الجامعة المستنصرية

The University of Mustansiriyah

Automata Theory

النظرية الإحتسابية

المرحلة الثانية

2009 - 2010

أ.حسن قاسم محمد

www.uomustansiriyah.edu.iq
www.mustansiriya.com

hassan.kassim@yahoo.com

Instructor: Hassan Kassim Mohammad

Theory of computation is the theoretical study of capabilities and limitations of Computers (Theoretical models of computation).

Objectives:

Providing students with:

- an understanding of basic concepts in the theory of computation through simple models of computational devices.
- apply models in practice to solving problems in diverse areas such as string searching, pattern matching, cryptography, and language design;
- understand the limitations of computing, the relative power of formal languages and the inherent complexity of many computational problems.
- be familiar with standard tools and notation for formal reasoning about machines and programs.

REFERENCES:

1. Introduction to Computer Theory 2nd Edition
   Daniel I. A. Cohen John Wiley & Sons, Inc 1997. ISBN 0-471-13772-3
2. Introduction to Automata Theory, Languages, and Computation, 2/E,
   John E. Hopcroft, Rajeev Motwani, Jeffrey D.Ullman, Addison-Wesley 2001. ISBN 0-201-44124-1.

Units: 6

Grading Policy

Semester	Exam	Attendance	Assignments & Quizzes	Total
1st semester	10	2	3	15
2nd semester	10	2	3	15
Final	70	-	-	70

Notes

Student must attend at least 80% of total classes to pass the course.
Any kind of cheating/plagiarism may result in a Fail grade in the course.
No labs. But you should write some programs with any language you may know.
There will be about 30 lectures 100 minutes each.
Late homework submissions will be penalized

Office Hours

Sunday, Monday, Tuesday, Wednesday

Contact Information

Office: computer science dept. room no. 67
E-mail: hassan.kassim@yahoo.com

Syllabus

Week	Date	Subject	Chapter
1		Introduction, terminology, definitions	1
2		Sets and operations	1
3		languages	2
4		Regular Expressions RE	4
5		Finite Automata FA	5
6		Deterministic Finite Automaton DFA	5
7		Non Deterministic Finite Automaton NDFA	8
8		Language Accepted by Finite Automata	5
9		Convert Regular Expression into NFA	
10		Constructing regular expression from Finite Automata	
11		Finite Automata with Epsilon moves	
12		Moore and Mealy machines	9
13		Converting between Moore and Mealy machine	
14		Pumping lemma for regular languages	
15		Kleene's Theorem	7
16		Regular Grammar	10
17		Myhill-Nerode Theorem Minimization of DFA	
		EXAM	
18		Context-free Languages	13
19		Pushdown Automata	17
20		CFG/CFL to PDA	18
21		PDA to CFG/CFL	
22		CFG derivation trees Parsing	22
23		Chomsky normal form	16
24		Greibach normal form	16
25		Ambiguous CFL's	
		EXAM	
26		TURING MACHINES TM	24
27		COMPUTABILITY and COMPLEXITY	
28		Unsolvable Problems	
29		Time Complexity	
30		CYK algorithm for CFG's	
31		CFL pumping lemma and properties	
32		Church Turing Thesis	

As a computer IT, you must study the following:

1- **Automata and formal language**.
Which answers - What are computers (Or what are models of computers)

2- **Compatibility**.
Which answers - What can be computed by computers?

3- **Complexity**.
Which answers - What can be efficiently computed?

In automata we will simulates parts of computers. Or we will make mathematical models of computers Automata are more powerful than any real computer because we can design any machine on papers that can do everything we want.

Theory of computation is the theoretical study of capabilities and limitations of Computers (Theoretical models of computation).

Sets

Let A, B, and C be subsets of the universal set U

Distributive properties

$A \cap (B \cup C) = (A \cap B) \cup (A \cap C$

$A \cup (B \cap C) = (A \cup B) \cap (A \cup C$

Idempotent properties

$A \cap A = A,$

$A \cup A = A.$

Double Complement property

$(A^{-})^{-} = A.$

De Morgan's laws

$(A \cup B)^{-} = A^{-} \cap B^{-}$

$(A \cap B)^{-} = A^{-} \cup B^{-}$

Commutative properties

$A \cap B = B \cap A,$

$A \cup B = B \cup A.$

Associative laws

$A \cap (B \cap C) = (A \cap B) \cap C$

$A \cup (B \cup C) = (A \cup B) \cup C$

Identity properties

$A \cup \emptyset = A,$

$A \cap U = A.$

Complement properties

$A \cup A^{-} = U,$

$A \cap A^{-} = \emptyset.$

Language

language is the set of all strings of terminal symbols derivable from alphabet.

alphabet is a finite set of symbols. For example {*0, 1*} is an alphabet with two symbols, {*a, b*} is another alphabet with two symbols and English alphabet is also an alphabet. A **string** (also called a word) is a finite sequence of symbols of an alphabet. *b, a* and *aabab* are examples of string over alphabet {*a, b*} and *0, 10* and *001* are examples of string over alphabet {*0, 1*}, A null string is a string with no symbols, usually denoted by epsilon or lambda (λ). A **language** is a set of strings over an alphabet. Thus {*a, ab, baa*} is a language (over alphabert {*a,b*}) and {*0, 111*} is a language (over alphabet {*0,1*}). The number of symbols in a string is called the **length of the string**. For a string w its length is represented by |w| . It can be The **empty string** (also called null string) it has no symbols. The empty string is denoted by λ Thus |λ| = 0.

For example |00100| = 5, |aab| = 3, | λ | = 0

Language = alphabet + string (word) + grammar (rules, syntax) + operations on languages (concatenation, union, intersection, Kleene star)

Kinds of languages:

1- Talking language: (e.g.: English, Arabic): It has alphabet: ∑={a,b,c,….z}From these alphabetic we make sentences that belong to the language.

Now we want to know is this sentence is true or false so -We need a grammar.

Ali is a clever student. (It is a sentence Є English language.)

2- Programming language: (e.g.: c++, Pascal):It has alphabetic:∑={a,b,c,.z , A,B,C,..Z , ?, /, - ,\.}

From these alphabetic we make sentences that belong to programming language.

Now we want to know if this sentence is true or false so we need a compiler to make sure that syntax is true.

3- Formal language*:* (any language we want.) It has strings from these strings we make sentences that belong to this formal language.

Now we want to know is this sentence is true or false so we need rules.

Example:

Alphabetic: ∑= {0, 1}.
Sentences: 0000001, 1010101.
Rules: Accept any sentence start with zero and refuse sentences that start with one.
So we accept: 0000001 as a sentence satisfies the rules.
And refuse: 1010101 as a sentence doesn't satisfy the rules.

Example:

Alphabetic: ∑= {a, b}.
Sentences: ababaabb, bababbabb
Rules: Accept any sentence start with a and refuse sentences that start with b.
So we accept: aaaaabba as a sentence satisfies the rules.
And refuse: baabbaab as a sentence doesn't satisfy the rules.

Regular Expression

is a set of symbols, Thus if alphabet= {a, b}, then aab, a, baba, bbbbb, and baaaaa would all be strings of symbols of alphabet.

In addition we include an empty string denoted by λ which has no symbols in it.

Examples of Kleene star:

1* is the set of strings {λ, 1, 11, 111, 1111, 11111, etc. }
(1100)* is the set of strings {λ, 1100, 11001100, 110011001100, etc. }
(00+11)* is the set of strings {epsilon, 00, 11, 0000, 0011, 1100, 1111, 000000, 000011, 001100, etc. }
(0+1)* is all possible strings of zeros and ones, often written as sigma * where sigma = {0, 1}
(0+1)* (00+11) is all strings of zeros and ones that end with either 00 or 11.

(w)+ is a shorthand for (w)(w)* w is any string or expression and the superscript plus, +

1- **Concatenation**:

Notation to the concatenation: . (The dot.):

if L1 = {x, xxx} and L2 = {xx} So (L1.L2) means L1 concatenated L2 and it is equal = {xxx, xxxxx}

Examples on concatenations:

Ex1:
L1 = {a, b}.
L2 = {c, d}.
L1.L2 = {ac, ad, bC, bd}
Note: ab differ from ba.

Ex2:
∑= {x}.
L1 = {set of all odd words over ∑ with odd length}.
L1 = {set of all even words over ∑ with odd length}.
L1= {x, xxx, xxxxx, xxxxxxx……}.
L2= {λ, xx, xxxx, xxxxxx…}.
L1.L2 = {x, xxx, xxxxx, xxxxxxx…}.
Note:
التكرار غير مسموح داخل المجموعة.

Ex3:
L1 = {x, xxx}.
L2 = {xx}.
L1.L2 = {xxx, xxxxx}.
Some rules on concatenation:
λ.x = x
L1.L2 = {set of elements}

Definition of a Regular Expression

A regular expression may be the null string, r = λ
A regular expression may be an element of the input alphabet, r = a
A regular expression may be the union of two regular expressions, r = r1 + r2
A regular expression may be the concatenation of two regular expressions, r = r1 r2
A regular expression may be the Kleene closure (star) of a regular expression r = r1*
A regular expression may be a regular expression in parenthesis r = (r1)

epsilon is the zero length string
0, 1, a, b, c, are symbols in sigma
x is a variable or regular expression
(...)(...) is concatenation
(...) + (...) is union
(...)* is the Kleene Closure = Kleene Star

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

λ is the null string (there are no symbols in this string)
* is the set of all strings of length greater than or equal to 0

Example:
A = {a,b} // the alphabet is composed of a and b
A* = {λ, a,b,aa,ab,ba,bb,aaa,aab,…}
The symbol * is called the Kleene star.
∅ (empty set)
λ (empty string)
( ) delimiter ,
∪ + union (selection)
concatenation

Given regular expressions x and y, x + y is a regular expression representing the set of all strings in either x or y (set union)

x = {a, b}, y = {c, d}, x + y = {a, b, c, d}

Example 1
Let A={0,1}, W1 = 00110011, W2 = 00000
W1W2 = 0011001100000
W2W1 = 0000000110011
W1 λ = W1 = 00110011
λ W2 = W2 = 00000

x = {a, b}, y = {c, d}, xy = {ac, ad, bc, bd}

Note:
$\mathbf{(a + b)^* = (a^* b^*)^*}$

Examples of regular expressions

Describe the language = what is the output (words, strings) of the following RE

Regular expression	output(set of strings)
λ	{λ}
λ*	{λ}
a	{ a }
aa	{ aa }
a*	{λ, a, aa, aaa, ….}
aa*	{ a, aa, aaa, ... }
a+	{ a, aa, aaa, ...}
ba+	{ ba, baa, baaa, ...}
(ba)+	{ ba, baba, bababa, ...}
(alb)	{ a, b }
alb*	{ a, λ, b, bb, bbb, ... }
(alb)*	{ λ, a, b, aa, ab, ba, bb, ... }
aa(ba)*bb	{ aabb, aababb, aabababb, ... }
(a + a)	{a}
(a + b)	{a, b}
(a + b)2	(a + b)(a + b) == {aa, ab, ba, bb}
(a + b + c)	{a, b, c}
(a + b)*	{λ, a, b, aa, bb, ab, ba, aaa, bbb, aab, bba, ….}
(abc)	{abc}
(λ + a) bc	{bc, abc}
ab*	{a, ab, abb, abbb, …}
(ab)*	{λ, ab, abab, ababab, …}
a + b*	{a, λ, b, bb, bbb, …}
a (a + b)*	{a, aa, ab, aaa, abb, aba, abaa, … }
(a + b)* a (a + b)*	{a, aaa, aab, baa, bab, …}
(a + λ)*	(a)* = {λ, a, aa, aaa, ….}
x* (a + b) + (a + b)	x* (a + b)
x* y + y	x* y
(x + λ)x*	x* (x + λ) = x*
(x + λ)(x + λ)* (x + λ)	x*

start with a	a (a + b)*
end with b	(a + b)* b
start with a and end with b	
start with a or b	
not start with b	
contains exactly 2 a's	(b)* a (b)* a (b)*
contains at least 2 a's	(a + b)* a (a + b)* a (a + b)*
contains exactly 2 a's or 2 b's	[(b)* a (b)* a (b)*] + [(a)* b (a)* b (a)*)]
contains even no of a	[(b)* a (b)* a (b)*]*
not start with a and not contain b	
with even length of a	(aa)+
Strings containing 101	
Even number of 0's and contains 101	
Even number of 0's or contains 101	
Every one has at least two zeros that follow it	
Second symbol not a one	
End with 00 or 01	

Exercise

Ex. 1: Find a regular expression over the alphabet { a, b } that contain exactly three a's.
Ex. 2: Find a regular expression over the alphabet { a, b } that end with ab.
Ex. 3: Find a regular expression over the alphabet { a, b } that has length of 3.

Ex. 4: Find a regular expression over the alphabet { a, b } that contain exactly two successive a's.
Ex. 5: Find the output (words) for the following regular expressions.

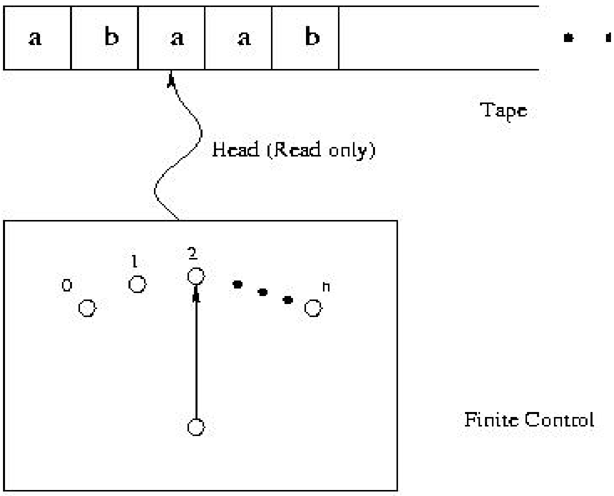
(λ)*	
(x)* + (λ)	
aa* b	
bba*a	
(a + b)* ba	
(0+1)* 00 (0+1)*	
(11 + 0)* (0+11)*	
01* + (00+101)*	
(a+b)* abb+	
(((01+10)* 11)* 00)*	

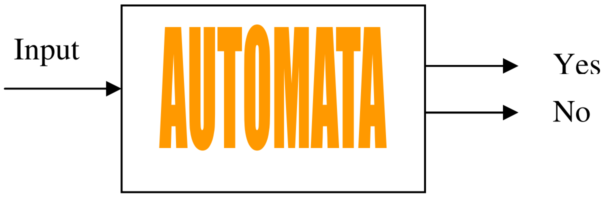
Finite Automata

شكل رسومي يمثل تنقلات بين مجموعة حالات تسيطر عليها قواعد عن طريق شريط من المدخلات

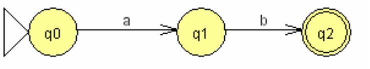
is a device consisting of a tape and a control circuit which satisfy the following conditions:

1. The tape start from left end and extends to the right without an end.
2. The tape is divide into squares in each a symbol.
3. The tape has a read only head.
4. The head moves to the right one square every time it reads a symbol. It never moves to the left. When it sees no symbol, it stops and the automata terminates its operation.
5. There is a control determines the state of the automaton and also controls the movement of the head.





A DFA represents a finite state machine that recognizes a RE.

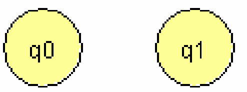
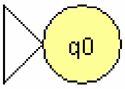
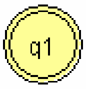
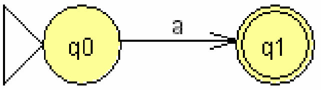
For example, the following FA:  recognize (accept) string ab

A finite automaton consists of a finite set of states, a set of transitions (moves), one start state, and a set of final states (accepting states). In addition, a DFA has a unique transition for every state combination.
it is a set of states, and its “control” moves from state to state in response to external “inputs” .
A finite automaton, FA, provides the simplest model of a computing device. It has a central processor of finite capacity and it is based on the concept of state.

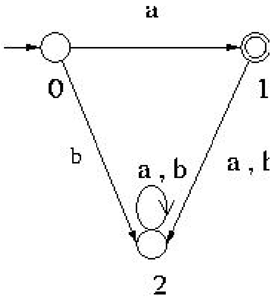
finite state machine is a 5 tuple M = (Q, A, T, S ,F), where

- Q --set of **states** = {q0, q1, q2, ….}
- A -- set of **input symbols** ={a,b, …, 0, 1, …}
- T --set of transitions or rules
- S -- an initial state
- F -- the final **state** -- could be more than one final state

Designing (drawing) FA

State with numbers or any name	Start - or small arrow	Final + or double circle	Transition (only one input or symbol on the edge) a,b allowed means (a or b)
			 loop

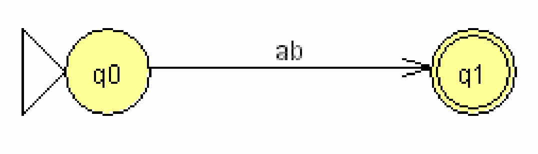
Example: Q = { 0, 1, 2 }, A= { a, b }, F = { 1 }, the initial state is 0 and T is shown in the following table.



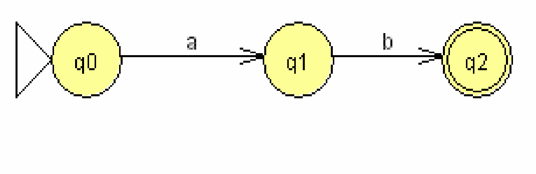
State (q)	Input (a)	Input (b)
0	1	2
1	2	2
2	2	2

Transition diagram:

TG has many inputs on the edge ab

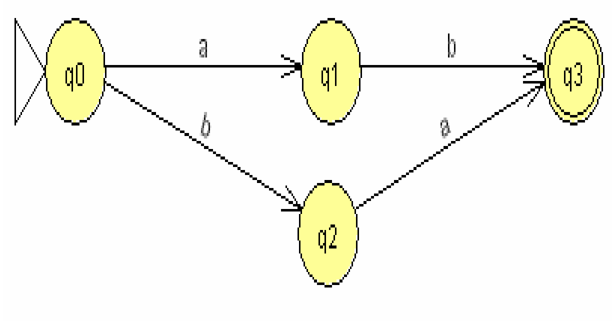


FA has only one input on the edge a

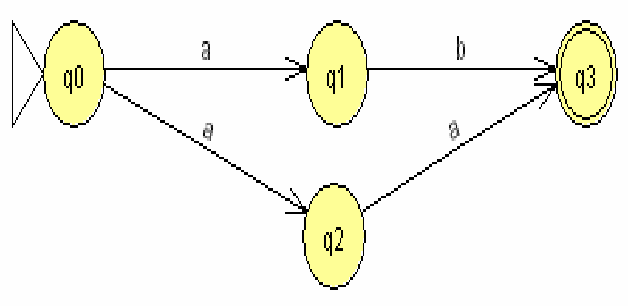


Deterministic Finite Automata DFA and Non Deterministic Finite Automata NFA

DFA: different input from state to different states



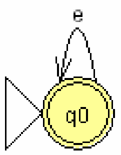
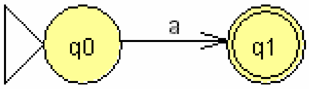
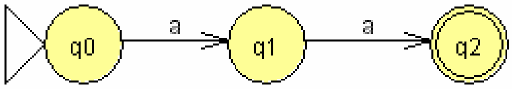
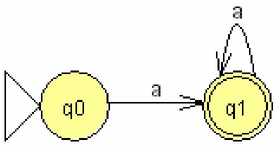
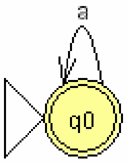
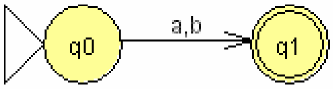
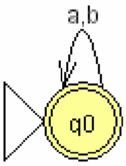
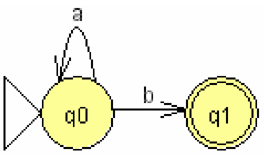
NFA: one input from state to different states

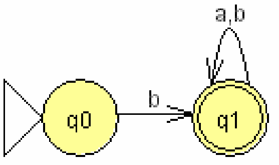
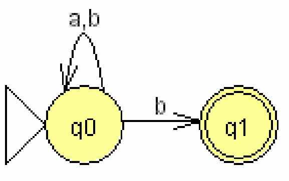
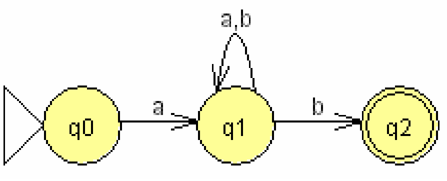
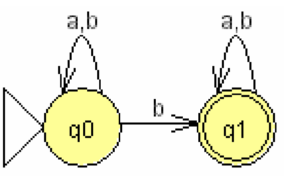
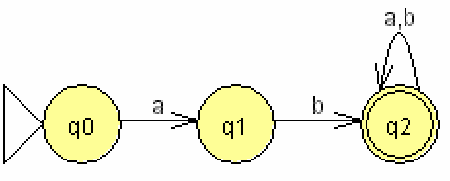
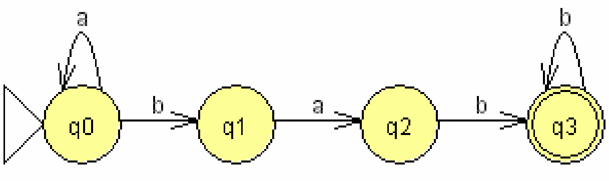
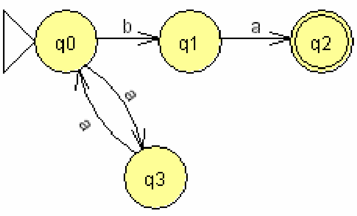
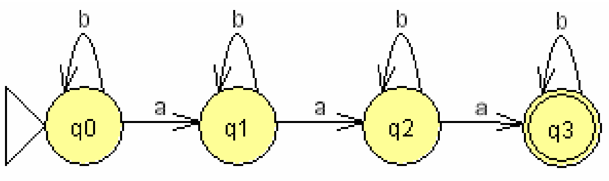


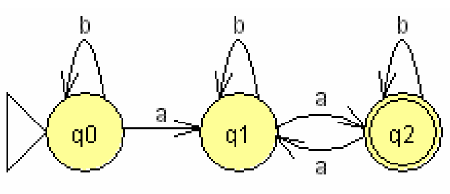
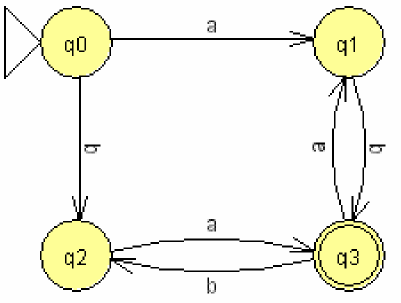
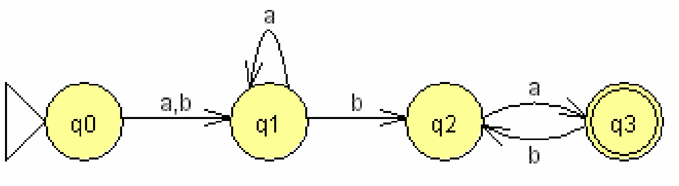
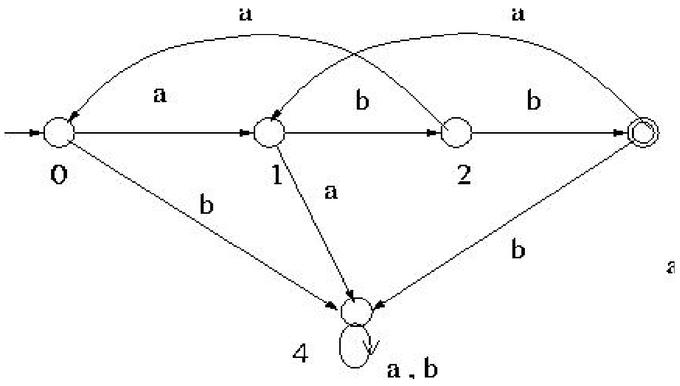
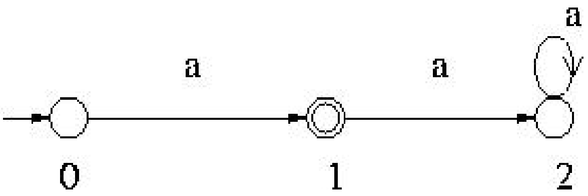
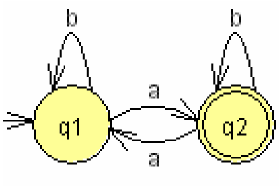
Language accepted by FA

String is accepted by a FA if and only if the FA starting at the initial state and ends in an accepting state after reading the string.

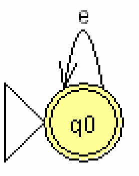
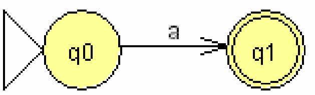
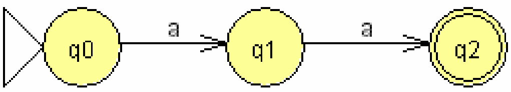
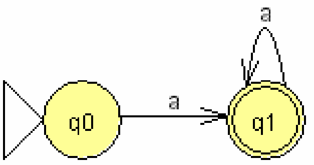
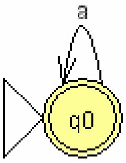
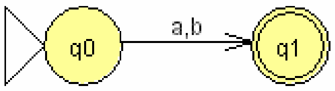
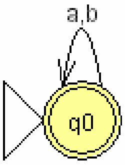
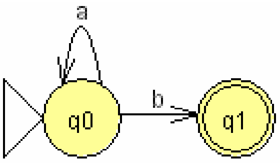
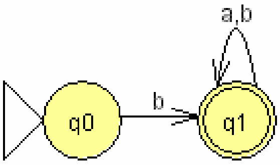
Examples of languages accepted by FA

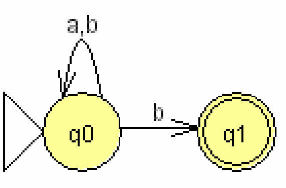
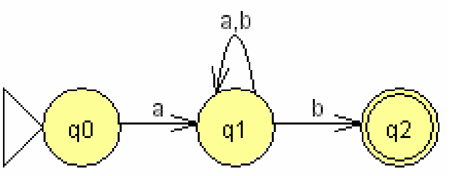
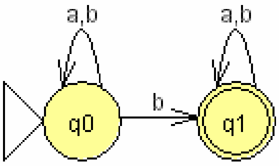
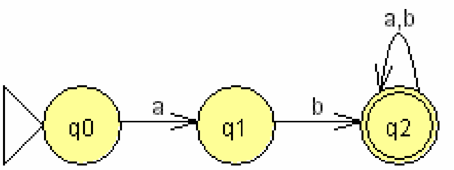
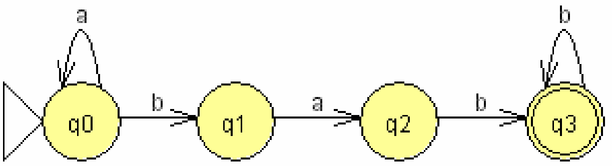
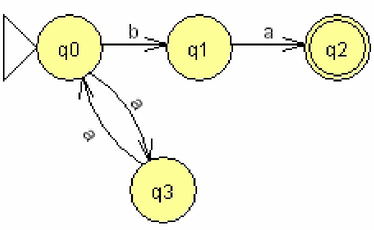
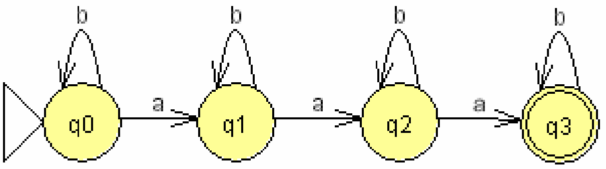
FA	RE
	λ
	a
	aa
	a+ = aa*
	a*
	a+b
	(a+b)*
	a*b

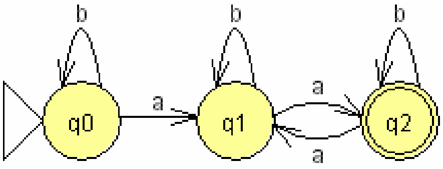
	b(a+b)*
	(a+b)*b
	a(a+b)*b
	(a+b)* b(a+b)*
	ab(a+b)*
	a*babb*
	(aa)*ba
	contains 3 a's b*ab*ab*ab*

	contains even number of a = (b*ab*ab*)+
	
	
 a a a b b 0 1 2 b a b a 4 a , b	a(bba + baa)*bb
	a
	

Converting Regular Expression into a Finite Automata

RE	FA
λ	
a	
aa	
a+ = aa*	 NFA
a*	
a+b	
(a+b)*	
a*b	
b(a+b)*	 NFA

(a+b)*b	 NFA
a(a+b)*b	 NFA
(a+b)* b(a+b)*	 NFA
ab(a+b)*	 NFA
a*babb*	 NFA
(aa)*ba	
contains 3 a's b*ab*ab*ab*	

contains even number of a = (b*ab*ab*)+	
dividable by 3	
all bit strings that begin with 0 and end with 1	
all bit strings whose number of 0's is a multiple of 5	
all bit strings with more 1's than 0's	
all bit strings with no consecutive 1's	

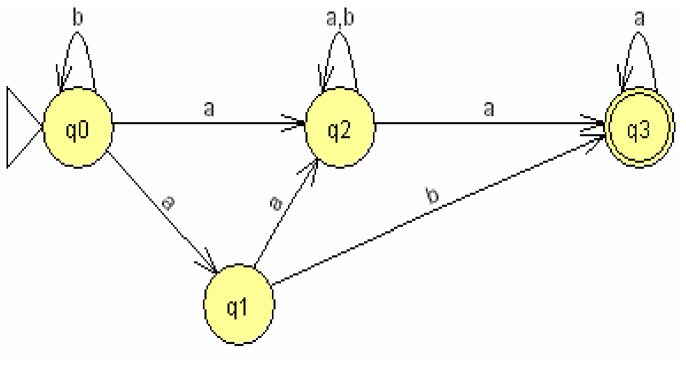
Converting NFA into DFA

Three steps : 1- find transition table عمل جدول التنقلات

2- drawing new design رسم الشكل الجديد

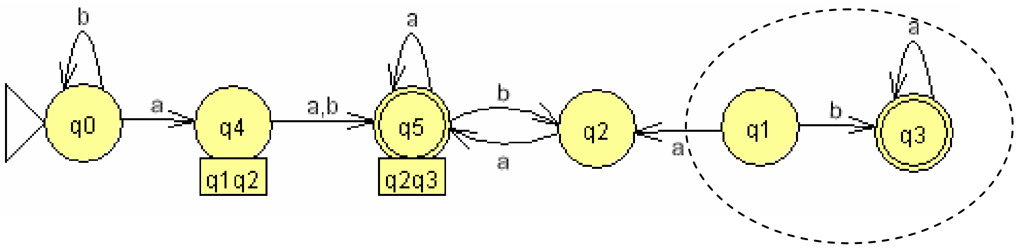
3- remove unreachable states إزالة الحالات الزائدة التي لا يمكن الوصول إليها

Example : convert the following NDFA into DFA

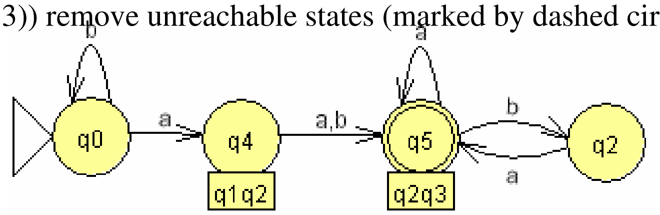


state	a	b
q0	q1q2	q0
q1	q2	q3
q2	q2q3	q2
q3	q3	-
q1q2	q2q3	q2q3
q2q3	q2q3	q2

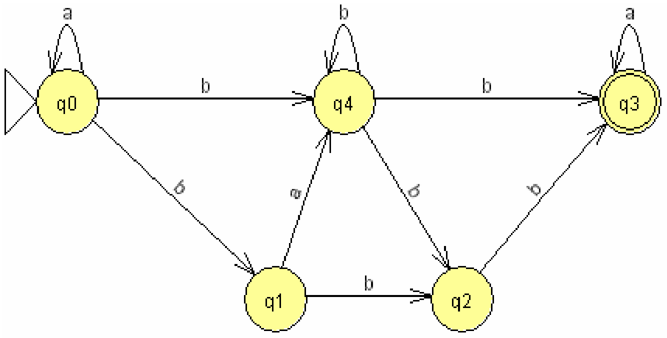
Note: Any state contains final mark it will be final state



3)) remove unreachable states (marked by dashed circle – state q1 and state q3) because we can not reach it.

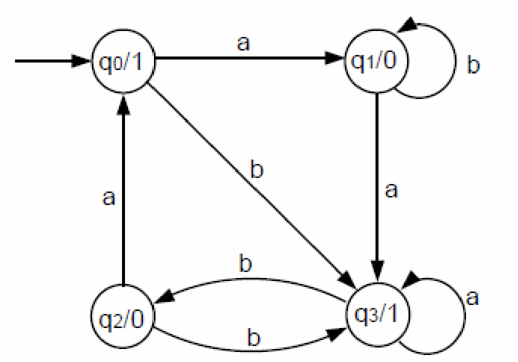


HW convert the following NFA into DFA



Finite State Machines with Output (Mealy and Moore Machines)

Moore Machines



Moore machine *M* is the 5tuple *M* = (*Q*, A, O, T, F, s) where

Q is a finite set of states
A is the finite input alphabet
O is the finite output alphabet
T is the transition function
F is the output function Q→A
in addition to the start state or the initial state

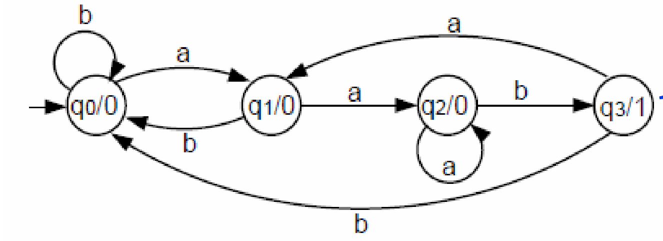
A Moore machine is very similar to a Finite Automaton (FA), with a few key differences:

- It has no final states.
- It does not accept or reject input, instead, it generates output from input.
- Moore machines cannot have nondeterministic states.

Every input gives output not if word belongs to the machine or language like FA

In each state we stop we print out what inside that state (it's content)

so the output will be more than input by one because we start with start state and print out it's content before we trace the input string



This machine might be considered as a "counting" machine

Input string	aaababbaabb
State	q0q1q2q2q3q1q0q0q1q2q3q0
Output	000010000010

↑ ↑

this machine gives 1 after each aab

aabaabaaababaab
0001001000100001

so we use Moore machine as a string ***recognizer*** to give us a mark (1) after each substring so we design a machine put 0 in all states except the one after the one represent end of substring aba

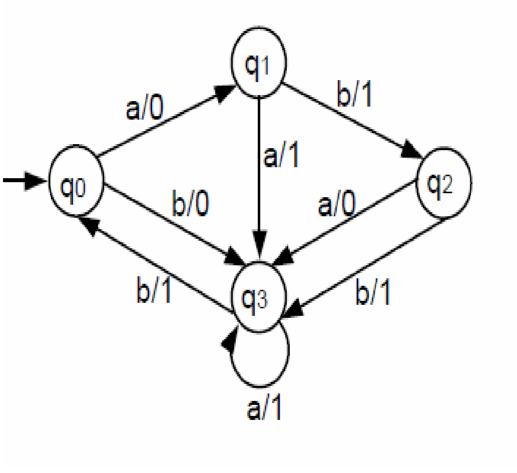
The output of the machine contains 1 for each occurrence of the substring aab found in the input string.

H.W. Construct a Moore machine that outputs a binary string that contains a 1 for every double letter substring in an input string composed of a's and b's. For example if *abba* is the input string 0010 is the corresponding output.

Mealy machines

Moore machine *M* is a 5 tuple *M* = (*Q*, A, O, T, F ,s) where

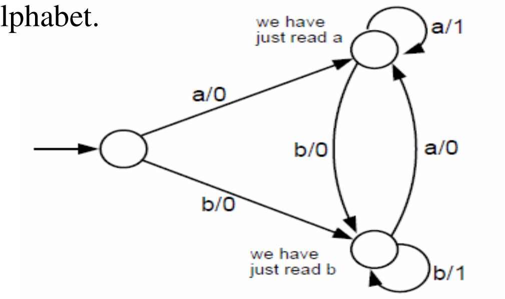
Q is a finite set of states
A is the finite input alphabet
O is the finite output alphabet
T is the transition function
F is the output function Q→A
in addition to the start state or the initial state

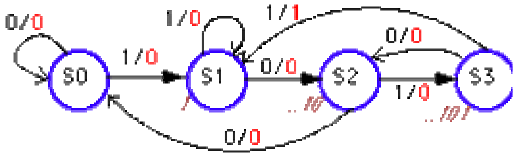


output on edge
same input to output

aaabb
01110

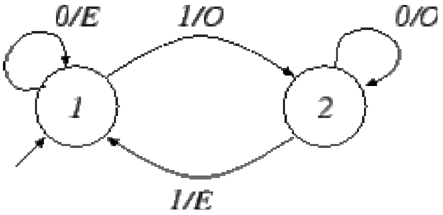
Mealy machines are finite-state machines that act as transducers or translators, taking a string on an input alphabet and producing a string of equal length on an output alphabet.
Mealy machine does not accept or reject an input string,





The machine represented in below, outputs an `E` if the number of `1`s read so far is even and an `O` if it is odd; for example, the translation of `11100101` is `OEOOOEEO`.

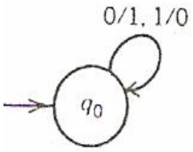
A Mealy machine that outputs
`E` if the number of `1` is even
`O` if the number of `1` is odd



Binary inverter

The following Mealy machine takes the one's complement of its binary input. In other words, it flips each digit from a 0 to a 1 or from a 1 to a 0.

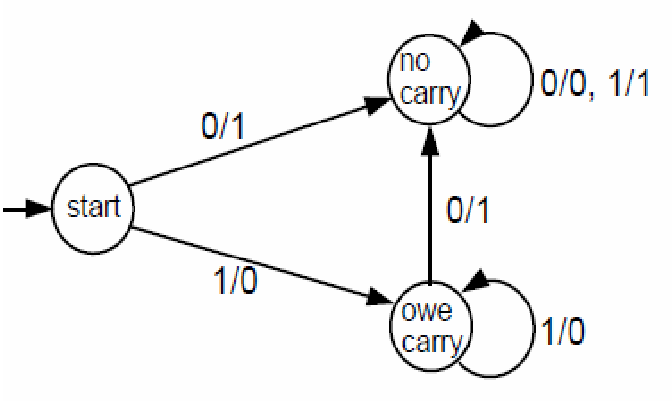
Input = 0010



Output=11010

There are no accept states in a Mealy machine because it is not a language recogniser, it is an output producer. Its output will be the same length as its input.

Binary Incrementer



One thing you will notice is the numbering of the states. Usually, if there are 3 states, we number them 00, 01, and 10.

- the input bit string is a binary number fed in backward
- The output string will be the binary number that is one greater and that is **generated right to left**.
- The machine will have 3 states: start, carry and no-carry. The carry state represents the overflow when two bits of 1's are added, we print a 0 and we carry a 1.

Let the input string be 1011 (binary representation of 11).

• The string is fed into the machine as 1101 (backwards).

• The output will be 0011, which when reversed is 1100 and is the binary representation of 12.

• In Mealy machine, output length = input length. Hence, if input were 1111, then output would be 0000 (**overflow situation**).

Homework:

Construct a Mealy machine that takes a string of a's and b's as input and outputs a binary string with a 1 at the position of every second double letter. For example, for *ababbaab* the machine produces *00001010* and for the input *bbb* the output string *011* is produced.

Kleene's Theorem

Any language that can be defined by: Regular expression/ Finite automata/ Transition graph Can be defined by all three methods.

Proof

There are three parts of our proof :

Part1: every language that can be defined by a FA == can be defined by a TG.
Part2: every language that can be defined by a TG == can be defined by a RE.
Part3: every language that can be defined by a RE == can be defined by a FA.

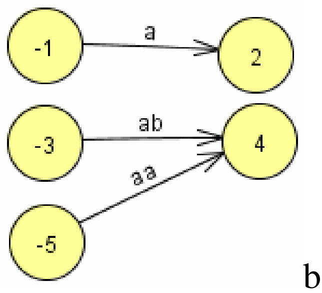
proof of part1

Every FA is itself a TG. Therefore, any language that has been defined by a FA has already been defined by a TG.

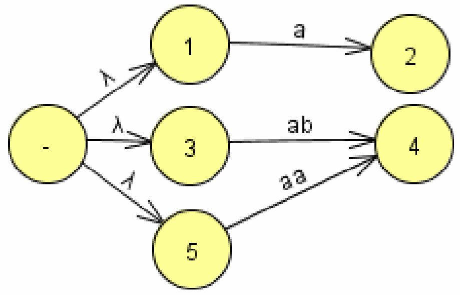
proof of part2

The proof of this part will be by constructive algorithm. This means that we present a procedure that starts out with a TG and ends up with a RE that defines the same language.

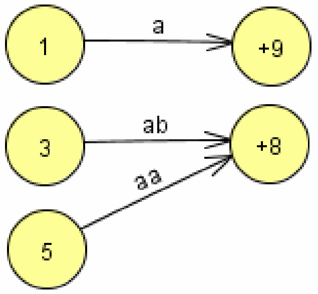
If we have many start states = become only one



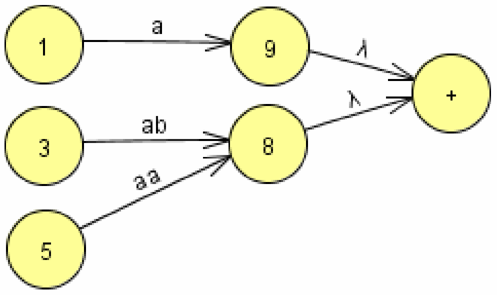
becomes



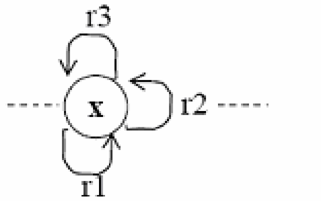
If we have many final states = become only one

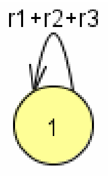


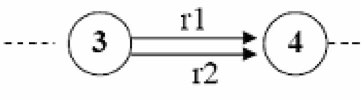
becomes



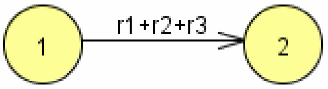
we are now going to build the RE that defines the same language as TG
reduce the number of edges or states in each time

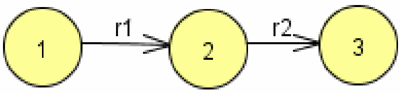




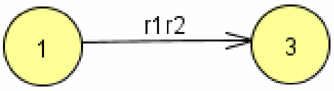


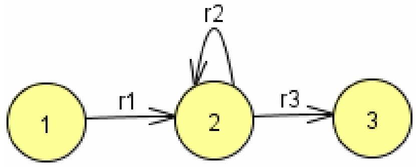
becomes



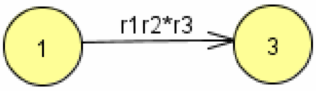


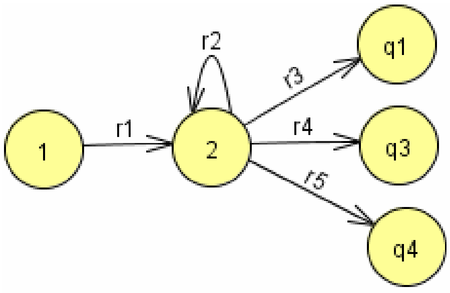
becomes

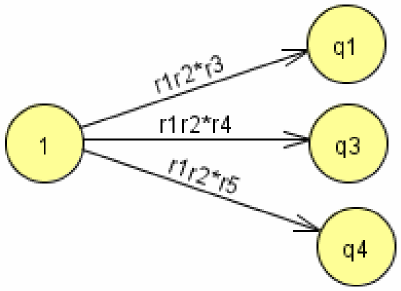




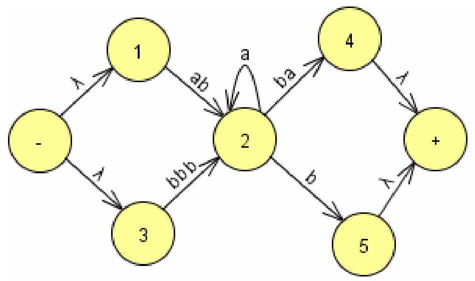
becomes



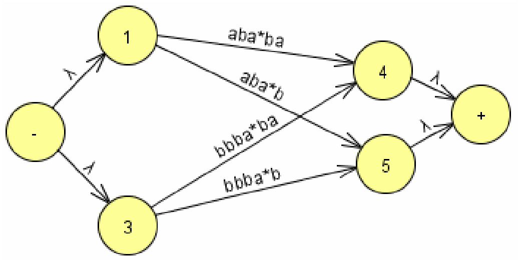




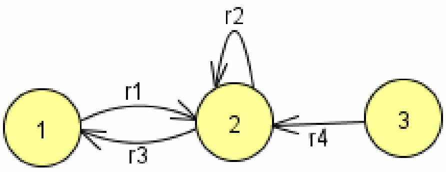
becomes



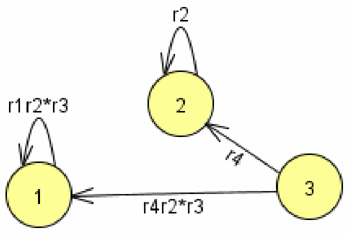
becomes



special case :



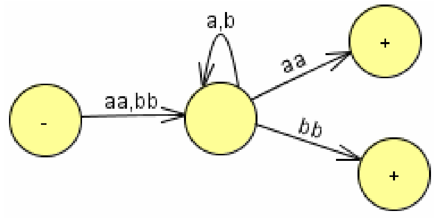
becomes

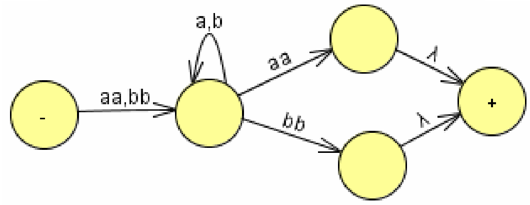


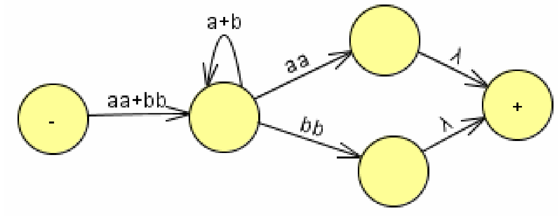
our goal: unique start state and unique final state.

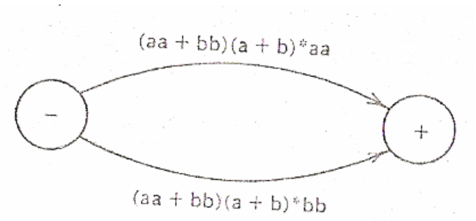
Example

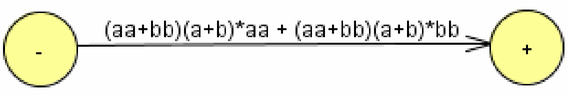
Find the RE that defines the same language accepted by the following TG using Kleenes theorem.







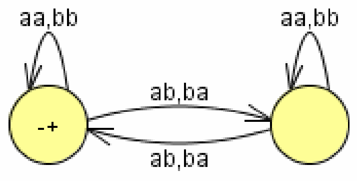


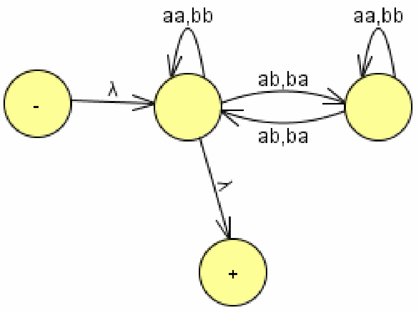


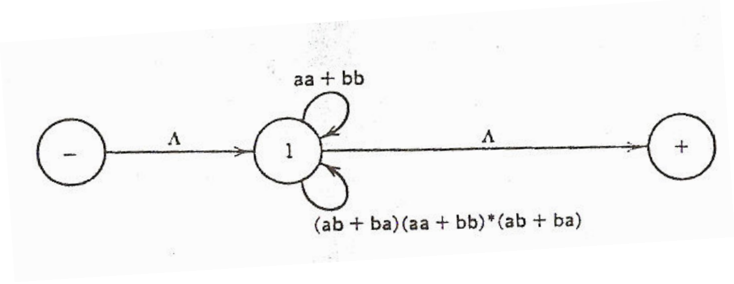
RE=(aa+bb)(a+b)*(aa+bb)

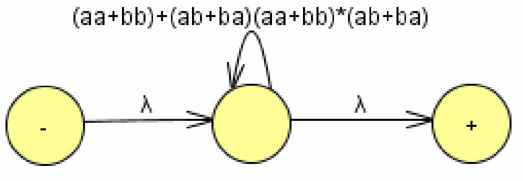
Example

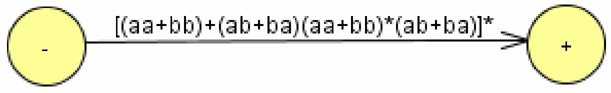
Find the RE that defines the same language accepted by the following TG using Kleenes theorem.









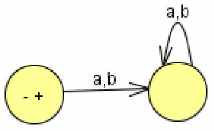


RE= [(aa+bb)+(ab+ba)(aa+bb)*(ab+ba)]*

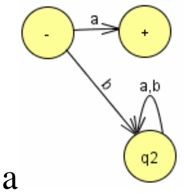
proof of part3

Rule1: there is a FA that accepts any particular letter of the alphabet.
There is an FA that accepts only the word λ.

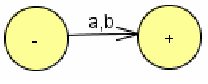
FA accepts only λ



FA accepts only a



FA accepts **a+b**

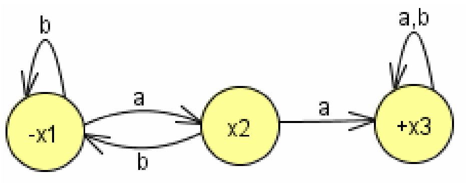


Rule2: if there is a FA called FA1, that accepts the language defined by the regular expression r1 and there is a FA called FA2, that accepts the language defined by the regular expression r2, then there is a FA calledFA3 that accepts language defined by the regular expression (r1+r2).

Example

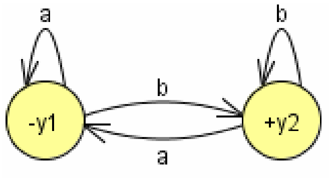
We have FA1 accepts all words with a double a in them, and FA2 accepts all words ending in b. we need to build FA3 that accepts all words that have double a or that end in b.

FA1



	a	b
-x1	X2	X1
X2	X3	X1
+x3	X3	X3

FA2

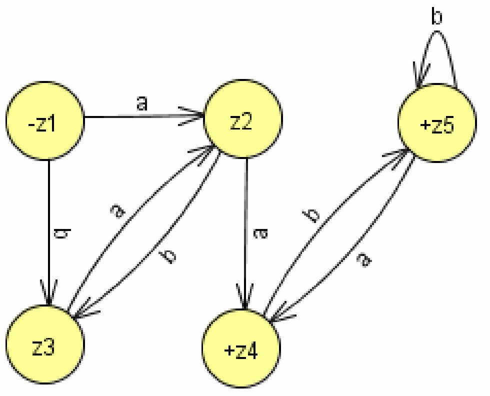


	a	b
-y1	Y1	Y2
+y2	Y1	Y2

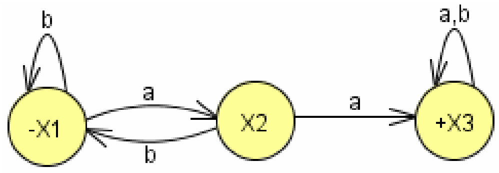
FA3

Z1 = x1 or y1
Z2 = x2 or y1
Z3 = x1 or y2
Z4 = x3 or y1
Z5 = x3 or y2

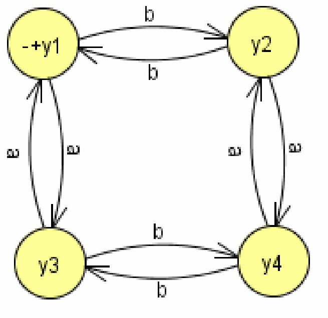
	a	b
Z1	Z2	Z3
Z2	Z4	Z3
Z3	Z2	Z3
Z4	Z4	Z5
Z5	Z4	Z5



Example



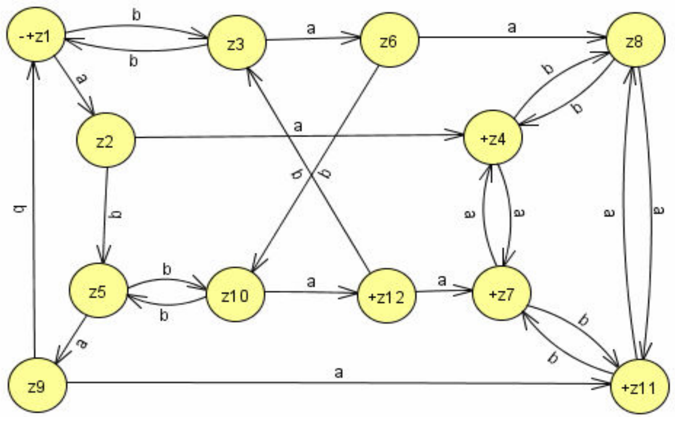
	a	b
-x1	X2	X1
x2	X3	X1
+x3	X3	X3



	a	b
-+y1	Y3	Y2
Y2	Y4	Y1
Y3	Y1	Y4
Y4	Y2	Y3

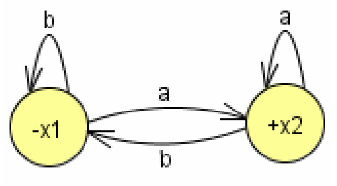
z1=x1 or y1
z2=x2 or y3
z3=x1 or y2
z4=x3 or y1
z5=x1 or y4
z6=x2 or y4
z7=x3 or y3
z8=x3 or y2
z9=x2 or y2
z10=x1 or y3
z11=x3 or y4
z12=x2 or y1

	a	b
-+z1	z2	z3
z2	Z4	z5
z3	Z6	z1
+z4	Z7	Z8
Z5	Z9	Z10
Z6	Z8	Z10
+Z7	Z4	Z11
Z8	Z11	Z4
Z9	Z11	Z1
Z10	Z12	Z5
+Z11	Z8	Z7
+Z12	z7	Z3

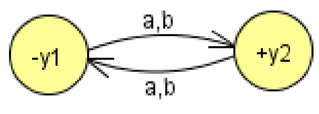


HomeWork

Let FA1 accepts all words ending in a, and let FA2 accepts all words with an odd number of letters (odd length). Build FA3 that accepts all words with odd length or end in a using Kleene's theorem.



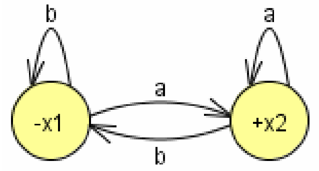
FA1



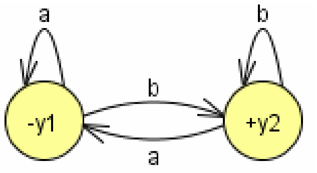
FA2

HomeWork

Let FA1 accepts all words ending in a, and let FA2 accepts all words end with b.
Build FA3 that accepts FA1+FA2 using Kleene's theorem.

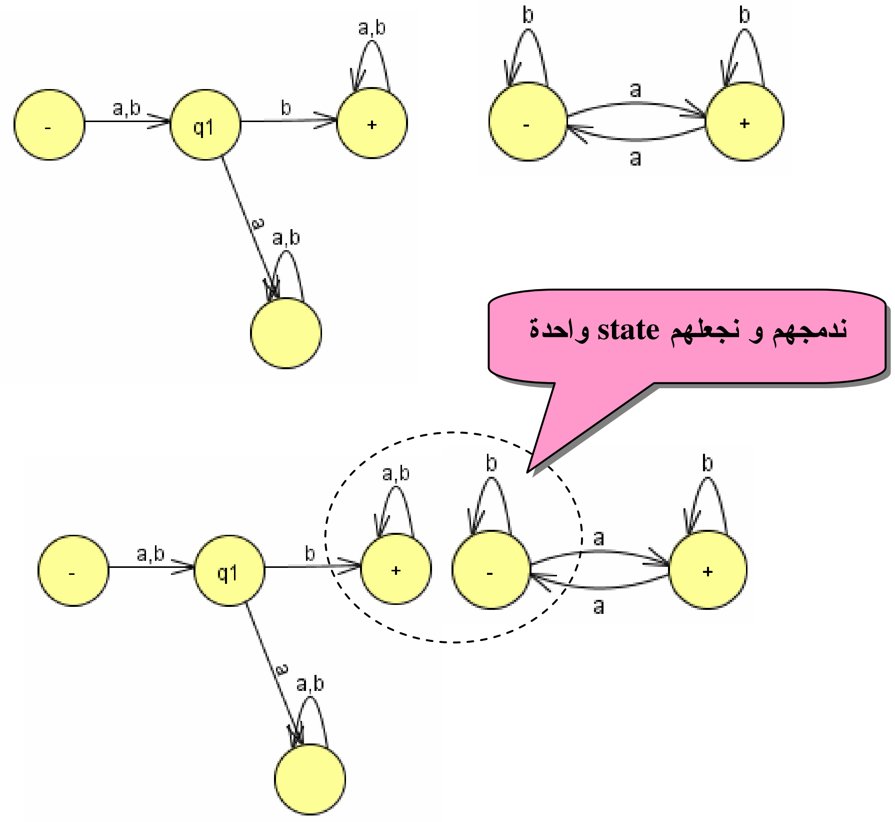


FA1



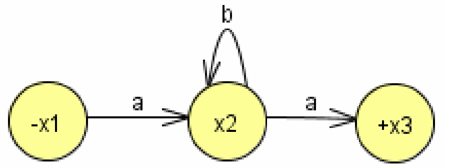
FA2

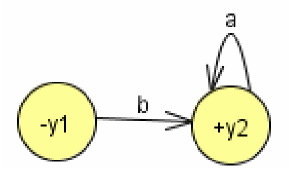
Rule3: if there is a FA1 that accepts the language defined by the regular expression r1 and a FA2 that accepts the language defined by the regular expression r2, then there is a FA3 that accepts the language defined by the concatenation r1r2.
We can describe the algorithm for forming FA3 as follows:
We make a z state for each none final x state in FA1. And for each final state in FA1 we establish a z state that expresses the options that we are continuing on FA1 or are beginning on FA2.



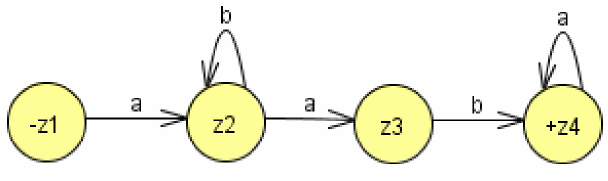
We have to connect (merge) the final state of FA1 with the start state of FA2 to produce new state (wich it is not final)

Example:





Z1=x1
Z2=x2
Z3=x3 or y1
Z4=y2

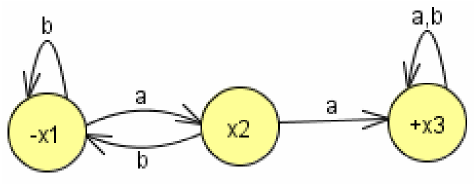


But it is not simple like that, so we have to take all possablites

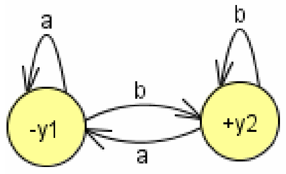
Example

We have FA1 accepts all words with a double a in them, and FA2 accepts all words ending in b. we need to build FA3 that accepts all words that have double a and end with b.

FA1



FA2

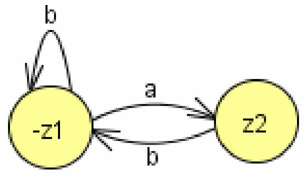


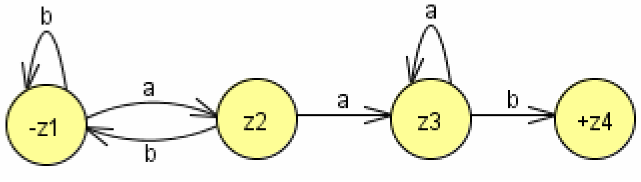
	a	B
-x1	X2	X1
X2	X3	X1
+x3	X3	X3

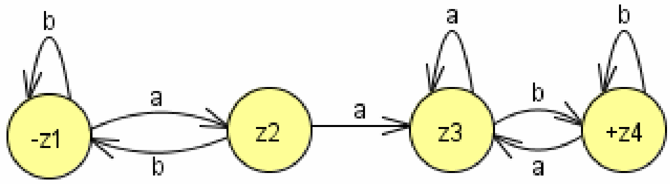
	a	B
-y1	Y1	y2
+y2	Y1	Y2

Z1= x1
Z2= x2
Z3= x3 or y1
Z4= x3 or y2 or y1

	a	b
-z1	Z2	Z1
z2	Z3	Z1
z3	Z3	Z4
+z4	Z3	Z4

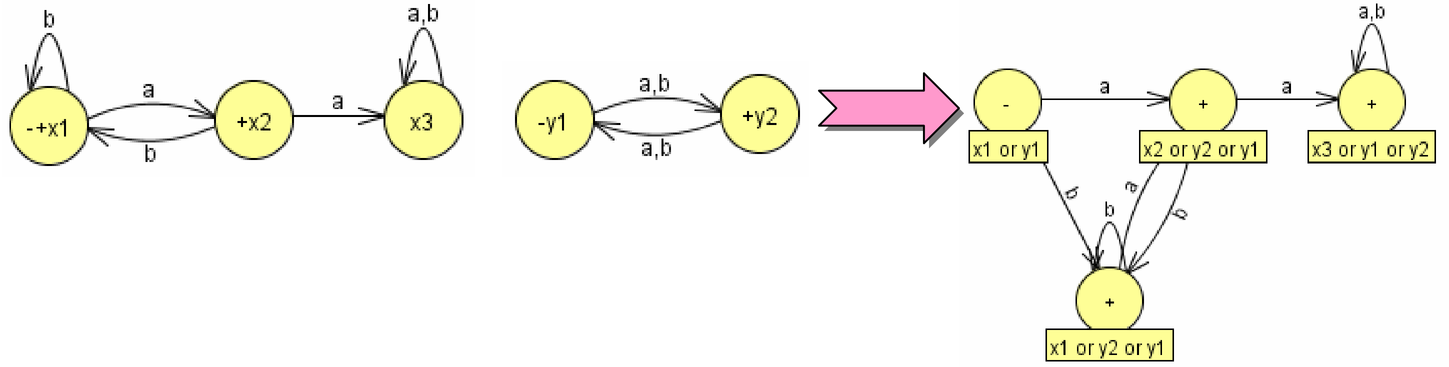






HomeWork

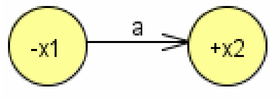
Let FA1 accepts all words with a double a in them, and let FA2 accepts all words with an odd number of letters (odd length). Build FA3 that accepts all words with odd length and have double a using Kleene's theorem.



Rule4: if r is a regular expression and FA1 accepts exactly the language defined by r, then there is an FA2 that will accept exactly the language defined by r*.
We can describe the algorithm for forming FA2 as follows:
Each z state corresponds to some collection of x states. We must remember each time we reach a final state it is possible that we have to start over again at x1.
Remember that the start state must be the final state also.

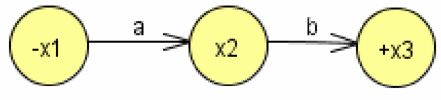
Example

If r=a find r*



Example

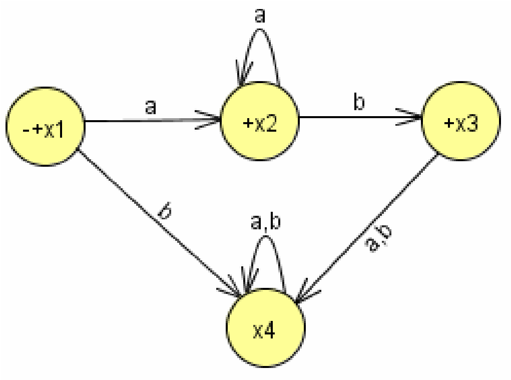
If r=ab find r*



Example

If we have FA1 that accepts the language defined by the regular expression: r=a*+aa*b
We want to build FA2 that accept the language defined by r*.

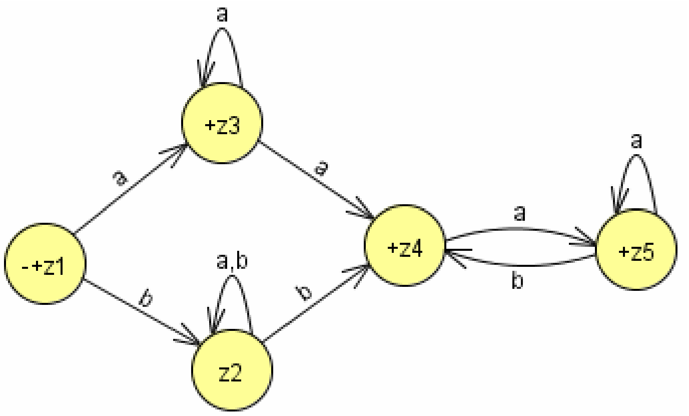
Note: We will try to connect the final state with start state.



	a	b
-+x1	X2	X4
X2	X2	X3
X3	X4	X4
X4	X4	X4

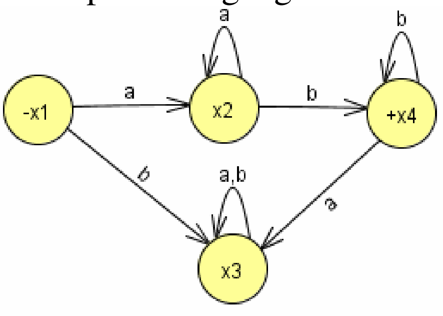
z1=x1
z2=x4
z3=x2 or x1
z4=x1 or x3 or x4
z5=x1 or x2 or x4

	a	b
-+z1	Z3	Z2
Z2	Z2	Z2
+z3	Z3	Z4
+z4	Z5	Z2
+z5	Z5	Z4



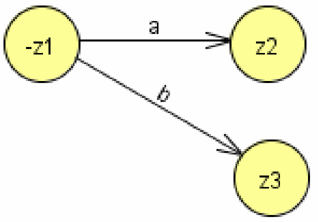
Example

Find FA2 that accept the language defined by r1* using Kleene's theorem. **r1= aa*bb***

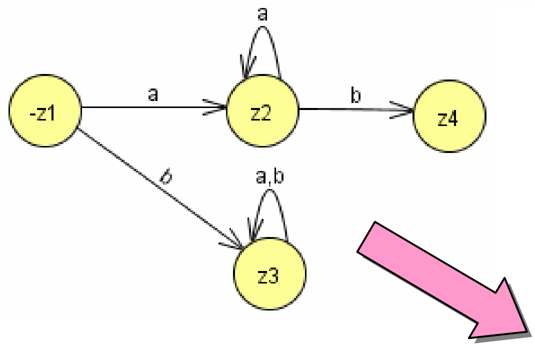


	a	b
-x1	X2	X3
X2	X2	X4
X3	X3	X3
+X4	X3	X4

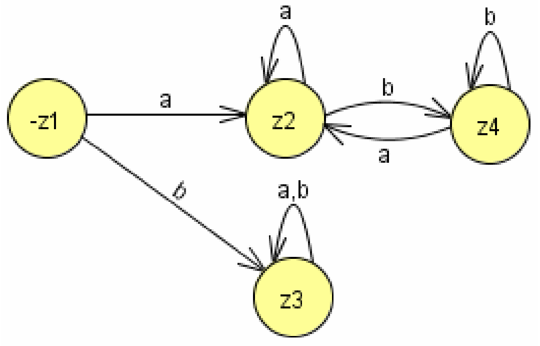
z1=x1
z2=x2
z3=x3

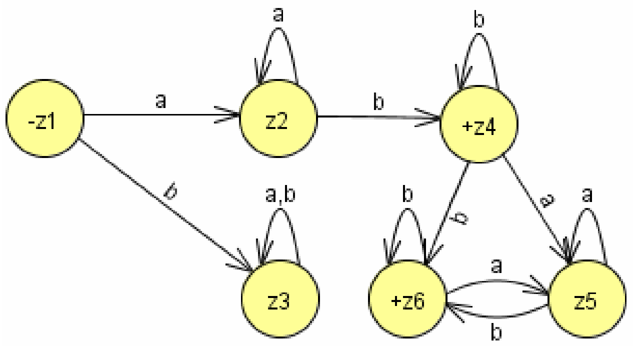


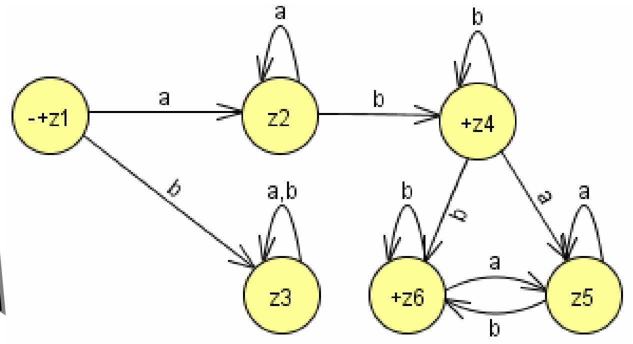
z4=x1 or x4



z5=x2 or x3
z6=x1 or x3 or x4

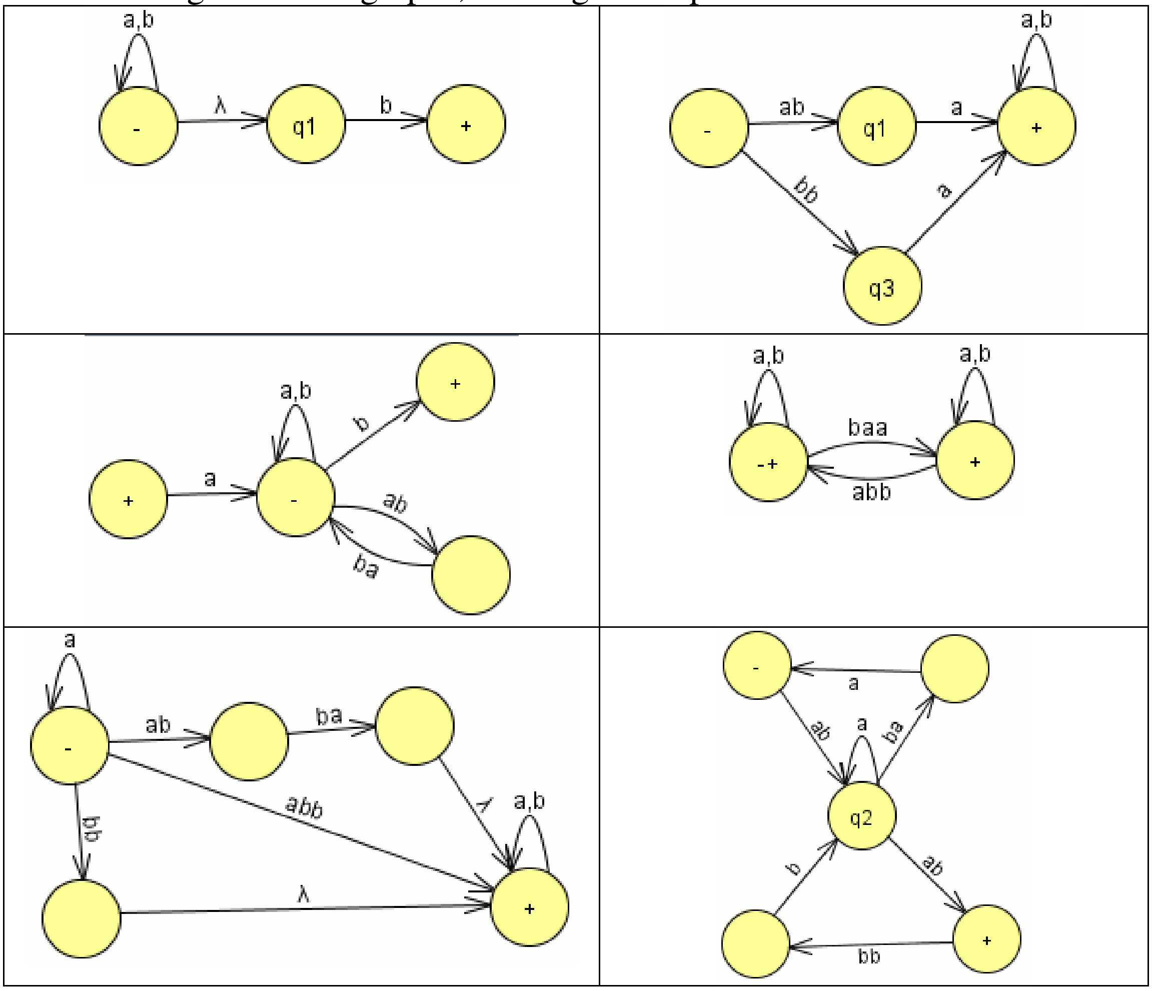




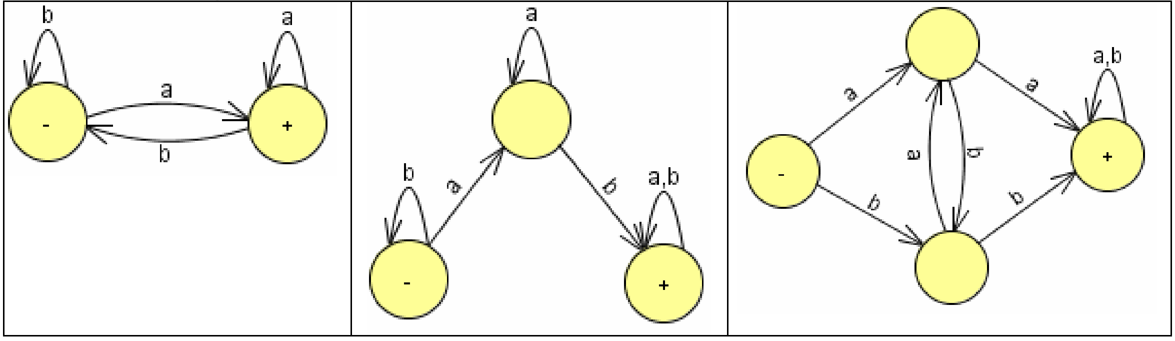


Problems

For the following transition graphs, find regular expression



Consider following FA



Find

r1+r2 r2 + r3 r1r2 r1r3 r2r1 r1r1 (r1)* (r2)* **(r1+r2)*** (r1r2)* (r2r1)*

- is r1r2 = r2r1 why ?
- is r1 + r2 = r2 + r1 why ?
- is r1r1 = (r1)* why ?

Grammars

A **grammar** is a set of rules which are used to construct a language (combine words to generate sentences).

Sentence = noun verb noun
Verb = went, eat, reading
Noun = lesson, boy, school, book

Boy reading book

may be there are sentences have no meaning = book reading boy
لذلك نحتاج القواعد لترتيب الكلمات واعطاء جمل مفهومة

G=(N, T, S, P)

N= set of **nonterminal symbols** (parts of speech (sentence, noun, verb, …))ex: S غير نهائية ويمكن اشتقاقها-حروف كبيرة

T= set of **terminal symbols** (words, or symbols in) ex: a نهائية ولا يمكن اشتقاقها-حروف صغيرة

S= **start symbol** non-terminal used to start every derivation.

P= set of productions.

Example

Start terminal nonterminal

productions: S → a S
S → λ

نتبع البداية S لنشتق الكلمات

The derivation for aaaa is:
S => aS
=> aaS
=> aaaS
=> aaaaS
=> aaaaλ = aaaa

RE= a+

Example

productions: S → SS
S → a
S → λ

يمكن كتابتها بهذا الشكل للتبسيط ➔ S → SS / a / λ

Derivation of aa
S => SS
=> SSS
=> SSa
=> SSSa
=> SaSa
=> λaSa
=> λaλa = aa

Example

S → aS | bS | a | b

Derive abbab

S => aS
=> abS
=> abbS
=> abbaS
=> abbab

Example

S → aA / bB
A → aS / a
B → bS / b
Find bbaaaa

Leftmost and rightmost derivation LMD RMD

The leftmost nonterminal (LMN) in a working string is the first nonterminal that we encounter when we scan the string from left to right.

S → aS | ab
S => aS
S => aaS
S => aaaS
S => aaaaS
S => aaaaab

Example

Consider the Grammar $G = (X, T, R, S)$ with $X = \{S, A, B, a, b\}$, $T = \{a, b\}$ and productions

S → AB
A → Aa | a
B → bBa | ba

Example

S → E
E → E + T | T
T → T*F | F
F → (E) | id

Converting Grammar into Regular Expression

Chomsky Normal Form CNF

Context free grammar (CFG) is the most important type of grammars because it is context free i.e. the right hand side is contains anything from terminal and nonterminal so that it is widely using in the programming languages to represent the language rules and grammars. It will be difficult to deal with this type of grammars because the right hand side contains everything of terminals/nonterminals, so Chomsky introduced a new formula for constrain this grammar to be :

The right hand side of a rule consists of: t / NN like the following grammar :

S → AB / BA / λ

A → AA / a

B → BB / b

Nonterminal → Nonterminal Nonterminal or **Nonterminal → terminal**

The conversion takes place in four stages.

Introduce a new start variable if the start in the right side.

1. Eliminate lambda
2. Eliminate all unit-rules: rules of the form *A* → *B*
3. Change the terminals into nonterminals
4. Reduce rules with more than tow nonterminals into tow nonterminals

Example: Convert the following grammar into CNF:

S → ASA | aB

A → B | S

B → b | λ

S' → S

S → ASA | aB

A → B | S

B → b | λ

Remove all epsilon productions, except from start variable. B ->e

S' → S

S → ASA | aB | a

A → B | S | λ

B → b | λ

Remove all epsilon productions, except from start variable. A -> λ

S' → S

S → ASA | aB | a | SA | AS | S

A → B | S | λ

B → b

Remove unit variable productions of the form S -> S

S’ → S l ASA l aB l a l SA l AS

S → ASA l aB l a l SA l AS l S

A → B l S

B → b

Remove unit variable productions of the form S’ -> S

S’ → Sl ASA l aB l a l SA l AS

S → ASA l aB l a l SA l AS

A → B l S

B → b

Remove unit variable productions of the form A -> B

S’ → Sl ASA l aB l a l SA l AS

S → ASA l aB l a l SA l AS

A → B l S l b

B → b

Remove unit variable productions of the form A -> S

S’ → Sl ASA l aB l a l SA l AS

S → ASA l aB l a l SA l AS

A → S l b l ASA l aB l a l SA l AS

B → b

S’ → Sl ASA l aB l a l SA l AS

S → ASA l aB l a l SA l AS

A → S l b l ASA l aB l a l SA l AS

B → b

Add variables and dyadic variable rules to replace any longer productions.

S’ → AA1 l UB l a l SA l AS

S → AA1 l UB l a l SA l AS

A → b l AA1 l UB l a l SA l AS

A1→ SA

U → a

B → b

Example: Convert the following grammar into CNF:

S → bA | aB

A → a | aS | bAA

B → b | bS | aBB

Step1: no lambda

Step2: no unit production

Step3: convert small into capital

S → YA / XB

A → a / XS / YAA

B → b / YS / XBB

X → a

Y → b

Step4: convert more than 2 nonterminal into 2 nonterminal

S → YA | XB

A → a | XS | YR1

B → b | YS | XR2

X → a

Y → b

R1 → AA

R2 → BB

Example: Convert the following grammar into CNF:

S → aSaS / SaSb / λ

S → SaS/SaSb/a/Sa/aS/ab/Sab/aSb 1

S → SAS/SASB/a/SA/AS/AB/SAB/ASB 3

A→a

B→b

S → SR1/SR2/a/SA/AS/AB/SR4/AR3 4

R1→ AS

R2→ AR3

R3→ SB

R4→ AB

Example

E ⟶ E + T | E – T | T

T ⟶ T*F | T/F | F

F ⟶ (E) | id

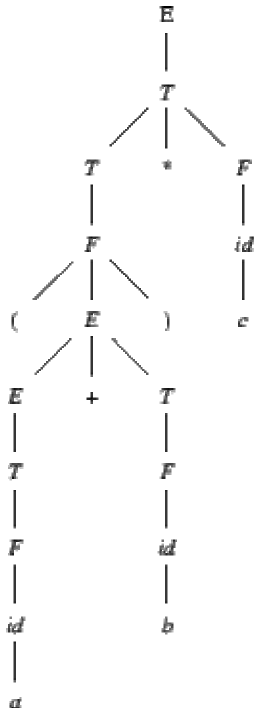
Id ⟶ a | b | c

Then the string (*a* + *b*)**c* belongs to above grammar and the derivation of this string:

E ⟶ T
⟶ T * F
⟶ F * F
⟶ (E) * F
⟶ (E + T) * F
⟶ (T + T) * F
⟶ (F + T) * F
⟶ (id + T) * F
⟶ (a + T) * F
⟶ (a + F) * F
⟶ (a + id) * F
⟶ (a + b) * F
⟶ (a + b) * id
⟶ (a + b) * c

Derivation can also be nicely represented in a tree form, as bellow

Derivation Tree for the Expression (*a* + *b*)**c*



The internal nodes of the derivation, or syntax, tree are nonterminal symbols and the frontier of the tree consists of terminal symbols. The start symbol is the root and the derived symbols are nodes. The string (a + b)*c obtained from the concatenation of the leaf nodes together from left to right.

A correct parse of the string $a + b*c$ as a sequence of shift/reduce actions is given bellow.

Parse of the expression $a + b*c$

Stack	**Input**	**Action**
\$	$a + b*c\$$	Shift
$id\$$	$+ b*c\$$	Reduce
$F\$$	$+ b*c\$$	Reduce
$T\$$	$+ b*c\$$	Reduce
$E\$$	$+ b*c\$$	Reduce
$+ E\$$	$b*c\$$	Shift
$b + E\$$	$*c\$$	Shift
$id + E\$$	$*c\$$	Reduce
$F + E\$$	$*c\$$	Reduce
$T + E\$$	$*c\$$	Reduce
$*T + E\$$	$c\$$	Shift
$c*T + E\$$	\$	Reduce
$id*T + E\$$	\$	Reduce
$F*T + E\$$	\$	Reduce
$T + E\$$	\$	Reduce
$E\$$	\$	Accept

Derivations and Parse Trees

For every derivation there is a unique corresponding parse tree.

A derivation is a leftmost derivation if the variable chosen for substitution, at any step, is the leftmost variable.

$E \rightarrow E + T \mid E - T \mid T$ we can write this grammar as:$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid id$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow id$

Derive id + id * id

$E \rightarrow E + T$

$\rightarrow T + T$

$\rightarrow F + T$

$\rightarrow id + T$

$\rightarrow id + T * F$

$\rightarrow id + F * F$

$\rightarrow id + id * F$

$\rightarrow id + id * id$

The *parse tree* of an input sequence according to a CFG is the tree of derivations. For example, the parse tree of id(x) + num(2) * id(y) is:

```
      E
    / | \
  E  +   T
  |     / | \
  T    T  *  F
  |    |     |
  F    F    id
  |    |
  id   num
```

So a parse tree has non-terminals for internal nodes and terminals for leaves.

There are tow types of parsing or derivation:

Top down parsing and Bottom up parsing

Top-down parsing starts from the start symbol of the grammar S and applies derivations until the entire input string is derived (ie, a sequence of terminals that matches the input tokens). For example,

$$\begin{aligned} E &\rightarrow E + T \\ &\rightarrow T + T \\ &\rightarrow F + T \\ &\rightarrow i + T \\ &\rightarrow i + T * F \\ &\rightarrow i + F * F \\ &\rightarrow i + i * F \\ &\rightarrow i + i * i \end{aligned}$$

Which matches the input sequence i + i * i

Bottom-up parsing starts from the input string and uses derivations in the opposite directions (ie, by replacing the right hand side sequence of a production with the nonterminal. It stops when it derives the start symbol. For example,

$$\begin{aligned} & i + i * i \\ \rightarrow\ & F + i * i \\ \rightarrow\ & T + i * i \\ \rightarrow\ & E + i * i \\ \rightarrow\ & E + F * i \\ \rightarrow\ & E + T * i \\ \rightarrow\ & T * T \\ \rightarrow\ & E * T \\ \rightarrow\ & E \end{aligned}$$

Ambiguity

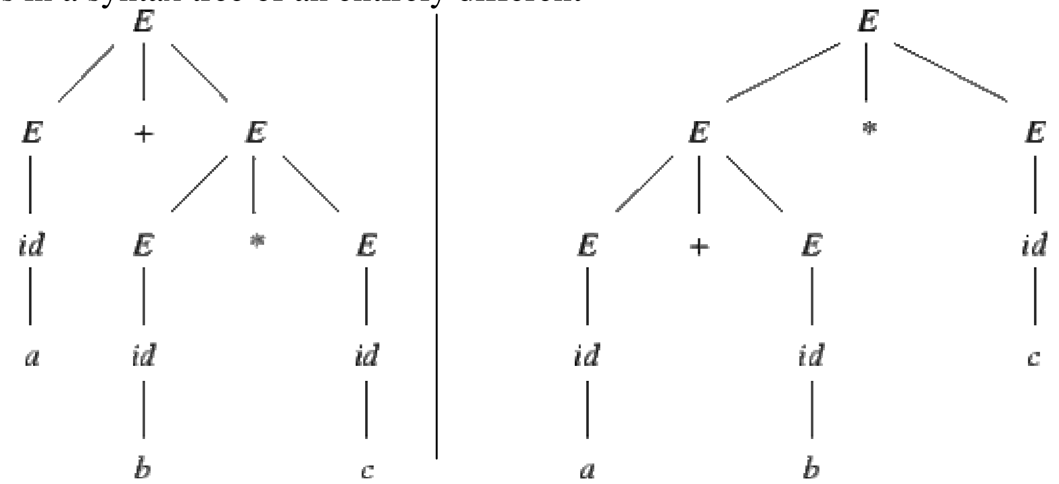
A CFG is ambiguous if it generates some string with more than one parse tree.

A grammar is *ambiguous* if it has more than one parse tree for the same input sequence.

For example, the grammar G3 is ambiguous since it has two parse trees.

A string w is derived ambiguously in CFG G if it has two or more leftmost derivations.

Suppose, that the rules of the expression grammar were written E →E + E | E*E | id, then two different syntax trees are the result. If the first production E →E + E were chosen then the result would be the tree on the left, On the other hand, choosing the production E →E*E first results in a syntax tree of an entirely different



Thus this grammar is ambiguous, because it is possible to generate two different syntax trees for the expression a + b*c.

Exercises: Convert the following grammars into CNF:

1. S →aSa | bSb | a | b | aa | bb

2. S →bA | aB
A →bAA | aS | a
B →aBB | bS | b

3. S→Aba
A →aab
B →AC

4. S →0A0 |1B1 | BB
A →C
B →S|A
C →S| λ

5. S →aAa | bBb| λ
A →C|a
B →C | b
C →CDE | λ
D →A | B | ab

6. S→abAB,
A→bAB|A,
B→ BAa|A|λ,

7. S→AB|aB
A→aab|λ
B→bbA

1- S→aSb|ab.
2- S→aSaA|A
A→abA|b

3- S→abAB,
A→bAB|A,
B→ BAa|A|λ,

5.S → aAD
A → aB | bAB
B → b
D → d

6. S → Aa | B
B → A | bb
A → a | bc | B

7.S → ASB | λ
A → aAS | a
B → SbS | A | bb

8. Show a derivation tree for the string aabbbb with the grammar
S → AB|x
A→ aB
B→ Sb

9. with the following grammar Derive 1+(0+(1+0)-1)
N → N – N / N + N / (N) / D
D → 0/1

The Chomsky Hierarchy

Noam Chomsky introduced the *Chomsky hierarchy* which classifies grammars and languages. This hierarchy can be amended by different types of machines (or automata) which can recognize the appropriate class of languages.

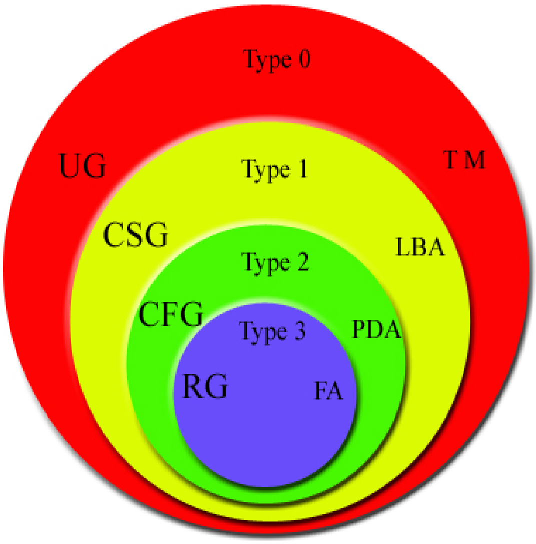
In the late fifties Noam Chomsky, a linguist at MIT, was investigating the relationship between the syntactic structure of languages and the meaning (semantics) of statements in a language.

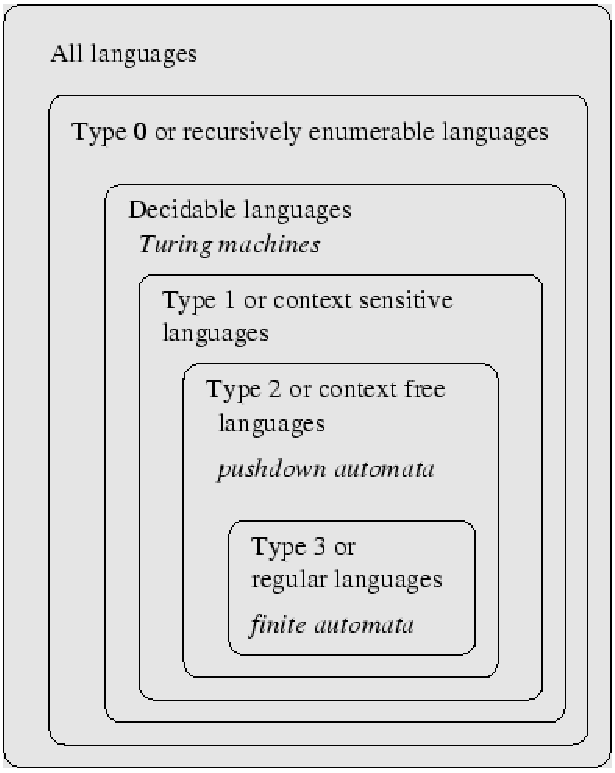
The Chomsky hierarchy comprises four types of languages and their associated grammars and machines.

Type	Language	Grammar	Machine	Example
Type 3	Regular language	Regular grammar **RG**	Finite Automata **FA**	a*b*
Type 2	Context free language	Context-free grammar **CFG**	Pushdown automaton **PDA**	$a^n b^n$
Type 1	Context sensitive language	Context sensitive grammar **CSG**	Linear bounded automaton **LBA**	$a^n b^n c^n$
Type 0	Recursively enumerable language	Unrestricted grammar **UG**	Turing machine **TM**	any computable function

regular languages ⊂context-free languages ⊂context-sensitive languages ⊂recursive enumerable languages.

Type 3 type2 type1 type0




All languages
Type 0 or recursively enumerable languages
Decidable languages
Turing machines
Type 1 or context sensitive languages
Type 2 or context free languages
pushdown automata
Type 3 or regular languages
finite automata

Type 3:

regular grammars generate the regular languages. Such aiii grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal. The rule S → λ is also here allowed if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

N → t/tN
A → a/aB
S → aS/b
Problem is left recursion A → Aa

Type 2:

context-free grammars generate the context-free languages. These are defined by rules of the form *A* → γ with *A* a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context free languages are the theoretical basis for the syntax of most programming languages.

S → (N U t)*
S → SS/aA/bA
S→λ
S→abB

Type 1:

context-sensitive grammars generate the context-sensitive languages. These grammars have rules of the form α*A*β → αγβ with *A* a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule S → ε is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a non-deterministic Turing machine whose tape is bounded by a constant times the length of the input.

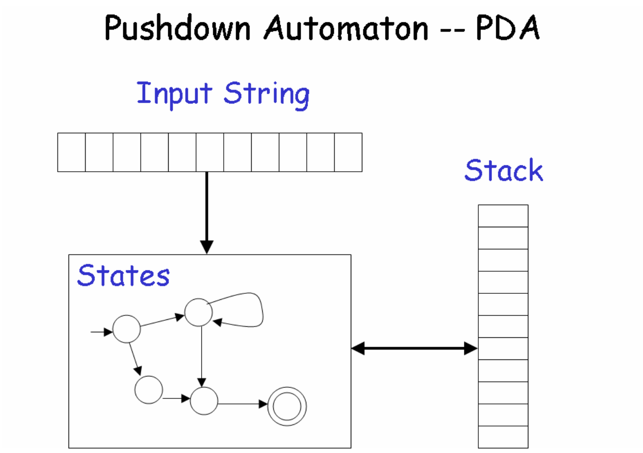
U → V ***U,V (N U t)+***
S→SS
aA→bAa
BB→aB
A→ λ wrong
Left side <= right side

Type 0:

unrestricted grammars include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. The language that is recognized by a Turing machine is defined as all the strings on which it halts. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which can be *decided* by an always halting Turing machine.

U → V ***U,V (N U t)****
S→SS
S→aAb
aA→Aa
BB→ a
aA→bAa
Ba→bAb

Push Down Automata PDA



PDA consists of 7 components (7-tuple):
M = (Q, Sigma, Gamma, delta, q0, Z0, F) where

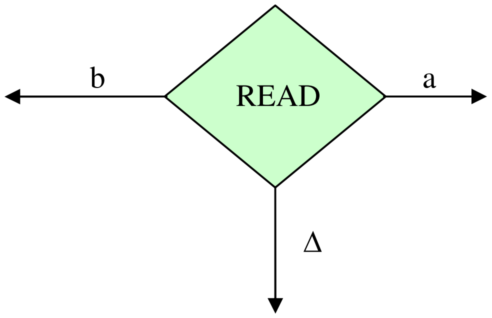
Q = a finite set of states
Sigma = a finite alphabet of input symbols (input tape)
Gamma = a finite set of push down stack symbols
Delta = a group of transitions
q0 = the initial state
Z0 = the initial stack contents - stack symbol Δ
F = the set of final, accepting, states

It is the machine that accepts CFG languages, the most important form is anbn , n>=1 i.e there is a relation between exponents of both variables and it is gives strings like aaabbbΔ, aaaaabbbbbΔ , aaaaxaaaaΔ, abbcbbaΔ Consist of basic shapes to represent PDA like triangle, diamond, trapezoidal …

START **ACCEPT** **REJECT**

The START symbol just points to the start state
The ACCEPT symbol represents a final state or accept
The REJECT symbol represents a reject state or error

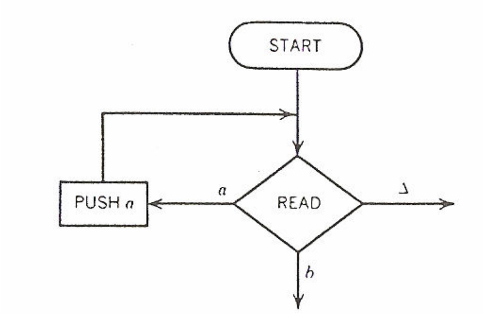
Some of states are decision state; it is represent every function performed in a state by a different type of box. The typical task performed in a state is to "read and branch" which will now be represented by a diamond shaped box, such as:



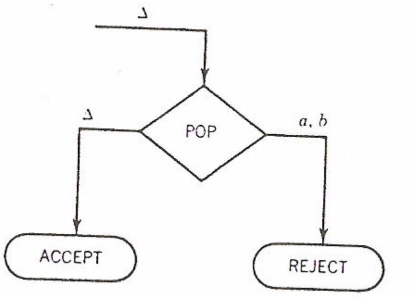
PUSH

POP

Then we have two main operations either push the input string into the stack or pop it from the stack depending on the reading string We need a type of memory which is the stack (First In Last Out) contain Δ simple to represent the empty stack



			a
		a	a
	a	a	a
Δ	Δ	Δ	Δ



We have to push the first part of string into the stack and then pop contents of the stack when we start reading the second part of the string

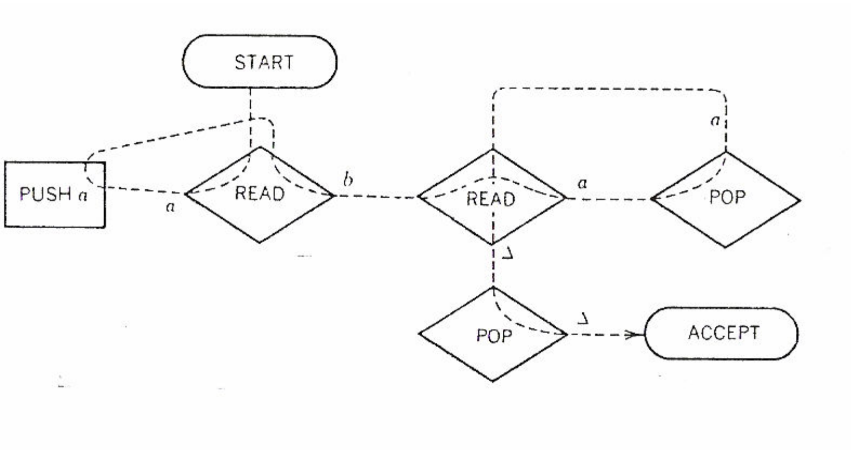
For example aaabbbΔ We will push all "aaa" into the stack then we will pop when we start reading "bbb"

We can divide string reading into tow stages

When we get the first part "aaa" we will push them into the stack

and when we read the second part "bbb" we will pop from the stack, we should get "aaa" and the stack will be empty and the string is reached to the space symbol

so if we read the space symbol we have to pop from the stack, if we got space symbol that is means the string is accepted

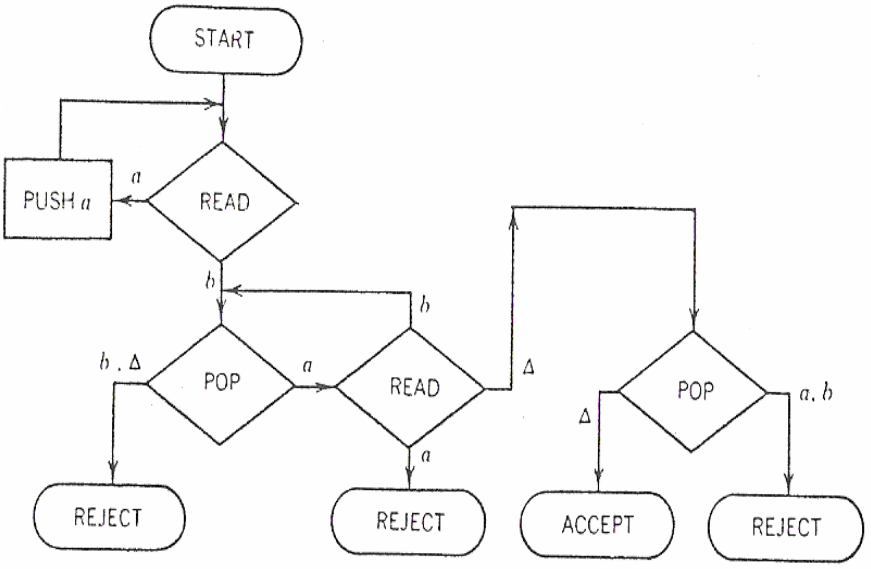


Tracing the input string on the PDA

We will trace and witching 3 variables state, stack and the tape (input string)

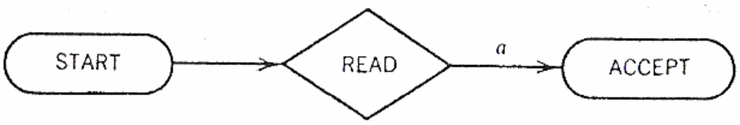
For example: trace aaabbbΔ on the PDA $a^n b^n$ n>=1

State	Stack	Tape
START	Δ	aaabbbΔ
READ	Δ	***a***aabbbΔ
PUSH	aΔ	***a***aabbbΔ
READ	aΔ	**aa**abbbΔ
PUSH	aaΔ	***aa***abbbΔ
READ	aaΔ	***aaa***bbbΔ
PUSH	aaaΔ	***aaa***bbbΔ
READ	aaaΔ	***aaab***bbΔ
POP	aaΔ	***aaab***bbΔ
READ	aaΔ	***aaabb***bΔ
POP	aΔ	***aaabb***bΔ
READ	aΔ	***aaabbb***Δ
POP	Δ	***aaabbb***Δ
READ	Δ	***aaabbb***Δ
POP	-	***aaabbb***Δ
ACCEPT		



يمكن تصميم PDA لاي تعبير RE وذلك من خلال قوراءة المدخلات فقط بدون ادخالها الى المكدس اي قراءة فقط

Example: a



Example a + b

Example (a + b)*

Example a* b

Example a b*

H.W. Design a PDA for a*b*

H.W. Design a PDA for $a^n b^n$ n>=1

H.W. Design a PDA for ca db a n>=1

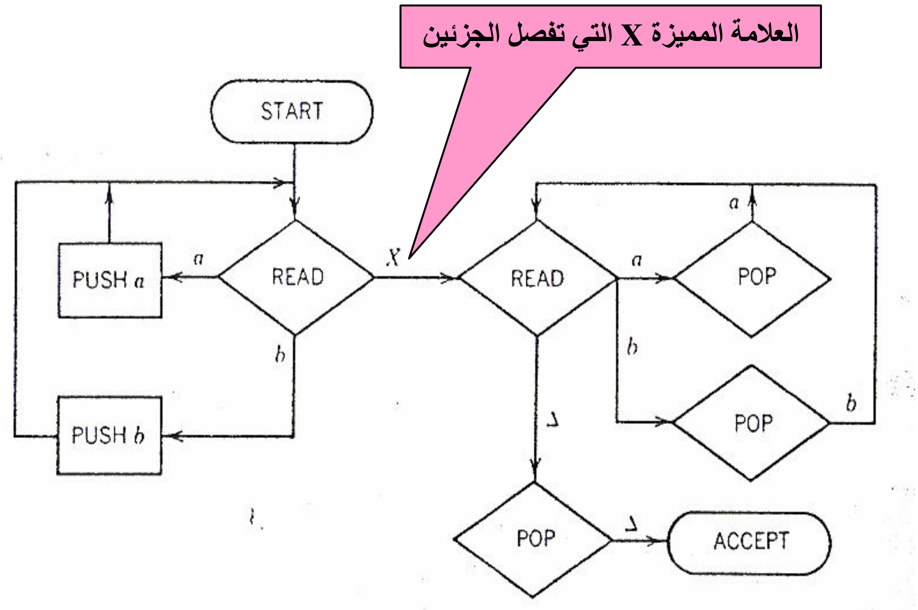
H.W. a b c d n>=1

H.W. a b c d n,m>=1

Palindrome

We can design a PDA to check the odd palindrome (string can be read from right or left) RADAR, MADAM
We will use X as a mark to distinguish the middle of the string
We will push all letters of the first part before X

Read x with no action
Then we will pop the contents of stack
Check it with the tape, if it is same
We continue pop and read,
Until we get Δ then we will pop
One time if it is Δ so we reached
ACCEPT state else go to Reject state

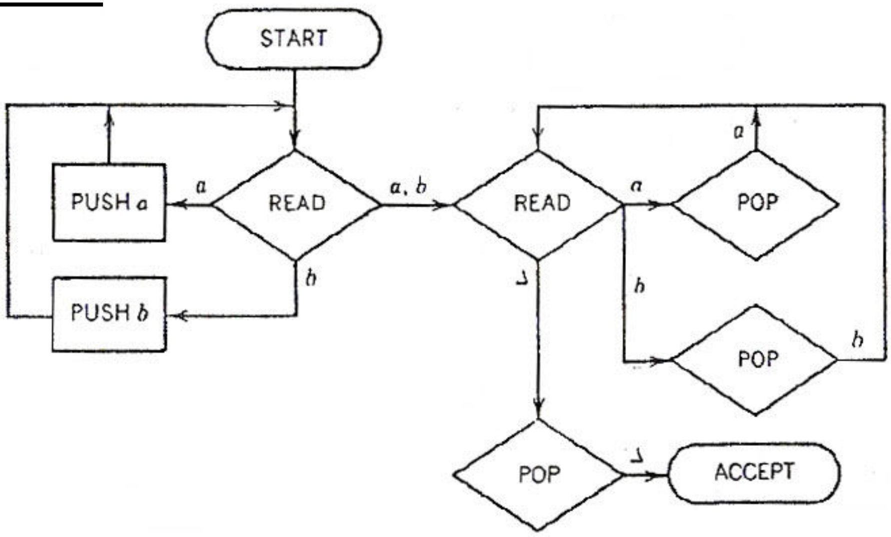


Example : aaba**X**xabaaΔ

State	Stack	Tape
START	Δ	aaaxbbbΔ
READ	Δ	***a***aaxbbbΔ
PUSH	aΔ	***a***aaxbbbΔ
READ	aΔ	**aa**axbbbΔ
PUSH	aaΔ	***aa***axbbbΔ
READ	aaΔ	***aaa***xbbbΔ
PUSH	aaaΔ	***aaa***xbbbΔ
READ	aaaΔ	***aaax***bbbΔ
READ	aaaΔ	***aaaxb***bbΔ
POP	aaΔ	***aaaxb***bbΔ
READ	aaΔ	***aaxabb***bΔ
POP	aΔ	***aaaxbb***bΔ
READ	aΔ	***aaaxbbb***Δ
POP	Δ	***aaaxbbb***Δ
READ	Δ	***aaaxbbb***Δ
POP	-	***aaaxbbb***Δ
ACCEPT		

Non deterministic palindrome NPDA

Here there is no distinguish mark,
but our palindrome is odd
(there is (a or b) in the middle)
so we will push all letters (a and b)
and then read one (a or b)
then continue reading the tape
and pop from the stack
until we get Δ

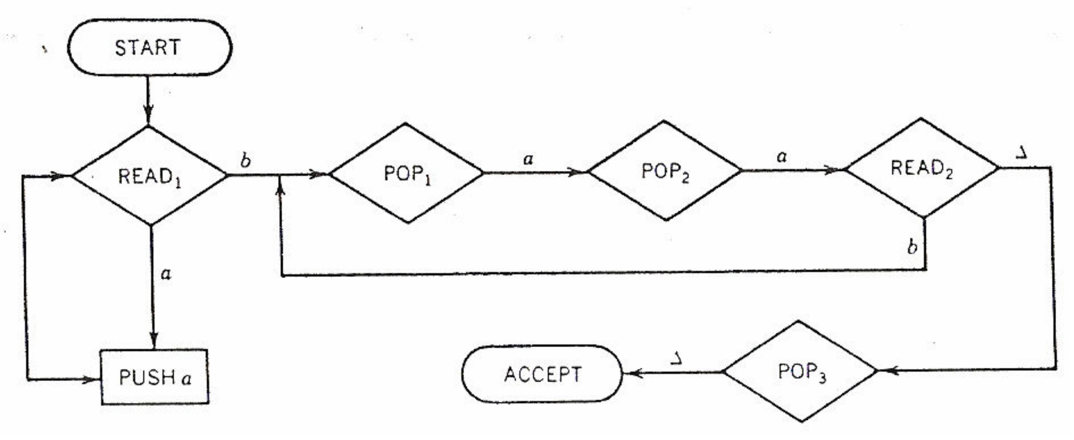


H.W.

Trace the string abbababbaΔ on the above PDA

State	**Stack**	**Tape**
START	Δ	abbababbaΔ

H.W. $a^n b^n$ n>=1 ➔ aaaaaabbbΔ



State	Stack	Tape
START	Δ	aaaaaabbbΔ

H.W.

Design a PDA for a2nbnamb2m n,m>=1

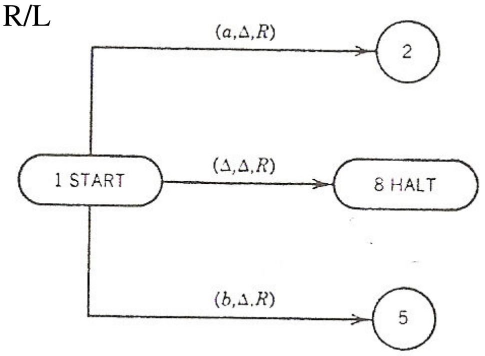
Turing Machine, TM

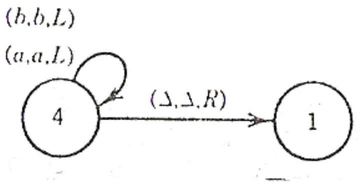
A Turing machine is defined by M = (Q, Sigma, Gamma, delta, q0, B, F) where
Q = finite set of states including q0
Sigma = finite set of input symbols not including B
Gamma = finite set of tape symbols including Sigma and B
delta = transitions mapping Q x Gamma to Q x Gamma x {L,R}
q0 = initial state
B = blank tape symbol, initially on all tape not used for input
F = set of final states

M = (Q, Sigma, Gamma, delta, q0, B, F)

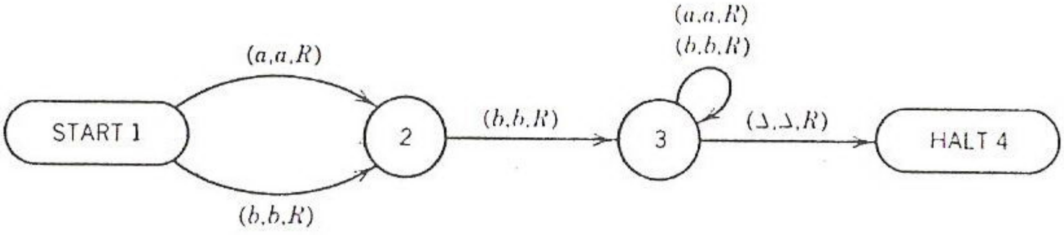
و هي الة تتميز بقرائتها للادخال ولكنها تستطيع التحرك لليمين واليسار ولذلك نستطيع رسمها على شكل حالات وبينهم اقواس للتنقل نكتب
عليها ثلاثة متغيرات الاول يمثل الادخال والثاني هو الاخراج والثالث هو الاتجاه
ونقوم بقراءة الحرف ونقوم بتغييره الى حرف كبير او نبقيه كما هو ونتجه لليمين او لليسار

It is designed to solve 3 or more of letters with same no of letters abc …. And it can be designed for any type of grammars
Read write direction
Input output direction
Tape same/different R/L

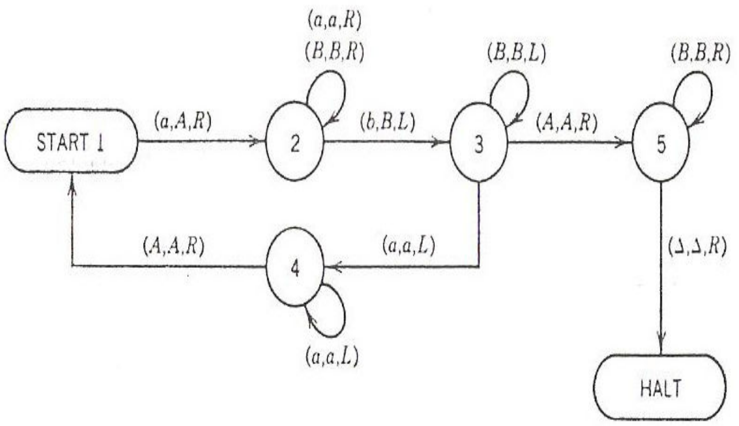




(a+b)b(a+b)*



$a^n b^n$, n>=1 aaabbbΔ

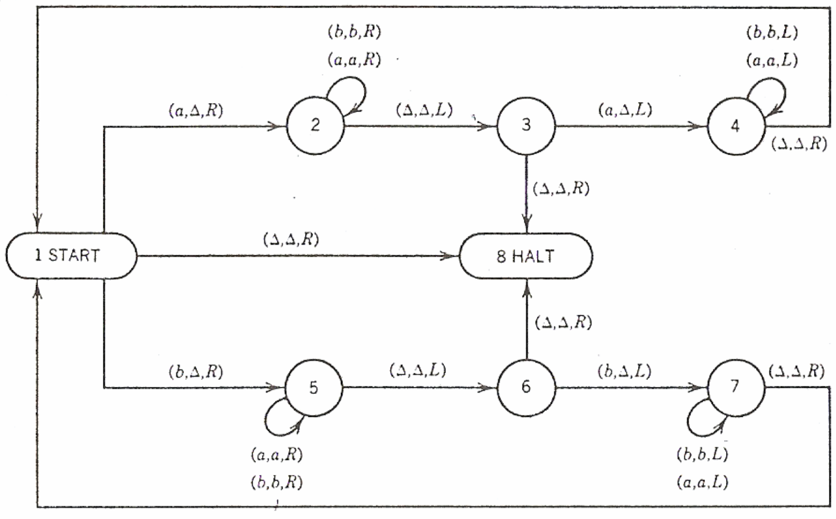


State	tape
Start	$\underline{a}$aabbbΔ
2	A$\underline{a}$abbbΔ
2	Aa$\underline{a}$bbbΔ
2	Aaa$\underline{b}$bbΔ
3	Aa$\underline{a}$BbbΔ
4	A$\underline{a}$aBbbΔ
Start	$\underline{A}$aaBbbΔ
	A$\underline{a}$aBbbΔ
	AA$\underline{a}$BbbΔ
	.
	.
	.

Palindrome

aabaabaaΔ

هنا نريد ان نتاكد من الكلمة هل يمكن قراءتها من اليمين واليسار
حيث سنقوم بتاشير اول حرف مع الاخير والثاني مع قبل الاخير و هكذا



H.W.

Design a Turing machine for $a^n b^n c^n$,n>=1

مواقع مفيدة