Introduction to Compiler

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called ***compilers.***
This COURSE is about how to design and implement compilers. We shall discover that a few basic ideas can be used to construct translators for a wide variety of languages and machines. Besides compilers, the principles and techniques for compiler design are applicable to so many other domains that they are likely to be reused many times in the career of a computer scientist.

The study of compiler writing touches upon programming languages, machine architecture, language theory, algorithms, and software engineering. In this preliminary chapter, we introduce the different forms of language translators, give a high level overview of the structure of a typical compiler, and discuss the trends in programming languages and machine architecture that are shaping compilers. We include some observations on the relationship between compiler design and computer-science theory and an outline of the applications of compiler technology that go beyond compilation. We end with a brief outline of key programming-language concepts that will be needed for our study of compilers.

Compiler *is also known as* ***Compiler Design (CD) / Language Processor (LP) / Compiler Construction (CC)***

Course Aims & Objectives:

1. Introduce students to the concepts of design and implementation of language processors. More specifically, by the end of the course, students will be able to answer these questions:
   - What language processors are, and what functionality do they provide to their users?
   - What core mechanisms are used for providing such functionality?
   - How are these mechanisms implemented?

2. Apart from providing a theoretical background, the course places a special emphasis in practical issues in designing language processors. Through labs and the assessed project, students will learn to:
   - Design and implement language processors in C/C++
   - Use tools to automate parts of the implementation process.

Notes Prepared By. Ahmad Sir
Contact: 9867597554

Course overview:

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System, shown in fig.

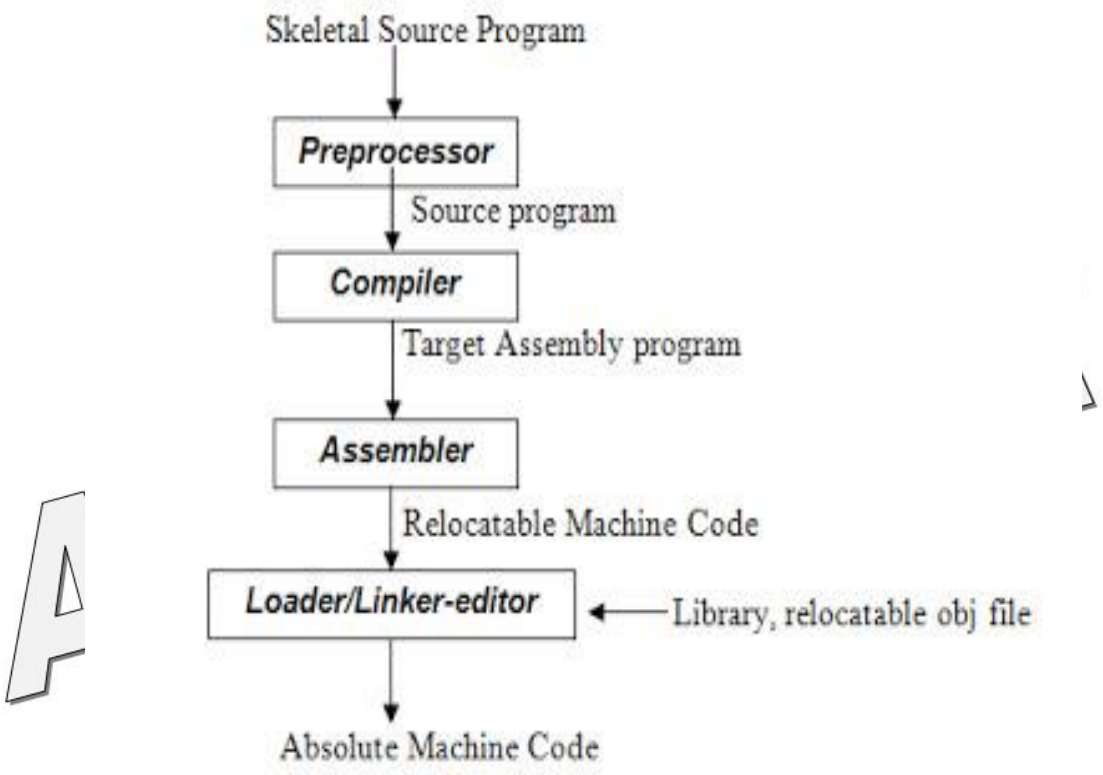


Fig 1.1 Language –processing System

Language Processing System:

Based on the input the translator takes and the output it produces, a language translator can be called as any one of the following.

Preprocessor:

A preprocessor takes the skeletal source program as input and produces an extended version of it, which is the resulted as expanding the Macros, manifest constants if any, and including header files etc. For example, the C preprocessor is a macro processor that is used automatically by the C compiler to transform our source before actual compilation. Over and above a preprocessor performs the following activities:

- Collects all the modules, files in case if the source program is divided into different modules stored at different files.
- Expands short hands / macros into source language statements.

Compiler:

Is a translator that takes as input a source program written in high level language and converts it into its equivalent target program in machine language. In addition to above the compiler also

- Reports to its user the presence of errors in the source program.
- Facilitates the user in rectifying the errors, and execute the code.

Assembler:

Is a program that takes as input an assembly language program and converts it into its equivalent machine language code.

Loader/Linker:

This is a program that takes as input a relocatable code and collects the library functions, re locatable object files, and produces its equivalent absolute machine code.
Specifically the **Loading** consists of taking the re locatable machine code, altering the re locatable addresses, and placing the altered instructions and data in memory at the proper locations.

Linking allows us to make a single program from several files of re locatable machine code. These files may have been result of several different compilations, one or more may be library routines provided by the system available to any program that needs them.

SYLLABUS

Computer Science and Engineering (CSE)

UNIT I:
Introduction: Phases of compilation and overview. Lexical Analysis (scanner): Regular languages, finite automata, regular expressions, relating regular expressions and finite automata, scanner generator (lex, flex).

UNIT II:
Syntax Analysis (Parser): Context-free languages and grammars, push-down automata, LL(1) grammars and top-down parsing, operator grammars, LR(O). SLR(1), LR(1), LALR(1) grammars and bottom-up parsing, ambiguity and LR parsing, LALR(1) parser generator (yace, bison)

UNIT III:
Semantic Analysis: Attribute grammars, syntax directed definition, evaluation and flow of attribute in a syntax tree. Symbol Table: Basic structure, symbol attributes and management. Runtime environment: Procedure activation, parameter passing, value return, memory allocation.

UNIT IV:
Intermediate Code Generation: Translation of different language features, different types of intermediate forms. Code Improvement (optimization): control-flow, data- flow dependence etc.; local optimization, global optimization, loop optimization, peep-hole optimization etc.

UNIT V:
Architecture dependent code improvement: instruction scheduling (for pipeline), loop optimization (for cache memory) etc. Register allocation and target code generation. Advanced topics: Type systems, data abstraction, compilation of Object Oriented features and non-imperative programming languages.

GATE SYLLABUS 2020

Compiler Design

1. Lexical analysis
2. Parsing
3. Syntax-directed translation (SDTS)
4. Runtime environments
5. Intermediate code generation.

UNIT-I

INTRODUCTION TO COMPILER

SYLLABUS:

1. Introduction :
   - Translators
   - Compilers
   - Phases of Compilation and overview.
2. Lexical Analysis (scanner):
   - Regular languages
   - Finite automata
   - Regular expressions
   - Relating RE and FA
   - Scanner generator (lex / flex)

1. INTRODUCTION TO COMPILER:

➢ Translator:

A program that takes as input a program written in one programming language (Source Program) and produces as output a program written in another language (Object or Target Program) is called as translator.

Fig: Translator

If the Source Program is a high level language such as FORTRAN, COBOL and the Object Program is a low level language such as Assembly language, then such Translator is called as Compiler.

➢ Compiler:

Compiler is a program that read the Source program written in high level language that converted into Object program written in low level language and also gives an error messages if any errors encountered in a Source Program.

A program written in high level language is basically executed in two steps, as shown below

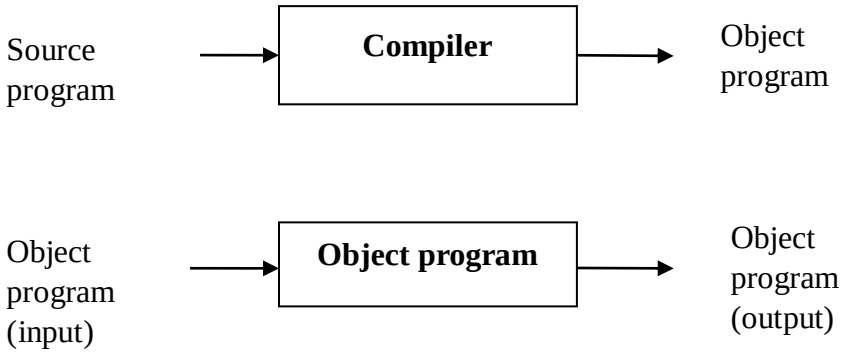


Fig. compilation and execution

- In first step the source program is compiled i.e. translated into object program.
- In second step the resulting object program is loaded into memory and executed.

Notes Prepared By. Ahmad Sir
Contact: 9867597554

➢ Phase of compilation and overview:

The process of compilation is very complex hence the compilation process is partitioned into series of sub-process called phases.

Phase:

A phase is a logically cohesive operation that takes input from one representation of the source program and produces as output for another representation.

Eg. Lexical analysis, syntax analysis, intermediate code generation, code optimization are all phases of compiler.

Pass:

In the implementation of the compiler **one or more phases** can be **combined into a module** called a pass.

The number of passes and grouping of phases into passes are usually dictated by particular language and machine.

There are following factors effects the number of passes:

- Source language
- Environment in which the compiler must operate.

A **multi-pass compiler** can also be used. It uses less space than a **single pass compiler,** since the space occupied by the compiler program for one pass can be reused by next pass.

A multi-pass compiler is slower than single-pass compiler, because each pass reads and writes an intermediate file. Thus, compilers running on computers with small memory would normally use several pass while, on the computer with large random access memory, a compiler with fewer passes would be possible.

Difference between Phase and Pass:

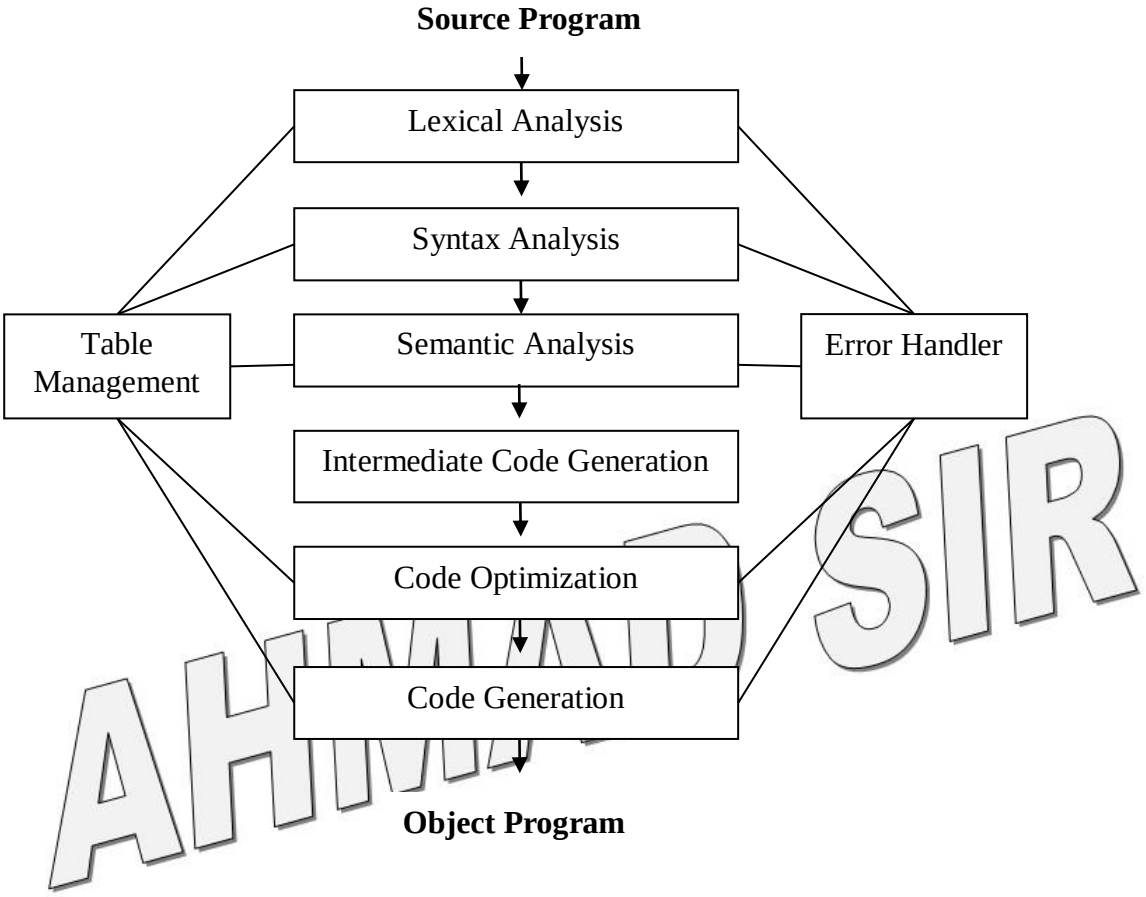
- In compilers, the process of compilation can be carried out with the help of various phases such as lexical analysis, syntax analysis, intermediate code Generation, code generation and code optimization.
- Whereas in case of passes, various phases are combined together to form a Single pass. An input program can be compiled using a single pass or using Two passes.

- Different groups of phase form corresponding front end and back ends of the compilers. The advantage of front-end and back-end model of compiler is That: same source language can be compiled on different machines or Several different languages can be compiled on the same machine.

- Compiling a source program into single pass is difficult because of forward Reference that may occur in the program. Similarly if we group all the phases Into a single pass then we may be forced to keep the entire program in the Memory. And then memory requirement may be large. On the other hand it Is desirable to have relatively few passes, because it becomes time consuming To read and write intermediate files

Phases of compiler:

A typical decomposition of process of compilation into phases is shown below:



1. Lexical Analysis:

Lexical analysis is the first phase of compiler which reads the character from source program and group them into the stream of token.

Token:
Token can be defined as the group of character having logical meaning such as identifiers, operators, keyword, constants, special symbol etc.

Lexeme:
Lexeme can be defined as the character sequence corresponding to the token.

Pattern:
Pattern can be defined as the set of rules to define a token.
The output of the lexical analyzer, is a stream of token which passed to the next phase i.e. syntax analysis.

2. Syntax and Semantic Analysis:

The syntax analyzer groups token together into syntactic structure. In this phase, a compiler verifies, whether the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language or not.

If the string of tokens are grouped according to rules of syntax, then the group of tokens generated by the lexical analyzer is accepted as the valid construct of language.
Syntax analyzer construct a parse tree with leaf nodes as token.
It checks if the expression is syntactically correct. Make hierarchical structure.

3. Intermediate Code Generation:

It transforms the parse tree into an intermediate language representation or source program. By using Three Address Code

Three address code (TAC):

A typical TAC statement is A = B op C
Where A, B and C are operands and op is binary operator.
Consider an expression: $a + b * c$
The three address code of above expression will be

$$t_1 = b * c$$
$$t_2 = a + t_1$$

Where t_1 and t_2 are name of temporary variables

4. Code Optimization:

The optimization phase is an optional phase which is designed to improve the intermediate code so that the object program runs faster or takes less space.

A good optimizing compiler can improve the target program by perhaps a factor of two in overall speed as compared to the compiler that generates code carefully but without using the specialized techniques referred as code optimization.

5. Code Generation:

The code generation phase converts the intermediate code into sequence of machine instructions. For example, the machine code of sequence of statement

$$x = a + b$$
LOAD a
ADD b
STORE x

6. Table Management (Symbol Table):

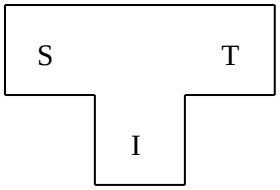
It keeps track of the names used in the program and essential information about each such as its type, value etc.
Data structure used to record this information is called as symbol table.

7. Error Handler:

This phase finds the errors in the source program and reports the errors.
The error message issues an appropriate diagnostic message, which allows the programmer to determine exactly what the error is.

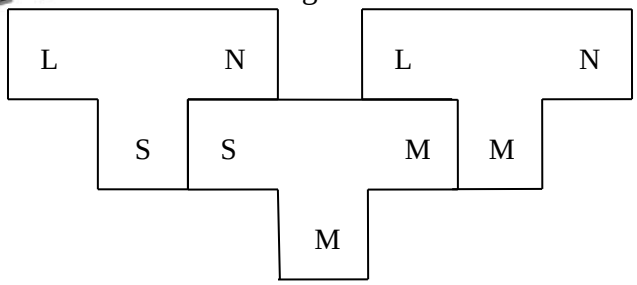
➢ Cross Compiler:

Cross compiler is an compiler which runs on one machine and produces an object code for another machine. A cross compiler is used for implementing a compiler itself.
Thus by using cross compilation technique platforms independency can be achieved.
Basically a compiler is characterized by three languages:
1. Source language
2. Implementation language
3. Target language
Cross compiler is represented as T diagram shown in fig:



Where S -Source language
I- Implementation language
T- Target language

Bootstrapping:
The arrangement in which a compiler is implemented in its own language is called as ***Bootstrap Arrangement***. The implementation is shown in fig



Suppose we want to write a cross compiler for new language L. The implementation language of this compiler is say S and the target code being generated is in language N. That is, we create LSN. Now if existing compiler S runs on machine M and generates code for M then it is denoted as SMM. Now if we run LSN using SMM then we get a compiler LSM. That means a compiler for source language L that generates a target code in language N and which runs on machine M.

For example, a **compiler** that runs on a Windows 7 PC but generates code that runs on Android smart phone is a **cross compiler**.

Notes Prepared By. Ahmad Sir
Contact: 9867597554

2. Lexical Analysis (scanner):

➢ Regular Languages (RL):

A Language is defined as the collection of strings over alphabet. The language accepted by the finite automata is called as regular language. It may be finite or infinite

Operations on languages:

Because languages are sets of strings, new languages can be constructed using the set operations. There are various possible operations are Union, Intersection and difference are also languages over Σ .besides these operations some more operations can also be performed on languages.

- **Concatenation:**

The concatenation AB of languages A and B is defined by A.B = { (uv) | u ∈ A and v ∈ B }
Example:
Let A = {a, b} B = {aa, ab, ba, bb}
A.B = {aaa, aab, aba, abb, baa, bab, bba, bbb}

- **Closure operations:**

There are two types of closure operations-

1. Star closure (kleene*):
Star closure is defined as:
$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots\ldots\ldots$
Example: L = {a, b}
{a,b}* = { ∈} ∪{a, b} ∪{aa, ab, ba, bb}
= {∈, a, b, aa, ab, ba, bb ……..}
L* is the infinite language which contains the all possible string over {a, b}.

2. Positive closure:
Positive closure is defined as:
$L^+ = L^* - \{ \in \}$
$= L^1 \cup L^2 \cup \ldots\ldots\ldots$
= {∈, a, b, aa, ab, ba, bb……} – { ϵ}
= { a, b, aa, ab, ba, bb ……..}
L^+ is the infinite language which contains the all possible string over {a, b} excluding with ϵ.

➢ Finite Automata (FA):

Finite Automata is a **Mathematical Model** of system with Finite inputs and outputs.
Finite Automata is called as Finite because number of possible states and number of inputs alphabets are both Finite, and Automata change the states when inputs are applied.
Finite Automata is a basic computational model, which will translate the inputs into outputs. It consists of finite sets of states, i/p alphabets and set of transitions.
It is the **Finite Representation** of the infinite or finite languages.
It contain **Limited** amount of Memory, hence it cannot perform any Mathematical operations.

Structure of Finite State Machines:

In Finite States Machine, the internal states of the Machine alters when the machine receives and input and generate the required output.

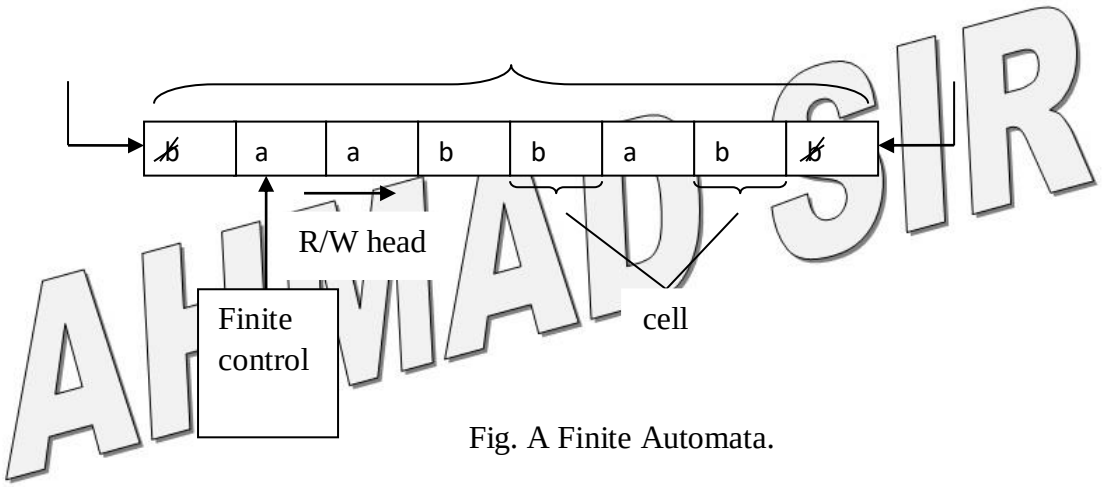


Fig. A Finite Automata.

It consists of three components:

1. **Finite input tape:**
It is used to store all the inputs symbol, at left end marker to right end marker. end marker are generally denotes as by three symbol b.

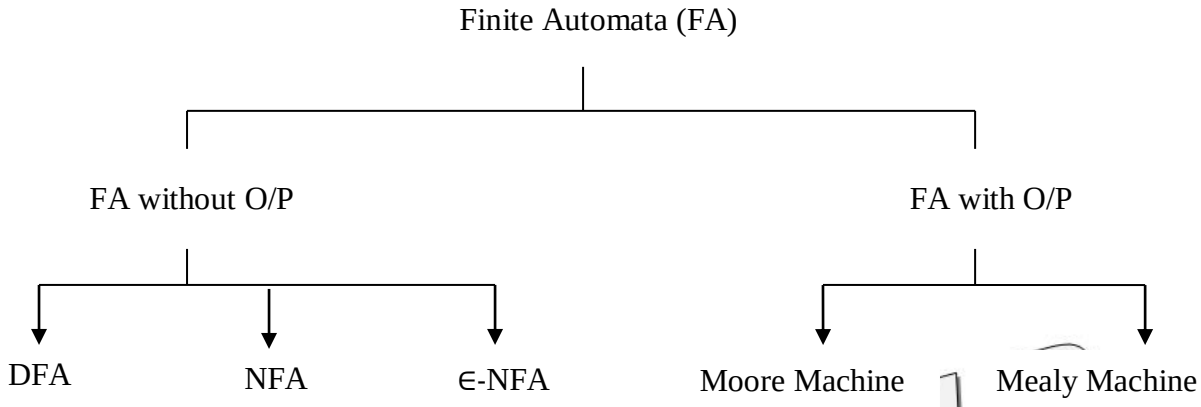
2. **Read/ Write head:**
It checks one input symbol at a time and move only one direction from left to right.

3. **Finite control:**
It stores one input symbol at a time, then read/ write head check as scan that symbol and goes towards next symbol. If it will get end number to scan, they say that scanning will be completed.

Notes Prepared By. Ahmad Sir
Contact: 9867597554
ahmadsircse.com

Types of Finite Automata:

Finite automata is classified in two categories shown in figure



1. Finite Automata without Output:

In this types of automata, is used to decides whether the input is accepted or rejected. It is also called as accepter. It cannot generate any outputs.

There are following three types of automata-

1.1. Deterministic Finite Automata (DFA):

There is a single (unique) transition from one state to other with a single input alphabet, then that Automata is called as Deterministic Finite Automata (DFA). For DFA, each and every state should contain exactly one out degree for each input symbol of alphabet.

It is represented as 5 tuples :

$$\mathbf{M = (Q, \Sigma, \delta, q_o, F)}$$

Where,

$\boldsymbol{Q}$ → Set of finite states

$\boldsymbol{\Sigma}$ → Set of input alphabet

$\boldsymbol{q_o}$ → Initial state

$\boldsymbol{F}$ → Set of final states

$\boldsymbol{\delta}$ → Transition function or mapping functions

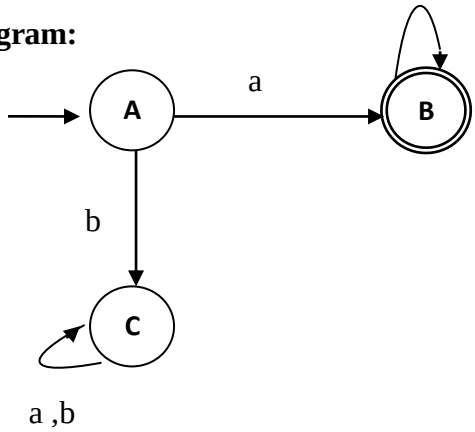
defined as $\boldsymbol{\delta}$: $\boldsymbol{Q \times \Sigma \rightarrow Q}$

Consider the following example:

DFA for starting with a over Σ = {a,b}

Representation of DFA:

Transition diagram:



Q = {A, B, C} $q_o = A$ F = { B } Σ = { a,b}

Transition function:

$$\delta: Q \times \Sigma \to Q$$

$\delta(A,a) \to B$, $\delta(A,b) \to C$

$\delta(B,a) \to B$, $\delta(B,a) \to B$

$\delta(C,a) \to C$, $\delta(C,a) \to C$

Transition Table :

States \ Σ	a	B
→A	B	C
*B	B	B
C	C	C

Design procedure of DFA:

To follow the following steps-

1. First of all we have to ***analyze*** the language for the set of sting.

2. Make sure that there is only ***one initial*** state and at least ***one final state.***

3. Complete the ***out degree*** for each and every state for each input alphabet must be ***single.***

Consider the example:

Design a DFA which accept the language over $\Sigma = \{0, 1\}$

$$L = (01)^{i}1^{2j} \quad | \, i \geq 1, j \geq 1$$

Solution:

First analyze the language

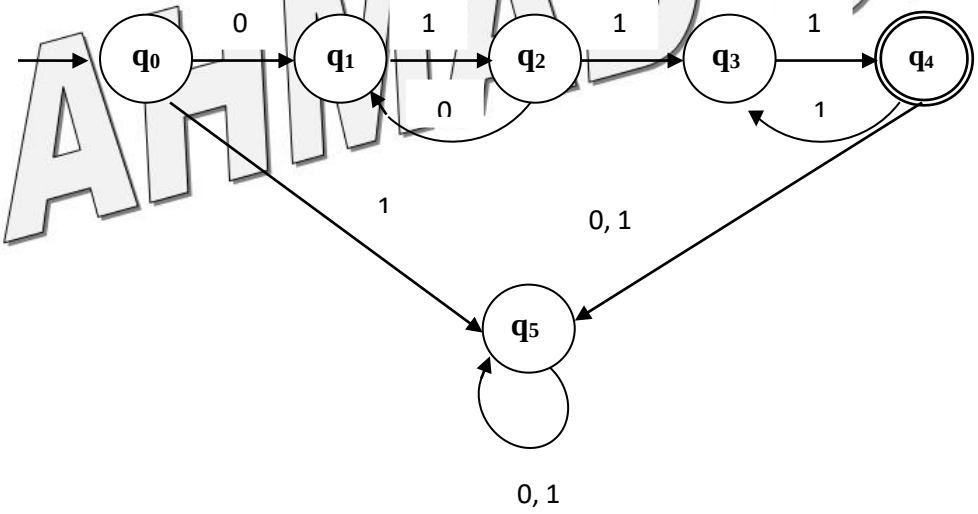
For i = 1,j = 1 (initial condition) , L = 0111

i = 2,j = 1 , L = 010111

i = ,j = 2 , L = 011111

We have to analyze that language always start with' 0' and end with '1', and i is the multiple of '01' and j is multiple of '11'.

Now construct the DFA from initial condition



Acceptance of DFA:

1. String Acceptance:
Scan the entire string if we reach a final state from initial state that we can say accept the sting.

2. Language acceptance:
A DFA is said to be accept a language if all the string in the language are accepted and all the string not in the language are rejected.

1.2. Non Deterministic Finite Automata (DFA):

There are zero's or more no. of transition from one state to other with single input symbol then that Automata is called as NFA.

It is represented as 5 tuple

$$\mathbf{M = (Q, \Sigma, \delta, q_o, F)}$$

Where,

$\boldsymbol{Q}$ → Set of finite states

$\boldsymbol{\Sigma}$ → Set of input alphabet

$\boldsymbol{q_o}$ → Initial state

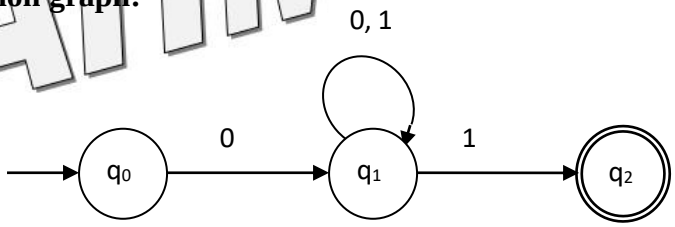
$\boldsymbol{F}$ → Set of final states

$\boldsymbol{\delta}$ → Transition function or mapping functions

defined as $\boldsymbol{\delta}: \boldsymbol{Q \times \Sigma \rightarrow 2^Q}$

Consider the example:

1. Transition graph:



Q = {q_0, q_1, q_2}

$\boldsymbol{\Sigma}$ = { a, b}

$\boldsymbol{q_o}$ = q_0

$\boldsymbol{F}$ = { q_2 }

2. Transition Function:

$\boldsymbol{\delta}: \boldsymbol{Q \times \Sigma \rightarrow 2^Q}$

{q_0, q_1, q_2} × { a, b} → **{** ∅,{q_0}, {q_1}, {q_2}, {q_0 , q_1 }, {q_0,q_2},{q_1, q_2},{q_0, q_1, q_2}

(q_0 , 0) → q_1 , (q_0 , 1) → ∅

(q_1 , 0) → q_1 (q_1 , 1) → **{**q_1, q_2}

(q_2 , 0) → ∅ (q_2 , 0) → ∅

Transition table:

States \ Σ	0	1
→ q_0	q_1	∅
q_1	q_1	{q_1, q_2}
*q_2	∅	∅

Design procedure of NFA:

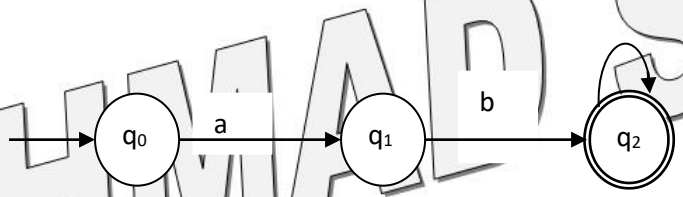
To follow the following steps-

1. First of all we have to ***analyze*** the language for the set of sting.

2. Make sure that there is only ***one initial*** state and at least ***one final state.***

Example: construct an NFA which contain substring 'ab' over Σ = {a, b}

Sol^n:



Note : The design of NFA is easy as compare to the DFA. The real world problems are based on DFA .

NFA can be help as to find the best solutions from the set of possible solutions.

1.3. Equivalence between NFA and DFA

NFA to DFA Conversion:

Steps for conversion of NFA to DFA are as follows-

Step 1. DFA (Σ **)** = NFA (Σ)

Step 2. DFA (q_0) = NFA (q_0)

Step 3. Find $Q^1 = 2^Q$ ie. set of all possible subset.

Step 4. Find δ^1 is the transition of DFA

The transition will start from initial state [q_0] in the given NFA. The resultant states will be considered as valid if it is presented in the power set. The transition generation process is assumed to be completed when no new states is left as unprocessed.

Step5. Defining new final states of DFA

All the subset which contain initial final state ie. Final state of given NFA.

Example:

Convert NFA to DFA, It is defined by

NFA = ({p, q, r, s}, {0,1}, δ, p, {s})

And δ is given by

States \ Σ	0	1
→p	p, q	p
q	r	r
r	s	∅
*s	s	s

Solution:

Step 1. DFA (Σ) = NFA (Σ)

Step 2. DFA (p) = NFA (p)

Step 3. Find $Q^1 = 2^Q$

Therefore $2^4 = 16$

Q^1 = {∅, {p}, {q},{r},{s},{p, q}, {p, r}{p, s}, {q, r}, {q, s}, {r, s}, {p, q, r},{p, r, s},{p, q, s}

, {q, r, s},{p, q, r, s}}

Step 4. Find δ^1 is the transition of DFA

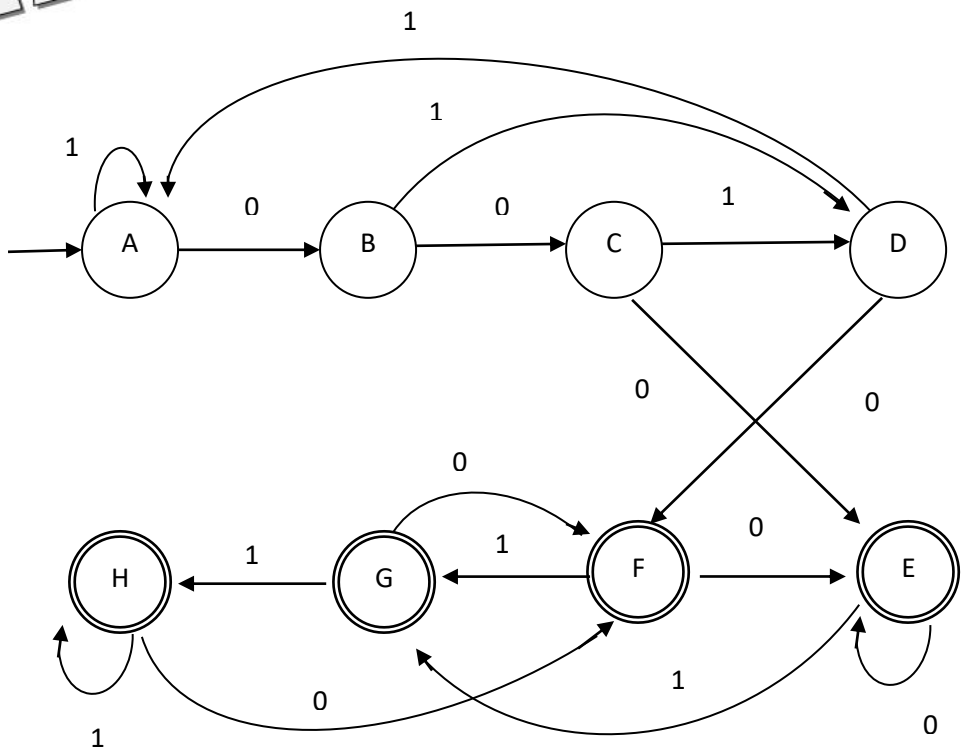
States \ Σ	0	1
→[p]	[pq]	[p]
[pq]	[pqr]	[pr]
[pqr]	*[pqrs]	[pr]
[pr]	*[pqs]	[p]
*[pqrs]	*[pqrs]	*[prs]
*[pqs]	*[pqrs]	*[prs]
*[prs]	*[pqs]	*[ps]
*[ps]	*[pqs]	*[ps]

Step 5. Defining final states and renaming :

A= [p], B = [pq], C = [pqr], D = [pr], *E= [pqrs], *F= [pqs], *G = [prs], *H = [ps]

States \ Σ	0	1
A	B	A
B	C	D
C	*E	D
D	*F	A
*E	*E	*G
*F	*E	*G
*G	*F	*H
*H	*F	*H

Transition diagram:



1.4. NFA with ∈- transitions:

An NFA is allowed to make a transition spontaneously, without receiving an input symbol is called as NFA with ∈ transitions.

It is defined by 5 tuples

M = (Q, Σ, δ, q_0, F)

Where,

Q → Set of finite states

Σ → Set of input alphabet

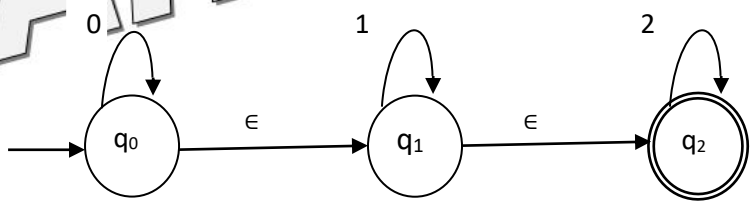
q_o → Initial state

F → Set of final states

δ → Transition function or mapping functions

defined as $\delta: Q \times \Sigma \cup \{\in\} \rightarrow 2^Q$

Consider an example:



$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1, 2\}$

$q_o = q_o$

$F = \{q_2\}$

δ → Transition function or mapping functions

defined as $\delta: Q \times \Sigma \cup \{\in\} \rightarrow 2^Q$

States\ Σ	0	1	2	∈
q_0	$\{q_0\}$	-	-	$\{q_1\}$
q_1	-	$\{q_1\}$	-	$\{q_2\}$
q_2	-	-	$\{q_2\}$	

1.5 ∈-Closure:

It is the set of all states p such that there is a path from q to p labeled " ∈ " i.e. ∈-closure of any states is state itself and all the states which have path from given states labeled ∈

For every NFA with ∈ and NFA without ∈ are accepting same language. To obtain an equivalent NFA with ∈, given with ∈-NFA what is required to be done is elimination of ∈-transition from a given automata. But simply eliminating the ∈-transition from a given NFA with ∈-NFA,will change the language accepted by the automata, hence for every ∈ -transition, to be eliminated we have to add some non ∈ -transitions as substitute so as to keep the language accepted by the automata same therefore, transformation of NFA with ∈ to NFA without ∈ involve finding out the non ∈ transition to be added to the automata for every ∈ transition to be eliminated.

Consider the above example

$\hat{\delta}(q_0, \in) = \in$-closure $(q_0) = \{q_0, q_1, q_2\}$

$\hat{\delta}(q_1, \in) = \in$-closure $(q_1) = \{q_1, q_2\}$

$\hat{\delta}(q_2, \in) = \in$-closure $(q_2) = \{q_2\}$

Procedure for conversion of ∈-NFA to NFA:

Step1: Find ∈-closure of each states.

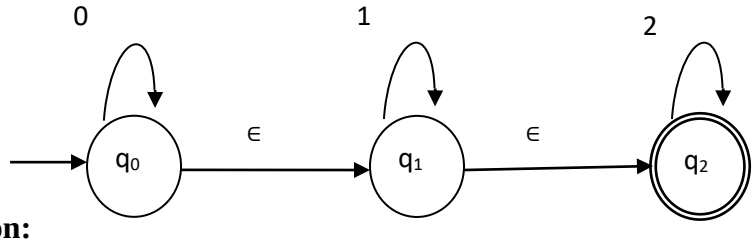
Step2: Find F^1 new set of final states include the states whose ∈-closure include initial state.

Step3: Find δ^1 is the transition of NFA.

Step4: Construct transition diagram of DFA.

Notes Prepared By. Ahmad Sir
Contact: 9867597554

Example: Convert the fallowing ∈-NFA to NFA



Solution:

Transition table for ∈-NFA

States\ Σ	**0**	**1**	**2**	∈
q_0	$\{q_0\}$	-	-	$\{q_1\}$
q_1	-	$\{q_1\}$	-	$\{q_2\}$
q_2	-	-	$\{q_2\}$	-

Step 1: Find ∈-closure of each states:

$\hat{\delta}(q_0, \in)$ = ∈-closure $(q_0) = \{q_0, q_1, q_2\}$

$\hat{\delta}(q_1, \in)$ = ∈-closure $(q_1) = \{q_1, q_2\}$

$\hat{\delta}(q_2, \in)$ = ∈-closure $(q_2) = \{q_2\}$

Step 2: find new set of final states

$$F' = \{q_0, q_1, q_2\}$$

Step 3: find δ^1 is the transition of NFA.

1. $\delta'(q_0, 0) = \epsilon - Closure\ (\delta(\hat{\delta}\ (q_0, \epsilon), 0))$

$= \epsilon - Closure\ (\delta(q_{0,}q_{1,}q_2), 0)$

$= \epsilon - Closure(\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0))$

$= \epsilon - Closure(q_0)$

$= \{q_0, q_1, q_2\}$

$\delta'(q_0, 1) = \epsilon - Closure\ (\delta(\hat{\delta}\ (q_0, \epsilon), 1))$

$= \epsilon - Closure\ (\delta(q_{0,}q_{1,}q_2), 0)$

$= \epsilon - Closure(\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1))$

$= \epsilon - Closure(q_1)$

$= \{q_1, q_2\}$

$\delta'(q_0, 2) = \epsilon - Closure\ (\delta(\hat{\delta}\ (q_0, \epsilon), 2))$

$= \epsilon - Closure\ (\delta(q_0, q_1, q_2), 2)$

$= \epsilon - Closure(\delta(q_0, 2) \cup \delta(q_1, 2) \cup \delta(q_2, 2))$

$= \epsilon - Closure(q_2)$

$= \{q_2\}$

2. $\delta'(q_1, 0) = \epsilon - Closure\ (\delta(\hat{\delta}\ (q_1, \epsilon), 0))$

$= \epsilon - Closure\ (\delta(q_1, q_2), 0)$

$= \epsilon - Closure(\delta(q_1, 0) \cup \delta(q_2, 0))$

$= \epsilon - Closure(\emptyset)$

$= \emptyset$

$\delta'(q_1, 1) = \epsilon - Closure\ (\delta(\hat{\delta}\ (q_1, \epsilon), 1))$

$= \epsilon - Closure\ (\delta(q_1, q_2), 1)$

$= \epsilon - Closure(\delta(q_1, 1) \cup \delta(q_2, 1))$

$= \epsilon - Closure(q_1)$

$= \{q_1, q_2\}$

$\delta'(q_1, 2) = \epsilon - Closure\ (\delta(\hat{\delta}\ (q_1, \epsilon), 2))$

$= \epsilon - Closure\ (\delta(q_1, q_2), 2)$

$= \epsilon - Closure(\delta(q_1, 2) \cup \delta(q_2, 2))$

$= \epsilon - Closure(q_2)$

$= \{q_2\}$

$\delta'(q_2, 0) = \epsilon - Closure\ (\delta(\hat{\delta}\ (q_2, \epsilon), 0))$

$= \epsilon - Closure\ (\delta(q_2), 0)$

$= \epsilon - Closure(\delta(\emptyset)$

$= \emptyset$

$\delta'(q_2, 1) = \epsilon - Closure\ (\delta(\hat{\delta}\ (q_2, \epsilon), 1))$

$= \epsilon - Closure\ (\delta(q_2), 1)$

$= \epsilon - Closure(\emptyset)$

$= \emptyset$

$\delta'(q_2, 2) = \epsilon - Closure\ (\delta(\hat{\delta}\ (q_2, \epsilon), 2))$

$= \epsilon - Closure\ (\delta(q_2), 2)$

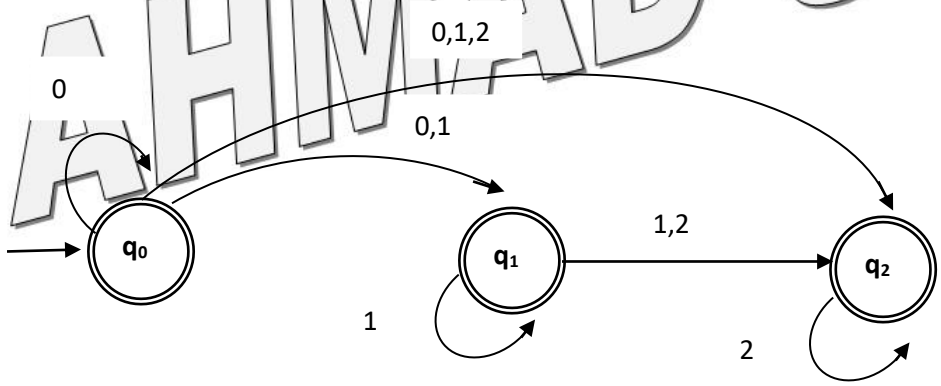
$= \epsilon - Closure(q_2)$

$= \{q_2\}$

Transition table of NFA:

States\ Σ	**0**	**1**	**2**
$\rightarrow * q_0$	$\{q_0, q_1, q_2\}$	$\{q_1,\ q_2\}$	$\{q_2\}$
$* q_1$	$\emptyset$	$\{q_1, q_2\}$	$\{q_2\}$
$* q_2$	$\emptyset$	$\emptyset$	$\{q_2\}$

Step4: Transition diagram:



➢ Regular Expressions (RE):

The language accepted by ***Finite Automata (FA)*** are represented by simple expression is called as ***Regular Expression***.

Formal Definition of Regular Expression:

Let Σ be a given alphabet. Then

1. ϕ,ϵ, and $a \in \Sigma$ are all regular expression. These are called ***primitive regular expression***.
2. If r_1, r_2 are regular expressions, then $r_1 + r_2$, $r_1.r_2$, r_1^* and (r_1) are also regular expressions.
3. A string is a regular expression if and only if it can be derived from ***the primitive regular expression*** by a finite number of applications of the rules in (2).

Example of some Basic Regular expressions:

Regular Expressions (RE)	Languages (L)
$\emptyset$	$\{\ \}$
ϵ	$\{\epsilon\}$
a	$\{a\}$
a^*	$\{\epsilon, a, aa, aaa, aaaa, \ldots.\}$
a^+	$\{a, aa, aaa, aaaa, \ldots.\}$
$(a+b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots.\}$
$(a.b)^*$	$\{\epsilon, ab, abab, ababab, \ldots.\}$
DIGIT	0|1|2|3|4|5|6|7|8|9 or [0-9]
DIGITS	$\{DIGIT\}^+$
LETTER	a|b|c|….|z|A|B|C|…..|Z | [a-z A-Z]
ID	$\{LETTER\}.(\{LETTER\} \mid \{DIGITS\})^*$

➢ Scanner Generator (Lex / flex):

Lex:

Lex is a compiler writing tool which is used to generate a lexical analyzer or scanner from description of tokens of the programming language to be implemented. And this description required as regular expression.

Lex is generally used in manner depicted in following fig.

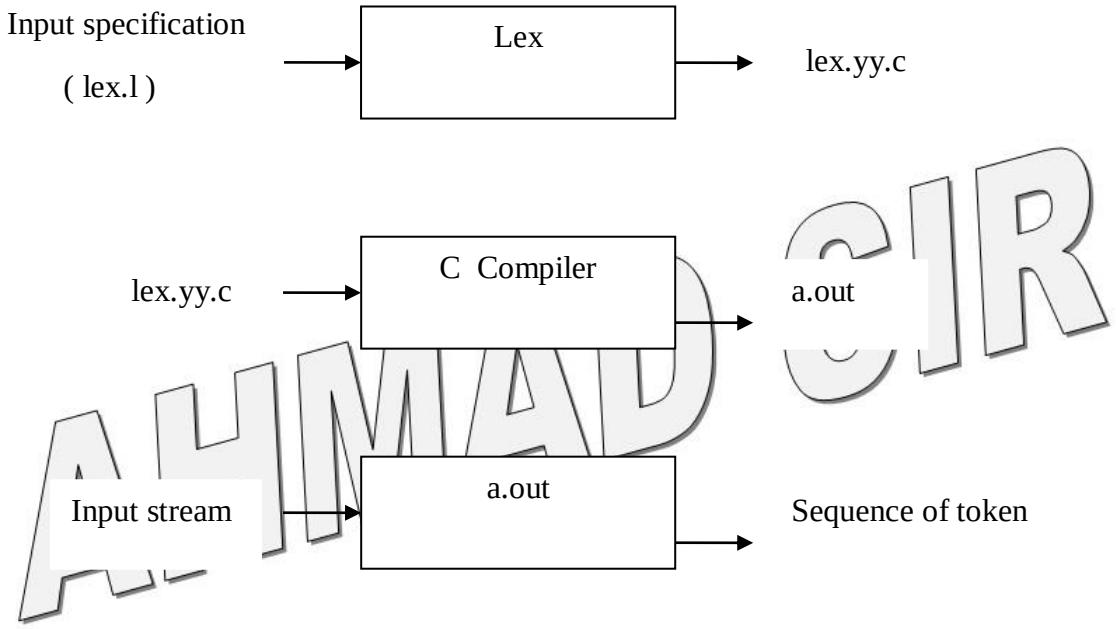


Fig: lex compiler

- First, a specification of lexical analyzer is prepared by creating a program lex.l in the lex language.
- Then lex.l is run through the lex compiler to produce a c program lex.yy.c. Lex.yy.c consist of a C language program containing recognizer for regular expression together with user supplied code.
- Finally lex.yy.c is run through the C compiler to produce an object program a.out which is a lexical analyzer that transforms an input stream into a sequence of token.

Lex Specification:

A lex program consist of three parts

Definition

%%

Rules

%%

Auxiliary procedure

- The definition includes declaration of variables, constant and regular definition between %{ %}.

The definition has format:

Name regular expression

- The rules contains the transition rules,

syntax:

Regular expression { c code }

- Auxiliary procedure contains user routines which are needed by action.

Alternatively, these procedure can be compiled separately and loaded with lexical analyzer

Consider the example:

%{ /* definition of manifest constant */

LT, LE, EQ, NE, GT, GE, IF,THEN,ELSE, ID, NUMBER, RELOP %}

/* Regular definition */

Delim	[|t |n]
ws	{delim}$^+$
Digit	0|1|2|3|4|5|6|7|8|9
Letter	a|b|c|….|z|A|B|C|…..|Z
Digits	{Digit}$^+$
Id	letter.(letter | digits)*

%%

{ ws }	{/* no action no return */ }
if	{ return (IF); }
then	{ return (THEN); }

else	{ return (ELSE); }
{ id }	{yylval = install.id( ); return (ID);}
{ number }	{yylval = install.num ( ); return (NUMBER);}
" < "	{yylval = LT ; return (RELOP); }
" > "	{yylval = GT ; return (RELOP); }
" < = "	{yylval = LE ; return (RELOP); }
" > = "	{yylval = GE ; return (RELOP); }
" < > "	{yylval = NE ; return (RELOP); }
" = "	{yylval = ET ; return (RELOP); }

%%

install_id( ):

{ /* procedure to install the lexeme whose first character is pointed to yytext and whose length is yylen to symbol table and return a pointer table. */

install_num ( ):

/* procedure to install the lexeme whose first character is pointed to yynum and whose length is yylen to symbol table and return a pointer table. */

Example:
Write a LEX program that recognizes-

1. **Keywords if, while, for**
2. **Identifier**
3. **Operator + |- |*| /**

Program:

```
%{
      #include<stdio.h>
%}
LETTER  [a-z A-Z]
DIGIT [0-9]
OPERATOR [ +,-,*,/]
%%
if        {
             printf("Recognized KEYWORD: %s \n", yytext);
          }

while     {
             printf("Recognized KEYWORD: %s \n", yytext);
          }
for       {
             printf("Recognized KEYWORD: %s \n", yytext);
          }
{LETTER}({LETTER}| {DIGIT})*  {
                                    printf("Recognized ID : %s \n", yytext);
                                 {
+         {
                printf("Recognized OPERATOR : %s \n", yytext);
          }

-         {
                printf("Recognized OPERATOR : %s \n", yytext);
          }

*         {
                printf("Recognized OPERATOR : %s \n", yytext);
          }


/         {
                printf("Recognized OPERATOR : %s \n", yytext);
          }
%%
main( )
{
    yylex();
}
```

The command to be given to generate the scanner are:

$lex test2.l

$cc lex.yy.c-ll

The scanner generated

$./a.out< data

The content of data file is shown below:

if

while

for

xyz

sum

+

-

*

/

The output produced for the above input is:

Recognized KEYWORD: if

Recognized KEYWORD: while

Recognized KEYWORD: for

Recognized ID: xyz

Recognized ID: sum

Recognized OPERATOR : +

Recognized OPERATOR : -

Recognized OPERATOR : *

Recognized OPERATOR : /

Compiler writing toots:

The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler. These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms. The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some commonly used compiler-construction tools include

1. Parser generators :

That automatically produce syntax analyzers from a grammatical description of a programming language.

2. *Scanner generators:*

That produce lexical analyzers from a regular-expression description of the tokens of a language.

3. *Syntax-directed translation engines:*

That produce collections of routines for walking a parse tree and generating intermediate code.

4. Code-generator generators:

That 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:*

That facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

6. *Compiler-construction toolkits:*

That provide an integrated set of routines for constructing various phases of a compiler.