigma^+ = \{0, 1, 00, 01, 10, 11, \dots\}$

If  $\Sigma = \{aab, c\}$

Then  $\Sigma^+ = \{aab, c, aabaab, aabc, caab, cc, \dots\}$

#### Remark

It is to be noted that Kleene Star can also be operated on any string *i.e.*  $a^*$  can be considered to be all possible strings defined over  $\{a\}$ , which shows that  $a^*$  generates  $\Lambda, a, aa, aaa, \dots$

It may also be noted that  $a^+$  can be considered to be all possible non empty strings defined over  $\{a\}$ , which shows that  $a^+$  generates  $a, aa, aaa, aaaa, \dots$

#### Recursive definition of languages

The following three steps are used in recursive definition

Some basic words are specified in the language.

Rules for constructing more words are defined in the language.

No strings except those constructed in above, are allowed to be in the language.

#### Examples

##### Defining language of INTEGER

Step 1: 1 is in **INTEGER**.

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

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

##### Defining language of EVEN

Step 1: 2 is in **EVEN**.

- Step 2: If  $x$  is in **EVEN** then  $x+2$  and  $x-2$  are also in **EVEN**.  
Step 3: No strings except those constructed in above, are allowed to be in **EVEN**.

**Defining the language factorial**

- Step 1: As  $0!=1$ , so 1 is in **factorial**.  
Step 2:  $n!=n*(n-1)!$  is in **factorial**.  
Step 3: No strings except those constructed in above, are allowed to be in **factorial**.

**Defining the language PALINDROME, defined over  $\Sigma = \{a,b\}$**

- Step 1:  $a$  and  $b$  are in **PALINDROME**  
Step 2: if  $x$  is palindrome, then  $s(x)\text{Rev}(s)$  and  $xx$  will also be palindrome, where  $s$  belongs to  $\Sigma^*$   
Step 3: No strings except those constructed in above, are allowed to be in palindrome

**Defining the language  $\{a^n b^n\}$ ,  $n=1,2,3,\dots$ , of strings defined over  $\Sigma=\{a,b\}$**

- Step 1:  $ab$  is in  $\{a^n b^n\}$   
Step 2: if  $x$  is in  $\{a^n b^n\}$ , then  $axb$  is in  $\{a^n b^n\}$   
Step 3: No strings except those constructed in above, are allowed to be in  $\{a^n b^n\}$

**Defining the language  $L$ , of strings ending in  $a$ , defined over  $\Sigma=\{a,b\}$**

- Step 1:  $a$  is in  $L$   
Step 2: if  $x$  is in  $L$  then  $s(x)$  is also in  $L$ , where  $s$  belongs to  $\Sigma^*$   
Step 3: No strings except those constructed in above, are allowed to be in  $L$

**Defining the language  $L$ , of strings beginning and ending in same letters, defined over  $\Sigma=\{a, b\}$**

- Step 1:  $a$  and  $b$  are in  $L$   
Step 2:  $(a)s(a)$  and  $(b)s(b)$  are also in  $L$ , where  $s$  belongs to  $\Sigma^*$   
Step 3: No strings except those constructed in above, are allowed to be in  $L$

**Defining the language  $L$ , of strings containing  $aa$  or  $bb$ , defined over  $\Sigma=\{a, b\}$**

- Step 1:  $aa$  and  $bb$  are in  $L$   
Step 2:  $s(aa)s$  and  $s(bb)s$  are also in  $L$ , where  $s$  belongs to  $\Sigma^*$   
Step 3: No strings except those constructed in above, are allowed to be in  $L$

**Defining the language  $L$ , of strings containing exactly one  $a$ , defined over  $\Sigma=\{a, b\}$**

- Step 1:  $a$  is in  $L$   
Step 2:  $s(a)s$  is also in  $L$ , where  $s$  belongs to  $b^*$   
Step 3: No strings except those constructed in above, are allowed to be in  $L$



## Theory of Automata

**Lecture N0. 3****Reading Material**Introduction to Computer Theory

## Chapter 4

**Summary**

RE, Recursive definition of RE, defining languages by RE,  $\{x\}^*$ ,  $\{x\}^+$ ,  $\{a+b\}^*$ , Language of strings having exactly one a, Language of strings of even length, Language of strings of odd length, RE defines unique language (as Remark), Language of strings having at least one a, Language of strings having at least one a and one b, Language of strings starting with aa and ending in bb, Language of strings starting with and ending in different letters.

**Regular Expression**

As discussed earlier that  $a^*$  generates  $\Lambda, a, aa, aaa, \dots$  and  $a^+$  generates  $a, aa, aaa, aaaa, \dots$ , so the language  $L_1 = \{\Lambda, a, aa, aaa, \dots\}$  and  $L_2 = \{a, aa, aaa, aaaa, \dots\}$  can simply be expressed by  $a^*$  and  $a^+$ , respectively.  $a^*$  and  $a^+$  are called the regular expressions (RE) for  $L_1$  and  $L_2$  respectively.

Note  $a^+$ ,  $aa^*$  and  $a^*a$  generate  $L_2$ .

**Recursive definition of Regular Expression(RE)**

Step 1: Every letter of  $\Sigma$  including  $\Lambda$  is a regular expression.

Step 2: If  $r_1$  and  $r_2$  are regular expressions then

$(r_1)$

$r_1 r_2$

$r_1 + r_2$  and

$r_1^*$

are also regular expressions.

Step 3: Nothing else is a regular expression.

**Method 3 (Regular Expressions)**

Consider the language  $L = \{\Lambda, x, xx, xxx, \dots\}$  of strings, defined over  $\Sigma = \{x\}$ .

We can write this language as the Kleene star closure of alphabet  $\Sigma$  or  $L = \Sigma^* = \{x\}^*$ .

This language can also be expressed by the regular expression  $x^*$ .

Similarly the language  $L = \{x, xx, xxx, \dots\}$ , defined over  $\Sigma = \{x\}$ , can be expressed by the regular expression  $x^+$ .

Now consider another language  $L$ , consisting of all possible strings, defined over  $\Sigma = \{a, b\}$ . This language can also be expressed by the regular expression  $(a+b)^*$ .

Now consider another language  $L$ , of strings having exactly one a, defined over  $\Sigma = \{a, b\}$ , then its regular expression may be  $b^*ab^*$ .

Now consider another language  $L$ , of even length, defined over  $\Sigma = \{a, b\}$ , then its regular expression may be  $((a+b)(a+b))^*$ .

Now consider another language  $L$ , of odd length, defined over  $\Sigma = \{a, b\}$ , then its regular expression may be  $(a+b)((a+b)(a+b))^*$  or  $((a+b)(a+b))^*(a+b)$ .

**Remark**

It may be noted that a language may be expressed by more than one regular expression, while given a regular expression there exist a unique language generated by that regular expression.

**Example**

Consider the language, defined over

$\Sigma = \{a, b\}$  of words having at least one a, may be expressed by a regular expression  $(a+b)^*a(a+b)^*$ .

Consider the language, defined over  $\Sigma = \{a, b\}$  of words having at least one a and one b, may be expressed by a regular expression  $(a+b)^*a(a+b)^*b(a+b)^* + (a+b)^*b(a+b)^*a(a+b)^*$ .

Consider the language, defined over  $\Sigma = \{a, b\}$ , 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  $\Sigma = \{a, b\}$  of words starting with a and ending in b OR starting with b and ending in a, then its regular expression may be  $a(a+b)^*b + b(a+b)^*a$

## Theory of Automata

**Lecture N0. 4****Reading Material**

Introduction to Computer Theory

Chapter 4, 5

**Summary**

Regular expression of EVEN-EVEN language, Difference between  $a^* + b^*$  and  $(a+b)^*$ , Equivalent regular expressions; sum, product and closure of regular expressions; regular languages, finite languages are regular, introduction to finite automaton, definition of FA, transition table, transition diagram

**An important example****The Language EVEN-EVEN**

Language of strings, defined over  $\Sigma = \{a, b\}$  having even number of a's and even number of b's. i.e.

EVEN-EVEN =  $\{\Lambda, aa, bb, aaaa, aabb, abab, abba, baab, baba, bbaa, bbbb, \dots\}$ , its regular expression can be written as  $(aa+bb+(ab+ba)(aa+bb)^*(ab+ba))^*$

**Note**

It is important to be clear about the difference of the following regular expressions

$$r_1 = a^* + b^*$$

$$r_2 = (a+b)^*$$

Here  $r_1$  does not generate any string of concatenation of a and b, while  $r_2$  generates such strings.

**Equivalent Regular Expressions****Definition**

Two regular expressions are said to be equivalent if they generate the same language.

**Example**

Consider the following regular expressions

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

$$r_2 = (a + b)^* aa + (a + b)^* bb \quad \text{then both regular expressions define the language of strings ending in aa or bb.}$$

**Note**

If  $r_1 = (aa + bb)$  and  $r_2 = (a + b)$  then

$$r_1 + r_2 = (aa + bb) + (a + b)$$

$$r_1 r_2 = (aa + bb) (a + b)$$

$$= (aaa + aab + bba + bbb)$$

$$(r_1)^* = (aa + bb)^*$$

**Regular Languages****Definition**

The language generated by any regular expression is called a **regular language**.

It is to be noted that if  $r_1, r_2$  are regular expressions, corresponding to the languages  $L_1$  and  $L_2$  then the languages generated by  $r_1 + r_2$ ,  $r_1 r_2$  (or  $r_2 r_1$ ) and  $r_1^*$  (or  $r_2^*$ ) are also regular languages.

**Note**

It is to be noted that if  $L_1$  and  $L_2$  are expressed by  $r_1$  and  $r_2$ , respectively then the language expressed by

$r_1 + r_2$ , is the language  $L_1 + L_2$  or  $L_1 \cup L_2$

$r_1 r_2$ , is the language  $L_1 L_2$ , of strings obtained by prefixing every string of  $L_1$  with every string of  $L_2$

$r_1^*$ , is the language  $L_1^*$ , of strings obtained by concatenating the strings of  $L_1$ , including the null string.

**Example**

If  $r_1 = (aa+bb)$  and  $r_2 = (a+b)$  then the language of strings generated by  $r_1+r_2$ , is also a regular language, expressed by  $(aa+bb) + (a+b)$

If  $r_1 = (aa+bb)$  and  $r_2 = (a+b)$  then the language of strings generated by  $r_1 r_2$ , is also a regular language, expressed by  $(aa+bb)(a+b)$

If  $r = (aa+bb)$  then the language of strings generated by  $r^*$ , is also a regular language, expressed by  $(aa+bb)^*$

**All finite languages are regular****Example**

Consider the language  $L$ , defined over  $\Sigma = \{a,b\}$ , of strings of length 2, starting with  $a$ , then  $L = \{aa, ab\}$ , may be expressed by the regular expression  $aa+ab$ . Hence  $L$ , by definition, is a regular language.

**Note**

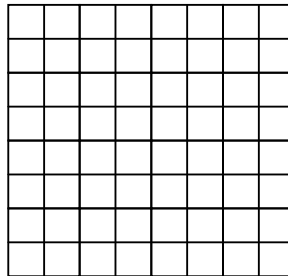
It may be noted that if a language contains even thousand words, its RE may be expressed, placing ' + ' between all the words.

Here the special structure of RE is not important.

Consider the language  $L = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$ , that may be expressed by a RE  $aaa+aab+aba+abb+baa+bab+bba+bbb$ , which is equivalent to  $(a+b)(a+b)(a+b)$ .

**Introduction to Finite Automaton**

Consider the following game board that contains 64 boxes



There are some pieces of paper. Some are of white colour while others are of black colour. The number of pieces of paper are 64 or less. The possible arrangements under which these pieces of paper can be placed in the boxes, are finite. To start the game, one of the arrangements is supposed to be initial arrangement. There is a pair of dice that can generate the numbers 2,3,4,...12. For each number generated, a unique arrangement is associated among the possible arrangements.

It shows that the total number of transition rules of arrangement are finite. One and more arrangements can be supposed to be the winning arrangement. It can be observed that the winning of the game depends on the sequence in which the numbers are generated. This structure of game can be considered to be a finite automaton.

**Method 4 (Finite Automaton)****Definition**

A Finite automaton (FA), is a collection of the followings

Finite number of states, having one initial and some (maybe none) final states.

Finite set of input letters ( $\Sigma$ ) from which input strings are formed.

Finite set of transitions i.e. for each state and for each input letter there is a transition showing how to move from one state to another.

**Example**

$\Sigma = \{a,b\}$

States:  $x, y, z$  where  $x$  is an initial state and  $z$  is final state.

**Transitions:**

At state  $x$  reading  $a$ , go to state  $z$

At state  $x$  reading  $b$ , go to state  $y$

At state  $y$  reading  $a, b$  go to state  $y$

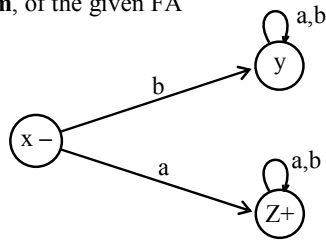
At state  $z$  reading  $a, b$  go to state  $z$

These transitions can be expressed by the following table called transition table

Old States	New States	
	Reading a	Reading b
x -	z	y
y	y	y
z +	z	z

Note

It may be noted that the information of an FA, given in the previous table, can also be depicted by the following diagram, called the **transition diagram**, of the given FA

Remark

The above transition diagram is an FA accepting the language of strings, defined over  $\Sigma = \{a, b\}$ , **starting with a**. It may be noted that this language may be expressed by the regular expression  $a(a + b)^*$

## Theory of Automata

**Lecture N0. 5****Reading Material**Introduction to Computer Theory

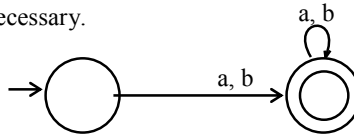
## Chapter 5

**Summary**

Different notations of transition diagrams, languages of strings of **even length**, **Odd length**, **starting with b**, **ending in a**, **beginning with b**, **not beginning with b**, **beginning and ending in same letters**

**Note**

It may be noted that to indicate the initial state, an arrow head can also be placed before that state and that the final state with double circle, as shown below. It is also to be noted that while expressing an FA by its transition diagram, the labels of states are not necessary.

**Example**

$\Sigma = \{a, b\}$

**States:** x, y, where x is both initial and final state.

**Transitions:**

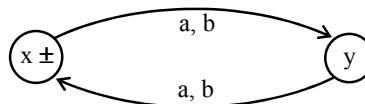
At state x reading a or b go to state y.

At state y reading a or b go to state x.

These transitions can be expressed by the following transition table

Old States	New States	
	Reading a	Reading b
x $\pm$	y	y
y	x	x

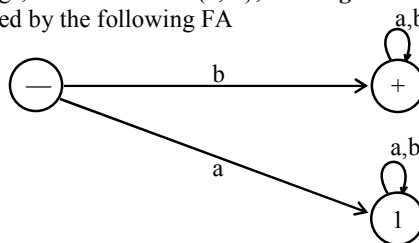
It may be noted that the above transition table may be depicted by the following transition diagram.



The above transition diagram is an FA accepting the language of strings, defined over  $\Sigma = \{a, b\}$  of **even length**. It may be noted that this language may be expressed by the regular expression  $((a + b)(a + b))^*$ .

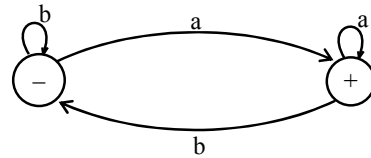
**Example:**

Consider the language L of strings, defined over  $\Sigma = \{a, b\}$ , **starting with b**. The language L may be expressed by RE  $b(a + b)^*$ , may be accepted by the following FA

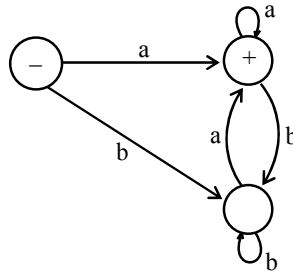
**Example**

Consider the language L of strings, defined over  $\Sigma = \{a, b\}$ , **ending in a**. The language L may be expressed by RE  $(a + b)^* a$ .

This language may be accepted by the FA shown aside



There may be another FA corresponding to the given language, as shown aside



#### Note

It may be noted that corresponding to a given language there may be more than one FA accepting that language, but for a given FA there is a unique language accepted by that FA.

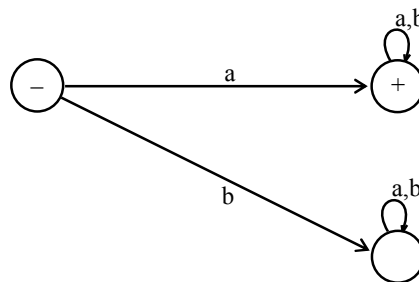
It is also to be noted that given the languages  $L_1$  and  $L_2$ , where

$L_1$  = The language of strings, defined over  $\Sigma = \{a, b\}$ , **beginning with a.**

$L_2$  = The language of strings, defined over  $\Sigma = \{a, b\}$ , **not beginning with b**

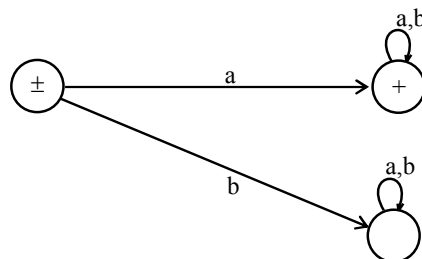
The  $\Lambda$  does not belong to  $L_1$  while it does belong to  $L_2$ . This fact may be depicted by the corresponding transition diagrams of  $L_1$  and  $L_2$ .

#### FA<sub>1</sub> Corresponding to $L_1$



The language  $L_1$  may be expressed by the regular expression  $a(a+b)^*$

#### FA<sub>2</sub> Corresponding to $L_2$



The language  $L_2$  may be expressed by the regular expression  $a(a+b)^* + \Lambda$

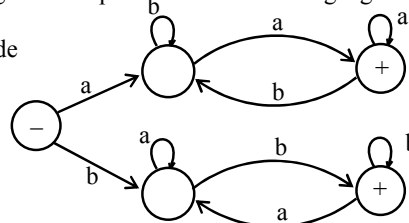
#### Example

Consider the Language  $L$  of Strings of **length two or more**, defined over  $\Sigma = \{a, b\}$ , **beginning with and ending in same letters.**

The language  $L$  may be expressed by the following regular expression  $a(a+b)^*a + b(a+b)^*b$

It is to be noted that if the condition on the length of string is not imposed in the above language then **the strings a and b will then belong to the language.**

This language  $L$  may be accepted by the FA as shown aside



## Theory of Automata

**Lecture N0. 6****Reading Material**Introduction to Computer Theory

## Chapter 5

**Summary**

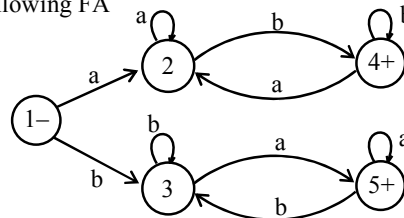
**Language of strings beginning with and ending in different letters, Accepting all strings, accepting non-empty strings, accepting no string, containing double a's, having double 0's or double 1's, containing triple a's or triple b's, EVEN-EVEN**

Example

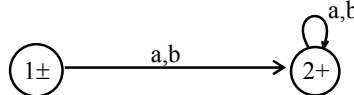
Consider the Language L of Strings, defined over  $\Sigma = \{a, b\}$ , **beginning with and ending in different letters**.

The language L may be expressed by the following regular expression  $a(a+b)^*b + b(a+b)^*a$

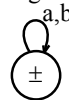
This language may be accepted by the following FA

Example

Consider the Language L, defined over  $\Sigma = \{a, b\}$  of **all strings including  $\Lambda$** . The language L may be accepted by the following FA



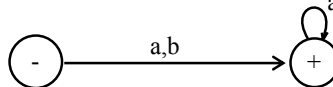
The language L may also be accepted by the following FA



The language L may be expressed by the regular expression  $(a+b)^*$

Example

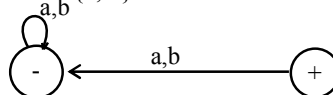
Consider the Language L, defined over  $\Sigma = \{a, b\}$  of **all non empty strings**. The language L may be accepted by the following FA



The above language may be expressed by the regular expression  $(a+b)^+$

Example

Consider the following FA, defined over  $\Sigma = \{a, b\}$

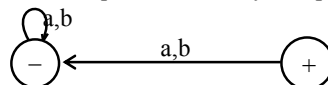


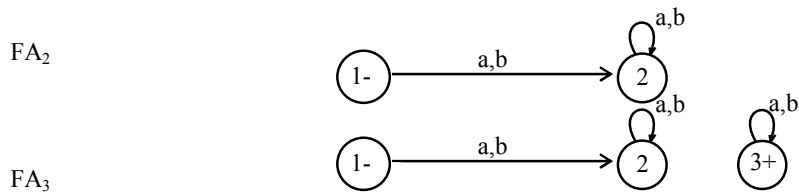
It is to be noted that the above FA **does not accept any string**, even it does not accept the null string; as there is no path starting from initial state and ending in final state.

**Equivalent FAs**

It is to be noted that two FAs are said to be equivalent, if they accept the same language, as shown in the following FAs.

FA<sub>1</sub>



Note

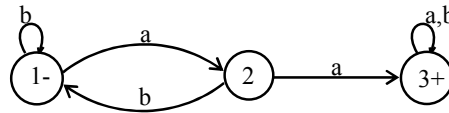
FA<sub>1</sub> has already been discussed, while in FA<sub>2</sub>, there is no final state and in FA<sub>3</sub>, there is a final state but FA<sub>3</sub> is disconnected as the states 2 and 3 are disconnected.

It may also be noted that the language of strings accepted by FA<sub>1</sub>, FA<sub>2</sub> and FA<sub>3</sub> is denoted by the empty set *i.e.*  $\{\}$  OR  $\emptyset$

Example

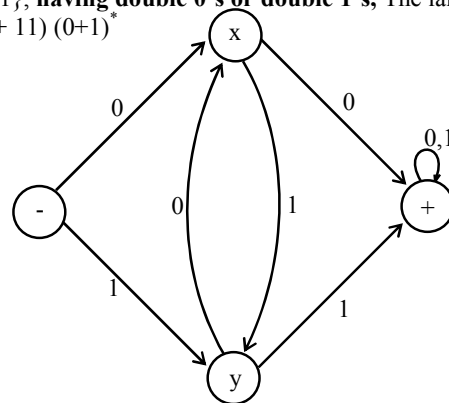
Consider the Language L of strings, defined over  $\Sigma = \{a, b\}$ , **containing double a**.

The language L may be expressed by the regular expression  $(a+b)^*(aa)(a+b)^*$ . This language may be accepted by the following FA.

Example

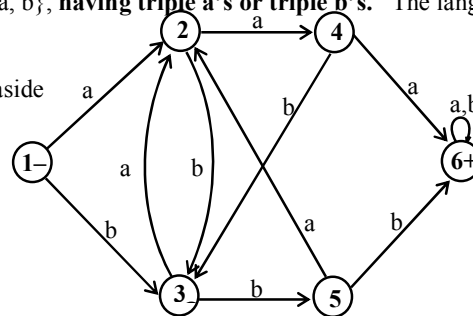
Consider the language L of strings, defined over  $\Sigma = \{0, 1\}$ , **having double 0's or double 1's**, The language L may be expressed by the regular expression  $(0+1)^*(00+11)(0+1)^*$

This language may be accepted by the following FA

Example

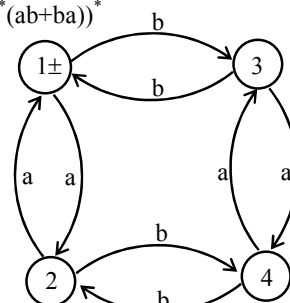
Consider the language L of strings, defined over  $\Sigma = \{a, b\}$ , **having triple a's or triple b's**. The language L may be expressed by RE  $(a+b)^*(aaa+bbb)(a+b)^*$

This language may be accepted by the FA as shown aside

Example

Consider the **EVEN-EVEN** language, defined over  $\Sigma = \{a, b\}$ . As discussed earlier that **EVEN-EVEN** language can be expressed by the regular expression  $(aa+bb+(ab+ba)(aa+bb)^*(ab+ba))^*$

**EVEN-EVEN** language may be accepted by the FA as shown aside





## Theory of Automata

**Lecture N0. 7****Reading Material**Introduction to Computer Theory

## Chapter 5, 6

**Summary**

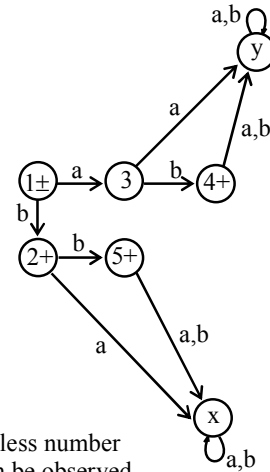
FA corresponding to finite languages(using both methods), Transition graphs.

**FA corresponding to finite languages**Example

Consider the language

$L = \{\Lambda, b, ab, bb\}$ , defined over  $\Sigma = \{a, b\}$ , expressed by  $\Lambda + b + ab + bb$  OR  $\Lambda + b(\Lambda + a + b)$ .

The language L may be accepted by the FA as shown aside

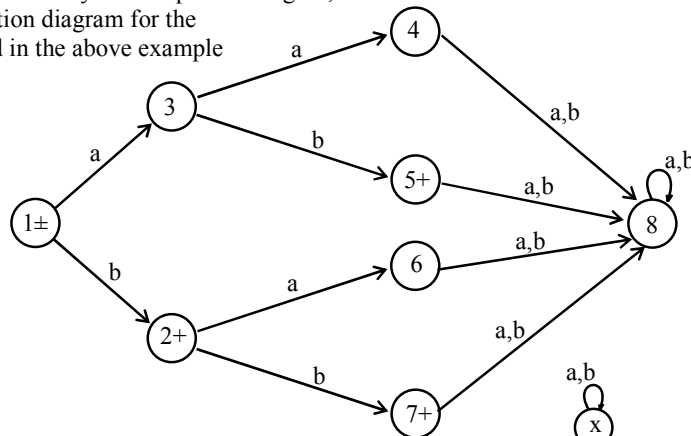


It is to be noted that the states x and y are called

**Dead States, Waste Baskets or Davey John Lockers**, as the moment one enters these states there is no way to leave it.

Note

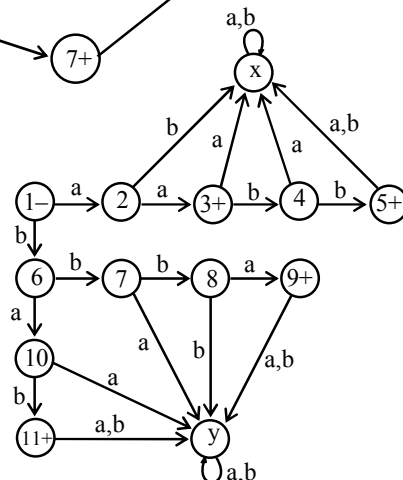
It is to be noted that to build an FA accepting the language having less number of strings, the tree structure may also help in this regard, which can be observed in the following transition diagram for the Language L, discussed in the above example

Example

Consider the language

$L = \{aa, bab, aabb, bbba\}$ , defined over  $\Sigma = \{a, b\}$ , expressed by  $aa + bab + aabb + bbba$  OR  $aa(\Lambda + bb) + b(ab + bba)$

The above language may be accepted by the FA as shown aside

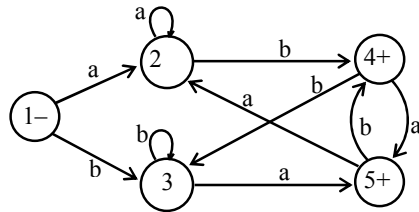


Example

Consider the language  $L = \{w \text{ belongs to } \{a,b\}^* : \text{length}(w) \geq 2 \text{ and } w \text{ neither ends in } \mathbf{aa} \text{ nor } \mathbf{bb}\}$ .

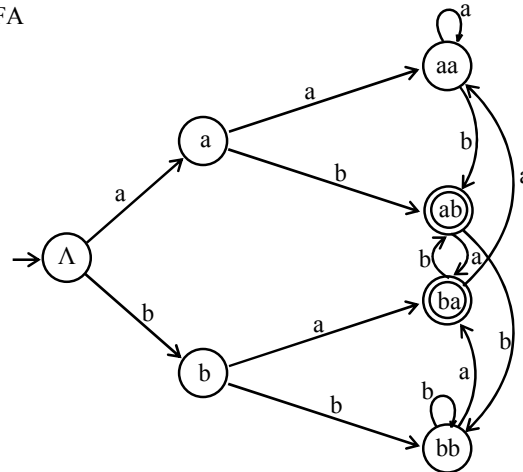
The language  $L$  may be expressed by the regular expression  $(a+b)^*(ab+ba)$

This language may be accepted by the following FA

Note

It is to be noted that building an FA corresponding to the language  $L$ , discussed in the above example, seems to be quite difficult, but the same can be done using tree structure along with the technique discussed in the book *Introduction to Languages and Theory of Computation*, by J. C. Martin

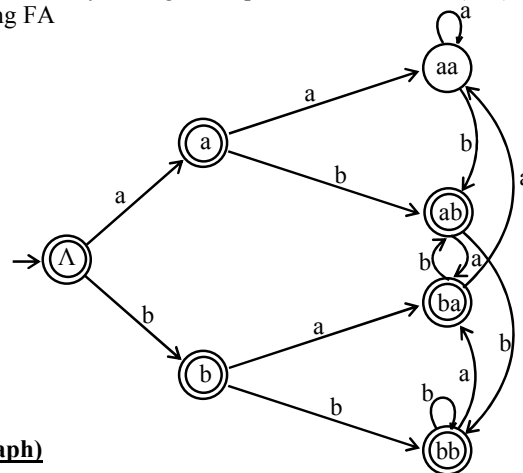
so that the strings ending in  $aa$ ,  $ab$ ,  $ba$  and  $bb$  should end in the states labeled as  $aa$ ,  $ab$ ,  $ba$  and  $bb$ , respectively; as shown in the following FA

Example

Consider the language FA corresponding to  $r_1 + r_2$  can be determined as

$L = \{w \text{ belongs to } \{a,b\}^* : w \text{ does not end in } \mathbf{aa}\}$ .

The language  $L$  may be expressed by the regular expression  $\Lambda + a + b + (a+b)^*(ab+ba+bb)$ . This language may be accepted by the following FA

Method 5 (Transition Graph)Definition:

A Transition graph (TG), is a collection of the followings

Finite number of states, at least one of which is start state and some (maybe none) final states.

Finite set of input letters ( $\Sigma$ ) from which input strings are formed.

Finite set of transitions that show how to go from one state to another based on reading specified substrings of input letters, possibly even the null string ( $\Lambda$ ).

## Theory of Automata

**Lecture N0. 8****Reading Material**Introduction to Computer Theory

## Chapter 6

**Summary**

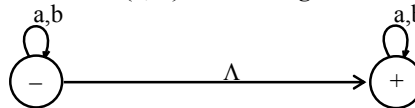
**Examples of TGs: accepting all strings, accepting none, starting with b, not ending in b, containing aa, containing aa or bb**

Note

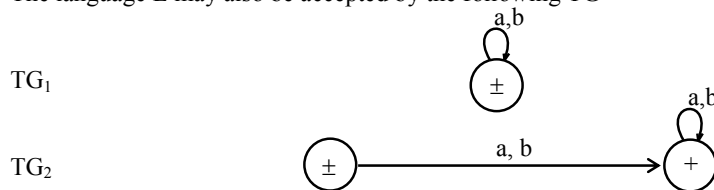
It is to be noted that in TG there may exist more than one paths for certain string, while there may not exist any path for certain string as well. If there exists at least one path for a certain string, starting from initial state and ending in a final state, the string is supposed to be accepted by the TG, otherwise the string is supposed to be rejected. Obviously collection of accepted strings is the language accepted by the TG.

Example

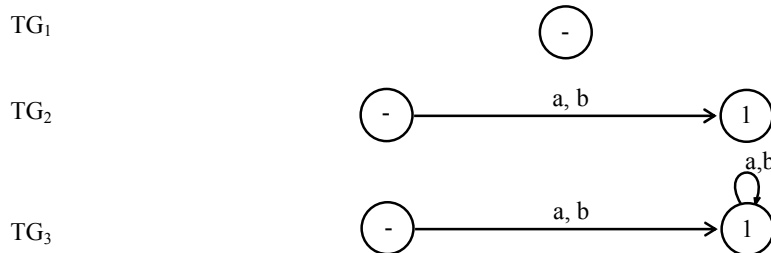
Consider the Language L, defined over  $\Sigma = \{a, b\}$  of **all strings including  $\Lambda$** . The language L may be accepted by the following TG



The language L may also be accepted by the following TG

Example

Consider the following TGs



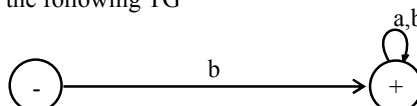
It may be observed that in the first TG, no transition has been shown. Hence this TG does not accept any string, defined over any alphabet. In TG<sub>2</sub> there are transitions for a and b at initial state but there is no transition at state 1. This TG still does not accept any string. In TG<sub>3</sub> there are transitions at both initial state and state 1, but it does not accept any string.

Thus none of TG<sub>1</sub>, TG<sub>2</sub> and TG<sub>3</sub> accepts any string, *i.e.* these TGs accept empty language. It may be noted that TG<sub>1</sub> and TG<sub>2</sub> are TGs but not FA, while TG<sub>3</sub> is both TG and FA as well.

It may be noted that every FA is a TG as well, but the converse may not be true, *i.e.* every TG may not be an FA.

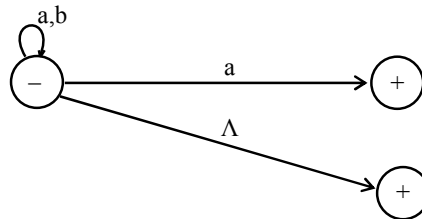
Example

Consider the language L of strings, defined over  $\Sigma = \{a, b\}$ , **starting with b**. The language L may be expressed by RE  $b(a + b)^*$ , may be accepted by the following TG



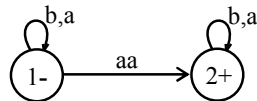
Example

Consider the language L of strings, defined over  $\Sigma = \{a, b\}$ , **not ending in b**. The language L may be expressed by RE  $\Lambda + (a + b)^*a$ , may be accepted by the following TG

Example

Consider the Language L of strings, defined over  $\Sigma = \{a, b\}$ , **containing double a**.

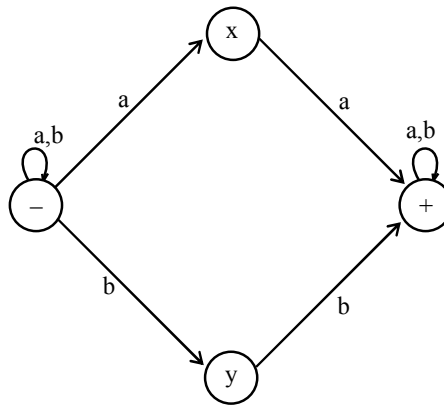
The language L may be expressed by the following regular expression  $(a+b)^*(aa)(a+b)^*$ . This language may be accepted by the following TG

Example

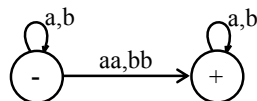
Consider the language L of strings, defined over  $\Sigma = \{a, b\}$ , **having double a or double b**.

The language L can be expressed by RE  $(a+b)^*(aa + bb)(a+b)^*$ .

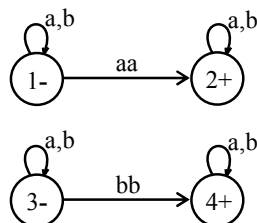
The above language may also be expressed by the following TGs.



**OR**



**OR**

Note

In the above TG if the states are not labeled then it may not be considered to be a single TG

## Theory of Automata

**Lecture N0. 9****Reading Material**Introduction to Computer Theory

## Chapter 6

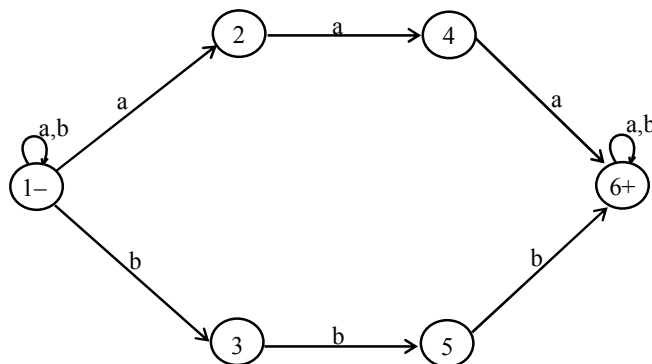
**Summary**

TGs accepting the languages: containing aaa or bbb, beginning and ending in different letters, beginning and ending in same letters, EVEN-EVEN, a's occur in even clumps and ends in three or more b's, example showing different paths traced by one string, Definition of GTG

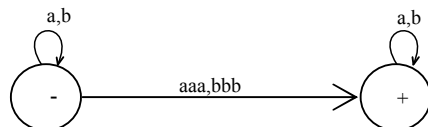
**Example**

Consider the language L of strings, defined over  $\Sigma = \{a, b\}$ , **having triple a or triple b**. The language L may be expressed by RE  $(a+b)^*(aaa + bbb)(a+b)^*$

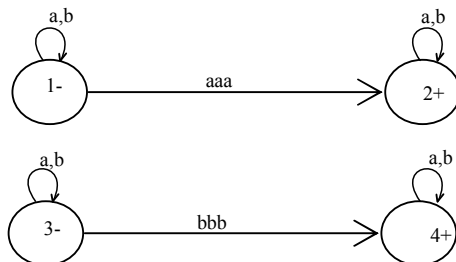
This language may be accepted by the following TG



OR



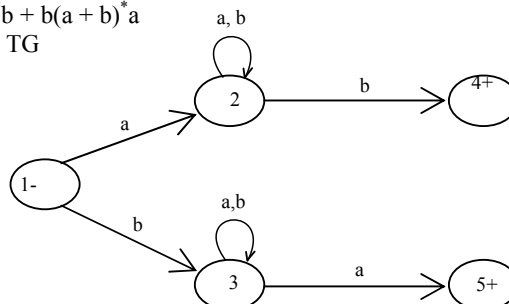
OR

**Example**

Consider the language L of strings, defined over  $\Sigma = \{a, b\}$ , **beginning and ending in different letters**.

The language L may be expressed by RE  $a(a+b)^*b + b(a+b)^*a$

The language L may be accepted by the following TG

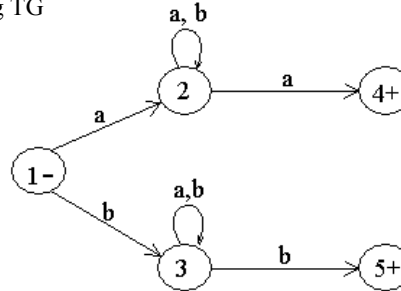


Example

Consider the Language L of strings **of length two or more**, defined over  $\Sigma = \{a, b\}$ , **beginning with and ending in same letters**.

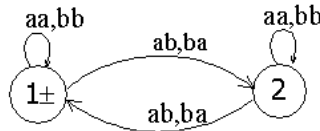
The language L may be expressed by the following regular expression  $a(a+b)^*a + b(a+b)^*b$

This language may be accepted by the following TG

Example

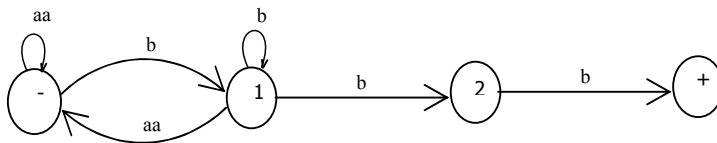
Consider the **EVEN-EVEN** language, defined over  $\Sigma = \{a, b\}$ . As discussed earlier that **EVEN-EVEN** language can be expressed by a regular expression  $(aa+bb+(ab+ba)(aa+bb)^*(ab+ba))^*$

The language **EVEN-EVEN** may be accepted by the following TG

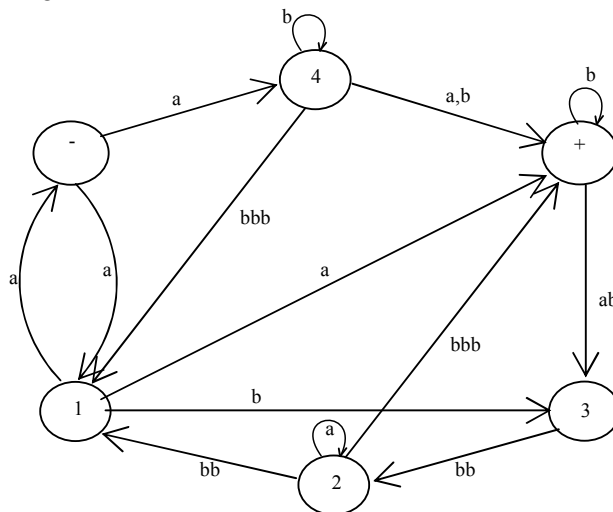
Example

Consider the language L, defined over  $\Sigma = \{a, b\}$ , in which **a's occur only in even clumps and that ends in three or more b's**. The language L can be expressed by its regular expression  $(aa)^*b(b^*+(aa(aa)^*b)^*)bb$  OR  $(aa)^*b(b^*+(aa)^*b)^*)bb$ .

The language L may be accepted by the following TG

Example

Consider the following TG



Consider the string abbbabbbabba. It may be observed that the above string traces the following three paths, (using the states)

(a)(b) (b) (b) (ab) (bb) (a) (bb) (a)

(-)(4)(4)(+)(+)(3)(2)(2)(1)(+)

(a)(b) ((b)(b)) (ab) (bb) (a) (bb) (a)

(-)(4)(+)(+)(+)(3)(2)(2)(1)(+)

(a)((b) (b)) (b) (ab) (bb) (a) (bb) (a)

(-) (4)(4)(4)(+) (3)(2)(2)(1)(+)

Which shows that all these paths are successful, (*i.e.* the path starting from an initial state and ending in a final state).

Hence the string abbbabbbabba is accepted by the given TG.

### **Generalized Transition Graphs**

A generalized transition graph (GTG) is a collection of three things

Finite number of states, at least one of which is start state and some (maybe none) final states.

Finite set of input letters ( $\Sigma$ ) from which input strings are formed.

Directed edges connecting some pair of states labeled with regular expression.

It may be noted that in GTG, the labels of transition edges are corresponding regular expressions

## Theory of Automata

**Lecture N0. 10****Reading Material**Introduction to Computer Theory

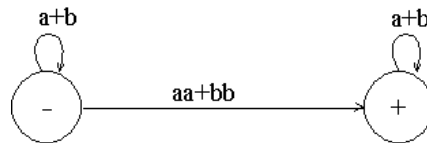
## Chapter 6, 7

**Summary**

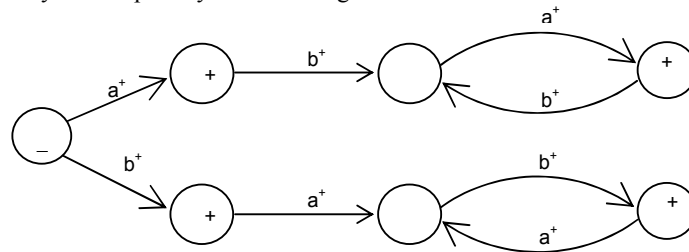
Examples of GTG accepting the languages of strings: containing aa or bb, beginning with and ending in same letters, beginning with and ending in different letters, containing aaa or bbb, Nondeterminism, Kleene's theorem (part I, part II, part III), proof of Kleene's theorem part I

Example

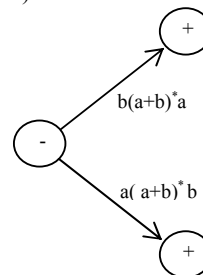
Consider the language L of strings, defined over  $\Sigma = \{a, b\}$ , containing **double a or double b**. The language L can be expressed by the following regular expression  $(a+b)^*(aa+bb)(a+b)^*$ . The language L may be accepted by the following GTG.

Example

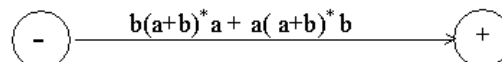
Consider the Language L of strings, defined over  $\Sigma = \{a, b\}$ , **beginning with and ending in same letters**. The language L may be expressed by the following regular expression  $(a+b)^+ a(a+b)^+ a + b(a+b)^+ b$ . This language may be accepted by the following GTG

Example

Consider the language L of strings of, defined over  $\Sigma = \{a, b\}$ , **beginning and ending in different letters**. The language L may be expressed by RE  $a(a+b)^*b + b(a+b)^*a$ . The language L may be accepted by the following GTG

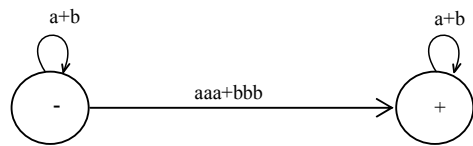


The language L may be accepted by the following GTG as well

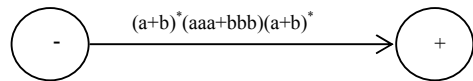
Example

Consider the language L of strings, defined over  $\Sigma = \{a, b\}$ , **having triple a or triple b**. The language L may be expressed by RE  $(a+b)^*(aaa+bbb)(a+b)^*$ . This language may be accepted by the following GTG





OR

**Nondeterminism**

TGs and GTGs provide certain relaxations *i.e.* there may exist more than one path for a certain string or there may not be any path for a certain string, this property creates **nondeterminism** and it can also help in differentiating TGs or GTGs from FAs. Hence an FA is also called a Deterministic Finite Automaton (DFA).

**Kleene's Theorem**

**If** a language can be expressed by

FA or

TG or

RE

**then** it can also be expressed by other two as well.

It may be noted that the theorem is proved, proving the following three parts

**Kleene's Theorem Part I**

If a language can be accepted by an FA then it can be accepted by a TG as well.

**Kleene's Theorem Part II**

If a language can be accepted by a TG then it can be expressed by an RE as well.

**Kleene's Theorem Part III**

If a language can be expressed by a RE then it can be accepted by an FA as well.

**Proof(Kleene's Theorem Part I)**

Since every FA can be considered to be a TG as well, therefore there is nothing to prove.

## Theory of Automata

**Lecture N0. 11****Reading Material**Introduction to Computer Theory

## Chapter 7

**Summary**

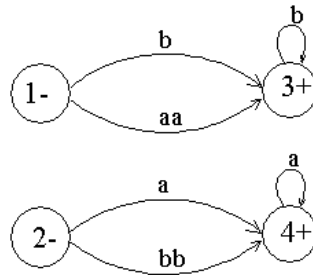
proof of Kleene's theorem part II (method with different steps), particular examples of TGs to determine corresponding REs.

**Proof(Kleene's Theorem Part II)**

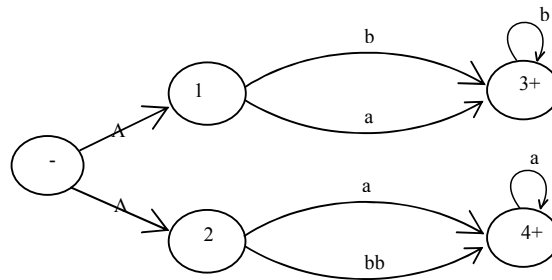
To prove part II of the theorem, an algorithm consisting of different steps, is explained showing how a RE can be obtained corresponding to the given TG. For this purpose the notion of TG is changed to that of GTG *i.e.* the labels of transitions are corresponding REs.

Basically this algorithm converts the given TG to GTG with one initial state along with a single loop, or one initial state connected with one final state by a single transition edge. The label of the loop or the transition edge will be the required RE.

**Step 1** If a TG has more than one start states, then introduce a new start state connecting the new state to the old start states by the transitions labeled by  $\Lambda$  and make the old start states the non-start states. This step can be shown by the following example

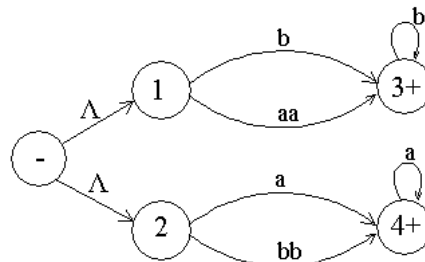
Example

The above TG can be converted to

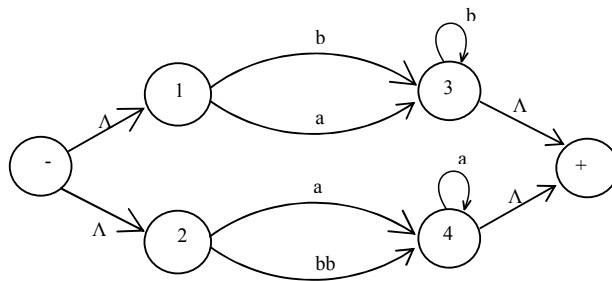
**Step 2:**

If a TG has more than one final states, then introduce a new final state, connecting the old final states to the new final state by the transitions labeled by  $\Lambda$ .

This step can be shown by the previous example of TG, where the step 1 has already been processed

Example

The above TG can be converted to

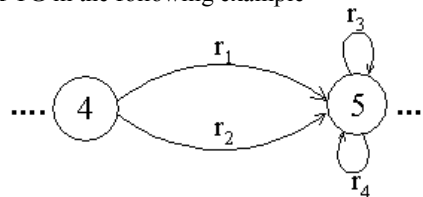


### Step 3:

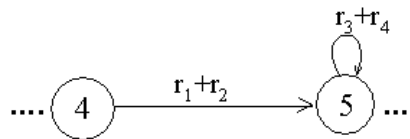
If a state has two (more than one) incoming transition edges labeled by the corresponding REs, from the same state (including the possibility of loops at a state), then replace all these transition edges with a single transition edge labeled by the sum of corresponding REs.

This step can be shown by a part of TG in the following example

#### Example

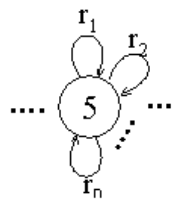


The above TG can be reduced to

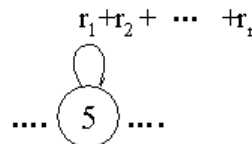


#### Note

The step 3 can be generalized to any finite number of transitions as shown below



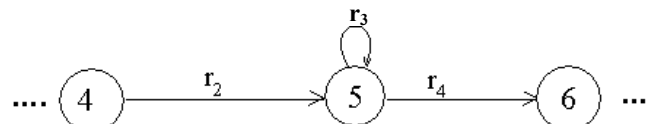
The above TG can be reduced to



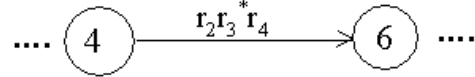
### Step 4 (bypass and state elimination)

If three states in a TG, are connected in sequence then eliminate the middle state and connect the first state with the third by a single transition (include the possibility of circuit as well) labeled by the RE which is the concatenation of corresponding two REs in the existing sequence. This step can be shown by a part of TG in the following example

#### Example



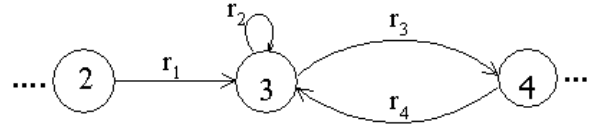
To eliminate state 5 the above can be reduced to



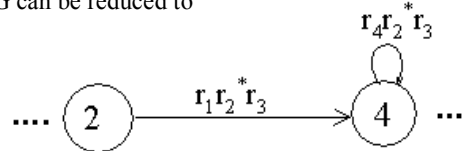
Consider the following example containing a circuit

#### Example

Consider the part of a TG, containing a circuit at a state, as shown below

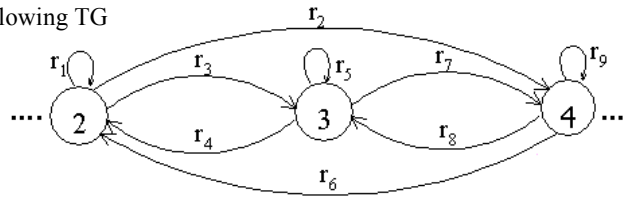


To eliminate state 3 the above TG can be reduced to

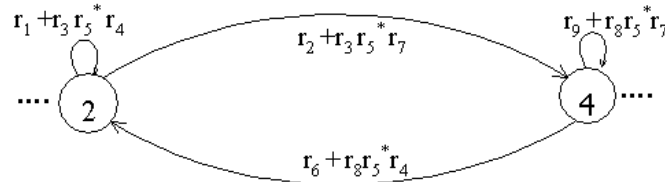


#### Example

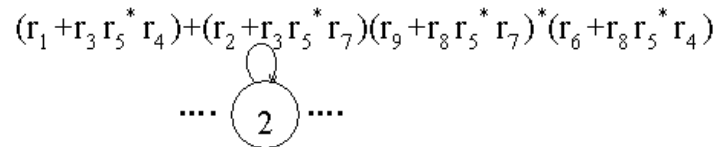
Consider a part of the following TG



To eliminate state 3 the above TG can be reduced to



To eliminate state 4 the above TG can be reduced to

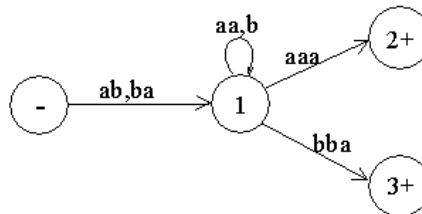


#### Note

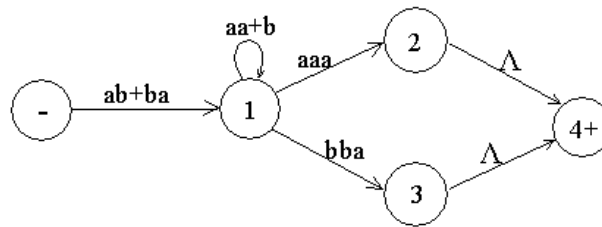
It is to be noted that to determine the RE corresponding to a certain TG, four steps have been discussed. This process can be explained by the following particular examples of TGs

#### Example

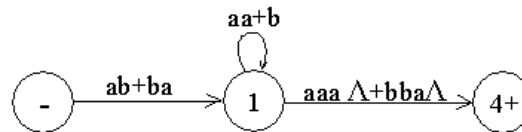
Consider the following TG



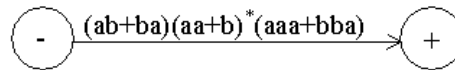
To have single final state, the above TG can be reduced to the following



To eliminate states 2 and 3, the above TG can be reduced to the following



To eliminate state 1 the above TG can be reduced to the following



Hence the required RE is  $(ab+ba)(aa+b)^*(aaa+bba)$

## Theory of Automata

**Lecture N0. 12****Reading Material**Introduction to Computer Theory

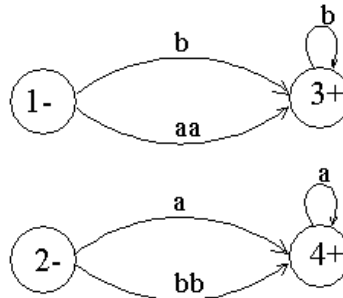
## Chapter 7

**Summary**

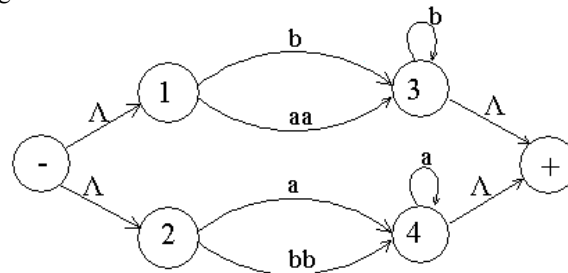
Examples of writing REs to the corresponding TGs, RE corresponding to TG accepting EVEN-EVEN language, Kleene's theorem part III (method 1: union of FAs), examples of FAs corresponding to simple REs, example of Kleene's theorem part III (method 1) continued

Example

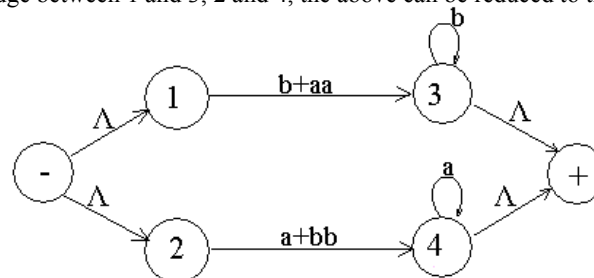
Consider the following TG



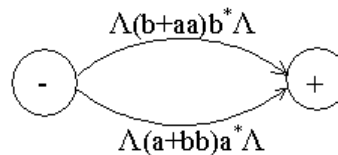
To have single initial and single final state the above TG can be reduced to the following



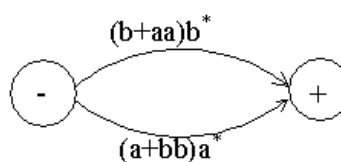
To obtain single transition edge between 1 and 3; 2 and 4, the above can be reduced to the following



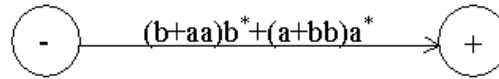
To eliminate states 1,2,3 and 4, the above TG can be reduced to the following TG



OR



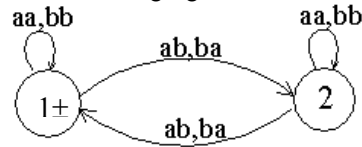
To connect the initial state with the final state by single transition edge, the above TG can be reduced to the following



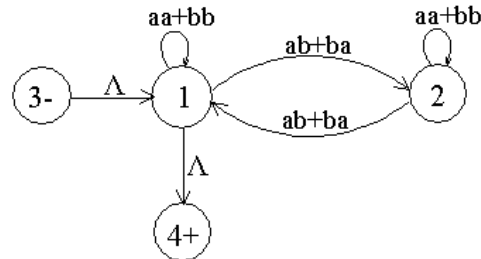
Hence the required RE is  $(b+aa)b^*+(a+bb)a^*$

#### Example

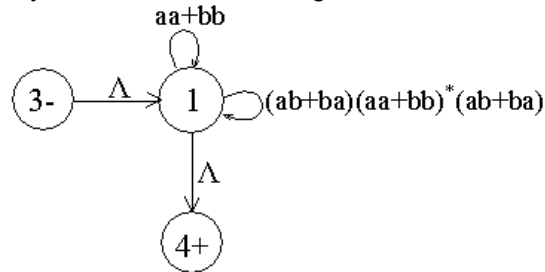
Consider the following TG, accepting EVEN-EVEN language



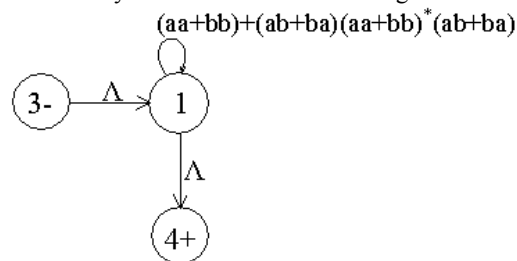
It is to be noted that since the initial state of this TG is final as well and there is no other final state, so to obtain a TG with single initial and single final state, an additional initial and a final state are introduced as shown in the following TG



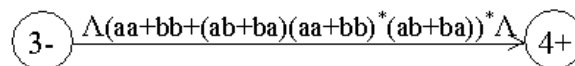
To eliminate state 2, the above TG may be reduced to the following



To have single loop at state 1, the above TG may be reduced to the following



To eliminate state 1, the above TG may be reduced to the following



Hence the required RE is  $(aa+bb+(ab+ba)(aa+bb)^*(ab+ba))^*$

#### Kleene's Theorem Part III

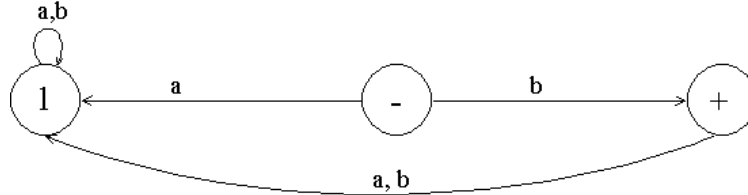
##### Statement:

If the language can be expressed by a RE then there exists an FA accepting the language.

As the regular expression is obtained applying addition, concatenation and closure on the letters of an alphabet and the Null string, so while building the RE, sometimes, the corresponding FA may be built easily, as shown in the following examples

#### Example

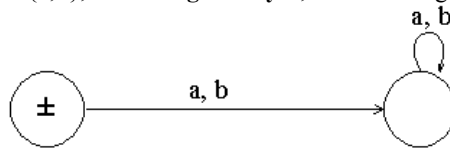
Consider the language, defined over  $\Sigma = \{a,b\}$ , **consisting of only b**, then this language may be accepted by the following FA



which shows that this FA helps in building an FA accepting only one letter

#### Example

Consider the language, defined over  $\Sigma = \{a,b\}$ , **consisting of only a**, then this language may be accepted by the following FA

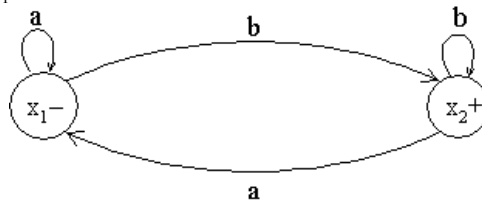


As, if  $r_1$  and  $r_2$  are regular expressions then their sum, concatenation and closure are also regular expressions, so an FA can be built for any regular expression if the methods can be developed for building the FAs corresponding to the sum, concatenation and closure of the regular expressions along with their FAs. These three methods are explained in the following discussion

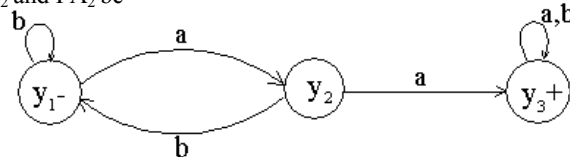
**Method1 (Union of two FAs):** Using the FAs corresponding to  $r_1$  and  $r_2$  an FA can be built, corresponding to  $r_1 + r_2$ . This method can be developed considering the following examples

#### Example

Let  $r_1 = (a+b)^*b$  defines  $L_1$  and the  $FA_1$  be



and  $r_2 = (a+b)^*aa(a+b)^*$  defines  $L_2$  and  $FA_2$  be

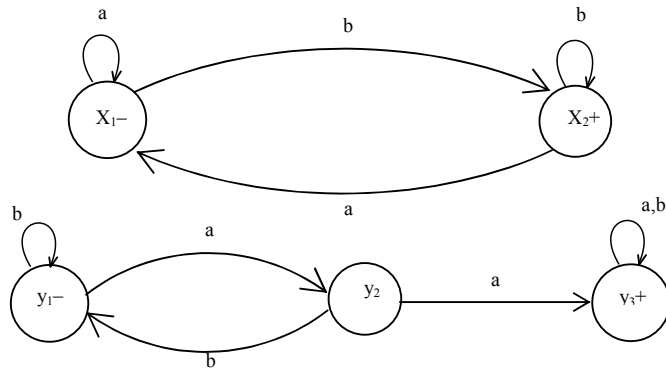


Let  $FA_3$  be an FA corresponding to  $r_1 + r_2$ , then the initial state of  $FA_3$  must correspond to the initial state of  $FA_1$  and the initial state of  $FA_2$ .

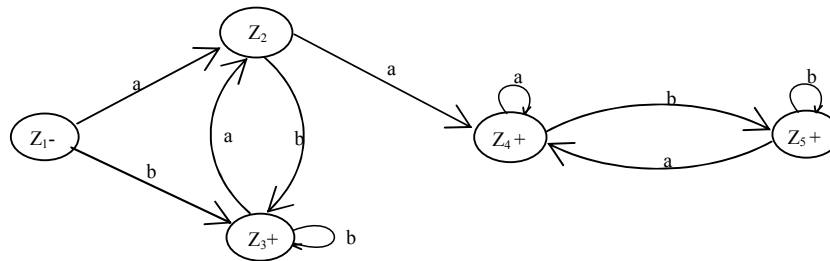
Since the language corresponding to  $r_1 + r_2$  is the union of corresponding languages  $L_1$  and  $L_2$ , consists of the strings belonging to  $L_1$  or  $L_2$  or both, therefore a final state of  $FA_3$  must correspond to a final state of  $FA_1$  or  $FA_2$  or both.

Since, in general,  $FA_3$  will be different from both  $FA_1$  and  $FA_2$ , so the labels of the states of  $FA_3$  may be supposed to be  $z_1, z_2, z_3, \dots$ , where  $z_1$  is supposed to be the initial state. Since  $z_1$  corresponds to the states  $x_1$  or  $y_1$ , so there will be two transitions separately for each letter read at  $z_1$ . It will give two possibilities of states either  $z_1$  or different from  $z_1$ . This process may be expressed in the following transition table for all possible states of  $FA_3$ .





Old States	New States after reading	
	a	b
$z_1 \equiv (x_1, y_1)$	$(x_1, y_2) \equiv z_2$	$(x_2, y_1) \equiv z_3$
$z_2 \equiv (x_1, y_2)$	$(x_1, y_3) \equiv z_4$	$(x_2, y_1) \equiv z_3$
$z_3^+ \equiv (x_2, y_1)$	$(x_1, y_2) \equiv z_2$	$(x_2, y_1) \equiv z_3$
$z_4^+ \equiv (x_1, y_3)$	$(x_1, y_3) \equiv z_4$	$(x_2, y_3) \equiv z_5$
$z_5^+ \equiv (x_2, y_3)$	$(x_1, y_3) \equiv z_4$	$(x_2, y_3) \equiv z_5$



RE corresponding to the above FA may be  $r_1 + r_2 = (a+b)^*b + (a+b)^*aa(a+b)^*$ .

Note: Further examples are discussed in the next lecture.

## Theory of Automata

**Lecture N0. 13****Reading Material**Introduction to Computer Theory

## Chapter 7

**Summary**

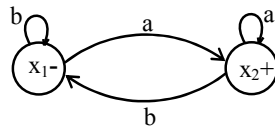
Examples of Kleene's theorem part III (method 1) continued, Kleene's theorem part III (method 2: Concatenation of FAs), Example of Kleene's theorem part III (method 2 : Concatenation of FAs)

**Note**

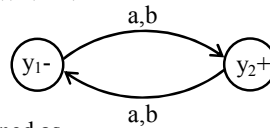
It may be noted that the example discussed at the end of previous lecture,  $FA_1$  contains two states while  $FA_2$  contains three states. Hence the total number of possible combinations of states of  $FA_1$  and  $FA_2$ , in sequence, will be six. For each combination the transitions for both a and b can be determined, but using the method in the example, number of states of  $FA_3$  was reduced to five.

**Example**

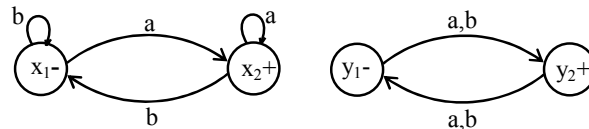
Let  $r_1 = (a+b)^*a$  and the corresponding  $FA_1$  be



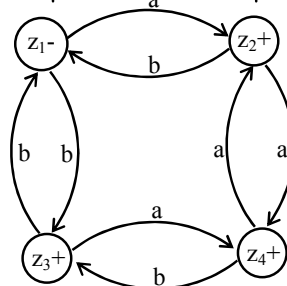
also  $r_2 = (a+b)((a+b)(a+b))^* \text{ or } ((a+b)(a+b))^*(a+b)$  and  $FA_2$  be



$FA$  corresponding to  $r_1+r_2$  can be determined as

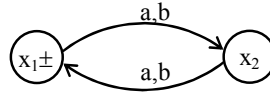


Old States	New States after reading	
	a	b
$z_1 \equiv (x_1, y_1)$	$(x_2, y_2) \equiv z_2$	$(x_1, y_2) \equiv z_3$
$z_2 \equiv (x_2, y_2)$	$(x_2, y_1) \equiv z_4$	$(x_1, y_1) \equiv z_1$
$z_3 \equiv (x_1, y_2)$	$(x_2, y_1) \equiv z_4$	$(x_1, y_1) \equiv z_1$
$z_4 \equiv (x_2, y_1)$	$(x_2, y_2) \equiv z_2$	$(x_1, y_2) \equiv z_3$

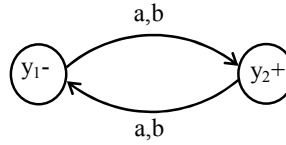


**Example**

Let  $r_1 = ((a+b)(a+b))^*$  and the corresponding FA<sub>1</sub> be



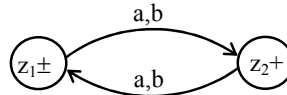
also  $r_2 = (a+b)((a+b)(a+b))^* (a+b)$  and FA<sub>2</sub> be



FA corresponding to  $r_1+r_2$  can be determined as

Old States	New States after reading	
	a	b
$z_1 \pm \equiv (x_1, y_1)$	$(x_2, y_2) \equiv z_2$	$(x_2, y_2) \equiv z_2$
$z_2 \pm \equiv (x_2, y_2)$	$(x_1, y_1) \equiv z_1$	$(x_1, y_1) \equiv z_1$

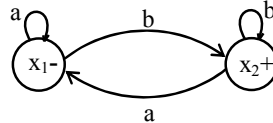
Hence the required FA will be as follows

**Method2 (Concatenation of two FAs):**

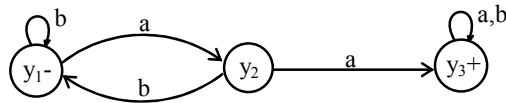
Using the FAs corresponding to  $r_1$  and  $r_2$ , an FA can be built, corresponding to  $r_1r_2$ . This method can be developed considering the following examples

**Example**

Let  $r_1 = (a+b)^*b$  defines  $L_1$  and FA<sub>1</sub> be

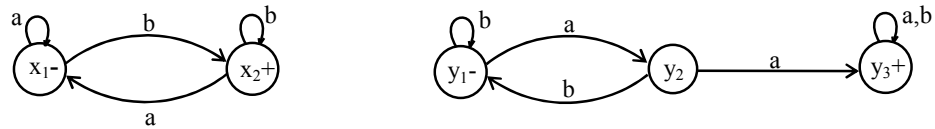


and  $r_2 = (a+b)^*aa(a+b)^*$  defines  $L_2$  and FA<sub>2</sub> be



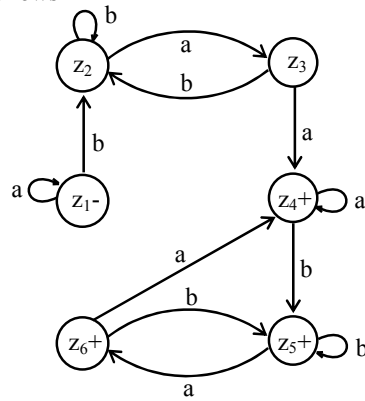
Let FA<sub>3</sub> be an FA corresponding to  $r_1r_2$ , then the initial state of FA<sub>3</sub> must correspond to the initial state of FA<sub>1</sub> and the final state of FA<sub>3</sub> must correspond to the final state of FA<sub>2</sub>. Since the language corresponding to  $r_1r_2$  is the concatenation of corresponding languages  $L_1$  and  $L_2$ , consists of the strings obtained, concatenating the strings of  $L_1$  to those of  $L_2$ , therefore **the moment a final state of first FA is entered, the possibility of the initial state of second FA will be included as well.**

Since, in general, FA<sub>3</sub> will be different from both FA<sub>1</sub> and FA<sub>2</sub>, so the labels of the states of FA<sub>3</sub> may be supposed to be  $z_1, z_2, z_3, \dots$ , where  $z_1$  stands for the initial state. Since  $z_1$  corresponds to the states  $x_1$ , so there will be two transitions separately for each letter read at  $z_1$ . It will give two possibilities of states which correspond to either  $z_1$  or different from  $z_1$ . This process may be expressed in the following transition table for all possible states of FA<sub>3</sub>



Old States	New States after reading	
	a	b
$z_1 \equiv x_1$	$x_1 \equiv z_1$	$(x_2, y_1) \equiv z_2$
$z_2 \equiv (x_2, y_1)$	$(x_1, y_2) \equiv z_3$	$(x_2, y_1) \equiv z_2$
$z_3 \equiv (x_1, y_2)$	$(x_1, y_3) \equiv z_4$	$(x_2, y_1) \equiv z_2$
$z_4 \equiv (x_1, y_3)$	$(x_1, y_3) \equiv z_4$	$(x_2, y_1, y_3) \equiv z_5$
$z_5 \equiv (x_2, y_1, y_3)$	$(x_1, y_2, y_3) \equiv z_6$	$(x_2, y_1, y_3) \equiv z_5$
$z_6 \equiv (x_1, y_2, y_3)$	$(x_1, y_3) \equiv z_4$	$(x_2, y_1, y_3) \equiv z_5$

Hence the required FA will be as follows



Note: Another example is discussed in the next lecture.

## Theory of Automata

**Lecture N0. 14**Reading Material

## Introduction to Computer Theory

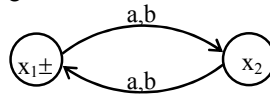
## Chapter 7

**Summary**

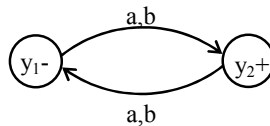
Examples of Kleene's theorem part III (method 1) continued, Kleene's theorem part III (method 2: Concatenation of FAs), Examples of Kleene's theorem part III (method 2: concatenation FAs) continued, Kleene's theorem part III (method 3: closure of an FA), examples of Kleene's theorem part III (method 3: Closure of an FA) continued

**Example**

Let  $r_1 = ((a+b)(a+b))^*$  and the corresponding FA<sub>1</sub> be



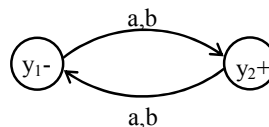
also  $r_2 = (a+b)((a+b)(a+b))^* \text{ or } ((a+b)(a+b))^* (a+b)$  and FA<sub>2</sub> be



FA corresponding to  $r_1 r_2$  can be determined as

Old States	New States after reading	
	a	b
$z_1 \pm \equiv (x_1, y_1)$	$(x_2, y_2) \equiv z_2$	$(x_2, y_2) \equiv z_2$
$z_2^+ \equiv (x_2, y_2)$	$(x_1, y_1) \equiv z_1$	$(x_1, y_1) \equiv z_1$

Hence the required FA will be as follows

**Method3: (Closure of an FA)**

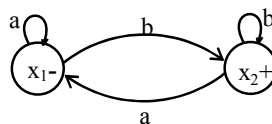
Building an FA corresponding to  $r^*$ , using the FA corresponding to  $r$ .

It is to be noted that if the given FA already accepts the language expressed by the closure of certain RE, then the given FA is the required FA. However the method, in other cases, can be developed considering the following examples

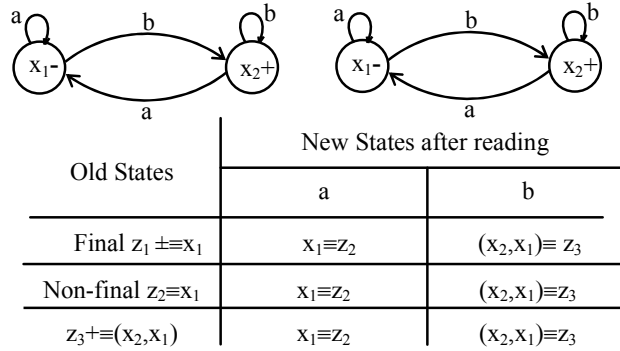
Closure of an FA, is same as concatenation of an FA with itself, except that the initial state of the required FA is a final state as well. Here the initial state of given FA, corresponds to the initial state of required FA and a non final state of the required FA as well.

**Example**

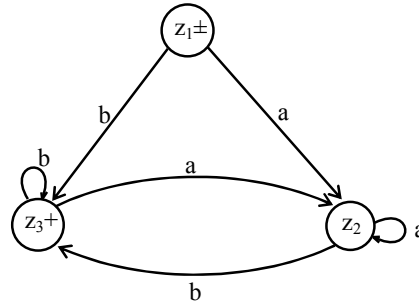
Let  $r = (a+b)^* b$  and the corresponding FA be



then the FA corresponding to  $r^*$  may be determined as under

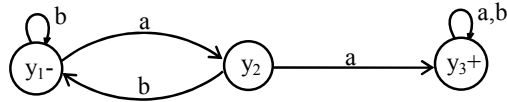


The corresponding transition diagram may be as under



#### Example

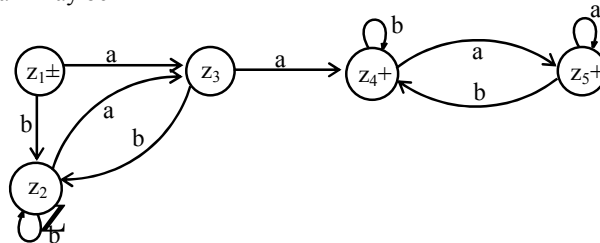
Let  $r = (a+b)^*aa(a+b)^*$  and the corresponding FA be



then the FA corresponding to  $r^*$  may be determined as under

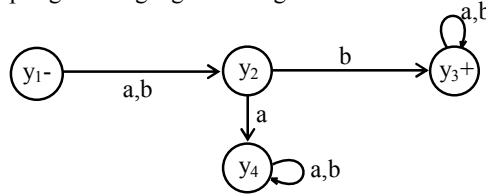
Old States	New States after reading	
	a	b
Final $z_1 \pm \equiv y_1$	$y_2 \equiv z_3$	$y_1 \equiv z_2$
Non-Final $z_2 \equiv y_1$	$y_2 \equiv z_3$	$y_1 \equiv z_2$
$z_3 \equiv y_2$	$(y_3, y_1) \equiv z_4$	$y_1 \equiv z_2$
$z_4^+ \equiv (y_3, y_1)$	$(y_3, y_1, y_2) \equiv z_5$	$(y_3, y_1) \equiv z_4$
$z_5^+ \equiv (y_3, y_1, y_2)$	$(y_3, y_1, y_2) \equiv z_5$	$(y_3, y_1) \equiv z_4$

The corresponding transition diagram may be



Example

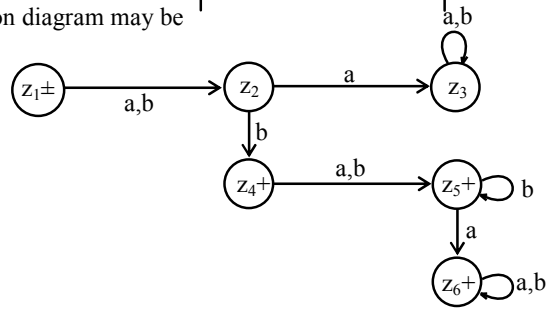
Consider the following FA, accepting the language of strings with **b** as second letter



then the FA corresponding to  $r^*$  may be determined as under

Old States	New States after reading	
	a	b
$z_1 \pm \equiv y_1$	$y_2 \equiv z_2$	$y_2 \equiv z_2$
$z_2 \equiv y_2$	$y_4 \equiv z_3$	$(y_3, y_1) \equiv z_4$
$z_3 \equiv y_4$	$y_4 \equiv z_3$	$y_4 \equiv z_3$
$z_4^+ \equiv (y_3, y_1)$	$(y_3, y_1, y_2) \equiv z_5$	$(y_3, y_1, y_2) \equiv z_5$
$z_5^+ \equiv (y_3, y_1, y_2)$	$(y_3, y_1, y_2, y_4) \equiv z_6$	$(y_3, y_1, y_2) \equiv z_5$
$z_6 \equiv (y_3, y_1, y_2, y_4)$	$(y_3, y_1, y_2, y_4) \equiv z_6$	$(y_3, y_1, y_2, y_4) \equiv z_6$

The corresponding transition diagram may be



## Theory of Automata

**Lecture N0. 15****Reading Material**Introduction to Computer Theory

## Chapter 7

**Summary**

Examples of Kleene's theorem part III (method 3), NFA, examples, avoiding loop using NFA, example, converting FA to NFA, examples, applying an NFA on an example of maze

**Note**

It is to be noted that as observed in the examples discussed in previous lecture, if at the initial state of the given FA, there is either a loop or an incoming transition edge, the initial state corresponds to the final state and a non-final state as well, of the required FA, otherwise the initial state of given FA will only correspond to a single state of the required FA (*i.e.* the initial state which is final as well).

**Nondeterministic Finite Automaton (NFA)****Definition**

An NFA is a TG with a unique start state and a property of having single letter as label of transitions. An NFA is a collection of three things

Finite many states with one initial and some final states

Finite set of input letters, say,  $\Sigma = \{a, b, c\}$

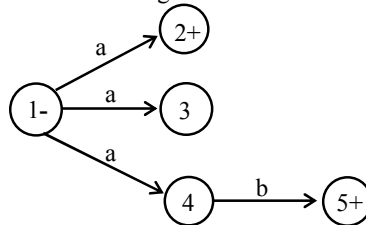
Finite set of transitions, showing where to move if a letter is input at certain state ( $\wedge$  is not a valid transition), there may be more than one transition for certain letters and there may not be any transition for certain letters.

**Observations**

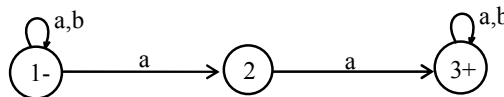
It may be observed, from the definition of NFA, that the string is supposed to be accepted, if there exists at least one successful path, otherwise rejected.

It is to be noted that an NFA can be considered to be an intermediate structure between FA and TG.

The examples of NFAs can be found in the following

**Example**

It is to be noted that the above NFA accepts the language consisting of a and ab.

**Example**

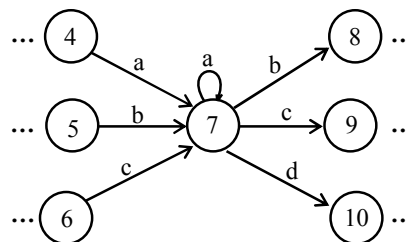
It is to be noted that the above NFA accepts the language of strings, defined over  $\Sigma = \{a, b\}$ , containing aa.

**Note**

It is to be noted that NFA helps to eliminate a loop at certain state of an FA. This process is done converting the loop into a circuit. But during this process the FA remains no longer FA and is converted to a corresponding NFA, which is shown in the following example.

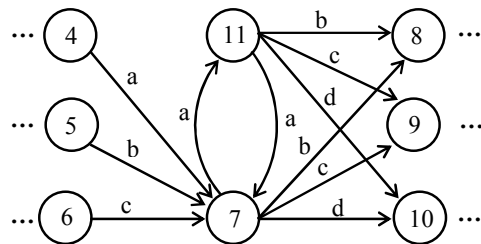
**Example**

Consider a part of the following FA with an alphabet  $\Sigma = \{a, b, c, d\}$





To eliminate the loop at state 7, the corresponding NFA may be as follows

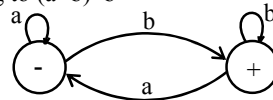


### Converting an FA to an equivalent NFA

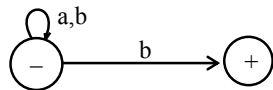
It is to be noted that according to the Kleene's theorem, if a language can be accepted by an FA, then there exists a TG accepting that language. Since, an NFA is a TG as well, therefore there exists an NFA accepting the language accepted by the given FA. In this case these FA and NFA are said to be equivalent to each others. Following are the examples of FAs to be converted to the equivalent NFAs

#### Example

Consider the following FA corresponding to  $(a+b)^*b$



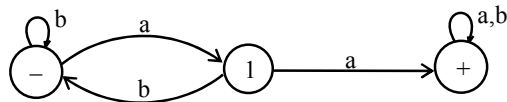
The above FA may be equivalent to the following NFA



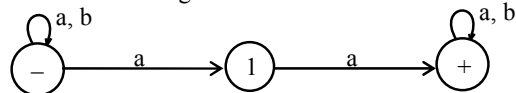
Can the structure of above NFA be compared with the corresponding RE ?

#### Example

Consider the following FA



The above FA may be equivalent to the following NFA



Can the structure of above NFA be compared with the corresponding RE ?

### Application of an NFA

There is an important application of an NFA in artificial intelligence, which is discussed in the following example of a maze

-	1	2	3
4	L	5	O
6	M	7	P
8	N	9	+

## Theory of Automata

**Lecture N0. 16****Reading Material**Introduction to Computer Theory

## Chapter 7

**Summary**

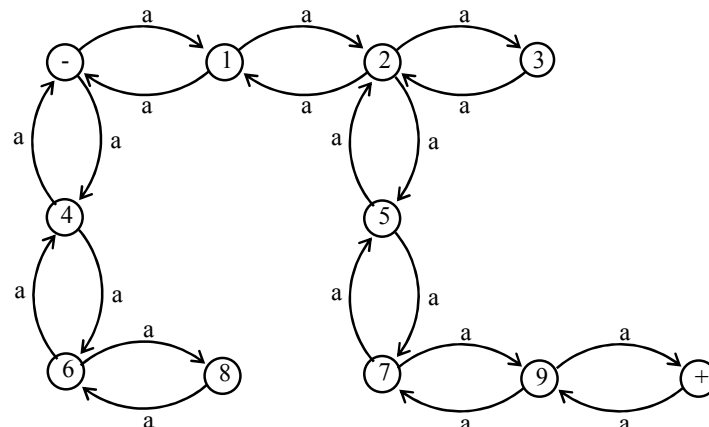
Applying an NFA on an example of maze, NFA with null string, examples, RE corresponding to NFA with null string (task), converting NFA to FA (method 1,2,3) examples

**Application of an NFA**

There is an important application of an NFA in artificial intelligence, which is discussed in the following example of a maze

-	1	2	3
4	L	5	O
6	M	7	P
8	N	9	+

- and + indicate the initial and final states respectively. One can move only from a box labeled by other then L, M, N, O, P to such another box. To determine the number of ways in which one can start from the initial state and end in the final state, the following NFA using only single letter a, can help in this regard



It can be observed that the shortest path which leads from the initial state and ends in the final state, consists of six steps i.e. the shortest string accepted by this machine is aaaaaa. The next larger accepted string is aaaaaaaa. Thus if this NFA is considered to be a TG then the corresponding regular expression may be written as aaaaaa(aa)\*

Which shows that there are infinite many required ways

**Note**

It is to be noted that every FA can be considered to be an NFA as well, but the converse may not true.

It may also be noted that every NFA can be considered to be a TG as well, but the converse may not true.

It may be observed that if the transition of null string is also allowed at any state of an NFA then what will be the behavior in the new structure. This structure is defined in the following

**NFA with Null String****Definition**

If in an NFA,  $\Lambda$  is allowed to be a label of an edge then the NFA is called NFA with  $\Lambda$  (NFA- $\Lambda$ ).

An NFA- $\Lambda$  is a collection of three things

Finite many states with one initial and some final states.

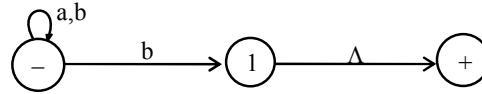
Finite set of input letters, say,  $\Sigma = \{a, b, c\}$ .

Finite set of transitions, showing where to move if a letter is input at certain state.

There may be more than one transitions for certain letter and there may not be any transition for a certain letter. The transition of  $\Lambda$  is also allowed at any state.

#### Example

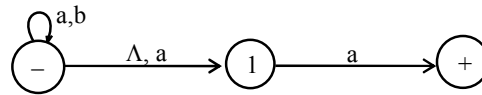
Consider the following NFA with Null string



The above NFA with Null string accepts the language of strings, defined over  $\Sigma = \{a, b\}$ , **ending in b**.

#### Example

Consider the following NFA with Null string



The above NFA with Null string accepts the language of strings, defined over  $\Sigma = \{a, b\}$ , **ending in a**.

#### Note

It is to be noted that every FA may be considered to be an NFA- $\Lambda$  as well, but the converse may not true. Similarly every NFA- $\Lambda$  may be considered to be a TG as well, but the converse may not true.

### NFA to FA

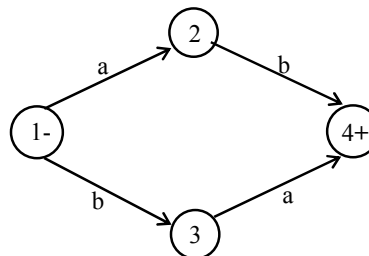
Two methods are discussed in this regard.

**Method 1:** Since an NFA can be considered to be a TG as well, so a RE corresponding to the given NFA can be determined (using Kleene's theorem). Again using the methods discussed in the proof of Kleene's theorem, an FA can be built corresponding to that RE. Hence for a given NFA, an FA can be built equivalent to the NFA. Examples have, indirectly, been discussed earlier.

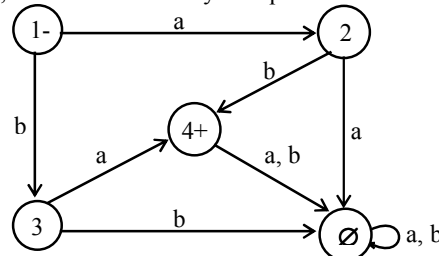
**Method 2:** Since in an NFA, there may be more than one transition for a certain letter and there may not be any transition for certain letter, so starting from the initial state corresponding to the initial state of given NFA, the transition diagram of the corresponding FA, can be built introducing an empty state for a letter having no transition at certain state and a state corresponding to the combination of states, for a letter having more than one transitions. Following are the examples

#### Example

Consider the following NFA

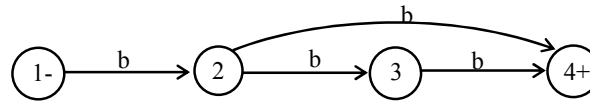


Using the method discussed earlier, the above NFA may be equivalent to the following FA

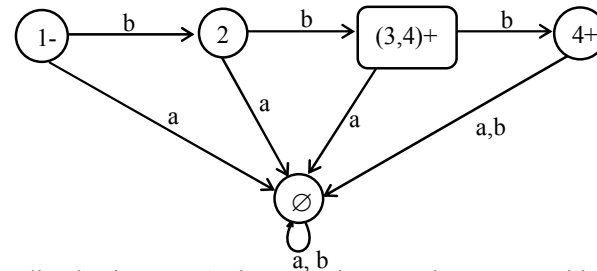


Example

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



The above NFA can be converted to the following FA



**Method 3:** As discussed earlier that in an NFA, there may be more than one transition for a certain letter and there may not be any transition for certain letter, so starting from the initial state corresponding to the initial state of given NFA, the transition table along with new labels of states, of the corresponding FA, can be built introducing an empty state for a letter having no transition at certain state and a state corresponding to the combination of states, for a letter having more than one transitions. Further examples are discussed in the next lecture.

## Theory of Automata

### Lecture N0. 17

#### Reading Material

#### Introduction to Computer Theory

#### Chapter 7

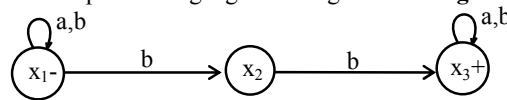
#### Summary

converting NFA to FA (method 3), example, NFA and Kleene's theorem method 1, examples, NFA and Kleene's theorem method 2, NFA corresponding to union of FAs, example

**Method 3:** As discussed earlier that in an NFA, there may be more than one transition for a certain letter and there may not be any transition for certain letter, so starting from the initial state corresponding to the initial state of given NFA, the transition table along with new labels of states, of the corresponding FA, can be built introducing an empty state for a letter having no transition at certain state and a state corresponding to the combination of states, for a letter having more than one transitions. Following are the examples

#### Example

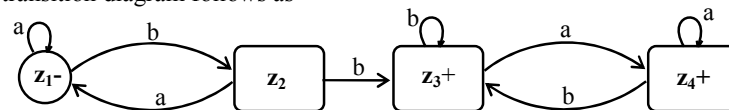
Consider the following NFA which accepts the language of strings **containing bb**



Using the method discussed earlier, the transition table corresponding to the required FA may be constructed as

Old States	New States after reading	
	a	b
$z_1 \equiv x_1$	$x_1 \equiv z_1$	$(x_1, x_2) \equiv z_2$
$z_2 \equiv (x_1, x_2)$	$(x_1, \emptyset) \equiv x_1 \equiv z_1$	$(x_1, x_2, x_3) \equiv z_3$
$z_3 \equiv (x_1, x_2, x_3)$	$(x_1, x_3) \equiv z_4$	$(x_1, x_2, x_3) \equiv z_3$
$z_4 \equiv (x_1, x_3)$	$(x_1, x_3) \equiv z_4$	$(x_1, x_2, x_3) \equiv z_3$

The corresponding transition diagram follows as



#### NFA and Kleene's Theorem

It has been discussed that, by Kleene's theorem part III, there exists an FA corresponding to a given RE. If the given RE is as simple as  $r = aa+bbb$  or  $r = a(a+b)^*$ , the corresponding FAs can easily be constructed. However, for a complicated RE, the RE can be decomposed into simple REs corresponding to which the FAs can easily be constructed and hence, using the method, constructing the FAs corresponding to sum, concatenation and closure of FAs, the required FA can also be constructed. It is to be noted that NFAs also help in proving Kleene's theorem part III, as well. Two methods are discussed in the following.

#### NFA and Kleene's Theorem

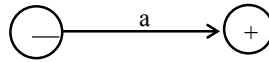
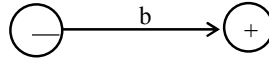
##### Method 1:

The method is discussed considering the following example.

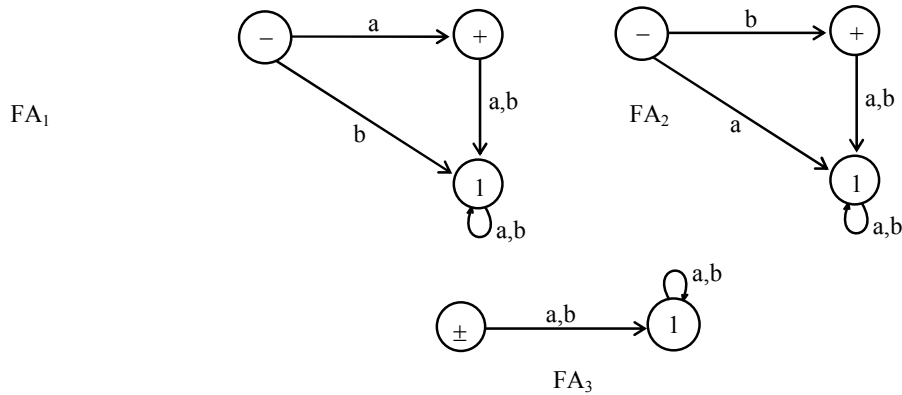
#### Example

To construct the FAs for the languages  $L_1 = \{a\}$ ,  $L_2 = \{b\}$  and  $L_3 = \{\wedge\}$

**Step 1:** Build  $NFA_1$ ,  $NFA_2$  and  $NFA_3$  corresponding to  $L_1$ ,  $L_2$  and  $L_3$ , respectively as shown in the following diagram

NFA<sub>1</sub>NFA<sub>2</sub>NFA<sub>3</sub>Step 2:

As discussed earlier for every NFA there is an FA equivalent to it, hence there must be FAs for the above mentioned NFAs as well. The corresponding FAs can be considered as follows

**NFA and Kleene's Theorem method 2**

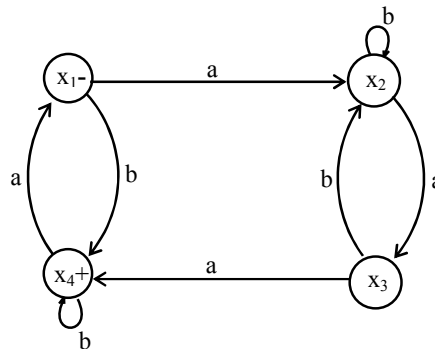
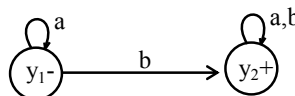
It may be observed that if an NFA can be built corresponding to union, concatenation and closure of FAs corresponding to the REs, then converting the NFA, thus, obtained into an equivalent FA, this FA will correspond to the given RE.

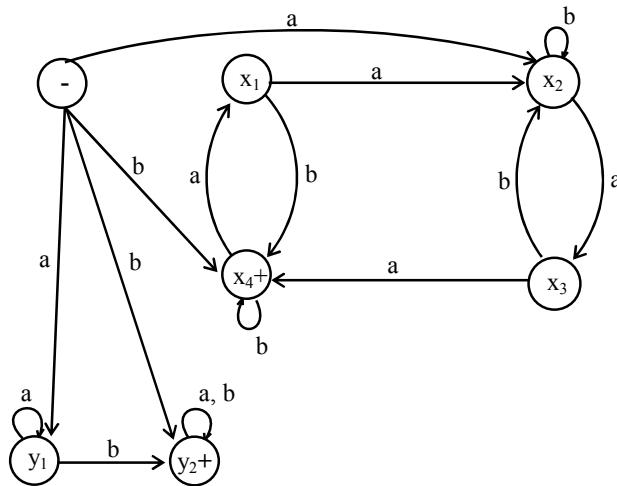
Followings are the procedures showing how to obtain NFAs equivalent to union, concatenation and closure of FAs

NFA corresponding to Union of FAs

**Method**

Introduce a new start state and connect it with the states originally connected with the old start state with the same transitions as the old start state, then remove the -ve sign of old start state. This creates non-determinism and hence results in an NFA.

**Example**FA<sub>1</sub>FA<sub>2</sub>



NFA equivalent to  $FA_1 \cup FA_2$

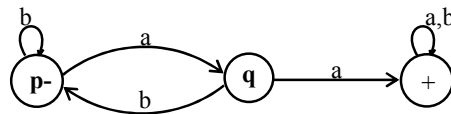
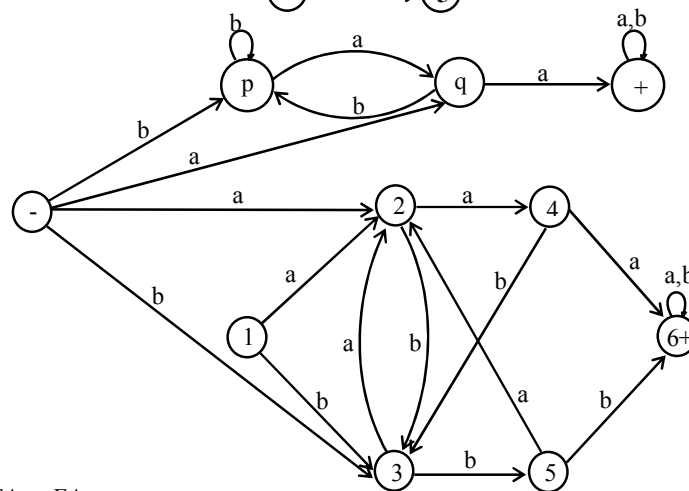
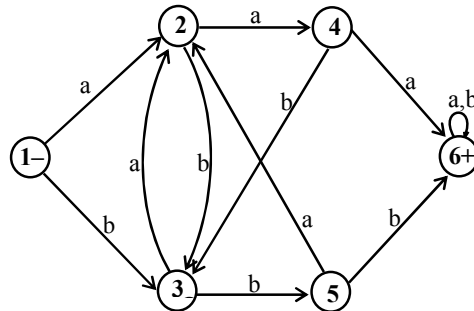
## Theory of Automata

**Lecture N0. 18****Reading Material**Introduction to Computer Theory

## Chapter 7

## Summary

NFA corresponding to union of FAs, example, NFA corresponding to concatenation of FAs, examples, NFA corresponding to closure of an FA, example

**Example**FA<sub>1</sub>FA<sub>2</sub>NFA equivalent to  $FA_1 \cup FA_2$ NFA corresponding to Concatenation of FAsMethod

Introduce additional transitions for each letter connecting each final state of the first FA with the states of second FA that are connected with the initial state of second FA corresponding to each letter of the alphabet. Remove the +ve sign of each of final states of first FA and -ve sign of the initial state of second FA. It will create non-determinism at final states of first FA and hence NFA, thus obtained, will be the required NFA.

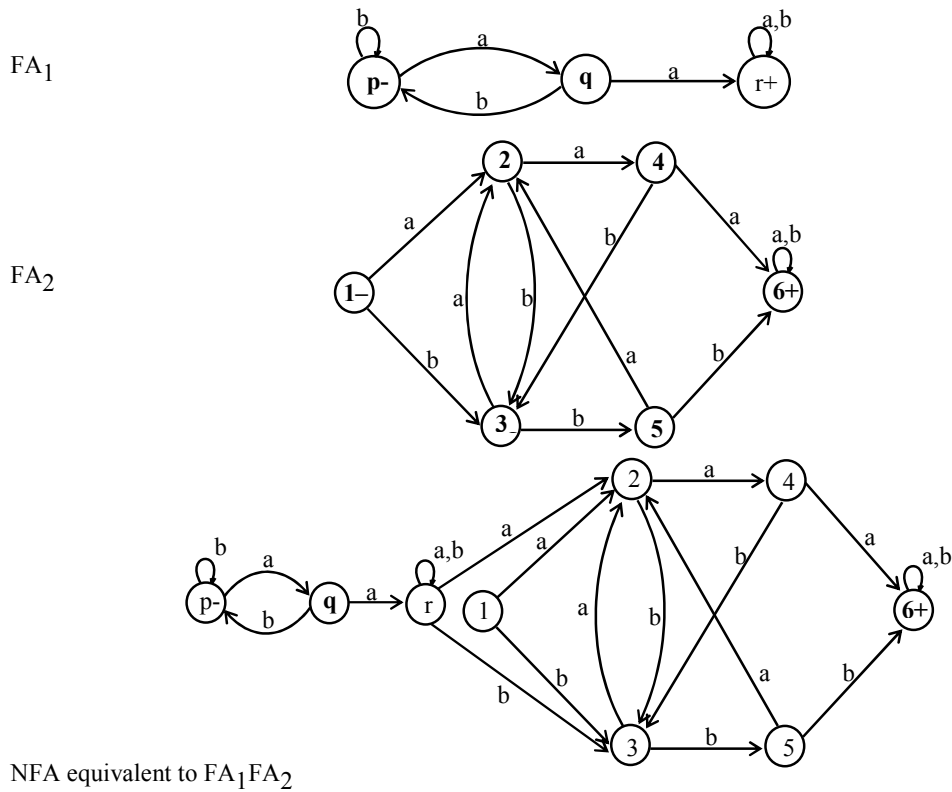
Note

It may be noted that if first FA accepts the Null string then every string accepted by second FA must be accepted by the concatenation of FAs as well. This situation will automatically be accommodated using the method discussed earlier. However if the second FA accepts Null string, then every string accepted by first FA must be accepted by the required FA as well. This target can be achieved as, while introducing new transitions at final states of first FA the +ve sign of these states will not be removed.

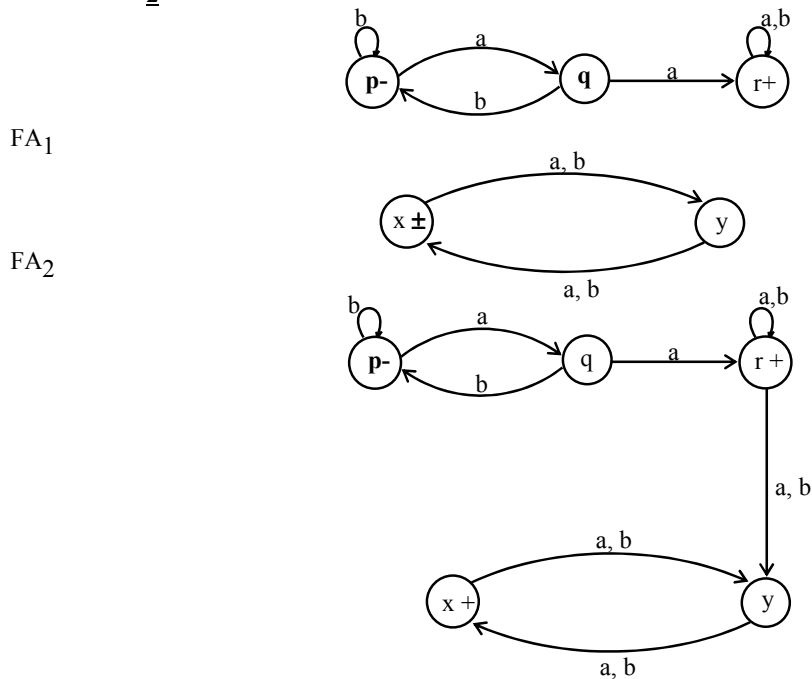


Lastly if both FAs accept the Null string, then the Null string must be accepted by the required FA. This situation will automatically be accommodated as the second FA accepts the Null string and hence the +ve signs of final states of first FA will not be removed.

Example (No FA accepts Null string)



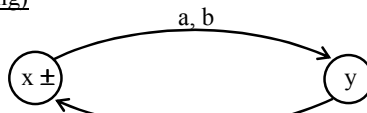
Example (FA<sub>2</sub> accepts Null string)



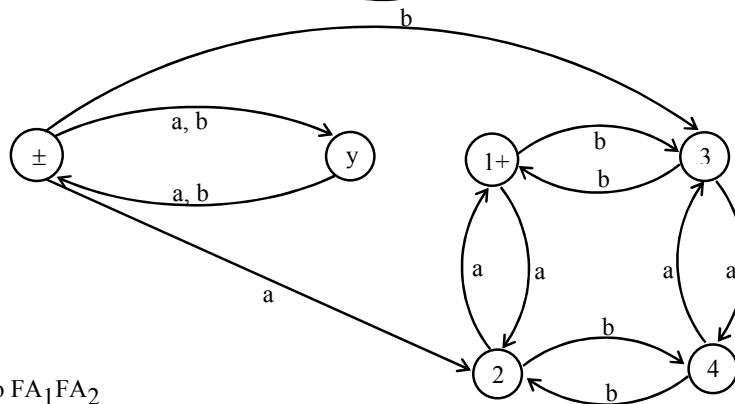
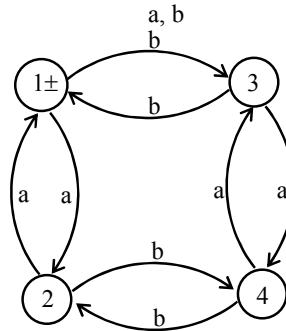
NFA equivalent to  $FA_1 FA_2$

Example (Both FAs accept Null string)

$FA_1$



$FA_2$



NFA equivalent to  $FA_1 FA_2$

#### NFA corresponding to the Closure of an FA

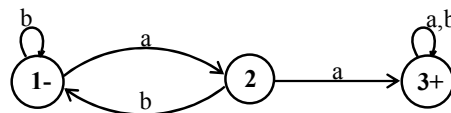
Apparently, it seems that since closure of an FA accepts the Null string, so the required NFA may be obtained considering the initial state of given FA to be final as well, but this may allow the unwanted string to be accepted as well. For example, an FA, with two states, accepting the language of strings, defined over  $\Sigma = \{a, b\}$ , **ending in a**, will accept all unwanted strings, if the initial state is supposed to be final as well.

#### Method

Thus, to accommodate this situation, introduce an initial state which should be final as well (so that the Null string is accepted) and connect it with the states originally connected with the old start state with the same transitions as the old start state, then remove the -ve sign of old start state. Introduce new transitions, for each letter, at each of the final states (including new final state) with those connected with the old start state. This creates non-determinism and hence results in the required NFA.

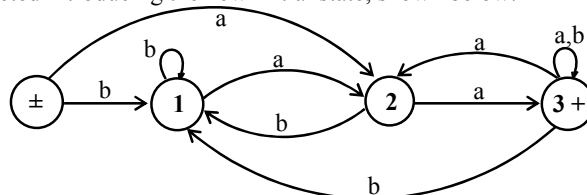
#### Example

Consider the following FA



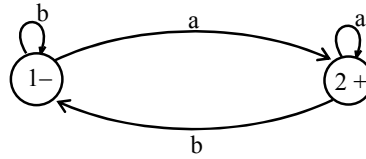
It may be observed that the  $FA^*$  accepts only the additional string which is the Null string.

Considering the state 1 to be final as well, will allow the unwanted strings be accepted as well. Hence the required NFA is constructed introducing the new initial state, shown below.



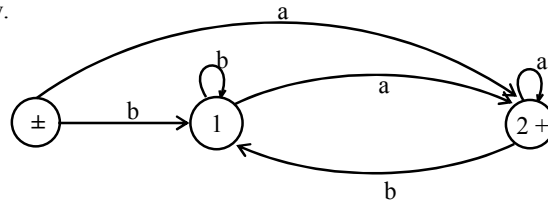
Example

Consider the following FA



It may be observed that the FA\* accepts only the additional string which is the Null string

As observed in the previous example the required NFA can be constructed only if the new initial state is introduced as shown below.



## Theory of Automata

**Lecture N0. 19****Reading Material**Introduction to Computer Theory

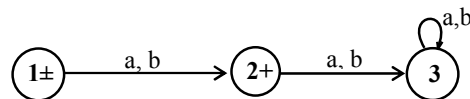
## Chapter 7

**Summary**

NFA corresponding to Closure of FA, Examples, Memory required to recognize a language, Example, Distinguishing one string from another, Example, Theorem, Proof

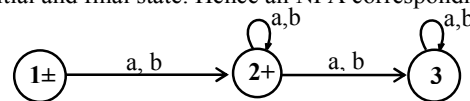
Example

Consider the following FA

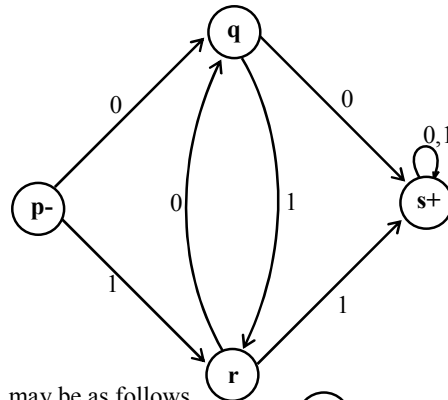


It can be observed that FA\* not only accepts the Null string but every other string as well.

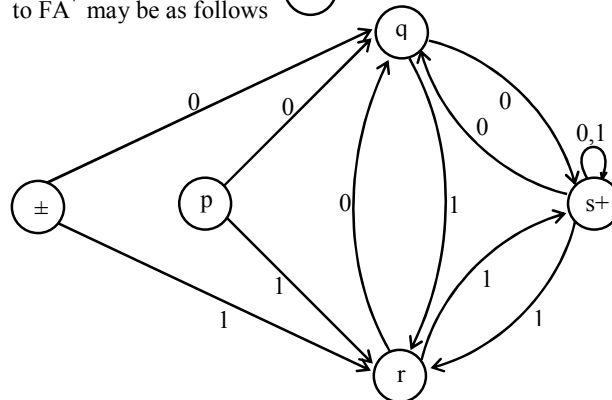
Here we don't need separate initial and final state. Hence an NFA corresponding to FA\* may be

Example

Consider the following FA



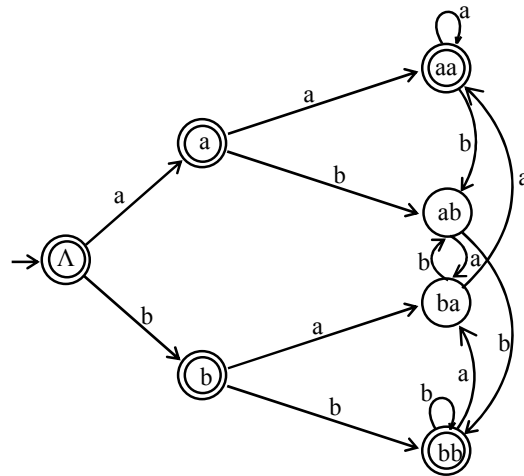
The NFA corresponding to FA\* may be as follows

**Memory required to recognize a language**

Memory required to recognize a language means to look at the machine which can recognize a language. As an FA can be considered to be a machine which is simple model of computation and every regular language is associated with certain FA, so to recognize a language there is a restriction that there is a single pass from left to right for any string to decide whether it belongs to certain language? This helps to remember the information about the initial part of the string read so far.

By this process the input string is examined and the string is decided either to be in a certain language or not.

Consider  $L = \{w \in \{a,b\}^* : w \text{ neither ends in } \mathbf{ab} \text{ nor in } \mathbf{ba}\}$ . i.e. L is the language of strings, defined over  $\Sigma = \{a,b\}$ , consisting of  $\Lambda$ , a, b and strings ending in aa or bb. L may be accepted by the following FA



As seen in the above FA, seven states are required to recognize the language  $L$ , while on the other hand it is very hard to recognize the language PALINDROME.

As seen in the above example of FA, seven states are required to recognize that language. Now consider another language  $L_3$  of strings of length three or more, defined over  $\Sigma = \{a,b\}$ , **and the third letter from the right is a**. As discussed by Martin, there is a straight forward method to build an FA recognizing  $L_3$  i.e. a distinct state for every possible substring of length less than or equal to 3. It is obvious that for each length  $i$ ,  $i=0,1,2,3$ , of substring, the number of states are  $2^i$  and thus total number of states required to recognize the language  $L_3$  are  $2^0+2^1+2^2+2^3 = 2^{3+1}-1=15$  (using  $2^0+2^1+2^2+\dots+2^n = 2^{n+1}-1$ )

**Remark:** Let  $L_{20}$  be the language of strings of length 20 or more, defined over  $\Sigma = \{a,b\}$ , **and the 20<sup>th</sup> letter from the right is 1**, then following the previous method, number of states for the corresponding FA is  $2^{20+1}-1=2,097,151$ .

However, it may be noted that any portion of memory of a computer that can accommodate 21 bits can be in  $2^{21}$  possible states i.e.  $2^{21}$  possible choices for the informational content.

### Distinguishable strings and Indistinguishable strings

Two strings  $x$  and  $y$ , belonging to  $\Sigma^*$ , are said to be **distinguishable** w.r.t a language  $L \subseteq \Sigma^*$  if there exists a string  $z$  belonging to  $\Sigma^*$  s.t.  $xz \in L$  but  $yz \notin L$  or  $xz \notin L$  but  $yz \in L$ .

Two strings  $x$  and  $y$ , belonging to  $\Sigma^*$ , are said to be **indistinguishable** with respect to a language  $L \subseteq \Sigma^*$  if for every string  $z$  belonging to  $\Sigma^*$ , either both  $xz$  or  $yz \in L$  or both don't belong to  $L$ .

### Example

Let  $L$  be the language of strings, defined over  $\Sigma = \{0,1\}$ , **ending in 01**.

The strings 110 and 010011 are **distinguishable** w.r.t  $L$ , as there exists 1 belonging to  $\Sigma^*$  s.t. 1101 belongs to  $L$  but 0100111 doesn't belong to  $L$ .

But 111 and 010011 are **indistinguishable**, for 1 belonging to  $\Sigma^*$  s.t. both 1111 and 010011 don't belong to  $L$  i.e. for every  $z$  belonging to  $\Sigma^*$ , either both 111z and 01001z belong to  $L$ , or both don't belong to  $L$ .

### Theorem

#### Statement

If  $L$  is a language over an alphabet  $\Sigma$  and for integer  $n$  there are  $n$  strings from  $\Sigma^*$ , any two of which are distinguishable w.r.t. language  $L$ , then any FA recognizes  $L$  must have at least  $n$  states.

(Note: There may not exist any FA which recognizes the given language.)

### Proof

Let  $S$  be set of strings, any two of which are distinguishable w.r.t. language  $L$ . Let  $F_1$  be the FA which recognizes the language  $L$ . To prove the theorem, it is sufficient to show that any two strings under  $F_1$  must be ended in different states i.e. corresponding to each string  $x$  belonging to  $S$ ,  $F_1$  ends in distinct states.

Thus if  $S$  has  $n$  strings then it is to be shown that  $F_1$  has at least  $n$  states.

Let  $x$  and  $y$  be any two strings from  $S$ . By supposition any two strings of  $S$  are distinguishable w.r.t.  $L$ , so there exists a string  $z$  belonging to  $\Sigma^*$  such that only one of  $xz$  and  $yz$  belongs to  $L$  *i.e.*  $F_1$  ends in a final state either for  $xz$  or  $yz$  which shows that  $F_1$  ends in distinct states for  $xz$  and  $yz$ .

Let  $F_1$  be ended in same state for both the strings  $x$  and  $y$ , which shows that  $F_1$  ends in same state for both  $xz$  and  $yz$ , a contradiction as  $x$  and  $y$  being distinguishable implies  $xz$  and  $yz$  are ended at distinct states of  $F_1$ .

Hence  $F_1$  does not end in a same state for both strings  $x$  and  $y$ , which shows that each pair of strings belonging to  $S$  ends in different states. Hence  $F_1$  must contain at least  $n$  states.

## Theory of Automata

**Lecture N0. 20****Reading Material**Introduction to Computer Theory

## Chapter 8

**Summary**

Example of previous Theorem, Finite Automaton with output, Moore machine, Examples

**Example**

Let  $L_{20} = \{w \in \{0,1\}^* : |w| \geq 20 \text{ and the } 20^{\text{th}} \text{ letter of } w, \text{ from right is, } 1\}$ . Let  $S$  be the set of all strings of length 20, defined over  $\Sigma$ , any two of which are distinguishable w.r.t.  $L_{20}$ . Obviously the number of strings belonging to  $S$ , is  $2^{20}$ . Let  $x$  and  $y$  be any two distinct strings i.e. they differ in  $i^{\text{th}}$  letter,  $i=1,2,3,\dots,20$ , from left. For  $i=1$ , they differ by first letter from left.

Then by definition of  $L_{20}$ , one is in  $L_{20}$  while other is not as shown below

0	.	...	.
1	.	...	.

So they are distinct w.r.t.  $L_{20}$  for  $z = \Lambda$  i.e. one of  $xz$  and  $yz$  belongs to  $L_{20}$ .

Similarly if  $i=2$  they differ by  $2^{\text{nd}}$  letter from left and are again distinguishable and hence for  $z$  belonging to  $\Sigma^*$ ,  $|z|=1$ , either  $xz$  or  $yz$  belongs to  $L_{20}$  because in this case the  $20^{\text{th}}$  letter from the right of  $xz$  and  $yz$  is exactly the  $2^{\text{nd}}$  letter from left of  $x$  and  $y$  as shown below

.	0	...	.	$z$
.	1	...	.	$z$

Hence  $x$  and  $y$  will be distinguishable w.r.t.  $L_{20}$  for  $i=2$ , as well. Continuing the process it can be shown that any pair of strings  $x$  and  $y$  belonging to  $S$ , will be distinguishable w.r.t.  $L_{20}$ . Since  $S$  contains  $2^{20}$  strings, any two of which are distinguishable w.r.t.  $L_{20}$ , so using the theorem any FA accepting  $L_{20}$  must have at least  $2^{20}$  states.

**Note**

It may be observed from the above example that using Martin's method, there exists an FA having  $2^{20+1}-1=2,097,151$  states. This indicates the memory required to recognize  $L_{20}$  will be the memory of a computer that can accommodate 21-bits i.e. the computer can be in  $2^{21}$  possible states.

**Finite Automaton with output**

Finite automaton discussed so far, is just associated with the RE or the language.

There is a question whether does there exist an FA which generates an output string corresponding to each input string? The answer is yes. Such machines are called machines with output.

There are two types of machines with output. Moore machine and Mealy machine

**Moore machine**

A Moore machine consists of the following

A finite set of states  $q_0, q_1, q_2, \dots$  where  $q_0$  is the initial state.

An alphabet of letters  $\Sigma = \{a,b,c,\dots\}$  from which the input strings are formed.

An alphabet  $\Gamma = \{x,y,z,\dots\}$  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.

An output table that shows what character is printed by each state as it is entered.

**Note**

It is to be noted that since in Moore machine no state is designated to be a final state, so there is no question of accepting any language by Moore machine. However in some cases the relation between an input string and the corresponding output string may be identified by the Moore machine. Moreover, the state to be initial is not important as if the machine is used several times and is restarted after some time, the machine will be started from the state where it was left off. Following are the examples

Example

Consider the following Moore machine having the states  $q_0, q_1, q_2, q_3$  where  $q_0$  is the start state and

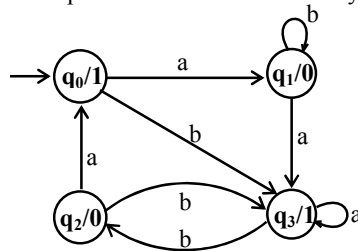
$\Sigma = \{a, b\}$ ,

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

the transition table follows as

Old States	New States after reading		Characters to be printed
	a	b	
$q_0$	$q_1$	$q_3$	1
$q_1$	$q_3$	$q_1$	0
$q_2$	$q_0$	$q_3$	0
$q_3$	$q_3$	$q_2$	1

the transition diagram corresponding to the previous transition table may be



It is to be noted that the states are labeled along with the characters to be printed. Running the string abbabbba over the above machine, the corresponding output string will be 100010101, which can be determined by the following table as well

Input		a	b	b	a	b	b	b	a
State	$q_0$	$q_1$	$q_1$	$q_1$	$q_3$	$q_2$	$q_3$	$q_2$	$q_0$
output	1	0	0	0	1	0	1	0	1

It may be noted that the length of output string is 1 more than that of input string as the initial state prints out the extra character 1, before the input string is read.



## Theory of Automata

**Lecture N0. 21****Reading Material**Introduction to Computer Theory

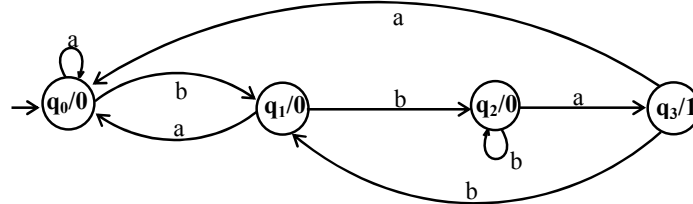
## Chapter 8

**Summary**

Example of Moore machine, Mealy machine, Examples, complementing machine, Incrementing machine.

**Example**

To identify the relation between the input strings and the corresponding output strings in the following Moore machine,



if the string bbbabaabbaa is run, the output string will be 000010000010, as shown below

Input		b	b	b	a	b	a	a	b	b	a	a
State	q <sub>0</sub>	q <sub>1</sub>	q <sub>2</sub>	q <sub>2</sub>	q <sub>3</sub>	q <sub>1</sub>	q <sub>0</sub>	q <sub>0</sub>	q <sub>1</sub>	q <sub>2</sub>	q <sub>3</sub>	q <sub>0</sub>
output	0	0	0	0	1	0	0	0	0	0	1	0

It can be observed from the given Moore machine that q<sub>3</sub> is the only state which prints out the character 1 which shows that the moment the state q<sub>3</sub> is entered, the machine will print out 1. To enter the state q<sub>3</sub>, starting from q<sub>0</sub> the string must contain bba. It can also be observed that to enter the state q<sub>3</sub> once more the string must contain another substring bba. In general the input string will visit the state q<sub>3</sub> as many times as the number of substring bba occurs in the input string. Thus the number of 1's in an output string will be same as the number of substring bba occurs in the corresponding input string.

**Mealy machine**

A Mealy machine consists of the following

A finite set of states q<sub>0</sub>, q<sub>1</sub>, q<sub>2</sub>, ... where q<sub>0</sub> is the initial state.

An alphabet of letters  $\Sigma = \{a, b, c, \dots\}$  from which the input strings are formed.

An alphabet  $\Gamma = \{x, y, z, \dots\}$  of output characters from which output strings are generated.

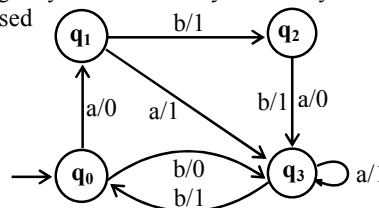
A pictorial representation with states and directed edges labeled by an input letter along with an output character. The directed edges also show how to go from one state to another corresponding to every possible input letter.

(It is not possible to give transition table in this case.)

**Note**

It is to be noted that since, similar to Moore machine, in Mealy machine no state is designated to be a final state, so there is no question of accepting any language by Mealy machine. However in some cases the relation between an input string and the corresponding output string may be identified by the Mealy machine. Moreover, the state to be initial is not important as if the machine is used several times and is restarted after some time, the machine will be started from the state where it was left

off. Following are the examples

**Example**

Consider the Mealy machine shown aside, having the states q<sub>0</sub>, q<sub>1</sub>, q<sub>2</sub>, q<sub>3</sub>, where q<sub>0</sub> is the start state and

$$\Sigma = \{a, b\},$$

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

Running the string abbabbba over the above machine, the corresponding output string will be 01111010, which can be determined by the following table as well

Input		a	b	b	a	b	b	b	a
States	$q_0$	$q_1$	$q_2$	$q_3$	$q_3$	$q_0$	$q_3$	$q_0$	$q_1$
output		0	1	1	1	1	0	1	0

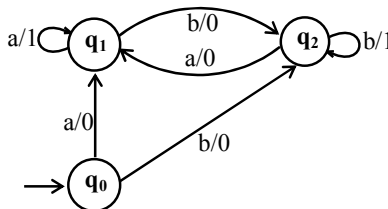
It may be noted that in Mealy machine, the length of output string is equal to that of input string.

#### Example

Consider the following Mealy machine having the states  $q_0, q_1, q_2$ , where  $q_0$  is the start state and

$$\Sigma = \{a, b\},$$

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



It is observed that in the above Mealy machine, if in the output string the  $n$ th character is 1, it shows that the  $n$ th letter in the input string is the second in the pair of double letter.

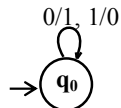
For babaababba as input string the machine will print 0000100010.

#### Example

Consider the following Mealy machine having the only state  $q_0$  as the start state and

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

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



If 0011010 is run on this machine then the corresponding output string will be 1100101.

This machine is called **Complementing machine**.

#### Constructing the incrementing machine

In the previous example of complementing machine, it has been observed that the input string and the corresponding output string are 1's complement of each other. There is a question whether the Mealy machine can be constructed, so that the output string is increased, in magnitude, by 1 than the corresponding input string? The answer is yes.

This machine is called the incrementing machine. Following is how to construct the incrementing machine.

Before the incrementing machine is constructed, consider how 1 is added to a binary number.

Since, if two numbers are added, the addition is performed from right to left, so while increasing the binary number by 1, the string (binary number) must be read by the corresponding Mealy machine from right to left, and hence the output string (binary number) will also be generated from right to left.

Consider the following additions

a) $\begin{array}{r} 100101110 \\ + 1 \\ \hline 100101111 \end{array}$	b) $\begin{array}{r} 1001100111 \\ + 1 \\ \hline 1001101000 \end{array}$
--	--

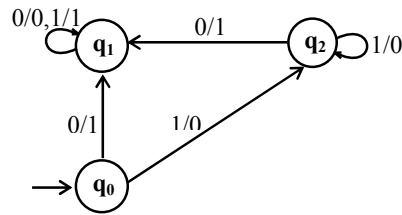
It may be observed from the above that

If the right most bit of binary number, to be incremented, is 0, the output binary number can be obtained by converting the right most bit to 1 and remaining bits unchanged.

If the right most bit of binary number is 1 then the output can be obtained, converting that 1 along with all its concatenated 1's to 0's, then converting the next 0 to 1 and remaining bits unchanged.

The observations (a) and (b) help to construct the following Incrementing (Mealy) machine.

The Mealy machine have the states  $q_0, q_1, q_2$ , where  $q_0$  is the start state and

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


It may be observed that, in the incrementing machine, if 0 is read at initial state  $q_0$ , that 0 is converted to 1 and a no change state  $q_1$  (no carry state) is entered where all 0's and all 1's remain unchanged. If 1 is read at initial state, that 1 is converted to 0 and the state  $q_2$  (owe carry state) is entered, where all 1's are converted to 0's and at that state if 0 is read that 0 is converted to 1 and the machine goes to no change state.

If the strings 100101110 and 1001100111 are run over this machine, the corresponding output strings will be 100101111 and 1001101000 respectively.

#### Note

It is to be noted that if the string 111111 is run over the incrementing machine, the machine will print out 000000, which is not increased in magnitude by 1. Such a situation is called an overflow situation, as the length of output string will be same as that of input string.

It may also be noted that there exists another incrementing machine with two states.

## Theory of Automata

**Lecture N0. 22****Reading Material**Introduction to Computer Theory

## Chapter 8

**Summary**

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

**Applications of Incrementing and Complementing machines**

1's complementing and incrementing machines which are basically Mealy machines are very much helpful in computing.

The incrementing machine helps in building a machine that can perform the addition of binary numbers.

Using the complementing machine along with incrementing machine, one can build a machine that can perform the subtraction of binary numbers, as shown in the following method

**Subtracting a binary number from another**Method

To subtract a binary number  $b$  from a binary number  $a$

Add 1's complement of  $b$  to  $a$  (ignoring the overflow, if any)

Increase the result, in magnitude, by 1 (use the incrementing machine). Ignoring the overflow if any.

**Note:** If there is no overflow in (1). Take 1's complement once again in (2), instead. This situation occurs when  $b$  is greater than  $a$ , in magnitude. Following is an example of subtraction of binary numbers

Example

To subtract the binary number 101 from the binary number 1110, let

$a = 1110$  and  $b = 101 = 0101$ .

(Here the number of digits of  $b$  are equated with that of  $a$ )

Adding 1's complement (1010) of  $b$  to  $a$ .

$$\begin{array}{r} 1110 \\ +1010 \\ \hline 11000 \end{array}$$

11000 which gives 1000 (ignoring the overflow)

Using the incrementing machine, increase the above result 1000, in magnitude, by 1

$$\begin{array}{r} 1000 \\ +1 \\ \hline 1001 \end{array}$$

1001 which is the same as obtained by ordinary subtraction.

Note

It may be noted that the above method of subtraction of binary numbers may be applied to subtraction of decimal numbers with the change that 9's complement of  $b$  will be added to  $a$ , instead in step (1).

**Equivalent machines**

Two machines are said to be **equivalent** if they print the same output string when the same input string is run on them.

Remark:

Two Moore machines may be equivalent. Similarly two Mealy machines may also be equivalent, but a Moore machine can't be equivalent to any Mealy machine. However, ignoring the extra character printed by the Moore machine, there exists a Mealy machine which is equivalent to the Moore machine.

**Theorem**Statement

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

Proof:

Let  $M$  be a Moore machine, then shifting the output characters corresponding to each state to the labels of corresponding incoming transitions, machine thus obtained will be a Mealy machine equivalent to  $M$ .

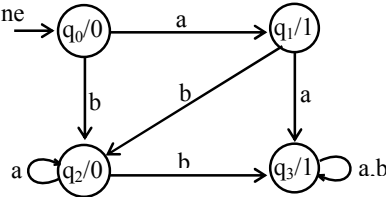
**Note**

It may be noted that while converting a Moore machine into an equivalent Mealy machine, the output character of a state will be ignored if there is no incoming transition at that state. A loop at a state is also supposed to be an incoming transition.

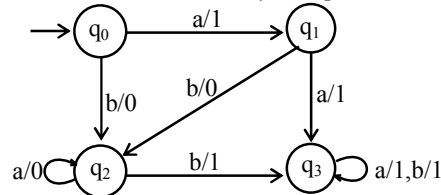
Following is the example of converting a Moore machine into an equivalent Mealy machine

**Example**

Consider the following Moore machine



Using the method described earlier, the above machine may be equivalent to the following Mealy machine



Running the string abbabbba on both the machines, the output string can be determined by the following table

Input		a	b	b	a	b	b	b	a
States	q <sub>0</sub>	q <sub>1</sub>	q <sub>2</sub>	q <sub>3</sub>	q <sub>3</sub>	q <sub>3</sub>	q <sub>3</sub>	q <sub>3</sub>	q <sub>3</sub>
Moore	0	1	0	1	1	1	1	1	1
Mealy		1	0	1	1	1	1	1	1

**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

The incoming transitions have the same output character.

The incoming transitions have different output characters.

If all the transitions have same output characters, then shift that character to the corresponding state.

If all the transitions have different output characters, then the state will be converted to as many states as the number of different output characters for these transitions, which shows that if this happens at state  $q_i$  then  $q_i$  will be converted to  $q_i^1$  and  $q_i^2$  i.e. if at  $q_i$  there are the transitions with two output characters then  $q_i^1$  for one character and  $q_i^2$  for other character.

Shift the output characters of the transitions to the corresponding new states  $q_i^1$  and  $q_i^2$ . Moreover, these new states  $q_i^1$  and  $q_i^2$  should behave like  $q_i$  as well. Continuing the process, the machine thus obtained, will be a Moore machine equivalent to Mealy machine M.

Following is a note

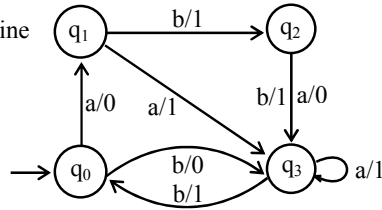
**Note**

It may be noted that if there is no incoming transition at certain state then any of the output characters may be associated with that state.

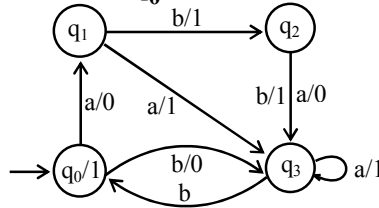
It may also be noted that if the initial state is converted into more than one new states then only one of these new states will be considered to be the initial state. Following is an example

Example

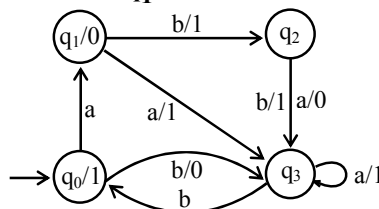
Consider the following Mealy machine



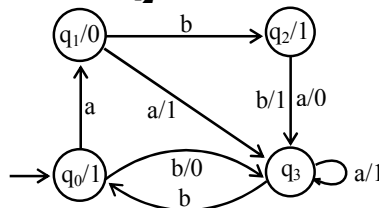
Shifting the output character 1 of transition b to **q<sub>0</sub>**



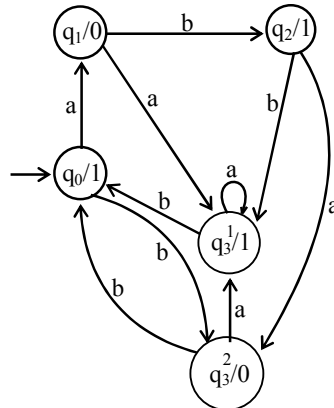
Shifting the output character 0 of transition a to **q<sub>1</sub>**



Shifting the output character 1 of transition b to **q<sub>2</sub>**



Splitting **q<sub>3</sub>** into **q<sub>3</sub><sup>1</sup>** and **q<sub>3</sub><sup>2</sup>**



Running the string abbabbba on both the machines, the output strings can be determined by the following table

Input		a	b	b	a	b	b	b	a
States	q <sub>0</sub>	q <sub>1</sub>	q <sub>2</sub>	q <sub>3</sub>	q <sub>3</sub>	q <sub>0</sub>	q <sub>3</sub>	q <sub>0</sub>	q <sub>1</sub>
Mealy		0	1	1	1	1	0	1	0
Moore	1	0	1	1	1	1	0	1	0

## Theory of Automata

**Lecture N0. 23****Reading Material**Introduction to Computer Theory

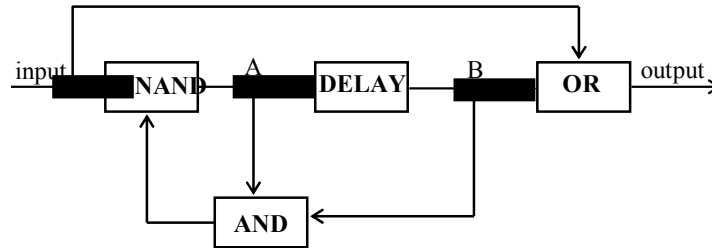
## Chapter 8

**Summary**

Mealy machines in terms of sequential circuit

**Example**

Consider the following sequential circuit



The following four types of boxes are used in this circuit

**NAND box (NOT AND):** For the given input, it provides the complement of Boolean AND output.

**DELAY box (Flip Flop box):** It delays the transmission of signal along the wire by one step (clock pulse).

**OR box:** For the given input, it provides the Boolean OR output.

**AND box:** For the given input, it provides the Boolean AND output.

The current in the wire is indicated by 1 and 0 indicates the absence of the current.

There are two points A and B w.r.t. to which following four states of the machine are identified according to the presence and absence of current at these points *i.e.*

$q_0 (A=0, B=0) \equiv (0,0)$

$q_1 (A=0, B=1) \equiv (0,1)$

$q_2 (A=1, B=0) \equiv (1,0)$

$q_3 (A=1, B=1) \equiv (1,1)$

The operation of the circuit is such that the machine changes its state after reading 0 or 1. The transitions are determined using the following relations

new B = old A

new A = (input) **NAND** (old A **AND** old B)

output = (input) **OR** (old B)

It is to be noted that old A and old B indicate the presence or absence of current at A and B before inputting any letter. Similarly new A and new B indicate the presence or absence of current after reading certain letter.

At various discrete pulses of a time clock, input is received by the machine and the corresponding output string is generated.

The transition at the state  $q_0$  after reading the letter 0, can be determined, along with the corresponding output character as under

new B = old A = 0

new A = (input) **NAND** (old A **AND** old B)  
 $= 0 \text{ NAND } (0 \text{ AND } 0) = 0 \text{ NAND } 0$   
 $= 1$

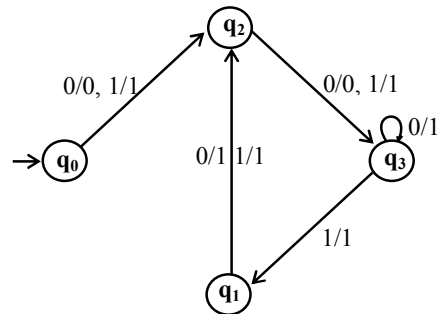
output = (input) **OR** (old B) = 0 **OR** 0 = 0

Thus after reading 0 at  $q_0$  new B is 0 and new A is 1 i.e. machine will be at state  $(1,0) \equiv q_2$  and during this process its output character will be 0.

The remaining action of this sequential circuit can be determined as shown by the following suggested transition table of the corresponding Mealy machine

Old state	Inputting 0		Inputting 1	
	State	Output	State	Output
$q_0 \equiv (0,0)$	$(1,0) \equiv q_2$	0	$(1,0) \equiv q_2$	1
$q_1 \equiv (0,1)$	$(1,0) \equiv q_2$	1	$(1,0) \equiv q_2$	1
$q_2 \equiv (1,0)$	$(1,1) \equiv q_3$	0	$(1,1) \equiv q_3$	1
$q_3 \equiv (1,1)$	$(1,1) \equiv q_3$	1	$(0,1) \equiv q_1$	1

The corresponding transition diagram may be as follows



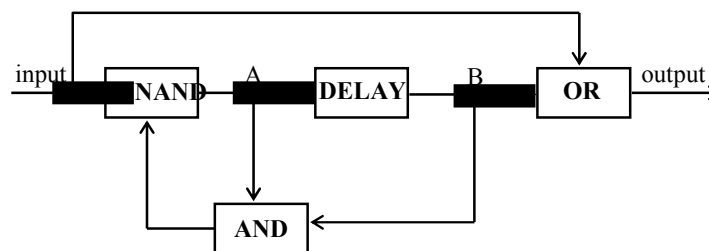
Note: It may be noted that if the string 00 is read at any state, it results in ending in state  $q_3$ .

Running the string 01101110 on the previous machine, the output string can be determined by the following table

Input		0	1	1	0	1	1	1	0
States	$q_0$	$q_2$	$q_3$	$q_1$	$q_2$	$q_3$	$q_1$	$q_2$	$q_3$
output		0	1	1	1	1	1	1	0

Following is a note regarding the sequential circuit under consideration

Note



It is to be noted that in this sequential circuit, delay box plays an important role in introducing four states of the machine.



## Theory of Automata

**Lecture N0. 24****Reading Material**Introduction to Computer Theory

## Chapter 9

**Summary**

Regular languages, complement of a language, theorem, proof, example, intersection of two regular languages

**Regular languages**

As already been discussed earlier that any language that can be expressed by a RE is said to be regular language, so if  $L_1$  and  $L_2$  are regular languages then  $L_1 + L_2$ ,  $L_1L_2$  and  $L_1^*$  are also regular languages. This fact can be proved by the following two methods

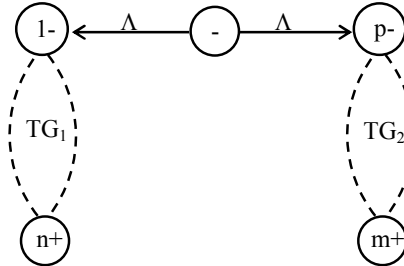
**By Regular Expressions**

As discussed earlier that if  $r_1, r_2$  are regular expressions, corresponding to the languages  $L_1$  and  $L_2$  then the languages  $L_1 + L_2$ ,  $L_1L_2$  and  $L_1^*$  generated by  $r_1 + r_2$ ,  $r_1r_2$  and  $r_1^*$ , are also regular languages.

**By TGs**

If  $L_1$  and  $L_2$  are regular languages then  $L_1$  and  $L_2$  can also be expressed by some REs as well and hence using Kleene's theorem,  $L_1$  and  $L_2$  can also be expressed by some TGs. Following are the methods showing that there exist TGs corresponding to  $L_1 + L_2$ ,  $L_1L_2$  and  $L_1^*$

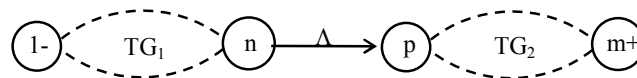
If  $L_1$  and  $L_2$  are expressed by  $TG_1$  and  $TG_2$  then following may be a TG accepting  $L_1 + L_2$



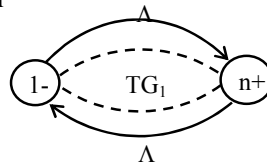
If  $L_1$  and  $L_2$  are expressed by the following  $TG_1$  and  $TG_2$



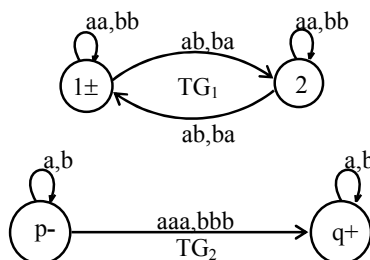
then following may be a TG accepting  $L_1L_2$



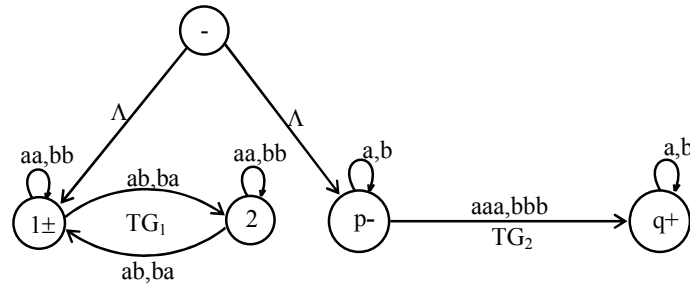
also a TG accepting  $L_1^*$  may be as under

**Example**

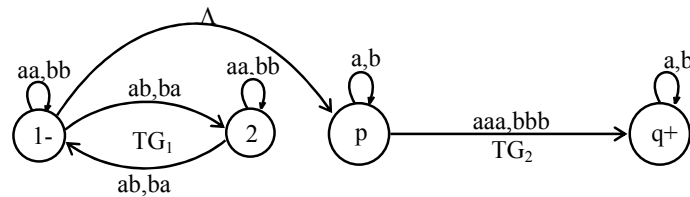
Consider the following TGs



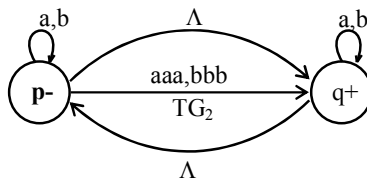
Following may be a TG accepting  $L_1 + L_2$



also a TG accepting  $L_1 L_2$  may be



and a TG accepting  $L_2^*$



### Complement of a language

Let  $L$  be a language defined over an alphabet  $\Sigma$ , then the language of strings, defined over  $\Sigma$ , **not belonging to**  $L$ , is called **Complement of the language  $L$** , denoted by  $L^c$  or  $L^c$ .

### Note

To describe the complement of a language, it is very important to describe the alphabet of that language over which the language is defined.

For a certain language  $L$ , the complement of  $L^c$  is the given language  $L$  i.e.  $(L^c)^c = L$

### Theorem

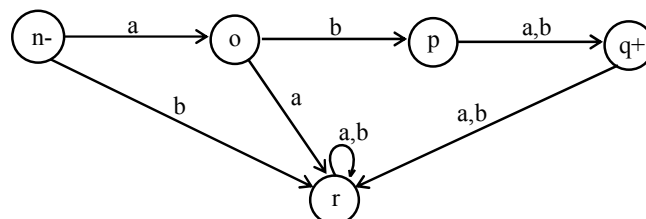
If  $L$  is a regular language then,  $L^c$  is also a regular language.

### Proof

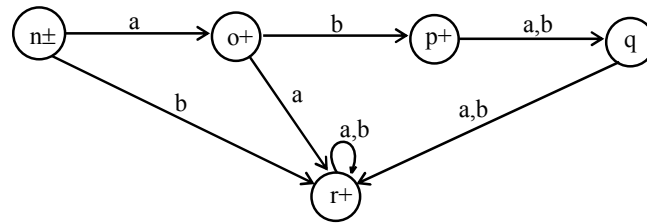
Since  $L$  is a regular language, so by Kleene's theorem, there exists an FA, say  $F$ , accepting the language  $L$ . Converting each of the final states of  $F$  to non-final states and old non-final states of  $F$  to final states, FA thus obtained will reject every string belonging to  $L$  and will accept every string, defined over  $\Sigma$ , not belonging to  $L$ . Which shows that the new FA accepts the language  $L^c$ . Hence using Kleene's theorem  $L^c$  can be expressed by some RE. Thus  $L^c$  is regular.

### Example

Let  $L$  be the language over the alphabet  $\Sigma = \{a, b\}$ , consisting of only two words  $aba$  and  $abb$ , then the FA accepting  $L$  may be



Converting final states to non-final states and old non-final states to final states, then FA accepting  $L^c$  may be



### **Theorem**

#### **Statement**

If  $L_1$  and  $L_2$  are two regular languages, then  $L_1 \cap L_2$  is also regular.

#### **Proof**

Using De-Morgan's law for sets

$$(L_1^c \cup L_2^c)^c = (L_1^c)^c \cap (L_2^c)^c = L_1 \cap L_2$$

Since  $L_1$  and  $L_2$  are regular languages, so are  $L_1^c$  and  $L_2^c$ .  $L_1^c$  and  $L_2^c$  being regular provide that  $L_1^c \cup L_2^c$  is also regular language and so  $(L_1^c \cup L_2^c)^c = L_1 \cap L_2$ , being complement of regular language is regular language.

## Theory of Automata

**Lecture N0. 25****Reading Material**Introduction to Computer Theory

## Chapter 9,10

**Summary**

Intersection of two regular languages, examples, non regular language, example

**Theorem****Statement**

If  $L_1$  and  $L_2$  are two regular languages, then  $L_1 \cap L_2$  is also regular.

**Proof**

Using De-Morgan's law for sets

$$(L_1^c \cup L_2^c)^c = (L_1^c)^c \cap (L_2^c)^c = L_1 \cap L_2$$

Since  $L_1$  and  $L_2$  are regular languages, so are  $L_1^c$  and  $L_2^c$ .  $L_1^c$  and  $L_2^c$  being regular provide that  $L_1^c \cup L_2^c$  is also regular language and so  $(L_1^c \cup L_2^c)^c = L_1 \cap L_2$ , being complement of regular language is regular language.

Following is a remark

**Remark**

If  $L_1$  and  $L_2$  are regular languages, then these can be expressed by the corresponding FAs. Finding regular expressions defining the language  $L_1 \cap L_2$  is not so easy and building corresponding FA is rather harder. Following are example of finding an FA accepting the intersection of two regular languages

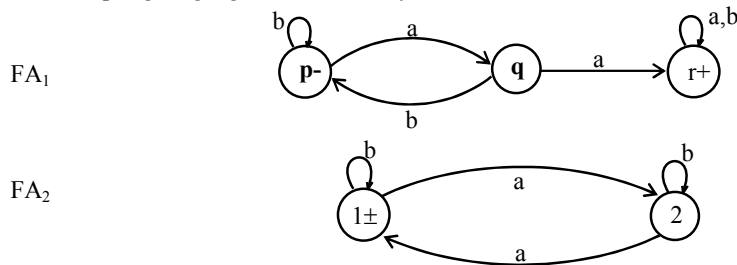
**Example**

Consider two regular languages  $L_1$  and  $L_2$ , defined over the alphabet  $\Sigma = \{a, b\}$ , where

$L_1$  = language of words with **double a's**.

$L_2$  = language of words containing **even number of a's**.

FAs accepting languages  $L_1$  and  $L_2$  may be as follows

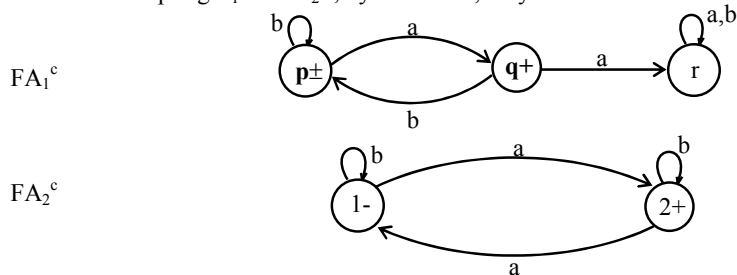


Their corresponding REs may be

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

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

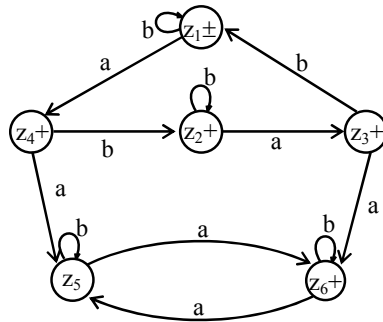
Now FAs accepting  $L_1^c$  and  $L_2^c$ , by definition, may be



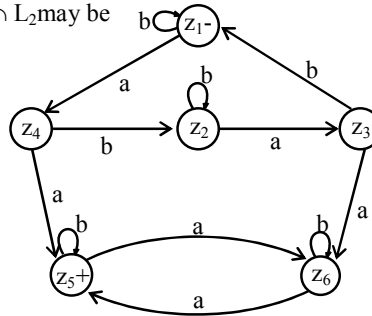
Now FA accepting  $L_1^c \cup L_2^c$ , using the method described earlier, may be as follows

Old States	New States after reading	
	a	b
$z_1 \pm \equiv (p, 1)$	$(q, 2) \equiv z_4$	$(p, 1) \equiv z_1$
$z_2 + \equiv (p, 2)$	$(q, 1) \equiv z_3$	$(p, 2) \equiv z_2$
$z_3 + \equiv (q, 1)$	$(r, 2) \equiv z_6$	$(p, 1) \equiv z_1$
$z_4 + \equiv (q, 2)$	$(r, 1) \equiv z_5$	$(p, 2) \equiv z_2$
$z_5 \equiv (r, 1)$	$(r, 2) \equiv z_6$	$(r, 1) \equiv z_5$
$z_6 + \equiv (r, 2)$	$(r, 1) \equiv z_5$	$(r, 2) \equiv z_6$

Here all the possible combinations of states of  $FA_1^c$  and  $FA_2^c$  are considered

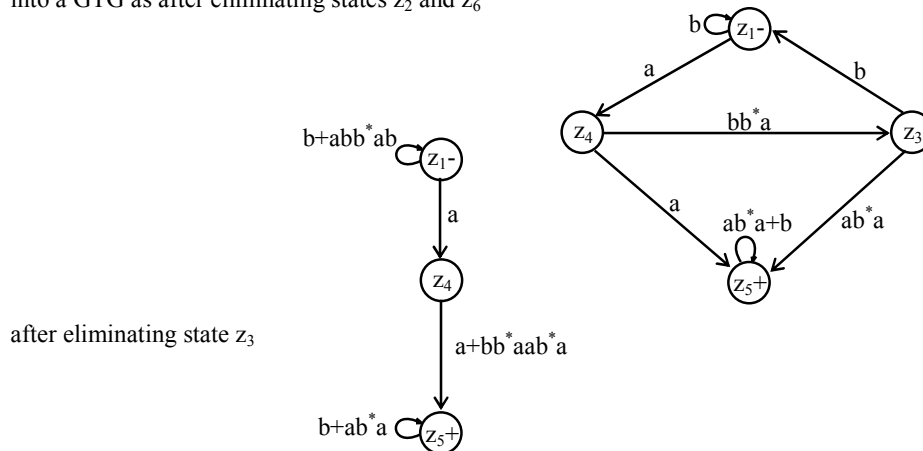


An FA that accepts the language  $(L_1^c \cup L_2^c)^c = L_1 \cap L_2$  may be

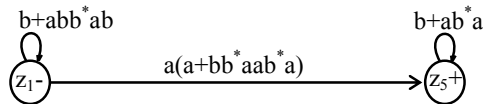


Corresponding RE can be determined as follows

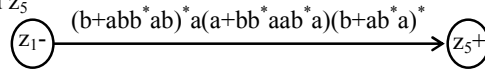
The regular expression defining the language  $L_1 \cap L_2$  can be obtained, converting and reducing the previous FA into a GTG as after eliminating states  $z_2$  and  $z_6$



$z_4$  can obviously be eliminated as follows



eliminating the loops at  $z_1$  and  $z_5$



Thus the required RE may be  $(b+abb^*ab)^*a(a+bb^*aab^*a)(b+ab^*a)^*$

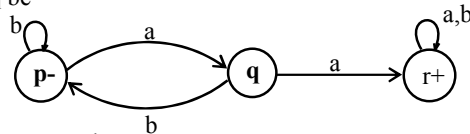
### **FA corresponding to intersection of two regular languages (short method)**

Let  $FA_3$  be an FA accepting  $L_1 \cap L_2$ , then the initial state of  $FA_3$  must correspond to the initial state of  $FA_1$  and the initial state of  $FA_2$ .

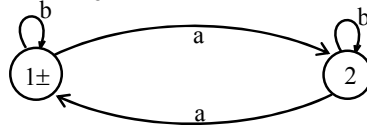
Since the language corresponding to  $L_1 \cap L_2$  is the intersection of corresponding languages  $L_1$  and  $L_2$ , consists of the strings belonging to both  $L_1$  and  $L_2$ , therefore a final state of  $FA_3$  must correspond to a final state of  $FA_1$  and  $FA_2$ . Following is an example regarding short method of finding an FA corresponding to the intersection of two regular languages.

#### Example

Let  $r_1 = (a+b)^*aa(a+b)^*$  and  $FA_1$  be



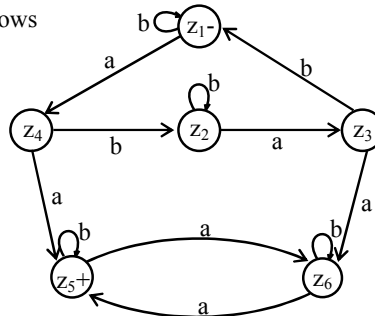
also  $r_2 = (b+ab^*a)^*$  and  $FA_2$  be



An FA corresponding to  $L_1 \cap L_2$  can be determined as follows

Old States	New States after reading	
	a	b
$z_1 \equiv (p, 1)$	$(q, 2) \equiv z_4$	$(p, 1) \equiv z_1$
$z_2 \equiv (p, 2)$	$(q, 1) \equiv z_3$	$(p, 2) \equiv z_2$
$z_3 \equiv (q, 1)$	$(r, 2) \equiv z_6$	$(p, 1) \equiv z_1$
$z_4 \equiv (q, 2)$	$(r, 1) \equiv z_5$	$(p, 2) \equiv z_2$
$z_5 \equiv (r, 1)$	$(r, 2) \equiv z_6$	$(r, 1) \equiv z_5$
$z_6 \equiv (r, 2)$	$(r, 1) \equiv z_5$	$(r, 2) \equiv z_6$

The corresponding transition diagram may be as follows



**Nonregular languages**

The language that cannot be expressed by any regular expression is called a **Nonregular language**.

The languages **PALINDROME** and **PRIME** are the examples of nonregular languages.

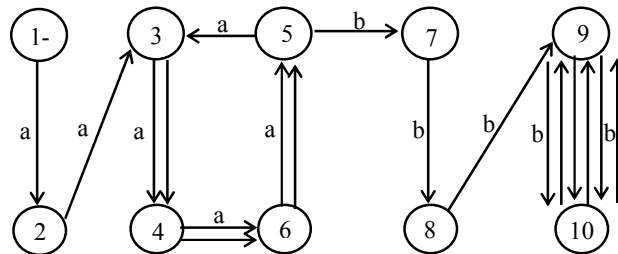
*Note:* It is to be noted that a nonregular language, by Kleene's theorem, can't be accepted by any FA or TG.

**Example**

Consider the language  $L = \{\Lambda, ab, aabb, aaabbb, \dots\}$  i.e.  $\{a^n b^n : n=0,1,2,3,\dots\}$

Suppose, it is required to prove that this language is nonregular. Let, contrary,  $L$  be a regular language then by Kleene's theorem it must be accepted by an FA, say,  $F$ . Since every FA has finite number of states then the language  $L$  (being infinite) accepted by  $F$  must have words of length more than the number of states. Which shows that,  $F$  must contain a circuit.

For the sake of convenience suppose that  $F$  has 10 states. Consider the word  $a^9 b^9$  from the language  $L$  and let the path traced by this word be shown as under



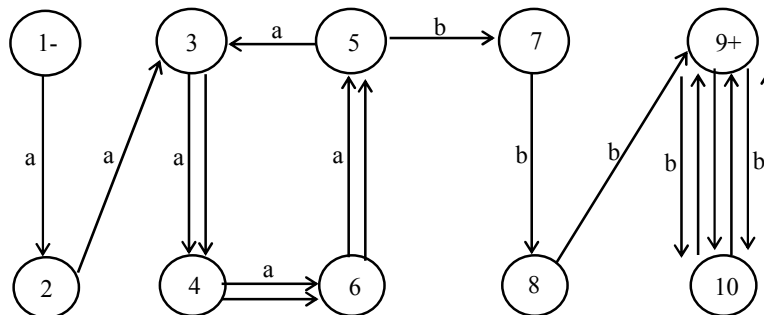
But, looping the circuit generated by the states 3,4,6,5,3 with  $a$ -edges once more,  $F$  also accepts the word  $a^{9+4} b^9$ , while  $a^{13} b^9$  is not a word in  $L$ . It may also be observed that, because of the circuit discussed above,  $F$  also accepts the words  $a^9 (a^4)^m b^9$ ,  $m = 1, 2, 3, \dots$

Moreover, there is another circuit generated by the states 9,10,9. Including the possibility of looping this circuit,  $F$  accepts the words  $a^9 (a^4)^m b^9 (b^2)^n$  where  $m, n=0, 1, 2, 3, \dots$  ( $m$  and  $n$  not being 0 simultaneously). Which shows that  $F$  accepts words that are not belonging to  $L$ .

### Example of nonregular language, pumping lemma version I, proof, examples

Consider the language  $L = \{\Lambda, ab, aabb, aaabbb, \dots\}$  i.e.  $\{a^n b^n : n=0,1,2,3,\dots\}$

For the sake of convenience suppose that  $F$  has 10 states. Consider the word  $a^9b^9$  from the language  $L$  and let the path traced by this word be shown as under



Similarly for finding FAs accepting other words from  $L$ , they will also accept the words which do not belong to  $L$ .

## Pumping Lemma

Let  $L$  be any infinite regular language (that has infinite many words), defined over an alphabet  $\Sigma$  then there exist three strings  $x$ ,  $y$  and  $z$  belonging to  $\Sigma^*$  (where  $y$  is not the null string) such that all the strings of the form  $xy^n z$  for  $n=1,2,3, \dots$  are the words in  $L$ .

Let  $w$  be a word in the language  $L$ , so that the length of word is greater than the number of states in  $F$ . In this case the path generated by the word  $w$ , is such that it cannot visit a new state for each letter *i.e.* there is a circuit in this path.

The substring which generates the path from initial state to the state which is revisited first while reading the word  $w$ . This part can be called  $x$  and  $x$  can be a null string.



The substring which generates the circuit starting from the state which was lead by x. This part can be called as y which cannot be null string.

The substring which is the remaining part of the word after y, call this part as z. It may be noted that this part may be null string as the word may end after y or z part may itself be a circuit.

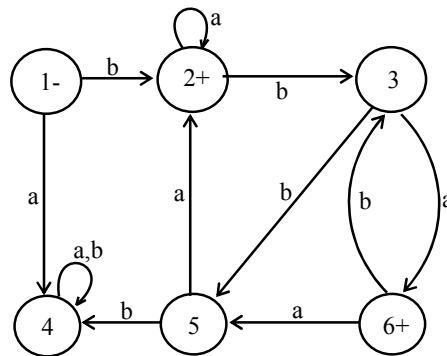
Thus the word may be written as  $w = xyz$  where x,y and z are the strings, also y can't be a null string.

Now this is obvious that, looping the circuit successively, the words  $xyyz, xyxyz, xyxyz, \dots$  will also be accepted by this FA i.e.  $xy^n z, n=1,2,3, \dots$  will be words in L.

**Remark:** In the above theorem, it is not affected if the z-part has circuit. To prove the theorem it is only to find a circuit and then looping that circuit, is all that is needed. While looping the circuit the volume of the string y (or z) is pumped, so the theorem is also called the Pumping lemma. Following are the examples

### Example

Consider the following 5 states FA, say, F which accepts an infinite language



Let the word  $w = bbbababa$ , belonging to the language L, so that the length of word is greater than 6 (the number of states in F).

In this case the path generated by this word is such that it cannot visit a new state for each letter i.e. there is a circuit in this path.

The word w, in this case, may be divided into three parts.

The substring which generates the path from initial state to the state which is revisited first while reading the word w. This can be called as part x and this may be null string.

The substring which generates the circuit starting from the start state which was lead by x, this part can be called as y and this cannot be null string.

The substring which is the remaining part of the word after y, this part can be called as z. It may be noted that this part may be null string as the word may end after y or z-part may itself be a circuit.

Thus the word w may be written as  $w = xyz$ , where x,y,z are strings belonging to  $\Sigma^*$  and y cannot be null string. The state 2 is such that it is revisited first while reading the word w. So the word w can be decomposed, according to pumping lemma, as  $w = xyz = (b)(bba)(baba)$

If y-part of w is continuously pumped, the resulting strings will be accepted by F and hence will be words in the language accepted by F. Thus, by pumping lemma, the language accepted by F is regular.

**Remark:** If the pumping lemma is applied directly on the language  $L = \{a^n b^n : n=0,1,2,3,\dots\}$ , it can be observed that for the word  $w = (aaa)(aaaabbbb)(bbb)$

where  $x = aaa$ ,  $y = aaaabbbb$  and  $z = bbb$

$xyyz$  will contain as many number of a's as there are b's but this string will not belong to L because the substring ab can occur at the most once in the words of L, while the string  $xyyz$  contains the substring ab twice. On the other hand if y-part consisting of only a's or b's, then  $xyyz$  will contain number of a's different from number of b's. This shows that pumping lemma does not hold and hence the language is not regular.

### Example

Consider the language **EQUAL**, of strings, defined over  $\Sigma = \{a,b\}$ , with number of a's equal to number of b's, i.e.  $\text{EQUAL} = \{\Lambda, ab, aabb, abab, baba, abba, \dots\}$

From the definition of **EQUAL**, it is clear that  $\{a^n b^n\} = a^* b^* \cap \text{EQUAL}$

Obviously  $a^* b^*$  defines a regular language while  $\{a^n b^n\}$  has been proved nonregular.

Using the theorem that intersection of two regular languages is, regular; it can be proved that the **EQUAL** is not regular. Because if it is considered regular then the language  $\{a^n b^n\}$  will, being intersection of regular languages, be regular language, which is impossible.

Following are the remarks regarding these examples

Remarks

In the previous examples, languages are proved to be regular or nonregular using pumping lemma. In fact to prove a certain language to be regular, it is not needed to use the full force of pumping lemma *i.e.* for a word with length greater than the number of states of the machine, decomposing the word into  $xyz$  and for a language to be regular it is sufficient that  $xyyz$  is in  $L$ . The condition that  $xy^n z$  is in  $L$  for  $n > 2$ , provides that the language is infinite.

Consider the language PALINDROME and a word  $w = aba$  belonging to PALINDROME. Decomposing  $w = xyz$  where  $x=a$ ,  $y=b$ ,  $z=a$ . It can be observed that the strings of the form  $xy^n z$  for  $n=1,2,3, \dots$ , belong to PALINDROME. Which shows that the pumping lemma holds for the language PALINDROME (which is non regular language). To overcome this drawback of pumping lemma, a revised version of pumping lemma is to be introduced.

## Theory of Automata

**Lecture N0. 27****Reading Material**Introduction to Computer Theory

## Chapter 10

**Summary**

Pumping lemma version II, proof, examples, Myhill Nerode theorem, examples

**Pumping Lemma version II**Statement

Let  $L$  be an infinite language accepted by a finite automaton with  $N$  states, then for all words  $w$  in  $L$  that have length more than  $N$ , there are strings  $x, y$  and  $z$  ( $y$  being non-null string) and  $\text{length}(x) + \text{length}(y) \leq N$  s.t.  $w = xyz$  and all strings of the form  $xy^n z$  are in  $L$  for  $n = 1, 2, 3, \dots$

Proof

The lemma can be proved, considering the following examples

Example

Consider the language PALINDROME which is obviously infinite language. It has already been shown that the PALINDROME satisfies pumping lemma version I (previous version). To check whether the new version of pumping lemma still holds in case of the PALINDROME, let the PALINDROME be a regular language and be accepted by an FA of 78 states. Consider the word  $w = a^{85}ba^{85}$ .

Decompose  $w$  as  $xyz$ , where  $x, y$  and  $z$  are all strings belonging to  $\Sigma^*$  while  $y$  is non-null string, s.t.

$\text{length}(x) + \text{length}(y) \leq 78$ , which shows that the substring  $xy$  is consisting of  $a$ 's and  $xyz$  will become  $a^{\text{more than } 85}ba^{85}$  which is not in PALINDROME. Hence pumping lemma version II is not satisfied for the language PALINDROME. Thus pumping lemma version II can't be satisfied by any non regular language. Following is another example in this regard

Example

Consider the language **PRIME**, of strings defined over  $\Sigma = \{a\}$ , as  $\{a^p : p \text{ is prime}\}$ , i.e.

**PRIME** =  $\{aa, aaa, aaaaa, aaaaaaa, \dots\}$

To prove this language to be nonregular, suppose contrary, i.e. **PRIME** is a regular language, then there exists an FA accepts the language PRIME. Let the number of states of this machine be 345 and choose a word  $w$  from PRIME with length more than 345, say, 347 i.e. the word  $w = a^{347}$

Since this language is supposed to be regular, therefore according to pumping lemma  $xy^n z$ , for  $n = 1, 2, 3, \dots$  are all in PRIME.

Consider  $n=348$ , then  $xy^n z = xy^{348} z = xy^{347} yz$ . Since  $x, y$  and  $z$  consist of  $a$ 's, so the order of  $x, y, z$  does not matter i.e.  $xy^{347} yz = xzy^{347} = a^{347} y^{347}$ ,  $y$  being non-null string and consisting of  $a$ 's it can be written  $y = a^m$ ,  $m=1, 2, 3, \dots, 345$ .

Thus  $xy^{348} z = a^{347} (a^m)^{347} = a^{347(m+1)}$

Now the number  $347(m+1)$  will not remain PRIME for  $m = 1, 2, 3, \dots, 345$ . Which shows that the string  $xy^{348} z$  is not in PRIME. Hence pumping lemma version II is not satisfied by the language PRIME. Thus PRIME is not regular.

Strings belonging to same class

Consider a regular language  $L$ , defined over an alphabet  $\Sigma$ . If, two strings  $x$  and  $y$ , defined over  $\Sigma$ , are run over an FA accepting the language  $L$ , then  $x$  and  $y$  are said to belong to the same class if they end in the same state, no matter that state is final or not.

**Note:** It is to be noted that this concept of strings  $x$  and  $y$  can be compared with indistinguishable strings w.r.t.  $L$  (discussed earlier). Equivalently, the strings  $x$  and  $y$  are said to belong to same class if for all strings  $z$ , either  $xz$  and  $yz$  belong to  $L$  or  $xz$  and  $yz$  don't belong to  $L$ .

**Myhill Nerode theorem**Statement

For a language  $L$ , defined over an alphabet  $\Sigma$ ,

$L$  partitions  $\Sigma^*$  into distinct classes.

If  $L$  is regular then,  $L$  generates finite number of classes.

If  $L$  generates finite number of classes then  $L$  is regular.

The proof is obvious from the following examples

#### Example

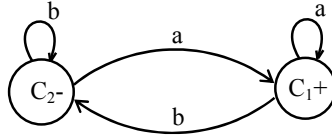
Consider the language  $L$  of strings, defined over  $\Sigma = \{a, b\}$ , **ending in a**.

It can be observed that  $L$  partitions  $\Sigma^*$  into the following two classes

$C_1$  = set of all strings ending in a.

$C_2$  = set of all strings not ending in a.

Since there are finite many classes generated by  $L$ , so  $L$  is regular and hence following is an FA, built with the help of  $C_1$  and  $C_2$ , accepting  $L$ .



#### Example

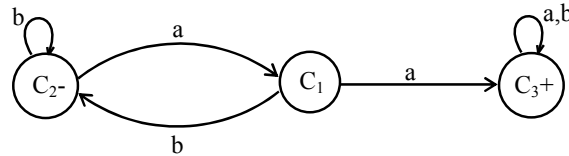
Consider the language  $L$  of strings, defined over  $\Sigma = \{a, b\}$ , **containing double a**. It can be observed that  $L$  partitions  $\Sigma^*$  into the following three classes

$C_1$  = set of all strings without aa but ending in a.

$C_2$  = set of  $\Lambda$  and all strings without aa but ending in b.

$C_3$  = set of all strings containing aa.

Since there are finite many classes generated by  $L$ , so  $L$  is regular and hence following is an FA, built with the help of  $C_1$ ,  $C_2$  and  $C_3$ , accepting  $L$ .



## Theory of Automata

## Lecture N0. 28

Reading MaterialIntroduction to Computer Theory

## Chapter 10

## Summary

Examples of Myhill Nerode theorem, Quotient of a language, examples, Pseudo theorem: Quotient of a language is regular, prefixes of a language, example

Example

Consider the language  $L$  which is EVEN-EVEN, defined over  $\Sigma = \{a,b\}$ . It can be observed that  $L$  partitions  $\Sigma^*$  into the following four classes

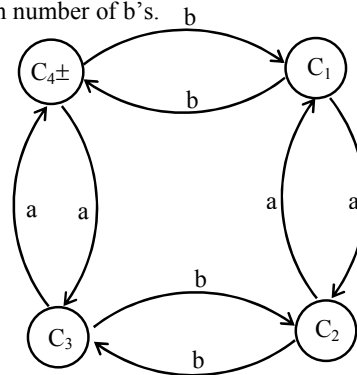
$C_1$  = set of all strings with even number of a's and odd number of b's.

$C_2$  = set of all strings with odd number of a's and odd number of b's.

$C_3$  = set of all strings with odd number of a's and even number of b's.

$C_4$  = set of all strings with even number of a's and even number of b's.

Since there are finite many classes generated by  $L$ , so  $L$  is regular and hence following is an FA, built with the help of  $C_1, C_2, C_3$  and  $C_4$ , accepting  $L$ .

Example

Consider the language  $L = \{w \in \{a,b\}^* : \text{length}(w) \geq 2, w \text{ ends in either } ab \text{ or } ba\}$ .

It can be observed that  $L$  partitions  $\Sigma^*$  into the following seven classes

$C_1$  = set containing only null string.

$C_2$  = set containing only letter a.

$C_3$  = set containing only letter b.

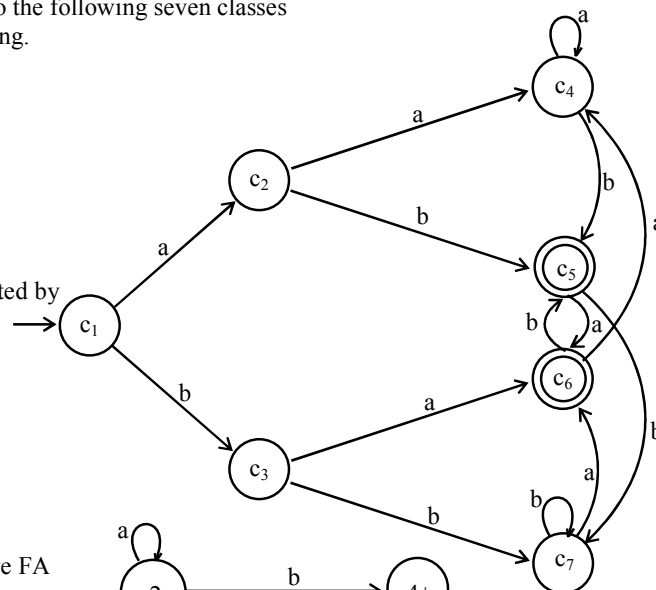
$C_4$  = set of strings ending in aa.

$C_5$  = set of strings ending in ab.

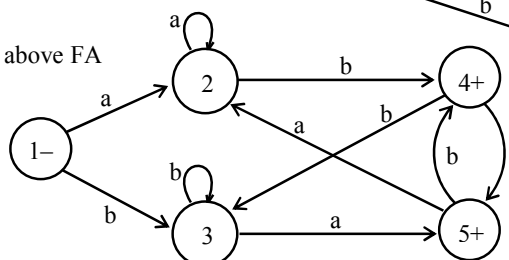
$C_6$  = set of strings ending in ba.

$C_7$  = set of strings ending in bb.

Since there are finite many classes generated by  $L$ , so  $L$  is regular and hence FA shown aside, is built with the help of  $C_1, C_2, C_3, C_4, C_5, C_6$  and  $C_7$ , accepting  $L$



Following is an FA equivalent to the above FA



Note It can be noted, from the above two FAs accepting the same language, that if the language  $L$ , partitions  $\Sigma^*$  into  $n$  distinct classes, then  $L$  may partition  $\Sigma^*$  into finite many distinct classes other than  $n$ .

### Quotient of a language into another

Remark The theorem has been proved to show under what conditions a language is regular. It has also been proved that the product of two regular languages is regular.

The question arises that whether there exists a theorem showing that quotient of regular languages is regular. There is a problem in defining the quotient of two regular languages. There is an approach in defining the quotient of regular languages i.e. the language  $Q$  is said to be **quotient of two regular languages**  $P$  and  $R$ , denoted by  $Q=R/P$  if  $PQ=R$ .

It is to be noted that this definition does not determine a unique language e.g. for  $P=Q=R$  expressed by  $a^*$  then  $PQ=R$  and so  $Q=R/P$  i.e.  $a^*=a^*/a^*$ . But for  $Q=\{\wedge\}$ ,  $P=R$  expressed by  $a^*$ ,  $PQ=R$  is still true which shows that  $Q=\{\wedge\}=R/P$  expressed by  $a^*/a^*$ .

Similarly, for the same  $P$  and  $R$ ,  $Q$  may be taken as  $\{\wedge\}, \{a\}, \{aaaa\}, \{aaaaaaaa\}, \dots$ . Thus there exist infinite many choices for defining the quotient language in this case of one-letter alphabet.

### Pseudo theorem

#### Statement

For three languages  $P, Q$  and  $R$ , while  $PQ=R$  the language  $Q$  must be regular if both  $P$  and  $R$  are regular.

(Note: It is to be noted that since this theorem is not true, so the theorem is called pseudo theorem.)

#### Disproof

The theorem can be disproved by contradiction i.e. supposing that  $Q$  is regular.

Let  $P=a^*$ ,  $Q$  be the product of  $\{a^n b^n : n=0,1,2,\dots\}$  and  $b^*$  then  $PQ=a^*\{a^n b^n\}b^*=a^*b^*=R$  which shows that  $R$  is regular.

To disproof this theorem, it is sufficient to prove that  $Q$  is not regular. By definition, the words in  $Q$  are of the form  $a^x b^y$  where  $x \leq y$ . Let  $Q$  be regular and hence there exists an FA that accepts  $Q$ . Suppose the number of states in this machine be  $N$ . Now the word  $a^N b^N$  is also in  $Q$  and must be accepted by this FA.

Since the number of states in this machine is  $N$ , there must be a circuit in this machine to run the substring  $a^N$ .

Thus while accepting the word  $a^N b^N$ , the machine looping the circuit once again, can accept the word  $a^{\text{more than } N} b^N$ , which is not in  $Q$ . Hence it is impossible to find any FA that accepts exactly the language  $Q$ . Thus  $Q$  is not regular and hence the theorem is disproved.

### Prefixes of a language in another language

If two languages  $R$  and  $Q$  are given, then the language **the prefixes of  $Q$  in  $R$**  denoted by **Pref( $Q$  in  $R$ )** is the set of strings of letters that, when concatenated to the front of some word in  $Q$  to produce some word in  $R$  i.e.

$\text{Pref}(Q \text{ in } R) =$  the set of all strings  $p$  such that there exists words  $q$  in  $Q$  and  $w$  in  $R$  such that  $pq = w$ . Following are the examples in this regard

#### Example

Let  $Q = \{aa, abaaabb, bbaaaaa, bbbbbb\}$  and  $R = \{b, bbbb, bbbaaa, bbbaaaa\}$

It can be observed that  $aa$  and  $bbaaaaa$  occur at the ending parts of some words of  $R$ , hence these words help in defining the language  $\text{pref}(Q \text{ in } R)$ . Thus  $\text{pref}(Q \text{ in } R) = \{b, bbba, bbbaaa\}$

Note: The language of prefixes may be consisting of word  $L$ , while there is also a possibility that this language may not contain any string (even not the null string).

## Theory of Automata

**Lecture N0. 29****Reading Material**Introduction to Computer Theory

## Chapter 10, 11

**Summary**

Example of prefixes of a language, Theorem:  $\text{pref}(Q \text{ in } R)$  is regular, proof, example, Decidability, deciding whether two languages are regular or not ?, method 1, example, method 2, example

**Example**

Let  $Q$  and  $R$  be expressed by  $ab^*a$  and  $(ba)^*$  respectively i.e.  $Q = \{aa, aba, abba, \dots\}$  and

$R = \{\wedge, ba, baba, bababa, \dots\}$ .  $aba$  is the only word in  $Q$  which can make a word in  $R$ , because the words in  $R$  don't contain the double letter. Thus  $\text{pref}(Q \text{ in } R) = \{b, bab, babab, \dots\}$ , which can be expressed by  $b(ab)^*$  or  $(ba)^*b$ .

**Remark**

It may be noted that the language  $R$  cannot be factorized with the help of language  $\text{Pref}(Q \text{ in } R)$  i.e.

$\text{Pref}(Q \text{ in } R)Q$  is not equal to  $R$  in general. However the following theorem shows that the language  $\text{pref}(Q \text{ in } R)$  is regular if  $R$  is regular, no matter whether the language  $Q$  is regular or not.

**Theorem****Statement**

If  $R$  is regular language and  $Q$  is any language (regular/ nonregular), then  $\text{Pref}(Q \text{ in } R)$  is regular.

**Proof**

Since  $R$  is regular there exists an FA that accepts this language. Choose a state, say,  $s$  of this FA and see whether this state can trace out a path ending up in a final state while running words from  $Q$ . If this state traces out a path ending up in a final state for any of the words of  $Q$ , mark this state with certain colour.

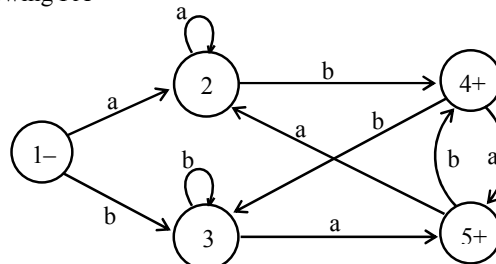
Repeat this process for remaining states of the machine. If at least one state of this machine is marked then it can be shown that the language  $\text{Pref}(Q \text{ in } R)$  is non-empty. Now build a new FA with some marked states by considering the initial state that of original FA and final states which are marked. The machine, thus obtained accepts exactly the language  $\text{Pref}(Q \text{ in } R)$ . Thus  $\text{Pref}(Q \text{ in } R)$  being accepted by an FA is regular.

**Remark**

There is a problem in deciding whether a state of FA should be marked or not when the language  $Q$  is infinite. This proof just gives non constructive method to prove that  $\text{Pref}(Q \text{ in } R)$  is regular.

**Example**

Consider the languages  $Q = \{aba, abb\}$  and  $R = \{w \in \{a,b\}^* : \text{length}(w) \geq 2, w \text{ ends in either } ab \text{ or } ba\}$ , where  $R$  may be accepted by the following FA



It can be observed that the string  $aba$  from  $Q$  make the words of  $R$  and hence the states 1,2,3,4 and 5 can easily be marked. Thus from the given FA, making the states 1, 2 and 3 to be final as well, the resulting FA will accept the language  $\text{pref}(Q \text{ in } R)$ . Moreover it can be observed that  $\text{pref}(Q \text{ in } R)$  can be expressed by  $(a+b)^*$ , which is the RE corresponding to the resulting FA as well.

**Decidability**

Effectively solvable problem

A problem is said to be effectively solvable if there exists an algorithm that provides the solution in finite number of steps *e.g.* finding solution for quadratic equation is effectively solvable problem, because the quadratic formula provides an algorithm that determines the solution in a finite number of arithmetic operations, (four multiplications, two subtractions, one square root and one division).

### Decision procedure

If an effectively solvable problem has answer in yes or no, then this solution is called decision procedure.

### Decidable problem

A problem that has decision procedure is called decidable problem *e.g.* the following problems

The two regular expressions define the same language.

The two FAs are equivalent.

### Determining whether the two languages are equivalent or not ?

If  $L_1$  and  $L_2$  are two regular languages, then they can be expressed by FAs. As shown earlier,  $L_1^c$ ,  $L_2^c$ ,  $L_1 \cup L_2$ ,  $L_1 \cap L_2$  are regular languages and the methods have already been developed to build their corresponding FAs. It can be observed that  $(L_1 \cap L_2^c) \cup (L_1^c \cap L_2)$  is regular language that accepts the words which are in  $L_1$  but not in  $L_2$  or else in  $L_2$  but not in  $L_1$ . The corresponding FA cannot accept any word which is in both  $L_1$  and  $L_2$  i.e. if  $L_1$  and  $L_2$  are equivalent, then this FA accepts not even null string. Following are the methods that determine whether a given FA accepts any words

#### Method 1

For FA corresponding to  $(L_1 \cap L_2^c) \cup (L_1^c \cap L_2)$ , the regular expression can be determined that defines the language accepted by this FA. From that regular expression one can determine whether this regular expression defines any word or not? Following are the steps to be followed

Remove all \*s from the regular expression.

Separate the right part of + and the plus itself.

The regular expression thus obtained if contains atleast one word then the language is not empty otherwise the language is empty.

#### Example

For  $(a+\wedge)(a^*b+ba^*)(a^*+\wedge)^*$  to be the regular expression of  $(L_1 \cap L_2^c) \cup (L_1^c \cap L_2)$ , it is required to find whether this language accepts any string or not?

After removing all \*s the RE will be  $(a+\wedge)(ab+ba)(a+\wedge)$

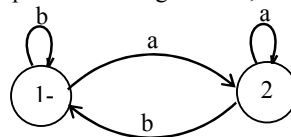
After separating the right part from + and the + itself the RE will be aaba

As this language contains atleast aaba as its word, so this language is not empty.

#### Remark

Sometimes, while determining regular expression for a given FA, it is impossible to write its regular expression *e.g.*

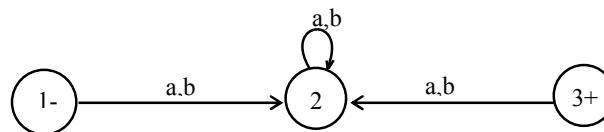
FA<sub>1</sub>



FA<sub>2</sub>



FA<sub>3</sub>



#### Method 2



To examine whether a certain FA accepts any words, it is required to seek the paths from initial to final state. But in large FA with thousands of states and millions of directed edges, without an effective procedure it is impossible to find a path from initial to final state. Following are the steps of this procedure

Mark the initial state.

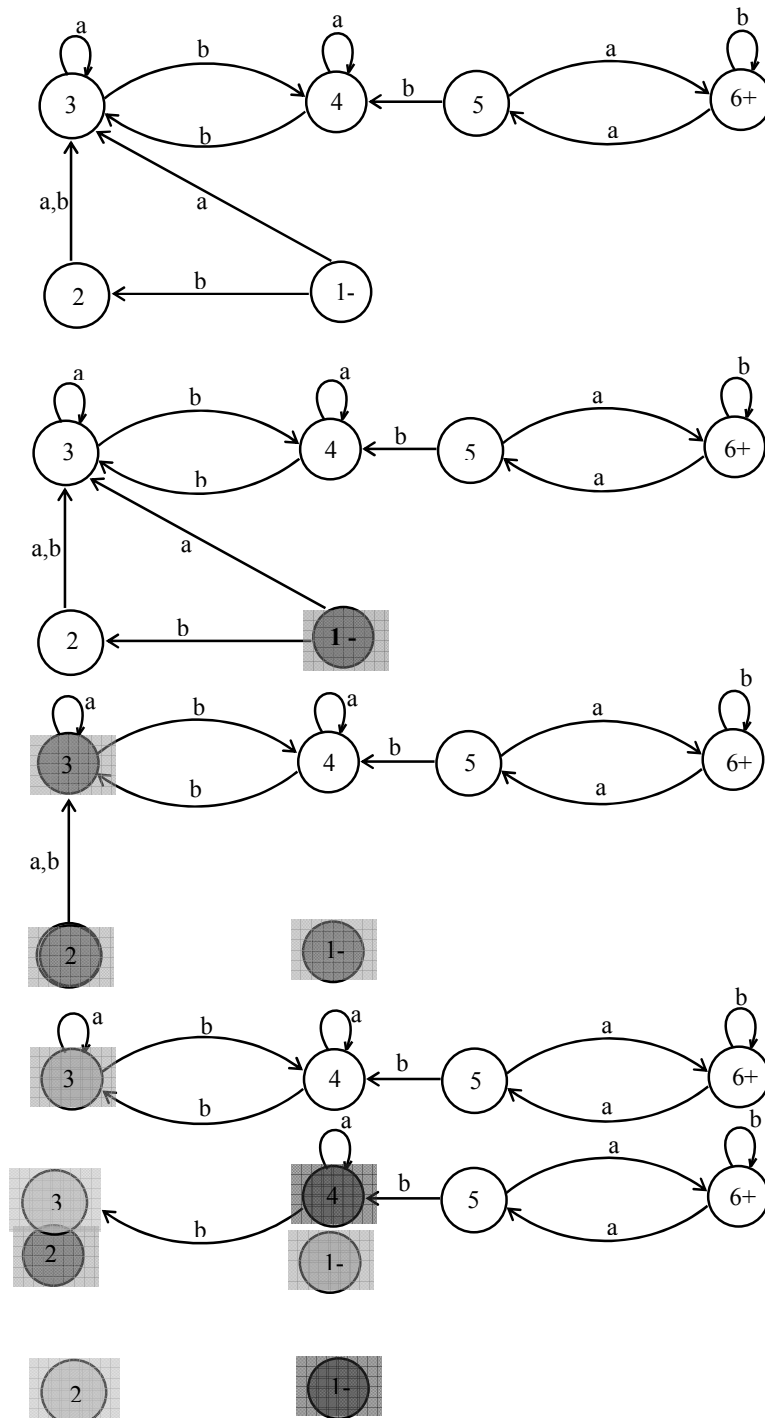
For every marked state follow each edge that leads out of it and mark the concatenating states and delete these edges.

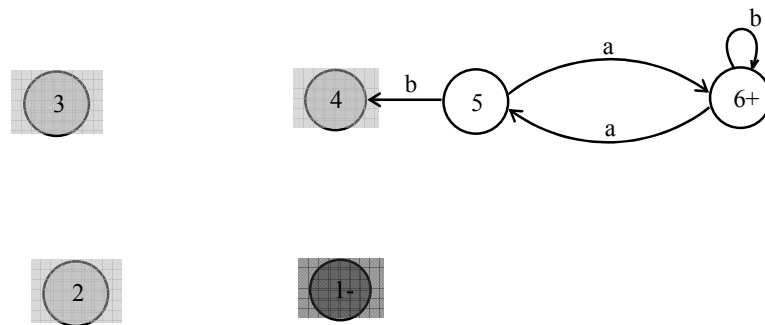
Repeat step 2 until no new state is marked.

If any of the final states are marked then the FA accepts some word, otherwise not.

#### Example

Suppose it is required to test the FA, which is given below, whether it accepts any string or not? Applying method 2 as





This FA accepts no string as after applying method 2, the final state is not marked.

## Theory of Automata

**Lecture N0. 30****Reading Material**Introduction to Computer Theory

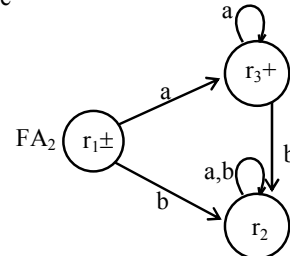
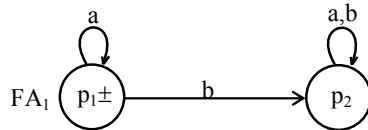
## Chapter 11,12

**Summary**

Deciding whether two languages are equivalent or not, example, deciding whether an FA accept any string or not, method 3, examples, finiteness of a language

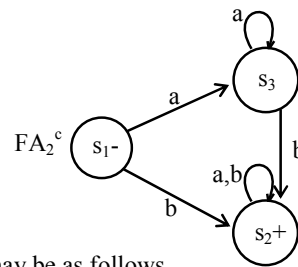
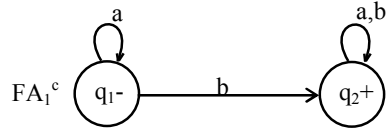
**Example**

Consider two languages  $L_1$  and  $L_2$ , expressed by the REs  $r_1 = a^*$  and  $r_2 = \wedge + aa^*$  respectively. To determine whether  $L_1$  and  $L_2$  are equivalent, let the corresponding FAs be

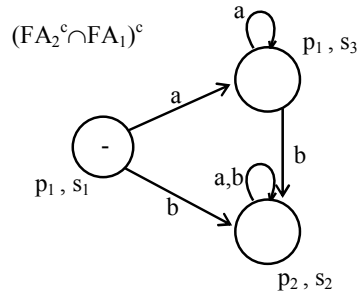
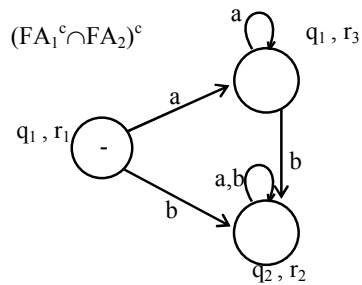


As discussed earlier, the FA corresponding to the language  $(L_1 \cap L_2^c) \cup (L_1^c \cap L_2)$  helps in this regard *i.e.* if this FA accepts any word then  $L_1$  and  $L_2$  are not equivalent other wise  $L_1$  and  $L_2$  are equivalent.

Following are the FAs corresponding to  $L_1^c$  and  $L_2^c$



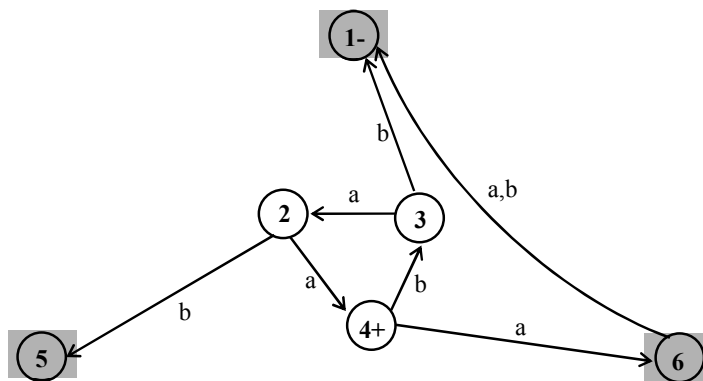
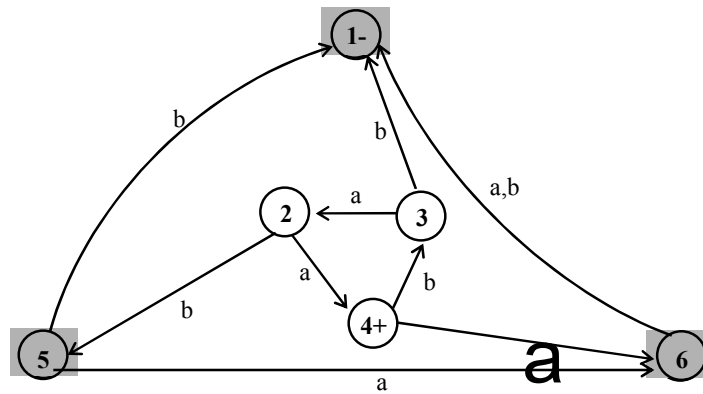
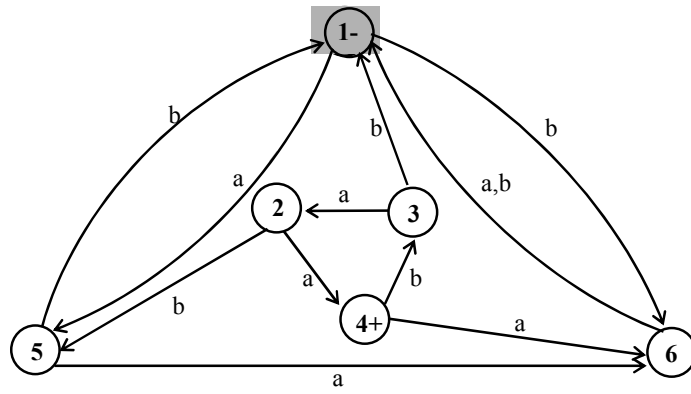
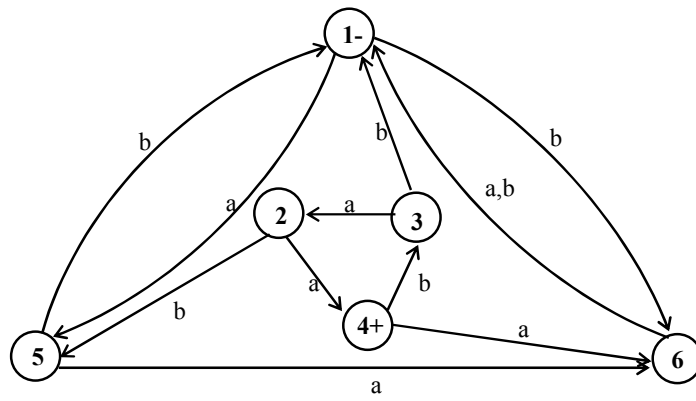
FAs corresponding to  $(FA_1^c \cap FA_2)^c$  and  $(FA_2^c \cap FA_1)^c$  may be as follows

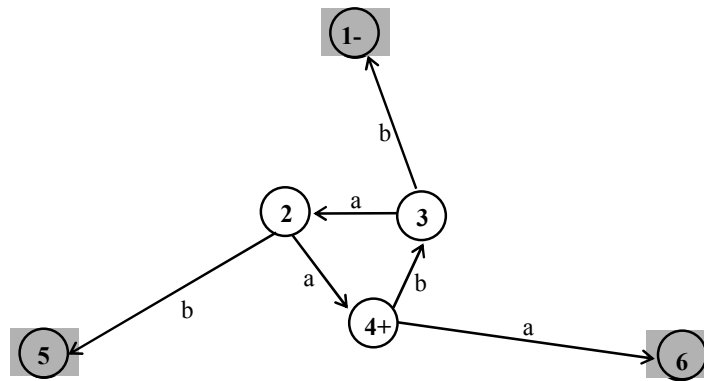


Both the FAs have no final state, so these FAs accept nothing. This implies that their union will not also accept any string. Hence FA corresponding to the language  $(L_1 \cap L_2^c) \cup (L_1^c \cap L_2)$  accepts nothing. Thus both the languages are equivalent.

**Example**

Following is an FA, for which it is required to decide whether it accepts any string or not? Using steps discussed in method 2, the following FA can be checked whether it accepts any word or not.





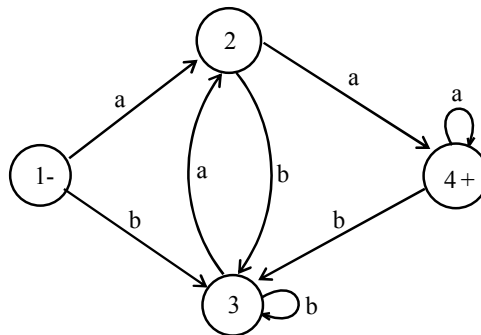
As no final state of the FA is marked, so the given FA accepts no word.

### Method 3

If the FA has N states, then test the words of length less than N. If no word is accepted by this FA, then it will accept no word.

**Note:** It is to be noted that all the methods discussed above, to determine whether an FA accepts certain word, are effective procedures.

**Example:** To determine whether the following FA accepts certain word, using method 3, all the strings of length less than 4 (*i.e.* less than the number of states of the FA) are sufficient to be tested

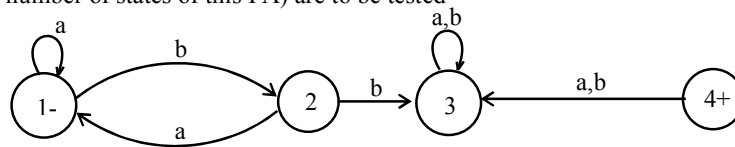


*i.e.*  $\Lambda$ , a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb are required to be tested.

It can be observed that the strings aa, baa, aaa are accepted by this FA, hence the language accepted by this FA is not empty.

### Example

Consider the following FA. To determine whether this FA accepts some word, all the strings of length less than 4 (*i.e.* less than the number of states of this FA) are to be tested



It can be observed that none of the strings  $\Lambda$ , a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, is accepted by this FA. Thus the given FA cannot accept any word.

### Observation

To find whether a regular expression defines an infinite language or not? The following possibilities are required to be checked.

If a regular expression contains \* then it may define an infinite language, with the exception  $\Lambda^*$  as  $\Lambda^* = \Lambda$  *e.g.*  $(\Lambda + a\Lambda^*)(\Lambda^* + \Lambda)^*$  defines finite language. While  $(\Lambda^* + a\Lambda^*)(\Lambda^* + \Lambda)^*$  defines an infinite language.

There are so many ways to decide whether an FA accepts a finite language or an infinite. Following is a theorem in this regard

**Theorem**

Let F be an FA having N states

If F accepts a word w s.t.  $N \leq \text{length}(w) < 2N$

then F accepts infinite language.

If F accepts an infinite language then there are some words w s.t.  $N \leq \text{length}(w) < 2N$

The first part can be proved, using the pumping lemma version II.

**Remark**

There is an effective procedure to decide whether an FA accepts finite or infinite language. For a machine with N number of states, the total number of strings to be tested, defined over an alphabet of m letters, is  $m^N + m^{N+1} + m^{N+2} + \dots + m^{2N-1}$ . After testing all these strings on the machine, if any is accepted then the machine accepts infinite language other wise not. *e.g.* for machine of three states and alphabet of two letters,  $2^3 + 2^4 + 2^5 = 56$  strings are to be tested.

## Theory of Automata

### Lecture N0. 31

#### Reading Material

#### Introduction to Computer Theory

#### Chapter 12

#### Summary

Context Free Grammar, Terminals, non-terminals, productions, CFG, context Free language, examples.

#### Context Free Grammar (CFG)

The earliest computers accepted no instructions other than their own assembly language. Every procedure, no matter how complicated, had to be encoded in the set of instructions, LOAD, STORE, ADD the contents of two registers and so on. The major problem was to display mathematical formulas as follows

$$S = \sqrt{\frac{(8 - 0)^2 + (7 - 10)^2 + (11 - 10)^2}{2}}$$

or

$$A = \frac{\frac{1}{2} + 9}{4 + \frac{8}{21} + \frac{5}{3 + \frac{1}{2}}}$$

So, it was necessary to develop a way of writing such expressions in one line of standard typewriter symbols, so that in this way a high level language could be invented. Before the invention of computers, no one would ever have dreamed of writing such complicated formula in parentheses e.g. the right side of formula can be written as ((1/2)+9)/(4+(8/21)+(5/(3+(1/2))))

The high level language is converted into assembly language codes by a program called compiler.

The compiler that takes the user's programs as its inputs and prints out an equivalent program written in assembly language.

Like spoken languages, high level languages for computer have also, certain grammar. But in case of computers, the grammatical rules, don't involve the meaning of the words.

It can be noted that the grammatical rules which involve the meaning of words are called **Semantics**, while those don't involve the meaning of the words are called **Syntactics**.

e.g. in English language, it can not be written "Buildings sing", while in computer language one number is as good as another.

e.g.  $X = B + 10$ ,  $X = B + 999$

#### Remark

In general, the rules of computer language grammar, are all syntactic and not semantic. A law of grammar is in reality a suggestion for possible substitutions.

#### CFG terminologies

Terminals: The symbols that can't be replaced by anything are called terminals.

Non-Terminals: The symbols that must be replaced by other things are called non-terminals.

Productions: The grammatical rules are often called productions.

#### CFG

CFG is a collection of the followings

An alphabet  $\Sigma$  of letters called terminals from which the strings are formed, that will be the words of the language.

A set of symbols called non-terminals, one of which is S, stands for "start here".

A finite set of productions of the form

non-terminal  $\rightarrow$  finite string of terminals and /or non-terminals.

#### Note

The terminals are designated by small letters, while the non-terminals are designated by capital letters.

There is at least one production that has the non-terminal S as its left side.

**Context Free Language (CFL)**

The language generated by CFG is called Context Free Language (CFL).

**Example**

$\Sigma = \{a\}$

productions:

$S \rightarrow aS$

$S \rightarrow \Lambda$

Applying production (1) six times and then production (2) once, the word aaaaaa is generated as

$S \Rightarrow aS$   
 $\Rightarrow aaS$   
 $\Rightarrow aaaS$   
 $\Rightarrow aaaaS$   
 $\Rightarrow aaaaaS$   
 $\Rightarrow aaaaaaS$   
 $\Rightarrow aaaaaa\Lambda$   
 $= aaaaaa$

It can be observed that prod (2) generates  $\Lambda$ , a can be generated applying prod. (1) once and then prod. (2), aa can be generated applying prod. (1) twice and then prod. (2) and so on. This shows that the grammar defines the language expressed by  $a^*$ .

**Example**

$\Sigma = \{a\}$

productions:

$S \rightarrow SS$

$S \rightarrow a$

$S \rightarrow \Lambda$

This grammar also defines the language expressed by  $a^*$ .

**Note:** It is to be noted that  $\Lambda$  is not considered to be terminal. It has a special status. If for a certain non-terminal  $N$ , there may be a production  $N \rightarrow \Lambda$ . This simply means that  $N$  can be deleted when it comes in the working string.

**Example**

$\Sigma = \{a,b\}$

productions:

$S \rightarrow X$

$S \rightarrow Y$

$X \rightarrow \Lambda$

$Y \rightarrow aY$

$Y \rightarrow bY$

$Y \rightarrow a$

$Y \rightarrow b$

All words of this language are of either X-type or of Y-type. *i.e.* while generating a word the first production used is  $S \rightarrow X$  or  $S \rightarrow Y$ . The words of X-type give only  $\Lambda$ , while the words of Y-type are words of finite strings of a's or b's or both *i.e.*  $(a+b)^+$ . Thus the language defined is expressed by  $(a+b)^*$ .

**Example**

$\Sigma = \{a,b\}$

productions:

$S \rightarrow aS$

$S \rightarrow bS$

$S \rightarrow a$

$S \rightarrow b$

$S \rightarrow \Lambda$

This grammar also defines the language expressed by  $(a+b)^*$ .



Example $\Sigma = \{a,b\}$ 

productions:

 $S \rightarrow XaaX$  $X \rightarrow aX$  $X \rightarrow bX$  $X \rightarrow \Lambda$ 

This grammar defines the language expressed by  $(a+b)^*aa(a+b)^*$ .

## Theory of Automata

**Lecture N0. 32****Reading Material**Introduction to Computer Theory

## Chapter 12

**Summary**

Examples of CFL, EVEN-EVEN, EQUAL, Language of strings containing bbb, PALINDROME,  $\{a^n b^n\}$ , Language of strings beginning and ending in different letters, Parsing tree, example

Example
 $\Sigma = \{a, b\}$ 

productions:

 $S \rightarrow SS$ 
 $S \rightarrow XS$ 
 $S \rightarrow \Lambda$ 
 $S \rightarrow YSY$ 
 $X \rightarrow aa$ 
 $X \rightarrow bb$ 
 $Y \rightarrow ab$ 
 $Y \rightarrow ba$ 

This grammar generates EVEN-EVEN language.

Example
 $\Sigma = \{a, b\}$ 

productions:

 $S \rightarrow aB$ 
 $S \rightarrow bA$ 
 $A \rightarrow a$ 
 $A \rightarrow aS$ 
 $A \rightarrow bAA$ 
 $B \rightarrow b$ 
 $B \rightarrow bS$ 
 $B \rightarrow aBB$ 

This grammar generates the language EQUAL (The language of strings, with number of a's equal to number of b's).

Note

It is to be noted that if the same non-terminal have more than one productions, it can be written in single line *e.g.*  $S \rightarrow aS, S \rightarrow bS, S \rightarrow \Lambda$  can be written as  $S \rightarrow aS|bS|\Lambda$

It may also be noted that the productions  $S \rightarrow SS|\Lambda$  always defines the language which is closed w.r.t. concatenation *i.e.* the language expressed by RE of type  $r^*$ . It may also be noted that the production  $S \rightarrow SS$  defines the language expressed by  $r^+$ .

Example
 $\Sigma = \{a, b\}$ 

productions:

 $S \rightarrow YXY$ 
 $Y \rightarrow aY|bY|\Lambda$ 
 $X \rightarrow bbb$ 

It can be observed that, using prod.2, Y generates  $\Lambda$ . Y generates a. Y generates b. Y also generates all the combinations of a and b. thus Y generates the strings generated by  $(a+b)$ . It may also be observed that the above

CFG generates the language expressed by  $(a+b)^*bbb(a+b)^*$ . Following are four words generated by the given CFG

$$\begin{aligned} S &\Rightarrow YXY \\ &\Rightarrow aYbbb\Lambda \\ &\Rightarrow abYbbb \\ &\Rightarrow ab\Lambda bbb \\ &= abbbb \end{aligned}$$

$$\begin{aligned} S &\Rightarrow YXY \\ &\Rightarrow \Lambda bbb aY \\ &\Rightarrow bbb abY \\ &\Rightarrow bbb ab aY \\ &\Rightarrow bbb ab a \Lambda \\ &= bbb ab a \end{aligned}$$

$$\begin{aligned} S &\Rightarrow YXY \\ &\Rightarrow bYbbbaY \\ &\Rightarrow b\Lambda bbb abY \\ &\Rightarrow bbb babbY \\ &\Rightarrow bbb babbaY \\ &\Rightarrow bbb babba \Lambda \\ &= bbb babba \end{aligned}$$

$$\begin{aligned} S &\Rightarrow YXY \\ &\Rightarrow bYbbbaY \\ &\Rightarrow b\Lambda bbb a \Lambda \\ &= bbbba \end{aligned}$$

#### Example

Consider the following CFG

$$S \rightarrow SS|XaXaX|\Lambda$$

$$X \rightarrow bX|\Lambda$$

It can be observed that, using prod.2, X generates  $\Lambda$ . X generates any number of b's. Thus X generates the strings generated by  $b^*$ . It may also be observed that the above CFG generates the language expressed by  $(b^*ab^*ab^*)^*$ .

#### Example

Consider the following CFG

$$\Sigma = \{a,b\}$$

productions:

$$S \rightarrow aSa|bSb|a|b|\Lambda$$

The above CFG generates the language PALINDROME. It may be noted that the CFG

$$S \rightarrow aSa|bSb|a|b$$
 generates the language NON-NULLPALINDROME.

#### Example

Consider the following CFG

$$\Sigma = \{a,b\}$$

productions:

$$S \rightarrow aSb|ab|\Lambda$$

It can be observed that the CFG generates the language  $\{a^n b^n : n = 0, 1, 2, 3, \dots\}$ . It may also be noted that the language  $\{a^n b^n : n = 1, 2, 3, \dots\}$  can be generated by the CFG,  $S \rightarrow aSb|ab$

#### Example

Consider the following CFG

$$S \rightarrow aXb|bXa$$

$$X \rightarrow aX|bX|\Lambda$$

The above CFG generates the language of strings, defined over  $\Sigma = \{a,b\}$ , **beginning and ending in different letters**.

#### Trees

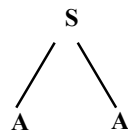
As in English language any sentence can be expressed by parse tree, so any word generated by the given CFG can also be expressed by the parse tree, e.g. consider the following CFG

$$S \rightarrow AA$$

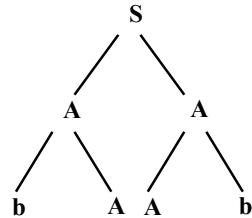
$$A \rightarrow AAA|bA|Ab|a$$

Obviously, baab can be generated by the above CFG. To express the word baab as a parse tree, start with S.

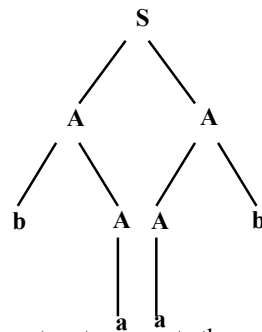
Replace S by the string AA, of nonterminals, drawing the downward lines from S to each character of this string as follows



Now let the left A be replaced by bA and the right one by Ab then the tree will be



Replacing both A's by a, the above tree will be



Thus the word baab is generated. The above tree to generate the word baab is called **Syntax tree or Generation tree or Derivation tree as well.**

## Theory of Automata

**Lecture N0. 33****Reading Material**Introduction to Computer Theory

## Chapter 12

**Summary**

Example of trees, Polish Notation, examples, Ambiguous CFG, example

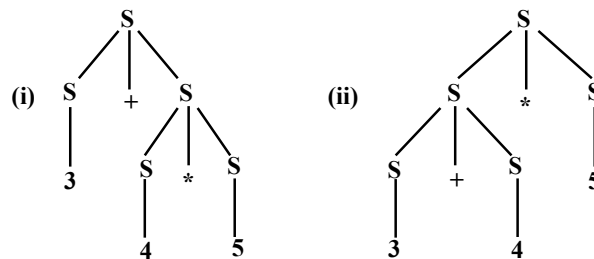
Example

Consider the following CFG

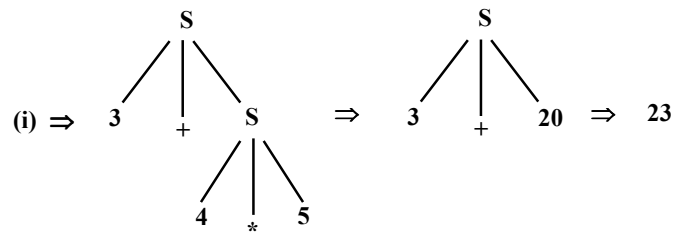
$$S \rightarrow S+S|S*S|number$$

where S and number are non-terminals and the operators behave like terminals.

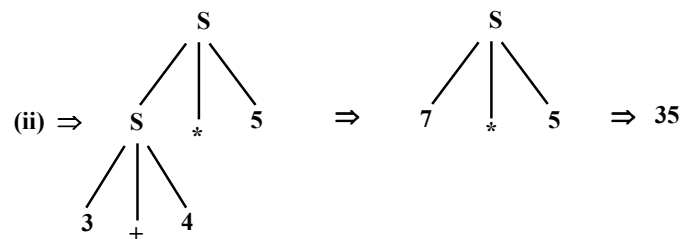
The above CFG creates ambiguity as the expression  $3+4*5$  has two possibilities  $(3+4)*5=35$  and  $3+(4*5)=23$  which can be expressed by the following production trees



The expressions can be calculated starting from bottom to the top, replacing each nonterminal by the result of calculation e.g.



Similarly



The ambiguity that has been observed in this example can be removed with a change in the CFG as discussed in the following example

Example

$$S \rightarrow (S+S)|(S*S)|number$$

where S and number are nonterminals, while (, \*, +, ) and the numbers are terminals.

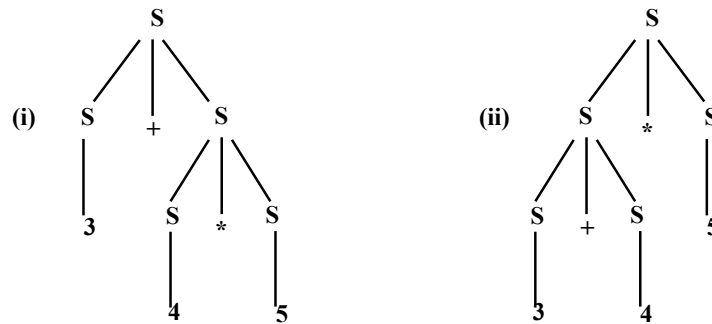
Here it can be observed that

$$\begin{aligned} S &\Rightarrow (S+S) \\ &\Rightarrow (S+(S*S)) \\ &\Rightarrow (3+(4*5)) = 23 \\ S &\Rightarrow (S*S) \\ &\Rightarrow ((S+S)*S) \end{aligned}$$

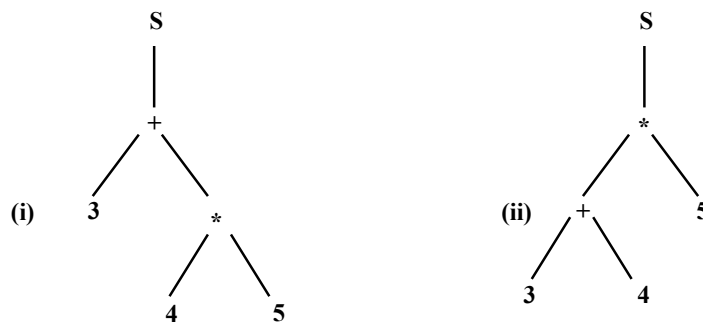
$$\Rightarrow ((3+4)*5) = 35$$

### Polish Notation (o-o-o)

There is another notation for arithmetic expressions for the CFG,  $S \rightarrow S+S|S*S|\text{number}$ . Consider the following derivation trees



The arithmetic expressions shown by the trees (i) and (ii) can be calculated from the following trees, respectively



Here most of the S's are eliminated.

The branches are connected directly with the operators. Moreover, the operators + and \* are no longer terminals as these are to be replaced by numbers (results).

To write the arithmetic expression, it is required to traverse from the left side of S and going onward around the tree. The arithmetic expressions will be as under

(i) + 3 \* 4 5      (ii) \* + 3 4 5

The above notation is called operator prefix notation.

To evaluate the strings of characters, the first substring (from the left) of the form operator-operand-operand (o-o-o) is found and is replaced by its calculation e.g.

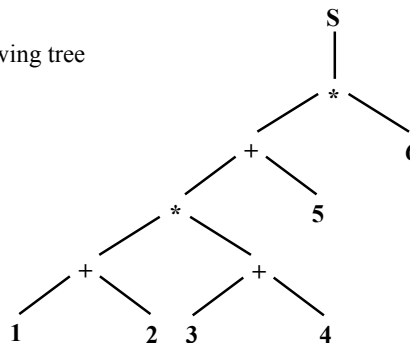
$$+3*4\ 5 = +3\ 20 = 23$$

$$*+3\ 4\ 5 = *7\ 5 = 35$$

It may be noted that  $4*5+3$  is an infix arithmetic expression, while an arithmetic expression in (o-o-o) form is a prefix arithmetic expression.

### Example

To calculate the arithmetic expression of the following tree



It can be written as  $*+*+1\ 2+3\ 4\ 5\ 6$

The above arithmetic expression in (o-o-o) form can be calculated as

$*+*+1\ 2+3\ 4\ 5\ 6 = *+*3+3\ 4\ 5\ 6$

$= *+*3\ 7\ 5\ 6 = *+21\ 5\ 6 = *26\ 6 = 156.$

#### Note

The previous prefix arithmetic expression can be converted into the following infix arithmetic expression as

$*+*+1\ 2+3\ 4\ 5\ 6$

$= *+*+1\ 2\ (3+4)\ 5\ 6$

$= *+*(1+2)\ (3+4)\ 5\ 6$

$= *(((1+2)*(3+4)) + 5)\ 6$

$= (((1+2)*(3+4)) + 5)*6$

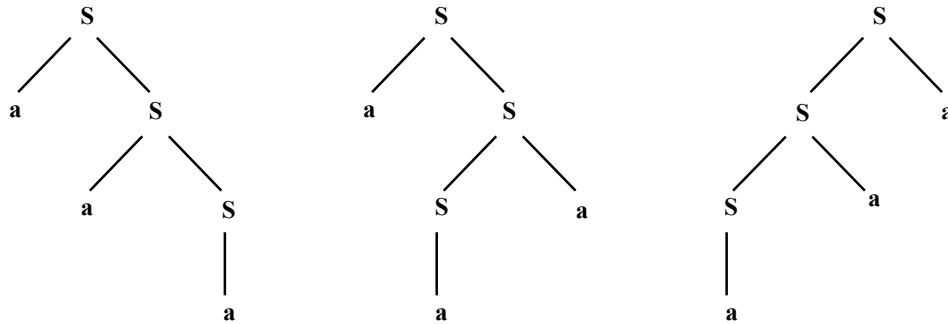
#### Ambiguous CFG

The CFG is said to be ambiguous if there exists atleast one word of it's language that can be generated by the different production trees.

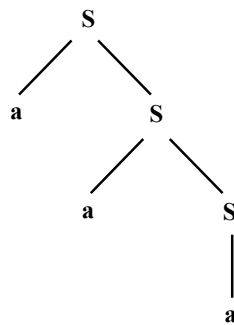
Example: Consider the following CFG

$S \rightarrow aS|Sa|a$

The word aaa can be generated by the following three different trees



Thus the above CFG is ambiguous, while the CFG,  $S \rightarrow aS|a$  is not ambiguous as neither the word aaa nor any other word can be derived from more than one production trees. The derivation tree for aaa is as follows



## Theory of Automata

**Lecture N0. 34****Reading Material**Introduction to Computer Theory

## Chapter 12,13

**Summary**

Example of Ambiguous Grammar, Example of Unambiguous Grammar (PALINDROME), Total Language tree with examples (Finite and infinite trees), Regular Grammar, FA to CFG, Semi word and Word, Theorem, Defining Regular Grammar, Method to build TG for Regular Grammar

**Example**

Consider the following CFG

$$S \rightarrow aS \mid bS \mid aaS \mid \Lambda$$

It can be observed that the word aaa can be derived from more than one production trees. Thus, the above CFG is ambiguous. This ambiguity can be removed by removing the production  $S \rightarrow aaS$

**Example**

Consider the CFG of the language PALINDROME

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \Lambda$$

It may be noted that this CFG is unambiguous as all the words of the language PALINDROME can only be generated by a unique production tree.

It may be noted that if the production  $S \rightarrow aaSaa$  is added to the given CFG, the CFG thus obtained will be no more unambiguous.

**Total language tree**

For a given CFG, a tree with the start symbol S as its root and whose nodes are working strings of terminals and non-terminals. The descendants of each node are all possible results of applying every production to the working string. This tree is called **total language tree**.

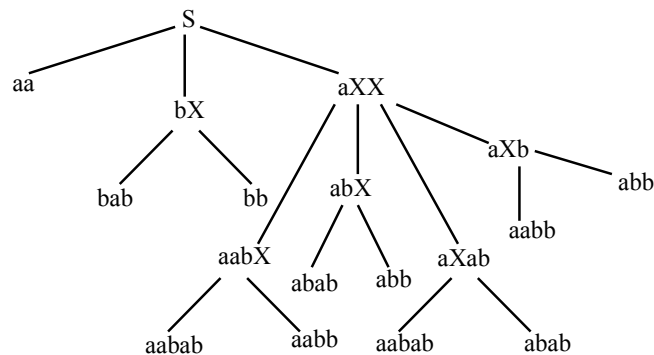
**Example**

Consider the following CFG

$$S \rightarrow aa \mid bX \mid aXX$$

$$X \rightarrow ab \mid b$$

then the total language tree for the given CFG may be



It may be observed from the above total language tree that dropping the repeated words, the language generated by the given CFG is {aa, bab, bb, aabab, aabb, abab, abb}

**Example**

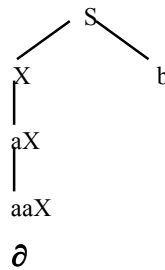
Consider the following CFG

$$S \rightarrow X \mid b, X \rightarrow aX$$

then following will be the total language tree of the above CFG



**Note:** It is to be noted that the only word in this language is b.



aaa ...aX

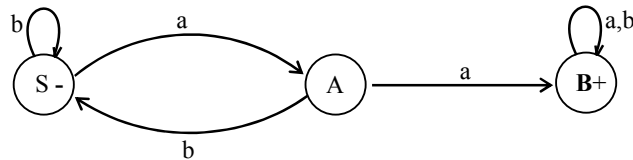
### **Regular Grammar**

All regular languages can be generated by CFGs. Some nonregular languages can be generated by CFGs but not all possible languages can be generated by CFG, *e.g.* the CFG  $S \rightarrow aSb|ab$  generates the language  $\{a^n b^n : n=1,2,3, \dots\}$ , which is nonregular.

**Note:** It is to be noted that for every FA, there exists a CFG that generates the language accepted by this FA.

### **Example**

Consider the language L expressed by  $(a+b)^*aa(a+b)^*$  *i.e.* the language of strings, defined over  $\Sigma = \{a,b\}$ , containing aa. To construct the CFG corresponding to L, consider the FA accepting L, as follows



CFG corresponding to the above FA may be

$S \rightarrow bS|aA$

$A \rightarrow aB|bS$

$B \rightarrow aB|bB|\Lambda$

It may be noted that the number of terminals in above CFG is equal to the number of states of corresponding FA where the nonterminal S corresponds to the initial state and each transition defines a production.

### **Semiword**

A semiword is a string of terminals (may be none) concatenated with exactly one nonterminal on the right *i.e.* a semiword, in general, is of the following form

(terminal)(terminal)... (terminal)(nonterminal)

### **word**

A word is a string of terminals.  $\Lambda$  is also a word.

### **Theorem**

If every production in a CFG is one of the following forms

Nonterminal  $\rightarrow$  semiword

Nonterminal  $\rightarrow$  word

then the language generated by that CFG is regular.

### **Regular grammar**

#### **Definition**

A CFG is said to be a **regular grammar** if it generates the regular language *i.e.* a CFG is said to be a **regular grammar** in which each production is one of the two forms

Nonterminal  $\rightarrow$  semiword

Nonterminal  $\rightarrow$  word

### Examples

The CFG  $S \rightarrow aaS|bbS|\Lambda$  is a regular grammar. It may be observed that the above CFG generates the language of strings expressed by the RE  $(aa+bb)^*$ .

The CFG  $S \rightarrow aA|bB$ ,  $A \rightarrow aS|a$ ,  $B \rightarrow bS|b$  is a regular grammar. It may be observed that the above CFG generates the language of strings expressed by RE  $(aa+bb)^+$ .

Following is a method of building TG corresponding to the regular grammar.

### TG for Regular Grammar

For every regular grammar there exists a TG corresponding to the regular grammar.

Following is the method to build a TG from the given regular grammar

Define the states, of the required TG, equal in number to that of nonterminals of the given regular grammar. An additional state is also defined to be the final state. The initial state should correspond to the nonterminal S.

For every production of the given regular grammar, there are two possibilities for the transitions of the required TG

If the production is of the form nonterminal  $\rightarrow$  semiword, then transition of the required TG would start from the state corresponding to the nonterminal on the left side of the production and would end in the state corresponding to the nonterminal on the right side of the production, labeled by string of terminals in semiword.

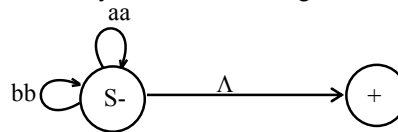
If the production is of the form nonterminal  $\rightarrow$  word, then transition of the TG would start from the state corresponding to nonterminal on the left side of the production and would end on the final state of the TG, labeled by the word.

### Example

Consider the following CFG

$$S \rightarrow aaS|bbS|\Lambda$$

The TG accepting the language generated by the above CFG is given below



The corresponding RE may be  $(aa+bb)^*$ .

## Theory of Automata

**Lecture N0. 35****Reading Material**Introduction to Computer Theory

## Chapter 13

**Summary**

Examples of building TG's corresponding to the Regular Grammar, Null productions with examples, Nullable productions with examples, Unit production with example, Chomsky Normal Form (Definition)

Example

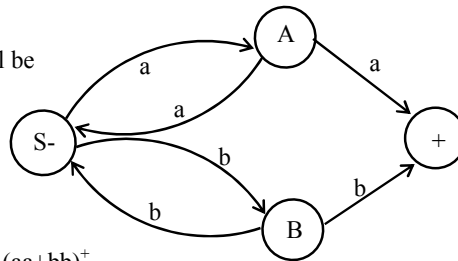
Consider the following CFG

$$S \rightarrow aA|bB$$

$$A \rightarrow aS|a$$

$$B \rightarrow bS|b$$

then the corresponding TG will be



The corresponding RE may be  $(aa+bb)^+$ .

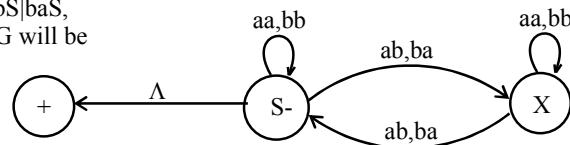
Example

Consider the following CFG

$$S \rightarrow aaS|bbS|abX|baX|\Lambda$$

$$X \rightarrow aaX|bbX|abS|baS,$$

then the corresponding TG will be



The corresponding language is EVEN-EVEN.

**Null Production**Definition

The production of the form nonterminal  $\rightarrow \Lambda$  is said to be **null production**.

Example: Consider the CFG,  $S \rightarrow aA|bB|\Lambda$ ,  $A \rightarrow aa|\Lambda$ ,  $B \rightarrow aS$

Here  $S \rightarrow \Lambda$  and  $A \rightarrow \Lambda$  are null productions.

Note

If a CFG has a null production, then it is possible to construct another CFG without null production accepting the same language with the exception of the word  $\Lambda$  i.e. if the language contains the word  $\Lambda$  then the new language cannot have the word  $\Lambda$ .

Following is a method to construct a CFG without null production for a given CFG

Method

Delete all the Null productions and add new productions e.g.

consider the productions of a certain CFG  $X \rightarrow aNbNa$ ,  $N \rightarrow \Lambda$ , delete the production  $N \rightarrow \Lambda$  and using the production  $X \rightarrow aNbNa$ , add the new productions  $X \rightarrow aNbNa$ ,  $X \rightarrow abNa$  and  $X \rightarrow aba$

Thus the new CFG will contain the productions  $X \rightarrow aNbNa|abNa|aba|aNbNa$

*Note:* It is to be noted that  $X \rightarrow aNbNa$  will still be included in the new CFG.

**Nullable Production**

Definition

A production is called **nullable production** if it is of the form  $N \rightarrow \Lambda$

or

there is a derivation that starts at  $N$  and leads to  $\Lambda$  i.e.  $N_1 \rightarrow N_2, N_2 \rightarrow N_3, N_3 \rightarrow N_4, \dots, N_n \rightarrow \Lambda$ , where  $N, N_1, N_2, \dots, N_n$  are non terminals.

Example

Consider the following CFG

$$S \rightarrow AA|bB, A \rightarrow aa|B, B \rightarrow aS | \Lambda$$

Here  $S \rightarrow AA$  and  $A \rightarrow B$  are nullable productions, while  $B \rightarrow \Lambda$  is null a production.

Following is an example describing the method to convert the given CFG containing null productions and nullable productions into the one without null productions

Example

Consider the following CFG

$$S \rightarrow XaY|YY|aX|ZYY$$

$$X \rightarrow Za|bZ|ZZ|Yb$$

$$Y \rightarrow Ya|XY|\Lambda$$

$$Z \rightarrow aX|YYY$$

It is to be noted that in the given CFG, the productions  $S \rightarrow YY, X \rightarrow ZZ, Z \rightarrow YYY$  are Nullable productions, while  $Y \rightarrow \Lambda$  is Null production.

Here the method of removing null productions, as discussed earlier, will be used along with replacing nonterminals corresponding to nullable productions like nonterminals for null productions are replaced.

Thus the required CFG will be

$$S \rightarrow XaY|Xa|aY|a|YY|Y|aX|ZYY|YX|ZX|ZY|X|Z$$

$$X \rightarrow Za|a|bZ|b|ZZ|Z|Yb$$

$$Y \rightarrow Ya|a|XY|X|Y$$

$$Z \rightarrow aX|a|YYY|YY|Y$$
Example

Consider the following CFG

$$S \rightarrow XY, X \rightarrow Zb, Y \rightarrow bW$$

$$Z \rightarrow AB, W \rightarrow Z, A \rightarrow aA|bA|\Lambda$$

$$B \rightarrow Ba|Bb|\Lambda.$$

Here  $A \rightarrow \Lambda$  and  $B \rightarrow \Lambda$  are null productions, while  $Z \rightarrow AB, W \rightarrow Z$  are nullable productions. The new CFG after, applying the method, will be

$$S \rightarrow XY$$

$$X \rightarrow Zb|b$$

$$Y \rightarrow bW|b$$

$$Z \rightarrow AB|A|B$$

$$W \rightarrow Z$$

$$A \rightarrow aA|a|bA|b$$

$$B \rightarrow Ba|a|Bb|b$$
Note

While adding new productions all Nullable productions should be handled with care. All Nullable productions will be used to add new productions, but only the Null production will be deleted.

Unit production

The productions of the form nonterminal  $\rightarrow$  one nonterminal, is called the **unit production**.

Following is an example showing how **to eliminate the unit productions from a given CFG**.

Example

Consider the following CFG

$$S \rightarrow A|bb$$

$$A \rightarrow B|b$$

$$B \rightarrow S|a$$

Separate the unit productions from the nonunit productions as shown below

unit prods.

$S \rightarrow A$

$A \rightarrow B$

$B \rightarrow S$

nonunit prods.

$S \rightarrow bb$

$A \rightarrow b$

$B \rightarrow a$

$S \rightarrow A$  gives  $S \rightarrow b$

(using  $A \rightarrow b$ )

$S \rightarrow A \rightarrow B$  gives  $S \rightarrow a$

(using  $B \rightarrow a$ )

$A \rightarrow B$  gives  $A \rightarrow a$

(using  $B \rightarrow a$ )

$A \rightarrow B \rightarrow S$  gives  $A \rightarrow bb$

(using  $S \rightarrow bb$ )

$B \rightarrow S$  gives  $B \rightarrow bb$

(using  $S \rightarrow bb$ )

$B \rightarrow S \rightarrow A$  gives  $B \rightarrow b$

(using  $A \rightarrow b$ )

Thus the new CFG will be

$S \rightarrow a|b|bb, A \rightarrow a|b|bb, B \rightarrow a|b|bb.$

Which generates the finite language  $\{a,b,bb\}$ .

### **Chomsky Normal Form**

If a CFG has only productions of the form

nonterminal  $\rightarrow$  string of two nonterminals

or

nonterminal  $\rightarrow$  one terminal

then the CFG is said to be in Chomsky Normal Form (CNF).

## Theory of Automata

**Lecture N0. 36****Reading Material**Introduction to Computer Theory

## Chapter 13,14

**Summary**

Chomsky Normal Form, Theorem regarding CNF, examples of converting CFG to be in CNF, Example of an FA corresponding to Regular CFG, Left most and Right most derivations, New format of FAs.

**Chomsky Normal Form (CNF)**

If a CFG has only productions of the form

nonterminal  $\rightarrow$  string of two nonterminals

or

nonterminal  $\rightarrow$  one terminal

then the CFG is said to be in Chomsky Normal Form (CNF).

Note

It is to be noted that any CFG can be converted to be in CNF, if the null productions and unit productions are removed. Also if a CFG contains nullable productions as well, then the corresponding new production are also to be added. Which leads the following theorem

**Theorem**

All NONNULL words of the CFL can be generated by the corresponding CFG which is in CNF *i.e.* the grammar in CNF will generate the same language except the null string.

Following is an example showing that a CFG in CNF generates all nonnull words of corresponding CFL.

Example

Consider the following CFG

$S \rightarrow aSa|bSb|a|b|aa|bb$

To convert the above CFG to be in CNF, introduce the new productions as

$A \rightarrow a, B \rightarrow b$ , then the new CFG will be

$S \rightarrow ASA|BSB|AA|BB|a|b$

$A \rightarrow a$

$B \rightarrow b$

Introduce nonterminals  $R_1$  and  $R_2$  so that

$S \rightarrow AR_1|BR_2|AA|BB|a|b$

$R_1 \rightarrow SA$

$R_2 \rightarrow SB$

$A \rightarrow a$

$B \rightarrow b$

which is in CNF.

It may be observed that the above CFG which is in CNF generates the NONNULLPALINDROME, which does not contain the null string.

Example

Consider the following CFG

$S \rightarrow ABAB$

$A \rightarrow a|\Lambda$

$B \rightarrow b|\Lambda$

Here  $S \rightarrow ABAB$  is nullable production and  $A \rightarrow \Lambda, B \rightarrow \Lambda$  are null productions. Removing the null productions

$A \rightarrow \Lambda$  and  $B \rightarrow \Lambda$ , and introducing the new productions as

$S \rightarrow BAB|AAB|ABB|ABA|AA|AB|BA|BB|A|B$

Now  $S \rightarrow A|B$  are unit productions to be eliminated as shown below

$S \rightarrow A$  gives  $S \rightarrow a$  (using  $A \rightarrow a$ )

$S \rightarrow B$  gives  $S \rightarrow b$  (using  $B \rightarrow b$ )

Thus the new resultant CFG takes the form

$S \rightarrow BAB|AAB|ABB|ABA|AA|AB|BA|BB|a|b$ ,  $A \rightarrow a$ ,  $B \rightarrow b$ .

Introduce the nonterminal  $C$  where  $C \rightarrow AB$ , so that

$S \rightarrow BAB|AAB|ABB|ABA|AA|AB|BA|BB|a|b$

$S \rightarrow BC|AC|CB|CA|AA|C|BA|BB|a|b$

$A \rightarrow a$

$B \rightarrow b$

$C \rightarrow AB$

is the CFG in CNF.

### Example

To construct an FA that accepts the grammar

$S \rightarrow abA$

$A \rightarrow baB$

$B \rightarrow aA|bb$

The language can be identified by the three words generated as follows

$S \Rightarrow abA$

$\Rightarrow abbaB$  (using  $A \rightarrow baB$ )

$\Rightarrow \underline{abba} \underline{bb}$  (using  $B \rightarrow bb$ )

$S \Rightarrow abA$

$\Rightarrow abbaB$  (using  $A \rightarrow baB$ )

$\Rightarrow abbaaA$  (using  $B \rightarrow aA$ )

$\Rightarrow abbaabaB$  (using  $A \rightarrow baB$ )

$\Rightarrow \underline{abbaaba} \underline{bb}$  (using  $B \rightarrow bb$ )

$S \Rightarrow abA$

$\Rightarrow abbaB$  (using  $A \rightarrow baB$ )

$\Rightarrow abbaaA$  (using  $B \rightarrow aA$ )

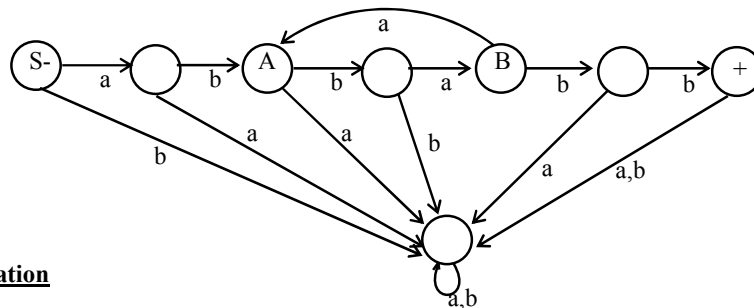
$\Rightarrow abbaabaB$  (using  $A \rightarrow baB$ )

$\Rightarrow abbaabaaA$  (using  $B \rightarrow aA$ )

$\Rightarrow abbaabaabaB$  (using  $A \rightarrow baB$ )

$\Rightarrow \underline{abbaabaaba} \underline{bb}$  (using  $B \rightarrow bb$ )

which shows that corresponding language has RE  $abba(aba)^*bb$ . Thus the FA accepting the given CFG may be



### Left most derivation

#### Definition

The derivation of a word  $w$ , generated by a CFG, such that at each step, a production is applied to the left most nonterminal in the working string, is said to be **left most derivation**.

It is to be noted that the nonterminal that occurs first from the left in the working string, is said to be **left most nonterminal**.

### Example

Consider the following CFG

$S \rightarrow XY$   
 $X \rightarrow XX|a$   
 $Y \rightarrow YY|b$

then following are the two left most derivations of aaabb

$S \Rightarrow \underline{X}Y$   
 $\Rightarrow \underline{XX}Y$   
 $\Rightarrow a\underline{X}Y$   
 $\Rightarrow a\underline{XX}Y$   
 $\Rightarrow aa\underline{X}Y$   
 $\Rightarrow aaa\underline{Y}$   
 $\Rightarrow aaa\underline{YY}$   
 $\Rightarrow aaab\underline{Y}$   
 $= aaabb$

$S \Rightarrow \underline{X}Y$   
 $\Rightarrow \underline{XX}Y$   
 $\Rightarrow \underline{XXX}Y$   
 $\Rightarrow a\underline{XX}Y$   
 $\Rightarrow aa\underline{X}Y$   
 $\Rightarrow aaa\underline{Y}$   
 $\Rightarrow aaa\underline{YY}$   
 $\Rightarrow aaab\underline{Y}$   
 $= aaabb$

### **Theorem**

Any word that can be generated by a certain CFG has also a left most derivation.

It is to be noted that the above theorem can be stated for right most derivation as well.

### **Example**

Consider the following CFG

$S \rightarrow YX$   
 $X \rightarrow XX|b$   
 $Y \rightarrow YY|a$

Following are the left most and right most derivations of abbbb

$S \Rightarrow \underline{Y}X$   
 $\Rightarrow a\underline{X}$   
 $\Rightarrow a\underline{XX}$   
 $\Rightarrow ab\underline{X}$   
 $\Rightarrow ab\underline{XX}$   
 $\Rightarrow abb\underline{X}$   
 $\Rightarrow abb\underline{XX}$   
 $\Rightarrow abbb\underline{X}$   
 $= abbbb$

$S \Rightarrow Y\underline{X}$   
 $\Rightarrow YX\underline{X}$   
 $\Rightarrow YX\underline{b}$   
 $\Rightarrow YX\underline{X}b$   
 $\Rightarrow YX\underline{b}b$   
 $\Rightarrow YX\underline{X}bb$   
 $\Rightarrow YX\underline{b}bbb$   
 $\Rightarrow \underline{Y}bbbb$   
 $= abbbb$

### **A new format for FAs**

A class of machines (FAs) has been discussed accepting the regular language *i.e.* corresponding to a regular language there is a machine in this class, accepting that language and corresponding to a machine of this class there is a regular language accepted by this machine. It has also been discussed that there is a CFG corresponding to regular language and CFGs also define some nonregular languages, as well

There is a question whether there is a class of machines accepting the CFLs? The answer is yes. The new machines which are to be defined are more powerful and can be constructed with the help of FAs with new format.

To define the new format of an FA, some terms are defined in the next lecture.



## Theory of Automata

**Lecture N0. 37****Reading Material**Introduction to Computer Theory

## Chapter 14

**Summary**

New format for FAs, input TAPE, START, ACCEPT, REJECT, READ states Examples of New Format of FA, PUSH Down STACK, PUSH and POP, Example of PDA

**A new format for FAs**

A class of machines (FAs) has been discussed accepting the regular language *i.e.* corresponding to a regular language there is a machine in this class, accepting that language and corresponding to a machine of this class there is a regular language accepted by this machine. It has also been discussed that there is a CFG corresponding to regular language and CFGs also define some nonregular languages, as well. There is a question whether there is a class of machines accepting the CFLs? The answer is yes. The new machines which are to be defined are more powerful and can be constructed with the help of FAs with new format.

To define the new format of an FA, the following are to be defined

**Input TAPE**

The part of an FA, where the input string is placed before it is run, is called the input TAPE.

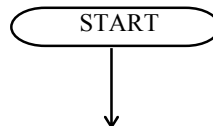
The input TAPE is supposed to accommodate all possible strings. The input TAPE is partitioned with cells, so that each letter of the input string can be placed in each cell. The input string abbaa is shown in the following input TAPE.

Cell i	Cell ii	Cell iii					
a	b	b	a	a	Δ	Δ	.

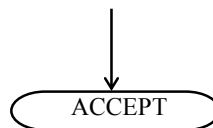
The character Δ indicates a blank in the TAPE. The input string is read from the TAPE starting from the cell (i). It is assumed that when first Δ is read, the rest of the TAPE is supposed to be blank.

**The START state**

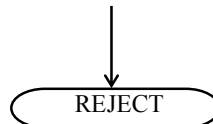
This state is like initial state of an FA and is represented by

**An ACCEPT state**

This state is like a final state of an FA and is expressed by

**A REJECT state**

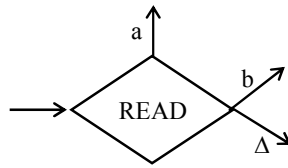
This state is like dead-end non final state and is expressed by



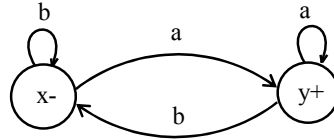
Note: It may be noted that the ACCEPT and REJECT states are called the halt states.

**A READ state**

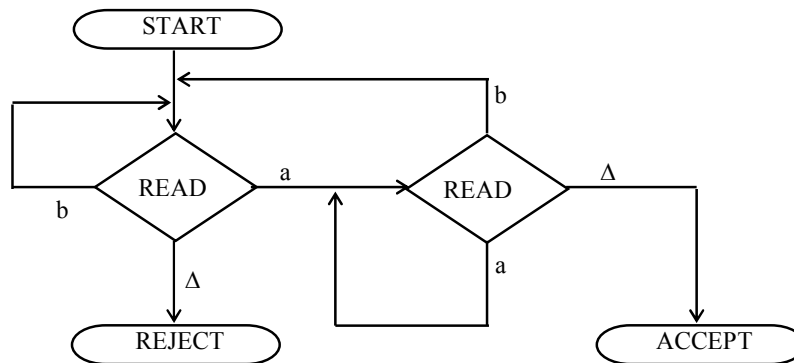
This state is to read an input letter and lead to some other state. The READ state is expressed by

**Example**

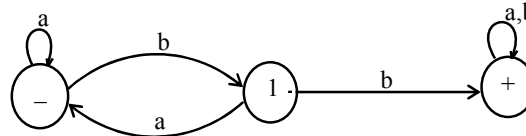
Before some other states are defined consider the following example of an FA along with its new format



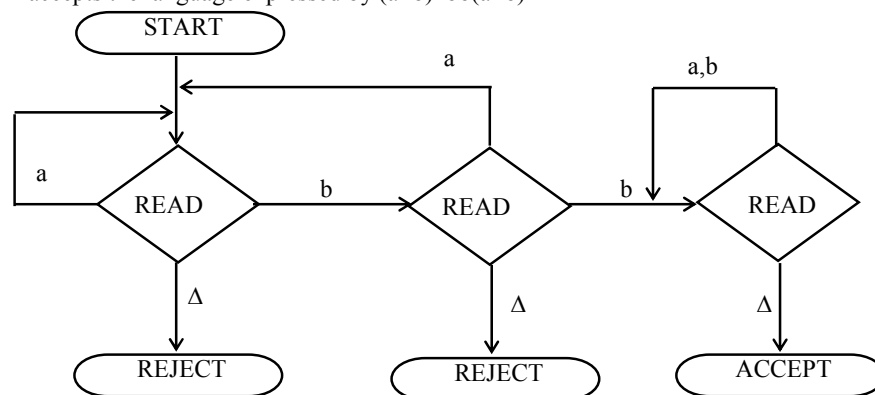
Obviously the above FA accepts the language of strings, expressed by  $(a+b)^*a$ . Following is the new format of the above FA

**Note**

The  $\Delta$  edge should not be confused with  $\wedge$ -labeled edge. The  $\Delta$ -edges start only from READ boxes and lead to halt states.

**Example**

The above FA accepts the language expressed by  $(a+b)^*bb(a+b)^*$



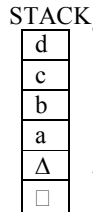
**PUSHDOWN STACK or PUSHDOWN STORE**

PUSHDOWN STACK is a place where the input letters can be placed until these letters are referred again. It can store as many letters as one can in a long column.

Initially the STACK is supposed to be empty *i.e.* each of its storage location contains a blank.

**PUSH**

A PUSH operator adds a new letter at the top of STACK, for *e.g.* if the letters a, b, c and d are pushed to the STACK that was initially blank, the STACK can be shown as



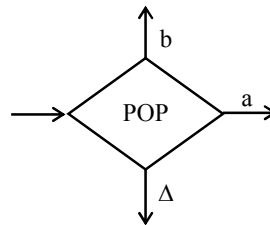
The PUSH state is expressed by



When a letter is pushed, it replaces the existing letter and pushes it one position below.

**POP and STACK**

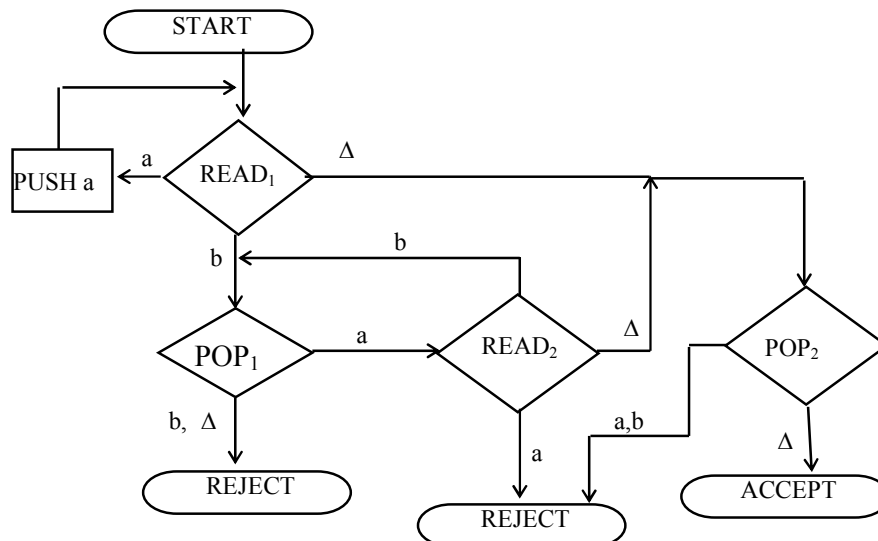
POP is an operation that takes out a letter from the top of the STACK. The rest of the letters are moved one location up. POP state is expressed as

**Note**

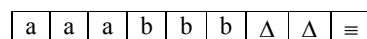
It may be noted that popping an empty STACK is like reading an empty TAPE, *i.e.* popping a blank character  $\Delta$ . It may also be noted that when the new format of an FA contains PUSH and POP states, it is called PUSHDOWN Automata or PDAs. It may be observed that if the PUSHDOWN STACK (the memory structure) is added to an FA then its language recognizing capabilities are increased considerably. Following is an example of PDA

**Example**

Consider the following PDA



The string aaabbb is to be run on this machine. Before the string is processed, the string is supposed to be placed on the TAPE and the STACK is supposed to be empty as shown below



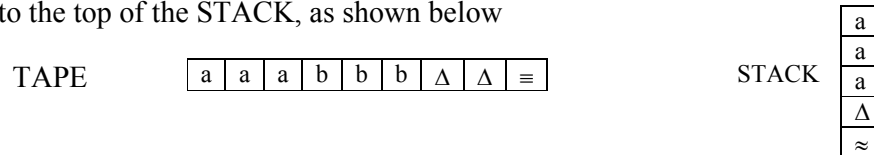
STACK



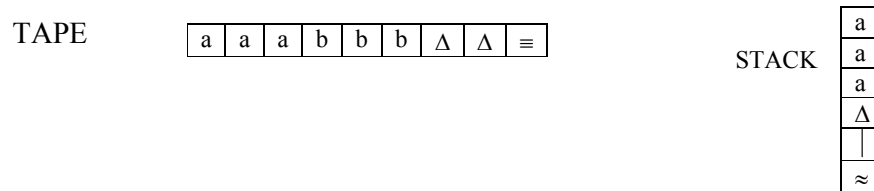
Reading first a from the TAPE we move from  $READ_1$  State to  $PUSH$  a state, it causes the letter a deleted from the TAPE and added to the top of the STACK, as shown below



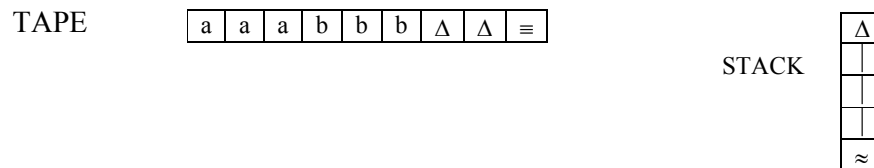
Reading next two a's successively, will delete further two a's from the TAPE and add these letters to the top of the STACK, as shown below



Then reading the next letter which is b from the TAPE will lead to the  $POP_1$  state. The top letter at the STACK is a, which is popped out and  $READ_2$  state is entered. Situation of TAPE and STACK is shown below



Reading the next two b's successively will delete two b's from the TAPE, will lead to the  $POP_1$  state and these b's will be removed from the STACK as shown below



Now there is only blank character  $\Delta$  left, to be read from the TAPE, which leads to  $POP_2$  state. While the only blank character is left in the STACK to be popped out and the ACCEPT state is entered, which shows that the string aaabbb is accepted by this PDA. It may be observed that the above PDA accepts the language

$\{a^n b^n : n = 0, 1, 2, \dots\}$ .

Since the null string is like a blank character, so to determine how the null string is accepted, it can be placed in the TAPE as shown below



Reading  $\Delta$  at state  $READ_1$  leads to  $POP_2$  state and  $POP_2$  state contains only  $\Delta$ , hence it leads to ACCEPT state and the null string is accepted.

Note: The process of running the string aaabbb can also be expressed in the table given in the next lecture.

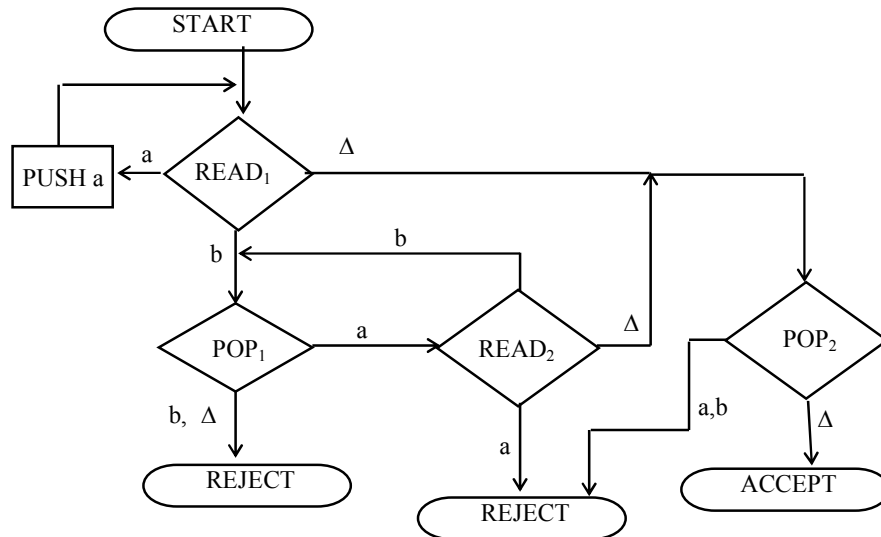
## Theory of Automata

**Lecture N0. 38****Reading Material**Introduction to Computer Theory

## Chapter 14

**Summary**

Example of PDA with table for running a string, Equivalent PDA, PDA for EVEN EVEN Language. Non-Derterministic PDA, Example of Non-Derterministic PDA, Definition of PUSH DOWN Automata, Example of Non-Derterministic PDA.

**Note**

The process of running the string aaabbb can also be expressed in the following table

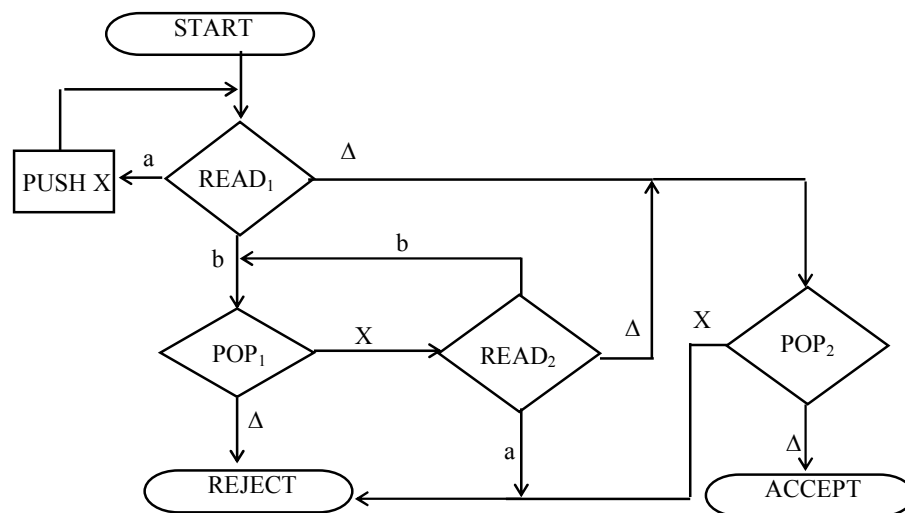
STATE	STACK	TAPE
START	$\Delta \dots$	<b>aaabbb</b> $\Delta \dots$
READ <sub>1</sub>	$\Delta \dots$	<u>a</u> aabbb $\Delta \dots$
PUSH a	a $\Delta \dots$	<u>aa</u> abbb $\Delta \dots$
READ <sub>1</sub>	a $\Delta \dots$	<u>aaa</u> bbb $\Delta \dots$
PUSH a	aa $\Delta \dots$	<u>aaaa</u> bb $\Delta \dots$
READ <sub>1</sub>	aa $\Delta \dots$	<u>aaaaa</u> b $\Delta \dots$
PUSH a	aaa $\Delta \dots$	<u>aaaaaa</u> $\Delta \dots$
READ <sub>1</sub>	aaa $\Delta \dots$	<u>aaaaaa</u> $\Delta \dots$
POP <sub>1</sub>	aa $\Delta \dots$	<u>aaaaa</u> b $\Delta \dots$

STATE	STACK	TAPE
READ <sub>1</sub>	aaΔ ...	<u>aaabbb</u> Δ ...
POP <sub>1</sub>	aΔ ...	<u>aaabbb</u> Δ ...
READ <sub>2</sub>	aΔ ...	<u>aaabbb</u> Δ ...
POP <sub>1</sub>	Δ ...	<u>aaabbb</u> Δ ...
READ <sub>2</sub>	Δ ...	<u>aaabbb</u> Δ ...
POP <sub>2</sub>	Δ ...	<u>aaabbb</u> Δ ...
ACCEPT	Δ ...	<u>aaabbb</u> Δ ...

It may be observed that the above PDA accepts the language  $\{a^n b^n : n=0,1,2,3, \dots\}$ .

#### Note

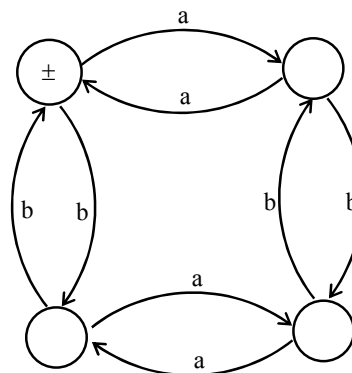
It may be noted that the TAPE alphabet  $\Sigma$  and STACK alphabet  $\Gamma$ , may be different in general and hence the PDA equivalent to that accepting  $\{a^n b^n : n=0,1,2,3, \dots\}$  discussed above may be



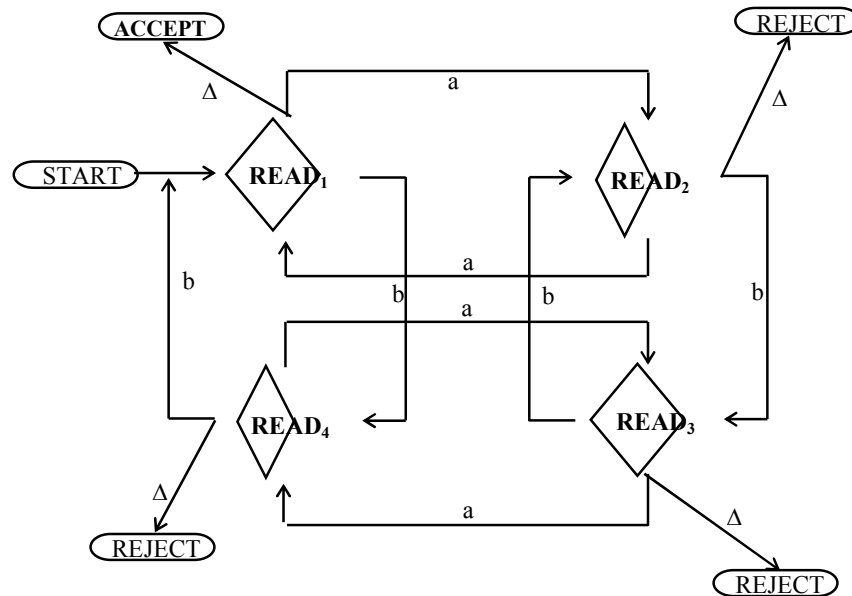
Following is an example of PDA corresponding to an FA

#### Example

Consider the following FA corresponding to the EVEN-EVEN language



The corresponding PDA will be



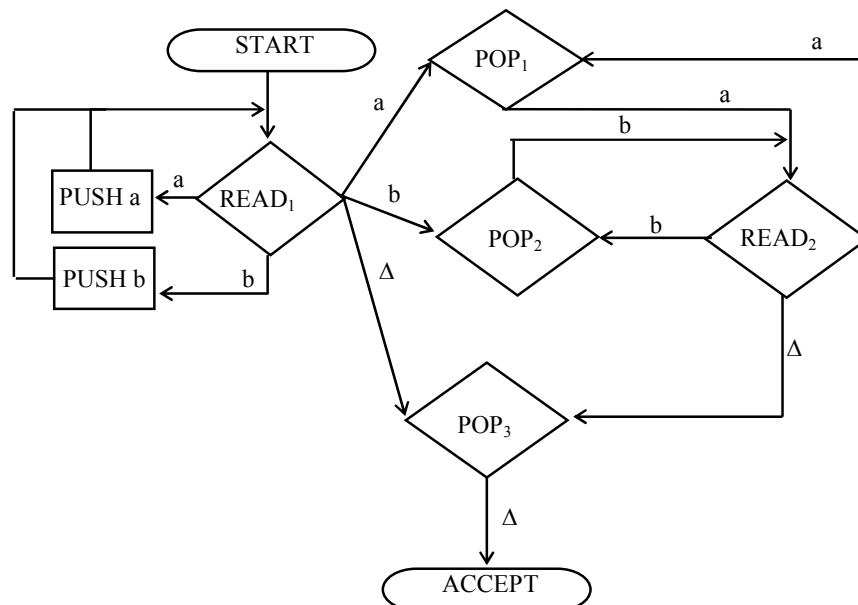
### Non-deterministic PDA

Like TGs and NFAs, if in a PDA there are more than one outgoing edges at READ or POP states with one label, then it creates nondeterminism and the PDA is called nondeterministic PDA.

In nondeterministic PDA no edge is labeled by string of terminals or nonterminals, like that can be observed in TGs. Also if there is no edge for any letter to be read from the TAPE, the machine crashes and the string is rejected.

In nondeterministic PDA a string may trace more than one path. If there exists at least one path traced by a string leading to ACCEPT state, then the string is supposed to be accepted, otherwise rejected.

Following is an example of nondeterministic PDA



Here the nondeterminism can be observed at state  $READ_1$ . It can be observed that the above PDA accepts the language

$$\begin{aligned} \text{EVENPALINDROME} &= \{w \text{ reverse}(w) : w \in \{a, b\}^*\} \\ &= \{\Lambda, aa, bb, aaaa, abba, baab, bbbb, \dots\} \end{aligned}$$

Now the definition of PDA including the possibility of nondeterminism may be given as follows

**PUSHDOWN AUTOMATON (PDA) including the possibility of non determinism**

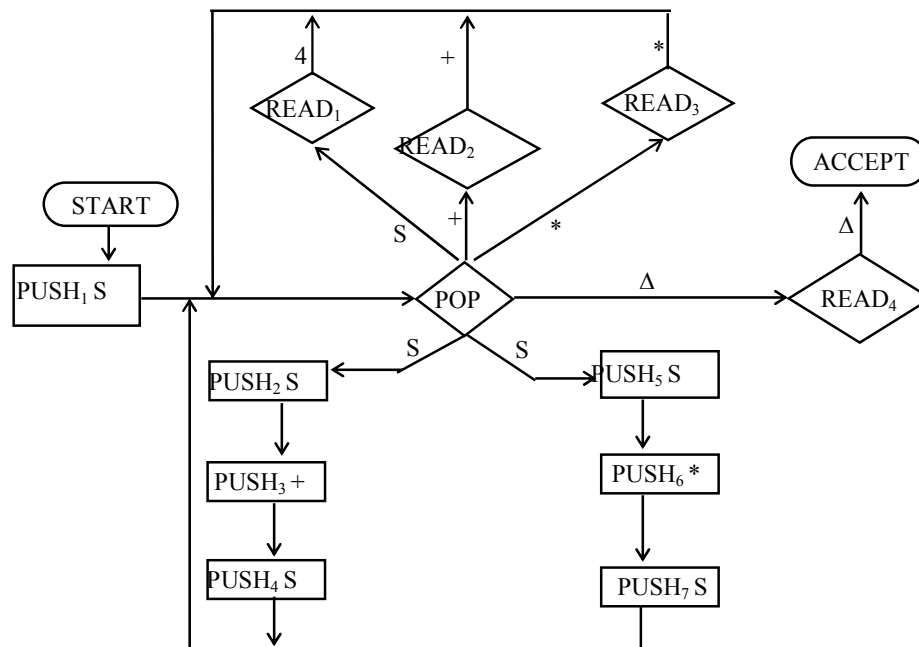
Pushdown Automaton (PDA), consists of the following

1. An alphabet  $\Sigma$  of input letters.
2. An input TAPE with infinite many locations in one direction. Initially the input string is placed in it starting from first cell, the remaining part of the TAPE is empty.
3. An alphabet  $\Gamma$  of STACK characters.
4. A pushdown STACK which is initially empty, with infinite many locations in one direction. Initially the STACK contains blanks.
5. One START state with only one out-edge and no in-edge.
6. Two halt states *i.e.* ACCEPT and REJECT states, with in-edges and no out-edges.
7. A PUSH state that introduces characters onto the top of the STACK.
8. A POP state that reads the top character of the STACK, (may contain more than one out-edges with same label).
9. A READ state that reads the next unused letter from the TAPE, (may contain more than one out-edges with same label).

Example: Consider the CFG

$$S \rightarrow S+S|S*S|4$$

Following is the PDA accepting the corresponding CFL



The string  $4 + 4 * 4$  traces the path shown in the following table

STATE	STACK	TAPE
START	$\Delta$	<b><math>4+4*4</math></b>
PUSH <sub>1</sub> S	<b>S</b>	<b><math>4+4*4</math></b>
POP	$\Delta$	<b><math>4+4*4</math></b>
PUSH <sub>2</sub> S	<b>S</b>	<b><math>4+4*4</math></b>
PUSH <sub>3</sub> +	<b>+S</b>	<b><math>4+4*4</math></b>
PUSH <sub>4</sub> S	<b>S+S</b>	<b><math>4+4*4</math></b>
POP	<b>+S</b>	<b><math>4+4*4</math></b>
READ <sub>1</sub>	<b>+S</b>	<b><math>+4*4</math></b>
POP	<b>S</b>	<b><math>+4*4</math></b>



STATE	STACK	TAPE
READ <sub>2</sub>	<b>S</b>	<b>4*4</b>
POP	$\Delta$	<b>4*4</b>
PUSH <sub>5</sub> S	<b>S</b>	<b>4*4</b>
PUSH <sub>6</sub> *	<b>*S</b>	<b>4*4</b>
PUSH <sub>7</sub> S	<b>S*S</b>	<b>4*4</b>
POP	<b>*S</b>	<b>4*4</b>
READ <sub>1</sub>	<b>*S</b>	<b>*4</b>
POP	<b>S</b>	<b>*4</b>
READ <sub>3</sub>	<b>S</b>	<b>4</b>
POP	$\Delta$	<b>4</b>
READ <sub>1</sub>	$\Delta$	$\Delta$
POP	$\Delta$	$\Delta$
READ <sub>4</sub>	$\Delta$	$\Delta$
ACCEPT	$\Delta$	$\Delta$

Note

It may be noted that the letters are deleted from the TAPE instead of underlined.

It may also be noted that the choice of path at POP state can be determined by the left most deviation of the string belonging to the CFL.

## Theory of Automata

### Lecture N0. 39

#### Reading Material

#### Introduction to Computer Theory

#### Chapter 15

#### Summary

PDA corresponding to CFG, Examples of PDA corresponding to CFG

#### PDA corresponding to CFG

##### Theorem

Corresponding to any CFG there exists a PDA accepting the language generated by the CFG.

Since an algorithm has already been discussed to convert the CFG in CNF, so the PDA can be constructed corresponding to the CFG. As the CFG in CNF generates all the nonnull words of the corresponding CFL, so accepting the null string (if it is contained in the CFL), can be managed separately.

##### Example

Consider the following CFG which is in CNF and does not generate the null string

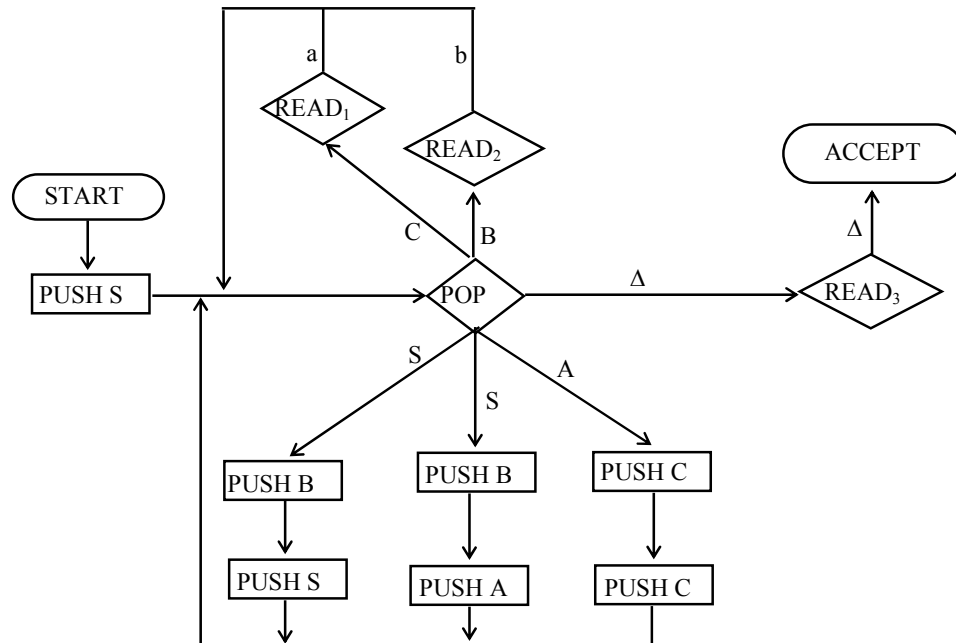
$S \rightarrow SB|AB$

$A \rightarrow CC$

$B \rightarrow b$

$C \rightarrow a$

The corresponding PDA will be



Here the STACK alphabet  $\Gamma = \{S, A, B, C\}$ , where the TAPE alphabet  $\Sigma = \{a, b\}$

**Note:** It may be noted that when the POP state is entered either a nonterminal is replaced by two nonterminals at the top of the STACK accommodating a production, or a nonterminal is popped out from the top of the stack and a READ state is entered to read a specified letter from the TAPE or else the machine crashes.

The choice of path taken at POP state to accommodate the word belonging to the CFL can be determined by the left most derivation of the word. Consider the word aab with its left most derivation, as follows

Working-String Generation	Production	Used
$S \Rightarrow AB$	$S \rightarrow AB$	step 1
$\Rightarrow CCB$	$A \rightarrow CC$	step 2
$\Rightarrow aCB$	$C \rightarrow a$	step 3
$\Rightarrow aaB$	$C \rightarrow a$	step 4
$\Rightarrow aab$	$B \rightarrow b$	step 5

First of all the START state is entered

STACK	TAPE
$\Delta \dots$	$aab\Delta \dots$

The PUSH S state is entered

STACK	TAPE
S	$aab\Delta \dots$

The POP state is entered and to accommodate the production  $S \rightarrow AB$ , PUSH B and PUSH A states are entered.

STACK	TAPE
AB	$aab\Delta \dots$

Then the POP state is entered and to accommodate the production  $A \rightarrow CC$ , PUSH C, PUSH C states are entered

STACK	TAPE
CCB	$aab$

The POP state is entered and to accommodate the production  $C \rightarrow a$ ,  $READ_1$  is entered and the letter a is read from the TAPE.

STACK	TAPE
CB	<u>a</u> ab

The POP state is entered and to accommodate the production  $C \rightarrow a$ ,  $READ_1$  state is entered and the letter a is read from the TAPE

STACK	TAPE
B	<u>a</u> ab

The POP state is entered and to accommodate the production  $B \rightarrow b$ ,  $READ_2$  state is entered and the letter b is read from the TAPE

STACK	TAPE
$\Delta$	<u>aab</u>

The  $\Delta$  shown in the STACK indicates that there are no nonterminals in the working string and  $\Delta$  is read from the STACK which leads to  $READ_3$  state where the  $\Delta$  is read from the TAPE and the ACCEPT state is entered which shows that the word aab is accepted by the PDA.

Following is the table showing all the observations discussed above, for the word aab

Left most derivation	STATE	STACK	TAPE
	START	$\Delta$	aab
S	PUSH S	S	aab
	POP	$\Delta$	aab
	PUSH B	B	aab
$\Rightarrow AB$	PUSH A	AB	aab
	POP	B	aab
	PUSH C	CB	aab
$\Rightarrow CCB$	PUSH C	CCB	aab
	POP	CB	aab
$\Rightarrow aCB$	READ <sub>1</sub>	CB	<u>a</u> ab
	POP	B	<u>a</u> ab
$\Rightarrow aaB$	READ <sub>1</sub>	B	<u>aa</u> b
	POP	$\Delta$	<u>aa</u> b
$\Rightarrow aab$	READ <sub>2</sub>	$\Delta$	<u>aab</u>
	POP	$\Delta$	<u>aab</u>
	READ <sub>3</sub>	$\Delta$	<u>aab</u>
	ACCEPT	$\Delta$	<u>aab</u>

Following is an example of building the PDA corresponding to the given CFG

Example

Consider the following CFG

$S \rightarrow XY$

$X \rightarrow aX \mid bX \mid a$

$Y \rightarrow Ya \mid Yb \mid a$

First of all, converting the CFG to be in CNF, introduce the nonterminals A and B as

$A \rightarrow a$

$B \rightarrow b$

The following CFG is in CNF

$S \rightarrow XY$

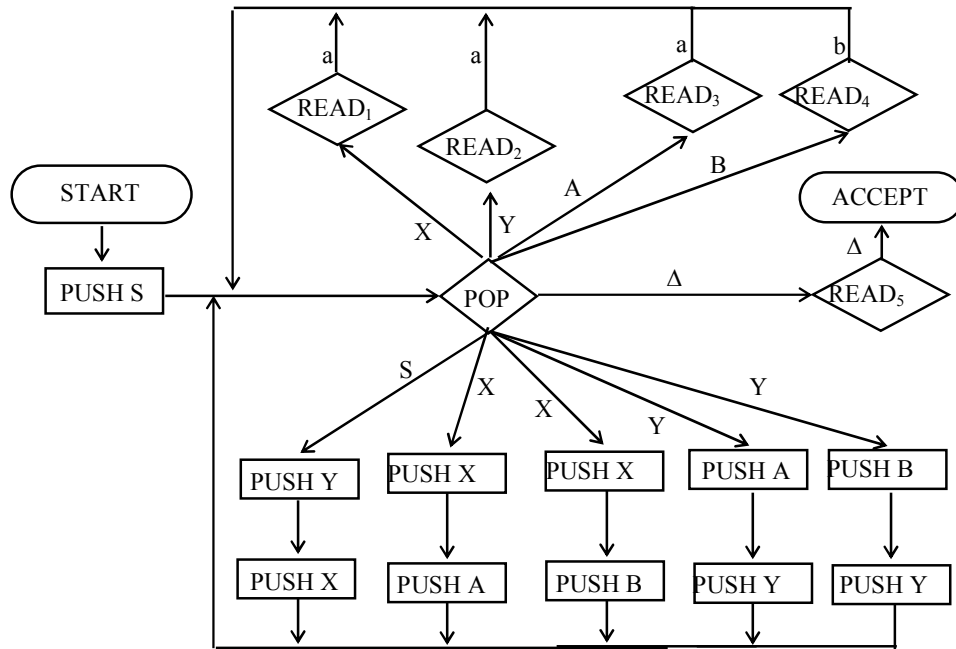
$X \rightarrow AX \mid BX \mid a$

$Y \rightarrow YA \mid YB \mid a$

$A \rightarrow a$

$B \rightarrow b$

The PDA corresponding to the above CFG will be



The word aaab can be generated as

Working-String Generation

$S \Rightarrow XY$   
 $\Rightarrow AXY$   
 $\Rightarrow aXY$   
 $\Rightarrow aaY$   
 $\Rightarrow aaYB$   
 $\Rightarrow aaaB$   
 $\Rightarrow aaab$

Production

$S \rightarrow XY$   
 $X \rightarrow AX$   
 $A \rightarrow a$   
 $X \rightarrow a$   
 $Y \rightarrow YB$   
 $Y \rightarrow a$   
 $B \rightarrow b$

Used

step 1  
 step 2  
 step 3  
 step 4  
 step 5  
 step 6  
 step 7

STACK	TAPE	STACK	TAPE
(START) $\Delta$	aaab	(POP) Y	<u>aa</u> ab
(PUSH S) S	aaab	(READ <sub>1</sub> ) Y	<u>aa</u> ab
(POP) $\Delta$	aaab	(POP) $\Delta$	<u>aa</u> bb
(PUSH Y) Y	aaab	(PUSH B) B	<u>aa</u> bb
(PUSH X) XY	aaab	(PUSH Y) YB	<u>aa</u> bb
(POP) Y	aaab	(POP) B	<u>aa</u> ab
(PUSH X) XY	aaab	(READ <sub>2</sub> ) B	<u>aa</u> ab
(PUSH A) AXY	aaab	(POP) $\Delta$	<u>aa</u> ab
(POP) XY	aaab	(READ <sub>4</sub> ) $\Delta$	<u>aa</u> ab
(READ <sub>3</sub> ) XY	<u>aa</u> ab	(POP) $\Delta$	$\Delta$

## Theory of Automata

**Lecture NO. 40****Reading Material**Introduction to Computer Theory

## Chapter 15

**Summary**

Recap of example of PDA corresponding to CFG, CFG corresponding to PDA. Theorem, HERE state, Definition of Conversion form, different situations of PDA to be converted into conversion form

Example

Consider the following CFG

$$S \rightarrow XY$$

$$X \rightarrow aX \mid bX \mid a$$

$$Y \rightarrow Ya \mid Yb \mid a$$

First of all, converting the CFG to be in CNF, introduce the nonterminals A and B as

$$A \rightarrow a$$

$$B \rightarrow b$$

The following CFG is in CNF

$$S \rightarrow XY$$

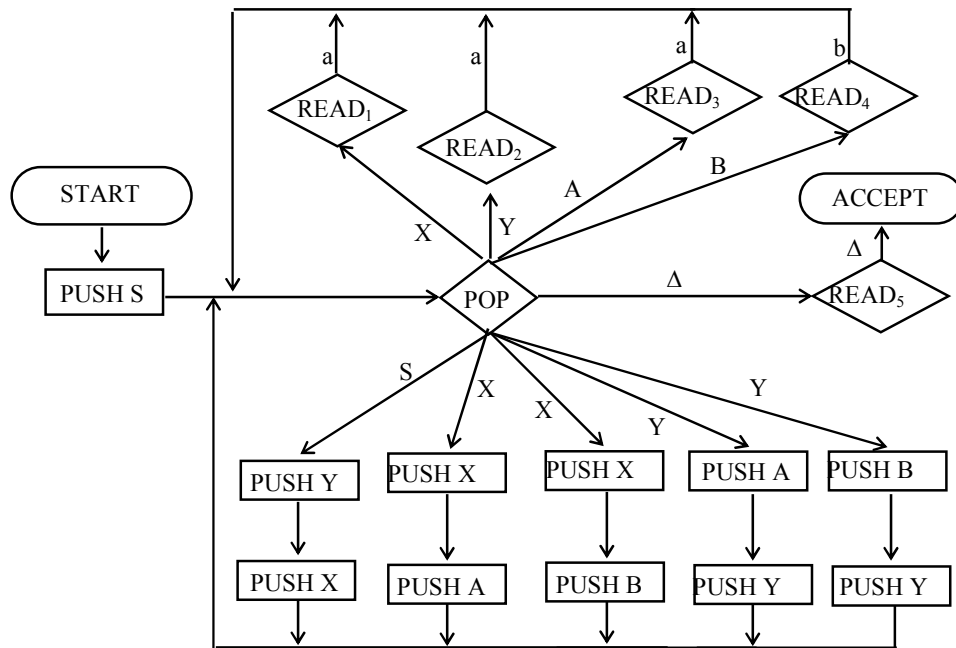
$$X \rightarrow AX \mid BX \mid a$$

$$Y \rightarrow YA \mid YB \mid a$$

$$A \rightarrow a$$

$$B \rightarrow b$$

The PDA corresponding to the above CFG will be

**Theorem**

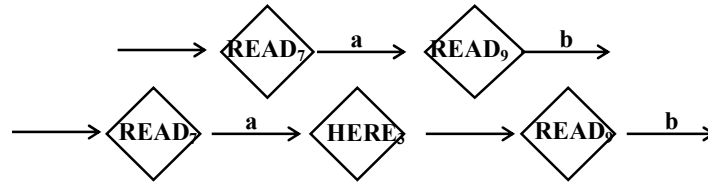
Given a PDA that accepts the language  $L$ , there exists a CFG that generates exactly  $L$ .

Before the CFG corresponding to the given PDA is determined, the PDA is converted into the **standard form** which is called the **conversion form**.

Before the PDA is converted into conversion form a new state HERE is defined which is placed in the middle of any edge.

Like READ and POP states, HERE states are also numbered e.g.

becomes



### Conversion form of PDA

#### Definition

A PDA is in conversion form if it fulfills the following conditions:

There is only one ACCEPT state.

There are no REJECT states.

Every READ or HERE is followed immediately by a POP *i.e.* every edge leading out of any READ or HERE state goes directly into a POP state.

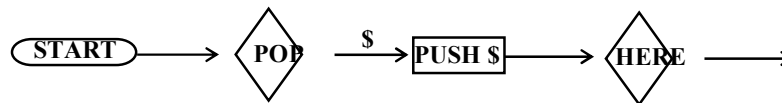
No two POPs exist in a row on the same path without a READ or HERE between them whether or not there are any intervening PUSH states (*i.e.* the POP states must be separated by READs or HEREs).

All branching, deterministic or nondeterministic occurs at READ or HERE states, none at POP states and every edge has only one label.

Even before we get to START, a “bottom of STACK” symbol \$ is placed on the STACK. If this symbol is ever popped in the processing it must be replaced immediately. The STACK is never popped beneath this symbol.

Right before entering ACCEPT this symbol is popped out and left.

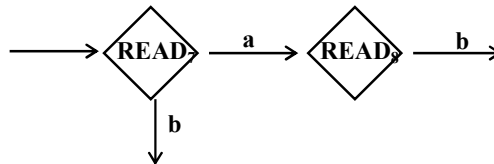
The PDA must begin with the sequence



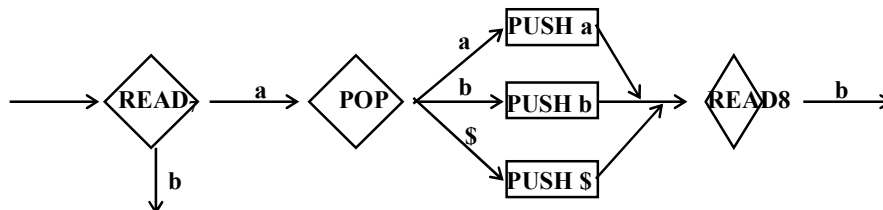
The entire input string must be read before the machine can accept the word.

Different situations of a PDA to be converted into **conversion form** are discussed as follows

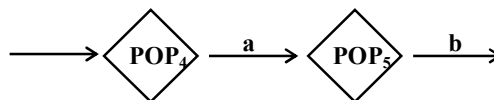
To satisfy condition 3,



becomes



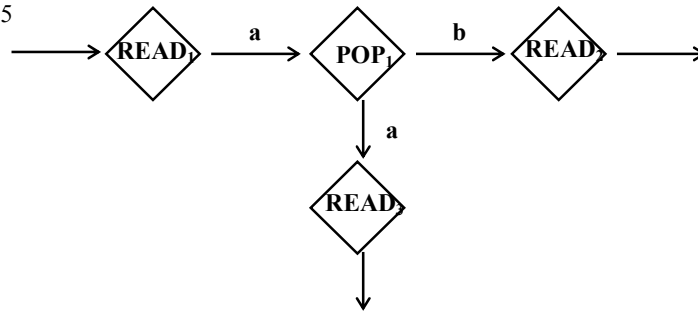
To satisfy condition 4,



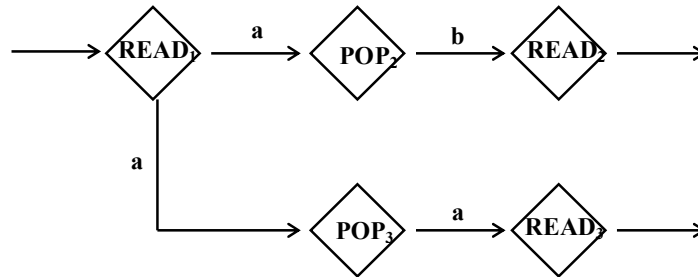
becomes



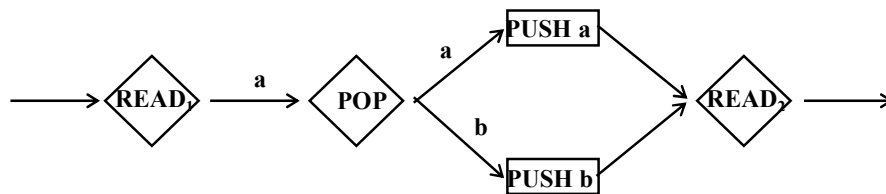
To satisfy condition 5



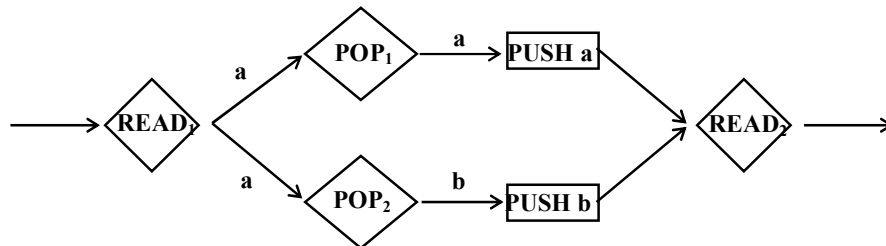
becomes



To satisfy condition 5

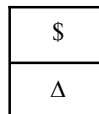


becomes



To satisfy condition 6, it is supposed that the STACK is initially in the position shown below

STACK





## Theory of Automata

**Lecture N0. 41****Reading Material**Introduction to Computer Theory

## Chapter 15

**Summary**

Recap of PDA in conversion form, example of PDA in conversion form, joints of the machine, new pictorial representation of PDA in conversion form, summary table, row sequence, row language.

**Conversion form of PDA**Definition

A PDA is in conversion form if it fulfills the following conditions:

There is only one ACCEPT state.

There are no REJECT states.

Every READ or HERE is followed immediately by a POP *i.e.* every edge leading out of any READ or HERE state goes directly into a POP state.

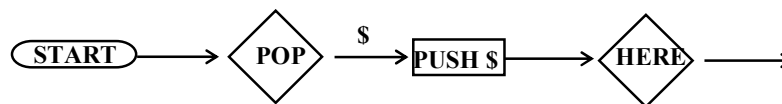
No two POPs exist in a row on the same path without a READ or HERE between them whether or not there are any intervening PUSH states (*i.e.* the POP states must be separated by READs or HEREs).

All branching, deterministic or nondeterministic occurs at READ or HERE states, none at POP states and every edge has only one label.

Even before we get to START, a “bottom of STACK” symbol \$ is placed on the STACK. If this symbol is ever popped in the processing it must be replaced immediately. The STACK is never popped beneath this symbol.

Right before entering ACCEPT this symbol is popped out and left.

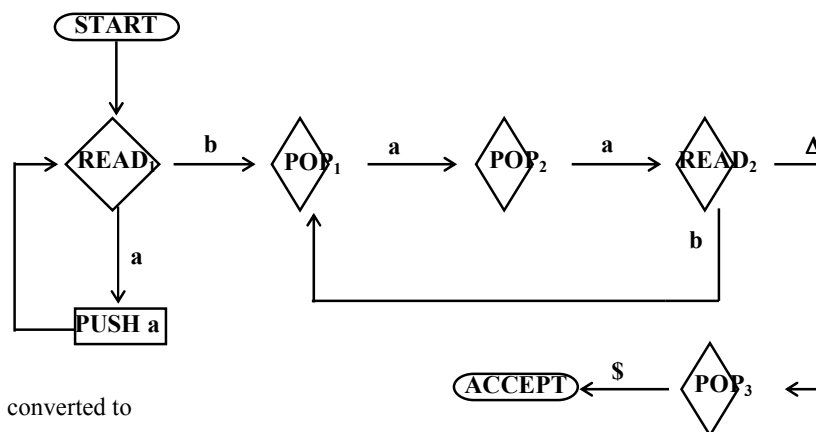
The PDA must begin with the sequence



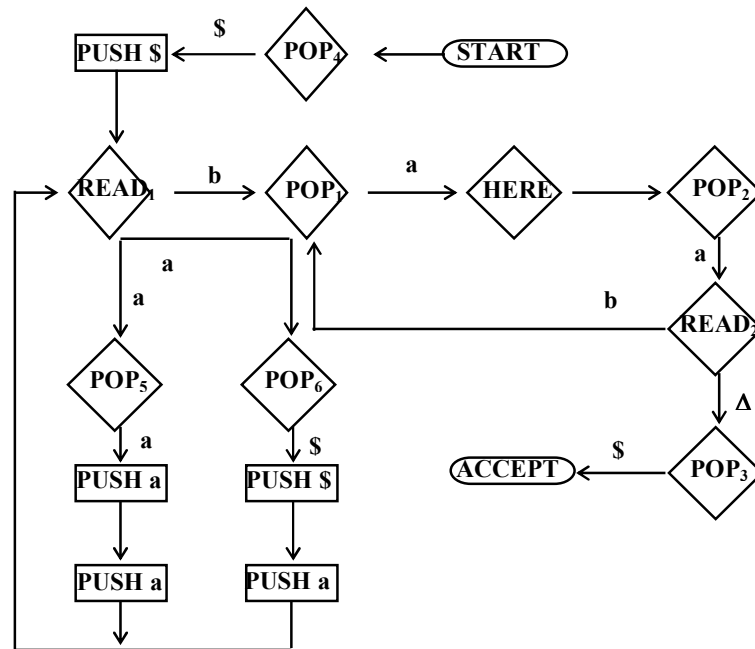
The entire input string must be read before the machine can accept the word.

Example

Consider the following PDA accepting the language  $\{a^{2n}b^n : n = 1, 2, 3, \dots\}$



Which may be converted to



The above PDA accepts exactly the same language

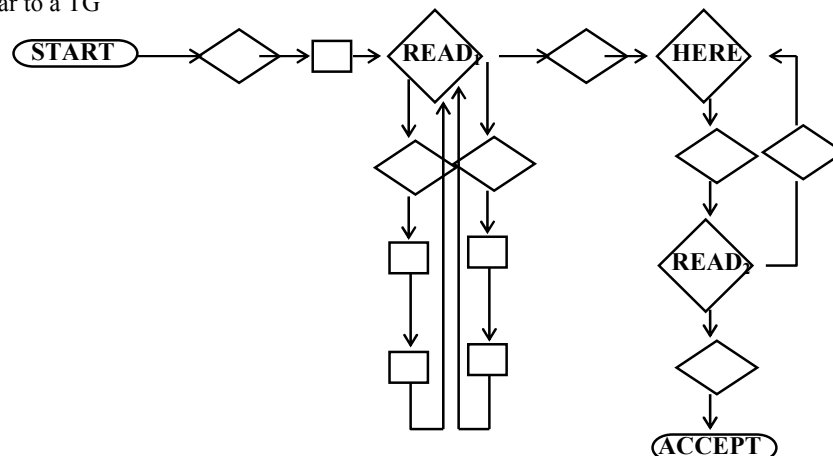
#### Note

It may be noted that any PDA which is in **conversion form** can be considered to be the collection of path segments, where each path segment is of the following form

FROM	TO	READ	POP	PUSH
START or READ or HERE	READ or HERE or ACCEPT	ONE or no input letter	Exactly one STACK character	Any string onto the STACK

START, READ, HERE and ACCEPT states are called the **joints of the machine**. Between two consecutive joints on a path exactly one character is popped and any number of characters can be pushed.

The PDA which is in the **conversion form** can be supposed to be the set of joints with path segments in between, similar to a TG



The above entire machine can be described as a list of all joint-to-joint path segments, called summary table. The PDA converted to the **conversion form** has the following summary table

FROM Where	TO Where	READ What	POP What	PUSH What	ROW Number
START	READ <sub>1</sub>	$\Lambda$	\$	\$	1
READ <sub>1</sub>	READ <sub>1</sub>	a	\$	a\$	2
READ <sub>1</sub>	READ <sub>1</sub>	a	a	aa	3
READ <sub>1</sub>	HERE	b	a	--	4
HERE	READ <sub>2</sub>	$\Lambda$	a	--	5
READ <sub>2</sub>	HERE	b	a	--	6
READ <sub>2</sub>	AT	$\Delta$	\$	--	7

Consider the word aaaabb. This word is accepted by the above PDA through the following path

START–POP<sub>4</sub>–PUSH\$–READ<sub>1</sub>–POP<sub>6</sub>–PUSH \$–PUSH a–READ<sub>1</sub>–POP<sub>5</sub>–PUSH a–PUSH a–READ<sub>1</sub>–POP<sub>5</sub>–PUSH a–PUSH a–READ<sub>1</sub>–POP<sub>5</sub>–PUSH a–PUSH a–READ<sub>1</sub>–POP<sub>1</sub>–HERE–POP<sub>2</sub>–READ<sub>2</sub>–POP<sub>1</sub>–HERE–POP<sub>2</sub>–READ<sub>2</sub>–POP<sub>3</sub>–ACCEPT.

The above path can also be expressed by the following path in terms of sequence of rows

Row<sub>1</sub> –Row<sub>2</sub> –Row<sub>3</sub> –Row<sub>3</sub> –Row<sub>3</sub> –Row<sub>4</sub> –Row<sub>5</sub> –Row<sub>6</sub> –Row<sub>5</sub> –Row<sub>7</sub>

It can be observed that the above path is not only **joint-to-joint consistent** but **STACK consistent** as well.

It may be noted that in FAs, paths correspond to strings of letters, while in PDAs, paths correspond to strings of rows from the summary table.

#### Note

It may be noted that since the HERE state reads nothing from the TAPE, therefore  $\Lambda$  is kept in the **READ what** column.

It may also be noted that the summary table contains all the information of the PDA which is in the pictorial representation. Every path through the PDA is a sequence of rows of the summary table. However, not every sequence of rows from the summary table represents a viable path, *i.e.* every sequence of rows may not be **STACK consistent**.

It is very important to determine which sequences of rows do correspond to possible paths through the PDA, because the paths are directly related to the language accepted, *e.g.* Row<sub>4</sub> cannot be immediately followed by Row<sub>6</sub> because Row<sub>4</sub> leaves in HERE, while Row<sub>6</sub> begins in Read<sub>2</sub>. Some information must be kept about the STACK before rows are concatenated.

To represent a path, a sequence of rows must be **joint-consistent** (the rows meet up end to end) and **STACK-consistent** (when a row pops a character it should be there at the top of the STACK).

The next target is to define **row language** whose alphabet is  $\Sigma = \{\text{Row}_1, \text{Row}_2, \dots, \text{Row}_7\}$  *i.e.* the alphabet consists of the letters which are the names of the rows in the summary table.

#### Note

It may be noted that the words of the row language trace **joint-to-joint** and **STACK consistent** paths, which shows that all the words of this language begin with Row<sub>1</sub> and end in Row<sub>7</sub>. Consider the following row sequence Row<sub>5</sub> Row<sub>5</sub> Row<sub>3</sub> Row<sub>6</sub>

This is string of 4 letters, but not word of the row language because

It does not represent a path starting from START and ending in ACCEPT state.

It is not joint consistent.

It is not STACK consistent.

Before the CFG that generates the language accepted by the given PDA, is determined, the CFG that generates the **row language** is to be determined. For this purpose new nonterminals are to be introduced that contain the information needed to ensure **joint** and **STACK consistency**.

It is not needed to maintain any information about what characters are read from the TAPE.

## Theory of Automata

**Lecture N0. 42****Reading Material**Introduction to Computer Theory

## Chapter 15

**Summary**

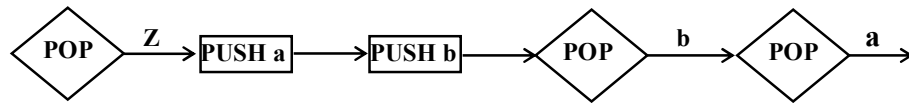
Row language, nonterminals defined from summary table, productions defined by rows, rules for defining productions, all possible productions of CFG for row language of the example under consideration, CFG corresponding to the given PDA

**Note**

As has already been discussed that the Row language is the language whose alphabet

$\Sigma = \{\text{Row}_1, \text{Row}_2, \dots, \text{Row}_7\}$ , for the example under consideration, so to determine the CFG of Row language, the nonterminals of this CFG are introduced in the form **Net(X, Y, Z)**

where X and Y are joints and Z is any STACK character. Following is an example of Net(X, Y, Z)



If the above is the path segment between two joints then, the net STACK effect is same as POP Z.

For a given PDA, some sets of all possible sentences Net(X, Y, Z) are true, while other are false. For this purpose every row of the summary table is examined whether the net effect of popping is exactly one letter.

Consider the Row<sub>4</sub> of the summary table developed for the PDA of the language  $\{a^{2^n}b^n\}$

FROM Where	TO Where	READ What	POP What	PUSH What	ROW Number
READ <sub>1</sub>	HERE	b	a	--	4

The nonterminal corresponding to the above row may be written as Net (READ<sub>1</sub>, HERE, a) *i.e.* Row<sub>4</sub> is a single Net row.

Consider the following row from an arbitrary summary table

FROM Where	TO Where	READ What	POP What	PUSH What	ROW Number
READ <sub>9</sub>	READ <sub>3</sub>	b	b	abb	11

which shows that Row<sub>11</sub> is not Net style sentence because the trip from READ<sub>9</sub> to READ<sub>3</sub> does not pop one letter from the STACK, while it adds two letters to the STACK. However Row<sub>11</sub> can be concatenated with some other Net style sentences *e.g.* Row<sub>11</sub>Net(READ<sub>3</sub>, READ<sub>7</sub>, a)Net(READ<sub>7</sub>, READ<sub>1</sub>, b)Net(READ<sub>1</sub>, READ<sub>8</sub>, b)

Which gives the nonterminal

Net(READ<sub>9</sub>, READ<sub>8</sub>, b), now the whole process can be written as

Net(READ<sub>9</sub>, READ<sub>8</sub>, b) → Row<sub>11</sub>Net(READ<sub>3</sub>, READ<sub>7</sub>, a) Net(READ<sub>7</sub>, READ<sub>1</sub>, b)Net(READ<sub>1</sub>, READ<sub>8</sub>, b)

Which will be a production in the CFG of the corresponding **row language**.

In general to create productions from rows of summary table, consider the following row in certain summary table

FROM Where	TO Where	READ What	POP What	PUSH What	ROW Number
READ <sub>x</sub>	READ <sub>y</sub>	u	w	<b>m<sub>1</sub>m<sub>2</sub>...m<sub>n</sub></b>	i

then for any sequence of joint states S<sub>1</sub>, S<sub>2</sub>, ...S<sub>n</sub>, the production in the row language can be included as

$\text{Net}(\text{READ}_x, S_n, w) \rightarrow \text{Row}_i \text{Net}(\text{READ}_y, S_1, m_1) \dots \text{Net}(S_{n-1}, S_n, m_n)$

It may be noted that in CFG, in general, replacing a nonterminal with string of some other nonterminals does not always lead to a word in the corresponding CFL e.g.  $S \rightarrow X|Y$ ,  $X \rightarrow ab$ ,  $Y \rightarrow aYY$

Here  $Y \rightarrow aYY$  does not lead to any word of the language.

Following are the three rules of defining all possible productions of CFG of the row language

The trip starting from START state and ending in ACCEPT state with the NET style

$\text{Net}(\text{START}, \text{ACCEPT}, \$)$  gives the production of the form  $S \rightarrow \text{Net}(\text{START}, \text{ACCEPT}, \$)$

From the summary table the row of the following form

FROM Where	TO Where	READ What	POP What	PUSH What	ROW Number
X	Y	anything	z	--	i

Defines the productions of the form  $\text{Net}(X, Y, z) \rightarrow \text{Row}_i$

For each row that pushes string of characters on to the STACK of the form

FROM Where	TO Where	READ What	POP What	PUSH What	ROW Number
$\text{READ}_x$	$\text{READ}_y$	u	w	$m_1 m_2 \dots m_n$	i

then for any sequence of joint states  $S_1, S_2, \dots, S_n$ , the production in the row language can be included as

$\text{Net}(\text{READ}_x, S_n, w) \rightarrow \text{Row}_i \text{Net}(\text{READ}_y, S_1, m_1) \dots \text{Net}(S_{n-1}, S_n, m_n)$

It may be noted that this rule introduces new productions. It does not mean that each production of the form Nonterminal  $\rightarrow$  string of nonterminals, helps in defining some word of the language.

#### Note

Considering the example of PDA accepting the language  $\{a^{2n}b^n : n=1, 2, 3, \dots\}$ , using rule1, rule2 and rule3 the possible productions for the CFG of the row language are

$S \rightarrow \text{Net}(\text{START}, \text{ACCEPT}, \$)$

$\text{Net}(\text{READ}_1, \text{HERE}, a) \rightarrow \text{Row}_4$

$\text{Net}(\text{HERE}, \text{READ}_2, a) \rightarrow \text{Row}_5$

$\text{Net}(\text{READ}_2, \text{HERE}, a) \rightarrow \text{Row}_6$

$\text{Net}(\text{READ}_2, \text{ACCEPT}, \$) \rightarrow \text{Row}_7$

$\text{Net}(\text{START}, \text{READ}_1, \$) \rightarrow \text{Row}_1 \text{Net}(\text{READ}_1, \text{READ}_1, \$)$

$\text{Net}(\text{START}, \text{READ}_2, \$) \rightarrow \text{Row}_1 \text{Net}(\text{READ}_1, \text{READ}_2, \$)$

$\text{Net}(\text{START}, \text{HERE}, \$) \rightarrow \text{Row}_1 \text{Net}(\text{READ}_1, \text{HERE}, \$)$

$\text{Net}(\text{START}, \text{ACCEPT}, \$) \rightarrow \text{Row}_1 \text{Net}(\text{READ}_1, \text{ACCEPT}, \$)$

$\text{Net}(\text{READ}_1, \text{READ}_1, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{READ}_1, a) \text{Net}(\text{READ}_1, \text{READ}_1, \$)$

$\text{Net}(\text{READ}_1, \text{READ}_1, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{READ}_2, a) \text{Net}(\text{READ}_2, \text{READ}_1, \$)$

$\text{Net}(\text{READ}_1, \text{READ}_1, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{HERE}, a) \text{Net}(\text{HERE}, \text{READ}_1, \$)$

$\text{Net}(\text{READ}_1, \text{READ}_2, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{READ}_1, a) \text{Net}(\text{READ}_1, \text{READ}_2, \$)$

$\text{Net}(\text{READ}_1, \text{READ}_2, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{READ}_2, a) \text{Net}(\text{READ}_2, \text{READ}_2, \$)$

$\text{Net}(\text{READ}_1, \text{READ}_2, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{HERE}, a) \text{Net}(\text{HERE}, \text{READ}_2, \$)$

$\text{Net}(\text{READ}_1, \text{HERE}, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{READ}_1, a) \text{Net}(\text{READ}_1, \text{HERE}, \$)$

$\text{Net}(\text{READ}_1, \text{HERE}, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{READ}_2, a) \text{Net}(\text{READ}_2, \text{HERE}, \$)$

$\text{Net}(\text{READ}_1, \text{HERE}, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{HERE}, a) \text{Net}(\text{HERE}, \text{HERE}, \$)$

$\text{Net}(\text{READ}_1, \text{ACCEPT}, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{READ}_1, a) \text{Net}(\text{READ}_1, \text{ACCEPT}, \$)$

$\text{Net}(\text{READ}_1, \text{ACCEPT}, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{READ}_2, a) \text{Net}(\text{READ}_2, \text{ACCEPT}, \$)$

$\text{Net}(\text{READ}_1, \text{ACCEPT}, \$) \rightarrow \text{Row}_2 \text{Net}(\text{READ}_1, \text{HERE}, a) \text{Net}(\text{HERE}, \text{ACCEPT}, \$)$

$\text{Net}(\text{READ}_1, \text{READ}_1, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{READ}_1, a) \text{Net}(\text{READ}_1, \text{READ}_1, a)$

$\text{Net}(\text{READ}_1, \text{READ}_1, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{READ}_2, a) \text{Net}(\text{READ}_2, \text{READ}_1, a)$

$\text{Net}(\text{READ}_1, \text{READ}_1, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{HERE}, a) \text{Net}(\text{HERE}, \text{READ}_1, a)$

$\text{Net}(\text{READ}_1, \text{READ}_2, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{READ}_1, a) \text{Net}(\text{READ}_1, \text{READ}_2, a)$

$\text{Net}(\text{READ}_1, \text{READ}_2, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{READ}_2, a) \text{Net}(\text{READ}_2, \text{READ}_2, a)$

$\text{Net}(\text{READ}_1, \text{READ}_2, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{HERE}, a) \text{Net}(\text{HERE}, \text{READ}_2, a)$

$\text{Net}(\text{READ}_1, \text{HERE}, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{READ}_1, a) \text{Net}(\text{READ}_1, \text{HERE}, a)$   
 $\text{Net}(\text{READ}_1, \text{HERE}, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{READ}_2, a) \text{Net}(\text{READ}_2, \text{HERE}, a)$   
 $\text{Net}(\text{READ}_1, \text{HERE}, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{HERE}, a) \text{Net}(\text{HERE}, \text{HERE}, a)$   
 $\text{Net}(\text{READ}_1, \text{ACCEPT}, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{READ}_1, a) \text{Net}(\text{READ}_1, \text{ACCEPT}, a)$   
 $\text{Net}(\text{READ}_1, \text{ACCEPT}, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{READ}_2, a) \text{Net}(\text{READ}_2, \text{ACCEPT}, a)$   
 $\text{Net}(\text{READ}_1, \text{ACCEPT}, a) \rightarrow \text{Row}_3 \text{Net}(\text{READ}_1, \text{HERE}, a) \text{Net}(\text{HERE}, \text{ACCEPT}, a)$

Following is a left most derivation of a word of row language

$S \Rightarrow \text{Net}(\text{START}, \text{ACCEPT}, \$) \quad \dots \quad \text{using 1}$   
 $\quad \Rightarrow \text{Row}_1 \text{Net}(\text{READ}_1, \text{ACCEPT}, \$) \quad \dots \quad \text{using 9}$   
 $\quad \Rightarrow \text{Row}_1 \text{Row}_2 \text{Net}(\text{RD}_1, \text{RD}_2, a) \text{Net}(\text{RD}_2, \text{AT}, \$) \quad \dots \quad \text{using 20}$   
 $\quad \Rightarrow \text{Row}_1 \text{Row}_2 \text{Row}_3 \text{Net}(\text{RD}_1, \text{HERE}, a) \text{Net}(\text{RD}_2, \text{HERE}, a) \text{Net}(\text{RD}_2, \text{AT}, \$) \dots \quad \text{using 27}$   
 $\Rightarrow \text{Row}_1 \text{Row}_2 \text{Row}_3 \text{Row}_4 \text{Net}(\text{HERE}, \text{RD}_2, a) \text{Net}(\text{RD}_2, \text{ACCEPT}, \$) \quad \dots \quad \text{using 2}$   
 $\Rightarrow \text{Row}_1 \text{Row}_2 \text{Row}_3 \text{Row}_4 \text{Row}_5 \text{Net}(\text{HERE}, \text{ACCEPT}, \$) \quad \dots \quad \text{using 3}$   
 $\Rightarrow \text{Row}_1 \text{Row}_2 \text{Row}_3 \text{Row}_4 \text{Row}_5 \text{Row}_7 \quad \dots \quad \text{using 5}$

Which is the shortest word in the whole row language.

It can be observed that each left most derivation generates the sequence of rows of the summary table, which are both **joint-** and **STACK- consistent**.

Note: So far the rules have been defined to create all possible productions for the CFG of the row language.

Since in each row in the summary table, the READ column contains  $\Lambda$  and  $\Delta$  in addition to the letters of the alphabet of the language accepted by the PDA, so each word of the row language generates the word of the language accepted by the given PDA.

Thus the following rule 4 helps in completing the CFG corresponding to the given PDA

Each row of the summary table defines a production of the form  $\text{Row}_i \rightarrow a$  where in  $\text{Row}_i$  the READ column consists of letter  $a$ .

Application of rule 4 to the summary table for the PDA accepting  $\{a^{2n}b^n : n=1,2,3,\dots\}$  under consideration adds the following productions

$\text{Row}_1 \rightarrow \Lambda$   
 $\text{Row}_2 \rightarrow a$   
 $\text{Row}_3 \rightarrow a$   
 $\text{Row}_4 \rightarrow b$   
 $\text{Row}_5 \rightarrow \Lambda$   
 $\text{Row}_6 \rightarrow b$   
 $\text{Row}_7 \rightarrow \Delta$

Which shows that the word  $\text{Row}_1 \text{Row}_2 \text{Row}_3 \text{Row}_4 \text{Row}_5 \text{Row}_7$  of the row language is converted to  $\Lambda aab \Lambda \Delta = abb$

## Theory of Automata

**Lecture N0. 43****Reading Material**Introduction to Computer Theory

## Chapter 16

**Summary**

Non-Context-Free-languages, Live Production, Dead Production, Theorem, self-embedded nonterminal, Pumping lemma for CFLs, Examples

**Non-Context-Free language**

There arises a question, whether all languages are CFL? The answer is no.

Languages which are not Context-Free, are called Non-CFL.

To prove the claim that all languages are not Context-Free, the study of machines of word production from the grammar is needed

Live production: A production of the form nonterminal  $\rightarrow$  string of two nonterminals is called a live production.

Dead production: A production of the form nonterminal  $\rightarrow$  terminal is called a dead production.

It may be noted that every CFG in CNF has only these types of productions.

**Theorem**

If a CFG is in CNF and if there is restriction to use the live production at most once each, then only the finite many words can be generated.

It may be noted that every time a live production is applied during the derivation of a word it increases the number of nonterminals by one.

Similarly applying dead production decreases the nonterminals by one. Which shows that to generate a word, one more dead production are applied than the live productions *e.g.*

$$\begin{aligned} S &\Rightarrow XY \\ &\Rightarrow aY \\ &\Rightarrow aa \end{aligned}$$

Here one live and two dead productions are used.

In general, if a CFG in CNF has  $p$  live and  $q$  dead productions then all words generated without repeating any live production have at most  $(p+1)$  letters.

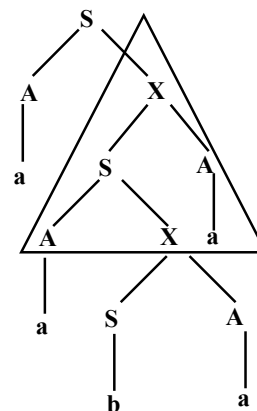
**Theorem**

If a CFG is in CNF with  $p$  live and  $q$  dead productions and if  $w$  is 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 descendant of the first  $z$ .

It can be observed from the above theorem that generation tree of word  $w$  has more than  $p$  rows.

**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, as shown in figure aside



Here the nonterminal X is self-embedded.

Note

Consider the following CFG in CNF

$S \rightarrow AB$

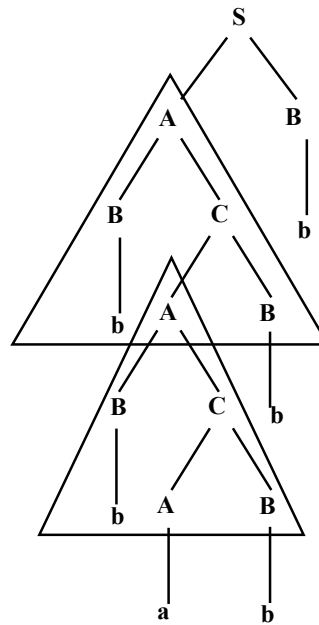
$A \rightarrow BC$

$C \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

and the derivation tree of the word bbabbb



Note

The part of tree enclosed in upper triangle is identical to that enclosed in lower triangle, there is still another option of replacing A by the same sequence of production shown in lower triangle.

The above fact provides the following pumping lemma for the CFLs.

**Pumping lemma for CFLs**

Theorem

If G is any CFG in CNF with p live productions, then every word w of length more than  $2^p$  can be partitioned into five substrings as  $w = uvxyz$ , where x is not null string and v and y are not both null string.

Then all the words of the form  $uv^nxy^n z$ ,  $n = 1, 2, 3, \dots$  can also be generated by G.

Example

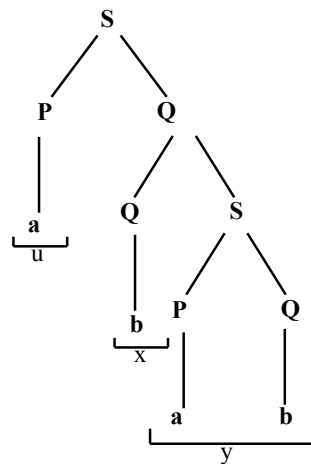
Consider the following CFG which is in CNF

$S \rightarrow PQ$

$Q \rightarrow QS|b$

$P \rightarrow a$

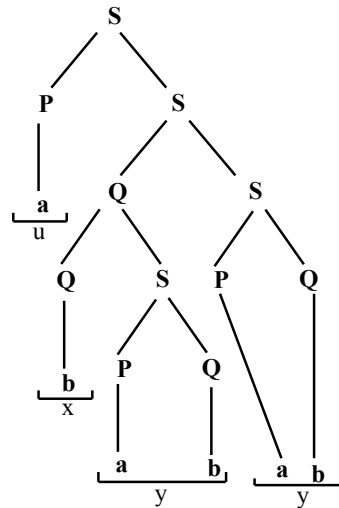
and a word abab generated by the above CFG with the following derivation tree





Then  $w$  can be broken up as  $w = uvxyz$  where  $u = a$ ,  $v = \Lambda$ ,  $x = b$ ,  $y = ab$ ,  $z = \Lambda$

Repeating the triangle from the second  $Q$  just as it descends from the first  $Q$ , the corresponding tree may be expressed as follows



Which shows that  $uvvxyyz = a\Lambda\Lambda babab\Lambda = ababab$  belongs to the language generated by the given CFG.

So, it can be generalized that words of the form  $uv^nxy^n z$ ,  $n=1,2,3,\dots$  belong to the language generated by the given CFG.

#### Note

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.

#### Example

Consider the language

$L = \{a^n b^n c^n : n=1,2,3,\dots\}$ , let the language  $L$  be Context Free language and 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.

#### Note

It can be observed that no matter what choices are made for the substrings  $u, v, x, y$  and  $z$ ,  $uv^2xy^2z$  can't belong to  $L$ , as all the words in  $a^n b^n c^n$  have

Only one substring  $ab$

Only one substring  $bc$

No substring  $ac$

No substring  $ba$

No substring  $ca$

No substring  $cb$

For any  $n=1,2,3,\dots$

The above observations shows that if  $v$  or  $y$  is not single letter or  $\Lambda$ , then  $uv^2xy^2z$  may contain either two or more substrings  $ab$  or  $bc$  or one or more substrings  $ac$  or  $ba$  or  $ca$  or  $cb$  i.e. these strings may be in the number more than the number they are supposed to be.

Moreover, if  $v$  and  $y$  are either single letter or  $\Lambda$ , then one or two of letters  $a, b, c$  will be increased, where as the other letter will not be increased in  $uv^2xy^2z$ , which shows  $uv^2xy^2z$  does not belong to  $L$ .

Thus pumping lemma is not satisfied. Hence  $L$  is non CFL.

It may be noted that the pumping lemma discussed for infinite regular language  $L$ , **the word  $w$  can be decomposed into three parts  $w=xyz$ , such that all words of the form  $xy^nz$ ,  $n=1,2,3,\dots$ , belong to  $L$ .**

Similarly, the pumping lemma discussed for CFLs can also stated as

**If  $w$  is a large enough word in a CF: then,  $w$  can be decomposed into  $w=uvxyz$  such that all words of the form  $uv^nx^nzy$  belong to  $L$**

It may be noted that proof of pumping lemma for regular languages needed that path of word  $w$  to be so large enough so that it contains a circuit and circuit can be looped as many times as one can. The proof of the pumping lemma for CFLs needs the derivation for  $w$  to be so large that it contains a sequence of productions that can be repeated as many times as one can.

Moreover, the pumping lemma for regular languages does not hold for non regular language as that language does not contain both  $xyz$  and  $xyyz$ .

Similarly pumping lemma for CFLs does not hold for non-CFL as that language does not contain both  $uvxyz$  and  $uvvxyyz$ .

There is another difference between the pumping lemma for regular languages and that for CFLs that first one acts on machines while other acts on algebraic structures *i.e.* grammar.

To achieve full power the pumping lemma for regular languages has modified by pumping lemma version II. Similarly, full power for pumping lemma for CFLs is achieved by stating the following theorem

#### Theorem

If  $L$  is a CFL in CNF with  $p$  live productions then any word  $W$  in  $L$  of length more than  $2^p$  can be decomposed as  $w=uvxyz$  s.t.  $\text{length}(vxy) \leq 2^p$ ,  $\text{length}(x) > 0$ ,  $\text{length}(v)+\text{length}(y) > 0$  then the words of the form  $uv^nx^nzy$  :  $n=1,2,3,\dots$  belong to  $L$ .

#### Example

Consider the language

$$L = \{a^n b^m a^n b^m : m, n = 1, 2, 3, \dots\}$$

$$= \{abab, aabaab, abbabb, aabbaabb, aaabaaab, \dots\}$$

The first version of pumping lemma for CFLs may be satisfied by  $L$ , but to apply the second version of pumping lemma to  $L$ , let  $L$  be generated by CFG which is in CNF and has  $p$  live productions.

Consider the word decomposing  $w$  into  $uvxyz$  where  $\text{length}(vxy) < 2^p$  which shows that  $v$  and  $y$  can't be single letters separated by clumps of other letter because the separator letter is longer than the length of whole substring  $vxy$ , which shows that  $uvvxyyz$  is not contained in  $L$ . Thus pumping lemma is not satisfied and  $L$  is non CFL.

#### Example

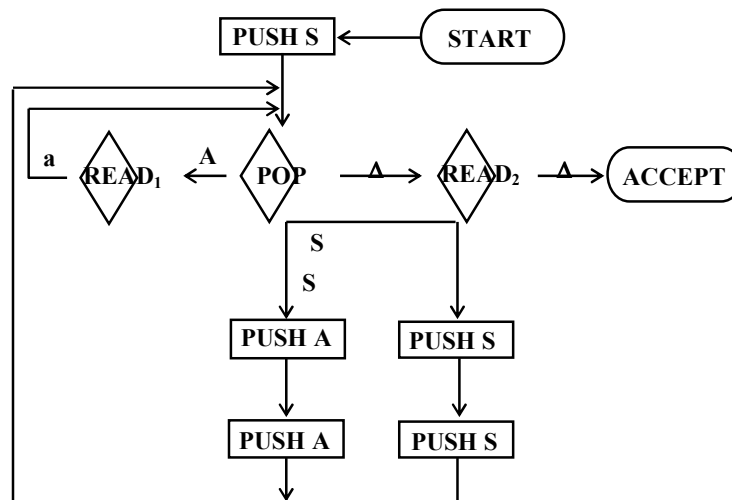
Consider the language EVENA *i.e.*

$$\text{EVENA} = (aa)^n = a^{2n} = \{aa, aaaa, aaaaaa, \dots\}$$

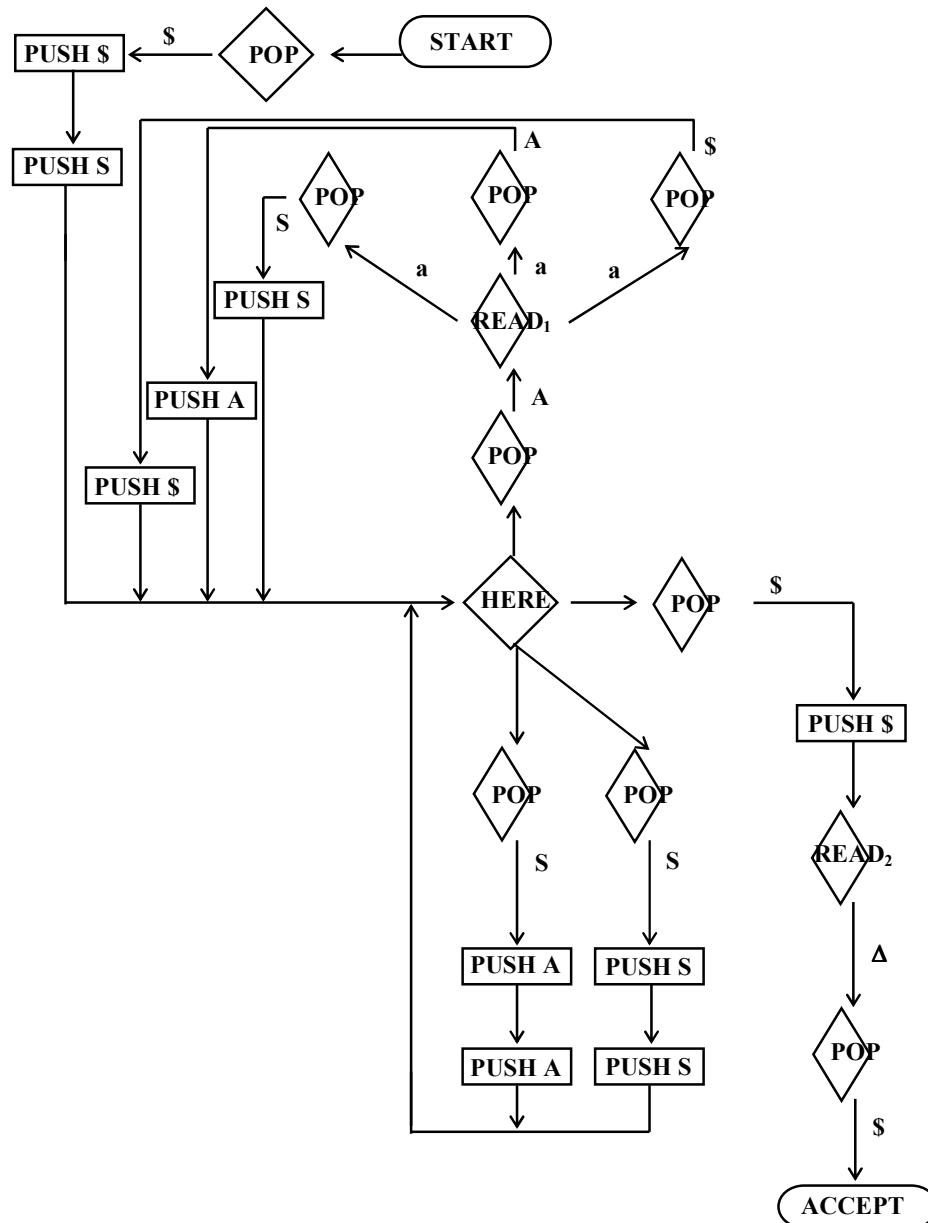
The grammar for this language must be

$$S \rightarrow SS|aa \text{ and its CNF will be}$$

$$S \rightarrow SS|AA, A \rightarrow a, \text{ the PDA for this grammar will be as under}$$



Its corresponding **conversion form** will be



The summary table corresponding to the above PDA in **conversion form** can be expressed as

FROM	TO	READ	POP	PUSH	ROW
START	HERE	$\Lambda$	\$	SS	1
HERE	HERE	$\Lambda$	S	SS	2
HERE	HERE	$\Lambda$	S	AA	3
HERE	READ <sub>1</sub>	$\Lambda$	A	--	4
READ <sub>1</sub>	HERE	a	S	S	5
RAED <sub>1</sub>	HERE	a	\$	\$	6
READ <sub>1</sub>	HERE	a	A	A	7
HERE	READ <sub>2</sub>	$\Lambda$	\$	\$	8
READ <sub>2</sub>	ACCEPT	$\Delta$	\$	--	9

Following are the productions defined from the summary table

$S \rightarrow \text{Net}(\text{START}, \text{ACCEPT}, \$)$   
 $\text{Net}(\text{HERE}, \text{READ}_1, A) \rightarrow \text{Row}_4$   
 $\text{Net}(\text{READ}_2, \text{ACCEPT}, \$) \rightarrow \text{Row}_9$   
 $\text{Net}(\text{START}, X, \$) \rightarrow \text{Row}_1 \text{Net}(\text{HERE}, Y, S) \text{Net}(Y, X, \$)$   
 $\text{Net}(\text{HERE}, X, S) \rightarrow \text{Row}_2 \text{Net}(\text{HERE}, Y, S) \text{Net}(Y, X, S)$   
 $\text{Net}(\text{START}, X, S) \rightarrow \text{Row}_3 \text{Net}(\text{HERE}, Y, A) \text{Net}(Y, X, A)$   
 $\text{Net}(\text{READ}_1, X, S) \rightarrow \text{Row}_5 \text{Net}(\text{HERE}, X, S)$  gives four productions  
 $\text{Net}(\text{READ}_1, X, \$) \rightarrow \text{Row}_6 \text{Net}(\text{HERE}, X, \$)$  gives four productions  
 $\text{Net}(\text{READ}_1, X, A) \rightarrow \text{Row}_7 \text{Net}(\text{HERE}, X, A)$  gives four productions  
 $\text{Net}(\text{HERE}, \text{ACCEPT}, \$) \rightarrow \text{Row}_8 \text{Net}(\text{READ}_2, \text{ACCEPT}, \$)$

Where X and Y are the corresponding joints

In addition to 44 productions following 9 productions complete the required CFG

$\text{Row}_1 \rightarrow \Lambda$   
 $\text{Row}_2 \rightarrow \Lambda$   
 $\text{Row}_3 \rightarrow \Lambda$   
 $\text{Row}_4 \rightarrow \Lambda$   
 $\text{Row}_5 \rightarrow a$   
 $\text{Row}_6 \rightarrow a$   
 $\text{Row}_7 \rightarrow a$   
 $\text{Row}_8 \rightarrow \Lambda$   
 $\text{Row}_9 \rightarrow \Lambda$

## Theory of Automata

**Lecture N0. 44****Reading Material**Introduction to Computer Theory

## Chapter 18

**Summary**

Decidability, whether a CFG generates certain string (emptiness), examples, whether a nonterminal is used in the derivation of some word (uselessness), examples, whether a CFL is finite (finiteness), example, whether the given string is generated by the given CFG (membership), example, parsing techniques, top down parsing, example

**Decidability**

Following are the decidable problems w.r.t. CFG

Whether or not the given CFG generates any word? Problem of emptiness of CFL.

Whether or not the given CFG generates the finite language? Problem of finiteness.

Whether or not the given string  $w$  can be generated by the given CFG? Problem of membership.

Following are algorithms showing that the answers to the above three questions are yes.

**Algorithm 1 (Emptiness)**

If the given CFG contains a production of the form  $S \rightarrow A$ , then obviously the corresponding CFL is not empty.

If the CFG contains the production of the form  $S \rightarrow t$ , where  $t$  is a terminal or string of terminal then  $t$  is a word of the corresponding CFL and CFL is not empty.

If the CFG contains no such production then

For each nonterminal  $N$  with  $N \rightarrow t$ , pick one production for  $N$  (if there are more than one) and replace  $N$  by  $t$  in the right side of each production wherever it lies. Remove all such productions from the CFG. Doing so the CFG will be changed, it will generate atleast one word of the old CFL.

Repeat the process until either it eliminates  $S$  or no new nonterminal is eliminated.

If  $S$  has been eliminated then CFG generates some words otherwise not.

**Example**

$S \rightarrow AB, A \rightarrow BSB, B \rightarrow CC$

$C \rightarrow SS$

$A \rightarrow a|b$

$C \rightarrow b|bb$

Step (1). Picking  $A \rightarrow a, C \rightarrow b$ , it can be written as

$S \rightarrow aB$

$A \rightarrow BSB$

$A \rightarrow bb$

$B \rightarrow aaS$

$B \rightarrow bb$

$C \rightarrow SS$

Step (1). Picking  $B \rightarrow bb$  and  $A \rightarrow bb$ , it can be written as

$S \rightarrow abb$

$A \rightarrow bbSbb$

$B \rightarrow aaS$

$C \rightarrow SS$

Since  $S \rightarrow abb$  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, following algorithm is used

**Algorithm 2 (Uselessness)**

Find all unproductive nonterminals (the nonterminal is unproductive if it cannot produce a string of terminals).

Eliminate all productions involving unproductive nonterminals.

Paint all X's blue.

If any nonterminal is in the left side of the production with any blue nonterminal in the right side, paint that nonterminal blue and paint that nonterminal blue at all occurrences of it throughout the grammar.

Repeat step 4 until no new nonterminal is painted.

If S is blue then X is useful member of CFG, otherwise not.

#### Example

Consider the following CFG

$$S \rightarrow Aba \mid bAZ \mid b$$

$$A \rightarrow Xb \mid bZa$$

$$B \rightarrow bAA$$

$$X \rightarrow aZa|aaa$$

$$Z \rightarrow ZAbA$$

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.

$$S \rightarrow Aba|b$$

$$A \rightarrow Xb$$

$$B \rightarrow bAA$$

$$X \rightarrow aaa$$

X is blue, so A is blue. Thus B and S are also blue. Since S is blue so X can be used to generate certain word from the given CFG.

Note: It may be noted that a nonterminal is called useless if it cannot be used in a production of some word. Following algorithm is used to determine whether the given CFG generate the finite language

#### Algorithm 3 (Finiteness)

Determine all useless nonterminals and eliminate all productions involving these nonterminals.

For each of the remaining nonterminals, determine whether they are self-embedded (using the following steps).

Stop if a self-embedded nonterminal is discovered.

To test whether X is self-embedded

Change all X's on the left side of the productions into a Greek letter  $\Psi$  and keep all X's on the right side as such.

Paint all X's blue.

If Y is any nonterminal on the left side of the production with X in the right side, then paint Y blue.

Repeat step (c) until no new nonterminal is painted.

If  $\Psi$  is painted, then the X is self-embedded, otherwise not.

If any nonterminal, left in the grammar, after step 1, is self-embedded then the language generated is infinite, otherwise finite.

#### Example

Consider the CFG

$$S \rightarrow ABa|bAZ|b$$

$$A \rightarrow Xb|bZa$$

$$B \rightarrow bAA$$

$$X \rightarrow aZa|bA|aaa$$

$$Z \rightarrow ZAbA$$

Here the nonterminal Z is useless, while all other are used in the derivation of some word. So eliminating the productions involving Z

$$S \rightarrow ABa|b$$

$$A \rightarrow Xb$$

$$B \rightarrow bAA$$

$$X \rightarrow bA|aaa$$

Starting with nonterminal X. Replacing X on left side of the production by  $\Psi$

$$S \rightarrow ABa|b$$

$$A \rightarrow Xb$$

$$B \rightarrow bAA$$

$$\Psi \rightarrow bA|aaa$$

X is blue so A is blue and so  $\Psi$  is blue. Since A is blue, so B is blue and so S is blue. Since  $\Psi$  is blue so X is self-embedded and hence the CFG generates the infinite language.

To determine whether a string is generated by the given CFG, following algorithm is used

Algorithm 4 (The CYK algorithm)

This algorithm was invented by John Cocke and later was published by Tandao Kasami and Daniel H. Younger. Convert the given CFG in CNF.

Let the string  $x$  under consideration has the form  $x = x_1x_2x_3 \dots x_n$  where all  $x_i$ s may not be different. List all the nonterminals in the given CFG, say,  $S, N_1, N_2, \dots$

List the nonterminals that generates single letter substrings of  $x$  *i.e.*

Substring	All producing nonterminals
$x_1$	$N \dots$
$x_2$	$N \dots$
$x_3$	$\partial$
$\partial$	$\partial$
$x_n$	$N \dots$

List the nonterminals that generates substrings of length 2 *i.e.*

Substring	All producing nonterminals
$x_1 x_2$	$N \dots$
$x_2 x_3$	$N \dots$
$x_3 x_4$	$\partial$
$\partial$	$\partial$
$x_{n-1} x_n$	$N \dots$

Similarly, list of nonterminals generating substring of  $x$  of length 3

Substring	All producing nonterminals
$x_1 x_2 x_3$	$N \dots$
$x_2 x_3 x_4$	$N \dots$
$x_3 x_4 x_5$	$\partial$
$\partial$	$\partial$
$x_{n-2} x_{n-1} x_n$	$N \dots$

Continuing the process, the nonterminals that generate  $x_1x_2x_3 \dots x_n$  can be determined as

Substring	All producing nonterminals
$x_1 x_2 x_3 \dots x_n$	$N \dots$

If  $S$  is among the set of all producing nonterminals, then  $x$  can be generated by the CFG, otherwise not.

**Example**

Consider the following CFG in CNF

$S \rightarrow AA$

$A \rightarrow AA$

$A \rightarrow a$

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

Substring	All producing nonterminals
$x_1 = a$	A
$x_2 = a$	A
$x_3 = a$	A
$x_1 x_2$	S, A
$x_2 x_3$	S, A
$x = x_1 x_2 x_3$	S, A

Since S is in the list of producing nonterminals, so  $aaa$  can be generated by the given CFG.

**Parsing Techniques**

Recall the CFG for arithmetic expression

$S \rightarrow S+S | S*S | \text{number}$

It was observed that the word  $3+4*5$  created ambiguity by considering its value either 23 or 35. To remove this ambiguity, the CFG was modified to

$S \rightarrow (S+S) | (S*S) | \text{number}$

There arises a question that whether a new CFG can be defined without having parentheses with operator hierarchy (*i.e.* \* before +)? The answer is yes. Following is the required PLUS-TIMES grammar

$S \rightarrow E, E \rightarrow T+E | T, T \rightarrow F*T | F, F \rightarrow (E) | i$

Where  $i$  stands for any identifier *i.e.* number or of storage location name (variable). Following is the derivation of  $i+i*i$

$S \Rightarrow E$   
 $\Rightarrow T+E$   
 $\Rightarrow F+E$   
 $\Rightarrow i+E$   
 $\Rightarrow i+T$   
 $\Rightarrow i+F*T$   
 $\Rightarrow i+i*T$   
 $\Rightarrow i+i*F$   
 $\Rightarrow i+i*i$

**Parsing of word****Definition**

The process of finding the derivation of word generated by particular grammar is called **parsing**.

There are different parsing techniques, containing the following three

Top down parsing.

Bottom up parsing.

Parsing technique for particular grammar of arithmetic expression.

**Top down parsing**

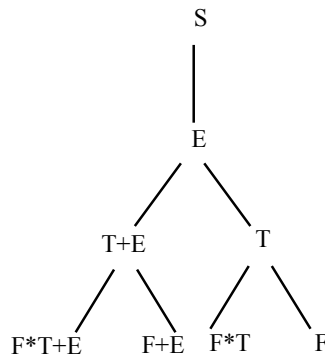
Following is an example showing top down parsing technique

**Example**

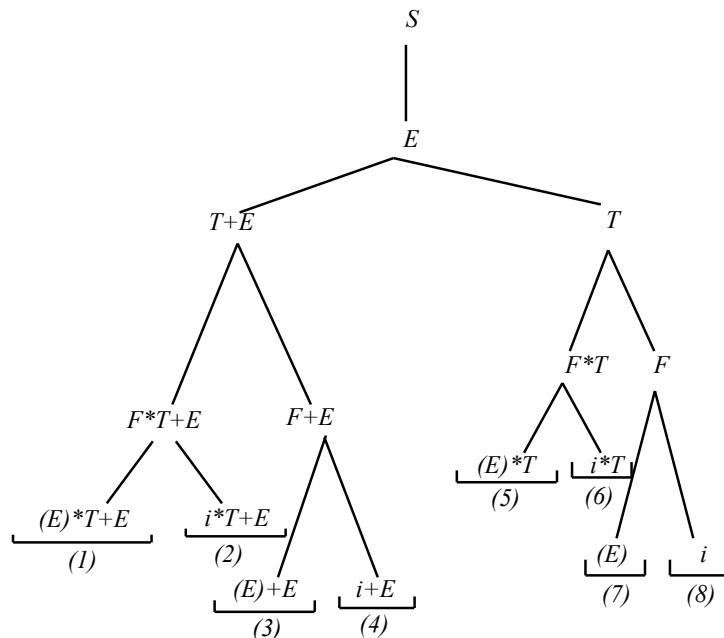
Consider PLUS-TIMES grammar and a word  $i+i*i$ .



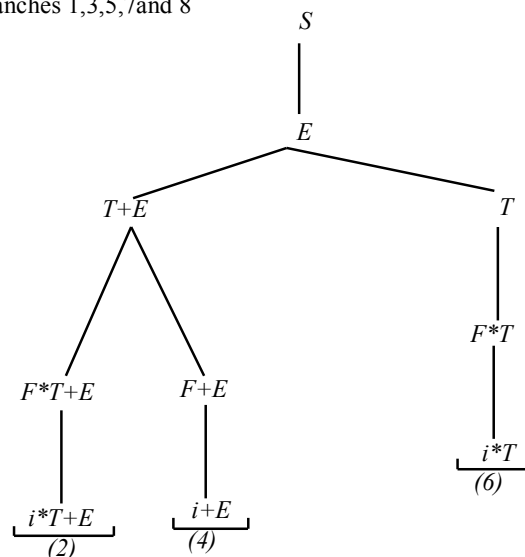
As can be observed from the name of top down parsing, the parsing starts from the nonterminal  $S$  and the structure similar to that of total language tree is developed. The branches of the tree are extended till the required word is found as a branch. Some unwanted branches ( the branches that don't lead to the required word) are dropped. For the word  $i+i*i$ , the total language tree can be started as



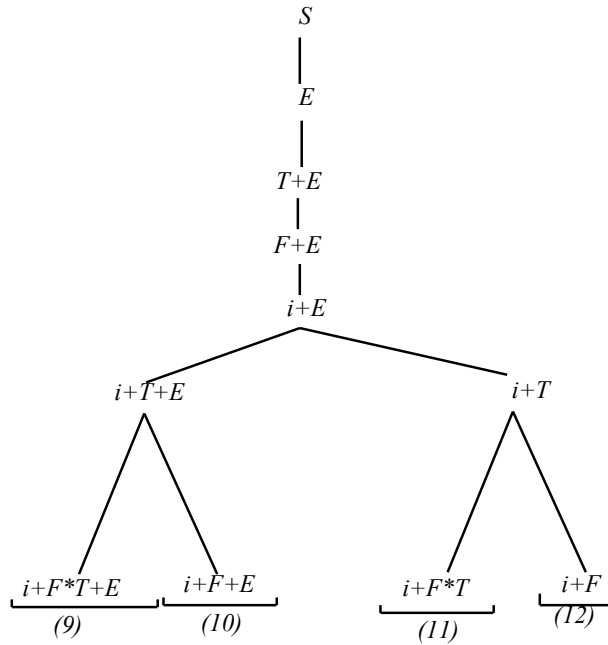
Which can further be extended to



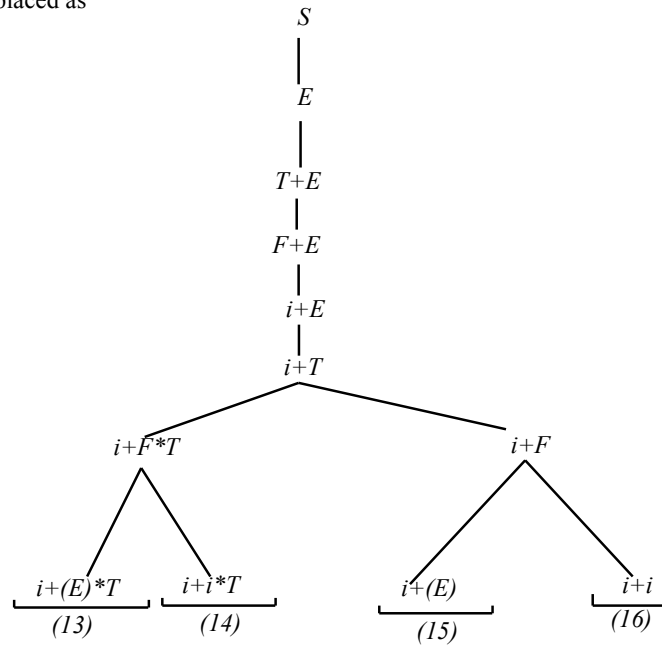
Dropping the unwanted branches 1,3,5,7 and 8



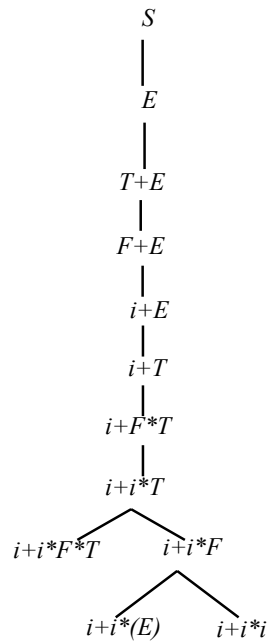
Since first two letters in branches 2 and 6 are not that in  $i+i^*i$ , so 2 and 6 can be dropped and using left most derivation the nonterminal T is replaced as



since (9) gives more than five letters and (10) contains two + so (9) and (10) are dropped and left most nonterminal F is replaced as



(13), (15) and (16) are again unwanted, so it can be written as



The above tree confirms the required derivation

$$\begin{aligned}
 &S \Rightarrow E \\
 &\quad \Rightarrow T+E \\
 &\Rightarrow F+E \\
 &\quad \Rightarrow i+E \\
 &\Rightarrow i+T \\
 &\quad \Rightarrow i+F*T \\
 &\Rightarrow i+i*T \\
 &\quad \Rightarrow i+i*F \\
 &\Rightarrow i+i*i
 \end{aligned}$$

#### Note

It can be noted that Bottom Up Parsing can be determined similar to that of Top Down Parsing with the change that in this case, the process is started with the given string and the tree is extended till S is obtained.

## Theory of Automata

**Lecture N0. 45****Reading Material**Introduction to Computer Theory

## Chapter 19

**Summary**

Turing machine, examples, DELETE subprogram, example, INSERT subprogram, example.

**Turing machine**

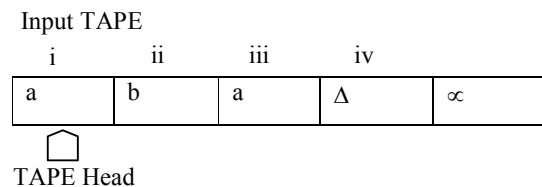
The mathematical models (FAs, TGs, PDAs) that have been discussed so far can decide whether a string is accepted or not by them *i.e.* these models are language identifiers. However, there are still some languages which can't be accepted by them *e.g.* there does not exist any FA or TG or PDA accepting any non-CFLs. Alan Mathison Turing developed the machines called Turing machines, which accept some non-CFLs as well, in addition to CFLs.


**Definition**

A Turing machine (TM) consists of the following

An alphabet  $\Sigma$  of input letters.

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 ( $\Delta$ 's).



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 .

An alphabet  $\Gamma$  of characters that can be printed on the TAPE by the TAPE Head.  $\Gamma$  may include the letters of  $\Sigma$ . Even the TAPE Head can print blank  $\Delta$ , which means to erase some character from the TAPE.

Finite set of states containing exactly one START state and some (may be none) HALT states that cause execution to terminate when the HALT states are entered.

A **program** which is the set of rules, which show that which state is to be entered when a letter is read from the TAPE and what character is to be printed. This program is shown by the states connected by directed edges labeled by triplet (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.

**Note**

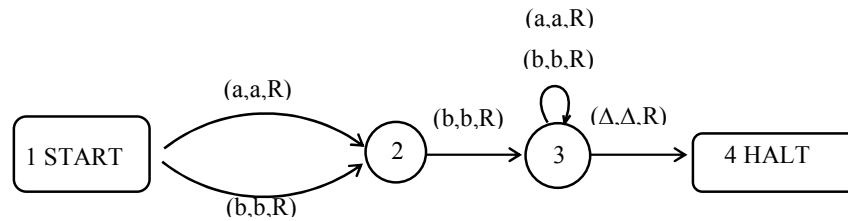
It may be noted that there may not be any outgoing edge at certain state for certain letter to be read from the TAPE, which creates nondeterminism in Turing machines. It may also be noted that at certain state, there can't be more than one out going edges for certain letter to be read from the TAPE. The machine crashes if there is not path for a letter to be read from the TAPE and the corresponding string is supposed to be rejected.

To terminate execution of certain input string successfully, a HALT state must be entered and the corresponding string is supposed to be accepted by the TM. The machine also crashes when the TAPE Head is instructed to move one cell to the left of cell *i*.

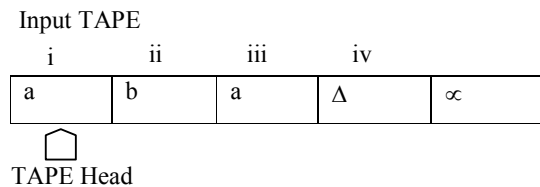
Following is an example of TM

**Example**

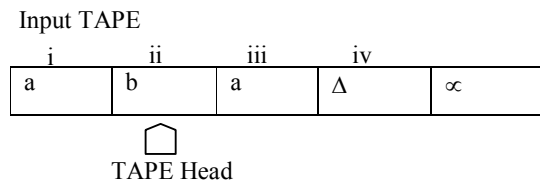
Consider the following Turing machine



Let the input string aba be run over this TM



Starting from the START state, reading a from 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

1      2  
 $\underline{a}ba \rightarrow ab\bar{a}$

At state 2 reading b, state 3 is entered and the letter b is replaced by b, *i.e.*

1      2      3  
 $\underline{a}ba \rightarrow ab\underline{a}$

At state 3 reading a, will keep the state of the TM unchanged. Lastly, the blank Δ is read and Δ is replaced by Δ and the HALT state is entered. Which can be expressed as

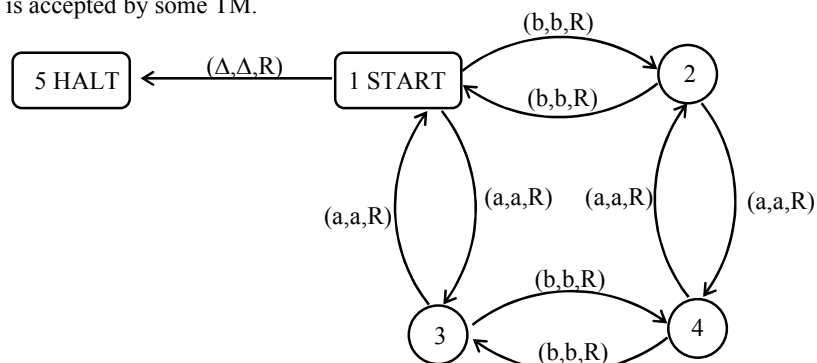
1      2      3      3      HALT  
 $\underline{a}ba \rightarrow ab\underline{a} \rightarrow aba \rightarrow aba\Delta$

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

### Theorem

Every regular language is accepted by some TM.

### Example

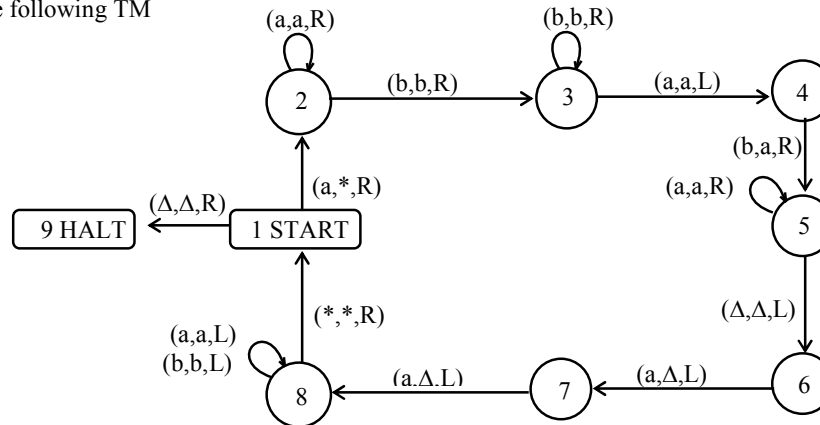


Consider the EVEN-EVEN language. Following is a TM accepting the EVEN-EVEN language.

It may be noted that the above diagram is similar to that of FA corresponding to EVEN-EVEN language. Following is another example

#### Example

Consider the following TM

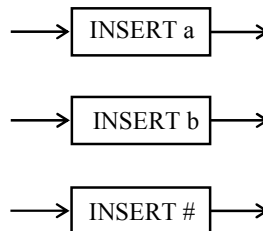


The string aaabbbaaa can be observed to be accepted by the above TM. It can also be observed that the above TM accepts the non-CFL  $\{a^n b^n a^n\}$ .

#### INSERT subprogram

Sometimes, a character is required to be inserted on the TAPE exactly at the spot where the TAPE Head is pointing, so that the character occupies the required cell and the other characters on the TAPE are moved one cell right. The characters to the left of the pointed cell are also required to remain as such.

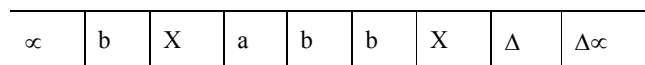
In the situation stated above, the part of TM program that executes the process of insertion does not affect the function that the TM is performing. The subprogram of insertion is independent and can be incorporated at any time with any TM program specifying what character to be inserted at what location. The subprogram of insertion can be expressed as



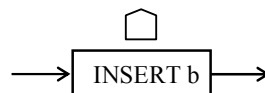
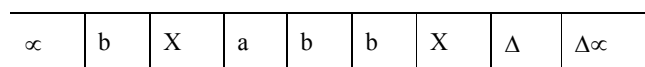
The above diagrams show that the characters a, b and # are to be inserted, respectively. Following is an example showing how does the subprogram INSERT perform its function

#### Example

If the letter b is inserted at the cell where the TAPE Head is pointing as shown below



then, it is expressed as



The function of subprogram INSERT b can be observed from the following diagram

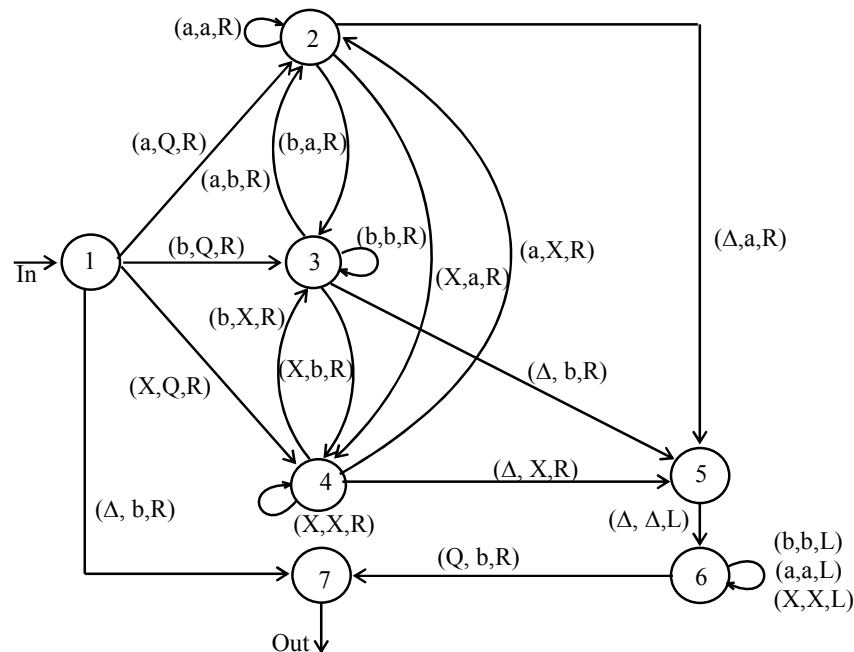
$\infty$	b	X	b	a	b	b	X	$\Delta$	$\Delta\infty$
----------	---	---	---	---	---	---	---	----------	----------------



Following is the INSERT subprogram

### The subprogram INSERT

Keeping in view the same example of inserting b at specified location, to determine the required subprogram, first Q will be inserted as marker at the required location, so that the TAPE Head must be able to locate the proper cell to the right of the insertion cell. The whole subprogram INSERT is given as



It is supposed that machine is at state 1, when b is to be inserted. All three possibilities of reading a, b or X are considered by introducing the states 2,3 and 4 respectively. These states remember what letter displaced during the insertion of Q.

Consider the same location where b is to be inserted

$\infty$	b	X	a	b	b	X	$\Delta$	$\Delta\infty$
----------	---	---	---	---	---	---	----------	----------------



After reading a from the TAPE, the program replaces a by Q and the TAPE Head will be moved one step right. Here the state 2 is entered. Reading b at state 2, b will be replaced by a and state 3 will be entered. At state 3, b is read which is not replaced by any character and the state 3 will not be left.

At state 3, the next letter to be read is X, which will be replaced by b and the state 4 will be entered. At state 4,  $\Delta$  will be read, which will be replaced by X and state 5 will be entered. At state 5,  $\Delta$  will be read and without any change state 6 will be entered, while TAPE Head will be moved one step left. The state 6 makes no change whatever (except Q) is read at that state. However at each step, the TAPE Head is moved one step left. Finally, Q is read which is replaced by b and the TAPE Head is moved to one step right. Hence, the required situation of the TAPE can be shown as

$\infty$	b	X	a	b	b	X	$\Delta$	$\Delta\infty$
----------	---	---	---	---	---	---	----------	----------------



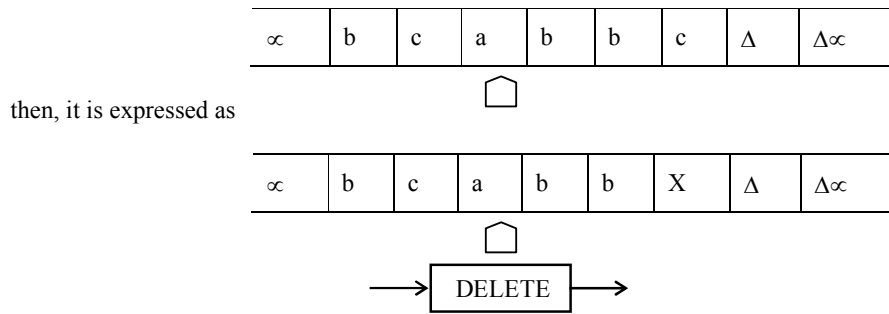
### DELETE subprogram

Sometimes, a character is required to be DELETED on the TAPE exactly at the spot where the TAPE Head is pointing, so that the other characters on the right of the TAPE Head are moved one cell left. The characters to the left of the pointed cell are also required to remain as such.

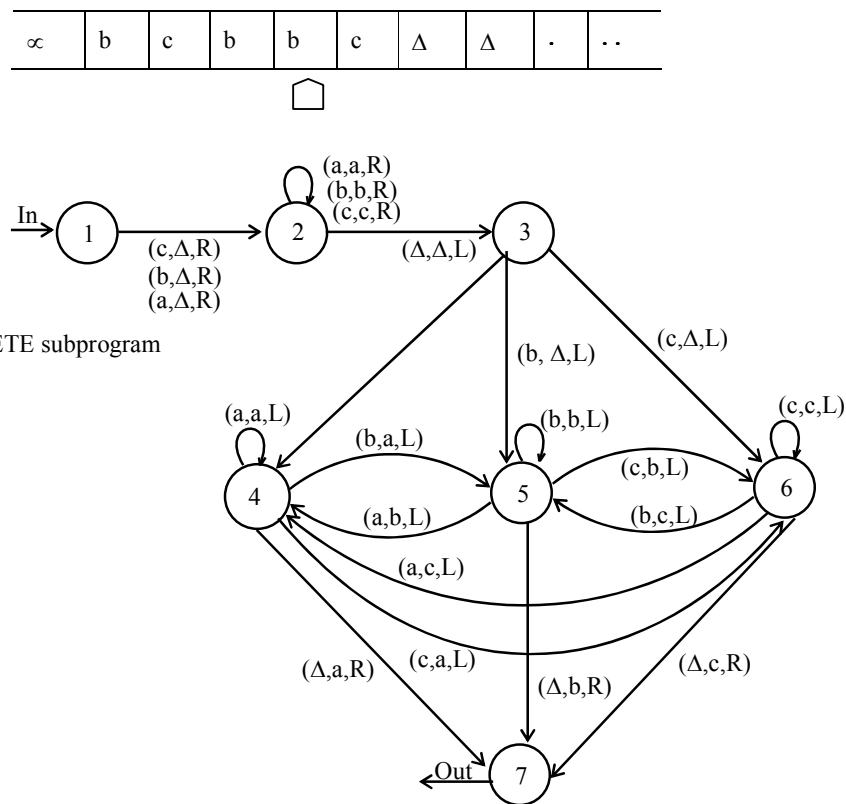
In the situation stated above, the part of TM program that executes the process of deletion does not affect the function that the TM is performing. The subprogram of deletion is independent and can be incorporated at any time with any TM program specifying what character to be deleted at what location. The subprogram of deletion can be expressed as

#### Example

If the letter a is to be deleted from the string bcabbc, shown below



The function of subprogram DELETE can be observed from the following diagram



The process of deletion of letter a from the string bcabbc can easily be checked, giving the TAPE situation as shown below

