2nd class

Computation theory

النظريه الاحتسابيه

استاذة الماده: د. سهاد مال الله

Reference: Introduction to Computer Theory/ Cohen

Languages

Till half of this century several people define the language as a way of understanding between the same group of beings, between human beings, animals, and even the tiny beings, this definition includes all kinds of understanding, talking, special signals and voices.

This definition works till the mathematician called Chomsky said that: the language can be defined mathematically as:

1. A set of letters which called **Alphabet**. This can be seen in any natural language, for example the alphabet of English can be defined as:
   E = {a, b,…, z}
2. By concatenate letters from alphabet we get **words**.
3. All words from the alphabet make **language**.

Language can be classified into two types as follows:

$$\begin{tikzcd}
 & \text{Natural Languages} \\
\text{Languages} \arrow[ur, no head] \arrow[dr, no head] & \\
 & \text{Formal Languages}
\end{tikzcd}$$

The definition above used with natural language and formal language.

As in the natural language not all concatenations make permissible words, the same things happen with the formal languages.

Note: formal language deals with form not meaning.

You can easily distinguish from this definition that:
Alphabet: finite
Words: finite
Language: infinite

There is one thing that we must not forget it, is the alphabet could be a set of an empty set (or null string) which is a string of no letters, denoted by (Λ).

Definitions

1. **Function concatenation:** we can concatenate the word xxx with the word xx we can obtain the word xxxxx
   X^n concatenated to X^m = X^{n+m}

   <u>Example</u> let a=xx, b=xxx then ab=xxxxx

2. **Function length**: used to find the length of any word in a language.
   <u>Example</u> lets a=xxxx, b=543, c= Λ
   then length(a)=4, length(b)=3, length(c)=0

   Note: in case where parentheses are letters of the alphabet: S={x ( ) }
   Then length(xxxxx)=5
   But length((xx)(xxx))=9

3. **Function reverse**: if a is a word in some language, then reverse of a is the same string of letter spelled backward.
   <u>Example</u> lets a=xxxx, b=543, c= aab
   then reverse(a)=xxxx, reverse(b)=345, reverse(c)=baa.

   Here we want to mention that if we apply this function on words some times the result does not satisfied with the definition of the language.

   <u>Example</u> Let A an alphabet of the language L1 be {0 1 2 3 4 5 6 7 8 9}
   Let L1 = {all words that does not start with zero}
   And c=210
   Then reverse(c)=012, which is not in L1.

4. **Palindrome** : is a language that={ Λ, all strings x such that reverse(x)=x}
   <u>Example</u> aba, aabaa, bab, bbb,…

5. **Kleene star * :** Given an alphabet ∑ we wish to define a language in which any string of letters from ∑ is a word, even the null string, this language we shall call the closure of the alphabet.
   <u>Example</u> if ∑={a,b,c}
   Then ∑*={Λ a b c aa ab ac ba bb bc ca cb cc aaa aab aac bba bbb bbc cca ccb ccc aaaa aaab …}

   Lets S= alphabet of language
   S*= closure of the alphabet

Example S= {x}

S*= { Λ, x^n |n>=1}

To prove a certain word in the closure language S* we must show how it can be written as a concatenation of words from the set S.

Example Let S={a,ab}

To find if the word abaab is in S* or not, we can factor it as follows:
(ab)(a)(ab)
Every factor in this word is a word in S* so as the whole word abaab.
In the above example, there is no other way to factor this word that we called *unique*.
While, some times the word can be factored in different ways.

Example Lets S={x}

S*={ Λ, x^n| n>=1}
If we factor the word xxxxxxxx, it can be:
(xx) (xx) (xx) (xx), or
(x) (xx) (xxx) (xx), or
(xxx) (xxx) (xx), …

We can obviously say that the method of proving that something exists by showing how to create it is called *proof by constructive algorithm.*

If S=ø then S*= { Λ}
This is not the same
If S= { Λ}
Then S*={ Λ}
If S= {w_1 w_2 w_3}
Then S*= { Λ w_1 w_2 w_3 $w_1w_1w_1$ $w_1w_1w_2$ $w_1w_2w_1$…}
And S^+= { w_1 w_2 w_3 $w_1w_1w_1$ $w_1w_1w_2$ $w_1w_2w_1$…}
Which is mean that S^+=S* except the Λ

Theorem1

S*= S**

Proof

Every word in S** is made up of factors from S*. Every factor from S* is made up of factors from S. Therefore, every word in S** is made up of factors from S. Therefore, every word in S** is also a word in S*. we can write this as:
S**⊂S*
Or S*⊂S**
This mean S*=S**.

Regular Expression

The language-defining symbols we are about to create are called regular expressions. The languages that are associated with these regular expressions are called regular languages.

Example consider the language L
where L={Λ x xx xxx …} by using star notation we may write
L=language(x*).
Since x* is any string of x's (including Λ).

Example if we have the alphabet ∑={a,b}
And L={a ab abb abbb abbbb …}
Then L=language(ab*)

Example (ab)*= Λ or ab or abab or ababab or abababab or ….

Example L1=language(xx*)
The language L1 can be defined by any of the expressions:
xx* or x^+ or xx*x* or x*xx* or x^+x^* or x^*x^+ or x*x*x*xx* …
Remember x* can always be Λ.

Example language(ab*a)={aa aba abba abbba abbbba …}

Example language(a*b*)={ Λ a b aa ab bb aaa aab abb bbb … }
ba and aba are not in this language so a*b* ≠ (ab)*

Example the following expressions both define the language
L2={x^{odd}}: x(xx)* or (xx)*x
But the expression x*xx* does not since it includes the word (xx)x(x).

Example consider the language T defined over the alphabet ∑={a,b,c}
T={a c ab cb abb cbb abbb cbbb abbbb cbbbb …}
Then T=language((a+c)b*)
T=language(either a or c then some b's)

Example consider a finite language L that contains all the strings of a's and b's of length exactly three.
L={aaa aab aba abb baa bab bba bbb}
L=language((a+b)(a+b)(a+b))
L=language($(a+b)^3$)

Note from the alphabet ∑={a,b} , if we want to refer to the set of all possible strings of a's and b's of any length (including Λ) we could write **(a+b)***

Example we can describe all words that begins with a and end with b with the expression *a(a+b)*b* which mean *a(arbitrary string)b*

Example if we have the expression *(a+b)*a(a+b)** then the word abbaab can be considerd to be of this form in three ways: (Λ)a(bbaab) or (abb)a(ab) or (abba)a(b)

Example (a+b)*a(a+b)*a(a+b)*
= (some beginning)(the first important a)(some middle)(the second important a)(some end)

Another expressions that denote all the words with at least two a's are:
b*ab*a(a+b)*, (a+b)*ab*ab*, b*a(a+b)*ab*

Then we could write:
language((a+b)*a(a+b)*a(a+b)*)
=language(b*ab*a(a+b)*)
=language((a+b)*ab*ab*)
=language(b*a(a+b)*ab*)
=all words with at least two a's.

Note: we say that two regular expressions are equivalent if they describe the same language.

Example if we want all the words with exactly two a's, we could use the expression: *b*ab*ab** which describe such words as aab, baba, bbbabbabbbb,…

Example the language of all words that have at least one a and at least one b is: *(a+b)*a(a+b)*b(a+b)*+(a+b)*b(a+b)*a(a+b)**

Note: (a+b)*b(a+b)*a(a+b)* ≠ bb*aa* since the left includes the word aba, which the expression on the right side does not.

Note: (a+b)* = (a+b)* + (a+b)*
(a+b)* = (a+b)*(a+b)*
(a+b)* = a(a+b)* + b(a+b)* + Λ
(a+b)* = (a+b)*ab(a+b)* + b*a*

Note: usually when we employ the star operation we are defining an infinite language. We can represent a finite language by using the plus alone.

Example L={abba baaa bbbb}
L=language(abba + baaa + bbbb)

Example L={ Λ a aa bbb}
L=language(Λ + a + aa + bbb)

Example L={Λ a b ab bb abb bbb abbb bbbb …}
We can define L by using the expression *b* + ab**

Definition

The set of regular expressions is defined by the following rules:

Rule1: every letter of ∑ can be made into a regular expression, Λ is a regular expression.
Rule2: if r1 and r2 are regular expressions, then so are: (r1) r1r2 r1+r2 r1*.
Rule3: nothing else is a regular expression.

Remember that $r1^+$=r1r1*

Definition

If S and T are sets of strings of letters (whether they are finite or infinite sets), we define the product set of strings of letters to be:
ST={all combination of a string from S concatenated with a string from T}

Example if S={a aa aaa} T={bb bbb}
Then ST={abb abbb aabb aabbb aaabb aaabbb}
(a+aa+aaa)(bb+bbb)=abb+abbb+ aabb+aabbb+aaabb+aaabbb)

Example if P={a bb bab} Q={Λ bbbb}
Then PQ={a bb bab abbbb bbbbbb babbbbb}
(a+bb+bab)(Λ+bbbb)=a+bb+bab+ab^4+b^6+bab^5

Example if M={Λ x xx} N={Λ y yy yyy yyyy …}
Then MN={Λ y yy yyy yyyy …
x xy xyy xyyy xyyyy …
xx xxy xxyy xxyyy xxyyyy …}
Using regular expression we could write:
(Λ+x+xx)(y*)=y*+xy*+xxy*

Definition

The following rules define the language associated with any regular expression.

Rule1: the language associated with the regular expression that is just a single letter is that one-letter word alone and the language associated with Λ is just{Λ}, a one-word language.

Rule2: if r1 is regular expression associated with the language L1 and r2 is regular expression associated with the language L2 then:

i) The regular expression (r1)(r2) is associated with the language L1 times L2.

Language(r1r2)=L1L2

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

Language(r1+r2)=L1+L2

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

Language(r1)*=L1*

Example L={baa abba bababa}

The regular expression for this language is: (baa+abba+bababa)

Example L={Λ x xx xxx xxxx xxxxx}

The regular expression for this language is: (Λ+x+xx+xxx+xxxx+xxxxx) $=(\Lambda+x)^5$

Example L= language((a+b)*(aa+bb)(a+b)*)

=(arbitrary)(double letter)(arbitrary)

{Λ a b ab ba aba bab abab baba …} these words are not included in L but they included by the regular expression: (Λ+b)(ab)*(Λ+a)

Example

E=(a+b)*a(a+b)*(a+Λ)(a+b)*a(a+b)*

E=(a+b)*a(a+b)*a(a+b)*a(a+b)*+(a+b)*a(a+b)*Λ(a+b)*a(a+b)*

We have: (a+b)*Λ(a+b)*=(a+b)*

Then: E=(a+b)*a(a+b)*a(a+b)*a(a+b)*+(a+b)*a(a+b)*a(a+b)*

The language associated with E is not different from the language associated with: (a+b)*a(a+b)*a(a+b)*

Note: (a+b*)*=(a+b)*

(a*)*=a*

(aa+ab*)*≠ (aa+ab)*

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

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

Even-even={Λ aa bb aabb abab abba baab baba bbaa aaaabb aaabab…}

Finite Automata (FA)

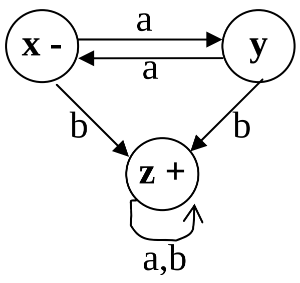
A finite automata is a collection of three things:

1. A finite set of states, one of which is designed as the initial state, called the start state, and some (may be none) of which are designed as final states.
2. An alphabet ∑ of possible input letters from which are formed strings that are to be read one letter at a time.
3. A finite set of transitions that tell for each state and for each letter of the input alphabet which state to go to next.

<u>Example</u> if ∑={a,b}, states={x,y,z}
Rules of transition:

1. From state x and input a go to state y.
2. From state x and input b go to state z.
3. From state y and input a go to state x.
4. From state y and input b go to state z.
5. From state z and any input stay at the state z.

Let x be the start state and z be the final state.



Transition Diagram

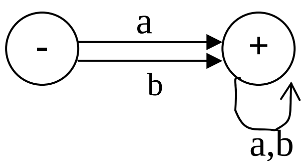
The FA above will accept all strings that have the letter b in them and no other strings. The language associated with(or accepted by) this FA is the one defined by the regular expression: *a*b(a+b)**

The set of all strings that do leave us in a final state is called the language defined by the FA. The word abb is accepted by this FA, but The word aaa is not.

	a	B
x -	y	Z
y	x	Z
z +	z	Z

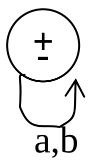
Transition Table

Example The following FA accept all strings from the alphabet {a,b} except Λ.



The regular expression is: $(a+b)(a+b)^*=(a+b)^+$

Example The following FA accept all words from the alphabet {a,b}.

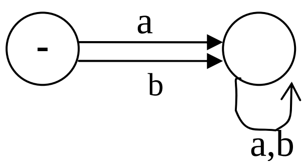


The regular expression is: $(a+b)^*$

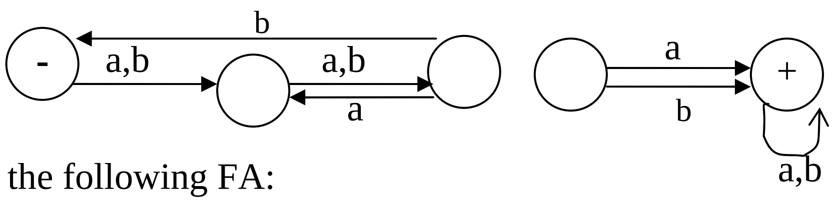
Note: every language that can be accepted by an FA can be defined by a regular expression and every language that can be defined by a regular expression can be accepted by some FA.

FA that accepts no language will be one of the two types:

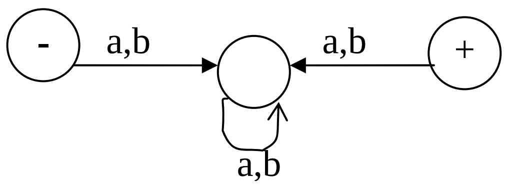
1. FA that have no final states. Like the following FA:



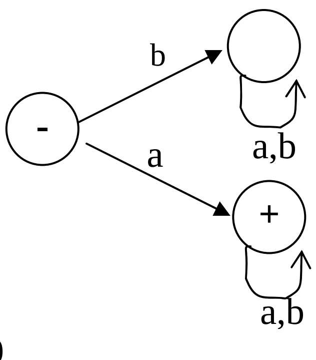
2. FA in which the final states cannot be reached. Like the following FA:



Or Like the following FA:

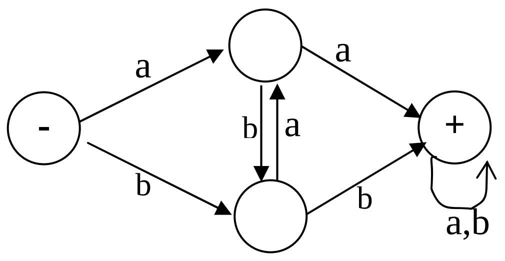


Example The following FA accept all strings from the alphabet {a,b} that start with a.



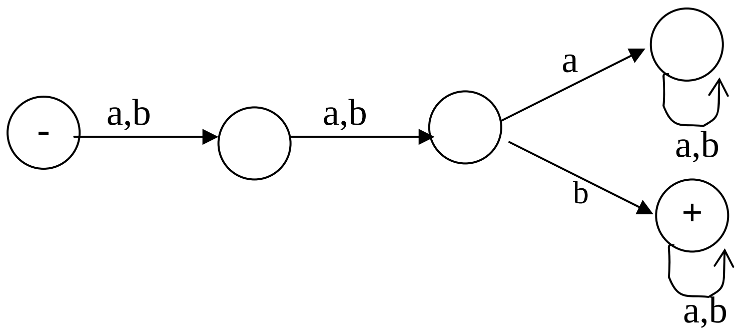
The regular expression is: $a(a+b)^*$

Example The following FA accept all strings from the alphabet {a,b} with double letter.

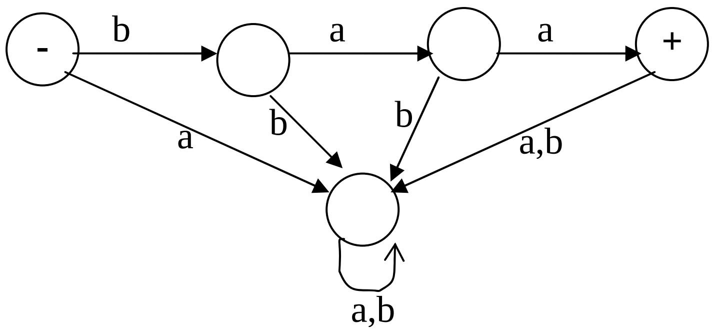


The regular expression is: (a+b)*(aa+bb) (a+b)*

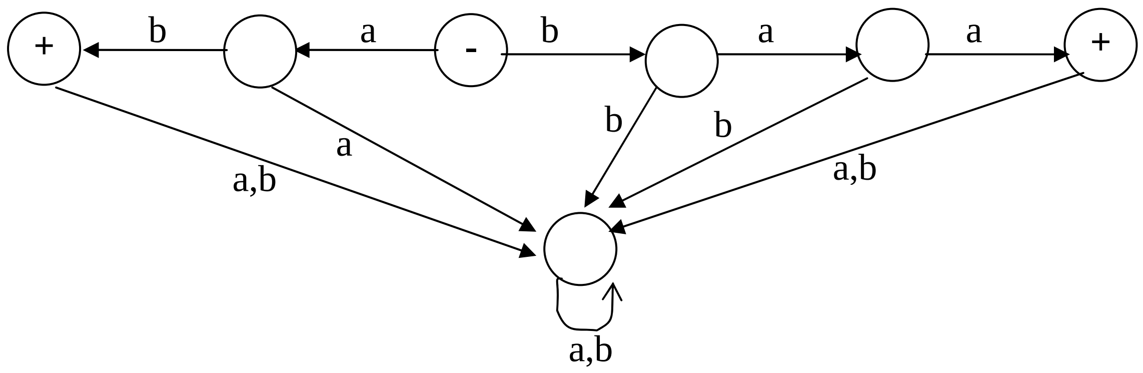
Example the following FA accepts the language defined by the regular expression: *(a+b)(a+b)b(a+b)**



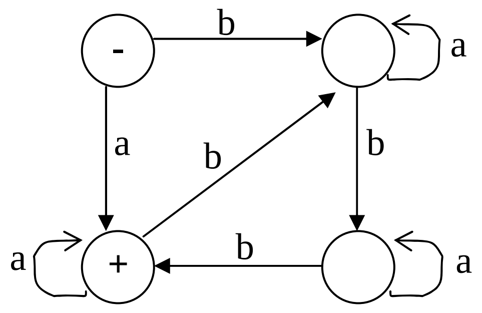
Example the following FA accepts only the word baa.



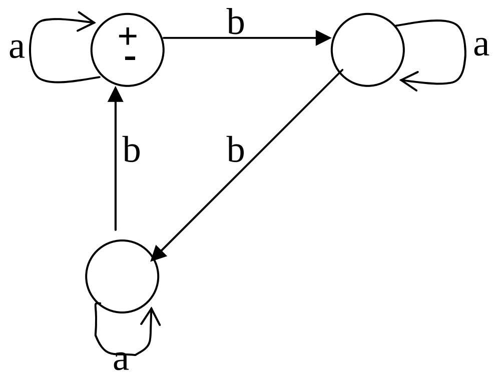
Example the following FA accepts the words baa and ab.



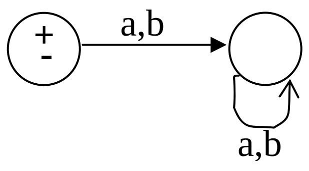
__Example__ the following FA accepts the language defined by the regular expression: $(a+ba^*ba^*b)^+$



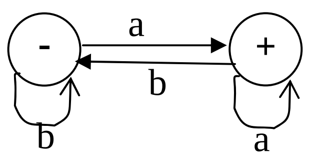
__Example__ the following FA accepts the language defined by the regular expression: $(a+ba^*ba^*b)^*$



__Example__ the following FA accepts only the word Λ.

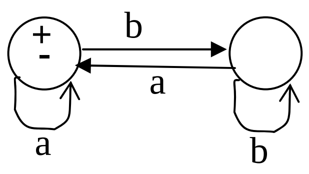


__Example__ the following FA accepts all words from the alphabet {a,b} that end with a.



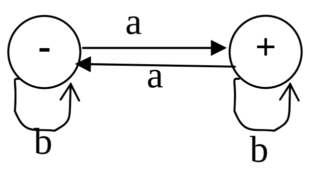
The regular expression for this language is: $(a+b)^*a$

Example the following FA accepts all words from the alphabet {a,b} that do not end in b and accept Λ.



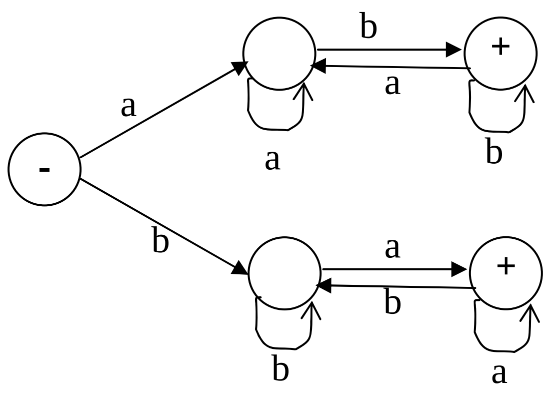
The regular expression for this language is: (a+b)*a + Λ

Example the following FA accepts all words from the alphabet {a,b} with an odd number of a's.



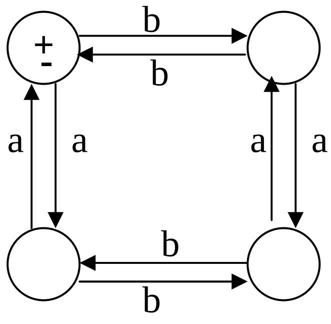
The regular expression for this language is: b*a(b*ab*ab*)*

Example the following FA accepts all words from the alphabet {a,b} that have different first and last letters.



The regular expression for this language is: a(a+b)*b + b(a+b)*a

Example the following FA accepts the language defined by the regular expression (even-even): [aa+bb+(ab+ba)(aa+bb)*(ab+ba)]*

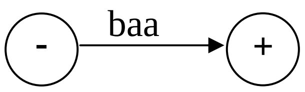


Transition Graph

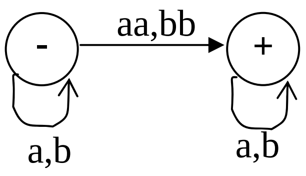
A Transition Graph (TG) is a collection of three things:

1. A finite set of states, at least one of which is designed as the start state (-) and some (may be none) of which are designed as final states (+).
2. An alphabet ∑ of possible input letters from which input string are formed.
3. A 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 Λ).

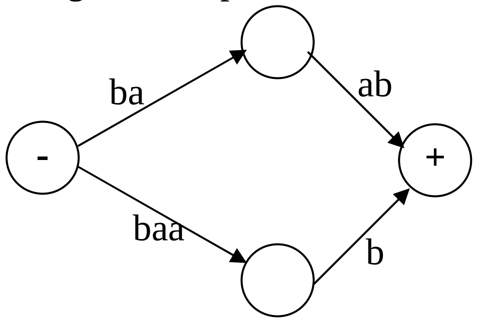
Example the following TG accepts the word baa.



Example the following TG accepts the words with double letters.



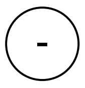
Example the following TG accepts the word baab in two different ways.



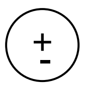
Note: in TG some words have several paths accept them while in FA there is only one

Note: every FA is also a TG.

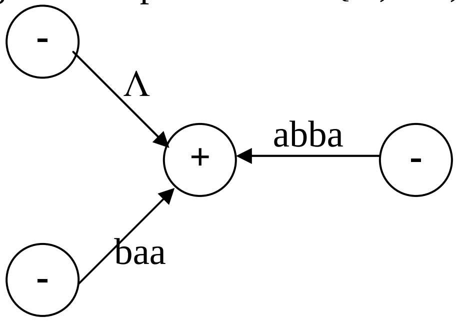
Example the following TG accept nothing.



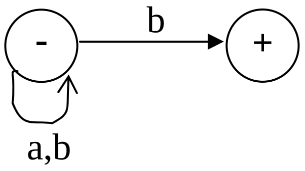
Example the following TG accept Λ.



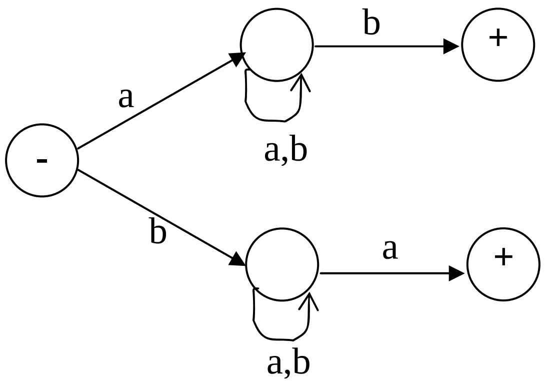
Example the following TG accept the words {Λ, baa, abba}.



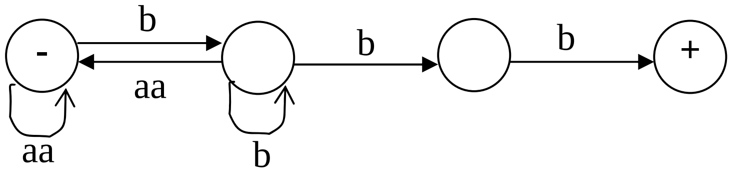
Example the following TG accept all words that end with b.



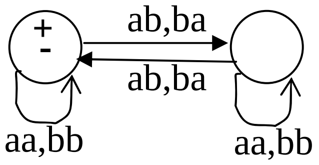
Example the following TG accept all words that have different first and last letters.



Example the following TG accept all words in which a's occur in even clumps only and end in three or more b's.



Example the following TG for even-even.



Kleen's Theorem

Kleen's Theorem

Any language that can be defined by:
1. Regular expression
or 2. Finite automata
or 3. Transition graph

Can be defined by all three methods.

Proof

The three sections of our proof will be:

Part1: every language that can be defined by a FA can also be defined by a TG.

Part2: every language that can be defined by a TG can also be defined by a RE.

Part3: every language that can be defined by a RE can also be defined by a FA.

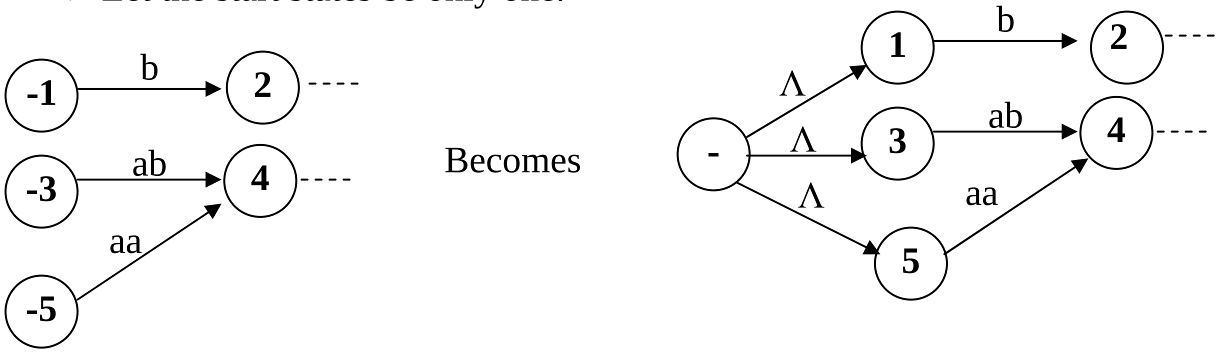
The proof of part1

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

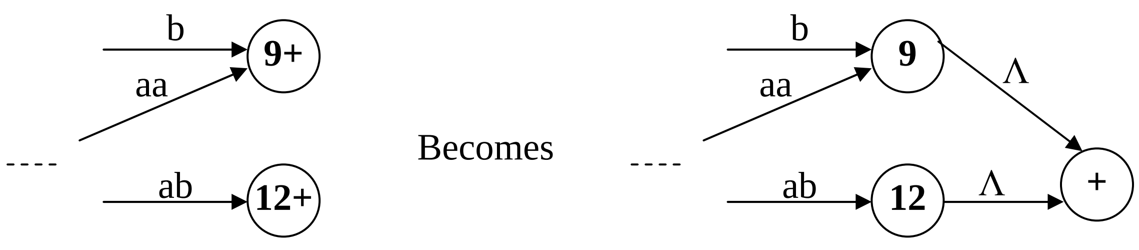
The proof of part2

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.

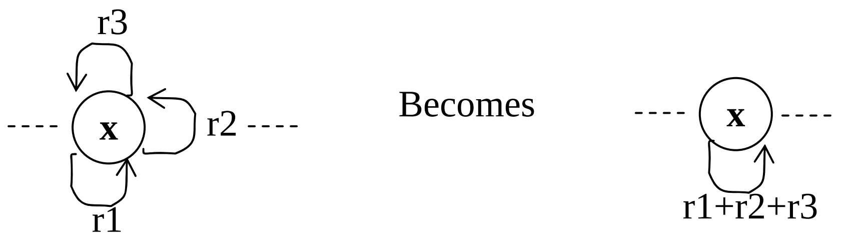
❖ Let the start states be only one.

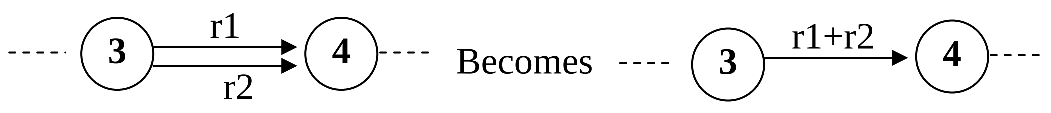


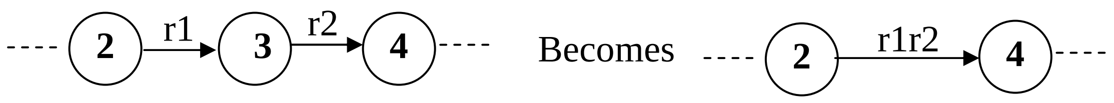
❖ Let the final states be only one.

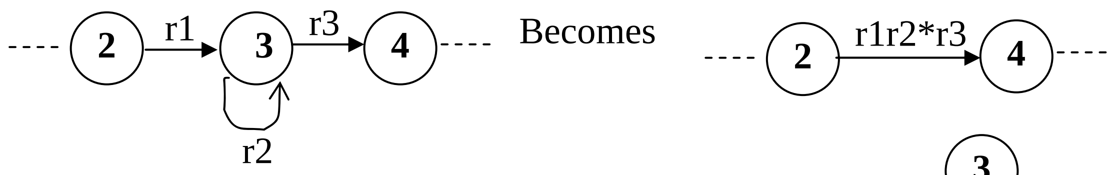


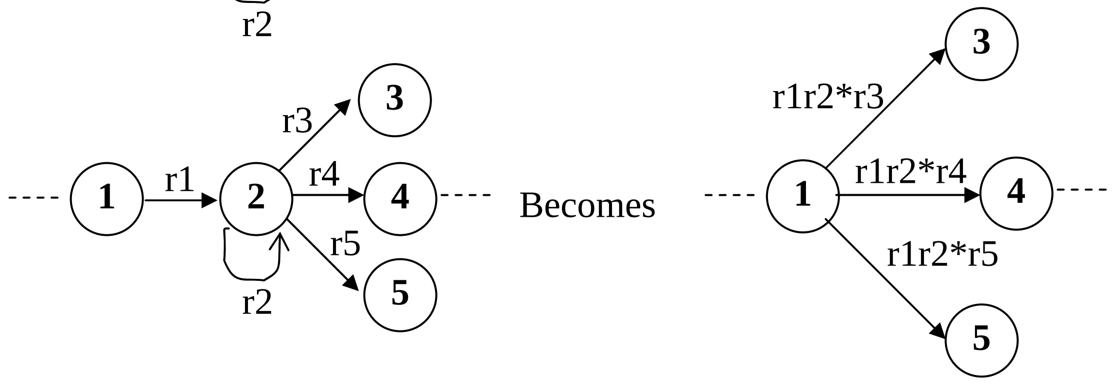
❖ Allow the edges to be labeled with regular expressions (reduce the number of edges or states in each time).

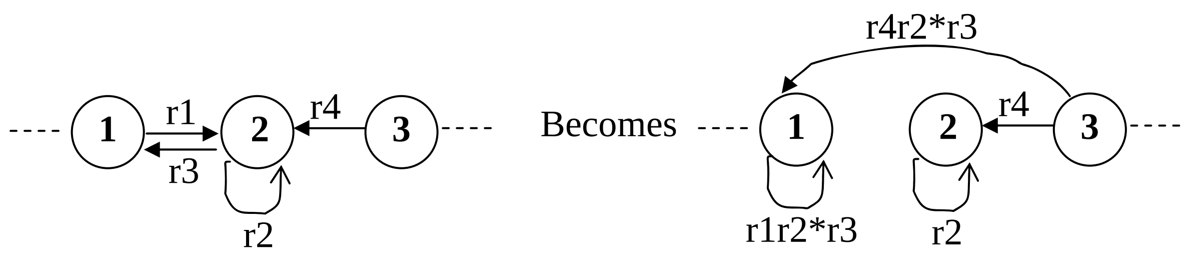




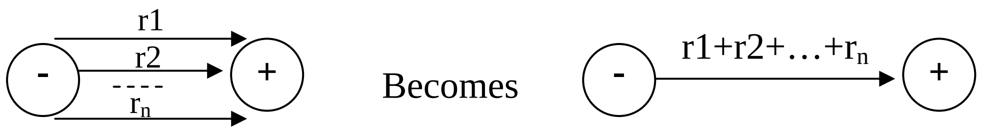






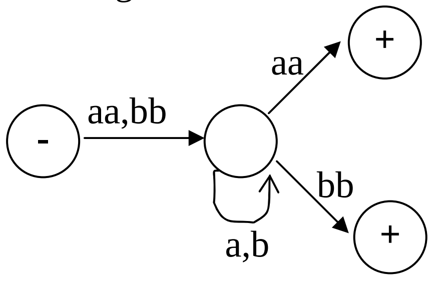


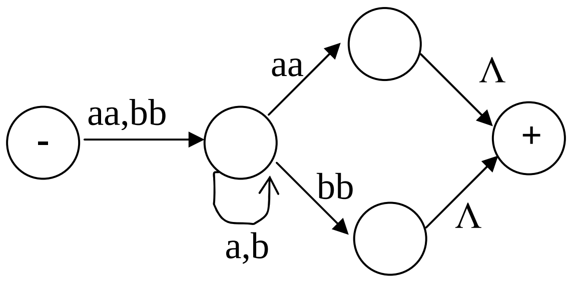
❖ Repeat the last step again and again until we eliminate all the states from TG except the unique start state and the unique final state.

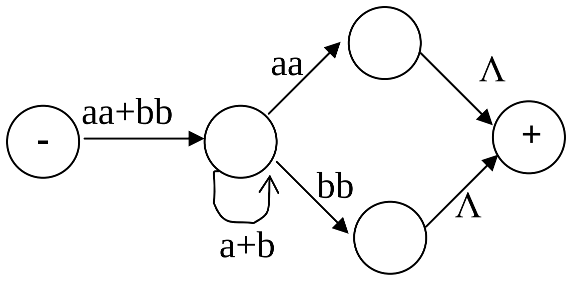


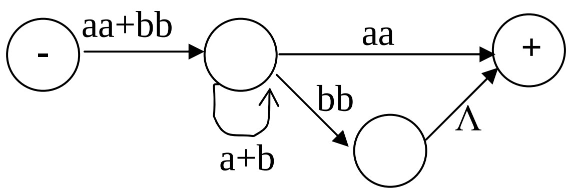
<u>Example</u>

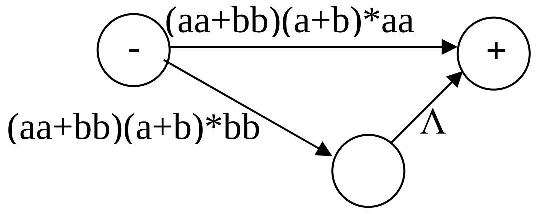
Find the RE that defines the same language accepted by the following TG using Kleenes theorem.

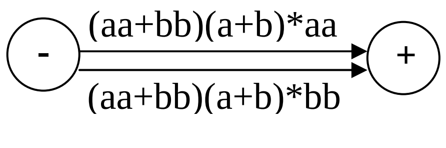








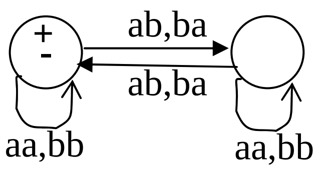




RE=(aa+bb)(a+b)*(aa+bb)

<u>H.W</u>

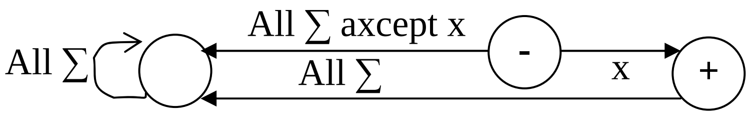
Find the RE that defines the same language accepted by the following TG using Kleenes theorem.



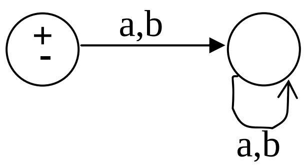
<u>The proof of part3</u>

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

For example, if x is in ∑, then the FA will be:



FA accepts only Λ will be:



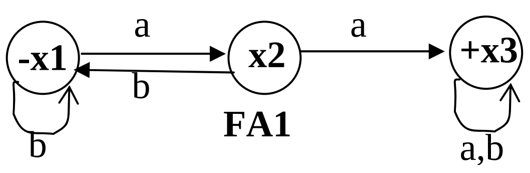
Rule2*: if there is an FA called FA1, that accepts the language defined by the regular expression r1 and there is an FA called FA2, that accepts the language defined by the regular expression r2, then there is an FA called FA3 that accepts the language defined by the regular expression (r1+r2).*

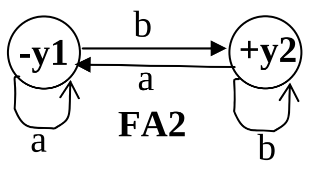
We can describe the algorithm for forming FA3 as follows:
Starting with two machines FA1, with states x1, x2, x3,…. And FA2 with states y1,y2,y3,…,build a new machine FA3 with states z1,z2,z3,… where each z is of the form "$x_{something}$ or $y_{something}$". If either the x part or the y part is a final state, then the corresponding z is a final state. To go from one z to another by reading a letter from the input string, we see what happens to the x part and to the y part and go to the new z accordingly. We could write this as a formula:

z_{new} after letter p=[x_{new} after letter p]or[y_{new} after letter p]

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.





	a	b
-x1	x2	x1
x2	x3	x1
+x3	x3	x3

The transition table for FA1

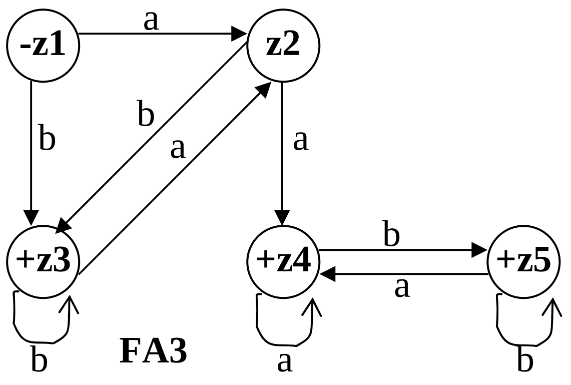
	a	b
-y1	y1	y2
+y2	y1	y2

The transition table for FA2

z1=x1 or y1
z2= x2 or y1
z3=x1 or y2
z4=x3 or y1
z5=x3 or y2

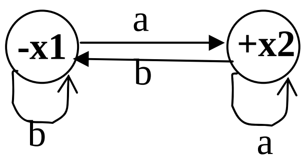
	a	b
-z1	z2	z3
z2	z4	z3
+z3	z2	z3
+z4	z4	z5
+z5	z4	z5

The transition table for FA3

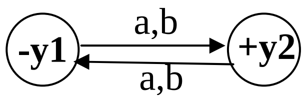


H.W.

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 Kleenes theorem.



FA1



FA2

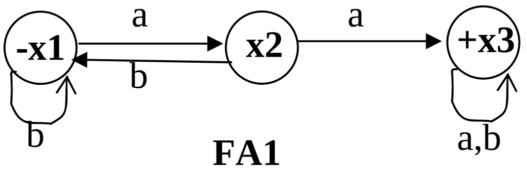
Rule3: *if there is an FA1 that accepts the language defined by the regular expression r1 and an FA2 that accepts the language defined by the regular expression r2, then there is an FA3 that accepts the language defined by the concatenation r1r2.*

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 option that we are continuing on FA1 or are beginning on FA2. From there we establish z states for all situations of the form:

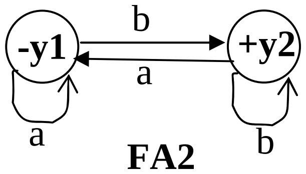
Are in $x_{\text{something}}$ continuing on FA1
Or
Have just started y1 about to continue on FA2
Or
Are in $y_{\text{something}}$ continuing on FA2

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 in b.



FA1



FA2

	a	b
-x1	x2	x1
x2	x3	x1
+x3	x3	x3

The transition table for FA1

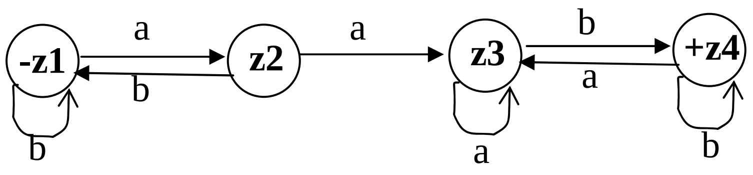
	a	b
-y1	y1	y2
+y2	y1	y2

The transition table for FA2

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

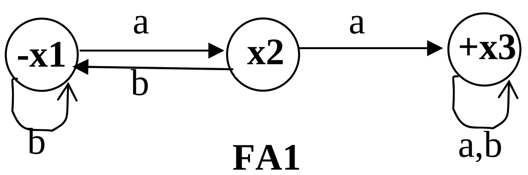
The transition table for FA3



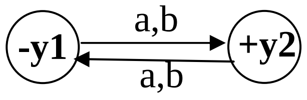
FA3

<u>H.W.</u>

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 in them using Kleen’s theorem.



FA1

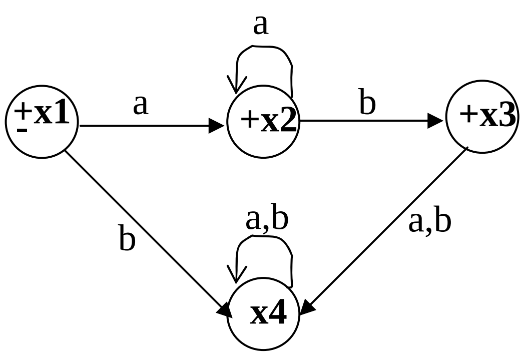


FA2

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.

<u>*Example*</u>

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*.



FA1

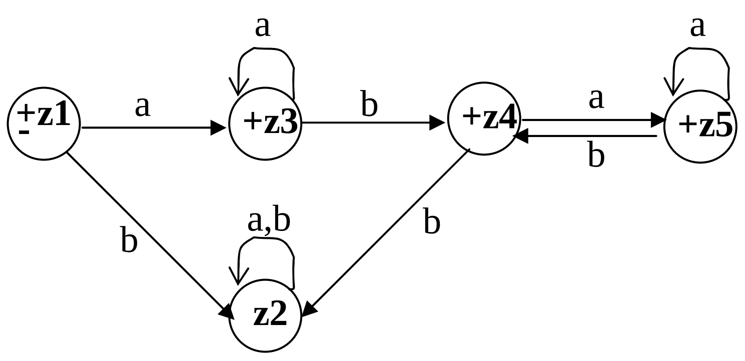
	a	b
-,+ x1	x2	x4
+x2	x2	x3
+x3	x4	x4
x4	x4	x4

The transition table for FA1

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

	a	b
-,+ z1	z3	z2
z2	z2	z2
+z3	z3	z4
+z4	z5	z2
+z5	z5	z4

The transition table for FA2

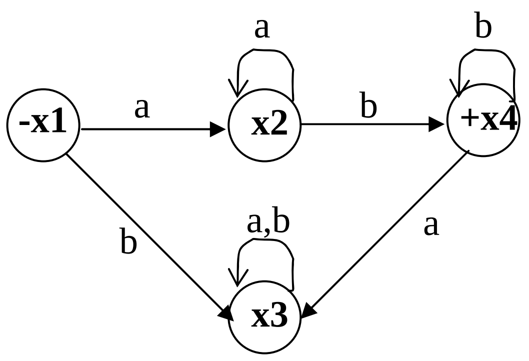


FA2

<u>H.W.</u>

Let FA1 accept the language defined by r1, find FA2 that accept the language defined by r1* using Kleene's theorem.

r1= aa*bb*



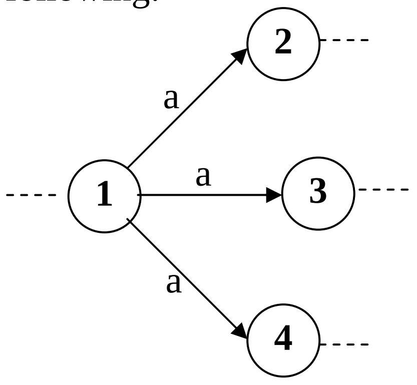
FA1

NONDETERMINISM

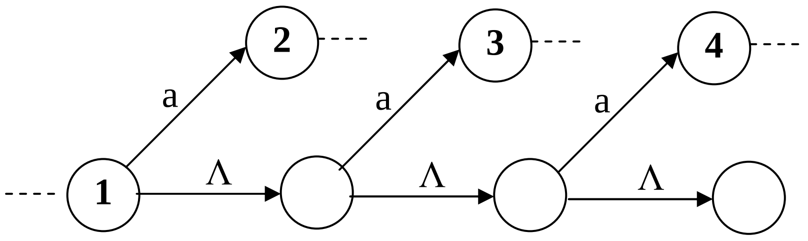
A *nondeterministic finite automaton* (NF) is a collection of three things:

1. A finite set of states with one start state (-) and some final states (+).
2. An alphabet ∑ of possible input letters.
3. A finite set of transitions that describe how to proceed from each state to other states along edges labeled with letters of the alphabet (but not the null string Λ), where we allow the possibility of more than one edge with the same label from any state and some states for which certain input letters have no edge.

We can convert any NFA into a TG with no repeated labels from any single state as in the following:



Is equivalent to



Any FA will satisfy the definition of an NFA. We have:

1. Every FA is an NFA.
2. Every NFA has an equivalent TG.
3. By Kleen's theorem, every TG has an equivalent FA.

Therefore:
Language of FA's ⊂ language of NFA's ⊂ language of TG's = language of FA's

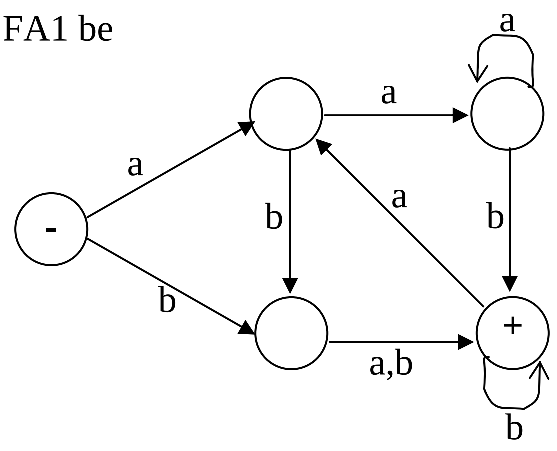
Theorem

FA = NFA

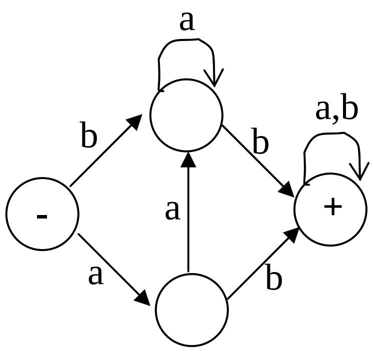
By which we mean that any language defined by a nondeterministic finite automaton is also definable by a deterministic (ordinary) finite automaton and vice versa.

Example

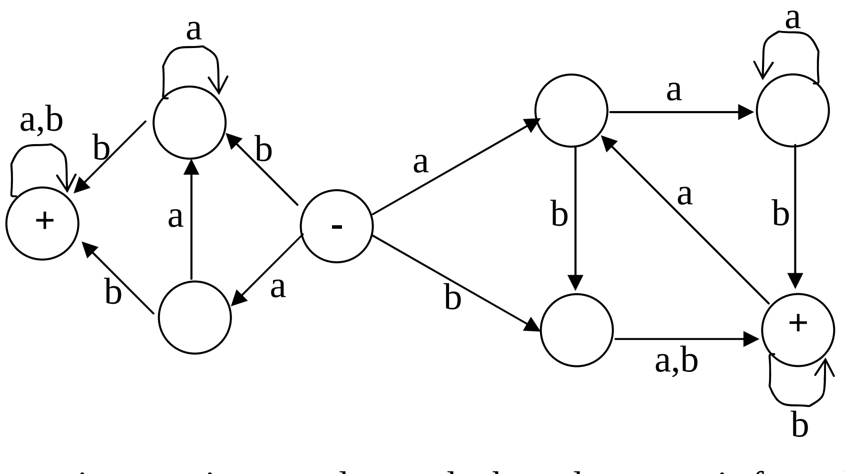
Let FA1 be



And let FA2 be



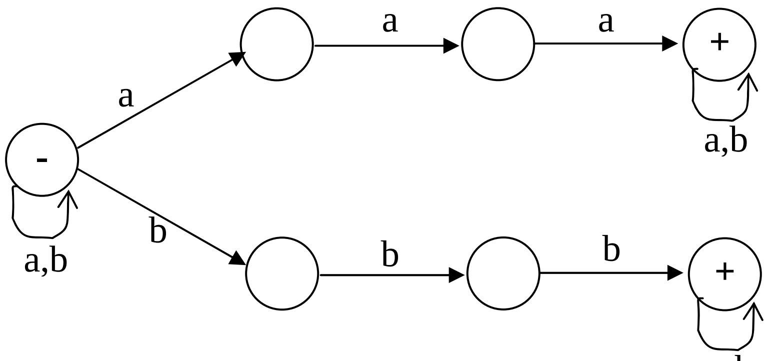
Then NFA3= FA1 + FA2 is



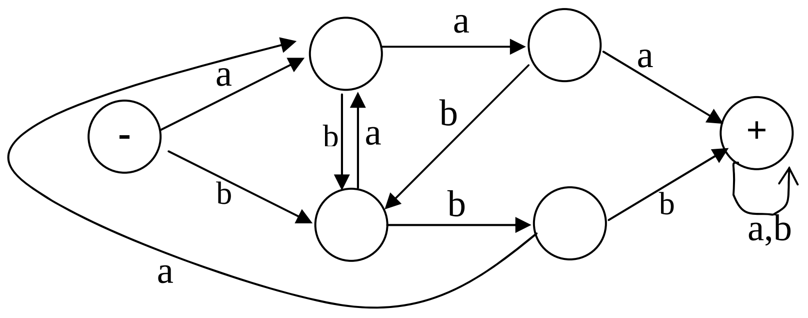
It is sometimes easier to understand what a language is from the picture of an NFA that accepts it than from the picture of an FA as in the following example.

Example

The NFA and FA Below accepts the language of all words that contains either a triple a (the substring aaa) or a triple b (the substring bbb) or both.



NFA



FA

COMPARISON TABLE FOR AUTOMATA

	FA	**TG**	**NFA**
Start states	one	One or more	one
Final states	Some or none	Some or none	Some or none
Edge labels	Letter from ∑	words from ∑*	Letter from ∑
Number of edges from each state	One for each letter in ∑	Arbitrary	Arbitrary
Deterministic (every input string has one path)	Yes	Not necessarily	Not necessarily
Every path represents one word	Yes	Yes	Yes

FINITE AUTOMATA WITH OUTPUT

We shall investigate two different models for FA's with output capabilities; these are *Moore machine* and *Mealy machine*.

A **Moore machine** is a collection of five things:

1. A finite set of states q0,q1,q2,… where q0 is designed as the start state.
2. An alphabet of letters for forming the input string ∑= { a, b, c, …}.
3. An alphabet of possible output characters Γ = { x, y, z, …}.
4. A transition table that shows for each state and each input letter what state is reached next.
5. An output table that shows what character from Γ is printed by each state that is entered.

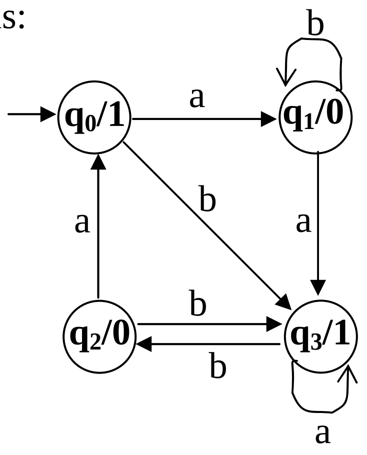
A Moore machine does not define a language of accepted words, since every input string creates an output string and there is no such thing as a final state. The processing is terminated when the last input letter is read and the last output character is printed.

Example

Input alphabet: ∑ = {a, b}
Output alphabet: Γ = {0, 1}
Names of states: q0, q1, q2, q3. (q0 = start state)

Old state	**Transition table** New state After input a	after input b	**Output table** (the character printed in the old state)
-q0	q1	q3	1
q1	q3	q1	0
q2	q0	q3	0
q3	q3	q2	1

The Moore machine is:

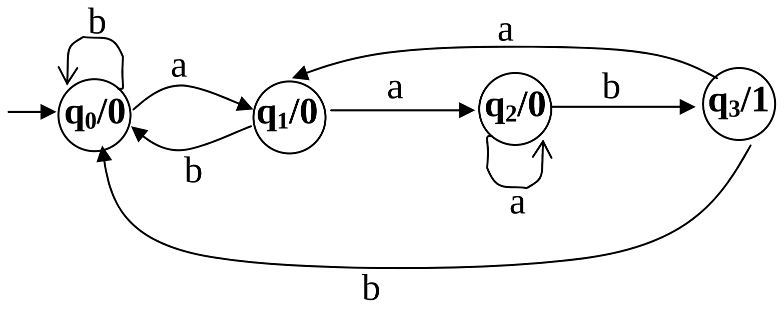


Note: the two symbols inside the circle are separated by a slash "/", on the left side is the name of the state and on the right is the output from that state.

If the input string is abab to the Moore machine then the output will be 10010.

Example

The following Moore machine will "count" how many times the substring aab occurs in a long input string.



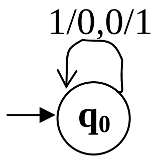
The number of substrings aab in the input string will be exactly the number of 1's in the output string.

Input string		a	a	a	b	a	b	b	a	a	b	b
State	q_0	q_1	q_2	q_2	q_3	q_1	q_0	q_0	q_1	q_2	q_3	q_0
Output	0	0	0	0	1	0	0	0	0	0	1	0

A **Mealy machine** is a collection of four things:

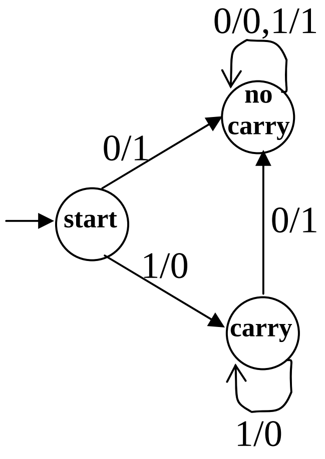
1. A finite set of states q0,q1,q2,… where q0 is designed as the start state.
2. An alphabet of letters for forming the input string ∑= { a, b, c, …}.
3. An alphabet of possible output characters Γ = { x, y, z, …}.
4. A pictorial representation with states represented by small circles and directed edges indicating transitions between states. Each edge is labeled with a compound symbol of the form i/o where i is an input letter and o is an output character. Every state must have exactly one outgoing edge for each possible input letter. The edge we travel is determined by the input letter i; while traveling on the edge we must print the output character o.

Example
The following Mealy machine prints out the 1's complement of an input bit string.



If the input is 001010 the output is 110101. This is a case where the input alphabet and output alphabet are both {0,1}.

Example
The following Mealy machine called the increment machine.



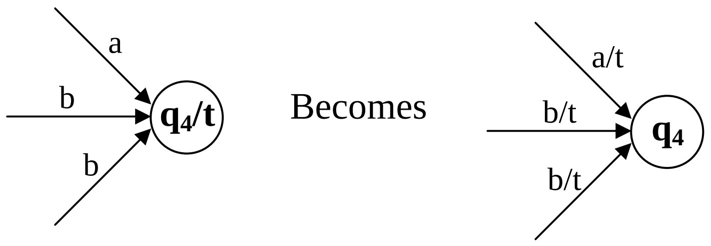
If the input is 1011 the output is 1100.

Definition
Given the Mealy machine Me and the Moore machine Mo, which prints the automatic start-state character x, we will say that these two machines are equivalent if for every input string the output string from Mo is exactly x concatenated with the output from Me.

Note: we prove that for every Moore machine there is an equivalent Mealy machine and for every Mealy machine there is an equivalent Moore machine. We can then say that the two types of machine are completely equivalent.

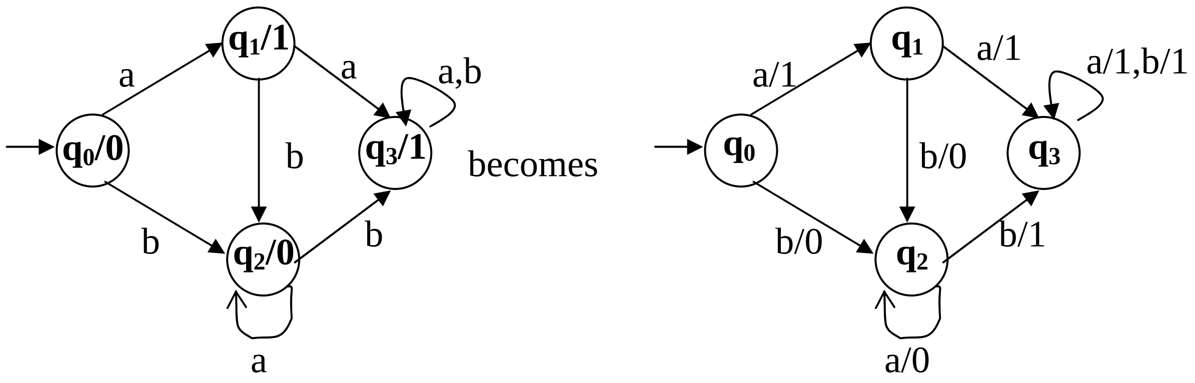
Theorem
If Mo is a Moore machine, then there is a Mealy machine Me that is equivalent to it.

Proof
The proof will be by constructive algorithm.



Example

Below, a Moore machine is converted into a Mealy machine:

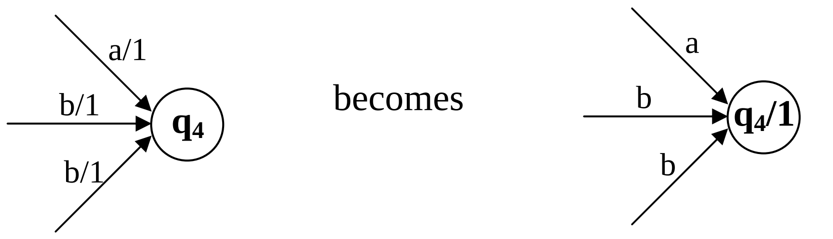


Theorem

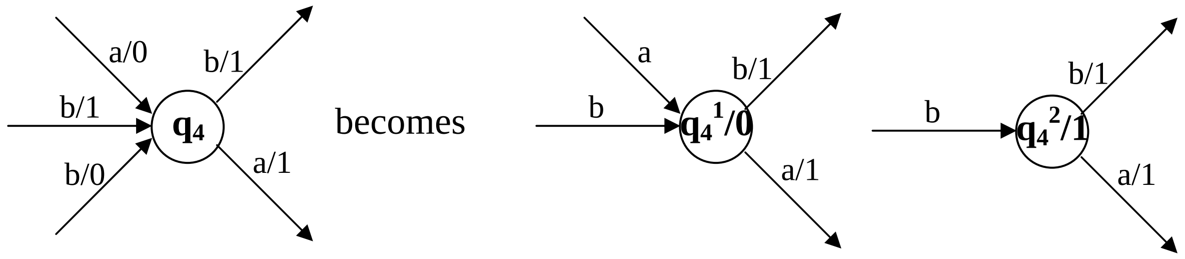
For every Mealy machine Me there is a Moore machine Mo that is equivalent to it.

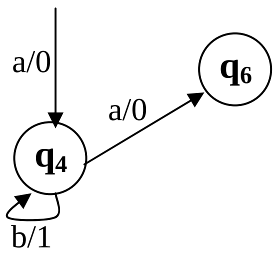
Proof

The proof will be by constructive algorithm.

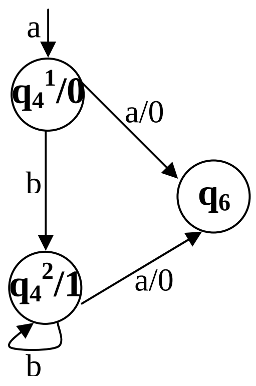


If there is more than one possibility for printing as we enter the state, then we need a copy of the state for each character we might have to print. (we may need as many copies as there are character in Γ).



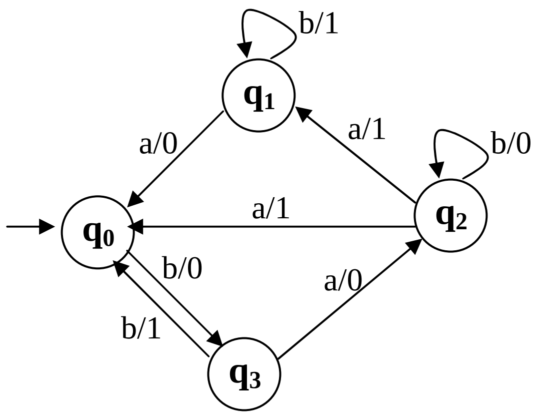


Becomes

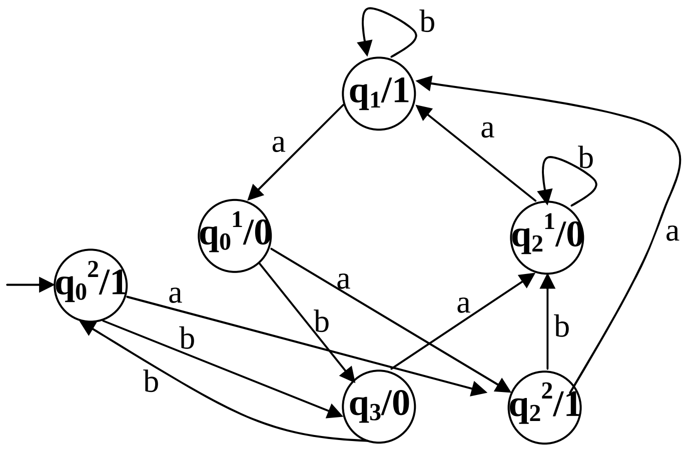


Example

Convert the following Mealy machine to Moore machine:



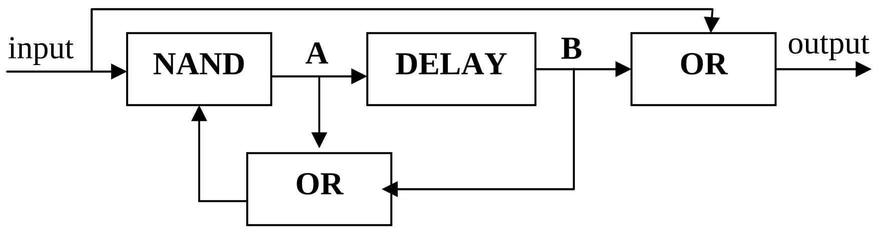
Mealy machine



Moore machine

Example

Draw the Mealy machine for the following sequential circuit:

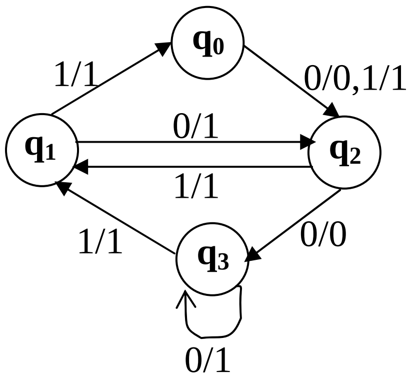


First we identify four states:

q0 is A= 0 B= 0
q1 is A= 0 B= 1
q2 is A= 1 B= 0
q3 is A= 1 B= 1

the operation of this circuit is such that after an input of 0 or 1 the state changes according to the following rules:
new B= old A
new A = (input) NAND (old A OR old B)
output = (input) OR (old B)
Suppose we are in q0 and we receive the input 0.
new B = old A = 0
new A = 0 NAND (0 OR 0)
= 0 NAND 0
= 1
output = 0 OR 0 = 0
the new state is q2 (since new A=1, new B=0)
if we are in state q0 and we receive the input 1:
new B= old A = 0
new A = 1 NAND (0 OR 0) =1
output = 1 OR 0 =1
the new state is q2.
We repeat this process for every state and for each input to produce the following table:

Old state	After input 0		After input 1	
	New state	Output	New state	Output
q0	q2	0	q2	1
q1	q2	1	q0	1
q2	q3	0	q1	1
q3	q3	1	q1	1



Mealy machine

Comparison table for automata

	FA	**TG**	**NFA**	**NFA- Λ**	**Moore**	**Mealy**
Start states	one	One or more	one	one	one	one
Final states	Some or none	Some or none	Some or none	Some or none	none	none
Edge labels	Letters from ∑	words from ∑*	Letters from ∑	Letters from ∑ or Λ	Letter from ∑	i/o i from ∑ o from Γ
Number of edges from each state	One for each letter in ∑	arbitrary	arbitrary	arbitrary	One for each letter in ∑	One for each letter in ∑
deterministic	yes	no	no	no	yes	yes
output	no	no	no	no	yes	yes

CONTEXT FREE GRAMMAR

A *context free grammar*, called **CFG**, is a collection of <u>three</u> things:

1. An alphabet ∑ of letters called **terminals** from which we are going to make strings that will be the words of a language.
2. A set of symbols called **nonterminals**, one of which is the symbol S, standing for "start here".
3. A finite set of production of the form:

One nonterminal → finite string of terminals and/ or nonterminals

Where the strings of terminals and nonterminals can consist of only terminals or of any nonterminals, or any mixture of terminals and nonterminals or even the empty string. We require that at least one production has the nonterminal S as its left side.

<u>Definition</u>

The language generated by the CFG is the set of all strings of terminals that can be produced from the start symbol S using the production as substitutions. A language generated by the CFG is called a **context free language** (**CFL**).

<u>Example</u>

Let the only terminal be a.

Let the only nonterminal be S.

Let the production be:

$$S \rightarrow aS$$
$$S \rightarrow \Lambda$$

The language generated by this CFG is exactly a*.

In this language we can have the following derivation:

$S \rightarrow aS \rightarrow aaS \rightarrow aaaS \rightarrow aaaaS \rightarrow aaaaaS \rightarrow aaaaa\Lambda = aaaaa$

Example

Let the only terminal be a.

Let the only nonterminal be S.

Let the production be:

$$S \rightarrow SS$$
$$S \rightarrow a$$
$$S \rightarrow \Lambda$$

The language generated by this CFG is also just the language a*.

In this language we can have the following derivation:

$S \rightarrow SS \rightarrow SSS \rightarrow SaS \rightarrow SaSS \rightarrow \Lambda aSS \rightarrow \Lambda aaS \rightarrow \Lambda aa\Lambda = aa$

Example

Let the terminals be a, b. And the only nonterminal be S.

Let the production be:

$$S \rightarrow aS$$
$$S \rightarrow bS$$
$$S \rightarrow a$$
$$S \rightarrow b$$

The language generated by this CFG is $(a+b)^+$.

In this language we can have the following derivation:

$S \rightarrow bS \rightarrow baS \rightarrow baaS \rightarrow baab$

Example

Let the terminals be a, b. And the only nonterminal be S.

Let the production be:

$$S \rightarrow aS$$
$$S \rightarrow bS$$
$$S \rightarrow \Lambda$$

The language generated by this CFG is $(a+b)^*$.

In this language we can have the following derivation:

$S \rightarrow bS \rightarrow baS \rightarrow baaS \rightarrow baa\Lambda = baa$

Example

Let the terminals be a, b,Λ. And the nonterminals be S,X,Y.

Let the production be:

$$\begin{aligned} S &\rightarrow X \\ S &\rightarrow Y \\ X &\rightarrow \Lambda \\ Y &\rightarrow aY \\ Y &\rightarrow bY \\ Y &\rightarrow a \\ Y &\rightarrow b \end{aligned}$$

The language generated by this CFG is $(a+b)^*$.

Example

Let the terminals be a, b,Λ. And the nonterminals be S,X,Y.

Let the production be:

$$\begin{aligned} S &\rightarrow XY \\ X &\rightarrow \Lambda \\ Y &\rightarrow aY \\ Y &\rightarrow bY \\ Y &\rightarrow a \\ Y &\rightarrow b \end{aligned}$$

The language generated by this CFG is $(a+b)^+$.

Example

Let the terminals be a, b.

Let the nonterminals be S,X.

Let the production be:

$$\begin{aligned} S &\rightarrow XaaX \\ X &\rightarrow aX \\ X &\rightarrow bX \\ X &\rightarrow \Lambda \end{aligned}$$

The language generated by this CFG is (a+b)* aa(a+b)*.

To generate baabaab we can proceed as follows:

$S \rightarrow XaaX \rightarrow bXaaX \rightarrow baXaaX \rightarrow baaXaaX \rightarrow baabXaaX \rightarrow baab\Lambda aaX = baabaaX$

$\rightarrow baabaabX \rightarrow baabaab\Lambda = baabaab$

Example

Let the terminals be a, b.

Let the nonterminals be S,X,Y.

Let the production be:

S → XY
X → aX
X → bX
X → a
Y → Ya
Y → Yb
Y → a

The language generated by this CFG is (a+b)* aa(a+b)*.

To drive babaabb:

S→XY→bXY→baXY→babXY→babaY→babaYb→babaYbb→babaabb

Example

Let the terminals be a, b.

Let the nonterminals be S,BALANCED,UNBALANCED.

Let the production be:

S → SS

S → BALANCED S

S → S BALANCED

S → Λ

S → UNBALANCED S UNBALANCED

BALANCED→ aa

BALANCED→ bb

UNBALANCED → ab

UNBALANCED → ba

The language generated by this CFG is even-even.

<u>Example</u>

Let the terminals be a, b.

Let the nonterminals be S,A,B.

Let the production be:

$$\begin{aligned} S &\rightarrow aB \\ S &\rightarrow bA \\ A &\rightarrow a \\ A &\rightarrow aS \\ A &\rightarrow bAA \\ B &\rightarrow b \\ B &\rightarrow bS \\ B &\rightarrow aBB \end{aligned}$$

The language generated by this CFG is the language EQUAL of all strings that have an equal number of a's and b's.

<u>H.W</u>

Find the RE for the following CFG:

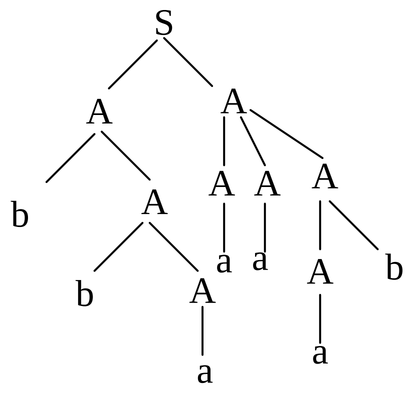
$$\begin{aligned} S &\rightarrow XS \mid \Lambda \\ X &\rightarrow ZY \\ Z &\rightarrow abZ \mid baZ \mid ab \mid ba \\ Y &\rightarrow aa \mid bb \end{aligned}$$

Trees

Example

S → AA
A → AAA | bA | Ab | a

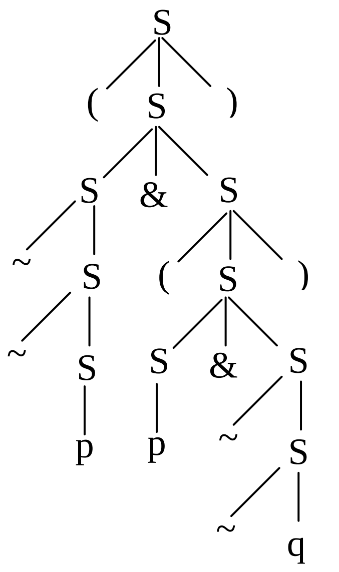
If we want to produce the word bbaaaab, the tree will be:



This diagram is called **syntax tree** or **parse tree** or **generation tree** or **production tree** or **derivation tree**.

Example

S → (S) | S&S | ~S | p| q
The derivation tree for the word (~ ~ p & (p & ~ ~ q)) will be:

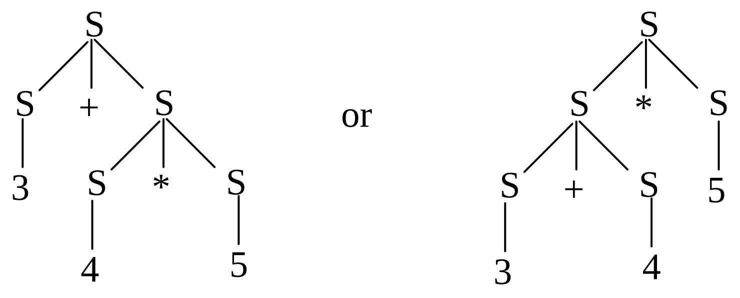


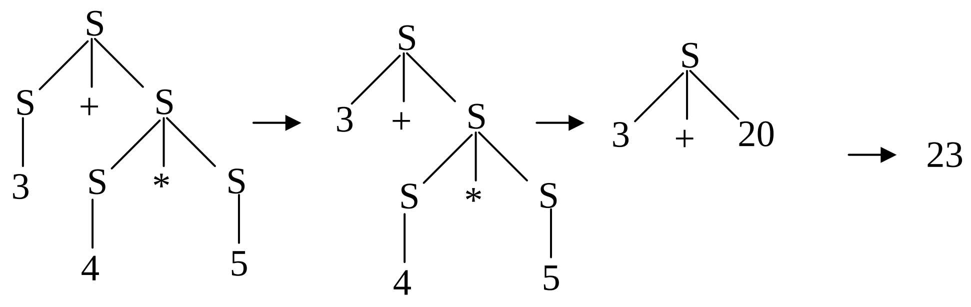
Example

S → S + S | S* S | number

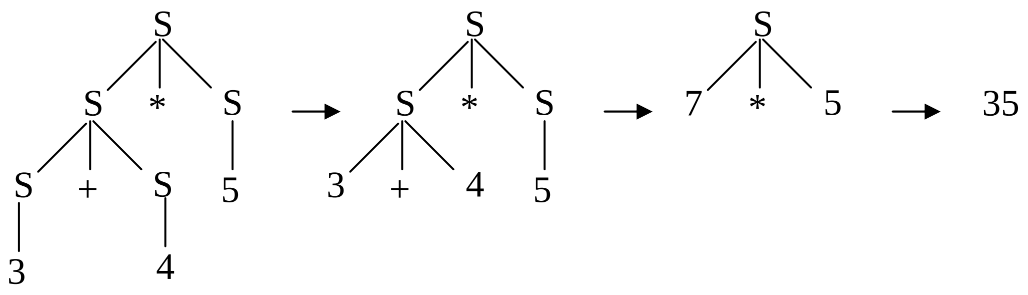
Does the expression 3+4*5 mean (3+4)*5 which is 35 or does it mean 3+(4*5) which is 23?

We can distinguish between these two possible meaning for the expression 3+4*5 by looking at the two possible derivation trees that might have produced it.





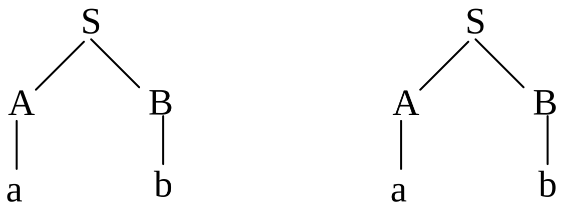
Or



Example

S → AB
A → a
B → b

S → AB → aB → ab or S → AB → Ab → ab



There is no ambiguity of interpretation.

Definition

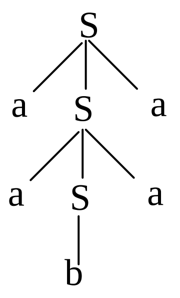
A CFG is called **ambiguous** if for at least one word in the language that it generates there are two possible derivations of the word that correspond to different syntax trees.

Example

The CFG for palindrome

S → aSa | bSb |a |b | Λ

S → aSa → aaSaa → aabaa

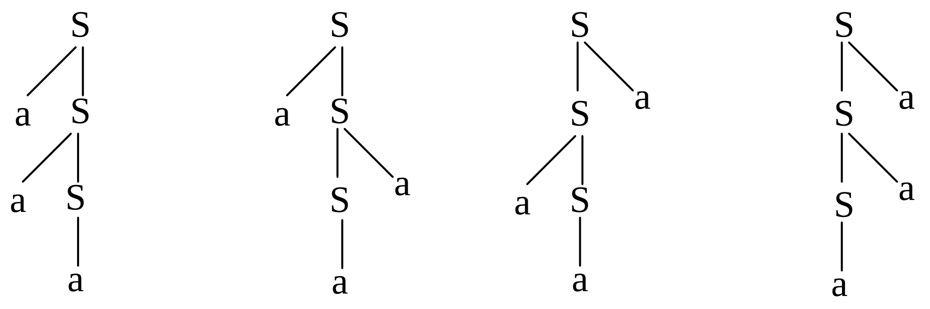


The CFG is unambiguous.

Example

S → aS |Sa | a

In this case the word a^3 can be generated by four different trees:

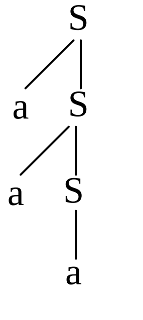


The CFG is therefore ambiguous.

The same language can be defined by the CFG:

S → aS | a

For which the word a^3 has only one production tree:

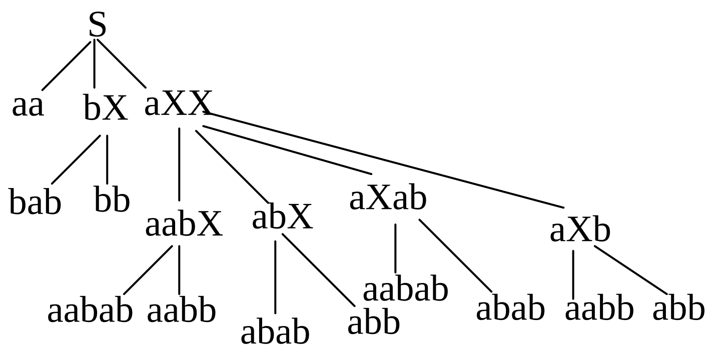


This CFG is not ambiguous.

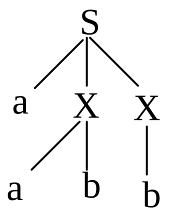
<u>Definition</u>
For a given CFG we define a tree with the start symbol S as its root and whose nodes are working strings of terminals and nonterminals. The descendants of each node are all possible results of applying every production to the working string, one at a time. A string of all terminals is a terminal node in the tree. The resultant tree is called the **total language tree** of the CFG.

<u>Example</u>

For the CFG
S → aa | bX | aXX
X → ab | b
The total language tree is:

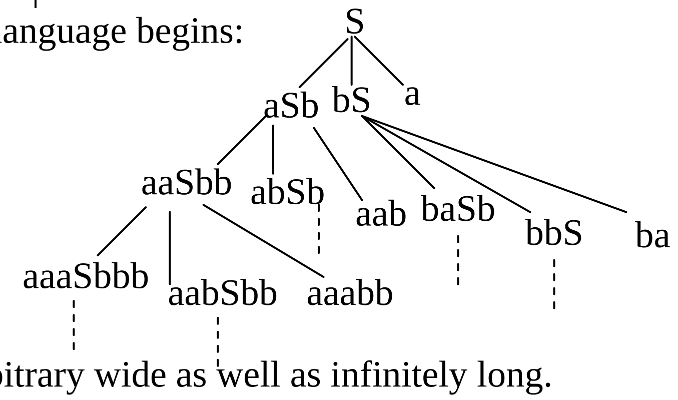


The total language has only seven different words. Four of it's words (abb, aabb, abab, aabab) have two different possible derivation because they appear as terminal nodes in this tree in two different places. However the words are not generated by two different derivation trees and the grammar is unambiguous. For example:



<u>Example</u>

Consider the CFG:
S → aSb | bS | a
The total tree of this language begins:



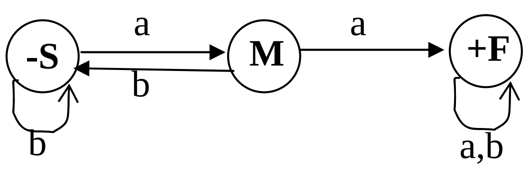
The trees may get arbitrary wide as well as infinitely long.

REGULAR GRAMMARS

Note : all regular languages can be generated by CFG's, and so can some non-regular languages but not all possible languages.

Example

Consider the FA below, which accepts the language of all words with a double a:



All the necessary to convert it to CFG is that:

1. every edge between states be a production:

$$\begin{tikzcd} X \arrow[r, "c"] & Y \end{tikzcd}$$ becomes X→ cY

and

2. every production correspond to an edge between states:

X→ cY comes from $$\begin{tikzcd} X \arrow[r, "c"] & Y \end{tikzcd}$$

or to the possible termination at a final state:

X→ Λ

only when X is a final state.

So the production rules of our example will be:

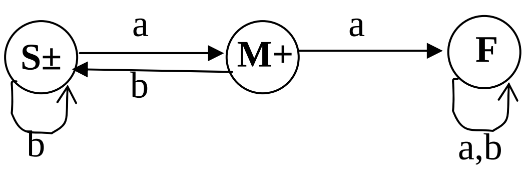
S → aM | bS
M → aF | bS
F →aF | bF | Λ

Definition

For a given CFG a **semiword** is a string of terminals (maybe none) concatenated with exactly one nonterminal (on the right), for example:

(terminal) (terminal) . . . (terminal) (Nonterminal)

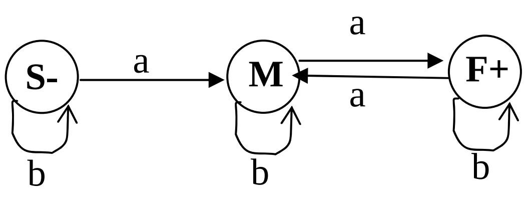
Example
Consider the following FA with two final states:



So the production rules of our example will be:
S → aM | bS | Λ
M → aF | bS | Λ
F →aF | bF

Theorem
All regular languages can be generated by CFG's. This can be stated as: All regular languages are CFL's.

Example
The language of all words with an even number of a's (with at least some a's) is regular since it can be accepted by this FA:



We have the following set of productions:
S → aM | bS
M → aF | bM
F →aM | bF | Λ

Theorem
If all the productions in a given CFG fit one of the two forms:

Nonterminal→ semiword

Or

Nonterminal → word

(where the word may be Λ) then the language generated by this CFG is regular.

Proof
We shall prove that the language generated by such a CFG is regular by showing that there is a TG that accepts the same language. We shall build this TG by constructive algorithm.

Let us consider a general CFG in the form:

$N_1 \rightarrow w_1N_2$	$N_{40} \rightarrow w_{10}$
$N_1 \rightarrow w_2N_3$	$N_{41} \rightarrow w_{23}$
$N_2 \rightarrow w_3N_4$	. . .
. . .	

Where N's are the nonterminals and w's are strings of terminals, and the parts w_yN_z are the semiwords used in productions. One of these N's must be S. Let N_1 = S.
Draw a small circle for each N and one extra circle labeled +. The circle for S we label - .

$$\begin{tikzcd} \boxed{S-} & \boxed{N_2} & \boxed{N_3} \dots & \boxed{N_x} & \dots \boxed{+} \end{tikzcd}$$

Draw a directed edge from state N_x to N_z and label it with the word w_y.

$$\begin{tikzcd} N_x \arrow[r, "w_v"] & N_z \end{tikzcd}$$

If the two nonterminals above are the same the path is a loop.
For every production rule of the form:

$$N_p \rightarrow w_q$$

Draw a directed edge from N_p to + and label it with the word w_q.

$$\begin{tikzcd} N_p \arrow[r, "w_q"] & + \end{tikzcd}$$

We have now constructed a transition graph.

Definition
A CFG is called a **regular grammar** if each of its productions is of one of the two forms:
Nonterminal → semiword
Nonterminal → word

Example
Consider the CFG:

$$S \rightarrow aaS \mid bbS \mid \Lambda$$

It is a regular grammar and the whole TG is:

$$\begin{tikzcd} - \arrow[r, "\Lambda"] \arrow[loop below, "{aa,bb}"] & + \end{tikzcd}$$

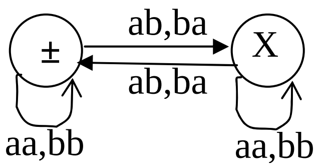
It is corresponds to the regular expression: (aa+bb)*

Example

Consider the CFG:

$$S \rightarrow aaS \mid bbS \mid abX \mid baX \mid \Lambda$$
$$X \rightarrow aaX \mid bbX \mid abS \mid baS$$

The TG for even-even is:

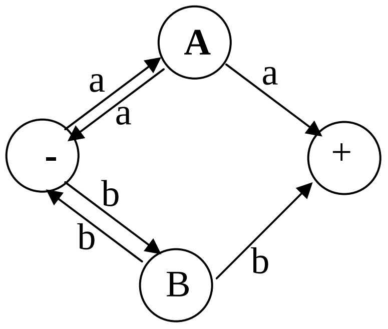


Example

Consider the CFG:

$$S \rightarrow aA \mid bB$$
$$A \rightarrow aS \mid a$$
$$B \rightarrow bS \mid b$$

The corresponding TG is:



This language can be defined by the regular expression: $(aa+bb)^+$

H.W

Find CFG that generate the regular language over the alphabet ∑={a,b} of all strings without the substring aaa.

CHOMSKY NORMAL FORM (CNF)

Theorem

If L is a context-free language generated by CFG that includes Λ-productions, then there is a different CFG that has no Λ-production that generates either the whole language L(if L does not include the word Λ) or else generates the language of all the words in L that are not Λ.

Definition

In a given CFG, we call a nonterminal N **nullable** if:

- There is a production: N→ Λ

Or

- There is a derivation that start at N and leads to Λ: N→…→ Λ

Example

Consider the CFG:

S → a| Xb| aYa

X → Y | Λ

Y → b | X

X and Y are nullable.

The new CFG is:

S → a| Xb| aYa| b| aa

X → Y

Y → b | X

Example

Consider the CFG:

S → Xa

X → aX| bX | Λ

X is the only nullable nonterminal.

The new CFG is:

$S \to Xa \mid a$

$X \to aX \mid bX \mid a \mid b$

Example

Consider the CFG:

$S \to XY$

$X \to Zb$

$Y \to bW$

$Z \to AB$

$W \to Z$

$A \to aA \mid bA \mid \Lambda$

$B \to Ba \mid Bb \mid \Lambda$

A, B, W and Z are nullable.

The new CFG is:

$S \to XY$

$X \to Zb \mid b$

$Y \to bW \mid b$

$Z \to AB \mid A \mid B$

$W \to Z$

$A \to aA \mid bA \mid a \mid b$

$B \to Ba \mid Bb \mid a \mid b$

Definition

A production of the form: One Nonterminal → One Nonterminal

is called a **unit** production.

Theorem

If there is a CFG for the language L that has no Λ-production, then there is also a CFG for L with no Λ-production and no unit production.

Example

Consider the CFG:

$$S \rightarrow A |\ bb$$
$$A \rightarrow B |\ b$$
$$B \rightarrow S |\ a$$

$S \rightarrow A$	gives	$S \rightarrow b$
$S \rightarrow A \rightarrow B$	gives	$S \rightarrow a$
$A \rightarrow B$	gives	$A \rightarrow a$
$A \rightarrow B \rightarrow S$	gives	$A \rightarrow bb$
$B \rightarrow S$	gives	$B \rightarrow bb$
$B \rightarrow S \rightarrow A$	gives	$B \rightarrow b$

The new CFG for this language is:

$$S \rightarrow bb |\ b |\ a$$
$$A \rightarrow b |\ a |\ bb$$
$$B \rightarrow a |\ bb |\ b$$

Theorem

If L is a language generated by some CFG then there is another CFG that generates all the non- Λ words of L, all of these productions are of one of two basic forms:

Nonterminal → string of only Nonterminals

Or

Nonterminal → One Terminal

Example

Consider the CFG:

$S \rightarrow X_1 | X_2aX_2 | aSb | b$

$X_1 \rightarrow X_2X_2 | b$

$X_2 \rightarrow aX_2 | aaX_1$

Becomes:

$S \rightarrow X_1$

$S \rightarrow X_2AX_2$

$S \rightarrow ASB$

$S \rightarrow B$

$X_1 \rightarrow X_2X_2$

$X_1 \rightarrow B$

$X_2 \rightarrow AX_2$

$X_2 \rightarrow AAX_1$

$A \rightarrow a$

$B \rightarrow b$

Example

Consider the CFG:

$S \rightarrow Na$

$N \rightarrow a | b$

Becomes:

$S \rightarrow NA$
$N \rightarrow a | b$
$A \rightarrow a$

Theorem

For any CFL the non- Λ words of L can be generated by a grammar in which all productions are of one of two forms:

Nonterminal → string of exactly two Nonterminals

Or

Nonterminal → One Terminal

Definition

If a CFG has only productions of the form:

Nonterminal → string of two Nonterminals

Or of the form:

Nonterminal → One Terminal

It is said to be in **Chomsky Normal Form (CNF)**.

Example

Convert the following CFG into CNF: $S \to aSa \mid bSb \mid a \mid b \mid aa \mid bb$

$S \to ASA$

$S \to BSB$

$S \to AA$

$S \to BB$

$S \to a$

$S \to b$

$A \to a$

$B \to b$

The CNF:

$S \to AR_1$

$R_1 \to SA$

$S \to BR_2$

$R_2 \to SB$

$S \to AA$

$S \to BB$

$S \to a$

$S \to b$

$A \to a$

$B \to b$

Example

Convert the following CFG into CNF:

$S \to bA \mid aB$

$A \to bAA \mid aS \mid a$

$B \to aBB \mid bS \mid b$

The CNF:

$S \to YA \mid XB$

$A \to YR_1 \mid XS \mid a$

$B \to XR_2 \mid YS \mid b$

$X \to a$

$Y \to b$

$R_1 \to AA$

$R_2 \to BB$

Example

Convert the following CFG into CNF:

$S \to AAAAS$

$S \to AAAA$

$A \to a$

The CNF:

$S \to AR_1$

$R_1 \to AR2$

$R_2 \to AR_3$

$R_3 \to AS$

$S \to AR_4$

$R_4 \to AR_5$

$R_5 \to AA$

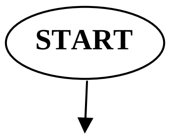
$A \to a$

PUSHDOWN AUTOMATA (PDA)

Definition

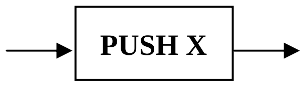
A **PDA** is a collection of eight things:

1. An alphabet ∑ of input letters.
2. An input TAPE (infinite in one direction). Initially the string of input letters is placed on the TAPE starting in cell i. The rest of the TAPE is blanks.
3. An alphabet Γof STACK characters.
4. A pushdown STACK (infinite in one direction). Initially the STACK is empty (contains all blanks)
5. One START state that has only out_adges, no in-edges.

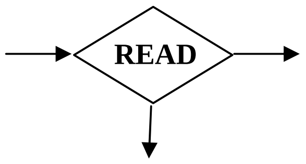


6. HALT states of two kinds: some ACCEPT and some REJECT they have in-edges and no out-edges.

7. Finitely many non branching PUSH states that introduce characters onto the top of the STACK. they are of the form:

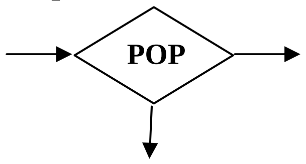


   Where X is any letter in Γ.
8. Finitely many branching states of two kinds:
   i. States that read the next unused letter from the TAPE.



   Which may have out-edges labeled with letters from ∑ and the blank character Δ, with no restrictions on duplication of labels and no insistence that there be a label for each letter of ∑, or Δ.

   ii. States that read the top character of STACK.



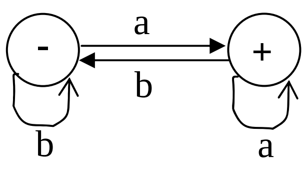
   Which may have out-edges labeled with letters from Γand the blank character Δ, again with no restrictions.

Note: we require that the states be connected so as to become a connected directed graph.

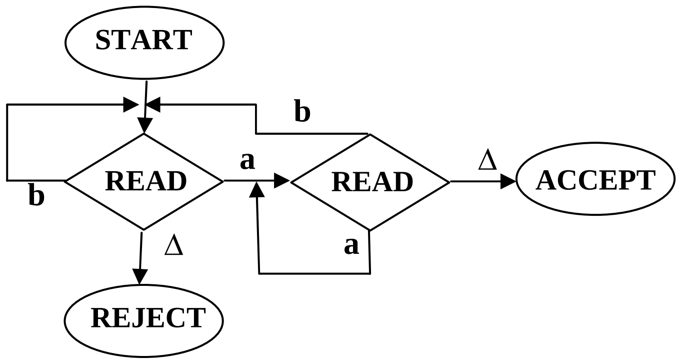
Theorem
For every regular language L there is some PDA that accepts it.

Poof
Since L is regular, so it is accepted by some FA, then we can convert FA to PDA (as in the following example).

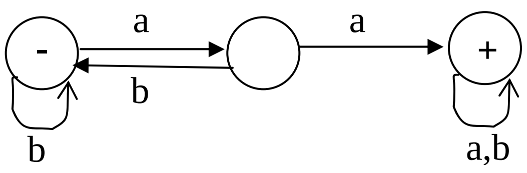
Example



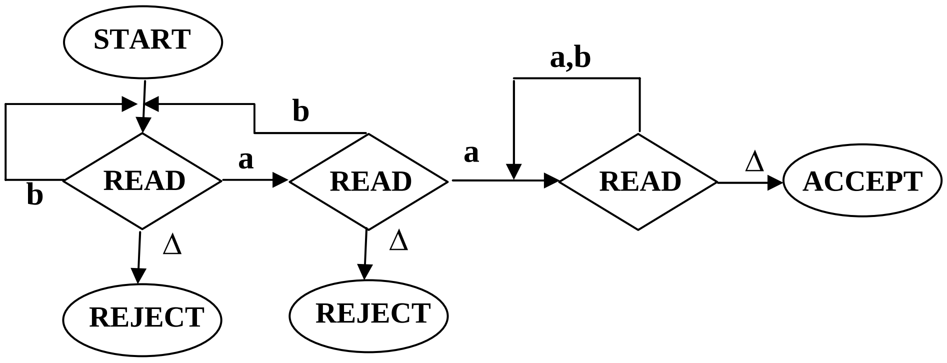
Becomes:



Example



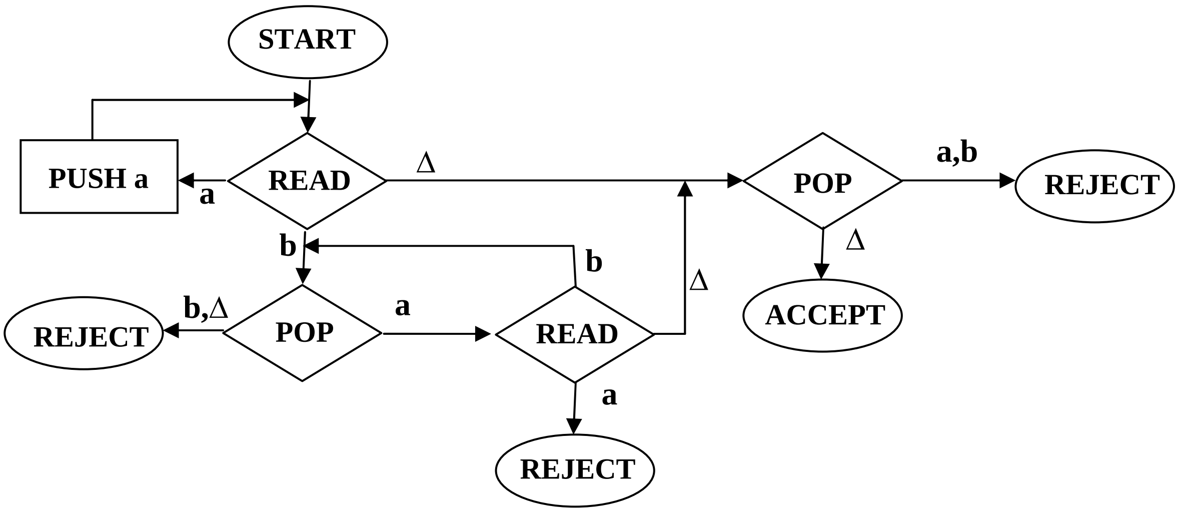
Becomes:



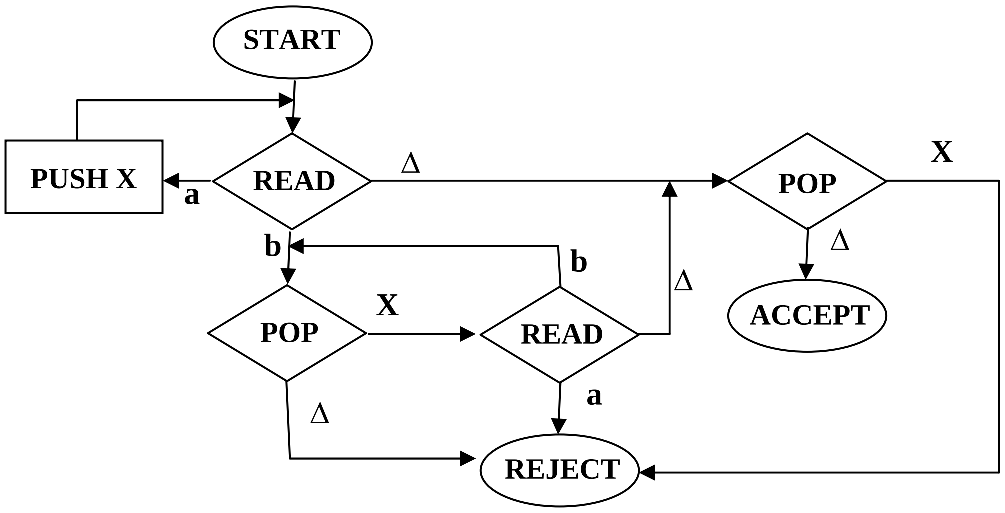
Note: we can find PDA accepts some non regular languages(as in the following example).

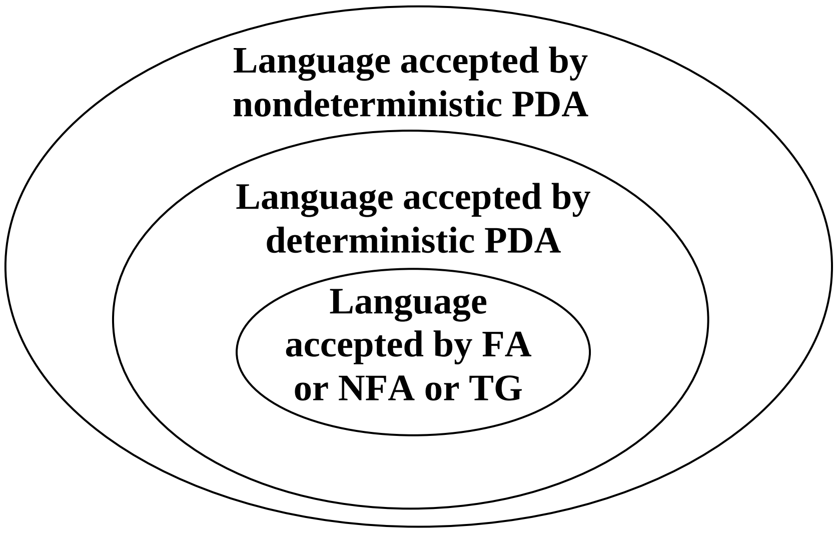
Example

The language accepted by this PDA is exactly: $\{a^n b^n, n=0,1,2,\ldots\}$

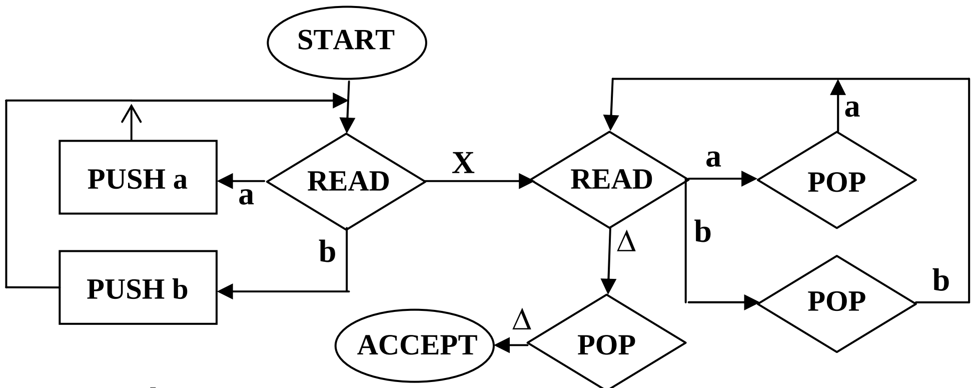


Or

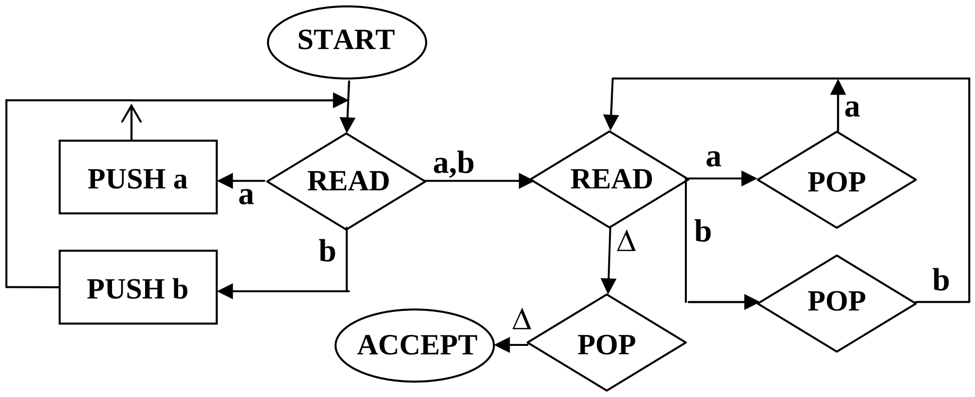



Language accepted by nondeterministic PDA
Language accepted by deterministic PDA
Language accepted by FA or NFA or TG

Example
Consider the palindrome X, language of all words of the form:
sXreverese(s), where s is any string in (a+b)*, such as {X aXa bXb aaXaa abXba aabXbaa …}

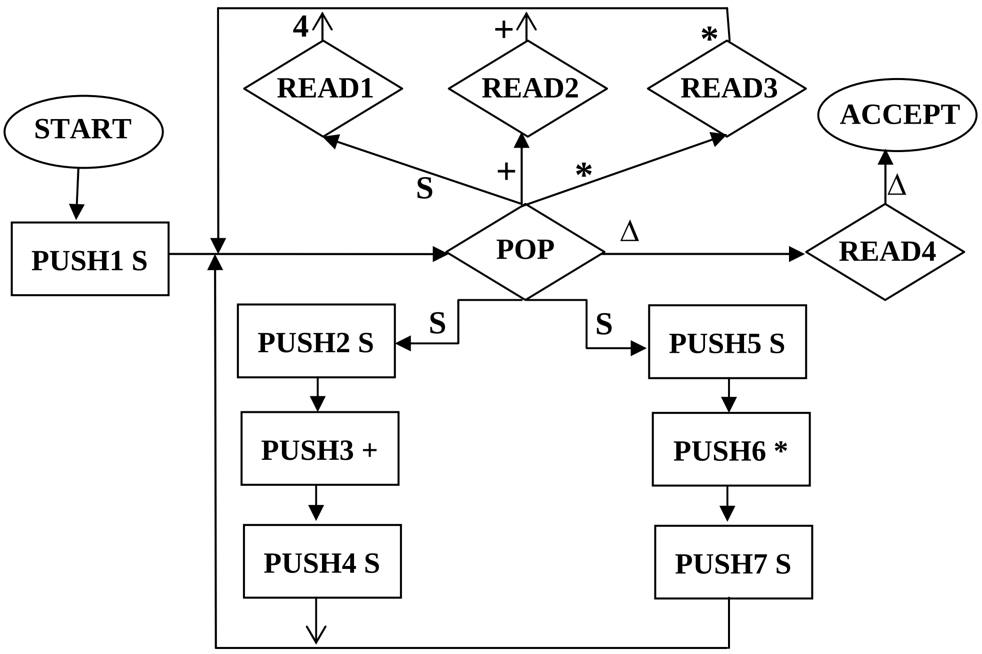


Example
odd palindrome ={a b aaa aba bab bbb …}



Nondeterministic PDA

Example
Consider the language generated by CFG: **S→S+S|S*S|4**



Now we trace the acceptance of the string: **4+4*4**

State	**Stack**	**Tape**
start	Δ	4+4*4
push1 S	S	4+4*4
pop	Δ	4+4*4
push2 S	S	4+4*4
push3 +	+ S	4+4*4
push4 S	S+ S	4+4*4
pop	+ S	4+4*4
read1	+ S	+4*4
pop	S	+4*4
read2	S	4*4
pop	Δ	4*4
push5 S	S	4*4
push6 *	* S	4*4
push7 S	S * S	4*4
pop	*S	4*4
read1	*S	*4
pop	S	*4
read3	S	4
pop	Δ	4
read1	Δ	Δ
pop	Δ	Δ
read4	Δ	Δ
accept	Δ	Δ

H.W

Find a PDA that accepts the language: $\{a^m b^n a^n$, m=1,2,3,…,n=1,2,3,…$\}$

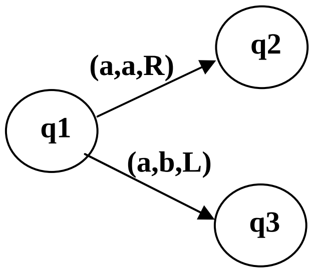
TURING MACHINE

Definition

A Turing machine (**TM**) is a collection of six things:

1. An alphabet ∑ of input letters.
2. A TAPE divided into a sequence of numbered cells each containing one character or a blank.
3. A TAPE HEAD that can in one step read the contains of a cell on the TAPE, replace it with some other character, and reposition itself to the next cell to the right or to the left of the one it has just read.
4. An alphabet Γof character that can be printed on the TAPE by the TAPE HEAD.
5. A finite set of states including exactly one START state from which we begin execution, and some (may be none) HALT states that cause execution to terminate when we enter them. The other states have no functions, only names: q1, q2, … or 1, 2, 3, …
6. A program, which is a set of rules that tell us on the basis of the letter the TAPE HEAD has just read, how to change states, what to print and where to move the TAPE HEAD. We depict the program as a collection of directed edge connecting the states. Each edge is labeled with a triplet of information: (letter, letter, direction). The first letter (either Δ or from ∑ or Γ) is the character that the TAPE HEAD reads from the cell to which it is pointing, the second letter (also Δ or from Γ) is what the TAPE HEAD prints in the cell before it leaves, the third component, the direction, tells the TAPE HEAD whether to move one cell to the right(R) or to the left (L).

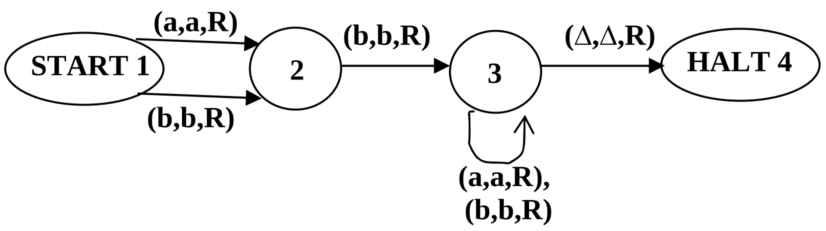
Note: TM is deterministic. This means that there is no state q that has two or more edges leaving it labeled with the same first letter. For example, the following TM is <u>not allowed</u>:



<u>Example</u>

Find TM that can accepts the language defined by the regular expression:

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



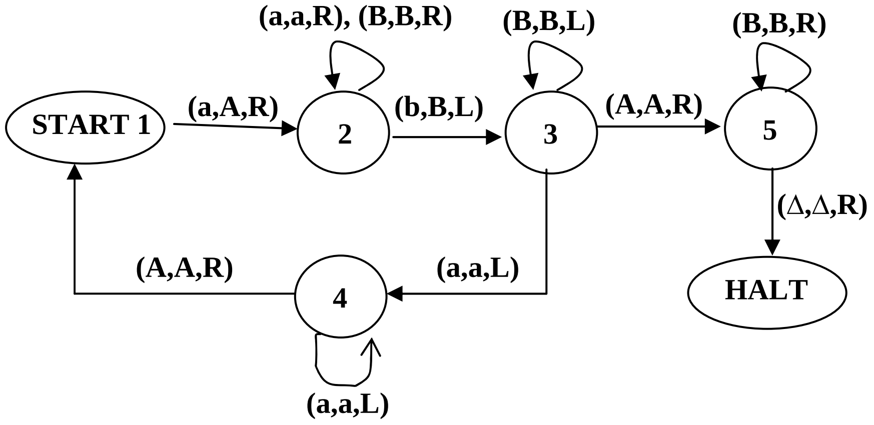
Now we trace the acceptance of the string: **aba**

$1 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow 4$

$\underline{a}ba$ $a\underline{b}a$ $ab\underline{a}\Delta$ $aba\underline{\Delta}$ $aba\Delta\underline{\Delta}$

<u>Example</u>

Find TM that can accepts the language $\{a^n b^n\}$



Now we trace the acceptance of the string: **aaabbb**

$\mathrm{\underline{a}aabbb}$ $\mathrm{A\underline{a}abbb}$ $\mathrm{Aa\underline{a}bbb}$ $\mathrm{Aaa\underline{b}bb}$ $\mathrm{Aa\underline{a}Bbb}$ $\mathrm{A\underline{a}aBbb}$ $\mathrm{\underline{A}aaBbb}$ $\mathrm{A\underline{a}aBbb}$

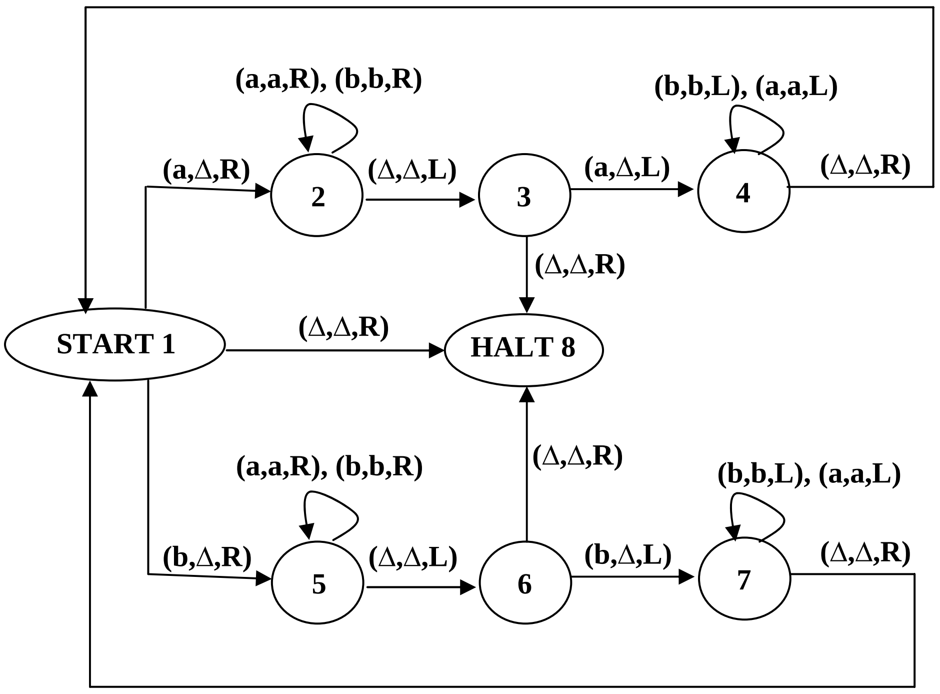
$\mathrm{Aa\underline{a}Bbb}$ $\mathrm{Aaa\underline{B}bb}$ $\mathrm{AaaB\underline{b}b}$ $\mathrm{AAa\underline{B}Bb}$ $\mathrm{AA\underline{a}BBb}$ $\mathrm{A\underline{A}aBBb}$ $\mathrm{AA\underline{a}BBb}$

$\mathrm{AAA\underline{B}Bb}$ $\mathrm{AAAB\underline{B}b}$ $\mathrm{AAABB\underline{b}}$ $\mathrm{AAAB\underline{B}B}$ $\mathrm{AAA\underline{B}BB}$ $\mathrm{AA\underline{A}BBB}$

$\mathrm{AAA\underline{B}BB}$ $\mathrm{AAAB\underline{B}B}$ $\mathrm{AAABB\underline{B}}$ $\mathrm{AAABBB\underline{\Delta}}$ HALT

Example

Find TM that can accepts the language palindrome.



Let us trace the running of this TM on the input string: ababa

1 → 2 → 2 → 2 → 2 → 2 → 3 →

$\mathrm{\underline{a}baba}$ $\Delta\mathrm{\underline{b}aba}$ $\Delta\mathrm{b\underline{a}ba}$ $\Delta\mathrm{ba\underline{b}a}$ $\Delta\mathrm{bab\underline{a}}$ $\Delta\mathrm{baba}\underline{\Delta}$ $\Delta\mathrm{bab\underline{a}}$

4 → 4 → 4 → 4 → 1 → 5 → 5 →

$\Delta\mathrm{ba\underline{b}}\Delta$ $\Delta\mathrm{b\underline{a}b}\Delta$ $\Delta\mathrm{\underline{b}ab}\Delta$ $\underline{\Delta}\mathrm{bab}\Delta$ $\Delta\mathrm{\underline{b}ab}\Delta$ $\Delta\Delta\mathrm{\underline{a}b}\Delta$ $\Delta\Delta\mathrm{a\underline{b}}\Delta$

5 → 6 → 7 → 7 → 1 → 2 → 3 → 8

$\Delta\Delta\mathrm{ab}\underline{\Delta}$ $\Delta\Delta\mathrm{a\underline{b}}\Delta$ $\Delta\Delta\mathrm{\underline{a}}\Delta\Delta$ $\Delta\underline{\Delta}\mathrm{a}\Delta\Delta$ $\Delta\Delta\mathrm{\underline{a}}\Delta\Delta$ $\Delta\Delta\Delta\underline{\Delta}\Delta$ $\Delta\Delta\underline{\Delta}\Delta\Delta$ HALT

H.W

Find TM for even-even.