Course Outcomes

CO 1: Acquire fundamental understanding of the core concepts in automata theory and Theory of Computation

CO 2: Design and develop lexical analyzers, parsers and code generators

CO 3: Design Grammars and Automata (recognizers) for different language classes and become knowledgeable about restricted models of Computation (Regular, Context Free) and their relative powers.

CO 4: Acquire fundamental understanding of the structure of a Compiler and Apply concepts automata theory and Theory of Computation to design Compilers

CO 5: Design computations models for problems in Automata theory and adaptation of such model in the field of compilers

Institution Vision

To produce top-quality engineers who are groomed for attaining excellence in their profession and competitive enough to help in the growth of nation and global society.

Institution Mission

M1: To offer affordable high-quality graduate program in engineering with value education and make the students socially responsible.

M2: To support and enhance the institutional environment to attain research excellence in both faculty and students and to inspire them to push the boundaries of knowledge base.

M3: To identify the common areas of interest amongst the individuals for the effective industry-institute partnership in a sustainable way by systematically working together.

M4: To promote the entrepreneurial attitude and inculcate innovative ideas among the engineering professionals.

Department Vision

To be a center of excellence in Information Science & Engineering education, research and training to meet the growing needs of the industry and society.

Department Mission

M1: To impart theoretical and practical knowledge through the concepts and technologies in Information Science and Engineering

M2: To foster research, collaboration and higher education with premier institutions and industries.

M3: Promote innovation and entrepreneurship to fulfill the needs of the society and industry

Program Educational Objectives

PEO1: Analyse, design and implement solutions to the real-world problems in the field of Information Science and Engineering with multidisciplinary setup

PEO2: Keep abreast with the technology, innovation and pursue higher education with high standards of social and professional ethics

PEO3: Develop professional and entrepreneurship skills to work effectively as an individual and in a team to meet the ever-changing goals of the organization

Program Outcomes

PO1: **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

PO2: **Problem Analysis:** Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural science and engineering sciences.

PO3: **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal and environmental considerations.

PO4: **Conduct investigations of complex problems:** Use research based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: **Modern tool usage:** Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations

PO6: **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice

PO7: **Environment sustainability:** Understand the impact of the professional engineering solutions in the societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions

PO11: **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to ones own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: **Lifelong learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broader context of technological change.

Program Specific Outcomes

PSO1: Design, implement and maintain the information systems that fulfill the current needs of the industry and society

PSO2: Apply computational theory, storage and networking concepts to solve the day to day problems of the world

Introduction to Automata Theory

Finite automata are computing devices that accept/recognize regular languages and are used to model operations of many systems we find in practice. Their operations can be simulated by a very simple computer program. A kind of systems finite automata can model and a computer program to simulate their operations are discussed.

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, ♩𝅝.𝅗𝅥.𝅘𝅥𝅯𝅘𝅥𝅰

Emptry String

The empty string is the string with zero occurrences of symbols. Denoted by ε

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

Powers of an Alphabet

If Σ is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an exponential notation_ We define Σ^k to be the set of strings of length k, each of whose symbols is in Σ

Example Note that $\Sigma^0 = \{\epsilon\}$, regardless of what alphabet Σ is. That is, ϵ is the only string whose length is 0.

If $\Sigma = \{0, 1\}$, then $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$,

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

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, ε, is the **identity** for concatenation of strings. *(x* ε = ε*x = 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: a^3 = aaa, $(bye)^2$ = byebye, a^0b^3 = bbb

$$w^0 = \varepsilon$$
$$w^{i+1} = w^i w$$

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, ...}

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 : ∃y∈ {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 contains a single string, ε. Note that *L* is different from Φ.

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

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

Ex-5: We define the following languages in terms of the prefix relation on strings:

L1 = {w∈{a, b}* : no prefix of *w* contains b}= { *e* , a, aa, aaa, aaaa, aaaaa, aaaaaa, } .

L2 ={w∈ {a, b}*: no prefix of *w* starts with b}={*w* ∈{a,b}*: the first character of w is a }U{ε}.

L3= {w∈ {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.

Deterministic Finite Automata

Definition:

An **automaton** is represented formally by a 5-tuple **(Q,Σ,δ,q0,F)**, where:

Q is a finite set of *states*.

Σ is a finite set of symbols, called the alphabet of the automaton.

δ is the **transition function**, that is, δ: Q × Σ → Q.

A transition function that takes as arguments a state and an input symbol and returns a state. The transition function will commonly be denoted δ. In our informal graph representation of automata δ was represented by arcs between states and the labels on the arcs. If q is a state, and a is an input symbol, then δ(q,a) is that state p such that there is an arc labeled a from q to p.

q_0 is the *start state*, been processed, where $q_0 \in$ Q.

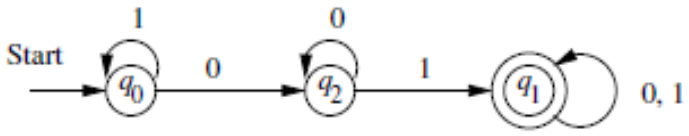
F is a set of states of Q (i.e. F ∈Q) called accept states.

A deterministic finite automaton will often be referred to by its acronym: DFA

The most succinct representation of a DFA is a listing of the five components above

A= (Q,Σ,δ,q0,F),

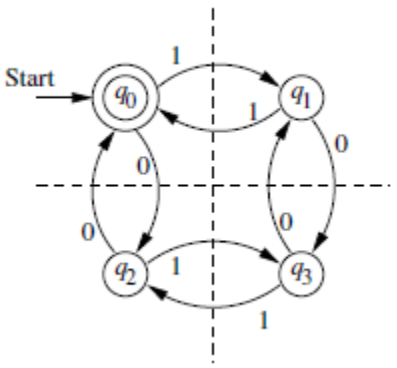
Example 1: The transition diagram and table for the DFA accepting all strings with a substring 01



	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Example 2: Design a DFA to accept the language

L = {w | w has both an even number of 0's and an even number of 1's}



$A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$

	0	1
$* \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

The check involves computing $\hat{\delta}(q_0, w)$ for each prefix w of 110101, starting at ϵ and going in increasing size. The summary of this calculation is:

- $\hat{\delta}(q_0, \epsilon) = q_0$.
- $\hat{\delta}(q_0, 1) = \delta\big(\hat{\delta}(q_0, \epsilon), 1\big) = \delta(q_0, 1) = q_1$.
- $\hat{\delta}(q_0, 11) = \delta\big(\hat{\delta}(q_0, 1), 1\big) = \delta(q_1, 1) = q_0$.
- $\hat{\delta}(q_0, 110) = \delta\big(\hat{\delta}(q_0, 11), 0\big) = \delta(q_0, 0) = q_2$.
- $\hat{\delta}(q_0, 1101) = \delta\big(\hat{\delta}(q_0, 110), 1\big) = \delta(q_2, 1) = q_3$.
- $\hat{\delta}(q_0, 11010) = \delta\big(\hat{\delta}(q_0, 1101), 0\big) = \delta(q_3, 0) = q_1$.
- $\hat{\delta}(q_0, 110101) = \delta\big(\hat{\delta}(q_0, 11010), 1\big) = \delta(q_1, 1) = q_0$.

Configurations of DFAs

A ***configuration*** of a DFA M is an element of: $K \times \Sigma^*$

It captures the two things that decide M's future behavior:

- its current state
- the remaining, unprocessed input

The ***initial configuration*** of a DFA A, on input w, is: (s, w)

The Yields Relations

The *yields-in-one-step* relation $|-_M$

$(q, w) |-_M (q', w')$ iff

- $w = a\ w'$ for some symbol $a \in \Sigma$, and
- $\delta (q, a) = q'$

The relation *yields*, $|-_M{}^*$, is the reflexive, transitive closure of $|-_M$

If $C_i |-_M{}^* C_j$, iff M can go from C_i to C_j in zero (due to"reflexive") or more (due to "transitive") steps.

Notation: |- and |- * are also used.

Path

A ***path*** by M is a **maximal** sequence of configurations $C_0, C_1, C_2 \ldots$ such that:

- C_0 is an initial configuration,
- $C_0 |-_M C_1 |-_M C_2 |-_M \ldots$

- In other words, a path is just a sequence of steps from the start configuration going as far as possible
- A path ends when it enters an **accepting configuration**, or it has no where to go (no transition defined for C*n*)
- This definition of path is applicable to all machines (FSM, PDA, TM, deterministic or nondeterministic).
- A path P can be infinite. For FSM, DPDA, or NFA and NDPDA without ε- transitions, P always ends. For NFA and NDPDA with ε-transitions, P can be infinite. For TM (deterministic or nondeterministic), P can be infinite.
- For deterministic machines, there is only one path (ends or not).
- A path accepts *w* if it ends at an accepting configuration
  - Accepting configuration varies for different machines
- A path rejects *w* if it ends at a non-accepting configuration

Accepting and Rejecting

A DFA *M* ***accepts*** a string *w* iff the path accepts it.

- i.e., $(s, w) \vdash_M^* (q, \varepsilon)$, for some $q \in A$.

- For DFA, (q, ε) where $q \in A$ is an accepting configuration

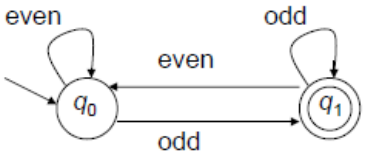
A DFA *M* ***rejects*** a string *w* iff the path rejects it.

- **The** path, because there is only one.

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

Theorem: Every DFA *M*, on input *w*, halts in at most |*w*| steps.

Accepting Example



On input 235, the configurations are:

$(q0, 235) \vdash_M (q0, 35)$

$\vdash_M$

$\vdash_M$

Thus $(q_0, 235) \vdash_M^* (q_1, \varepsilon)$

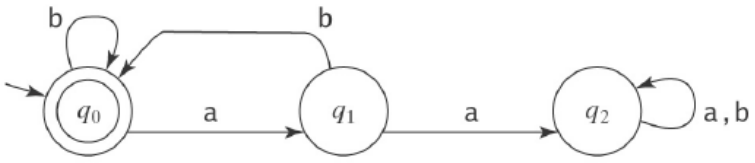
- If *M* is a DFA and $\varepsilon \in L(M)$, what simple property must be true of *M*?
  - The start state of *M* must be an accepting state

Regular Languages

A language is ***regular*** iff it is accepted by some FSM.

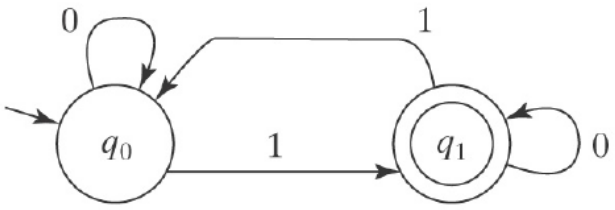
Example:

$L = \{w \in \{a, b\}^* :$ every a is immediately followed by a b$\}$.



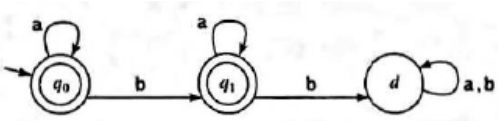
$L = \{w \in \{0, 1\}^* : w \text{ has odd parity}\}$.

A binary string has odd parity iff the number of 1's is odd



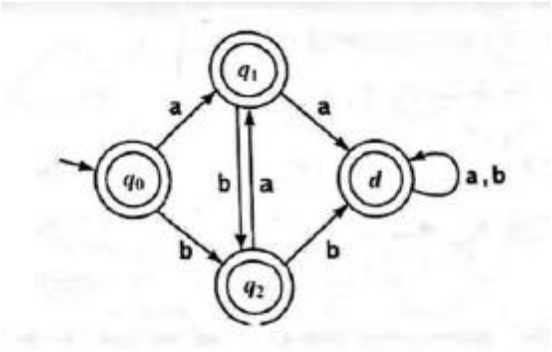
No More Than One b

$L = \{w \in \{a, b\}^* : w \text{ contains no more than one b}\}$.



Checking Consecutive Characters

$L = \{w \in \{a, b\}^* : \text{no two consecutive characters are the same}\}$.



Floating Point Numbers

Let FLOAT = { w : w is the string representation of a floating point number}.

Assume the following syntax for floating point numbers:

• A floating point number is an optional sign, followed by a decimal number, followed by an optional exponent.

• A decimal number may be of the form x or $x.y$, where x and y are nonempty strings of decimal digits.

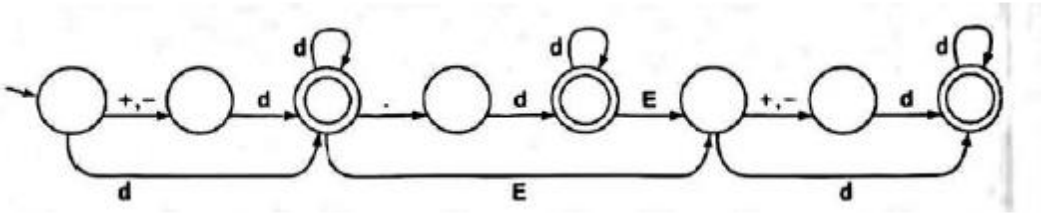
• An exponent begins with E and is followed by an optional sign and then an integer.

• An integer is a nonempty string of decimal digits.

So, for example, these strings represent floating point numbers:

+3.0, 3.0, 0.3El, 0.3E+l, -0.3E+1, -3E8

FLOAT is regular because it can be accepted by the DFA:



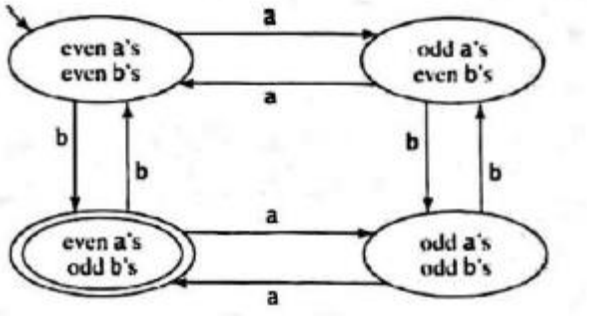
In this diagram, we have used the shorthand d to stand for any one of the decimal digits (0 - 9). And we have omitted the dead state to avoid arrows crossing over each other.

Designing Deterministic Finite State Machines

- Imagine any DFA *M* that accepts *L*. As a string *w* is being read by *M*, what properties of the part of *w* that has been seen so far are going to have any bearing on the ultimate answer that *M* needs to produce?
- *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. The smallest DFA for any language L is the one that has exactly one state for every group of initial substrings that share a common fate.

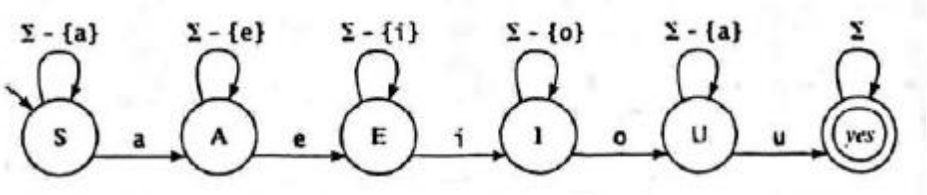
Even a's, Odd b's

Let $L = \{w \in \{a, b\}^* : w$ contains an even number of a's and an odd number of b's$\}$.



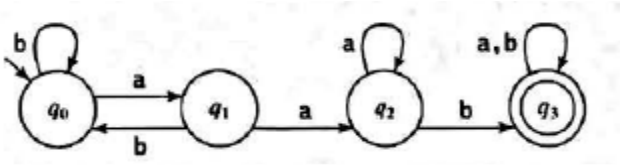
All the Vowels in Alphabetical Order

Let L = { w∈ {a - z} • : all five vowels, a, e, i. o, and u, occur in w in alphabetical order}.



A Substring that Doesn't Occur

Let L =, { w∈ {a, b }*: w does not contain the substring aab }.



It is straightforward to design an FSM that looks for the substring aab. So we can begin building a machine to accept L by building the following machine to accept ¬L. Then we can convert this machine into one that accepts L by making states q_0, q_1, and q_2 accepting and state q_3 nonaccepting.

The Missing Letter Language

Let Σ = {a, b, c, d}.

Let $L_{Missing}$ = {w : there is a symbol $a_i \in \Sigma$ not appearing in w}.

Try to make a DFA for $L_{Missing}$

- Doable, but complicated. Consider the number of accepting states
  - all missing (1)
  - 3 missing (4)
  - 2 missing (6)
  - 1 missing (4)

Nondeterministic Finite Automata

• In the theory of computation, a **nondeterministic finite state machine** or **nondeterministic finite automata (NFA)** is a finite state machine where for each pair of state and input symbol there may be several possible next states.

- This distinguishes it from the deterministic finite automaton (DFA), where the next possible state is uniquely determined.
- Although the DFA and NFA have distinct definitions, it may be shown in the formal theory that they are equivalent, in that, for any given NFA, one may construct an equivalent DFA, and vice-versa
- Both types of automata recognize only regular languages.
- Nondeterministic machines are a key concept in computational complexity theory, particularly with the description of complexity classes *P* and *NP*.

Definition of an NFA

$M = (Q, \Sigma, \boldsymbol{\delta}, q_0, F)$, where:

Q is a finite set of states

Σ is an alphabet

$q_0 \in Q$ is the initial state

$F \subseteq K$ is the set of accepting states, and

$\boldsymbol{\delta}$ is the transition relation. It is a finite subset of $(Q \times (\Sigma \cup \{\varepsilon\})) \times Q$

NFA and DFA

$\boldsymbol{\delta}$ is the transition relation. It is a finite subset of

$\boldsymbol{(Q \times (\Sigma \cup \{\varepsilon\})) \times Q}$

Recall the definition of DFA:

$A = (Q, \Sigma, \delta, q_0, F)$, where:

Q is a finite set of states

Σ is an alphabet

$q_0 \in K$ is the initial state

$F \subseteq K$ is the set of accepting states, and

δ is the transition function from $\boldsymbol{(K \times \Sigma)}$ **to** $\boldsymbol{K}$

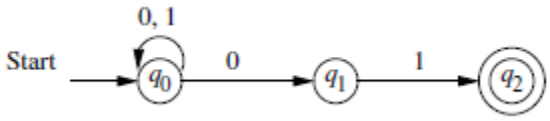
δ: $(K \times (\Sigma \cup \{\varepsilon\})) \times K$ δ: $(K \times \Sigma)$ **to** K

Key difference:

• In every configuration, a DFA can make exactly one move; this is not true for NFA

• *M* may enter a config. from which one or more competing moves are possible.

Example: NFA accepting all strings that end in 01



$(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$	$\rightarrow q_0$: $\{q_0, q_1\}$, $\{q_0\}$ q_1: $\emptyset$, $\{q_2\}$ $*q_2$: $\emptyset$, $\emptyset$ (columns: 0, 1)

Equivalence of Deterministic and Nondeterministic Finite Automata

The proof that DFA's can do whatever NFAs can do involves an important "construction" called the subset construction because it involves constructing all subsets of the set of states of the NFA. In general, many proofs about automata involve constructing one automaton from another. It is important for us to observe the subset construction as an example of how one formally describes one automaton in terms of the states and transitions of another without knowing the specifics of the latter automaton.

The subset construction starts from an NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Its goal is the description of a DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(D) = L(N)$. Notice that the input alphabets of the two automata are the same, and the start state of D is the set containing only the start state of N. The other components of D are constructed as follows.

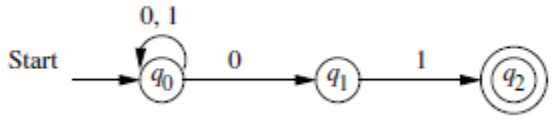
- Q_D is the set of subsets of Q_N; i.e., Q_D is the *power set* of Q_N. Note that if Q_N has n states, then Q_D will have 2^n states. Often, not all these states are accessible from the start state of Q_D. Inaccessible states can

be "thrown away," so effectively, the number of states of D may be much smaller than 2^n.

- F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of N's states that include at least one accepting state of N.
- For each set $S \subseteq Q_N$ and for each input symbol a in Σ,

$$\delta_D(S, a) = \bigcup_{p \text{ in } S} \delta_N(p, a)$$

That is, to compute $\delta_D(S, a)$ we look at all the states p in S, see what states N goes to from p on input a, and take the union of all those states.

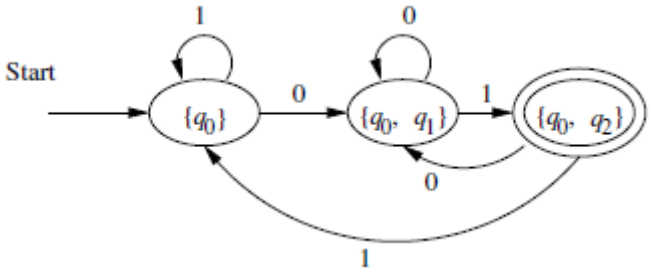


	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

The complete subset construction

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

Renaming the states



Finite Automata with Epsilon Transitions

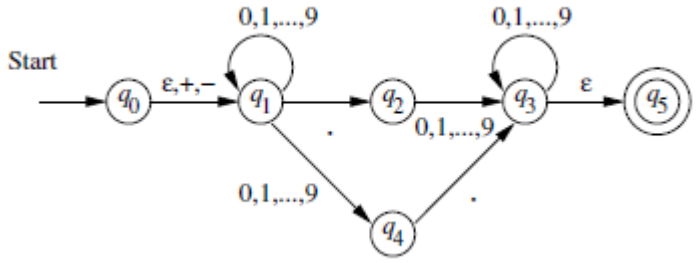
We shall now introduce another extension of the finite automaton. The new "feature" is that we allow a transition on ε the empty string. In effect, an NFA is allowed to make a transition spontaneously, without receiving an input symbol.

Uses of ε - Transitions

We shall begin with an informal treatment of ε - NFAs.

Example: In Fig is an ε - NFA that accepts decimal numbers consisting of:

1. An optional + or – sign,
2. A string of digits,
3. A decimal point, and
4. Another string of digits. Either this string of digits, or the string can be empty, but at least one of the two strings of digits must be nonempty

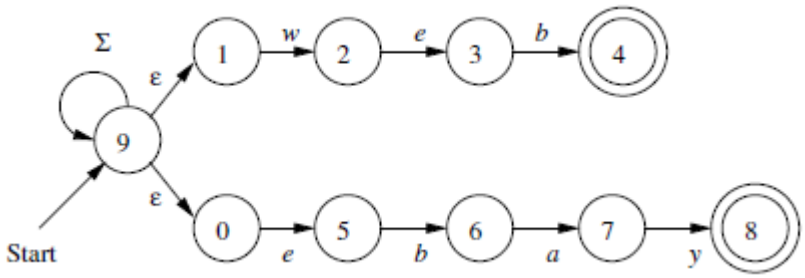


$$E = (\{q_0, q_1, \ldots, q_5\}, \{., +, -, 0, 1, \ldots, 9\}, \delta, q_0, \{q_5\})$$

where δ is defined by the transition table in Fig. 2.20. □

	ϵ	$+, -$	.	$0, 1, \ldots, 9$
q_0	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Example: The NFA recognizing the keywords web and ebay



The Formal Notation for an ε - NFA

We may represent an ε - NFA exactly as we do an NFA with one exception: the transition function must include information about transitions on ε. Formally, we represent an ε - NFA A by A = (Q, Σ, **δ**, q_0, F), where all components have their same interpretation as for an NFA, except that **δ** is now a function that takes as arguments:

1. A state in Q, and
2. A member of Σ ∪ { **ε** } that is, either an input symbol, or the symbol **ε**

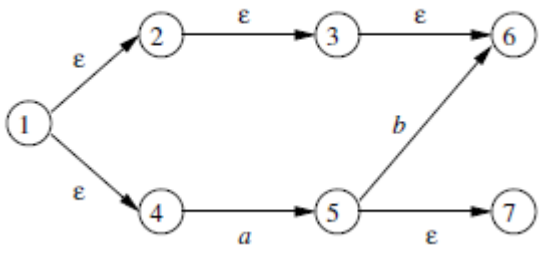
Epsilon_Closures

Informally, we **ε** - close a state q by following all transitions out of q that are labeled **ε.** However, when we get to other states by following **ε,** we follow the **ε** transitions out of those states, and so on, eventually finding every state that can be reached from q along any path whose arcs are all labeled **ε.** Formally, we define the **ε** - closure ECLOSE(q) recursively, as follows:

BASIS: State q is in ECLOSE(q)__

INDUCTION: If state p is in ECLOSE(q) and there is a transition from state p to state r labeled **ε** then r is in ECLOSE(q). More precisely, if **δ** is the transition function of the **ε** - NFA involved, and p is in ECLOSE(q) then ECLOSE(q) also contains all the states in **δ** (p, **ε)**

Example:



ECLOSE(1)={1,2,3,4,6}

Extended Transitions and Languages for ε – NFA's

The ϵ-closure allows us to explain easily what the transitions of an ϵ-NFA look like when given a sequence of (non-ϵ) inputs. From there, we can define what it means for an ϵ-NFA to accept its input.

Suppose that $E = (Q, \Sigma, \delta, q_0, F)$ is an ϵ-NFA. We first define $\hat{\delta}$, the extended transition function, to reflect what happens on a sequence of inputs. The intent is that $\hat{\delta}(q, w)$ is the set of states that can be reached along a path whose labels, when concatenated, form the string w. As always, ϵ's along this path do not contribute to w. The appropriate recursive definition of $\hat{\delta}$ is:

BASIS: $\hat{\delta}(q, \epsilon) = \text{ECLOSE}(q)$. That is, if the label of the path is ϵ, then we can follow only ϵ-labeled arcs extending from state q; that is exactly what ECLOSE does.

INDUCTION: Suppose w is of the form xa, where a is the last symbol of w. Note that a is a member of Σ; it cannot be ϵ, which is not in Σ. We compute $\hat{\delta}(q, w)$ as follows:

1. Let $\{p_1, p_2, \ldots, p_k\}$ be $\hat{\delta}(q, x)$. That is, the p_i's are all and only the states that we can reach from q following a path labeled x. This path may end with one or more transitions labeled ϵ, and may have other ϵ-transitions, as well.

2. Let $\bigcup_{i=1}^{k} \delta(p_i, a)$ be the set $\{r_1, r_2, \ldots, r_m\}$. That is, follow all transitions labeled a from states we can reach from q along paths labeled x. The r_j's are *some* of the states we can reach from q along paths labeled w. The additional states we can reach are found from the r_j's by following ϵ-labeled arcs in step (3), below.

3. Then $\hat{\delta}(q, w) = \text{ECLOSE}(\{r_1, r_2, \ldots, r_m\})$. This additional closure step includes all the paths from q labeled w, by considering the possibility that there are additional ϵ-labeled arcs that we can follow after making a transition on the final "real" symbol, a.

Example : Let us compute $\hat{\delta}(q_0, 5.6)$ for the ϵ-NFA of Fig. A summary of the steps needed are as follows:

- $\hat{\delta}(q_0, \epsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$.
- Compute $\hat{\delta}(q_0, 5)$ as follows:
  1. First compute the transitions on input 5 from the states q_0 and q_1 that we obtained in the calculation of $\hat{\delta}(q_0, \epsilon)$, above. That is, we compute $\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$.
  2. Next, ϵ-close the members of the set computed in step (1). We get $\text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$. That set is $\hat{\delta}(q_0, 5)$. This two-step pattern repeats for the next two symbols.
- Compute $\hat{\delta}(q_0, 5.)$ as follows:
  1. First compute $\delta(q_1, .) \cup \delta(q_4, .) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$.
  2. Then compute

$$\hat{\delta}(q_0, 5.) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$$

- Compute $\hat{\delta}(q_0, 5.6)$ as follows:
  1. First compute $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \emptyset = \{q_3\}$.
  2. Then compute $\hat{\delta}(q_0, 5.6) = \text{ECLOSE}(q_3) = \{q_3, q_5\}$.

Eliminating ε - Transitions

Given any ϵ-NFA E, we can find a DFA D that accepts the same language as E. The construction we use is very close to the subset construction, as the states of D are subsets of the states of E. The only difference is that we must incorporate ϵ-transitions of E, which we do through the mechanism of the ϵ-closure.

Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. Then the equivalent DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

is defined as follows:

1. Q_D is the set of subsets of Q_E. More precisely, we shall find that all accessible states of D are *ϵ-closed* subsets of Q_E, that is, sets $S \subseteq Q_E$ such that $S = \text{ECLOSE}(S)$. Put another way, the ϵ-closed sets of states S are those such that any ϵ-transition out of one of the states in S leads to a state that is also in S. Note that $\emptyset$ is an ϵ-closed set.

2. $q_D = \text{ECLOSE}(q_0)$; that is, we get the start state of D by closing the set consisting of only the start state of E. Note that this rule differs from the original subset construction, where the start state of the constructed automaton was just the set containing the start state of the given NFA.

3. F_D is those sets of states that contain at least one accepting state of E. That is, $F_D = \{S \mid S \text{ is in } Q_D \text{ and } S \cap F_E \neq \emptyset\}$.

4. $\delta_D(S, a)$ is computed, for all a in Σ and sets S in Q_D by:

   (a) Let $S = \{p_1, p_2, \ldots, p_k\}$.

   (b) Compute $\bigcup_{i=1}^{k} \delta_E(p_i, a)$; let this set be $\{r_1, r_2, \ldots, r_m\}$.

   (c) Then $\delta_D(S, a) = \text{ECLOSE}(\{r_1, r_2, \ldots, r_m\})$.

Example: Let us eliminate ε - transitions from the ε-NFA

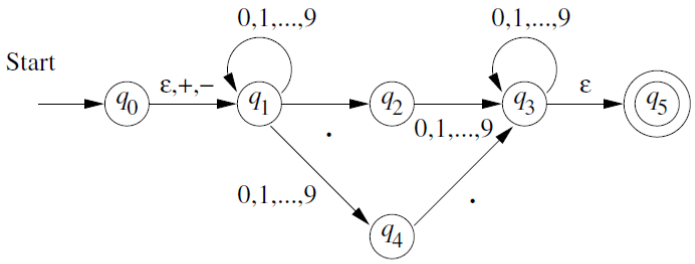


Figure : An ϵ-NFA accepting decimal numbers

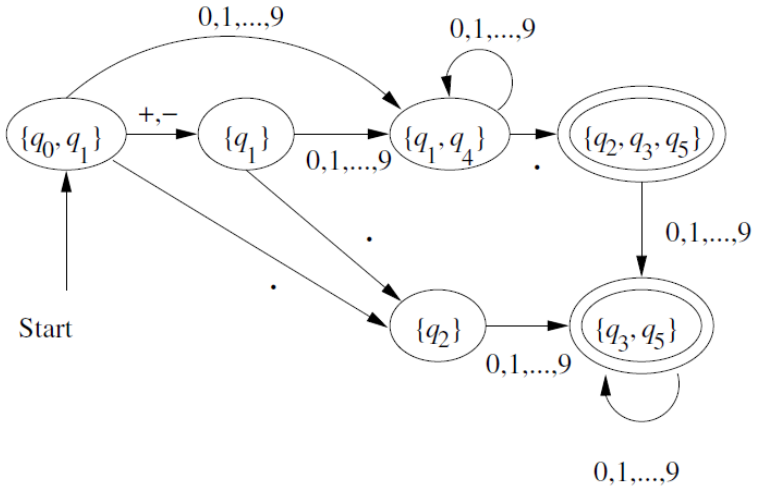


Figure : The DFA D that eliminates ϵ-transitions

Equivalence and Minimization of Automata

In this section, we discuss how to test whether two descriptors for regular languages are equivalent in the sense that they denote the same language. An important consequence of this test is that there is a way to minimize a DFA. That is, we can take any DFA and an equivalent DFA that has the minimum number of states. In fact, this DFA is essentially unique given any two minimum state DFAs that are equivalent, we can always and a way to rename the states so that the two DFAs become the same.

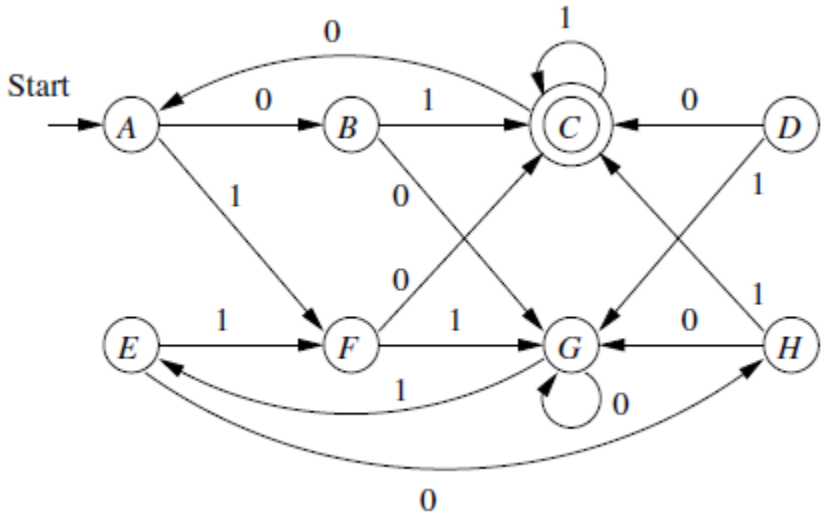
Testing Equivalence of States

When two distinct states p and q can be replaced by a single state that behaves like both p and q the w say that states p and q are equivalent if:

- For all input strings w, $\hat{\delta}(p,w)$ is an accepting state if and only if $\hat{\delta}(q,w)$ is an accepting state.

Less formally, it is impossible to tell the difference between equivalent states p and q merely by starting in one of the states and asking whether or not a given input string leads to acceptance when the automaton is started in this unknown state. Note we do not require that $\hat{\delta}(p,w)$ and $\hat{\delta}(q,w)$ are the same state, only that either both are accepting or both are nonaccepting.

If two states are not equivalent, then we say they are distinguishable. That is, state p is distinguishable from state q if there is at least one string w such that one of $\hat{\delta}(p,w)$ and $\hat{\delta}(q,w)$ is accepting, and the other is not accepting.



To find states that are equivalent, we make our best efforts to find pairs of states that are distinguishable. It is perhaps surprising, but true, that if we try our best, according to the algorithm

to be described below, then any pair of states that we do not find distinguishable are equivalent. The algorithm which we refer to as the table filling algorithm, is a recursive discovery of distinguishable pairs in a DFA A = **(Q,Σ,δ,q0,F)**,

BASIS: If p is an accepting state and q is nonaccepting, then the pair (p,q} is distinguishable.

INDUCTION: Let p and q be states such that for some input symbol a, $r = \delta(p, a)$ and $s = \delta(q, a)$ are a pair of states known to be distinguishable. Then $\{p, q\}$ is a pair of distinguishable states. The reason this rule makes sense is that there must be some string w that distinguishes r from s; that is, exactly one of $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$ is accepting. Then string aw must distinguish p from q, since $\hat{\delta}(p, aw)$ and $\hat{\delta}(q, aw)$ is the same pair of states as $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$.

B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x
	A	*B*	*C*	*D*	*E*	*F*	*G*

However, then we can discover no more distinguishable pairs. The three remaining pairs which are therefore equivalent pairs are {A,E}{B,H} and {D, F}

Minimization of DFAs

Another important consequence of the test for equivalence of states is that we can "minimize" DFAs. That is, for each DFA we can find an equivalent DFA that has as few states as any DFA accepting the same language. Moreover, except for our ability to call the states by whatever names we choose, this minimum state DFA is unique for the language.

The algorithm is as follows:

1. First, eliminate any state that cannot be reached from the start state.
2. Then, partition the remaining states into blocks, so that all states in the same block are equivalent, and no pair of states from different blocks are equivalent.

We are now able to state succinctly the algorithm for minimizing a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

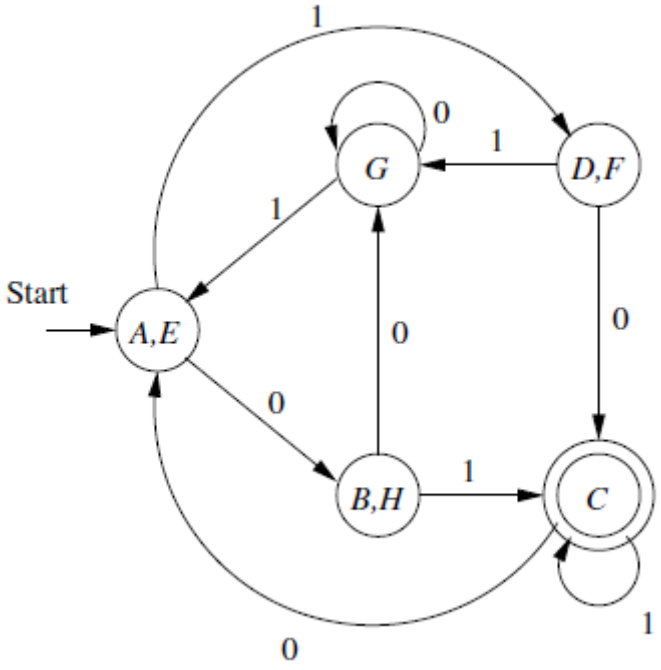
1. Use the table-filling algorithm to find all the pairs of equivalent states.

2. Partition the set of states Q into blocks of mutually equivalent states by the method described above.

3. Construct the minimum-state equivalent DFA B by using the blocks as its states. Let γ be the transition function of B. Suppose S is a set of equivalent states of A, and a is an input symbol. Then there must exist one block T of states such that for all states q in S, $\delta(q, a)$ is a member of block T. For if not, then input symbol a takes two states p and q of S to states in different blocks, and those states are distinguishable
That fact lets us conclude that p and q are not equivalent, and they did not both belong in S. As a consequence, we can let $\gamma(S, a) = T$. In addition:

   (a) The start state of B is the block containing the start state of A.

   (b) The set of accepting states of B is the set of blocks containing accepting states of A. Note that if one state of a block is accepting, then all the states of that block must be accepting. The reason is that any accepting state is distinguishable from any nonaccepting state, so you can't have both accepting and nonaccepting states in one block of equivalent states.



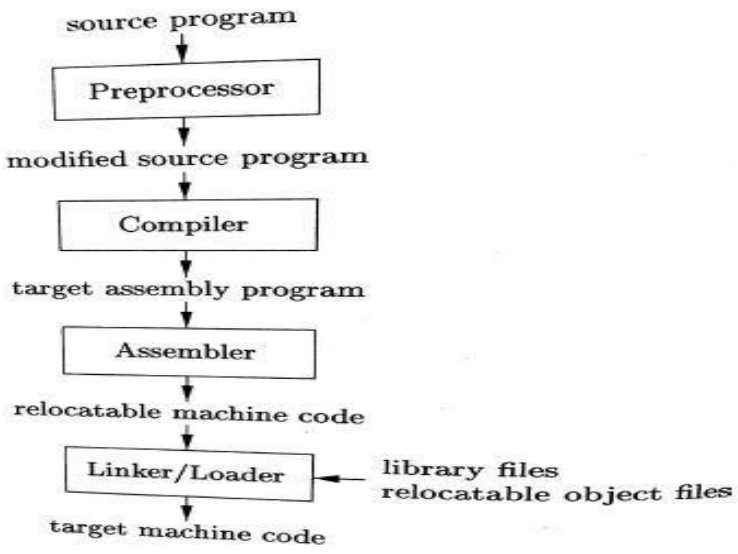
Introduction to Compiler Design

The need for translator/compiler.

The world depends on programming languages. But the computers can understand only machine languages. So, before a program can run, it first must be translated into a form in which it can be executed by a computer. Translations are done by a set of programs called language processors/ translators.

The language processing system.

Language processor can be defined as a system software which converts the program written in assembly level language or high level language into equivalent machine language. Ex: assemblers, compilers, interpreters, loaders/linkers, preprocessors etc.



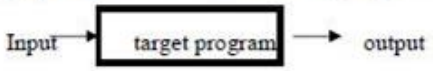
Preprocessor: The large source program may be divided into modules and stored in separate files. The preprocessor collects such modules. The preprocessor may also expand shorthands, called macros, into source language statements.

Compiler/Interpreter: A compiler is a program that can read a program in one language (source language) and translate it into an equivalent program in another language (target language). If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs. An important role of the compiler is to report any errors in the source program.

COMPILER

Pictorial representation of compiler is given below

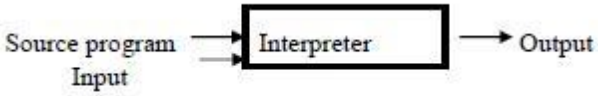
If target program is executable machine language then



Interpreter: An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter directly executes the operations specified in the source program on inputs supplied by the user.

INTERPRETER

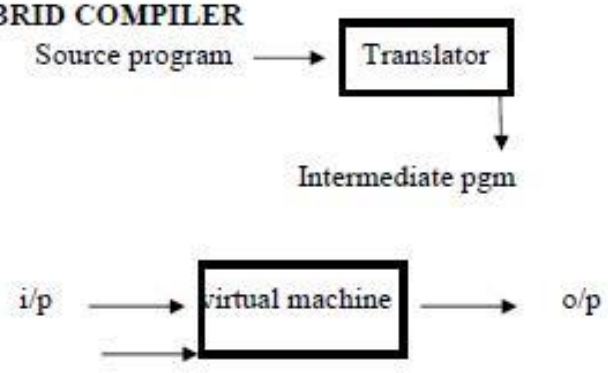
Pictorial representation of an Interpreter is given below



The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement. A language-processing system typically involves – preprocessor, compiler, assembler and linker/loader – in translating source program to target machine code.

Example: Java language processors combine compilation and interpretation as shown in fig. below. A java source program may first be compiled into an intermediate form called bytecodes. The bytecodes are then interpreted by a virtual machine. Bytecodes compiled on one machine can be interpreted on another machine. Java compilers are called **"**just-in-time**"** compilers or hybrid compilers.

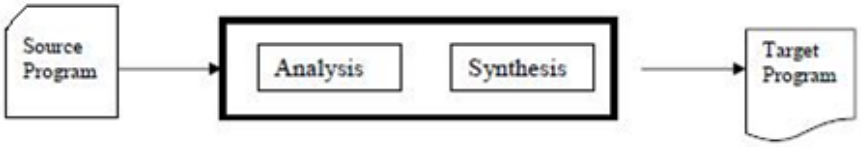
HYBRID COMPILER



Linkers and Loaders: Large programs are compiled in pieces, so re-locatable machine codes may have to be linked together with other re-locatable object files and library files. It is the job of

a linker. It also resolves memory references, where code in one file may refer to a location in other file. The loader then puts together all of the executable object files into memory for execution.

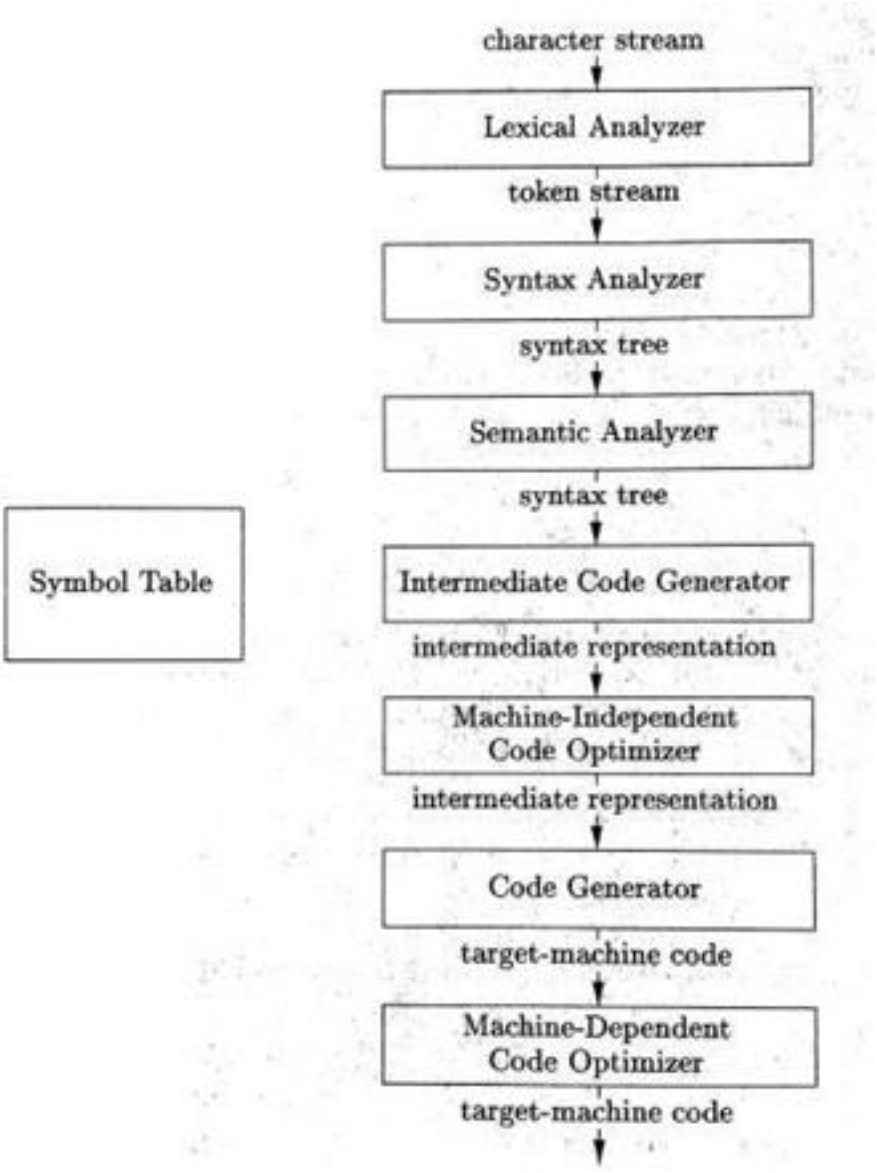
The structure of a compiler



Analysis: source program to intermediate representation (front end)

Synthesis: intermediate representation to target program (back end)

- The *analysis* part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a *symbol table*, which is passed along with the intermediate representation to the synthesis part.
- The *synthesis* part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the *front end* of the compiler; the synthesis part is the *back end*



The phases of a compiler are:

- Lexical analyzer (scanning) (linear analysis),
- Syntax analyzer (hierarchical analysis) (parsing),
- Semantic analyzer,
- Intermediate code generator,
- Machine independent code optimizer,
- Code generator
- Machine-dependent code optimizer

Lexical analysis: The first phase of a compiler is called *lexical analysis* or *scanning.* The lexical analyzer reads the stream of characters making up the source program and groups the characters into

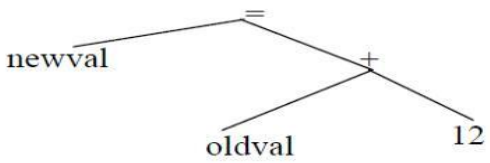
meaningful sequences called *lexemes.* For each lexeme, the lexical analyzer produces as output a *token* of the form *(token-name, attribute-value)* that it passes on to the subsequent phase, syntax analysis. In the token, the first component *token-name* is an abstract symbol that is used during syntax analysis, and the second component *attribute-value* points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

Ex: an assignment statement a= b+c; can be converted into <id,1> <=> <id, 2> <+> <id,3>

Syntax analysis:

The second phase of the compiler is *syntax analysis* or *parsing.* The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation.

Example: newval = oldval +12 can be drawn as:



Semantic analyzer:

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

- Type checking: compilers check that
  - Whether each operator has matching operands?
  - Whether array index is an integer?
  - Type conversion(coercions): binary arithmetic operator may be applied to either pair of integers or pair of floating point numbers. If it is applied to in and float, then int is converted into float.

Intermediate Code Generation: In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during

syntax and semantic analysis. Another IR is 3-address code, whose characteristics are as follows:

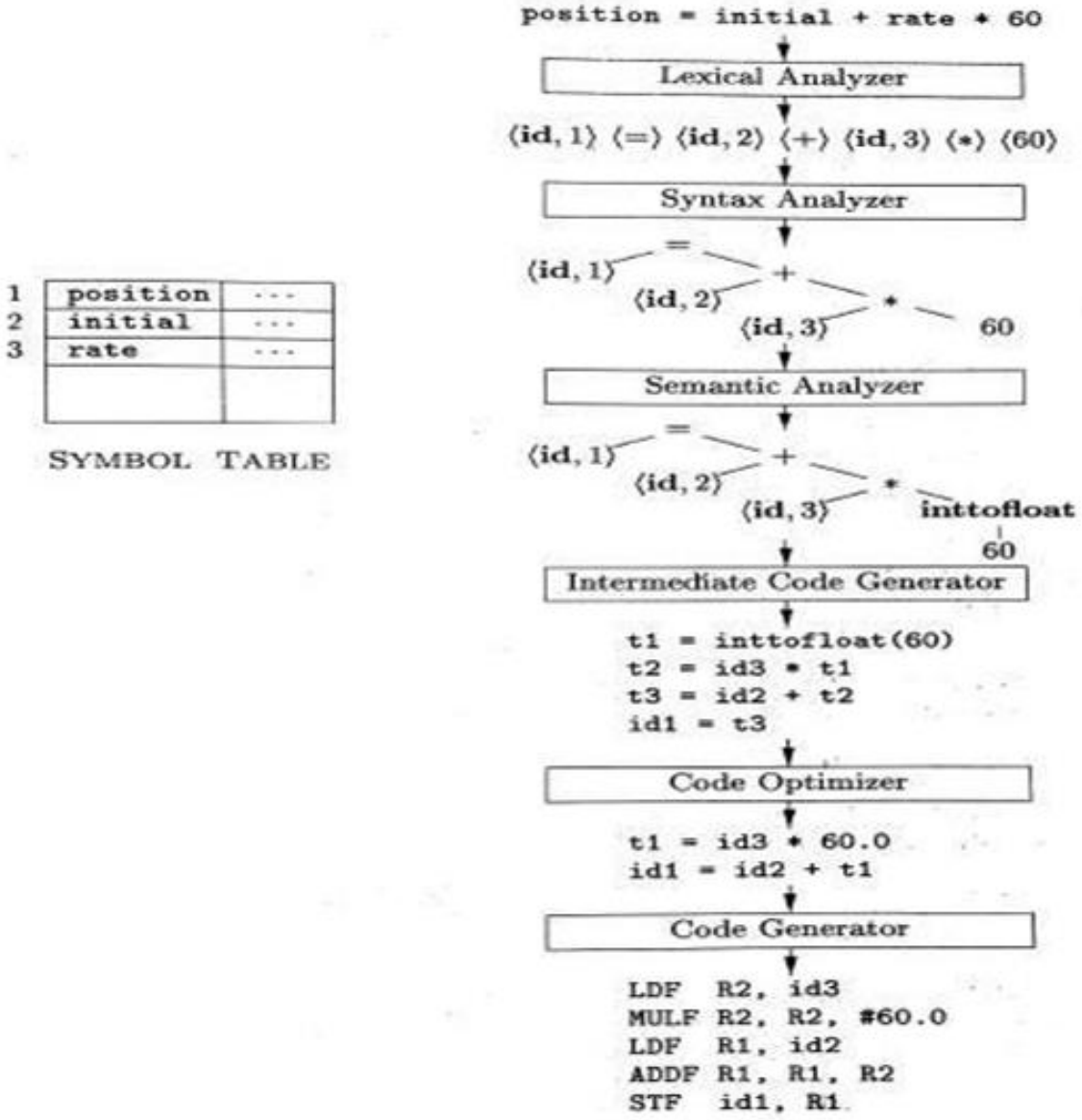
- It has at most one operator on RHS.
- Compiler must generate a temporary name to hold the value computed by a 3 address instruction.

Code Optimization: The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

Code generation: The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

Symbol table manager and error handler are two independent modules which will interact with all phases of compilation. A symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier. When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table. Each phase can encounter errors. After detecting an error, a phase must somehow deal with that error, so that compilation must proceed, allowing further errors in the source program to be detected.

Example:



• **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program. Lexical Analyzer is also called as Scanner. It reads the stream of characters making up

the source program and groups the characters into meaningful sequences called **Lexemes.**

For each lexeme, it produces as output a token of the form

<token_name, attribute_value>

• *token_name* is an abstract symbol that is used during syntax analysis, and the second component *attribute_value* points to an entry in the symbol table for this token.

• **A** *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

Ex: newval := oldval + 12

=> tokens are: newval (identifier)

:= (assignment operator)

oldval (identifier) + (add operator)

12 (a number)

• **Puts information about identifiers into the symbol table not all** attributes. Regular expressions are used to describe tokens (lexical constructs).

Syntax analysis:

A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program. A syntax analyzer is also called as a **parser**. A **parse tree** describes a syntactic structure.

Semantic Analysis:

A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation. It determines meaning of source string.

– Matching of parenthesis

– Matching if else stmt.

– Checking scope of operation

Type-checking is an important part of semantic analyzer where the compiler checks that each operator has matching operands.

Intermediate Code Generation:

A compiler may produce an explicit intermediate codes representing the source program. It is Easy to generate, and easy to convert to target code. It can be in three address code or quadruples, triples etc. Ex:

t1 = inttofloat(60)

t2 = id3*t1

t3 = id2+t2

id1 = t3

Code Optimizer:

The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space. Ex:

t1 = id3*60.0

id1 = id2+t1

Code Generator

Produces the target language in a specific architecture. The target program is normally is a relocatable object file containing the machine codes. If target code is machine code then registers (memory locations) are selected for each variable in program. Then intermediate instructions are translated to sequences of machine instruction which perform same task.

Ex: LDF R2, id3

MULF R2, R2, #60.0

LDF R1, id2

ADDF R1, R1, R2

STF id1, R1

Symbol Table Management

This records variable name used in the source program and collects the information about various attributes of each name. Eg: name, its type, its scope, Method of passing each argument(by value or by reference), Return type. Implementation of symbol table can be done is either linear list or hash table.

Grouping of phases into passes

In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file. Front-end phases of lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. There could be a back-end pass consisting of code generation for a particular target machine.

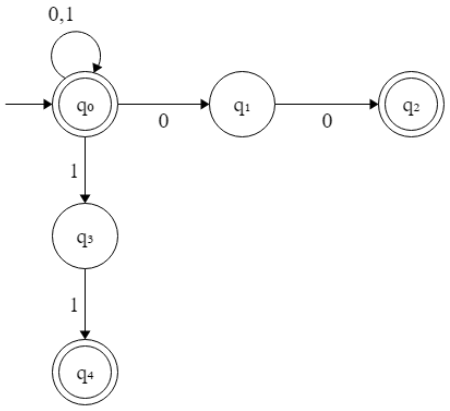
Compiler construction tools

1. Parser generators: automatically produce syntax analyzers from a grammatical description of a programming language.
2. Scanner generators: produce lexical analyzers from a regular expression description of the tokens.
3. Syntax directed translation engines: produce collections of routines for parse tree and generating intermediate code.
4. Code-generator generators: produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. Data-flow analysis engines: facilitate the gathering of information about how values are transmitted from one part of a program to each other part.
6. Compiler construction toolkits: provide an integrated set of routines for constructing various phases of a compiler.

Question Bank

1. Design a DFA to accept each of the following language:
   i) L={ w ε {a,b}* ; w has all strings that ends with sub string abb }
   ii) L={w; where |w| mod 3 = 0 where Σ = {a}}
   iii) L={w ε {a,b}* every a region in w of even length }

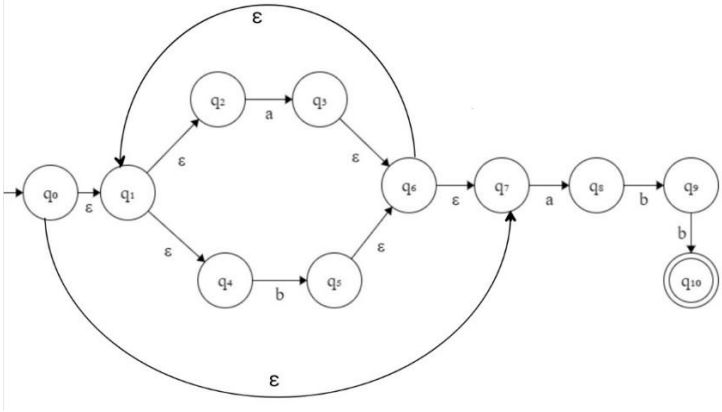
2. Construct an equivalent DFA from the following given NFA using subset construction method



3. Construct a minimum state automation equivalent to the FA given table

State	0	1
→q0	q1	q5
q1	q6	q2
*q2	q0	q2
q3	q2	q6
q4	q7	q5
q5	q2	q6
q6	q6	q4
q7	q6	q2

4. Consider the following NFA with ε-moves construct on equivalent DFA



5. Define DFA. Design a DFA to accept each of the following languages:
   i) L= {wϵ{0,1}* : w has 001 as a substring}
   ii) L={ wϵ{0,1}* : w has even number of a's and even number of b's}

6. Convert the following **ε** -NFA to DFA.

δ	ϵ	a	b	c
->p	{q,r}	{}	{q}	{r}
q	{}	{p}	{r}	{p,q}
*r	{}	{}	{}	{}

7. Define distinguishable and indistinguishable states. Minimize the following DFA.

δ	a	b
->A	B	F
B	G	C
*C	A	C
D	C	G
E	H	F
F	C	G
G	G	E
H	G	C

8. Draw a DFA to accept strings of a's and b's ending with 'bab'.
9. Convert the following **ε** -NFA Fig Q1(c) to its equivalent DFA.

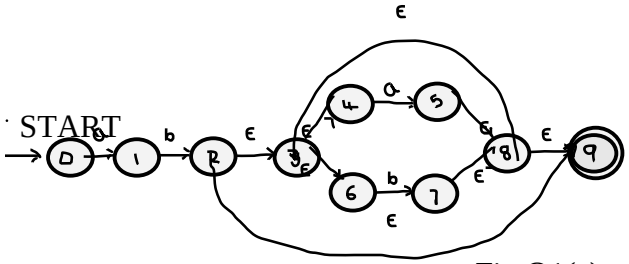


Fig Q1(c)

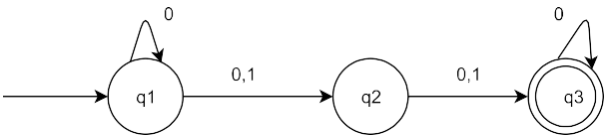
10. Draw a DFA to accept the language,
    L={$\omega \in$ {a,b}* : $\forall$ x,y $\in$ {a,b}*((ω=x abbaay) V (ω=x babay))}
11. Minimize the following DFA.

S	0	1
A	B	A
B	A	C
C	D	B
*D	D	A
E	D	F
F	G	E
G	F	G
H	G	D

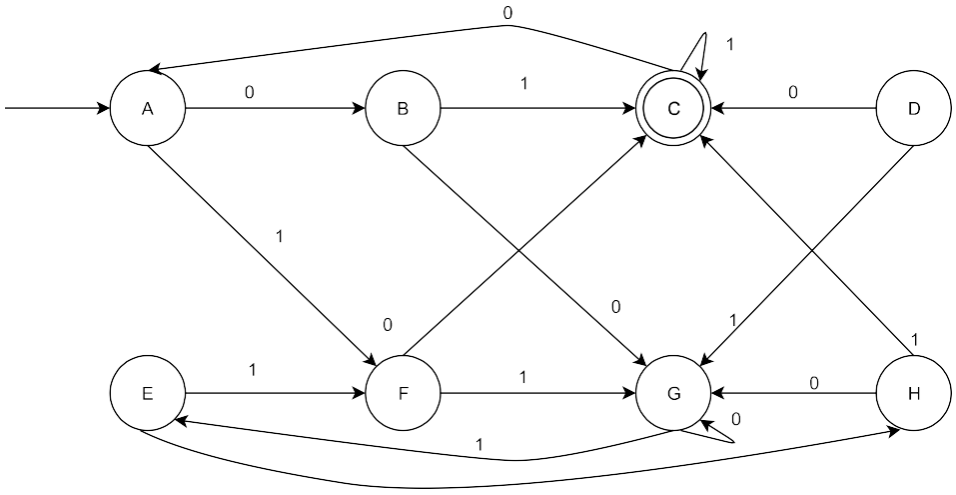
i. Draw tables of distinguishable and indistinguishable state for the automata.
ii. Construct minimum state equivalent of automata.

12. Design a DFA to accept each of the following language!
    i. L = (W e {0, 1} : W has 001 as a substring)
    ii. L = {W e {a, b} : W has even number of a's and even number of b's}.

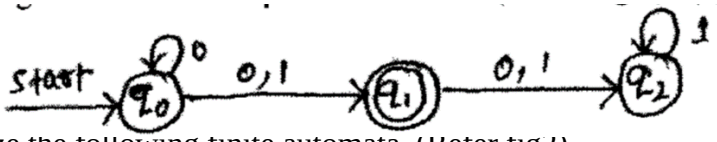
13. Define **ε** -NFA. Convert the following **ε** -NFA to its equivalent DFA.



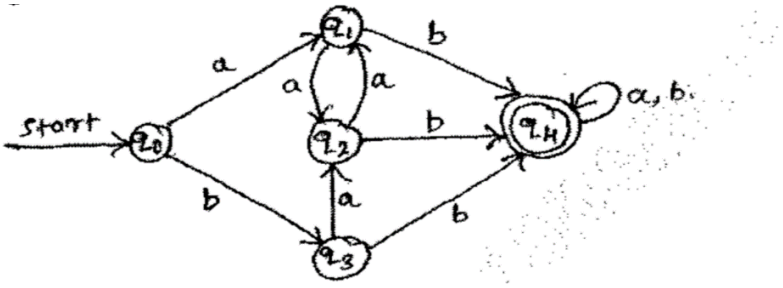
14. Minimize the following DFA



15. Draw a DFA to accept decimal strings which are divisible by 3.
16. Convert the following **ε** -NFA to its equivalent DFA. (Refer fig1).
fig1:



17. Minimize the following finite automata. (Refer fig2).
fig2:



18 a) Define the following: i) string ii) alphabet iii) language
b) Design a deterministic finite state machine for the following language over ∑={a, b}.
i) L= {W| |W| mod 3> |W| mod 2}
ii) L= {W| W ends either with ab or ba}

19 Define DFA. Minimize the following FSM.[refer fig Q2(b)]

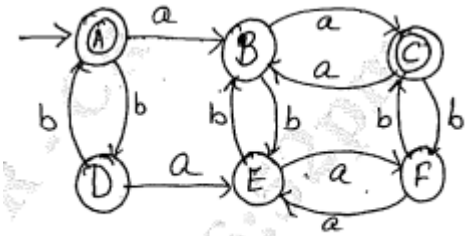


Fig.Q2(b)

20. Write the DFA's for the following languages over ∑={a,b}:
    i)The set of all strings ending with abb
    ii)The set of all strings not containing the substring aab
    iii) L= {a w a| w € (a + b)*}
    iv) L= {w | |w| mod3 =0}
21. Convert the following NFA to its equivalent DFA.

	ϵ	a	b
->p	{r}	{q}	{p,r}
q	Φ	{p}	Φ
r	{p,q}	{r}	{p}
*s	{p}	{p}	{p}

22. Design an ϵ-NFA for the regular expression (a + b)* ab.
23. Design a DFA which accepts all strings of 0's and 1's, beginning with a 1 that, when interpreted as a binary integer, is a multiple of 5. For example, strings 101,1010 and 1111 are in the language: 0,100,0101,111 are not.

24. Convert the following NFA to DFA using construction method:

δ	0	1
->p	{p,q}	{p}
q	null	{r}
*r	{p,q}	{q}

25. a. Consider the following ϵ-NFA

δ	∈	a	b
->p	{r}	{q}	{p,r}
q	null	{p}	null
*r	{p.q}	{r}	{p}

1. compute the e-closure for each state.
2. Give the set of all strings of length 3or less accepted by automation
3. Convert the automation to DFA.

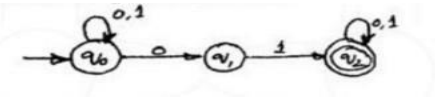
31. Minimize the DFA using table filling algorithm.

δ	0	1
->q1	q2	q3
q2	q3	q5
*q3	q4	q3
q4	q3	q5
*q5	q2	q5

32.Design a DFA to accept a string of a's and b's not ending with abb.

33.Design a DFA which accepts odd number od 0's and odd number of 1's.

34.Convert the following NFA to DFA.



35.Design DFA's for the following languages

a) Set of all strings with atleast one a's and exactly two b's on ∑ = {a,b}.

b) Set of all strings such that numbers of 1's is even and the number of 0's is a multiple of 3 and ∑ = {0,1}

36.Design an NFA with no more than 5 states for the following languages:

$L=\{aba^n \mid n\geq 0\} \cup \{ab^n \mid n\geq 0\}$

37. Convert the following € -NFA into an equivalent DFA :

δ	€	a	b	c
→p	{q,r}	ϕ	{q}	{r}
*q	ϕ	{p}	{r}	{p,q}
r	ϕ	ϕ	ϕ	ϕ

38.Define regular expression and also write the regular expression for the following languages:

a) L= {w€ {a,b}* |w has exactly one pair of consecutive a's}.

b) Set of all strings not ending in substring 'ab' over ∑ = {a,b}.

39 a. Write the DFAs for the following languages over ∑={a,b}

(i) The set of all strings ending with a and b.

(ii) The set of all strings not containing the substring aab.

(iii) Set of all strings with exactly three consecutive a's.

b. Define NFA. Convert the following NFA to its equivalent DFA.

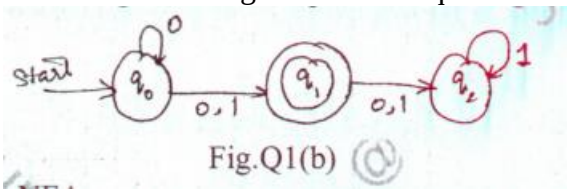


Fig.Q1(b)

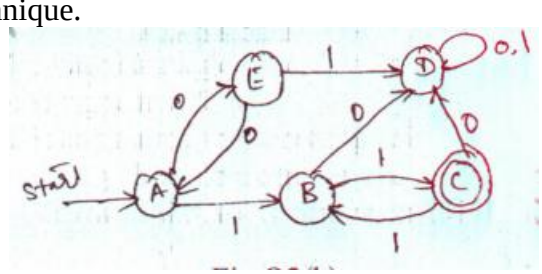
40 a. Consider the following ε - NFA:

	∈	a	b	c
→p	ϕ	{p}	{q}	{r}
q	{p}	{q}	{r}	ϕ
* r	{q}	{r}	ϕ	{p}

(i) Compute the ε - closure of each state.

(ii) Convert the ε - NFA to DFA.

b. Define Regular expression. Convert the following automation to a regular expression using state elimination technique.



41: Explain the need for translator/compiler.

42: With neat block diagrams, explain language processing system.

43: Explain the different phases of a compiler, with an example.

44: Explain different compiler construction tools