K G R L PG COURSES - BHIMAVARAM

FORMAL LANGUAGES & AUTOMATA THEORY

BY

K.ISSACK BABU

Assistant Professor

FORMAL LANGUAGES & AUTOMATA THEORY

UNIT-I

Finite Automata and Regular Expressions: Basic Concepts of Finite State Systems, Chomsky Hierarchy of Languages, Deterministic and Non-Deterministic Finite Automata, Finite Automata with ϵ-moves, Regular Expressions. Regular sets &Regular Grammars: Basic Definitions of Formal Languages and Grammars, Regular Sets and Regular Grammars, Closure Properties of Regular Sets, Pumping Lemma for Regular Sets, Decision Algorithm for Regular Sets, Minimization of Finite Automata.

UNIT-II

Context Free Grammars and Languages: Context Free Grammars and Languages, Derivation Trees, simplification of Context Free Grammars, Normal Forms, Pumping Lemma for CFL, Closure properties of CFL's. Push down Automata: Informal Description, Definitions, Push-Down Automata and Context free Languages, Parsing and Push-Down Automata.

UNIT-III

Turing Machines: The Definition of Turing Machine, Design and Techniques for Construction of Turing Machines, Combining Turing Machines. Universal Turing Machines and Undecidability: Universal Turing Machines. The Halting Problem, Decidable & Undecidable Problems - Post Correspondence Problem.

UNIT-IV

The Propositional calculus: The Prepositional Calculus : Introduction – Syntax of the Prepositional Calculus – Truth-Assignments – Validity and Satisfiability – Equivalence and Normal Forms – resolution in Prepositional Calculus. The Predicate calculus: Syntax of the Predicate Calculate Calculus – Structures and Satisfiability – Equivalence – Un-solvability and NP-Completeness.

TEXT BOOKS:

1. Introduction to Automata Theory, Languages and Computations – J.E. Hopcroft, & J.D. Ullman , Pearson Education Asia.
2. Elements of The Theory Of Computation, Harry R Lewis, Cristos h. Papadimitriou, Pearson Education / Prentice-Hall of India Private Limited.

REFERENCE BOOKS:

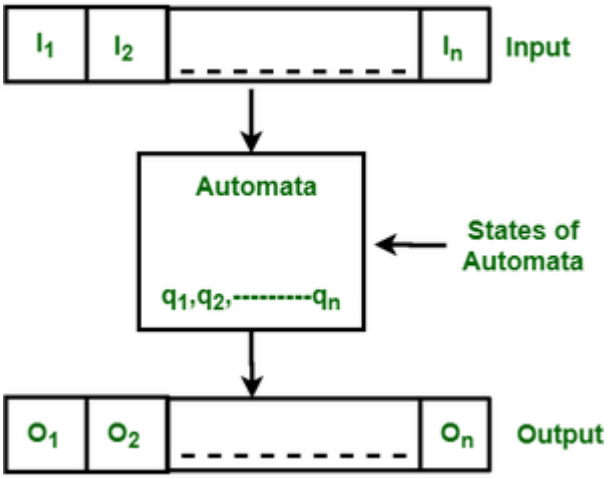
1. Introduction to languages and theory of computation – John C. Martin (MGH)

2. Theory of Computation, KLP Mishra and N. Chandra Sekhar, IV th Edition, PHI 3. Introduction to Theory of Computation – Michael Sipser (Thomson Nrools/Cole)

UNIT-I

Introduction of Finite Automata:

Finite Automata (FA) is the simplest machine to recognize patterns. The finite automata or finite state machine is an abstract machine that has five elements or tuples. It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. Basically, it is an abstract model of a digital computer. The following figure shows some essential features of general automation.



1. Input
2. Output
3. States of automata
4. State relation
5. Output relation

A Finite Automata consists of the follow :

Q: Finite set of states.

Σ : set of Input Symbols.

q : Initial state.

F : set of Final States.

δ : Transition Function.

Formal specification of machine is

{ Q, Σ, q, F, δ }

FA is characterized into two types:

1) Deterministic Finite Automata (DFA) –
DFA consists of 5 tuples {Q, Σ, q, F, δ}.

Q : set of all states.

Σ : set of input symbols. (Symbols which machine takes as input)

q : Initial state. (Starting state of a machine)

F : set of final state.

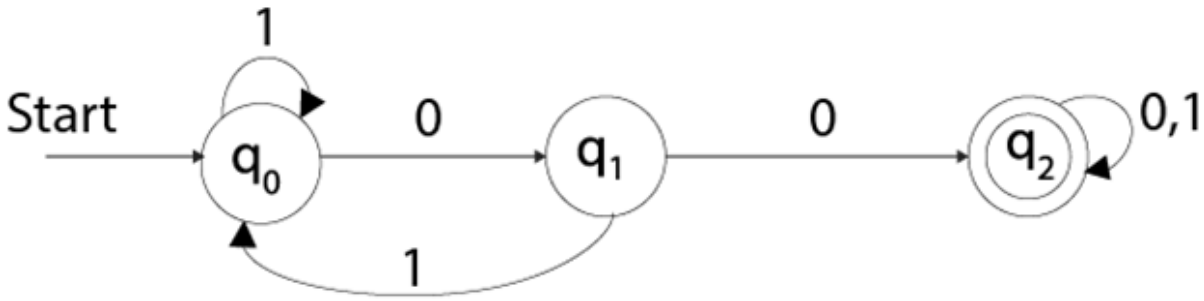
δ : Transition Function, defined as δ : Q X Σ --> Q.

In a DFA, for a particular input character, the machine goes to one state only. A transition function is defined on every state for every input symbol. Also in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.

For example, below DFA with Σ = {0, 1} accepts all strings ending with 0.

See an example of deterministic finite automata:

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q3}



One important thing to note is, *there can be many possible DFAs for a pattern*. A DFA with a minimum number of states is generally preferred.

2) Nondeterministic Finite Automata(NFA)

NFA is similar to DFA except following additional features:

1. Null (or ε) move is allowed i.e., it can move forward without reading symbols.
2. Ability to transmit to any number of states for a particular input.

However, these above features don't add any power to NFA. If we compare both in terms of power, both are equivalent.

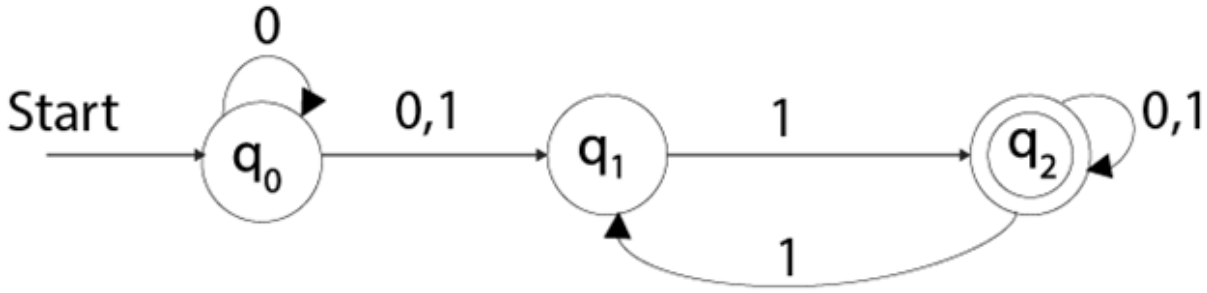
Due to the above additional features, NFA has a different transition function, the rest is the same as DFA.

δ: Transition Function

δ: Q X (Σ U ε) --> 2 ^ Q.

See an example of non deterministic finite automata:

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q3}



As you can see in the transition function is for any input including null (or ε), NFA can go to any state number of states.

One important thing to note is, *in NFA, if any path for an input string leads to a final state, then the input string* is *accepted*. For example, in the above NFA, there are multiple paths for the input string "00". Since one of the paths leads to a final state, "00" is accepted by the above NFA.

Some Important Points:

- Justification:

Since all the tuples in DFA and NFA are the same except for one of the tuples, which is Transition Function (δ)

In case of DFA

$\delta : Q \times \Sigma \rightarrow Q$

In case of NFA

$\delta : Q \times \Sigma \rightarrow 2^Q$

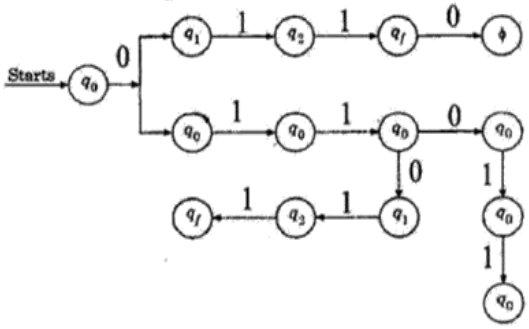
Now if you observe you'll find out $Q \times \Sigma \rightarrow Q$ is part of $Q \times \Sigma \rightarrow 2^Q$.

On the RHS side, Q is the subset of 2^Q which indicates Q is contained in 2^Q or Q is a part of 2^Q, however, the reverse isn't true. So mathematically, we can conclude that every DFA is NFA but not vice-versa. Yet there is a way to convert an NFA to DFA, so there exists an equivalent DFA for every NFA.

Both NFA and DFA have the same power and each NFA can be translated into a DFA.

1. There can be multiple final states in both DFA and NFA.
2. NFA is more of a theoretical concept.
3. DFA is used in Lexical Analysis in Compiler.

3. Transition sequence for the string "011011" is as follows :



One execution ends in hang state ϕ, second ends in non - final state q_0, and third ends in final state q_f hence string "011011" is accepted by third execution.

Difference between DFA and NFA

Strictly speaking the difference between DFA and NFA lies only in the definition of δ. Using this difference some more points can be derived and can be written as shown :

DFA	NFA
1. The DFA is 5 - tuple or quintuple $M=(Q,\Sigma,\delta,q_0,F)$ where Q is set of finite states Σ is set of input alphabets $\delta: Q\times\Sigma$ to Q q_0 is the initial state $F\subseteq Q$ is set of final states	The NFA is same as DFA except in the definition of δ. Here, δ is defined as follows : $\delta: Q\times(\Sigma\cup\in)$ to subset of 2^Q
2. There can be zero or one transition from a state on an input symbol	There can be zero, one or more transitions from a state on an input symbol
3. No $\in-$ transitions exist i.e., there should not be any transition or a transition if exist it should be on an input symbol	$\in-$ transitions can exist i. e., without any input there can be transition from one state to another state.
4. Difficult to construct	Easy to construct

Chomsky Hierarchy

Chomsky Hierarchy represents the class of languages that are accepted by the different machine. The category of language in Chomsky's Hierarchy is as given below:

1. Type 0 known as Unrestricted Grammar.
2. Type 1 known as Context Sensitive Grammar.
3. Type 2 known as Context Free Grammar.
4. Type 3 Regular Grammar.

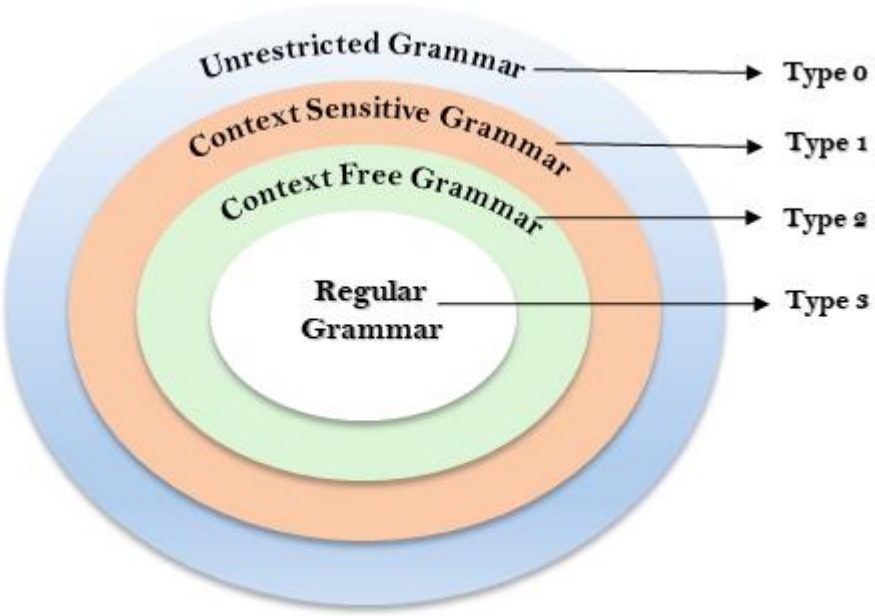


Fig: Chomsky Hierarchy

This is a hierarchy. Therefore every language of type 3 is also of type 2, 1 and 0. Similarly, every language of type 2 is also of type 1 and type 0, etc.

Type 0 Grammar:

Type 0 grammar is known as Unrestricted grammar. There is no restriction on the grammar rules of these types of languages. These languages can be efficiently modeled by Turing machines.

For example:

1. bAa → aa
2. S → s

Type 1 Grammar:

Type 1 grammar is known as Context Sensitive Grammar. The context sensitive grammar is used to represent context sensitive language. The context sensitive grammar follows the following rules:

- The context sensitive grammar may have more than one symbol on the left hand side of their production rules.
- The number of symbols on the left-hand side must not exceed the number of symbols on the right-hand side.
- The rule of the form A → ε is not allowed unless A is a start symbol. It does not occur on the right-hand side of any rule.
- The Type 1 grammar should be Type 0. In type 1, Production is in the form of V → T

Where the count of symbol in V is less than or equal to T.

For example:

1. S → AT
2. T → xy
3. A → a

Type 2 Grammar:

Type 2 Grammar is known as Context Free Grammar. Context free languages are the languages which can be represented by the context free grammar (CFG). Type 2 should be type 1. The production rule is of the form.

1. A → α

Where A is any single non-terminal and is any combination of terminals and non-terminals.

For example:

1. A → aBb
2. A → b
3. B → a

Type 3 Grammar:

Type 3 Grammar is known as Regular Grammar. Regular languages are those languages which can be described using regular expressions. These languages can be modeled by NFA or DFA. Type 3 is most restricted form of grammar. The Type 3 grammar should be Type 2 and Type 1. Type 3 should be in the form of V → T*V / T*

The NFA accepts strings a, ab, abbb etc. by using $\in$ path between q_1 and q_2 we can move from q_1 state to q_2 without reading any input symbol. To accept ab first we are moving from q_0 to q_1 reading a and we can jump to q_2 state without reading any symbol there we accept b and we are ending with final state so it is accepted.

Equivalence of NFA with $\in$– Transitions and NFA without $\in$– Transitions

Theorem : If the language L is accepted by an NFA with $\in$– transitions, then the language L_1 is accepted by an NFA without $\in$– transitions.

Proof : Consider an NFA 'N' with $\in$– transitions where $N=(Q, \Sigma, \delta, q_0, F)$

Construct an NFA N_1 without $\in$– transitions $N_1=(Q_1, \Sigma, \delta_1, q_0, F_1)$

where $Q_1 = Q$ and

$$F_1 = \begin{cases} F \cup \{q_0\} & \text{if } \in\text{– closure}(q_0) \text{ contains a state of } F \\ F & \text{otherwise} \end{cases}$$

and $\delta_1(q,a)$ is $\hat{\delta}(q,a)$ for q in Q and a in Σ.

Consider a non - empty string ω. To show by induction $|\omega|$ that $\delta_1(q_0, \omega) = \hat{\delta}(q_0,\omega)$

For $\omega = \in$, the above statement is not true. Because

$$\delta_1(q_0, \in) = \{q_0\},$$

while $$\hat{\delta}(q_0, \in) = \in\text{–closure}\ (q_0)$$

Basis :

Start induction with string length one.

i. e., $|\omega| = 1$

Then w is a symbol a, and $\delta_1(q_0,a) = \hat{\delta}(q_0,a)$ by definition of δ_1.

Induction : $|\omega| > 1$

Let $\omega = xy$ for symbol a in Σ.

Then $$\delta_1(q_0, xy) = \delta_1(\delta_1(q_0,x), y)$$

NFA TO DFA CONVERSION

Problem Statement

Let X = $(Q_x, \sum, \delta_x, q_0, F_x)$ be an NDFA which accepts the language L(X). We have to design an equivalent DFA Y = $(Q_y, \sum, \delta_y, q_0, F_y)$ 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 $\{Q_0, Q_1, \ldots Q_n\}$ 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

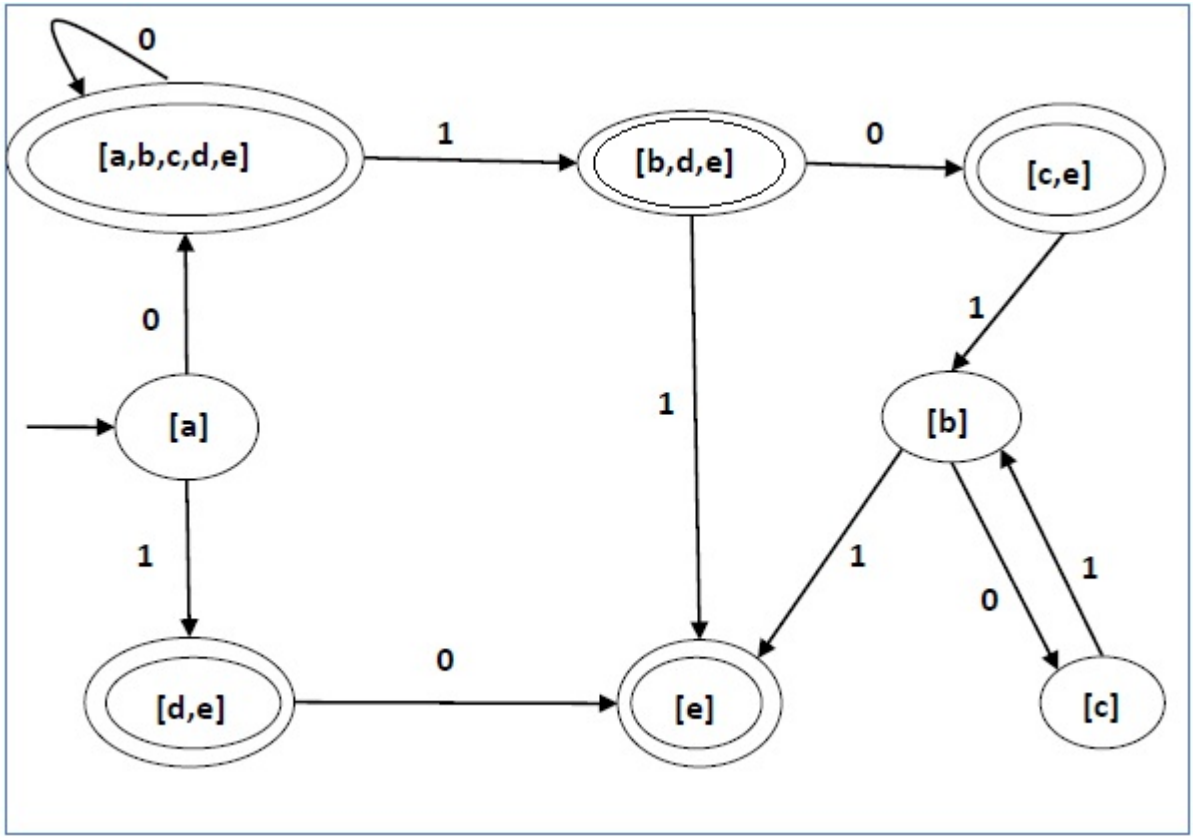
Let us consider the NDFA shown in the figure below.

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 −



REGULAR EXPRESSIONS:

Regular Expression can be recursively defined as follows −

- ε is a Regular Expression indicates the language containing an empty string. (L (ε) = {ε})
- φ is a Regular Expression denoting an empty language. (L (φ) = { })
- x is a Regular Expression where L = {x}
- If X is a Regular Expression denoting the language L(X) and Y is a Regular Expression denoting the language L(Y), then
    - X + Y is a Regular Expression corresponding to the language L(X) ∪ L(Y) where L(X+Y) = L(X) ∪ L(Y).
    - X.Y is a Regular Expression corresponding to the language L(X) . L(Y) where L (X.Y) = L(X) . L(Y)
    - R* is a Regular Expression corresponding to the language L(R*)where L(R*) = (L(R))*
- If we apply any of the rules several times from 1 to 5, they are Regular Expressions.

Some RE Examples

Regular Expressions	Regular Set

(0 + 10*)	L = { 0, 1, 10, 100, 1000, 10000, … }
(0*10*)	L = {1, 01, 10, 010, 0010, …}
(0 + ε)(1 + ε)	L = {ε, 0, 1, 01}
(a+b)*	Set of strings of a's and b's of any length including the null string. So L = { ε, a, b, aa , ab , bb , ba, aaa…….}
(a+b)*abb	Set of strings of a's and b's ending with the string abb. So L = {abb, aabb, babb, aaabb, ababb, ……..}
(11)*	Set consisting of even number of 1's including empty string, So L= {ε, 11, 1111, 111111, …….}
(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's , so L = {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, ……..}
(aa + ab + ba + bb)*	String of a's and b's of even length can be obtained by concatenating any combination of the strings aa, ab, ba and bb including null, so L = {aa, ab, ba, bb, aaab, aaba, ……..}

Grammar

A grammar G can be formally written as a 4-tuple (N, T, S, P) where −

- N or V_N is a set of variables or non-terminal symbols.
- T or ∑ is a set of Terminal symbols.
- S is a special variable called the Start symbol, S ∈ N
- P is Production rules for Terminals and Non-terminals. A production rule has the form α → β, where α and β are strings on V_N ∪ ∑ and least one symbol of α belongs to V_N.

Example

Grammar G1 −

({S, A, B}, {a, b}, S, {S → AB, A → a, B → b})

Here,

- S, A, and B are Non-terminal symbols;
- a and b are Terminal symbols
- S is the Start symbol, S ∈ N
- Productions, P : S → AB, A → a, B → b

REGULAR SETS

Any set that represents the value of the Regular Expression is called a Regular Set.

Properties of Regular Sets

Property 1. *The union of two regular set is regular.*

Proof −

Let us take two regular expressions

RE_1 = a(aa)* and RE_2 = (aa)*

So, L_1 = {a, aaa, aaaaa,.....}(Strings of odd length excluding Null)

and L_2 ={ ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 \cup L_2$ = { ε, a, aa, aaa, aaaa, aaaaa, aaaaaa,.....}

(Strings of all possible lengths including Null)

RE ($L_1 \cup L_2$) = a* (which is a regular expression itself)

Hence, proved.

Property 2. *The intersection of two regular set is regular.*

Proof −

Let us take two regular expressions

RE_1 = a(a*) and RE_2 = (aa)*

So, L_1 = { a,aa, aaa, aaaa ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 \cap L_2$ = { aa, aaaa, aaaaaa,.....} (Strings of even length excluding Null)

RE ($L_1 \cap L_2$) = aa(aa)* which is a regular expression itself.

Hence, proved.

Property 3. *The complement of a regular set is regular.*

Proof −

Let us take a regular expression −

RE = (aa)*

So, L = {ε, aa, aaaa, aaaaaa, ...} (Strings of even length including Null)

Complement of L is all the strings that is not in L.

So, L' = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

RE (L') = a(aa)* which is a regular expression itself.

Hence, proved.

Property 4. *The difference of two regular set is regular.*

Proof −

Let us take two regular expressions −

RE_1 = a (a*) and RE_2 = (aa)*

So, L_1 = {a, aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 – L_2$ = {a, aaa, aaaaa, aaaaaaa, ...}

(Strings of all odd lengths excluding Null)

RE ($L_1 – L_2$) = a (aa)* which is a regular expression.

Hence, proved.

Property 5. *The reversal of a regular set is regular.*

Proof −

We have to prove L^R is also regular if L is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

L^R = {10, 01, 11, 01}

RE (L^R) = 01 + 10 + 11 + 10 which is regular

Hence, proved.

Property 6. *The closure of a regular set is regular.*

Proof −

If L = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)*

L* = {a, aa, aaa, aaaa , aaaaa,……} (Strings of all lengths excluding Null)

RE (L*) = a (a)*

Hence, proved.

Property 7. *The concatenation of two regular sets is regular.*

Proof −

Let RE_1 = (0+1)*0 and RE_2 = 01(0+1)*

Here, L_1 = {0, 00, 10, 000, 010, ...} (Set of strings ending in 0)

and L_2 = {01, 010,011 ...} (Set of strings beginning with 01)

Then, L_1 L_2 = {001,0010,0011,0001,00010,00011,1001,10010,.....}

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)*001(0 + 1)*

Hence, proved.

Closure Properties of Regular Languages

Union : If L1 and If L2 are two regular languages, their union L1 ∪ L2 will also be regular. For example, L1 = $\{a^n \mid n \geq 0\}$ and L2 = $\{b^n \mid n \geq 0\}$

L3 = L1 ∪ L2 = $\{a^n \cup b^n \mid n \geq 0\}$ is also regular.

Intersection : If L1 and If L2 are two regular languages, their intersection L1 ∩ L2 will also be regular. For example,

L1= $\{a^m b^n \mid n \geq 0$ and $m \geq 0\}$ and L2= $\{a^m b^n \cup b^n a^m \mid n \geq 0$ and $m \geq 0\}$

L3 = L1 ∩ L2 = $\{a^m b^n \mid n \geq 0$ and $m \geq 0\}$ is also regular.

Concatenation : If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular. For example,

L1 = $\{a^n \mid n \geq 0\}$ and L2 = $\{b^n \mid n \geq 0\}$

L3 = L1.L2 = $\{a^m . b^n \mid m \geq 0$ and $n \geq 0\}$ is also regular.

Kleene Closure : If L1 is a regular language, its Kleene closure L1* will also be regular. For example,

L1 = (a ∪ b)

L1* = (a ∪ b)*

Complement : If L(G) is regular language, its complement L'(G) will also be regular. Complement of a language can be found by subtracting strings which are in L(G) from all possible strings. For example,

L(G) = $\{a^n \mid n > 3\}$

L'(G) = $\{a^n \mid n <= 3$

<u>Pumping Lemma for Regular Sets</u>

Theorem

Let L be a regular language. Then there exists a constant 'c' such that for every string w in L −

$|w| \geq c$

We can break w into three strings, w = xyz, such that −

- $|y| > 0$
- $|xy| \leq c$
- For all $k \geq 0$, the string xy^kz is also in L.

<u>Applications of Pumping Lemma</u>

Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular.

- If L is regular, it satisfies Pumping Lemma.

- If L does not satisfy Pumping Lemma, it is non-regular.

Method to prove that a language L is not regular

- At first, we have to assume that L is regular.
- So, the pumping lemma should hold for L.
- Use the pumping lemma to obtain a contradiction −
    - Select w such that $|w| \geq c$
    - Select y such that $|y| \geq 1$
    - Select x such that $|xy| \leq c$
    - Assign the remaining string to z.
    - Select k such that the resulting string is not in L.

Hence L is not regular.

Problem

Prove that $L = \{a^i b^i \mid i \geq 0\}$ is not regular.

Solution −

- At first, we assume that L is regular and n is the number of states.
- Let $w = a^n b^n$. Thus $|w| = 2n \geq n$.
- By pumping lemma, let w = xyz, where $|xy| \leq n$.
- Let $x = a^p$, $y = a^q$, and $z = a^r b^n$, where $p + q + r = n$, $p \neq 0$, $q \neq 0$, $r \neq 0$. Thus $|y| \neq 0$.
- Let k = 2. Then $xy^2z = a^p a^{2q} a^r b^n$.
- Number of as = $(p + 2q + r) = (p + q + r) + q = n + q$
- Hence, $xy^2z = a^{n+q} b^n$. Since $q \neq 0$, xy^2z is not of the form $a^n b^n$.
- Thus, xy^2z is not in L. Hence L is not regular.

<u>Decision algorithms for regular sets:</u>

Remember: An algorithm must always terminate to be called an algorithm!

Basically, an algorithm needs to have four properties

1) It must be written as a finite number of unambiguous steps

2) For every possible input, only a finite number of steps
will be performed, and the algorithm will produce a result

3) The same, correct, result will be produced for the same input

4) Each step must have properties 1) 2) and 3)

Remember: A regular language is just a set of strings over a finite alphabet.

Every regular set can be represented by an regular expression
and by a minimized finite automata, a DFA.

We choose to use DFA's, represented by the usual M=(Q, Sigma, delta, q0, F) There are countably many DFA's yet every DFA we look at has a finite description. We write down the set of states Q, the alphabet Sigma, the transition table delta, the initial state q0 and the final states F. Thus we can analyze every DFA and even simulate them.

Theorem: The regular set accepted by DFA's with n states:

1) the set is non empty if and only if the DFA accepts at least one string
of length less than n. Just try less than |Sigma|^|Q| strings (finite)

2) the set is infinite if and only if the DFA accepts at least one string
Of length k, n <= k < 2n. By pumping lemma, just more to try (finite)

Rather obviously the algorithm proceeds by trying the null string first.
The null string is either accepted or rejected in a finite time.
Then try all strings of length 1, i.e. each character in Sigma.
Then try all strings of length 2, 3, ... , k. Every try results in
an accept or reject in a finite time.
Thus we say there is an algorithm to decide if any regular set
represented by a DFA is a) empty, b) finite and c) infinite

MINIMIZATION OF FINITE AUTOMATA

Minimization of DFA

Minimization of DFA means reducing the number of states from given FA. Thus, we get the FSM(finite state machine) with redundant states after minimizing the FSM.

We have to follow the various steps to minimize the DFA. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1. δ (q, a) = p
2. . δ (r, a) = p

That means, find the two states which have the same value of a and b and remove one of them.

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Example:

Solution:

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states.

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q5	q5

*q5	q5	q5

Step 3: Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

State	0	1
q0	q1	q3
q1	q0	q3

2. Another set contains those rows, which starts from final states.

State	0	1
q3	q5	q5
q5	q5	q5

Step 4: Set 1 has no similar rows so set 1 will be the same.

Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

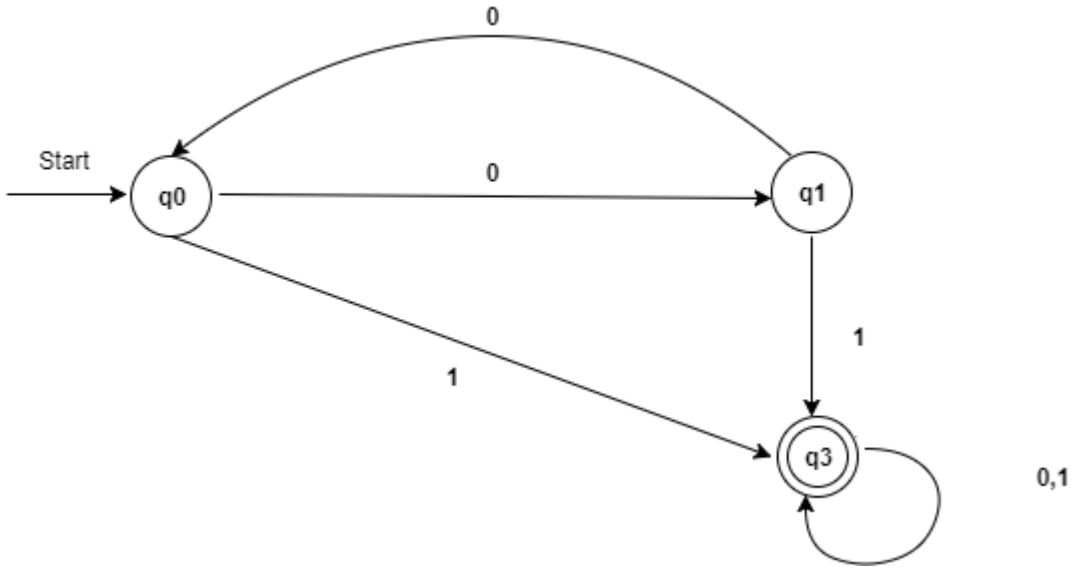
State	0	1
q3	q3	q3

Step 6: Now combine set 1 and set 2 as:

State	0	1
→q0	q1	q3

q1	q0	q3
*q3	q3	q3

Now it is the transition table of minimized DFA.



UNIT-II

Context Free Grammars and Languages:

Context free grammar and Languages:

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

Where,

G describes the grammar

T describes a finite set of terminal symbols.

V describes a finite set of non-terminal symbols

P describes a set of production rules

S is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba

4. S ⇒ abbcbba

   By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

Derivation Tree

Derivation tree is a graphical representation for the derivation of the given production rules for a given CFG. It is the simple way to show how the derivation can be done to obtain some string from a given set of production rules. The derivation tree is also called a parse tree.

Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

A parse tree contains the following properties:

1. The root node is always a node indicating start symbols.
2. The derivation is read from left to right.
3. The leaf node is always terminal nodes.
4. The interior nodes are always the non-terminal nodes.

Example 1:

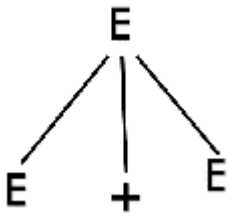
Production rules:

1. E = E + E
2. E = E * E
3. E = a | b | c

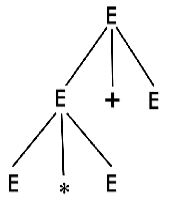
Input

1. a * b + c

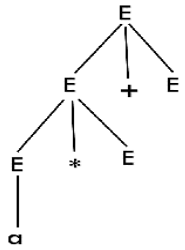
Step 1:



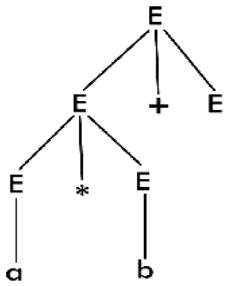
Step 2:



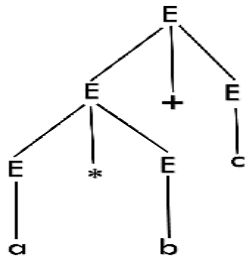
Step 3:



Step 4:



Step 5:



Simplification of CFG

As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammars are not always optimized that means the grammar may consist of some extra symbols (non-terminal). Having extra symbols, unnecessary increase

the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The properties of reduced grammar are given below:

1. Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
2. There should not be any production as X → Y where X and Y are non-terminal.
3. If ε is not in the language L then there need not to be the production X → ε.

Let us study the reduction process in detail. / p>

Removal of Useless Symbols

A symbol can be useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string. That symbol is known as a useless symbol. Similarly, a variable can be useless if it does not take part in the derivation of any string. That variable is known as a useless variable.

For Example:

1. T → aaB | abA | aaT
2. A → aA
3. B → ab | b
4. C → ad

In the above example, the variable 'C' will never occur in the derivation of any string, so the production C → ad is useless. So we will eliminate it, and the other productions are written in such a way that variable C can never reach from the starting variable 'T'.

Production A → aA is also useless because there is no way to terminate it. If it never terminates, then it can never produce a string. Hence this production can never take part in any derivation.

To remove this useless production A → aA, we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'B' occurs.

Elimination of ε Production

The productions of type S → ε are called ε productions. These type of productions can only be removed from those grammars that do not generate ε.

Step 1: First find out all null able non-terminal variable which derives ε.

Step 2: For each production A → a, construct all production A → x, where x is obtained from a by removing one or more non-terminal from step 1.

Step 3: Now combine the result of step 2 with the original production and remove ε productions.

Example:

Remove the production from the following CFG by preserving the meaning of it.

1. S → XYX
2. X → 0X | ε
3. Y → 1Y | ε

Solution:

Now, while removing ε production, we are deleting the rule X → ε and Y → ε. To preserve the meaning of CFG we are actually placing ε at the right-hand side whenever X and Y have appeared.

Let us take

1. S → XYX

If the first X at right-hand side is ε. Then

1. S → YX

Similarly if the last X in R.H.S. = ε. Then

1. S → XY

If Y = ε then

1. S → XX

If Y and X are ε then,

1. S → X

If both X are replaced by ε

1. S → Y

   Now,

1. S → XY | YX | XX | X | Y

   Now let us consider

1. X → 0X

   If we place ε at right-hand side for X then,

1. X → 0
2. X → 0X | 0

   Similarly Y → 1Y | 1

   Collectively we can rewrite the CFG with removed ε production as

1. S → XY | YX | XX | X | Y
2. X → 0X | 0
3. Y → 1Y | 1

Removing Unit Productions

The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

Step 1: To remove X → Y, add production X → a to the grammar rule whenever Y → a occurs in the grammar.

Step 2: Now delete X → Y from the grammar.

Step 3: Repeat step 1 and step 2 until all unit productions are removed.

For example:

1. S → 0A | 1B | C
2. A → 0S | 00
3. B → 1 | A
4. C → 01

Solution:

S → C is a unit production. But while removing S → C we have to consider what C gives. So, we can add a rule to S.

1. S → 0A | 1B | 01

Similarly, B → A is also a unit production so we can modify it as

1. B → 1 | 0S | 00

Thus finally we can write CFG without unit production as

1. S → 0A | 1B | 01
2. A → 0S | 00
3. B → 1 | 0S | 00
4. C → 01

Chomsky's Normal Form (CNF)

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal. For example, S → a.

For example:

1. G1 = {S → AB, S → c, A → a, B → b}
2. G2 = {S → aA, A → a, B → c}

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as S → aZ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

1. S1 → S

Where S1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:

1. S → RA
2. R → a

Step 4: Eliminate RHS with more than two non-terminals. For example, S → ASB can be decomposed as:

1. S → RS
2. R → AS

Example:

Convert the given CFG to CNF. Consider the given grammar G1:

1. S → a | aA | B
2. A → aBB | ε
3. B → Aa | b

Solution:

Step 1: We will create a new production S1 → S, as the start symbol S appears on the RHS. The grammar will be:

1. S1 → S
2. S → a | aA | B
3. A → aBB | ε
4. B → Aa | b

Step 2: As grammar G1 contains A → ε null production, its removal from the grammar yields:

1. S1 → S
2. S → a | aA | B
3. A → aBB
4. B → Aa | b | a

Now, as grammar G1 contains Unit production S → B, its removal yield:

1. S1 → S
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Also remove the unit production S1 → S, its removal from the grammar yields:

1. S0 → a | aA | Aa | b
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Step 3: In the production rule S0 → aA | Aa, S → aA | Aa, A → aBB and B → Aa, terminal a exists on RHS with non-terminals. So we will replace terminal a with X:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → XBB
4. B → AX | b | a
5. X → a
6. **Step 4:** In the production rule A → XBB, RHS has more than two symbols, removing it from grammar yield:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → RB
4. B → AX | b | a
5. X → a
6. R → XB

Hence, for the given grammar, this is the required CNF.

Pumping Lemma for CFLs:

In any sufficiently long string in a CFL, it is possible to find at most two short, nearby substrings that we can "pump" i times in tandem, for any integer i, and the resulting string will still be in that language. Pumping lemma for CFLs: Let L be a CFL. Then there exists a constant n such that if z ∈ L with |z| ≥ n, then we can write z = uvwxy, subject to the following conditions: 1. |vwx| ≤ n. 2. vx 6= . 3. For all i ≥ 0, we have uviwxiy ∈ L.

If the string z is sufficiently long, then the parse tree produced by z has a variable symbol that is repeated on a path from the root to a leaf. Suppose Ai = Aj , such that the overall parse tree has

yield z = uvwxy, the subtree for root Aj has yield w, and the subtree for root Ai has yield vwx. We can replace the subtree for root Ai with the subtree for root Aj , giving a tree with yield uwy (corresponding to the case i = 0), which also belongs to L. We can replace the subtree for root Aj with the subtree for root Ai , giving a tree with yield uv2wx2y (corresponding to the case i = 2), which also belongs to L.

While CFLs can match two sub-strings for (in) equality of length, they cannot match three such sub-strings.

Example 1: Consider L = {0m1m2m | m ≥ 1}. Pick n of the pumping lemma. Pick z = 0n1 n2 n. Break z into uvwxy, with |vwx| ≤ n and vx 6= . Hence vwx cannot involve both 0s and 2s, since the last 0 and the first 2 are at least n + 1 positions apart. There are two cases: • vwx has no 2s. Then vx has only 0s and 1s. Then uwy, which would have to be in L, has n 2s, but fewer than n 0s or 1s. • vwx has no 0s. Analogous. Hence L is not a CFL

CFLs cannot match two pairs of sub-strings of equal lengths if the pairs interleave.

Example 2: Consider L = {0 i1 j2 i3 j | i, j ≥ 1}. Pick n of the pumping lemma. Pick z = 0n1 n2 n3 n. Break z into uvwxy, with |vwx| ≤ n and vx 6= . Then vwx contains one or two different symbols. In both cases, the string uwy cannot be in L. CFLs cannot match two sub-strings of arbitrary length over an alphabet of at least two symbols.

Example 3: Consider L = {ww | w ∈ {0, 1} ∗}. Pick n of the pumping lemma. Pick z = 0n1 n0 n1 n. In all cases, the string uwy cannot be in L.

CFL Closure Property

Context-free languages are **closed** under −

- Union
- Concatenation
- Kleene Star operation

Union

Let L_1 and L_2 be two context free languages. Then $L_1 \cup L_2$ is also context free.

Example

Let $L_1 = \{ a^n b^n , n > 0\}$. Corresponding grammar G_1 will have P: S1 → aAb|ab

Let $L_2 = \{ c^m d^m , m \geq 0\}$. Corresponding grammar G_2 will have P: S2 → cBb| ε

Union of L_1 and L_2, $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

The corresponding grammar G will have the additional production S → S1 | S2

Concatenation

If L_1 and L_2 are context free languages, then L_1L_2 is also context free.

Example

Union of the languages L_1 and L_2, $L = L_1L_2 = \{ a^n b^n c^m d^m \}$

The corresponding grammar G will have the additional production $S \rightarrow S1\ S2$

Kleene Star

If L is a context free language, then L* is also context free.

Example

Let $L = \{ a^n b^n , n \geq 0\}$. Corresponding grammar G will have P: $S \rightarrow aAb| \varepsilon$

Kleene Star $L_1 = \{ a^n b^n \}^*$

The corresponding grammar G_1 will have additional productions $S1 \rightarrow SS_1 | \varepsilon$

Context-free languages are **not closed** under −

- **Intersection** − If L1 and L2 are context free languages, then L1 ∩ L2 is not necessarily context free.
- **Intersection with Regular Language** − If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.
- **Complement** − If L1 is a context free language, then L1' may not be context free

Pushdown Automata Introduction

Basic Structure of PDA

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is −

"Finite state machine" + "a stack"

A pushdown automaton has three components −

- an input tape,
- a control unit, and
- a stack with infinite size.

The stack head scans the top symbol of the stack.

A stack does two operations −

- **Push** − a new symbol is added at the top.
- **Pop** − the top symbol is read and removed.

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.

A PDA can be formally described as a 7-tuple $(Q, \sum, S, \delta, q_0, I, F)$ −

- **Q** is the finite number of states
- ∑ is input alphabet
- **S** is stack symbols
- **δ** is the transition function: Q × (∑ ∪ {ε}) × S × Q × S*
- $\mathbf{q_0}$ is the initial state ($q_0 \in Q$)
- **I** is the initial stack top symbol (I ∈ S)
- **F** is a set of accepting states (F ∈ Q)

The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as ab → c −

This means at state $\mathbf{q_1}$, if we encounter an input string **'a'** and top symbol of the stack is **'b'**, then we pop **'b'**, push **'c'** on top of the stack and move to state $\mathbf{q_2}$.

Terminologies Related to PDA

Instantaneous Description

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where

- **q** is the state
- **w** is unconsumed input
- **s** is the stack contents

Turnstile Notation

The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol "⊢".

Consider a PDA (Q, ∑, S, δ, q_0, I, F). A transition can be mathematically represented by the following turnstile notation −

```
(p, aw, Tβ) ⊢ (q, w, αb)
```

This implies that while taking a transition from state **p** to state **q**, the input symbol **'a'** is consumed, and the top of the stack **'T'** is replaced by a new string **'α'**.

Note − If we want zero or more moves of a PDA, we have to use the symbol (⊢*) for it.

a grammar **G** is context-free, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the context-free grammar **G**. A parser can be built for the grammar **G**.

Also, if **P** is a pushdown automaton, an equivalent context-free grammar G can be constructed where

L(G) = L(P)

In the next two topics, we will discuss how to convert from PDA to CFG and vice versa.

Algorithm to find PDA corresponding to a given CFG

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F)

Step 1 − Convert the productions of the CFG into GNF.

Step 2 − The PDA will have only one state {q}.

Step 3 − The start symbol of CFG will be the start symbol in the PDA.

Step 4 − All non-terminals of the CFG will be the stack symbols of the PDA and all the terminals of the CFG will be the input symbols of the PDA.

Step 5 − For each production in the form **A → aX** where a is terminal and **A, X** are combination of terminal and non-terminals, make a transition **δ (q, a, A)**.

Problem

Construct a PDA from the following CFG.

G = ({S, X}, {a, b}, P, S)

where the productions are −

S → XS | ε , A → aXb | Ab | ab

Solution

Let the equivalent PDA,

P = ({q}, {a, b}, {a, b, X, S}, δ, q, S)

where δ −

δ (q, ε , S) = {(q, XS), (q, ε)}

δ (q, ε , X) = {(q, aXb), (q, Xb), (q, ab)}

δ(q, a, a) = {(q, ε)}

δ(q, 1, 1) = {(q, ε)}

Algorithm to find CFG corresponding to a given PDA

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F) such that the non- terminals of the grammar G will be $\{X_{wx} \mid w,x \in Q\}$ and the start state will be $A_{q0,F}$.

Step 1 − For every w, x, y, z ∈ Q, m ∈ S and a, b ∈ ∑, if δ (w, a, ε) contains (y, m) and (z, b, m) contains (x, ε), add the production rule $X_{wx} \rightarrow a\ X_{yz} b$ in grammar G.

Step 2 − For every w, x, y, z ∈ Q, add the production rule $X_{wx} \rightarrow X_{wy} X_{yx}$ in grammar G.

Step 3 − For w ∈ Q, add the production rule $X_{ww} \rightarrow \varepsilon$ in grammar G.

Parsing is used to derive a string using the production rules of a grammar. It is used to check the acceptability of a string. Compiler is used to check whether or not a string is syntactically correct. A parser takes the inputs and builds a parse tree.

A parser can be of two types −

- **Top-Down Parser** − Top-down parsing starts from the top with the start-symbol and derives a string using a parse tree.
- **Bottom-Up Parser** − Bottom-up parsing starts from the bottom with the string and comes to the start symbol using a parse tree.

Design of Top-Down Parser

For top-down parsing, a PDA has the following four types of transitions −

- Pop the non-terminal on the left hand side of the production at the top of the stack and push its right-hand side string.
- If the top symbol of the stack matches with the input symbol being read, pop it.
- Push the start symbol 'S' into the stack.
- If the input string is fully read and the stack is empty, go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q_0, I, F), then the top-down parsing is −

(x+y*z, I) ⊢(x +y*z, SI) ⊢ (x+y*z, S+XI) ⊢(x+y*z, X+XI)

⊢(x+y*z, Y+X I) ⊢(x+y*z, x+XI) ⊢(+y*z, +XI) ⊢ (y*z, XI)

⊢(y*z, X*YI) ⊢(y*z, y*YI) ⊢(*z,*YI) ⊢(z, YI) ⊢(z, zI) ⊢(ε, I)

Design of a Bottom-Up Parser

For bottom-up parsing, a PDA has the following four types of transitions −

- Push the current input symbol into the stack.
- Replace the right-hand side of a production at the top of the stack with its left-hand side.
- If the top of the stack element matches with the current input symbol, pop it.
- If the input string is fully read and only if the start symbol 'S' remains in the stack, pop it and go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q_0, I, F), then the bottom-up parsing is −

(x+y*z, I) ⊢ (+y*z, xI) ⊢ (+y*z, YI) ⊢ (+y*z, XI) ⊢ (+y*z, SI)

⊢(y*z, +SI) ⊢ (*z, y+SI) ⊢ (*z, Y+SI) ⊢ (*z, X+SI) ⊢ (z, *X+SI)

⊢ (ε, z*X+SI) ⊢ (ε, Y*X+SI) ⊢ (ε, X+SI) ⊢ (ε, SI)

UNIT-III

Turing Machines

Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing.

Definition

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A TM can be formally described as a 7-tuple (Q, X, ∑, δ, q_0, B, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- ∑ is the input alphabet
- **δ** is a transition function; δ : Q × X → Q × X × {Left_shift, Right_shift}.
- **q_0** is the initial state
- **B** is the blank symbol
- **F** is the set of final states

Example of Turing machine

Turing machine M = (Q, X, ∑, δ, q_0, B, F) with

- Q = {q_0, q_1, q_2, q_f}
- X = {a, b}
- ∑ = {1}
- q_0 = {q_0}
- B = blank symbol
- F = {q_f }

δ is given by –

Tape alphabet symbol	Present State 'q_0'	Present State 'q_1'	Present State 'q_2'
a	$1Rq_1$	$1Lq_0$	$1Lq_f$
b	$1Lq_2$	$1Rq_1$	$1Rq_f$

Here the transition $1Rq_1$ implies that the write symbol is 1, the tape moves right, and the next state is q_1. Similarly, the transition $1Lq_2$ implies that the write symbol is 1, the tape moves left, and the next state is q_2.

Time and Space Complexity of a Turing Machine

For a Turing machine, the time complexity refers to the measure of the number of times the tape moves when the machine is initialized for some input symbols and the space complexity is the number of cells of the tape written.

Time complexity all reasonable functions −

T(n) = O(n log n)

TM's space complexity −

S(n) = O(n)

Construct Turing machine for L = {an bm a(n+m) - n,m≥1} in C++

Turing Machine − A Turing machine is a device used to accept words of a language generated by type 0 grammars. A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A TM can be formally described as a 7-tuple (Q, X, Σ, δ, q0, B, F) where −

- Q is a finite set of states
- X is the tape alphabet
- Σ is the input alphabet
- δ is a transition function; δ : Q × X → Q × X × {Left_shift, Right_shift}.
- q0 is the initial state
- B is the blank symbol
- F is the set of final states

Our goal is to construct a Turing machine TM which accepts the language

L= anbma(n+m) where n,m>=1

Let us take examples of words that TM can accept,

- abaa, n=1,m=1
- aabaaa, n=2,m=1
- abbaaa, n=1,m=2
- aaabaaaa, n=3,m=1

That is n times a followed by m times b followed by n+m times a again. n,m>=1

The least no. of a' s will always be 3 and and b be 1. When both n=m=1.

The approach is summarized below −

The machine first accepts all n no. of a's followed by all m no. of b's. Then as more a's are encountered, it starts deleting previous input b's and a's. In the end when no new a's are coming and the head reaches to the first input character, it means all characters are processed correctly. Let us follow step by step for input string −

Transitions from state q0

- δ (q0, a) → (q1, x, R). At state q0 if character read is a then transit to state q1, make it x and move right to point the next character in the string.

  Ex − aabaaa → xabaaa (first character became x head moved right to next a)

- δ (q0, b) → (q3, x, R). At state q0 if character read is a then transit to state q3, make it x and move right to point the next character in string.

  Ex − babaaa…. → xabaaa (first character became x head moved right to next a)

Here x is used to represent the first character.

Transitions from state q1

- δ (q1, a) → (q1, a, R). At state q1 if character read is a then remain at state q1, and move right to point the next character in the string.

  Ex: xaabaaa… → xaabaaa… (for rest a's do nothing and move right)

- δ (q1, b) → (q2, b, R). At state q1 if character read is b then remain at state q1, and move right to point the next character in string.

  Ex − xaabaaa… → xaabaaa… (for rest b's do nothing and move right)

Transitions from state q2

- δ (q2, b) → (q2, b, R). At state q2 if character read is b then remain at state q2, and move right to point the next character in the string.

  Ex − xaabbbaaa… → xaabbbaaa… (for rest b's do nothing and move right)

- δ (q2, z) → (q2, z, R). At state q2 if character read is z then remain at state q2, and move right to point the next character in string.

  Ex − xaabaazz… → xaabaazz… (for rest z's do nothing and move right)

- δ (q2, a) → (q3, z, L). At state q2 if character read is a, make it z then transit to state q3, and move left to point the previous character in string.

Ex − xaabaazz… → xaabazzz… (for a's replace with z and move left)

Transitions from state q3

- δ (q3, z) → (q3, z, L). At state q3 if character read is z then remain at state q3, and move left to point the previous character in the string.

  Ex − xaabzzzz… → xaabzzzzz… (for z's do nothing and move left)

- δ (q3, b) → (q2, z, R). At state q2 if character read is b then make it z and transit at state q2, and move right to point the next character in string. Replace all b's

  Ex − xaabzzzz… → xaazzzzz… (for b's replace with z and move right)

- δ (q3, a) → (q2, z, R). At state q2 if character read is a then make it z and transit to state q3, and move right to point the next character in string. Replace all a's

  Ex − xaazzzz… → xaazzzzz… (for a's replace with z and move right)

- δ (q3, x) → (q4, z, R). At state q2 if character read is x make it z then transit to state q3, and move right to point the next character in string. The first symbol is reached.

  Ex − xzzzzzzz… → zzzzzzzz… (for x replace with z and move right)

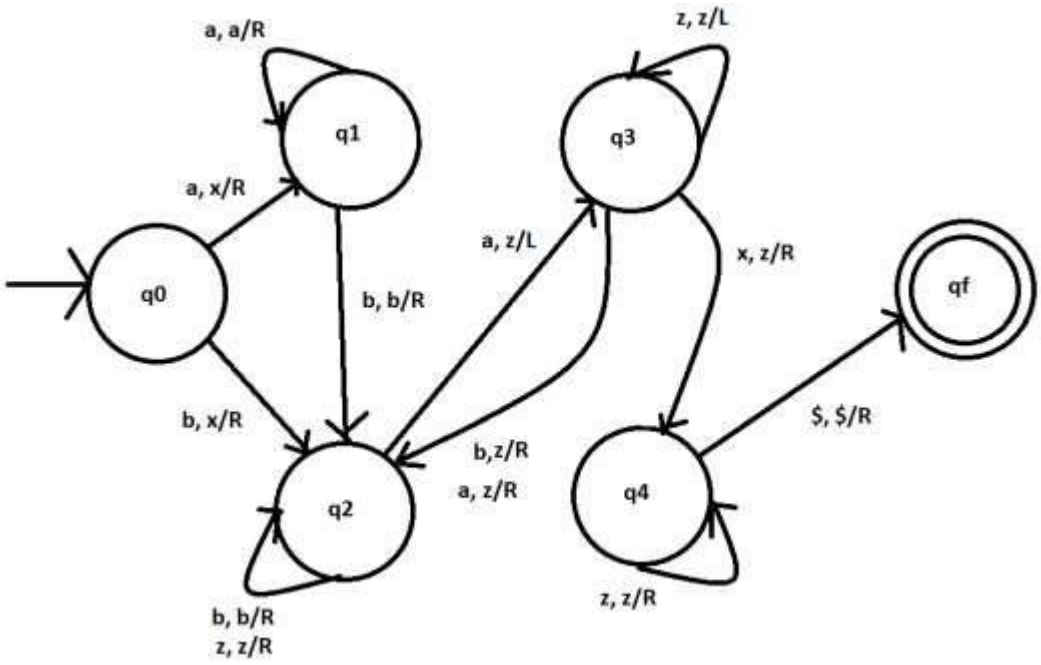
Transitions from state q4

- δ (q4, z) → (q4 z, R). At state q4 if character read is z then remain at state q4, and move right to point the next character in the string. All characters are z now.

  Ex − zzzzzzzz… → zzzzzzzz… (for all z's do nothing and move right)

- δ (q4,)→(qf,)→(qf, , R). At state q4 if no characters remain, end reached. Transit to final state qf. Which means string is accepted.
  - Ex − zzzzzzzz→ zzzzzzzz (forendof string,→zzzzzzzz(forendofstring, do nothing and move to final state)

Diagram shows the turing machine −



Input

aabaaa

q0: aabaaa → q1: xabaaa → q1: xabaaa → q2: xabaaa → q3: xabzaa → q2: xazzaa

q2: xazzaa → q3: xazzza → q3: xazzza → q3: xazzza → q2: xzzzzza → q2: xzzzzza

q2: xzzzzza → q2: xzzzzza → q2: xzzzzza → q2: xzzzzzz → q3: xzzzzzz……..

q3: xzzzzzz → q3: xzzzzzz → q4: zzzzzzz → q4: zzzzzzz…….q4: xzzzzzz$ → qf: xzzz

Combining Turing Machines

• Our examples so far have been somewhat unimpressive. But Turing machines are of ultimate generality: we can design Turing machines for every computable computational problem.

• To make life easier, we will now show how to combine simple Turing machines into more complex ones.

• We'll develop a graphical notation for these complex Turing machines, so that we don't get bogged down in details of transition functions.

The Basic Turing Machines

• Symbol-writing machines: For each symbol in Σ, we can build a machine that writes that symbol and halts. E.g. for a: δ a b . . . q0 a, h a, h . . . a, h Call this machine Wa (and, similarly, Wb, Wc, . . . , W)

• Head-moving machines: We can build a machine that moves one cell left or right and halts. E.g. for left: δ a b . . . q0 L, h L, h . . . L, h Call this machine ML (and, similarly, MR)

Rules for Combining Machines •

If TM1 and TM2 are Turing machines, we can create a Turing machine which will first behave like TM1 and then like TM2.

• How? 1. Change all state names in TM2 so they don't clash with state names in TM1. 2. Change all halts in TM1's transition table to the new name of the start state of TM2. 3. Append TM2's transition table to the foot of TM1's transition table.

• E.g. For Σ = {a, b, } , let's combine Wa (a machine for writing an a) with ML (a machine that moves its head one cell to the left). δ a b q0 a, q1 a, q1 a, q1 q1 L, h L, h L, h

• In general, if TM1 and TM2 are combined in this way, we will write TM1 −→ TM2 So this machine starts off in the initial state of TM1, operates as per TM1 until TM1 would halt, then it launches TM2 and operates as TM2, until TM2 would halt.

• We will also write > to highlight the start of this combined machine.

• E.g. > Wa −→ ML • E.g. > Wa −→ MR −→ Wb −→ MR −→ Wb −→ MR −→ Wa

• The connection between two Turing machines may depend on the symbol that is under the read/write head at the point when the first machine halts. Example Turing Machines Combining Turing . . . Module Home Page Title Page JJ II J I Page 5 of 9 Back Full Screen Close Quit • We will depict this with a test alongside the arrow: TM1 test −→ TM2

• E.g. ML =a −→ W This machine first moves left. Then, if there is an a under the read/write head, it overwrites it with a blank and then halts. If there had been any other symbol under the read/write head after moving left, it would have halted immediately.

• E.g. ML ∈ {a,b} −→ W • E.g. ML 6=a −→ W • E.g. ML 6∈ {a,b} −→ W

• Multiple arrows are allowed, provided their tests are mutually exclusive.

• E.g.: = a = b > ML Wb Wa This machine first moves left. Then, if there is an a under the head, it writes a b and halts; if there is a b under the head, it writes an a and halts. If there were something else under the head, it would halt immediately after moving left.

• How is the transition table for this machine built? – Rename the states in Wb and Wa to avoid clashes. Example Turing Machines Combining Turing . . . Module Home Page Title Page JJ II J I Page 6 of 9 Back Full Screen Close Quit – Change halts in ML. Any halts in the a column are changed to the renamed start state of Wb. Any halts in the b column are changed to the renamed start state of Wa. – Append the tables together. δ a b q0 L, h/q1 L, h/q2 L, h q/ q0 1 b, h b, h b, h q/ q0 2 a, h a, h a, h

• Loops are allowed • E.g.: TM E.g. >ML test ε{a, b, c} TM is executed. When it would halt, if the test is true, it returns to state q0 instead. In the example, the machine moves left repeatedly, for as long as there is an a, b or c under the read/write head. When the symbol under the read/write head is not one of a, b or c, it halts. It is usual to include a test, otherwise you have an infinite loop.

h - halt in a final state 10. a;b !g w! If the current symbol is a or b, let w represent the current symbol. Example Assume input string w 2 + ,= fa; bg. If jwj is odd, then write a b at the end of the string. The tape head should pointing at the leftmost symbol of w. input: bab, output: babb input: ba, output: ba sR R b L B B B B R h B What is the running time? Example Assume input string w 2 + ,= fa; bg, jwj > 0 For each a in the string, append a b to the end of the string. input: abbabb, output: abbabbbb The tape head should finish pointing at the leftmost symbol of w. Turing's Thesis Any computation that can be carried out by a mechanical means can be performed by a TM. Denition: An algorithm for a function f:D!R is a TM M, which given input d2D, halts with answer f(d)2R. 3 Example: f (x + y) = x + y, x and y unary numbers. start with: 111+1111 " end with: 1111111 " Example: Copy a String, f(w)=w0w, w2 ,= fa; b; cg Denoted by C start with: abac " end with: abac0abac " Algorithm: Write a 0 at end of string For each symbol in string { make a copy of the symbol s R 0L B B R a,b,c w BR wL w B B 0 LRh B 4 Example: Shift the string that is to the left of the ape head to the right, denoted by SR (shift right) Below, \ba" is to the left of the tape head, so shift \ba" to the right. start with: aaBbabca " end with: aaBBbaca " Algorithm: remember symbol to the right and erase it for each symbol to the left do { shift the symbol one cell to the right replace symbol erased move tape head to appropriate position a,b,c w LBL BR wL R vLh 0 s R v 0 a,b,c,B B Example: Shift the string that is to the right of tape head to the left,

denote by SL (shift left) start with: babcaBba " end with: bacaBBba " (similar to SR) 5 a,b,c w B B w v h 0 s v 0 a,b,c,B L R R LR L R B Example: Add unary numbers This time use shift. Example: Multiply two unary numbers, f(xy)=xy, x and y unary numbers. Assume x,y>0. start with: 111111 " end with: 11111111

Universal Turing Machines and Undecidability:

The Turing Machine (TM) is the machine level equivalent to a digital computer.

It was suggested by the mathematician Turing in the year 1930 and has become the most widely used model of computation in computability and complexity theory.

The model consists of an input and output. The input is given in binary format form on to the machine's tape and the output consists of the contents of the tape when the machine halts

The problem with the Turing machine is that a different machine must be constructed for every new computation to be performed for every input output relation.

This is the reason the Universal Turing machine was introduced which along with input on the tape takes the description of a machine M.

The Universal Turing machine can go on then to simulate M on the rest of the content of the input tape.

A Universal Turing machine can thus simulate any other machine.

The idea of connecting multiple Turing machine gave an idea to Turing −

- Can a Universal machine be created that can 'simulate' other machines?
- This machine is called as Universal Turing Machine

This machine would have three bits of information for the machine it is simulating

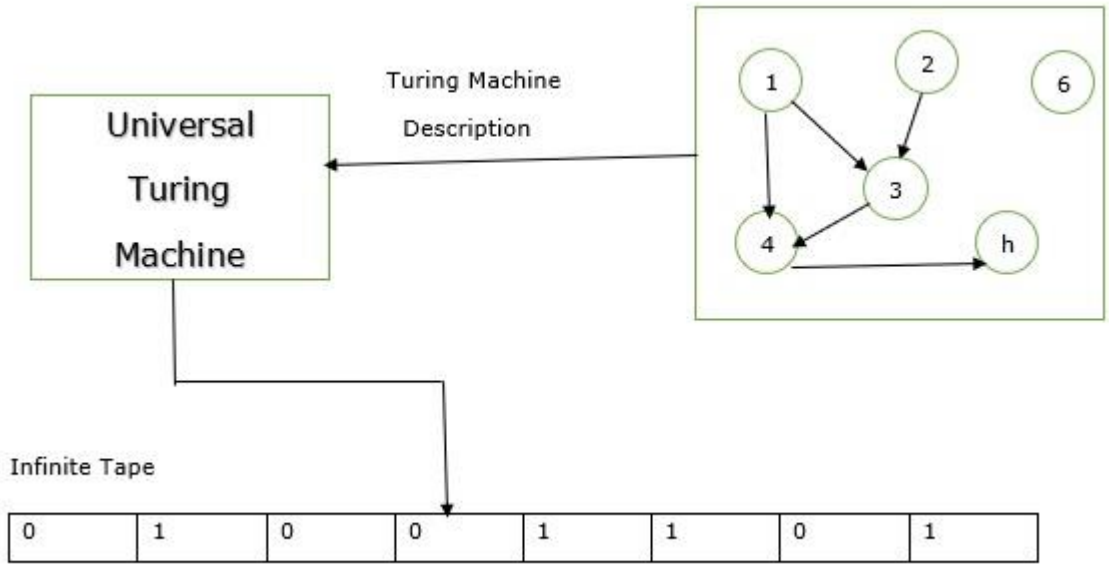
- A basic description of the machine.
- The contents of machine tape.
- The internal state of the machine.

The Universal machine would simulate the machine by looking at the input on the tape and the state of the machine.

It would control the machine by changing its state based on the input. This leads to the idea of a "computer running another computer".

It would control the machine by changing its state based on the input. This leads to the idea of a "computer running another computer".

The schematic diagram of the Universal Turing Machine is as follows −

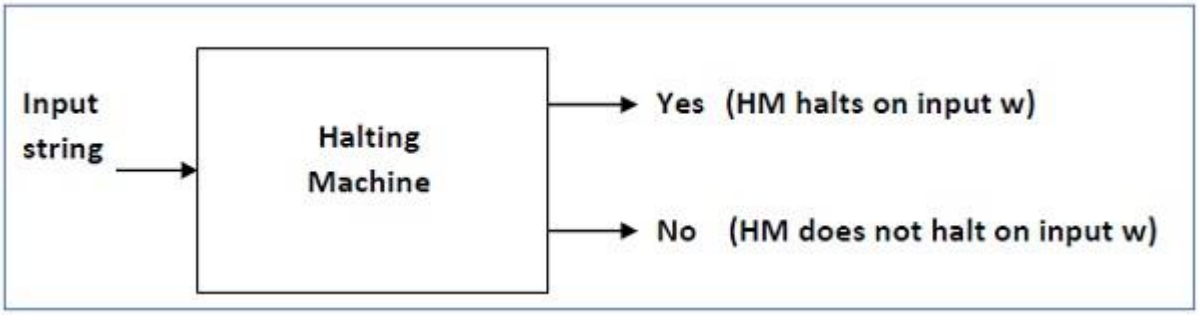


The Halting Problem:

Input − A Turing machine and an input string **w**.

Problem − Does the Turing machine finish computing of the string **w** in a finite number of steps? The answer must be either yes or no.

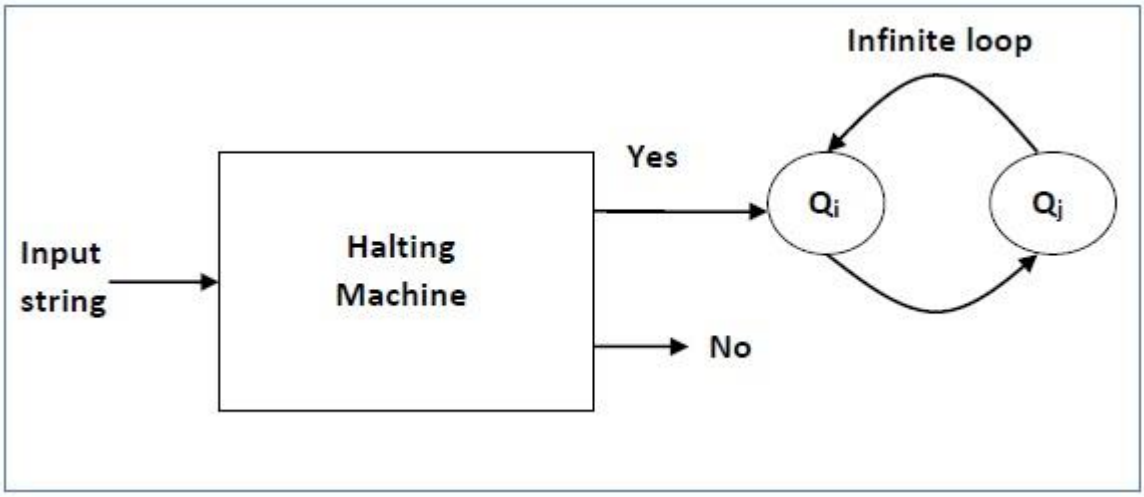
Proof − At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine −



Now we will design an **inverted halting machine (HM)'** as −

- If **H** returns YES, then loop forever.
- If **H** returns NO, then halt.

The following is the block diagram of an 'Inverted halting machine'



Further, a machine **$(HM)_2$** which input itself is constructed as follows −

- If $(HM)_2$ halts on input, loop forever.
- Else, halt.

Here, we have got a contradiction. Hence, the halting problem is **undecidable**.

Decidable & Undecidable Problems:

Before we understand about the decidable and undecidable problems in the theory of computation (TOC), we must learn about the decidable and undecidable language. Hence, let us first see what do you mean by decidable language.

Decidable Language

A language L is called decidable if there is a decider M such that L(M) = L.

- Given a decider M, you can learn whether or not a string w ∈ L(M).
    - Run M on w.
    - Although it might take a long time, M will accept or reject w.
- The set R is the set of all decidable languages.
    - L ∈ R if L is decidable.

Undecidable Language

A decision problem P is undecidable if the language L of all yes instances to P is not decidable.

An undecidable language may be partially decidable but not decidable. Suppose, if a language is not even partially decidable, then there is no Turing machine that exists for the respective language.

Problem

Find whether the problem given below is decidable or undecidable.

“Let the given input be some Turing Machine M and some string w. The problem is to determine whether the machine M, executed on the input string w, ever moves its read head to the left for three delta rules in a row.”

Solution

Define M' is a Turing machine that takes a pair (M,w) as input, where M is a Turing machine recognized by M' and w is the input to M.

Whenever the head of simulated machine M moves to left while processing input w , M' stops and accepts (M,w)

For a particular input to M' (M,w), construction the Turing machine P is −

- P executes M' on (M,w)
- P stops and accepts any input if M' accepts (M,w)

We have tried to reduce the Universal Turing Machine U to P, because we know that L(U) is not decidable, and also we conclude that L(P) is not decidable. Consequently, M' is not decidable.

The proof is by contradiction.

Let us assume that this problem is decidable, and then we have to show that the altering turing machine (ATM) is also decidable −

ATM = { M is a TM and M accepts w}.

Let R be a Turing machine which decides the leftmost problem.

That is, R decides the language

leftmost = { M on input w ever attempts to move its head left when it's head is on the left-most tape cell }.

Now, the idea is to construct a Turing machine S which decides ATM in such a way that it uses R.

On input, S first modifies machine M to M´, so that M´ moves its head to the left from the left-most cell only when M accepts its input.

To ensure that during its computation M´ does not move the head left from the left-most position,

First machine M´ shifts the input w one position to the right, and places a special symbol on the left-most tape cell. The computation of M´ starts with the head on the second tape cell.

During its computation M´ ever attempts to move its head to the left-most tape cell, M´ finds out by reading the special symbol and puts the head back to the second cell, and continues its execution. If M enters an accept state, then M´ enters a loop that forces the head to always move to the left.

After S has constructed M´ it runs the decider R on input < M´; w>.

If R accepts then S accepts, otherwise if R rejects then S rejects.

Therefore, ATM is decidable, which is a contradiction.

Post Correspondence Problem:

The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem. The PCP problem over an alphabet ∑ is stated as follows −

Given the following two lists, **M** and **N** of non-empty strings over ∑ −

M = (x_1, x_2, x_3,……, x_n)

$N = (y_1, y_2, y_3, \ldots\ldots, y_n)$

We can say that there is a Post Correspondence Solution, if for some $i_1, i_2, \ldots\ldots i_k$, where $1 \leq i_j \leq n$, the condition $x_{i1} \ldots\ldots x_{ik} = y_{i1} \ldots\ldots y_{ik}$ satisfies.

Example 1

Find whether the lists

M = (abb, aa, aaa) and N = (bba, aaa, aa)

have a Post Correspondence Solution?

Solution

Here,

$\mathbf{x_2x_1x_3}$ = **'aaabbaaa'**

and $\mathbf{y_2y_1y_3}$ = **'aaabbaaa'**

We can see that

$\mathbf{x_2x_1x_3 = y_2y_1y_3}$

Hence, the solution is **i = 2, j = 1, and k = 3.**

	X1	X2	X3
M	Abb	aa	aaa
N	Bba	aaa	aa

Example 2

Find whether the lists **M = (ab, bab, bbaaa)** and **N = (a, ba, bab)** have a Post Correspondence Solution?

	X1	X2	X3
M	ab	bab	bbaaa
N	a	ba	bab

In this case, there is no solution because −

$|\mathbf{x_2x_1x_3}| \neq |\mathbf{y_2y_1y_3}|$ (Lengths are not same)

Hence, it can be said that this Post Correspondence Problem is **undecidable**.

UNIT-IV

Propositional calculus is a branch of logic. It is also called propositional logic, statement logic, sentential calculus, sentential logic, or sometimes zeroth-order logic. It deals with propositions (which can be true or false) and relations between propositions, including the construction of arguments based on them. Compound propositions are formed by connecting propositions by logical connectives. Propositions that contain no logical connectives are called atomic propositions.

Unlike first-order logic, propositional logic does not deal with non-logical objects, predicates about them, or quantifiers. However, all the machinery of propositional logic is included in first-order logic and higher-order logics. In this sense, propositional logic is the foundation of first-order logic and higher-order logic.

Logical connectives are found in natural languages. In English for example, some examples are "and" (conjunction), "or" (disjunction), "not" (negation) and "if" (but only when used to denote material conditional).

The following is an example of a very simple inference within the scope of propositional logic:

Premise 1: If it's raining then it's cloudy.
Premise 2: It's raining.
Conclusion: It's cloudy.

Both premises and the conclusion are propositions. The premises are taken for granted, and with the application of modus ponens (an inference rule), the conclusion follows.

As propositional logic is not concerned with the structure of propositions beyond the point where they can't be decomposed any more by logical connectives, this inference can be restated replacing those *atomic* statements with statement letters, which are interpreted as variables representing statements:

Premise 1: p $\rightarrow$ Q
Premise 2: P
Conclusion: Q

The same can be stated succinctly in the following way:

Syntax Formulas are certain strings of symbols as specified below.

In this chapter we use formula to mean propositional formula. Later the meaning of formula will be extended to first-order formula. (Propositional) formulas are built from atoms P1, P2, P3, . . , the unary connective ¬, the binary connectives ∧, ∨, and parentheses (,). (The symbols ¬, ∧ and ∨ are read “not”, “and” and “or”, respectively.) We use P, Q, R ... to stand for atoms.

Formulas are defined recursively as follows: Definition of Propositional Formula

1) Any atom P is a formula.

2) If A is a formula so is ¬A.

3) If A, B are formulas, so is (A ∧ B). 4) If A, B are formulas, so is (A ∨ B).

All (propositional) formulas are constructed from atoms using rules

4). Examples of formulas: P , (P ∨ Q), (¬(P ∧ Q) ∧ (¬P ∨ ¬Q)). A s ubformula of a formula A is any substring of A which is a formula. For example, P, Q, (P ∧ Q) and ¬ (P ∧ Q) are all subformulas of ¬(P ∧ Q), but P∧ is not a sub formula. We will use ⊃ ("implies") and ↔ ("is equivalent to") as abbreviations as follows: (A ⊃ B) stands for (¬A ∨ B) (A ↔ B) stands for ((A ⊃ B) ∧ (B ⊃ A)) .

A truth assignment for a propositional vocabulary is a function assigning a truth value to each of the proposition constants of the vocabulary. In this book, we use the digit 1 as a synonym for *true* and 0 as a synonym for *false*; and we refer to the value of a constant or expression under a truth assignment *i* by superscripting the constant or expression with *i* as the superscript.

The assignment shown below is an example for the case of a propositional vocabulary with just three proposition constants, viz. *p*, *q*, and *r*.

$$p^i = 1$$
$$q^i = 0$$
$$r^i = 1$$

Validity and Satisfiability A sentence is **valid if and only if it is satisfied by every truth assignment**. A sentence is unsatisfiable if and only if it is not satisfied by any truth assignment. A sentence is contingent if and only if it is both satisfiable and falsifiable, i.e. it is neither valid nor unsatisfiable.

Validity: – A sentence is valid if it is true in every interpretation (every interpretation is a model). – A sentence s is a valid consequence of a set S of sentences if (S => s) is valid. – Proof methods: Truth -Tables and Inference Rules

Satisfiability: – A set of sentences is satisfiable if there exists an interpretation in which every sentence is true (it has at least one model). – Proof Methods: Truth-Tables and The Davis-Putnam-Logeman-Loveland procedure (DPLL).

Propositional Satisfiability : An instance of SAT is defined as (X, S) – X: A set of 0-1 (propositional) variables – S: A set of sentences (formulas) on X Goal: Find an assignment f: X -> {0, 1} so that every sentence becomes true. SAT is the first NP-complete problem. – Good News: Thousands of problems can be transformed into SAT – Bad News: There are no efficient algorithms for SAT.

EQUIVALENCE : A "brute-force" method to show that two formulas are logically equivalent is to use truth tables. In this lecture we introduce an alternative approach that is more practical in many cases, namely equational reasoning. The idea is to start from some basic equivalence (the Boolean algebra axioms) and derive new equivalences using the closure of logical equivalence under substitution.

1.1 Boolean Algebra Axioms Proposition

1. The following equivalences hold for all formulas F, G and

H: F ∧ F ≡ F F ∨ F ≡ F (Idempotence)

F ∧ G ≡ G ∧ F F ∨ G ≡ G ∨ F (Commutativity)

(F ∧ G) ∧ H ≡ F ∧ (G ∧ H) (F ∨ G) ∨ H ≡ F ∨ (G ∨ H) (Associativity)

F ∧ (F ∨ G) ≡ F F ∨ (F ∧ G) ≡ F (Absorption)

F ∧ (G ∨ H) ≡ (F ∧ G) ∨ (F ∧ H) F ∨ (G ∧ H) ≡ (F ∨ G) ∧ (F ∨ H)

(Distributivity) ¬¬F ≡ F (Double negation) ¬(F ∧ G) ≡ (¬F ∨ ¬G) ¬(F ∨ G) ≡ (¬F ∧ ¬G) (De Morgan's Laws) F ∨ ¬F ≡ true F ∧ ¬F ≡ false (Complementation) F ∨ true ≡ true F ∧ false ≡ false (Zero Laws) F ∨ false ≡ F F ∧ true ≡ F (Identity Laws) Proof.

These can all be shown using truth tables. Notice that the Boolean algebra axioms come in pairs: the equivalences in each pair are dual to each other in the sense that one is obtained from the other by by interchanging ∨ and ∧ and interchanging true and false.

Using the Substitution Theorem and the fact that logical equivalence is reflexive, symmetric, and transitive, we can derive new equivalences from the Boolean algebra axioms. In the following example, each line of the deduction is annotated with the Boolean-algebra axiom that is being used together with ST if the Substitution Theorem is being invoked.

Example . We give an eruptional derivation of the equivalence

(P ∨ (Q ∨ R) ∧ (R ∨ ¬P)) ≡ R ∨ (¬P ∧ Q).

We have (P ∨ (Q ∨ R)) ∧ (R ∨ ¬P) ≡ (P ∨ Q) ∨ R) ∧ (R ∨ ¬P) (Assoc. and ST) ≡ (R ∨ (P ∨ Q)) ∧ (R ∨ ¬P) (Comm. and ST) ≡ R ∨ ((P ∨ Q) ∧ ¬P) (Distr.) ≡ R ∨ (¬P ∧ (P ∨ Q)) (Comm. and ST) ≡ R ∨ ((¬P ∧ P) ∨ (¬P ∧ Q)) (Distr. and ST) ≡ R ∨ (false ∨ (¬P ∧ Q)) (Complement. and ST) ≡ R ∨ (¬P ∧ Q) (Ident. and ST)

The algebraic laws in Proposition 1 allow us to relax some distinctions among formulas. For example, the associativity law allows us to unambiguously write Vn i=1 Fi and Wn i=1 Fi respectively for the conjunction and disjunction of F1, F2, . . . , Fn.

NORMAL FORMS:

Normal Forms A literal is a propositional variable or the negation of a propositional variable. In the former case the literal is positive and in the latter case it is negative.

A formula F is in conjunctive normal form (CNF) if it is a conjunction of clauses, where each clause is a disjunction of literals Li,j : F = Vn i=1 (mWi j=1 Li,j).

A formula F is in disjunctive normal form (DNF) if it is a disjunction of clauses, where each clause is a conjunction of literals Li,j : F = Wn i=1 (mVi j=1 Li,j). Note that we consider true to be a CNF formula with no clauses, and we consider false to be a CNF formula with a single clause, which contains no literals.

Example:.

The formulas representing the 3-colouring problem and the Sudoku problem in the previous lecture are both CNF formulas. 2.1 Equational Transformation to CNF and DNF Theorem 7. For every formula F there is an equivalent formula in CNF and an equivalent formula in DNF.

Proof. We can transform a formula F into an equivalent CNF formula using equation reasoning as follows:

1. Using the Double Negation law and De Morgan's laws, substitute in F every occurrence of a subformula of the form ¬¬G by G ¬(G ∧ H) by (¬G ∨ ¬H) ¬(G ∨ H) by (¬G ∧ ¬H) ¬true by false ¬false by true until no such formulas occur (i.e., push all negations inward until negation is only applied to propositional variables).

2. Using the Distributivity laws, substitute in F every occurrence of a subformula of the form G ∨ (H ∧ R) by (G ∨ H) ∧ (G ∨ R) (H ∧ R) ∨ G by (H ∨ G) ∧ (R ∨ G) G ∨ true by true true ∨ G by true until no such formulas occur (i.e., push all disjunctions inward until no conjunction occurs under a disjunction).

3 A B C F 0 0 1 0 0 1 0 0 1 0 0 0 1 1 1 1 0 0 1 1 0 1 0 1 1 0 0 1 1 1 1

. Use the Identity and Zero laws to remove false from any clause and to delete all clauses containing true. The resulting formula is then in CNF. The translation of F to DNF has the same first step, but dualises steps 2 and 3 (swap ∧ and ∨, and swap true and false). 2.2 Normal Forms from Truth Tables Alternatively, given a formula F, we can read off equivalent.

Resolution in Prepositional Calculus

• Resolution is a valid inference rule producing a new clause implied by two clauses containing complementary literals – A literal is an atomic symbol or its negation, i.e., P, ~P • Amazingly, this is the only interference rule you need to build a sound and complete theorem prover – Based on proof by contradiction and usually called resolution refutation

• The resolution rule was discovered by Alan Robinson (CS, U. of Syracuse) in the mid 60s Resolution • A KB is actually a set of sentences all of which are true, i.e., a conjunction of sentences.

• To use resolution, put KB into conjunctive normal form (CNF), where each sentence written as a disjunction of (one or more) literals Example • KB: [P→Q , Q→R∧S]

• KB in CNF: [~P∨Q , ~Q∨R , ~Q∨S] • Resolve KB(1) and KB(2) producing: ~P∨R (i.e., P→R) • Resolve KB(1) and KB(3) producing: ~P∨S (i.e., P→S) • New KB: [~P∨Q , ~Q∨~R∨~S , ~P∨R , ~P∨S]

Predicate

A predicate is an expression of one or more variables defined on some specific domain. A predicate with variables can be made a proposition by either assigning a value to the variable or by quantifying the variable.

Consider the following statement.

- Ram is a student.

Now consider the above statement in terms of Predicate calculus.

- Here "is a student" is a predicate and Ram is subject.
- Let's denote "Ram" as x and "is a student" as a predicate P then we can write the above statement as P(x).
- Generally a statement expressed by Predicate must have at least one object associated with Predicate. In our case, Ram is the required object with associated with predicate P.

Statement Function

Earlier we denoted "Ram" as x and "is a student" as predicate P then we have statement as P(x). Here P(x) is a statement function where if we replace x with a Subject say Sunil then we'll be having a statement "Sunil is a student."

Thus a statement function is an expression having Predicate Symbol and one or multiple variables. This statement function gives a statement when we replaced the variables with objects. This replacement is called substitution instance of statement function.

Quantifiers

The variable of predicates is quantified by quantifiers. There are two types of quantifier in predicate logic − Universal Quantifier and Existential Quantifier.

Universal Quantifier

Universal quantifier states that the statements within its scope are true for every value of the specific variable. It is denoted by the symbol ∀.

∀ x P(x) is read as for every value of x, P(x) is true.

Example − "Man is mortal" can be transformed into the propositional form ∀ x P(x) where P(x) is the predicate which denotes x is mortal and ∀ x represents all men.

Existential Quantifier

Existential quantifier states that the statements within its scope are true for some values of the specific variable. It is denoted by the symbol ∃.

∃ x P(x) is read as for some values of x, P(x) is true.

Example − "Some people are dishonest" can be transformed into the propositional form ∃ x P(x) where P(x) is the predicate which denotes x is dishonest and ∃ x represents some dishonest men.

Predicate Formulas

Consider a Predicate P with n variables as $P(x_1, x_2, x_3, ..., x_n)$. Here P is n-place predicate and $x_1, x_2, x_3, ..., x_n$ are n individuals variables. This n-place predicate is known as atomic formula of predicate calculus. For Example: P(), Q(x, y), R(x,y,z)

Well Formed Formula

Well Formed Formula (wff) is a predicate holding any of the following −

- All propositional constants and propositional variables are wffs

- If x is a variable and Y is a wff, ∀ x Y and ∀ x Y are also wff
- Truth value and false values are wffs
- Each atomic formula is a wff
- All connectives connecting wffs are wffs

Free and Bound variables

Consider a Predicate formula having a part in form of (∃ x) P(x) of (x)P(x), then such part is called x-bound part of the formula. Any occurrence of x in x-bound part is termed as bound occurrence and any occurrence of x which is not x-bound is termed as free occurrence. See the examples below -

- (∃ x) (P(x) ∧ Q(x))
- (∃ x) P(x) ∧ Q(x)

In first example, scope of (∃ x) is (P(x) ∧ Q(x)) and all occurrences of x are bound occurrences. Whereas in second example, scope of (∃ x) is P(x) and last occurrence of x in Q(x) is a free occurrence.

Universe of Discourse

We can limit the class of individuals/objects used in a statment. Here limiting means confining the input variable to a set of particular individuals/objects. Such a restricted class is termed as Universe of Discourse/domain of individual or universe. See the example below:

Some cats are black.

- C(x) : x is a cat.
- B(x) : x is black.
- (∃ x)(C(x) ∧ B(x))

If Universe of discourse is E = { Katy, Mille } where katy and Mille are white cats then our third statement is false when we replace x with either Katy or Mille where as if Universe of discourse is E = { Jene, Jackie } where Jene and Jackie black cats then our third statement stands true for Universe of Discourse F.

Syntax of Predicate Calculus

The predicate calculus uses the following types of symbols: Constants: A constant symbol denotes a particular entity.

E.g. John, Muriel, 1. Functions: A function symbol denotes a mapping from a number of entities to a single entity:

E.g. Father Of is a function with one argument. Plus is a function with two arguments. Father Of (John) is some person. Plus (2,7) is some number. Predicates: A predicate denotes a relation on a number of entities. e.g. Married is a predicate with two arguments. Odd is a predicate with one argument. Married(John, Sue) is a sentence that is true if the relation of marriage holds between the people John and Sue. Odd (Plus (2,7)) is a true sentence. Variables: These represent some undetermined entity. Examples: x, s1, etc. Boolean operators: ¬, ∨, ∧, ⇒, ⇔. Quantifiers: The

symbols ∀ (for all) and ∃ (there exists). Grouping symbols: The open and close parentheses and the comma. A term is either 1. A constant symbol; or 2. A variable symbol; or 3. A function symbol applied to terms. Examples: John, x, Father Of (John), Plus(x,Plus (1,3)). An atomic formula is a predicate symbol applied to terms. Examples: Odd(x). Odd (plus (2,2)). Married (Sue,Fathe r Of (John)). A formula is either 1. An atomic formula; or 2. The application of a Boolean operator to formulas; or 3. A quantifier followed by a variable followed by a formula. Examples: Odd(x). Odd(x) ∨ ¬Odd(Plus (x,x)). ∃x Odd (Plus (x,y)). ∀x Odd (x) ⇒ ¬Odd (Plus(x,3)). A sentence is a formula with no free variables. (That is, every occurrence of every variable is associated with some quantifier.)

NP COMPLETENESS:

Certain problems in NP have their individual complexity related to that of the entire class (Stephen Cook and Leonid Levin, 1970s) B If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable B These problems are called NP-complete and the phenomenon of NP-completeness is important for both theoretical and practical reasons

B Research trying to show that P 6= NP may focus on an NP-complete problem B If any problem in NP requires more than polynomial time, an NP-complete one also does B Research trying to show that P = NP only needs to find a polynomial time algorithm for one NP-complete problem.

The phenomenon of NP-completeness may prevent wasting time searching for a non-existent polynomial time algorithm to solve a particular problem B Even though we may not have the necessary mathematics to prove that the problem is unsolvable in polynomial time, showing that it is NP-complete is enough evidence, since we believe that P 6= NP.

A Boolean formula is an expression involving Boolean variables and operations. For example ϕ = (¬x∧y)∨(x∧ ¬z) is a Boolean formula B A Boolean formula is satisfiable if some assignments of 1 and 0 (true and false) to its variables makes the formula evaluate to 1. For example, x = 0, y = 1,z = 0 makes ϕ evaluate to 1 B The satisfiability problem (SAT) is to test whether a Boolean formula is satisfiable, i.e. SAT = {ϕ | ϕ is a satisfiable Boolean formula} This is the equivalent for propositional logic of the satisfiability problem for first-order logic we have seen before. Here models have as universe the set {1,0} and only Boolean variables require an interpretation function.