Lecture (1) Theory of Computation Second level

Introduction

1. Purpose and motivation

This course of the Theory of Computation, which tries to answer the following questions:

• What are the mathematical properties of computer hardware and software?
• What is a computation and what is an algorithm? Can we give rigorous mathematical definitions of these notions?
• What are the limitations of computers? Can "everything" be computed?

Purpose of the Theory of Computation: Develop formal mathematical models of computation that reflect real-world computers.

2. General concepts

1. Set :- is an unordered collection of objects, and as such a set is determined by the objects it contains.

-Operations on 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
    - $A \bigcup \phi = A$
    - $A \bigcap U = A$
- Complement properties
    - A U A’= U
    - A ∩ A’= ϕ

2. Alphabets:- is a finite, nonempty set of symbols. Conventionally, will used the symbols ∑ for an alphabet. Common alphabets include:

- ∑={1,0}, is the binary alphabet.
- ∑={a,b,…, z}, is the set of lower _case letters.
- ∑={ 0,1,2,…,9}.

3. Strings:- A string over an alphabet ∑ is a finite sequence of symbols, where each symbol is an element of ∑. The length of a string *w*, denoted by |*w*|, is the number of symbols contained in *w*. The empty string, denoted by ε is the string having length zero. For example, if the alphabet ∑ is equal to {0, 1}, then 10, 1000, 0, 101, and ε are strings over ∑, having lengths 2, 4, 1, 3, and 0, respectively.

- **Power of an alphabet**

If ∑ is an alphabet, can be expressed the set of all strings of a certain length from that alphabet by using an exponential notation. We define Σ^k to be the set of all strings with length *k*, each of whose symbols in ∑.

Example: for binary alphabet:

$\Sigma^0 = \{\varepsilon\}$

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

$\Sigma^2 = \{00,11,01,10\}$

$\Sigma^3 = \{000,111,011,001,010,110,100,101\}$

4. A language:- is a set of strings that can be consist it from that alphabet depends on the special grammar L.

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

Kinds of languages:

A-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 in English language.)

B- 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.

C- 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 (Grammars).

Note:

- $\sum^*$:-denotes the set of all sequences of strings that are composed of zero or more symbols of $\sum$.

- $\sum^+$:- denotes the set of all sequences of strings composed of one or more symbols of $\sum$. That denotes ($\Sigma^+ = \Sigma^* - \{\varepsilon\}$)

-{a,b}:this mean may be appear only a in the string or only b or may be appear the two symbols in the string .

-{ab}: this mean must be appear both symbols in the string respectively.

- if there is any symbols must be appear in the start or end the string this symbols must be put out side the Brackets

Example 1:

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.

Language is 0{0,1}*1

Example 2:

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.

Language is: a{a,b}*b

Example 3: 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.

Example 4: The language that ={ all strings x such that reverse(x)=x}
The string as aba, aabaa, bab, bbb,…

Example 5: Find the language for the following string (ababababab)
Answer: {ab}*.

Example 6: find the language for the following string (a, ab, abb,….)
Answer: a{b}*.

Example 7: Consider the language {a,b}*.How many words does this language have of length 2? of length 3? Of length n?
Answer: 4 words with length 2
9 words with length 3
L^n : for general length L with n alphabet .

H.W:
1. What are the string that accept in the language 0^n1^n
2. Write the language that accept the string(a,b,ab,aab,bb,abb,abbba,……).
3. Write the language that accept the string(0,001,00101,0010101…….).
4. Let language S accept *{ab, bb}* and let language *T accept {ab, bb, bbbb}.* Show that S* = T*. What principle does this illustrate?
5. Consider the language S*, where S = *{aa, b}*.How many words does this language have of length 4? of length 5? Of length 6? What can be said in general?

Intrudection

5. Grammar:-A formal of grammar can be define as quad- tuple G=(N,∑,P,S) :

- P: a finite set of production rules (left-hand side →right-hand side) where each side consists of a sequence of the following symbols:
- N: a finite set of nonterminal symbols (indicating that some production rule can yet be applied)
- ∑: a finite set of terminal symbols (indicating that no production rule can be applied)
- S: a start symbol (a distinguished nonterminal symbol)

A formal grammar defines (or generates) a formal language, which is a (usually infinite) set of finite-length sequences of symbols (i.e. strings) that may be constructed by applying production rules to another sequence of symbols which initially contains just the start symbol. A rule may be applied to a sequence of symbols by replacing an occurrence of the symbols on the left-hand side of the rule with those that appear on the right-hand side. A sequence of rule applications is called a derivation. Such a grammar defines the formal language: all words consisting solely of terminal symbols which can be reached by a derivation from the start symbol.

Note: Nonterminals are often represented by uppercase letters, terminals by lowercase letters, and the start symbol by S.

Example1: the grammar with terminals $\{a, b\}$, nonterminals $\{S, A, B\}$, and start symbol S, defines the language of all words of the form $a^n b^n$(i.e. $n$copies of $a$followed by $n$copies of b). The following is a simpler grammar that defines the same language: Terminals $\{a, b\}$, Nonterminals $\{S\}$, Start symbol S, Production rules

$S \to aSb$
$S \to \varepsilon$

Example2: productions: S→aS
S→ε

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

Example3: productions: S→SS
S→a
S→ε

Derivation of aa

S => SS => SSS => SSa => SSSa => SaSa => ε aSa => ε a ε a = aa

Note: can be rewrite an above grammar as S→SS|a| ε

Example4: Consider the grammar G where Vn = {S,B, C}, Vt = a, b, c,
S is the start symbol, and P consists of :

1. S → aBC
2. S → aSBC
3. aB → ab
4. bB → bb
5. CB → BC
6. bC → bc
7. cC → cc

try to provide one derivation for sentence aaabbbccc.
S ⇒2 aSBC ⇒2 aaSBCBC ⇒1 aaaBCBCBC ⇒3
aaabCBCBC ⇒5 aaabBCCBC ⇒4 aaabbCCBC ⇒5
aaabbCBCC ⇒5 aaabbBCCC ⇒4 aaabbbCCC ⇒6
aaabbbcCC ⇒7 aaabbbccC ⇒7 aaabbbccc

H.W:

1.production S→ aB| ε
B →bS| ε
Derivation the string of language {ab}*.

2. derivation the string (()()) depends the grammar with rules given below:
Rule1: S → SS
Rule2: S → (S)
Rule3: S → ( )

3. Derivations Trees

A string w accept with L(G) may have many derivations, corresponding to how we choose the rules to apply and how we choose which symbol to expand.

- At a given step we may be able to expand two or more non-terminal symbols. The order in which we expand the non-terminal symbols will determine the derivation, i.e. different orders will result in different derivations, even if we choose same rules.
- At a given step for a given non-terminal symbol there may be several rules to choose. Choosing different rules will result in different derivations.

Each derivation can be depicted using a derivation tree , also called a parse tree.

Each non-terminal symbol is expanded by applying a grammar rule that contains the symbol in its left- hand side. Its children are the symbols in the right-hand side of the rule.
Note: The order of applying the rules depends on the symbols to be expanded. At each tree level we may apply several different rules corresponding to the nodes to be expanded.

Parse Trees

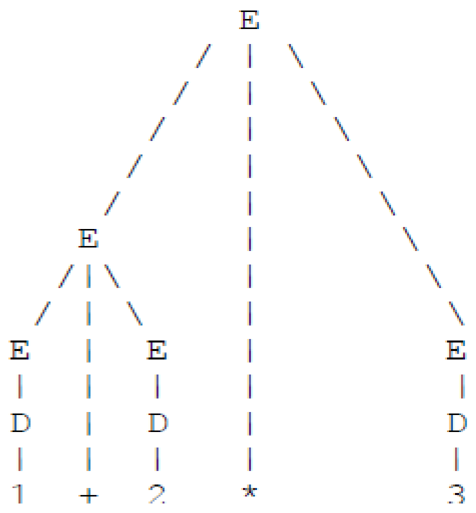
A parse tree for a string in L(G) is a tree where

- the root is the start symbol for G
- the interior nodes are the nonterminals of G
- the leaf nodes are the terminal symbols of G.
- the children of a node T (from left to right) correspond to the symbols on the right hand side of some rule for T in G.

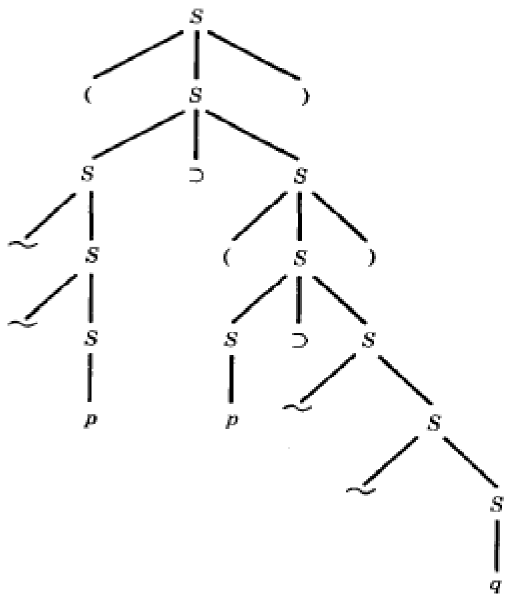
Every terminal string generated by a grammar has a corresponding parse tree; every valid parse tree represents a string generated by the grammar (called the yield of the parse tree).
Example: Given the grammar G = (V, Σ, R, E),
Σ = { 1,2,3,4,5,6,7,8,9,0,+,-,*,/,(,)}, and R contains the following rules:

1. E → D
2. E → (E)
3. E →E + E
4. E →E - E
5. E →E * E
6. E →E/ E
7. D → 0 | 1 | 2 | ... 9

find a parse tree for the string 1 + 2 * 3:



Example: S→ (S) | S⊃S |~S | p | q
The only nonterminal is S. The terminals are p q ~⊃ ( ) where "⊃" isthe symbol for implication. In this grammar consider the diagram:



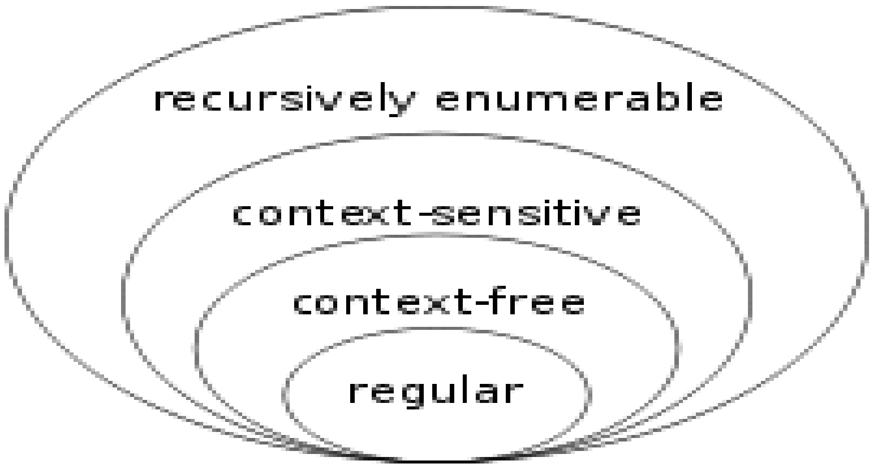
This is a derivation tree for the 13-letter word (~ ~ p⊃(p⊃~ ~ q))

H.W:-
1. If you have grammar S →(S + S)| (S * S) | number , derivate the arithmetic expression, and draw the derivation tree for it.
1. ((1 + 2) * (3 + 4) + 5) * 6. **2.** ((1 + **2)*** 3) + 4**.**

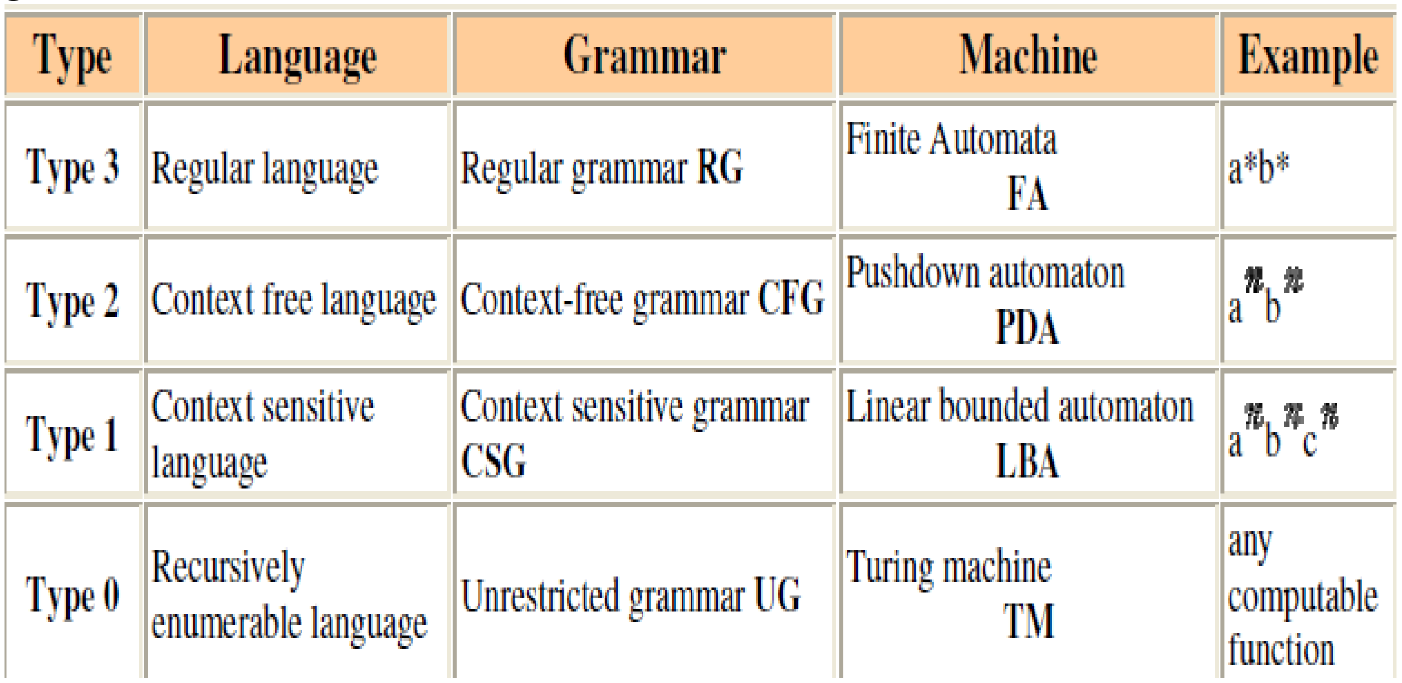
Lecture (3) Theory of Computation Second level

Chomsky hierarchy

The Chomsky hierarchy consists of the following levels:



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-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. 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.

  $U \rightarrow V$

S→SS
S→aAb
aA→Aa
BB→a
aA→bAa
Ba →bAb

- Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$with A anonterminal and α, $\beta$and γstrings of terminals and/or nonterminals. The strings $\alpha$and βmay be empty, but γmust be nonempty. The rule $S \rightarrow \epsilon$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 linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input.)

  U →V
  S →SS
  aA →bAa
  BB →aB
  A →ε wrong
  Left side <= right side

- Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form $A \rightarrow \gamma$with A is a nonterminal and γ is a string of terminals and/or nonterminals. Context-free languages – or rather the subset of deterministic context-free language – are the theoretical basis for the phrase structure of most programming languages, though their syntax also includes context-sensitive name resolution due to declarations and scope.

  S → (N U t)*
  S→SS/aA/bA
  S→ε
  S→ abB

- Type-3 grammars (regular grammars) generate the regular languages. Such a 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 (right regular). Alternatively, the right-hand side of the grammar can consist of a single terminal, possibly preceded by a single nonterminal (left regular); these generate the same languages. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

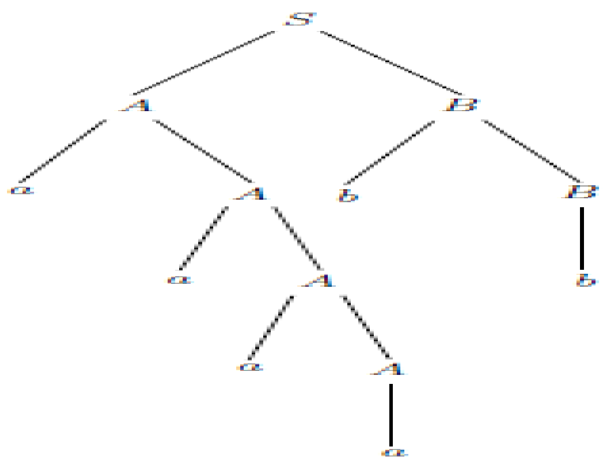
N _→ t|tN
A→ a|aB
S→ aS|b
Problem is left recursion A → Aa

Lecture (4) Theory of Computation Second level

Context-Free Grammars(CFG)

Definition : A context-free grammar is a 4-tuple G = (V,∑,R, S), where
1. V is a finite set, whose elements are called variables,
2. ∑ is a finite set, whose elements are called terminals,
3. $V \cap \Sigma = \emptyset$
4. S is an element of V ; it is called the start variable,
5. R is a finite set, whose elements are called rules. Each rule has the form A → w, where $A \in V$ and $w \in (V \cup \Sigma)^*$.

In our example. Consider the following five (substitution) rules:
S → AB
A →a
A →aA
B → b
B →bB
Here, S, A, and B are variables, S is the start variable, and (a, b) are terminals. We use these rules to derive strings consisting of terminals (i.e., elements of {a, b}*), in the following manner:
1. Initialize the current string to be the string consisting of the start variable S.
2. Take any variable in the current string and take any rule that has this variable on the left-hand side. Then, in the current string, replace this variable by the right-hand side of the rule.
3. Repeat 2. until the current string only contains terminals.
For example, the string aaaabb can be derived in the following way:
S → AB →aAB → aAbB →aaAbB →aaaAbB →aaaabB →aaaabb
This derivation can also be represented using a parse tree, as in the figure below:



Figure(1): derivation for string aaaabb with CFG

Example: Let us illustrate this on the CFG

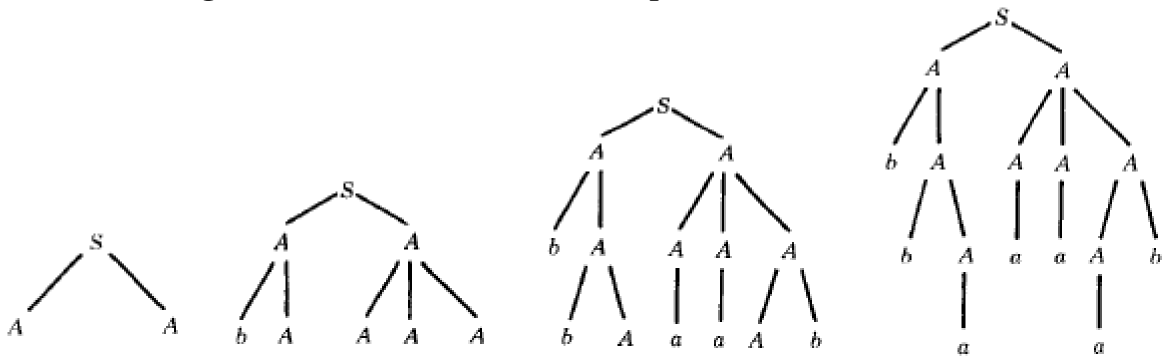
S → AA

A→ AAA |bA |Ab| a

We begin with S and apply the production S→ AA. To the left-hand A let us apply the production

A → bA. To the right-hand A. let us apply A → AAA.

The b that we have on the bottom line is a terminal, so it does not descend further. In the terminology of trees it is called a terminal node. Let the four A's, left to right, undergo the productions A → bA, A → a, A → a, A → Ab , respectively. Let us finish off the generation of a word with the productions A → a and A → a:



Figure(2): derivation tree for the string bbaaaab

H.W:

1.Detect if the following grammar CFG or not: S →S + S |S * S | number.

2.Write the Grammar for the following language $\{a^m b^n : m >= 1, n >= 1\}$, and detect if its Context free grammar or not.

Context-free languages

Definition : Let G = (V,∑,R, S) be a context-free grammar. The language of G is defined to be the set of all strings in ∑ that can be derived from the start variable S:

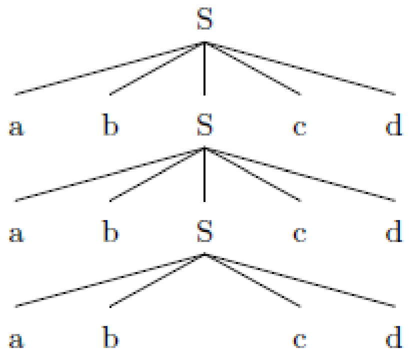
$$L(G) = \{w \in \sum^{*} : S \overset{*}{\Rightarrow} w\} = L$$

This point can be illustrated with the language $(ab)^n(cd)^n$. A simple grammar for it has only two rules:

• S → abScd,

• S → abcd.

The derivation for the string abababcdcdcd can succinctly be represented by the phrase structure tree given in the following Figure.



Figure(3): derivation tree for the string abababcdcdcd

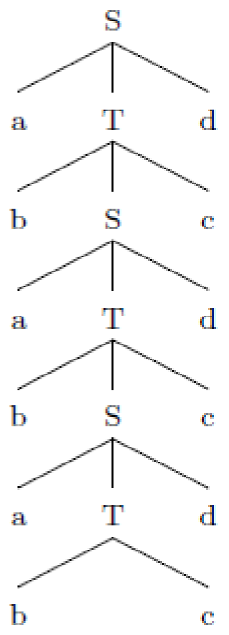
In such a tree diagram, each local tree (i.e. each node together with the nodes below it that are connected to it by a direct line) represents one rule application, with the node on top being the left-hand side and the nodes on the bottom the right-hand side. The sequence that is derived can be read off the leaves (the nodes from which no line extends downward) of the tree.

The same language can also be described by a some what more complex grammar, using the rules:

• S → aTd,

• T → bSc,

• T → bc.



Figure(4) : Different phrase structure tree for the same string

Example: Let the terminals be a and b, let the nonterminals be S and X, and let the productions be

S → XaaX

X→ aX

X→*bX*

X→*ε*

We already know from the previous example that the last three productions will allow us to generate any word we want from the nonterminal X. If the nonterminal X appears in any working string we can apply productions to turn it into any word we want. Therefore, the words generated from S have the form **(anything aa anything) .** Or Context free language (a + b)*aa(a + b)*, which is the language of all words with a double a in them somewhere.

For example, to generate *baabaab* we can proceed as follows:

S→ XaaX **:** *bXaaX* → baXaaX→ baaXaaX→ *baabXaaX*→ baabAaaX→*baabaaX*→*baabaabX* → *baabaabε* →*baabaab.*

There are other sequences that also derive the word *baabaab.*

Note: The membership problem for context-free languages is solvable in cubic time, i.e. the maximum time that is needed to decide whether a given string x belongs to L(G) for some context-free grammar G grows with the third power of the length of x. This means that there are efficient algorithms to solve this problem.

H.W:

1.find the Context free language for the CFG, S→aSbS|bSaS|ε, and Prove L(G) is a set of all string with an equal number of a's and b's.
2. find the CFG for the following language {a,b}*a.
3. find the string that accept by the language {a,b}*bbb{a,b}*.

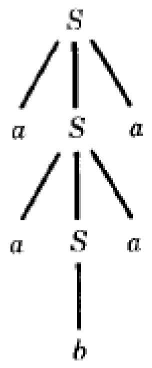
Note: in some references {a,b}=(a+b), {ab}=(ab)or(a*b)

Lecture (5) Theory of Computation Second level

Ambiguity

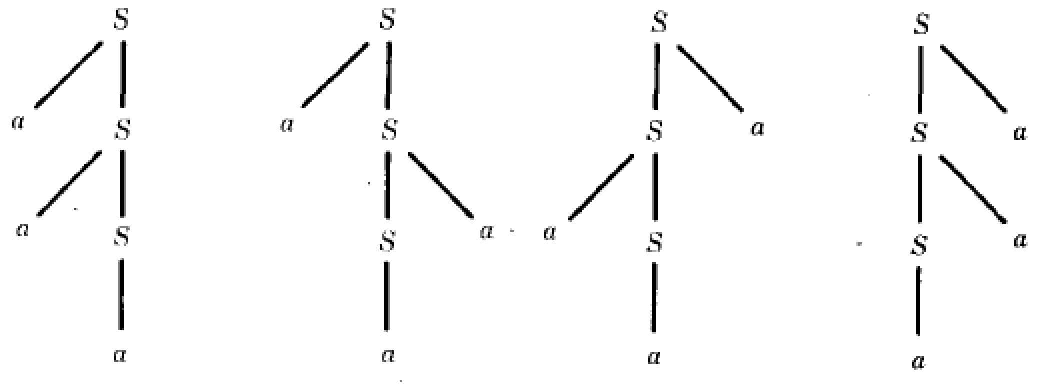
A CFG is called ambiguous if for at least one word in the language that it generates there are two possible derivations of the word that correspond to different syntax trees.

Example: Let us reconsider the language PALINDROME, which we can now define by the CFG below:
S→ aSa | bSb a | b A
At every stage in the generation of a word by this grammar the working string contains only the one nonterminal S smack dab in the middle. The word grows like a tree from the center out.
For example: baSab → babSbab→ babbSbbab → babbaSabbab ...
When we finally replace S by a center letter (or A if the word has no center letter) we have completed the production of a palindrome. The word aabaa has only one possible generation:
S → aSa
→ aaSaa
→aabaa



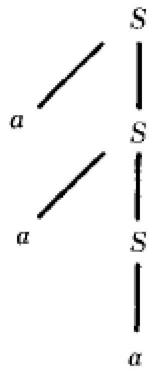
We see then that this CFG is unambiguous.

Example: The language of all nonnull strings of a's can be defined by a CFG as follows:
S→ aS | Sa | a
In this case the word a3 can be generated by four different trees:



This CFG is therefore ambiguous.

However the same language can also be defined by the CFG: S → aS | a
for which the word a^3 has only one production:



From this last example we see that we must be careful to say that it is the CFG that is ambiguous, not that the language itself is ambiguous.

Grammar G: E → E +E|E *E|(E)|I and I → Ia|Ib|I0|I1|a|b is ambiguous since a+b*a has two parse trees.

- Some ambiguous grammars have an equivalent unambiguous grammar. For example, an unambiguous grammar for the simple expressions is G0: E → E +T|T, T → T *F|F, F → (E)|I, and I → Ia|Ib|I0|I1|a|b.
- A context-free language is said to be inherently ambiguous if all its grammars are ambiguous. For example, $\{a^nb^nc^md^m \ |m,n >=1\} \cup \{a^nb^mc^md^n \ |m,n >=1\}$. Its grammar is S→ S1|S2, S1 → AB, A→ aAb|ab, B→ cBd|cd, S2 → aS2d|aCd, and

C→ bCc|bc. The grammar is ambiguous (considering abcd). There is a not so easy proof that all grammars for the language are ambiguous, thus it is inherently ambiguous.

- There is no algorithm to determine whether a given CFG is ambiguous. There is no algorithm to remove ambiguity from an ambiguous CFG. There is no algorithm to determine whether a given CFL is inherently ambiguous.

<u>The empty string in Context Free Grammar's</u>

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

Definition:
In a given CFG, we call a nonterminal N **nullable** if:
• There is a production: N→ ε
Or
• There is a derivation that start at N and leads to ε : $N \xrightarrow{*} \varepsilon$

Theorem:
For any CFL (L), there exists an ε-free CFG,G, such that L(G) = L\{ε}

Let G = (N,T,P,S) be any CFG with ε- productions. Then G' =(N,T,P',S), where P' is constructed from P as follows:
1. Put all the ε-free productions of P into P'.
2. Find all the nonterminals AЄN such that A →ε.
Example:

S→ [E] | E
E →T| E+T | E-T
T →F| T*F | T/F
F →a| b| c| ε
The ε-free grammar constructed from G has productions
S → [E] | E |[ ]
E →T| E+T | E-T | E+ | E- | +T | -T | + | -
T →F| T*F | T/F
Example:
Consider the CFG:
S → a| Xb| aYa
X → Y | ε
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 | ε
X is the only nullable nonterminal.
The new CFG is:
S → Xa| a
X → aX| bX |a| b
Example:
Consider the CFG:
S → XY
X → Zb
Y→ bW
Z → AB
W→ Z
A → aA| bA| ε
B → Ba| Bb| ε
A, B, W and Z are nullable.
The new CFG is:
S → XY
X → Zb| b
Y→ bW| b

$Z \rightarrow AB|\ A|\ B$
$W \rightarrow Z$
$A \rightarrow aA|\ bA|\ a|\ b$
$B \rightarrow Ba|\ Bb|\ a|\ b$

Simplification of Context Free Grammar

Since there are different people may come up with different but equivalent CFGs. There for must be Simplify CFG . That can be done with several steps:

- Eliminating ε rules for $L(G)-\{\varepsilon\}$ by finding nullable variables ($A \xrightarrow{*} \varepsilon$)

  For example:

  $S \to AB$

  $A \to aAA|\varepsilon$

  $B \to bBB|\varepsilon$

  can be changed to:

  $S \to AB|A|B$

  $A \to aAA|aA|a$

  $B \to bBB|bB|b$.

- Eliminating unit rules.

  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 ε-production, then there is also a CFG for L with no ε-production and no unit production.

  Example:

  Consider the CFG:

  S → A| bb

  A → B| b

  B → S| a

  S → A gives S → b

  S → A → B gives S → a

  A → B gives A → a

  A → B → S gives A → bb

  B → S gives B → bb

  B → S → A gives B → b

  The new CFG for this language is:

  S → bb| b| a

  A → b| a| bb

  B → a| bb| b

  Example:

E →T|E +T
T → F|T*F
F →I|(E)
I → Ia|Ib|I0|I1|a|b
can be changed to:
E→E+T|T*F|(E)|Ia|Ib|I0|I1|a|b
T →T*F|(E)|Ia|Ib|I0|I1|a|b
F → (E)|Ia|Ib|I0|I1|a|b
I →Ia|Ib|I0|I1|a|b.

- Eliminating useless variables (and thus associated rules) by finding non generating and unreachable variables.
  Nongenerating: is loop derivation with out ending.
  Unreachable: is a symbol that cannot reach to it from the start symbol.
  Example:
  S→AB|a
  A→b
  can be simplified to:
  S→a.

Lecture (8) Theory of Computation Second level

Chomsky Normal Form (CNF)

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

1. Nonterminal → string of only Nonterminals (*A→BC)*
2. Nonterminal → One Terminal (*A→a*)

where *A*,*B*,*C* are variables, and *a* is a 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)**.

Note: that one of the uses of CNF is to turn parse trees into binary trees.

To convert a CFG to a grammar in CNF:

- Add a new start variable S_0 in the case when the old start variable S appears in the body of some rules.
- Simplify the grammar by removing ε rules, unit rules, and useless variables.
- Convert the rules in the simplified grammar into the proper forms of CNF by adding additional variables and rules.

Example:
Consider the CFG:
$S \to X_1 | X_2aX_2 | aSb | b$
$X_1 \to X_2X_2 | b$
$X_2 \to aX_2 | aaX_1$
Becomes:
$S \to X_1$
$S \to X_2AX_2$
$S \to ASB$
$S \to B$
$X_1 \to X_2X_2$
$X_1 \to B$
$X_2 \to AX_2$
$X_2 \to AAX_1$
$A \to a$
$B \to b$
Example:
Consider the CFG:
$S \to Na$
$N \to a | b$
Becomes:
$S \to NA$
$N \to a | b$
$A \to a$
Example:
Convert the following CFG into CNF:
$S \to aSa | bSb | a | b | aa | bb$
$S \to ASA$
$S \to BSB$
$S \to AA$
$S \to BB$
$S \to a$
$S \to b$

$A \to a$
$B \to b$
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:
$S \to YA \mid XB$
$A \to YR_1 \mid XS \mid a$
$B \to XR_2 \mid YS \mid b$
$X \to a$
$Y \to b$
$R_1 \to AA$
$R_2 \to BB$

H.W: Convert the following CFG into CNF:
1.$S \to AAAAS$
$S \to AAAA$
$A \to a$
2. $S \to Aba$
$A \to aab$
$B \to AC$

Greibach normal form(GNF)

A context-free grammar is in Greibach normal form (GNF) if the right-hand sides of all productions start with a terminal symbol. A context-free grammar is in Greibach normal form, if all production rules are of the form:

A→aX

where *A* is a nonterminal symbol,

a is a terminal symbol,

X is a (possibly empty) sequence of nonterminal symbols not including the start symbol.

Example1:convert the following CFG to GNF:

S→ AB

A →BS| b

B →SA| a

1. Convert the CFG to CNF
2. Rename nonterminals (A,S,B)

    S becomes A1

    A becomes A2

    B becomes A3

    The grammar becomes:

    A1→ A2A3

    A2 →A3A1| b

    A3→A1A2| a

3. Compare the value of i,j as Ai→Aj (j > i)

A1→A2A3	(2 > 1)
A2→A3A1| b	(3 > 2)
A3→A1A2| a	(1 < 3)
A3→A2A3A2 | a	(2 < 3)
A3→A3A1A3A2 | bA3A2 | a	(3 = 3)

    B3→A1A3A2 | A1A3A2B3

    ****

    The GNF :

    A3→bA3A2 | bA3A2B3 | a | aB3

A2⟶bA3A2A1| aA1 | bA3A2B3A1 | aB3A1| b

A1⟶bA3A2A1A3| aA1A3 | bA3A2B3A1A3 | aB3A1A3| bA3

B3⟶bA3A2A1A3A3A2| aA1A3 A3A2 | bA3A2B3A1A3 A3A2 | aB3A1A3 A3A2| bA3 A3A2 | bA3A2A1A3A3A2B3| aA1A3 A3A2B3 | bA3A2B3A1A3 A3A2B3 | aB3A1A3 A3A2B3| bA3 A3A2B3

Example: convert CFG to GNF

S⟶ ASB | AB

A⟶a

B⟶b

Convert to CNF

S⟶AR1 | AB

R1⟶SB

A⟶a

B⟶b

S becomes A1
R1 becomes A2
A becomes A3
B becomes A4

The grammar becomes:

A1⟶A3A2 | A3A4 3>1

A2⟶A1A4 1< 2

A3⟶a

A4⟶b

A2⟶A3A2A4 | A3A4A4

The GNF:

A2⟶aA2A4 | aA4A4

A1⟶aA2 | aA4

A3⟶a

A4⟶b

Regular expression

Regular expressions denote languages. For a simple example, the regular expression 01*+10* denotes the language consisting of all strings that are either a single of 0 followed by any number of 1's or single 1 followed by any number of 0's .

Before the describing the regular expression notation we need to learn the three operations on languages that the operators of regular expressions represent. These operations are:

1. The union of two languages L and M denoted $L\cup M$, is the set of string that are in either L or M or both.
   Example : L={001, 01,111}, M={ε,001} that make $L\cup M$={ ε,01,001,111}.
2. The concatenation of languages L and M is the set of the string that can be formed be taking any string in L and concatenating it with any string in M. We denote concatenation language either with dot or no operation at all.
   Example:L={001,01,111},M={ε,001},
   Concatenation L and M is LM=L.M={001,01,111,001001,01001,111001}.
   The first three string in LM are the concatenation the string of L with ε and the last three string in LM are formed by taking each string in L and concatenation with second string in M.
   Example: If S = {a, aa, aaa}, T = {bb, bbb}
   Then ST = {abb, abbb, aabb, aabbb, aaabb, aaabbb}
   Note that these words are not in proper order.
   Example: If **S =** {a, bb, bab} T **=** {a, ab}
   Then ST = {aa, aab ,bba, bbab, baba, babab}
3. The closure (or star or Keene closure) of language L denoted L* and represent the set of those string that can be formed by taking any number of string from L, possible with repetitions (i.e. that same string may be selected more than once) and concatenating all of them.
   Example: If L={0,11} then L*={ ε, 0,11110,0011,11011,…} but not (1011).

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

Rule 1: Every letter of U can be made into a regular expression by writing it in
Bold face ;ε is a regular expression.

Rule 2: If r1, and r2 are regular expressions, then so are (rl) ,r1r2 ,r 1 + r 2, rl*

Rule 3 Nothing else is a regular expression.

Note: We could have included the plus sign as a superscript $r1^+$ as part of the definition, but since we know that $rl^+ = rlrl^*$, this would add nothing valuable.

As with the recursive definition of arithmetic expressions, we have included the use of parentheses as an option, not a requirement. Let us emphasize again the implicit parentheses in rl*.

If r1 = aa + b

then the expression r1* technically refers to the expression

r1* = aa **+ b***

which is the formal concatenation of the symbols for r, with the symbol *, but what we generally mean when we write r1* is actually (rl)*

(r1)* = (aa **+ b)***

which is different. Both are regular expressions and both can be generated from the rule.

Note:

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

(a*)*=a*

(aa+ab*)*≠ (aa+ab)*

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

Examples:

The regular expression (00+11)*(101+110) represents the regular set (regular language) {101, 110, 00101, 00110, 11101, 11110, 0000101, 0000110, 0011101, 0011110, . . .}.

H.W: What are some other strings in this language? Is 00110011110 in the language? How about 00111100101110?

Regular Grammar

Definition:

A CFG is called a regular grammar if each of its productions is of one of the two forms

Nonterminal → semiword

Nonterminal → word

The two previous proofs imply that all regular languages can be generated by regular grammars and all regular grammars generate regular languages.

Definition:

For a given CFG a **semiword** is a string of terminals (maybe one) concatenated with exactly one nonterminal (on the right), for example:

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

Contrast this with **word**, which is a string of all terminals, and **working string**, which is a string of any number of terminals and nonterminals in any order.

Example: Consider the CFG: (aa + **bb)***

S→aaS|bbS **|** ε

Regular language

Regular languages are those languages that are defined by regular grammars.
In such a grammar, all rules take one of the following two forms:
A → a, A → aB.
Here A and B stand for non-terminal symbols and a for a terminal symbol

Closure Properties of regular language

Recall a closure property is a statement that a certain operation on languages, when applied to languages in a class(e.g., the regular languages), produces a result that is also in that class.

1. The union of two regular language is regular.
2. The closure (stare) of regular language is a regular.
3. The concatenation of c regular language is a regular.
4. The intersection of two regular language is regular.
5. The complement of a regular language is a regular.
6. A homomorphism (substitution of strings for symbols) of regular language is regular.
7. The difference between two regular language is a regular.
8. The reverce of regular language is a regular.
9. The inverse of homomorphism of regulare language is regular.

Pumping lemma for regular languages

From Wikipedia, the free encyclopedia

In the theory of formal languages, the **pumping lemma for regular languages** describes an essential property of all regular languages. Informally, it says that all sufficiently long words in a regular language may be *pumped* — that is, have a middle section of the word repeated an arbitrary number of times — to produce a new word that also lies within the same language.

Specifically, the pumping lemma says that for any regular language L there exists a constant p such that any word w in L with length at least p can be split into three substrings, $w = xyz$, where the middle portion y must not be empty, such that the words xz, xyz, $xyyz$, $xyyyz$, … constructed by repeating y an arbitrary number of times (including zero times) are still in L. This process of repetition is known as "pumping". Moreover, the pumping lemma guarantees that the length of xy will be at most p, imposing a limit on the ways in which *w* may be split. Finite languages trivially satisfy the pumping lemma by having *p* equal to the maximum string length in *L* plus one.

The pumping lemma is useful for disproving the regularity of a specific language in question.

Formal statement

Let *L* be a regular language. Then there exists an integer $p \geq 1$ depending only on *L* such that every string *w* in *L* of length at least *p* (*p* is called the "pumping length"[4]) can be written as $w = xyz$ (i.e., *w* can be divided into three substrings), satisfying the following conditions:

1. $|y| \geq 1$;
2. $|xy| \leq p$
3. for all $i \geq 0$, $xy^iz \in L$

y is the substring that can be pumped (removed or repeated any number of times, and the resulting string is always in *L*). (1) means the loop *y* to be pumped must be of length at least one; (2) means the loop must occur within the first *p* characters. $|x|$ must be smaller than *p* (conclusion of (1) and (2)), apart from that there is no restriction on *x* and *z*.

In simple words, for any regular language L, any sufficiently long word w (in L) can be split into 3 parts. i.e. w = xyz , such that all the strings xy^kz for $k\geq0$ are also in L.

Below is a formal expression of the Pumping Lemma.

$$
\begin{array}{l}
(\forall L \subseteq \Sigma^*) \\
\quad (\text{regular}(L) \Rightarrow \\
\quad ((\exists p \geq 1)((\forall w \in L)((|w| \geq p) \Rightarrow \\
\quad ((\exists x, y, z \in \Sigma^*)(w = xyz \wedge (|y| \geq 1 \wedge |xy| \leq p \wedge (\forall i \geq 0)(xy^iz \in L)))))))
\end{array}
$$

Lecture (11) Theory of Computation Second level

Finite Automata

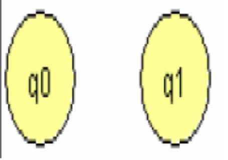
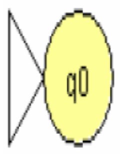
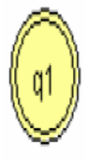
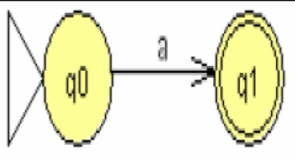
is a device consisting of a tape and a control circuit which satisfy the following conditions:

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.

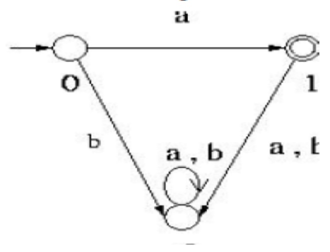
4. The head moves to the right one square every time it reads a symbol. It never moves to the left. When it sees no symbol, it stops and the automata terminates its operation.
5. There is a control determines the state of the automaton and also controls the movement of the head.

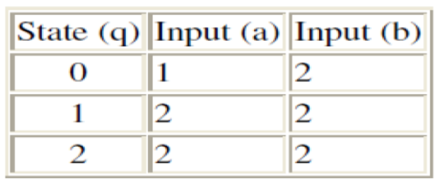
A finite automaton (FA): provides the simplest model of a computing device. It has a central processor of finite capacity and it is based on the concept of state.
finite state machine is a 5 tuple M = (Q, A, T, S ,F), where
o Q --set of states = {q0, q1, q2, ….}
o A -- set of input symbols(alphabet) ={a,b, …, 0, 1, …}
o T --set of transitions or rules
o S -- an initial state
o 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

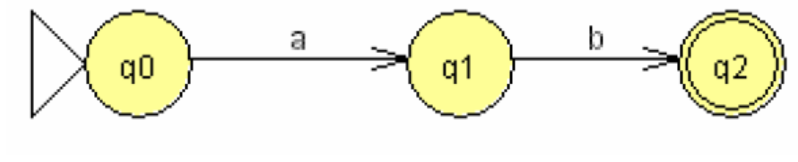
Example: Q = { 0, 1, 2 }, A= { a, b }, F = { 1 }, the initial state is 0 and T is shown in the following table.





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

Example: the following FA: recognize (accept) string ab

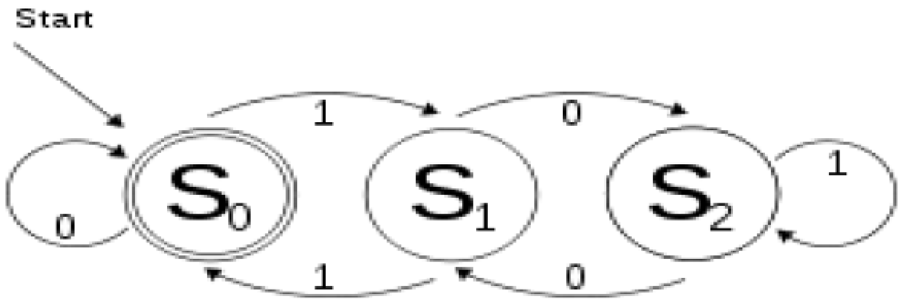


Type of FA:

A. **Deterministic Finite State Automata (DFSA)**
B. **Non Deterministic Finite State Automata (NFA)**

A.Deterministic Finite State Automata (DFSA)

A DFA represents a finite state machine that recognizes a RE(regular expression).

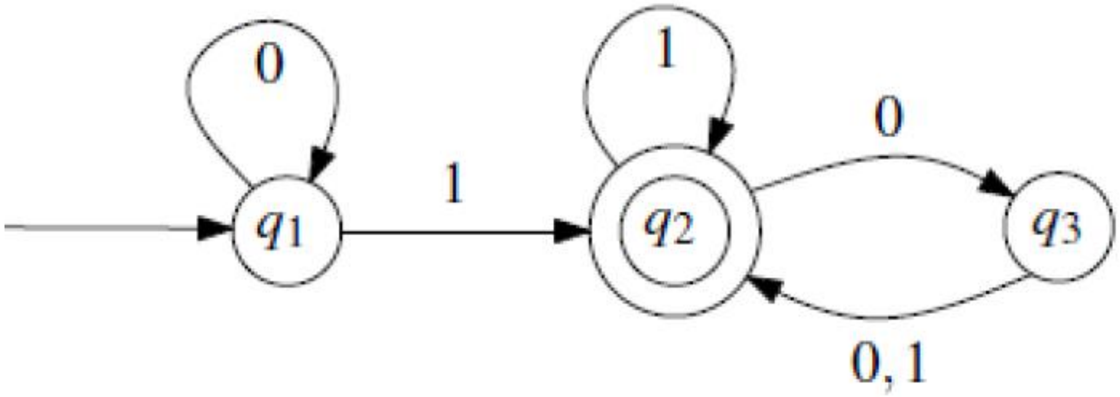


An example of a deterministic finite automaton that accepts only binary numbers that are multiples of 3. The state *S*0 is both the start state and an accept state.
The figure on above illustrates a deterministic finite automaton using a state diagram. In the automaton, there are three states: S0, S1, and S2 (denoted graphically by circles). The automaton takes a finite sequence of 0s and 1s as input. For each state, there is a transition arrow leading out to a next state for both 0 and 1. Upon reading a symbol, a DFA jumps *deterministically* from a state to another by following the transition arrow. For example, if the automaton is currently in state S0 and current input symbol is 1 then it deterministically jumps to state S1. A DFA has a *start state* (denoted graphically by an arrow coming in from nowhere) where computations begin, and a set of *accept states* (denoted graphically by a double circle) which help define when a computation is successful.
Note:- in the DFSA for each input string there is single target state.
Definition: A deterministic finite automaton *M* is a 5-tuple, (*Q*, Σ, δ, *q0*, *F*), consisting of
- a finite set of states (Q)
- a finite set of input symbols called the alphabet (Σ)
- a transition function (δ : $Q \times \Sigma \rightarrow Q$)
- a start state ($q0 \in Q$)

 a set of accept states (F ⊆ Q)

In words, the first condition says that the machine starts in the start state *q*0. The second condition says that given each character of string *w*, the machine will transition from state to state according to the transition function δ. The last condition says that the machine accepts *w* if the last input of *w* causes the machine to halt in one of the accepting states. Otherwise, it is said that the automaton *rejects* the string. The set of strings *M* accepts is the language *recognized* by *M* and this language is denoted by *L(M)*.
A deterministic finite automaton without accept states and without a starting state is known as a transition system or ***semiautomaton.***

Accept and Generate modes
A DFA representing a regular language can be used either in an accepting mode to validate that an input string is part of the language, or in a generating mode to generate a list of all the strings in the language.
In the accept mode an input string is provided which the automaton can read in left to right, one symbol at a time. The computation begins at the start state and proceeds by reading the first symbol from the input string and following the state transition corresponding to that symbol. The system continues reading symbols and following transitions until there are no more symbols in the input, which marks the end of the computation. If after all input symbols have been processed the system is in an accept state then we know that the input string was indeed part of the language, and it is said to be accepted, otherwise it is not part of the language and it is not accepted.
Example: Consider the following state diagram:



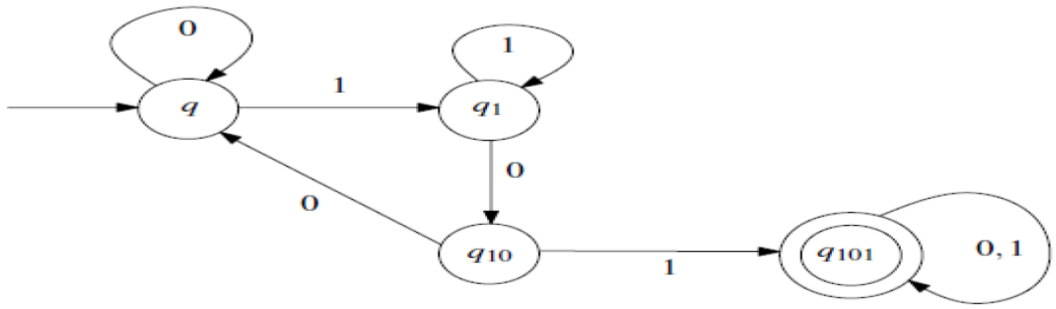
We say that q1 is the start state and q2 is an accept(final) state. Consider the input string 1101. This string is processed in the following way:
• Initially, the machine is in the start state q1.
• After having read the first 1, the machine switches from state q1 to state q2.

• After having read the second 1, the machine switches from state q2 to state q2. (So actually, it does not switch.)
• After having read the first 0, the machine switches from state q2 to state q3.
• After having read the third 1, the machine switches from state q3 to state q2. After the entire string 1101 has been processed, the machine is in state q2, which is an accept state. We say that the string 1101 is accepted by the machine.
Consider now the input string 0101010. After having read this string from left to right (starting in the start state q1), the machine is in state q3. Since q3 is not an accept state, we say that the machine rejects the string 0101010.
We hope you are able to see that this machine accepts every binary string that ends with a 1. In fact, the machine accepts more strings:
• Every binary string having the property that there are any number of 0s following the rightmost 1, is accepted by this machine.
• Every other binary string is rejected by the machine.
Example:
Define the language A as A = {w : w is a binary string containing 101 as a substring}.
Again, we claim that A is a regular language. In other words, we claim that there exists a finite automaton M that accepts A, i.e., A = L(M).
The finite automaton M will do the following, when reading an input string from left to right:
• It skips over all 0s, and stays in the start state.
• At the first 1, it switches to the state "maybe the next two symbols are 01".
– If the next symbol is 1, then it stays in the state "maybe the next two symbols are 01".
– On the other hand, if the next symbol is 0, then it switches to the state "maybe the next symbol is 1".
_ If the next symbol is indeed 1, then it switches to the accept state (but keeps on reading until the end of the string).
_ On the other hand, if the next symbol is 0, then it switches to the start state, and skips 0s until it reads 1 again.
By defining the following four states, this process will become clear:
• q1: M is in this state if the last symbol read was 1, but the substring 101 has not been read.
• q10: M is in this state if the last two symbols read were 10, but the substring 101 has not been read.
• q101: M is in this state if the substring 101 has been read in the input string.
• q: In all other cases, M is in this state.
Here is the formal description of the finite automaton that accepts the language A:
• Q = {q, q1, q10, q101},
• Σ = {0, 1},
• the start state is q,

• the set F of accept states is equal to F = {q101}, and
• the transition function is given by the following table:

	0	1
q	q	q_1
q_1	q_{10}	q_1
q_{10}	q	q_{101}
q_{101}	q_{101}	q_{101}

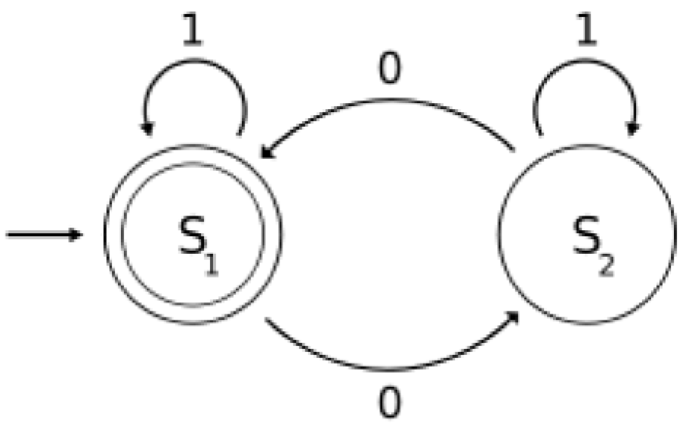
The figure below gives the state diagram of the finite automaton :



This finite automaton accepts the language A consisting of all binary strings that contain the substring 101.

Example: The following example is of a DFA *M*, with a binary alphabet, which requires that the input contains an even number of 0s.

The state diagram for *M* = (*Q*, Σ, δ, *q0*, *F*) where:



The state diagram for $M = (Q, \Sigma, \delta, q_0, F)$ where:

- $Q = \{S_1, S_2\}$,
- $\Sigma = \{0, 1\}$,
- $q_0 = S_1$,
- $F = \{S_1\}$, and
- δ is defined by the following state transition table:

	0	1
S_1	S_2	S_1
S_2	S_1	S_2

The language recognized by M is the regular language given by the regular expression 1*(0 (1*) 0 (1*))*

Lecture (13) Theory of Computation Second level

Converting an NFA to a DFA

Given:
A *non-deterministic* finite state machine (NFA)
Goal:
Convert to an equivalent *deterministic* finite state machine (DFA)
Why?
Faster recognizer!
Approach:
Consider simulating a NFA.
Work with sets of states.

IDEA: Each *state* in the DFA will correspond to a *set of* NFA states.
Worst-case:
There can be an exponential number **O**(2N) of sets of states.
The DFA can have exponentially many more states than the NFA
... but this is rare.

Let **X = (Q_x, ∑, δ_x, q_0, F_x)** be an NDFA which accepts the language L(X). We have to design an equivalent DFA **Y = (Q_y, ∑, δ_y, q_0, F_y)** such that **L(Y) = L(X)**. The following procedure converts the NDFA to its equivalent DFA −

Algorithm

Input − An NDFA

Output − An equivalent DFA

Step 1 − Create state table from the given NDFA.

Step 2 − Create a blank state table under possible input alphabets for the equivalent DFA.

Step 3 − Mark the start state of the DFA by q0 (Same as the NDFA).

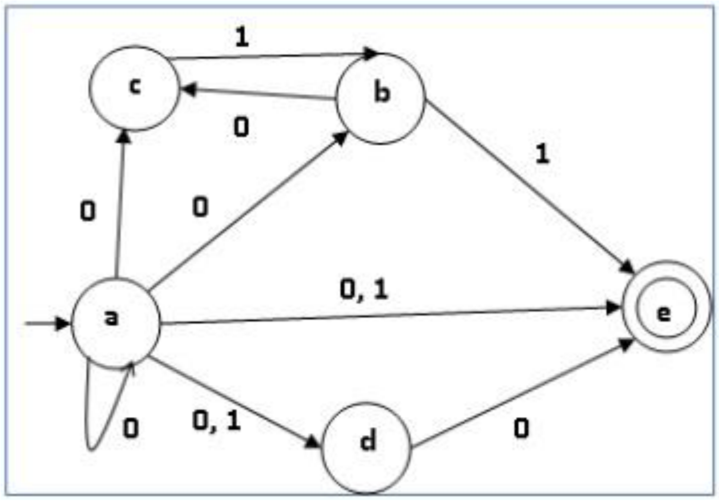
Step 4 − Find out the combination of States {Q_0, Q_1,... , Q_n} for each possible input alphabet.

Step 5 − Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

Step 6 − The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

Example

Let us consider the NDFA shown in the figure below.

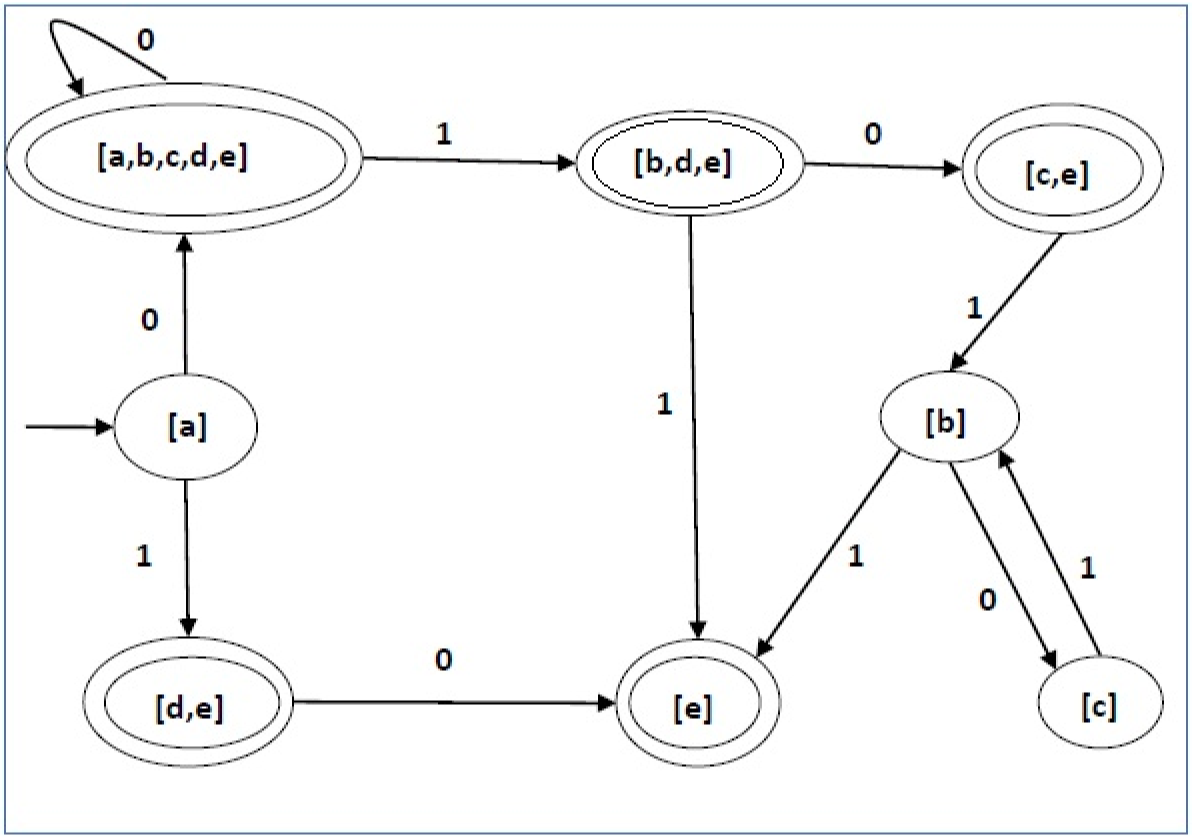


q	δ(q,0)	δ(q,1)
a	{a,b,c,d,e}	{d,e}
b	{c}	{e}
c	∅	{b}
d	{e}	∅
e	∅	∅

Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

q	δ(q,0)	δ(q,1)
[a]	[a,b,c,d,e]	[d,e]
[a,b,c,d,e]	[a,b,c,d,e]	[b,d,e]
[d,e]	[e]	∅
[b,d,e]	[c,e]	[e]
[e]	∅	∅
[c, e]	∅	[b]
[b]	[c]	[e]
[c]	∅	[b]

The state diagram of the DFA is as follows −


[a,b,c,d,e]
[b,d,e]
[c,e]
[a]
[b]
[d,e]
[e]
[c]
0
1
0
0
1
1
1
1
0
0
1
1