A T M E
College of Engineering

Affiliated to VTU, Belagavi; Approved by AICTE, New Delhi and recognized by Government of Karnataka
Programs accredited by NBA, New Delhi – CV, EC, EE and ME
(Validity: 2022-23 to 2024-25)

Department of Computer Science and Design

Course with code: Theory of Computation-BCG503

Semester:5th

Academic Year 2024-2024

Course Coordinator: Ashwini P, CSE, ATMECE

Phone: 0821-2954081/11
Email: info@atme.in | web: www.atme.in
13th Kilometer, Mysore - Kanakapura
- Bangalore Road, Mysore – 570028

MODULE I:

1. Introduction

The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting". An **automaton** (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a **Finite Automaton (FA)** or **Finite State Machine (FSM).**

2. Why to study Theory of Computation?

Theory of computation is mainly concerned with the study of how problems can be solved using algorithms. It is the study of mathematical properties both of problems and of algorithms for solving problems that depend on neither the details of today's technology nor the programming language.

It is still useful in two key ways:

- It provides a set of ***abstract structures*** that are useful for solving certain classes of problems. These abstract structures can he implemented on whatever hardware/software platform is available
- It defines provable limits to ***what can be computed*** regardless of processor speed or memory size. An understanding of these limits helps us to focus our design effort in areas in which it can pay off, rather than on the computing equivalent of the search for a perpetual motion machine.

The goal is to discover fundamental properties of the problems like:

- Is there any computational ***solution*** to the problem? 1f not. is there a restricted but useful variation of the problem for which a solution does exist?
- If a solution exists, can it be implemented using some ***fixed amount of memory***?
- If a solution exists. how ***efficient*** is it? More specifically. how do its time and space requirements grow as the size of the problem grows?
- Are there groups of problems that are ***equivalent*** in the sense that if there is an efficient

solution to one member of the group there is an efficient solution to all the others?

Applications of theory of computation:

- ***Development of Machine Languages***: Enables both machine-machine and person-machine communication. Without them, none of today's applications of computing could exist. Example: Network communication protocols, HTML etc.
- ***Development of modern programming languages:*** Both the design and the implementation of modern programming languages rely heavily on the theory of context-free languages. Context- free grammars are used to document the languages syntax and they form the basis for the parsing techniques that all compilers use.
- ***Natural language processing***: It is a field of computer science, artificial intelligence, and computational linguistics concerned with the interactions between computers and human (natural) languages.
- ***Automated hardware systems:*** Systems as diverse as parity checkers, vending machines, communication protocols, and building security devices can be straightforwardly described as finite state machines, which is a part of theory of computation.
- ***Video Games:*** Many interactive video games are use large nondeterministic finite state machines.
- ***Security*** is perhaps the most important property of many computer systems. The undecidability results of computation show that there cannot exist a general-purpose method for automatically verifying arbitrary security properties of programs.
- ***Artificial intelligence:*** Artificial intelligence programs solve problems in task domains ranging from medical diagnosis to factory scheduling. Various logical frameworks have been proposed for representing and reasoning with the knowledge that such programs exploit.
- ***Graph Algorithms:*** Many natural structures, including ones as different as organic molecules and computer networks can be modeled as graphs. The theory of complexity tells us that, is there exist efficient algorithms for answering some important questions about graphs. Some questions are "hard", in the sense that no efficient algorithm for them is known nor is one likely to be developed.

3. Strings

Alphabet

Definition: An **alphabet** is any finite set of symbols denoted by Σ (Sometimes also called as **characters** or **symbols**).

Example: Σ = {a, b, c, d} is an alphabet set where ‘a’, ‘b’, ‘c’, and ‘d’ are symbols.

String

Definition: A **string** is a finite sequence of symbols taken from Σ.

Example: 'cabcad' is a valid string on the alphabet set Σ = {a, b, c, d}

Alphabet name	*Alphabet symbols*	*Example strings*
The lower case English alphabet	{a, b, c, …, z}	ε, aabbcg, aaaaa
The binary alphabet	{0, 1}	ε, 0, 001100, 11
A star alphabet	{★, ✪, ☆, ✭, ✯, ✰}	ε, ✪✪, ✪✭✯✰★☆
A music alphabet	{𝅝, 𝅗𝅥, ♩, ♪, 𝅘𝅥𝅯, 𝅘𝅥𝅰, ●}	ε, ♩𝅝𝅗𝅥♪𝅘𝅥𝅯

3.1. Functions on Strings

Length of a String

Definition: It is the number of symbols present in a string. (Denoted by |.|).

Examples: If ***s*** ='cabcad', | ***s*** |= 6; Also |11001101| = 7

If | ***s*** |= 0, it is called an empty string, denoted by **ε.** |ε| = 0

Concatenation of strings: The *concatenation* of two strings *s* and *t,* written *s*||*t* or simply *st,* is the string formed by appending t to *s.* For example, if *x* = *good* and *y* = *bye*, then *xy* = *goodbye*. So |xy| = |x| + |y|.

The empty string, *e,* is the **identity** for concatenation of strings. *(xe = ex = x).*

Concatenation, as a function defined on strings is associative. *(st)w = s (tw).*

String Replication

For each string *w* and each natural number *i,* the string w^i is defined as:Example: $w^0 = \varepsilon$, $w^{i+1} = w^i w$

a^3 = aaa, $(bye)^2$ = byebye, a^0b^3 = bbb

String Reversal: For each string ***w***, the reverse of ***w***, written as w^R, is defined as:

Theorem: If w and x are strings, then $(wx)^R = x^R w^R$.
For example, $(\text{nametag})^R = (\text{tag})^R(\text{name})^R$ = gateman.

Proof: The proof is by induction on $|x|$:

Base case: $|x| = 0$. Then $x = \varepsilon$, and $(wx)^R = (w\varepsilon)^R = (w)^R = \varepsilon w^R = \varepsilon^R w^R = x^R w^R$.

Prove: $\forall n \geq 0\ (((|x| = n) \rightarrow ((wx)^R = x^R w^R)) \rightarrow ((|x| = n + 1) \rightarrow ((wx)^R = x^R w^R)))$.

Consider any string x, where $|x| = n + 1$. Then $x = ua$ for some character a and $|u| = n$. So:

$(wx)^R = (w(ua))^R$	rewrite x as ua
$= ((wu)a)^R$	associativity of concatenation
$= a(wu)^R$	definition of reversal
$= a(u^R w^R)$	induction hypothesis
$= (au^R)w^R$	associativity of concatenation
$= (ua)^R w^R$	definition of reversal
$= x^R w^R$	rewrite ua as x

3.2. Relations on strings

Substring: A string *s* is a substring of a string of t iff *s* occurs contiguously as part of t.

For example: aaa is a substring of aaabbbaaa, aaaaaa is not a substring of aaabbbaaa

Proper Substring: A string r is a proper substring of a string t, iff t is a substring of t and s ≠ t. Every string is a substring (although not a proper substring) of itself. The empty string. e. is a substring of every string.

Prefix: A string s is a prefix of t, iff $\exists x \in \sum^*(t = sx)$. A string s is a proper prefix of a string tiff s is a prefix of t and s≠t. Every string is a prefix (although not a proper prefix) of itself. The empty string ε, is a prefix of every string. For example. the prefixes of abba are: ε, a, ab, abb, abba.

Suffix: A string s is a suffix of t, iff $\exists x \in \sum^*(t = xs)$. A string s is a proper suffix of a string t iff s is a suffix of t and s≠t. Every string is a suffix (although not a proper suffix) of itself. The empty string ε, is a suffix of every string. For example. the prefixes of abba are: ε, a, ba, bba, abba.

4. Languages

A language is a (finite or infinite) set of strings over a finite alphabet ∑. When we are talking about more than one language, we will use the notation $\sum_L$, to mean the alphabet from which the strings in the language L are formed.

Let ∑ = {a, b}. ∑* = {ε, a, b, aa, ab , ba, bb, aaa, aab }.

Some examples of languages over ∑ are:

Φ, {ε}, {a, b}, {ε, a, aa, aaa, aaaa, aaaaa}, {ε, a, aa, aaa, aaaa, aaaaa,.....}

1.1. Techniques for Defining Languages

There are many ways. Since languages are sets. we can define them using any of the set- defining techniques

Ex-1: All a's Precede All b's,

L = {w ϵ {a,b}*: an a's precede all b's in w}. The strings ε, a, aa, aabbb, and bb are in *L* . The strings aba, ba, and abc are not in *L*.

Ex-2: ***Strings that end in 'a'***

L = {x : ∃yϵ {a, b}*, (*x* = ya)}. The strings a, aa, aaa, bbaa and ba are in *L*. The strings ε, bab, and bca are not in *L*. *L* consists of all strings that can be formed by taking some string in {a, b}*and concatenating a single a onto the end of it.

Ex-3: ***Empty language***

L = { } = Φ, the language that contains no strings. ***Note:*** L = { ε } the language that containsa single string, ε. Note that *L* is different from Φ.

Ex-4: Strings of all 'a' s containing zero or more 'a's

Let *L* = { $a^n : n \geq 0$}. *L* = (ε, a, aa, aaa, aaaa, aaaaa,.....)

Ex-5: We define the following languages in terms of the prefix relation on strings: L1 = {wϵ{a, b}* : no prefix of *w* contains b}= { *e* , a, aa, aaa, aaaa, aaaaa, aaaaaa, } .

L2 ={wϵ {a, b}*: no prefix of *w* starts with b}={*w* ϵ{a,b}*: the first character of w is a }U{ε}. L3= {wϵ {a, b}*; every prefix of w starts with b} =Φ. L3 is equal to Φ because ε is a prefix of every string. Since ε does not start with b, no strings meet L3 's requirement.

Languages are sets. So, a computational definition of a language can be given in two ways;

- a **language generator**, which enumerates (lists) the elements of the language, or
- a **language recognizer**, which decides whether or not a candidate string is in the language and returns *True* if it is and *False* if it isn't.

For example, the logical definition. $L = \{x: \exists y \in \{a, b\}^* (x = ya)\}$ can be turned into either alanguage generator (enumerator) or a language recognizer.

In some cases, when considering an enumerator for a language, we may care about the order in which the elements of *L* are generated. If there exists n total order *D* of the elements of $\sum_L$, then we can use *D* to define on *L* a total order called ***lexicographic order*** (written $<_L$.):

- Shorter strings precede longer ones: $\forall x (\forall y ((|x| < |y|) @ (x <_L y)))$ and
- Of strings that are the same length sort them in dictionary order using *D*.

Let L= {w ϵ {a, b}*; all a's precede all b's}. The lexicographic enumeration of Lis:ε, a. b.

aa. ab. bb. aaa. aab. abb. bbb. aaaa, aaab. aabb. abbb. bbbb. aaaaa ....

1.2. Cardinality of a Language

- Cardinality refers to the number of strings in the language.
- The smallest language over any alphabet is ϕ, whose cardinality is 0.
- The largest language over any alphabet ∑ is ∑*. Suppose that ∑ = Φ, then ∑* = {ε} and |∑*| = l. In general, |∑*| is infinite.

Theorem: If $\Sigma \neq \emptyset$ then Σ^* is countably infinite.

Proof: The elements of Σ^* can be lexicographically enumerated by a straightforward procedure that:

- Enumerates all strings of length 0, then length 1, then length 2, and so forth.
- Within the strings of a given length, enumerates them in dictionary order.

This enumeration is infinite since there is no longest string in Σ^*. By Theorem A.1, since there exists an infinite enumeration of Σ^*, it is countably infinite.

Let: $\Sigma = \{a, b\}$.
$L_1 = \{$strings with an even number of a's$\}$.
$L_2 = \{$strings with no b's$\} = \{\varepsilon, a, aa, aaa, aaaa, aaaaa, aaaaaa, \ldots\}$.

$L_1 \cup L_2 = \{$all strings of just a's plus strings that contain b's and an even number of a's$\}$.

$L_1 \cap L_2 = \{\varepsilon, aa, aaaa, aaaaaa, aaaaaaaa, \ldots\}$.

$L_2 - L_1 = \{a, aaa, aaaaa, aaaaaaa, \ldots\}$.

$\neg(L_2 - L_1) = \{$strings with at least one b$\} \cup \{$strings with an even number of a's$\}$.

Concatenation

Let L_1 and L_2 be two languages defined over some alphabet $\sum$. Then their concatenation.written L_1L_2 is:

$$L_1L_2 = \{w \in \Sigma^* : \exists s \in L_1 (\exists t \in L_2 (w = st))\}.$$

Example: Let: L_1 = {cat, dog, mouse, bird}. L_2 = { bone, food}.
L_1L_2 = { catbone, catfood, dogbane, dogfood, mousebone, mousefood, birdbone, birdfood}.

The language {ε} is the **identity for concatenation** of languages. So for all languages L, $L\{\varepsilon\} = \{\varepsilon\}L = L$.

The language Φ is a zero for concatenation of languages. So, for all languages *L*, $L\Phi = \Phi L = \Phi$. That Φ is a zero follows from the definition of the concatenation of two languages as the set consisting of all strings that can he formed by selecting some string 's' from the first language and some string 'l' from the second language and then concatenating them together. There are no ways to select a string from the empty set.

Concatenation on languages is **associative**. So, for all languages L_1L_2 and L_3:

$$((L_1L_2)L_3 = L_1 (L_2L_3)).$$

Reverse

Let L be a language defined over some alphabet $\sum$. Then the reverse of L , written L^R is:$L^R = \{w \in \sum^*: w = x^R$ for some $x \in L\}$.
In other words, L^R is the set of strings that can be formed by taking some string in L andreversing it

Theorem: If L_1 and L_2 are languages, then $(L_1L_2)^R = L_2^R L_1^R$.

Proof: If *x* and *y* are strings, then $\forall x\ (\forall y\ ((xy)^R = y^R x^R))$ Theorem 2.1

$(L_1L_2)^R = \{(xy)^R : x \in L_1 \text{ and } y \in L_2\}$	Definition of concatenation of languages
$= \{y^R x^R : x \in L_1 \text{ and } y \in L_2\}$	Lines 1 and 2
$= L_2^R L_1^R$	Definition of concatenation of languages

Kleene Star

Definition: The **Kleene star** denoted by Σ^*, is a unary operator on a set of symbols or strings,Σ, that gives the infinite set of all possible strings of all possible lengths over Σ including ε.

Representation: $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots\ldots$ where Σ^p is the set of all possible strings of lengthp.

Example: If Σ = {a, b}, Σ^*= {λ, a, b, aa, ab, ba, bb,..... }

Let L = {dog, cat, fish}. Then:
L^* = {ε, dog, cat, fish, dogdog, dogcat,...,
fishdog,..., fishcatfish, fishdogfishcat,... }.

Kleene Closure / Plus

Definition: The set Σ^+ is the infinite set of all possible strings of all possible lengths over Σexcluding ε. $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ Representation:

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots\ldots$

Example: If Σ = {a, b} , Σ^+ ={ a, b, aa, ab, ba, bb, ...}

Closure: A set S is closed under the operation @ if for every element x & y in S, x@y is also anelement of S.

4.5. A Language Hierarchy

A Machine-Based Hierarchy of Language Classesare shown in the diagram.

We have four language classes:

1. **Regular languages**, which can be accepted by some finite state machine.
2. **Context-free languages**, which can be accepted by some pushdown automaton.
3. **Decidable** (or simply D) languages. which can decided by some Turing machine that always halts.
4. **Semi-decidable** (or SD) languages, which can be semi-decided by some Turing machine that halts on all strings in the language.

Each of these classes is a proper subset of the next class, as illustrated in the Figure.
As we move outward in the language hierarchy, we have access to tools with greater and expressive power. We can define $A^nB^nC^n$ as a decidable language but not as a context-free or a regular one. These matters because expressiveness generally comes at a price. The price maybe: Computational efficiency, decidability and clarity.

- *Computational efficiency:* Finite state machines run in time that is linear in the length of the input string. A general context-free parser based on the idea of a pushdown automaton requires time that grows as the cube of the length of the input string. A

Turing machine may require time that grows exponentially (or faster) with the length of the input string.

- *Decidability:* There exist procedures to answer many useful questions about finite state machines. For example, does an FSM accept some particular string? Is an FSM minimal? Are two FSMs identical? A subset of those questions can be answered for pushdown automata. None of them can be answered for Turing machines.
- *Clarity:* There exist tools that enable designers to draw and analyze finite state machines. Every regular language can also be described using the regular expression pattern language. Every context-free language, in addition to being recognizable by some pushdown automaton, can be described with a context-free grammar

5. Finite State Machines (FSM)

A ***finite state machines*** (or FSM) is a computational device whose input is a string and whose output is one of two values; Accept and Reject. FSMs are also sometimes called ***finite state automata*** or ***FSAs.***

5.1. Deterministic FSM

- We begin by defining the class of FSMs whose behavior is **deterministic**.
- These machines, makes exactly one move at each step
- The move is determined by the current state and the next input character.

Definition: Deterministic Finite State Machine (DFSM) is M: $M = (K, \Sigma, \delta, s, A)$, where:
K is a finite set of states
Σ is an alphabet
$s \in K$ is the initial state
$A \subseteq K$ is the set of accepting states, and
δ is the transition function from $(K \times \Sigma)$ to K

Configuration: A ***Configuration*** of a DFSM M is an element of $K \times \Sigma^*$. Configuration captures the two things that make a difference to M's future behavior: i) its current state, the input that remains to be read.

The ***Initial Configuration*** of a DFSM M, on input w, is (s_M, w) , where s_M is start state of M

The transition function δ defines the operation of a DFSM M one step at a time. δ is set of all pairs of states in M & characters in Σ. (Current State, Current Character) ⓪ New State

Relation 'yields': Yields-in-one-step relates configuration, to configuration-1to configuration- 2iff M can move from canfiguration-1, to configuration-2 in one step. Let c be any element of
Σ and let ***w*** be any element of Σ*, then,

$$(q_1, cw) \vdash_M (q_2, w) \text{ iff } ((q_1, c), q_2) \in \delta$$

$\vdash_M^*$ is the reflexive, transitive closure of $\vdash_M$

Complete FSM: A transition is defined for every possible state and every possible character inthe alphabet. Note: This can cause FSM to be larger than necessary, but ALWAYS processes the entire string

Incomplete FSM: One which defines a transition for every possible state & every possible character in the alphabet which can lead to an accepting state Note: If no transition is defined,the string is *Rejected*

Computation: A Computation by M is a finite sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ε), for some state $q \in K_M$
  - ε indicates empty string, entire string is processed & implies a complete DFSM
- $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_n$.

However, M **Halts** when the last character has to be processed or a next transition is not defined

Acceptance / Rejection

A DFSM M, **Accepts** a string w iff $(s, w) \vdash_M^* (q, \varepsilon)$, for some $q \in A_M$.A DFSM M,

Rejects a string w iff $(s, w) \vdash_M^* (q, \varepsilon)$, for some $q \notin A_M$.

The **language accepted by M**, denoted $L(M)$, is the set of all strings accepted by M.

Regular languages

A language is regular iff it is accepted by some DFSM. Some examples are listed below.

- $\{w \in \{a, b\}^* \mid$ every a is immediately followed by b $\}$.
- $\{w \in \{a, b\}^* \mid$ every **a region** in w is of even length$\}$
- binary strings with odd parity.

Designing Deterministic Finite State Machines

Given some language L. how should we go about designing a DFSM to accept L? In general. asin any design task. There is no magic bullet. But there are two related things that it is helpful to think about:

- Imagine any DFSM M that accepts L. As a string w is being read by M, what propertiesof the part of w that has been seen so far are going to have any bearing on the ultimateanswer that M needs to produce? Those are the properties that M needs to record.
- If L is infinite but M has a finite number of states, strings must "cluster". In other words,multiple different strings will all drive M to the same state. Once they have done that, none of their differences matter anymore. If they've driven M to the same state, they share a fate. No matter what comes next, either all of them cause M to accept or all of them cause M to reject.

MODULE – 2

Regular Expressions and Lexical Analysis

Structure:

1. 2.1 **Regular Expression**
2. Equivalence of Re and NFA
3. **DFA**
4. Pumping lemma for regular languages
5. Lexical Analysis

1. Regular Expressions

- A regular expression is used to specify a language, and it does so precisely.

- Regular expressions are very intuitive.
- Regular expressions are very useful in a variety of contexts.
- Given a regular expression, an NFA-ε can be constructed from it automatically.
- Thus, so can an NFA, a DFA, and a corresponding program, all automatically!

Definition:

- Let Σ be an alphabet. The regular expressions over Σ are:

– Ø Represents the empty set { }
– ε Represents the set {ε}
– a Represents the set {a}, for any symbol a in Σ

Let r and s be regular expressions that represent the sets R and S, respectively.

– r+s Represents the set R U S (precedence 3)
– rs Represents the set RS (precedence 2)
– r^* Represents the set R* (highest precedence)
– (r) Represents the set R (not an op, provides precedence)

- If r is a regular expression, then L(r) is used to denote the corresponding language.

- **Examples:** Let Σ = {0, 1}

(0 + 1)* All strings of 0's and 1's
0(0 + 1)* All strings of 0's and 1's, beginning with a 0
(0 + 1)*1 All strings of 0's and 1's, ending with a 1
(0 + 1)*0(0 + 1)* All strings of 0's and 1's containing at least one 0 (0 +
1)*0(0 + 1)*0(0 + 1)* All strings of 0's and 1's containing at least two 0's(0 +

1*(01*01*)*	All strings of 0's and 1's containing an even number of 0's
(1*01*0)*1*	All strings of 0's and 1's containing an even number of 0's

Identities:

Øu = uØ = Ø Multiply by 0

- u + Ø = u
- u + u = u

8. u* = (u*)*

9. u(v+w) = uv+uw

10. (u+v)w = uw+vw

11. (uv)*u = u(vu)*

12. (u+v)* = (u*+v)*

= u*(u+v)*

= (u+vu*)*

= (u*v*)*

= u*(vu*)*

= (u*v)*u*

2. Equivalence of Regular Expressions and NFA-ε

- **Note:** Throughout the following, keep in mind that a string is accepted by an NFA-ε if there exists a path from the start state to a final state.
- **Lemma 1:** Let r be a regular expression. Then there exists an NFA-ε M such that L(M) = L(r). Furthermore, M has exactly one final state with no transitions out of it.
- **Proof:** (by induction on the number of operators, denoted by OP(r), in r).
- **Basis:** OP(r) = 0

Then r is either Ø, ε, or **a**, for some symbol **a** in Σ

- **Inductive Hypothesis:** Suppose there exists a $k \geq 0$ such that for any regular expression r where $0 \leq OP(r) \leq k$, there exists an NFA-ε such that L(M) = L(r). Furthermore, suppose that M has exactly one final state.

- **Inductive Step:** Let r be a regular expression with k + 1 operators (OP(r) = k + 1), where k + 1 >= 1.

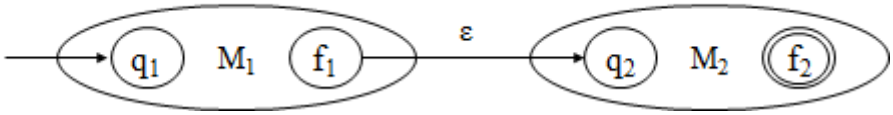
Case 1) $r = r_1 + r_2$

Since OP(r) = k +1, it follows that 0<= $OP(r_1)$, $OP(r_2)$ <= k. By the inductive hypothesis there exist NFA-ε machines M_1 and M_2 such that $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. Furthermore, both M_1 and M_2 have exactly one final state.

Case 2) $r = r_1 r_2$

Since OP(r) = k+1, it follows that 0<= $OP(r_1)$, $OP(r_2)$ <= k. By the inductive hypothesis there exist NFA-ε machines M_1 and M_2 such that $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. Furthermore, both M_1 and M_2 have exactly one final state.

Construct M as:



Case 3) $r = r_1^*$

Since OP(r) = k+1, it follows that 0<= $OP(r_1)$ <= k. By the inductive hypothesis there exists an NFA-ε machine M_1 such that $L(M_1) = L(r_1)$. Furthermore, M_1 has exactly one final state.

- Example:

Problem: Construct FA equivalent to RE, r = 0(0+1)*

Solution:
r = r1r2
r1 = 0
r2 = (0+1)*
r2 = r3*
r3 = 0+1
r3 = r4 + r5
r4 = 0
r5 = 1

Transition graph:

3. Definitions Required to Convert a DFA to a Regular Expression

- Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA with state set $Q = \{q_1, q_2, \ldots, q_n\}$, and define:
  $R_{i,j} = \{ x \mid x \text{ is in } \Sigma^* \text{ and } \delta(q_i,x) = q_j\}$
  $R_{i,j}$ is the set of all strings that define a path in M from q_i to q_j.

- **Lemma 2:** Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA. Then there exists a regular expression r such that $L(M) = L(r)$.

if i=j.Case 1) No transitions from q_i to q_j and i != j $r^0_{i,j} = \emptyset$

Case 2) At least one ($m \geq 1$) transition from q_i to q_j and i != j

$$r^0_{i,j} = a_1 + a_2 + a_3 + \ldots + a_m \quad \text{where } \delta(q_i, a_p) = q_j, \text{ for all } 1 \leq p \leq m$$

Case 3) No transitions from q_i to q_j and i = j$r^0_{i,j} = \varepsilon$

Case 4) At least one ($m \geq 1$) transition from q_i to q_j and i = j

$$r^0_{i,j} = a_1 + a_2 + a_3 + \ldots + a_m + \varepsilon \quad \text{where } \delta(q_i, a_p) = q_j \text{ for all } 1 \leq p \leq m$$

- Inductive Hypothesis:
  Suppose that $R^{k-1}_{i,j}$ can be represented by the regular expression $r^{k-1}_{i,j}$ for all$1 \leq i,j \leq n$, and some $k \geq 1$.

- Inductive Step:
  Consider $R^k_{i,j} = R^{k-1}_{i,k} (R^{k-1}_{k,k})^* R^{k-1}_{k,j} \cup R^{k-1}_{i,j}$. By the inductive hypothesis there exist regular expressions $r^{k-1}_{i,k}$, $r^{k-1}_{k,k}$, $r^{k-1}_{k,j}$, and $r^{k-1}_{i,j}$ generating $R^{k-1}_{i,k}$, $R^{k-1}_{k,k}$, $R^{k-1}_{k,j}$, and $R^{k-1}_{i,j}$, respectively. Thus, if we let

$$r^k_{i,j} = r^{k-1}_{k,k} (r^{k-1}_{k,k})^* r^{k-1}_{k,j} + r^{k-1}_{i,j}$$

  then $r^k_{i,j}$ is a regular expression generating $R^k_{i,j}$, i.e., $L(r^k_{i,j}) = R^k_{i,j}$.

- Finally, if $F = \{q_{j1}, q_{j2}, \ldots, q_{jr}\}$, then
  $r^n_{1,j1} + r^n_{1,j2} + \ldots + r^n_{1,jr}$
  is a regular expression generating L(M).•

4. Pumping Lemma for Regular Languages

- Pumping Lemma relates the size of string accepted with the number of states in a DFA
- What is the largest string accepted by a DFA with *n* states?
- Suppose there is no loop?
  Now, if there is a loop, what type of strings are accepted *via* the loop(s)?
- **Lemma:** (the pumping lemma)

  Let M be a DFA with $|Q| = n$ states. If there exists a string x in L(M), such that $|x| \geq n$,then there exists a way to write it as x = uvw, where u,v, and w are all in Σ^* and:

  - $1 \leq |uv| \leq n$
  - $|v| \geq 1$
  - such that, the strings uv^iw are also in L(M), for all $i \geq 0$

- Let:
  - $u = a_1 \ldots a_s$
  - $v = a_{s+1} \ldots a_t$
- Since $0 \leq s < t \leq n$ and $uv = a_1 \ldots a_t$ it follows that:
  - $1 \leq |v|$ and therefore $1 \leq |uv|$
  - $|uv| \leq n$ and therefore $1 \leq |uv| \leq n$
- In addition, let:
  - $w = a_{t+1} \ldots a_m$
- It follows that $uv^iw = a_1 \ldots a_s(a_{s+1} \ldots a_t)^i a_{t+1} \ldots a_m$ is in L(M), for all $i \geq 0$.

  In other words, when processing the accepted string x, the loop was traversed once, but could have been traversed as many times as desired, and the resulting string would still be accepted.

4.1 Closure Properties of Regular Languages

1. Closure Under Union

- If L and M are regular languages, so is L ∪ M.
- Proof: Let L and M be the languages of regular expressions R and S, respectively.Then
- R+S is a regular expression whose language is L ∪ M.

2. Closure Under Concatenation and Kleene Closure

- RS is a regular expression whose language is LM.
- R* is a regular expression whose language is L*.

3. Closure Under Intersection

- If L and M are regular languages, then so is L ∩ M.
- Proof: Let A and B be DFA's whose languages are L and M, respectively.

4. Closure Under Difference

- If L and M are regular languages, then so is L – M = strings in L but not M.Proof:
- Let A and B be DFA's whose languages are L and M, respectively.

5. Closure Under Complementation

- The complement of language L (w.r.t. an alphabet Σ such that Σ* contains L) is Σ* – L.Since
- Σ* is surely regular, the complement of a regular language is always regular.

6. Closure Under Homomorphism

- If L is a regular language, and h is a homomorphism on its alphabet,then
  h(L) = {h(w) | w is in L} is also a regular language.

5. Grammar

- **Definition:** A grammar G is defined as a 4-tuple, G = (V, T, S, P)
  Where,
  - V is a finite set of objects called variables,
  - T is a finite set of objects called terminal symbols,
  - S ∈ V is a special symbol called start variable,
  - P is a finite set of productions.

  Assume that V and T are non-empty and disjoint.

- Example:
  Consider the grammar G = ({S}, {a, b}, S, P) with P given by S → aSb, S → ε_.
  For instance, we have S ⇒ aSb ⇒ aaSbb ⇒ aabb. It is not hard to conjecture that L(G) = $\{a^n b^n \mid n \geq 0\}$.

5.1 Right, Left-Linear Grammar

- **Right-linear Grammar:** A grammar G = (V, T, S, P) is said to be right-linear if all productions are of the form:
  A → xB,
  A → x,
  Where A, B ∈ V and x ∈ T*.

  - Example#1:

    $S \rightarrow abS \mid a$ is an example of a right-linear grammar.

    - Can you figure out what language it generates?
    - $L = \{w \in \{a,b\}^* \mid w$ Contains alternating a's and b's , begins with an a, and ends with a $b\}$

      $\cup \{a\}$

    - $L((ab)^*a)$

- **Left-linear Grammar:** A grammar G = (V, T, S, P) is said to be left-linear if all productions are of the form:
  A → Bx,
  A → x,
  Where A, B ∈ V and x ∈ T*.
  - Example#2:

    $S \rightarrow Aab$
    $A \rightarrow Aab \mid aB$
    $B \rightarrow a$
    is an example of a left-linear grammar.

    - Can you figure out what language it generates?
    - $L = \{w \hat{I} \{a,b\}^* \mid w$ is aa followed by at least one set of alternating ab's$\}$

    - $L(aaab(ab)^*)$

  - Example#3:

    Consider the grammar
    $S \rightarrow A$
    $A \rightarrow aB \mid \lambda$
    $B \rightarrow Ab$
    This grammar is NOT regular.

- No "mixing and matching" left- and right-recursive productions.

5.2 Regular Grammar

- A linear grammar is a grammar in which at most one variable can occur on the right side of any production without restriction on the position of this variable.

- An example of linear grammar is G = ({S, S1, S2}, {a, b}, S, P) with
  S → S1ab,
  S1 → S1ab | S2,
  S2 → a.

- A **regular grammar** is one that is either right-linear or left-liner.

5.3 Testing Equivalence of Regular Languages

- Let L and M be reg langs (each given in some form).

  To test if L = M

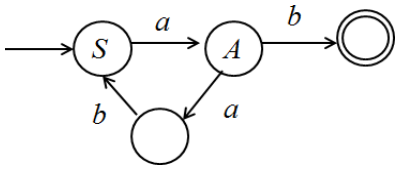
  1. Convert both L and M to DFA's.
  2. Imagine the DFA that is the union of the two DFA's (never mind there are two start states)
  3. If TF-algo says that the two start states are distinguishable, then L 6= M, otherwise, L = M.

  We can “see" that both DFA accept L(ε+(0+1)*0). The result of the TF-algo isTherefore

  the two automata are equivalent.

5.4 Regular Grammars and NFA's

- It's not hard to show that regular grammars generate and nfa's accept the same class of languages: the regular languages!
- It's a long proof, where we must show that
  - Any finite automaton has a corresponding left- or right-linear grammar,
  - And any regular grammar has a corresponding nfa.
- Example:
  - We get a feel for this by example.



Let $S \rightarrow aA \quad A \rightarrow abS \mid b$ONTEXT FREE-GRAMMAR

- **Definition:** Context-Free Grammar (CFG) has 4-tuple: G = (V, T, P, S)

Where,

V	-	A finite set of variables or *non-terminals*
T	-	A finite set of *terminals* (V and T do not intersect) P
	-	A finite set of *productions*, each of the form A –> α, Where A is in V and α is in (V U T)* Note: that α may be ε.
S	-	A starting non-terminal (S is in V)

- Example#1 CFG:

G = ({S}, {0, 1}, P, S)P:

(1) S –> 0S1 or just simply S –> 0S1 | ε

(2) S –> ε

- Example Derivations:

S	=> 0S1	(1)
S	=> ε	(2)
	=> 01	(2)
S	=> 0S1	(1)
	=> 00S11	(1)
	=> 000S111	(1)
	=> 000111	(2)

- Note that G "generates" the language $\{0^k1^k \mid k>=0\}$

6. Derivation (or Parse) Tree

- **Definition:** Let G = (V, T, P, S) be a CFG. A tree is a derivation (or parse) tree if:
  - Every vertex has a label from V U T U {ε}
  - The label of the root is S
  - If a vertex with label A has children with labels $X_1, X_2, \ldots, X_n$, from left to right, then

    $$A \rightarrow X_1, X_2, \ldots, X_n$$

    must be a production in P
  - If a vertex has label ε, then that vertex is a leaf and the only child of its' parent
- More Generally, a derivation tree can be defined with any non-terminal as the root.

Definition: A derivation is *leftmost (rightmost)* if at each step in the derivation a production is applied to the leftmost (rightmost) non-terminal in the sentential form.

- The first derivation above is **leftmost**, second is **rightmost** and the third is neither.

```
        break;
    cases for the other characters;
}
```

A *language* is any countable set of strings over some fixed alphabet. This definition is very broad. Abstract languages like 0, the *empty set,* or {e}, the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed C programs and the set of all grammatically correct English sentences, although the latter two languages are difficult to specify exactly.

Operations on languages

LUD: Union operation, where L=set of alphabets {A..Z,a..z} and D=set of digits {0..9}
LD: Concatenation
L4:exponentiation: set of strings with 4 letters L0=
Li=Li-1L
L*=all strings ith : Kleene closure of L D+: set of all strings of digits of one or more

- *L* U *D* is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
- *LD* is the set of strings of length two, each consisting of one letter followed by one digit.
- L4 is the set of all 4-letter strings.
- *L** is the set of all strings of letters, including e, the empty string.
- *L(L* U *D)** is the set of all strings of letters and digits beginning with a letter.
- *D+* is the set of all strings of one or more digits.

➢ Specification of tokens

1. In theory of compilation regular expressions are used to formalize the specification of tokens
2. Regular expressions are means for specifying regular languages
3. Example:
    i. Letter_(letter_ | digit)*
4. Each regular expression is a pattern specifying the form of strings

➢ Regular expressions

1. Ɛ is a regular expression, L(Ɛ) = {Ɛ}
2. If a is a symbol in ∑then a is a regular expression, L(a) = {a}
3. (r) | (s) is a regular expression denoting the language L(r) L(s)
4. (r)(s) is a regular expression denoting the language L(r)L(s)

5. (r)* is a regular expression denoting (L(r))*
6. (r) is a regular expression denoting L(r)

➢ Regular definitions

1. d1 -> r1
2. d2 -> r2
3. …
4. dn -> rn

5. Example:

6. letter_ -> A | B | … | Z | a | b | … | Z | _
7. digit ->0|1|…|9
8. id -> letter_ (letter_ | digit)*

Extensions

- One or more instances: (r)+
- Zero of one instances: r?
- Character classes: [abc]
- Example:
  - letter_ -> [A-Za-z_]
  - digit -> [0-9]
  - id -> letter_(letter_|digit)*

Transition diagrams

These are the flow charts, as an intermediate step in the construction of a lexical analyzer. This takes actions when a lexical analyzer is called by the parser to get the next token. We use transition diagram to keep track of information about characters that are seen as and when the forward pointer scans the input. Lexeme beginning pointer points to the character following the last lexeme found.

E=M* C**2eof

Transition diagrams have a collection of nodes or circles, called *states.* Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of at ate as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer.

Edges are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. If we are in some state**,** and the next input symbol is *a,* we look for an edge out of state *s* labeled by *a* (and perhaps by other symbols, as well). If we find such an edge, we advance the *forward* pointer arid enter the state of the transition diagram to which that edge leads. We shall as assume that our transition diagrams are *deterministic,* meaning that there is never more than one edge out of a given state with a given symbol among its labels. We shall

MODULE III:

1. Context Free Grammar
2. Minimization of Context Free Grammar

1. Ambiguity in Context Free Grammar

- **Definition:** Let G be a CFG. Then G is said to be ambiguous if there exists an x in L(G) with >1 leftmost derivations. Equivalently, G is said to be ambiguous if there exists an x in L(G) with >1 parse trees, or >1 rightmost derivations.
- Note: Given a CFL L, there may be more than one CFG G with L = L(G). Some ambiguous and some not.
- Definition: Let L be a CFL. If every CFG G with L = L(G) is ambiguous, then L is inherently ambiguous.
- **Example:** Consider the string aaab and the preceding grammar.
- The string has two left-most derivations, and therefore has two distinct parse trees and is ambiguous .

1.1 Eliminations of Useless Symbols

- **Definition:**
  Let $G = (V, T, S, P)$ be a context-free grammar. A variable $A \in V$ is said to be useful if and only if there is at least one $w \in L(G)$ such that

  $$S \Rightarrow^* xAy \Rightarrow^* w$$
  with $x, y \in (V \cup T)^*$.

  In words, a variable is useful if and only if it occurs in at least on derivation. A variable that is not useful is called useless. A production is useless if it involves any useless variable
- For a grammar with productions

$S \rightarrow aSb \mid \lambda \mid A$
$A \rightarrow aA$

A is useless variable and the production $S \rightarrow A$ plays no role since A cannot be eventually transformed into a terminal string; while A can appear in a sentential form derived from$S$, this sentential form can never lead to sentence!

Hence, removing $S \rightarrow A$ (and $A \rightarrow aA$) does not change the language, but does simplifythe grammar.

- For a grammar with productions

$S \rightarrow A$
$A \rightarrow aA \mid \lambda$
$B \rightarrow bA$

B is useless so is the production $B \rightarrow bA$! Observe that, even though a terminal string canbe derived from B, there is no way to get to B from S, i.e. cannot achieve
$S \Rightarrow^* xBy$.

- Example:
Eliminate useless symbols and productions from $G = (V, T, S, P)$, where
$V = \{S, A, B, C\}$, $T = \{a, b\}$ and
P consists of

$S \rightarrow aS \mid A \mid C$
$A \rightarrow a$
$B \rightarrow aa$
$C \rightarrow aCb$

First, note that the variable C cannot lead to any terminal string, we can then remove C and its associated productions, we get G_1 with $V_1 = \{S, A, B\}$, $T_1 = \{a\}$ and P_1 consisting of

$S \rightarrow aS \mid A$
$A \rightarrow a$
$B \rightarrow aa$

Next, we identify variables that cannot be reached from the start variable. We can createa dependency graph for V_1. For a context-free grammar, a dependency graph has its vertices labeled with variables with an edge between any two vertices I and J if there is a production of the form

$I \rightarrow xJy$

Consequently, the variable B is shown to be useless and can be removed together with its associated production.

The resulting grammar $G' = (V', T', S, P')$ is with $V' = \{S, A\}$, $T' = \{a\}$ and P' consistingof

$$S \to aS \mid A$$
$$A \to a$$

1.2 Eliminations of λ-Production

- **Definition :**
  a) Any production of a context-free grammar of the form
  $$A \to \lambda$$
  is called a λ-production.

  b) Any variable A for which the derivation
  $$A \Rightarrow^* \lambda$$
  is possible is called nullable.

- If a grammar contains some λ-productions or nullable variables but does not generate the language that contains an empty string, the λ-productions can be removed!

- Example:
  Consider the grammar, G with productions
  $$S \to aS_1b$$
  $$S_1 \to aS_1b \mid \lambda$$
  $L(G) = \{a^nb^n \mid n \geq 1\}$ which is a λ-free language. The λ-production can be removed afteradding new productions obtained by substituting λ for S_1 on the right hand side.

  We get an equivalent G' with productions
  $$S \to aS_1b \mid ab$$
  $$S_1 \to aS_1b \mid ab$$

- Theorem:
  Let G be any context-free grammar with $\lambda \notin L(G)$. There exists an equivalent grammar G' without λ-productions.

Proof :

Find the set V_N of all nullable variables of G

1. For all productions $A \to \lambda$, put A in V_N

2. Repeat the following step until no further variables are added to V_N:
For all productions
$$B \to A_1A_2 \ldots A_n$$

where $A_1, A_2, \ldots, A_n$ are in V_N, put B in V_N.

With the resulting V_N, P' can be constructed by looking at all productions in P of theform
$$A \to x_1x_2 \ldots x_m,\ m \geq 1$$

where each $x_i \in V \cup T$.

For each such production of P, we put in P' the production plus all productions generatedby replacing nullable variables with λ in all possible combinations. However, if all x_i are nullable, the resulting production $A \to \lambda$ is not put in P'.

- Example:

For the grammar G with

$$S \to ABaC$$
$$A \to BC$$
$$B \to b \mid \lambda$$
$$C \to D \mid \lambda$$
$$D \to d$$

the nullable variables are A, B, and C.

The equivalent grammar G' without λ-productions has P' containing

$S \to ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Ba \mid Aa \mid a$
$A \to BC \mid C \mid B$
$B \to b$
$C \to D$
$D \to d$

1.3 Eliminations of MODULE-Production

- **Definition:**

Any production of a context-free grammar of the form

$$A \to B$$

where $A, B \in V$ is called a MODULE-production.

- Theorem:

Let $G = (V, T, S, P)$ be any context-free grammar without λ-productions. There exists a context-free grammar $G' = (V', T', S, P')$ that does not have any MODULE-productions and that is equivalent to G.

Proof:

First of all, Any MODULE-production of the form $A \to A$ can be removed without anyeffect.
We then need to consider productions of the form $A \to B$ where A and B are different variables.

Straightforward replacement of B (with $x_1 = x_2 = \lambda$) runs into a problem when we have

$$A \to B$$
$$B \to A$$

We need to find for each A, all variables B such that

$$A \Rightarrow^* B$$

This can be done via a dependency graph with an edge (I, J) whenever the grammar G has a MODULE-production $I \to J$; $A \Rightarrow^* B$ whenever there is a walk from A to B in the graph.

The new grammar G' is generated by first putting in P' all non-MODULE-productionsof P. Then, for all A and B with $A \Rightarrow^* B$, we add to P'

$$A \to y_1 \mid y_2 \mid \ldots \mid y_n$$

where $B \to y_1 \mid y_2 \mid \ldots \mid y_n$ is the set of all rules in P' with B on the left. Not that the rules are taken from P', therefore, none of y_i can be a single variable! Consequently, no MODULE- productions are created by this step.

- Example:

Consider a grammar G with

$$S \to Aa \mid B$$
$$A \to a \mid bc \mid B$$
$$B \to A \mid bb$$

We have $S \Rightarrow^* A$, $S \Rightarrow^* B$, $A \Rightarrow^* B$ and $B \Rightarrow^* A$.

First, for the set of original non-MODULE-productions, we have

$$S \to Aa$$
$$A \to a \mid bc$$
$$B \to bb$$

We then add the new rules

$$S \to a \mid bc \mid bb$$
$$A \to bb$$
$$B \to a \mid bc$$

We finally obtain the equivalent grammar G' with P' consisting of

$$S \to Aa \mid a \mid bc \mid bb$$
$$A \to a \mid bc \mid bb$$
$$B \to bb \mid a \mid bc$$

Notice that B and its associate production become useless.

2 Minimization of Context Free Grammar

- **Theorem:**

Let L be a context-free language that does not contain λ. There exists a context-free grammar that generates L and that does not have any useless productions, λ-productions or MODULE-productions.

Proof:

We need to remove the undesirable productions using the following sequence of steps.

1. Remove λ-productions
2. Remove MODULE-productions
3. Remove useless productions

MODULE V:

1. **Turing Machines**
2. The Halting Problem
3. **The Universal language**
4. A Church- Turing thesis
5. **Linear Bounded Automata.**

1. Turing Machines (TM)

- **Generalize the class of CFLs:**

- Recursively enumerable languages are also known as *type 0* languages.
- Context-sensitive languages are also known as *type 1* languages.
- Context-free languages are also known as *type 2* languages.
- Regular languages are also known as *type 3* languages.

- TMs model the computing capability of a general purpose computer, which informally can be described as:
  - Effective procedure
    - Finitely describable
    - Well defined, discrete, "mechanical" steps
    - Always terminates
  - Computable function
    - A function computable by an effective procedure

- TMs formalize the above notion.

1.1 Deterministic Turing Machine (DTM)

- Two-way, infinite tape, broken into cells, each containing one symbol.
- Two-way, read/write tape head.
- Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current state.
- An input string is placed on the tape, padded to the left and right infinitely with blanks, read/write head is positioned at the left end of input string.
- In one move, depending on the current state and the current symbol being scanned, the TM 1) changes state, 2) prints a symbol over the cell being scanned, and 3) moves its' tape head one

cell left or right.

- Many modifications possible.

1.2 Formal Definition of a DTM

– A DTM is a seven-tuple:

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Q	A finite set of states
Γ	A finite tape alphabet
B	A distinguished blank symbol, which is in Γ
Σ	A finite input alphabet, which is a subset of Γ– {B}q_0
	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Qδ A

next-move function, which is a *mapping* from

Q x Γ –> Q x Γ x {L,R}

Intuitively, δ(q,s) specifies the next state, symbol to be written and the direction of tapehead movement by M after reading symbol s while in
state q.

- **Example #1:** $\{0^n1^n \mid n >= 1\}$

	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	(q_1, 0, R)	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	(q_2, 0, L)	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

- **Example #1:** $\{0^n1^n \mid n >= 1\}$

	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

  - The TM basically matches up 0's and 1's
  - q_1 is the "scan right" state
  - q_2 is the "scan left" state
  - q_4 is the final state

- **Example #2:** $\{w \mid w$ is in $\{0,1\}^*$ and w ends with a $0\}$

0
00
10
10110
Not ε

$Q = \{q_0, q_1, q_2\}$
$\Gamma = \{0, 1, B\}$
$\Sigma = \{0, 1\}$

$F = \{q_2\}$

	0	1	B
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	(q_1, B, L)
q_1	$(q_2, 0, R)$	-	-
q_2	-	-	-

  - q_0 is the "scan right" state
  - q_1 is the verify 0 state

– **Definition:** Let M = (Q, Σ, Γ, δ, q_0, B, F) be a TM, and let w be a string in Σ*. Then w is *accepted* by M iff

$$q_0w \vdash^* \alpha_1 p \alpha_2$$

Where p is in F and α_1 and α_2 are in Γ*

– **Definition:** Let M = (Q, Σ, Γ, δ, q_0, B, F) be a TM. The *language accepted by M*, denoted L(M), is the set

L={w | w is in Σ* and w is accepted by M}

In contrast to FA and PDAs, if a TM simply *passes through* a final state then the string is accepted.

- Given the above definition, no final state of an TM need have any exiting transitions. *Henceforth, this is our assumption*.
- If x is not in L(M) then M may enter an infinite loop, or halt in a non-final state**.**
- Some TMs halt on all inputs, while others may not. In either case the language defined by TM is still well defined.

– **Definition:** Let L be a language. Then L is *recursively enumerable* if there exists a TM M such that L = L(M).

– If L is r.e. then L = L(M) for some TM M, and
  - If x is in L then M halts in a final (accepting) state.
  - If x is not in L then M may halt in a non-final (non-accepting) state, or loop forever.

– **Definition:** Let L be a language. Then L is *recursive* if there exists a TM M such that L = L(M) and M halts on all inputs.

– If L is recursive then L = L(M) for some TM M, and
  - If x is in L then M halts in a final (accepting) state.
  - If x is not in L then M halts a non-final (non-accepting) state.

– The set of all recursive languages is a subset of the set of all recursively enumerable languages

– Terminology is easy to confuse: A *TM* is not recursive or recursively enumerable, rather a *language* is recursive or recursively enumerable.

– **Observation:** Let L be an r.e. language. Then there is an infinite list M_0, M_1, … of TMs such that L = L(M_i).

– **Question:** Let L be a recursive language, and M_0, M_1, … a list of all TMs such that L = L(M_i), and choose any i>=0. Does M_i always halt?

Answer: Maybe, maybe not, but *at least one in the list does*.

– **Question:** Let L be a recursive enumerable language, and M_0, M_1, … a list of all TMs such that $L = L(M_i)$, and choose any $i>=0$. Does M_i always halt?

Answer: Maybe, maybe not. Depending on L, none might halt or some may halt.

– If L is also recursive then L is recursively enumerable.

Question: Let L be a recursive enumerable language that is not recursive (L is in r.e. – r), andM_0, M_1, … a list of all TMs such that $L = L(M_i)$, and choose any $i>=0$. Does M_i always halt?
Answer: No! If it did, then L would not be in r.e. – r, it would be recursive.

- Let M be a TM.
  - Question: Is L(M) r.e.?
    Answer: Yes! By definition it is!
  - Question: Is L(M) recursive?
    Answer: Don't know, we don't have enough information.
  - Question: Is L(M) in r.e – r?
    Answer: Don't know, we don't have enough information.
- Let M be a TM that halts on all inputs:
  - Question: Is L(M) recursively enumerable?
    Answer: Yes! By definition it is!
  - Question: Is L(M) recursive?
    Answer: Yes! By definition it is!
  - Question: Is L(M) in r.e – r?
    Answer: No! It can't be. Since M always halts, L(M) is recursive.
- Let M be a TM.
  - As noted previously, L(M) is recursively enumerable, but may or may not be recursive.
  - Question: Suppose that L(M) is recursive. Does that mean that M always halts?
    Answer: Not necessarily. However, some TM M' must exist such that L(M') = L(M) and M' always halts.
  - Question: Suppose that L(M) is in r.e. – r. Does M always halt?
    Answer: No! If it did then L(M) would be recursive and therefore not in r.e. – r.
- Let M be a TM, and suppose that M loops forever on some string x.

- Question: Is L(M) recursively enumerable?
  Answer: Yes! By definition it is.

- Question: Is L(M) recursive?
  Answer: Don't know. Although M doesn't always halt, some other TM M' may existsuch

  that L(M') = L(M) and M' always halts.

- Question: Is L(M) in r.e. – r?
  Answer: Don't know.

Closure Properties for Recursive and Recursively Enumerable Languages

- **TMs Model General Purpose Computers:**
  - If a TM can do it, so can a GP computer
  - If a GP computer can do it, then so can a TM

  If you want to know if a TM can do X, then some equivalent question are:
  - *Can a general purpose computer do X?*
  - *Can a C/C++/Java/etc. program be written to do X?*

  For example, is a language L recursive?
  - *Can a C/C++/Java/etc. program be written that always halts and accepts L?*

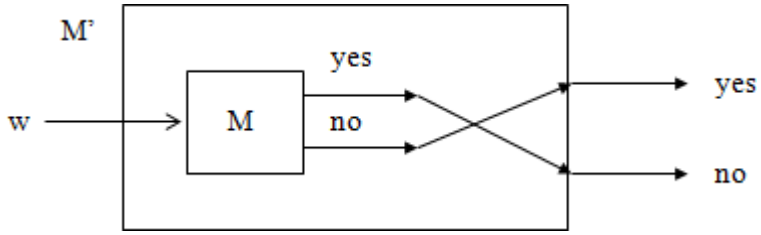
- TM Block Diagrams:
  - If L is a recursive language, then a TM M that accepts L and always halts can be pictorially represented by a "chip" that has one input and two outputs.

  - If L is a recursively enumerable language, then a TM M that accepts L can be pictorially represented by a "chip" that has one output.

  - Conceivably, M could be provided with an output for "no," but this output cannot be counted on. Consequently, we simply ignore it.

– **Theorem:** The recursive languages are closed with respect to complementation, i.e., if L is a recursive language, then so is

  Proof: Let M be a TM such that L = L(M) and M always halts. Construct TM M' as



- Note That:
  - M' accepts iff M does not
  - M' always halts since M always halts

From this it follows that the complement of L is recursive. •

- **Theorem:** The recursive languages are closed with respect to union, i.e., if L_1 and L_2 are recursive languages, then so is

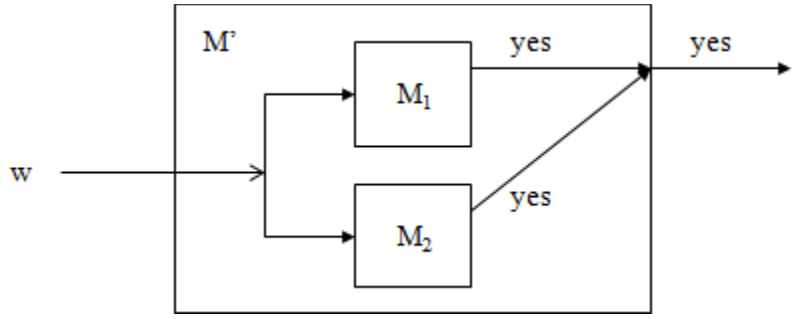
Proof: Let M_1 and M_2 be TMs such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$ and M_1 and M_2 alwayshalts. Construct TM M' as follows:

- Note That:
  - $L(M') = L(M_1) \cup L(M_2)$
    - $L(M')$ is a subset of $L(M_1) \cup L(M_2)$
    - $L(M_1) \cup L(M_2)$ is a subset of $L(M')$
  - M' always halts since M_1 and M_2 always halt

It follows from this that $L_3 = L_1 \cup L_2$ is recursive.

- **Theorem:** The recursive enumerable languages are closed with respect to union, i.e., if L_1 and L_2 are recursively enumerable languages, then so is $L_3 = L_1 \cup L_2$

Proof: Let M_1 and M_2 be TMs such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$. Construct M' asfollows:



- Note That:
  - L(M') = L(M_1) U L(M_2)
    - L(M') is a subset of L(M_1) U L(M_2)
    - L(M_1) U L(M_2) is a subset of L(M')
  - M' halts and accepts iff M_1 or M_2 halts and accepts

It follows from this that is recursively enumerable.

2. The Halting Problem – Background

- **Definition:** A decision problem is a problem having a yes/no answer (that one presumably wants to solve with a computer). Typically, there is a list of parameters on which the problem is based.
  - Given a list of numbers, is that list sorted?
  - Given a number x, is x even?
  - Given a C program, does that C program contain any syntax errors?
  - Given a TM (or C program), does that TM contain an infinite loop?

From a practical perspective, many decision problems do not seem all that interesting.However, from a theoretical perspective they are for the following two reasons:

- Decision problems are more convenient/easier to work with when proving complexity results.
- Non-decision counter-parts are typically at least as difficult to solve.

- Notes:
  - The following terms and phrases are analogous:

Algorithm	-	A halting TM program
Decision Problem	-	A language
(un)Decidable	-	(non)Recursive

Statement of the Halting Problem

- **Practical Form:** (P1)
  Input: Program P and input I. Question:
  Does P terminate on input I?

- Theoretical Form: **(P2)**
  Input: Turing machine M with input alphabet Σ and string w in Σ*.
  Question: Does M halt on w?

- A Related Problem We Will Consider First: **(P3)**
  Input: Turing machine M with input alphabet Σ and one final state, and string w in Σ*.Question:
  Is w in L(M)?

- Analogy:
  Input: DFA M with input alphabet Σ and string w in Σ*.
  Question: Is w in L(M)?

  Is this problem decidable? Yes!

- Over-All Approach:
  - We will show that a language L_d is not recursively enumerable
  - From this it will follow that is not recursive
  - Using this we will show that a language L_u is not recursive
  - From this it will follow that the halting problem is undecidable.

3. The Universal Language

- Define the language L_u as follows:

  $L_u = \{x \mid x \text{ is in } \{0, 1\}^* \text{ and } x = \langle M,w \rangle \text{ where M is a TM encoding and w is in L(M)}\}$

- Let x be in {0, 1}*. Then either:

1. x doesn't have a TM prefix, in which case x is **not** in L_u
2. x has a TM prefix, i.e., x = <M,w> and either:
   a) w is not in L(M), in which case x is **not** in L_u
   b) w is in L(M), in which case x is in L_u

- Compare P3 and L_u:

  (P3):
  Input: Turing machine M with input alphabet Σ and one final state, and string w in Σ*.

- Notes:
  - L_u is P3 expressed as a language
  - Asking if L_u is recursive is the same as asking if P3 is decidable.
  - We will show that L_u is not recursive, and from this it will follow that P3 is un-decidable.
  - From this we can further show that the halting problem is un-decidable.
  - Note that L_u is recursive if M is a DFA.

4. Church-Turing Thesis

- There is an effective procedure for solving a problem if and only if there is a TM that halts for all inputs and solves the problem.

- There are many other computing models, but all are equivalent to or subsumed by TMs. *There is no more powerful machine* (Technically cannot be proved).

- DFAs and PDAs do not model all effective procedures or computable functions, but only a subset.

- If something can be "computed" it can be computed by a Turing machine.

- Note that this is called a ***Thesis***, not a theorem.

- It can't be proved, because the term "can be computed" is too vague.

- But it is universally accepted as a true statement.

- Given the ***Church-Turing Thesis***:

  - What does this say about "computability"?

  - Are there things even a Turing machine can't do?

  - If there are, then there are things that simply can't be "computed."

    - Not with a Turing machine

        - Not with your laptop
        - Not with a supercomputer
    - There ARE things that a Turing machine can't do!!!
- *The* **Church-Turing Thesis:**
    - In other words, there is no problem for which we can describe an algorithm that can't be done by a Turing machine.

The Universal Turing machine

- If Tm's are so damned powerful, can't we build one that simulates the behavior of any Tm on any tape that it is given?
- Yes. This machine is called the ***Universal Turing machine***.
- How would we build a Universal Turing machine?
    - We place an encoding of any Turing machine on the input tape of the Universal Tm.
    - The tape consists entirely of zeros and ones (and, of course, blanks)
    - Any Tm is represented by zeros and ones, using unary notation for elements and zeros as separators.
- Every Tm instruction consists of four parts, each a represented as a series of **1**'s and separated by **0**'s.
- Instructions are separated by **00**.
- We use unary notation to represent components of an instruction, with
    - 0 = **1**,
    - 1 = **11**,
    - 2 = **111**,

- 3 = **1111**,
- ***n*** = **111...111** (***n***+1 1's).

- We encode q_n as $n + 1$ 1's
- We encode symbol a_n as $n + 1$ 1's
- We encode move left as 1, and move right as 11

111101110111110111010010111011011011100

q_3, a_2, q_4, a_2, L q_0, a_1, q_1, a_1, R

- Any Turing machine can be encoded as a unique long string of zeros and ones, beginning with a **1**.
- Let T_n be the Turing machine whose encoding is the number ***n***.

5. Linear Bounded Automata

- A Turing machine that has the length of its tape limited to the length of the input string is called a linear-bounded automaton (LBA).
- A linear bounded automaton is a 7-tuple *nondeterministic* Turing machine M = (Q, S, G, d, q_0,q_{accept}, q_{reject}) except that:

a. There are two extra tape symbols < and >, which are not elements of G.

b. The TM begins in the configuration (q_0<*x*>), with its tape head scanning the symbol < in cell 0. The > symbol is in the cell immediately to the right of the input string *x*.

c. The TM cannot replace < or > with anything else, nor move the tape head left of < or right of >.

Context-Sensitivity

- *Context-sensitive production* any production □ □ □ satisfying |□≤| □|.
- *Context-sensitive grammar* any generative grammar $G = \langle \square \square, \square, \Pi, \square \rangle$ such that every production in Π context-sensitive.
- No empty productions.

Context-Sensitive Language

- Language *L context-sensitive* if there exists context-sensitive grammar *G* such that either $L = L(G)$ or $L = L(G) \cup \{ \}$.

- Example**:**
  The language L = $\{a^n b^n c^n : n \geq 1\}$ is a C.S.L. the grammar isS →

  abc/ aAbc,

  Ab → bA,

  AC → Bbcc,

  bB → Bb,

  aB → aa/ aaA

  The derivation tree of $a^3b^3c^3$ is looking to be as followingS ⇒

  aAbc

  ⇒ abAc

  ⇒ abBbcc

  ⇒ aBbbcc ⇒ aaAbbcc

  ⇒ aabAbcc

  ⇒ aabbAcc ⇒ aabbBbccc

  ⇒ aabBbbccc

  ⇒ aaBbbbccc

  ⇒ aaabbbccc

CSG = LBA

- A language is accepted by an LBA iff it is generated by a CSG.
- Just like equivalence between CFG and PDA
- Given an $x \in$ CSG G, you can intuitively see that and LBA can start with S, and nondeterministically choose all derivations from S and see if they are equal to the input string x. Because CSL's are non-contracting, the LBA only needs to generate derivations of length $\leq |x|$. This is because if it generates a derivation longer than $|x|$, it will never be able to shrink to the size of $|x|$.