Module 1

Introduction to Compilers & Lexical Analysis

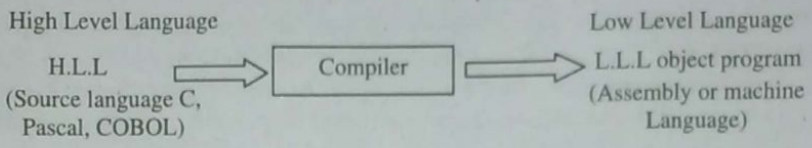
COMPILERS

Ques 1) What is compiler? Why compilers are needed?

Ans: Compiler

Compilers are software programs (or group of program modules) that translate source programs into equivalent object programs (target language program).

In other words, compilers are computer programs that take input written in one language, i.e., source language (such as C, Pascal, COBOL etc.,) and produce output in another language such as machine language or object language.



The computer understands only machine level code which is in the form of binary numbers. Therefore the compiler is needed to convert your program code into machine level code.

Compiler first parses the program and checks for any errors to verify the correctness of the program then it generates an output called the object code which enables the computer to execute code by single instruction at a time. Therefore it is necessary to have compiler so that you can check and verify the results.

Some of the compilers are:

1) Ada compilers
2) ALGOL compilers
3) BASIC compilers
4) C# compilers
5) C compilers
6) C++ compilers
7) COBOL compilers

Need of Compiler

1) We need compiler to run the programme.
2) Compilers provide portability.
3) Compilers enable high performance and productivity.
4) Compilers are needed to bridge this semantic gap.
5) Compiler will check the error in the problem. Compiler can check the error with the help of run time compiler.

Ques 2) Explain the structure of compiler.

Or

What are the different parts of compilers?

Ans: Structure of Compiler

Compilers bridge source programs in high-level languages with the underlying hardware. Compilers are required:

1) To recognize legitimacy of programs,
2) To generate correct and efficient code,
3) Run-time organization,
4) To format output according to assembler or linker conventions.

Parts of Compiler

A compiler consists of three main parts (**figure 1.1**):

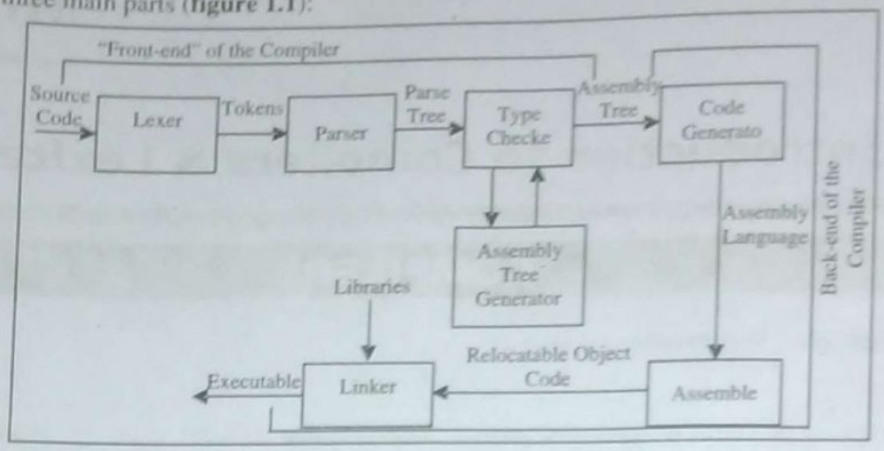


Figure 1.1: Parts of Compiler

1) **Front-end:** The front-end analyses the source code to build an internal representation of the program, called **intermediate representation (IR)**. It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope.
2) **Middle-end**: Middle-end is where the optimizations for performance take place. Typical transformations for optimization are removal of useless or unreachable code, discovering and propagating constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specializing a computation based on the context. Middle-end generates IR for the following backend. Most optimization efforts are focused on this part.
3) **Back-end:** Back-end is responsible for translation of IR into the target assembly code. The target instruction(s) are chosen for each IR instruction. Variables are also selected for the registers. Backend utilizes the hardware by figuring out how to keep parallel FUs busy, filling delay slots, and so on.

Ques 3) What is pass in compiler? Define its types and compare them.

Ans: Passes

Pass is a collection of one or more phases of compilation process. Pass simply reads the output of the previous pass and provides output to the next pass, if there is no previous pass, then it reads source program and generates intermediate result for the next pass. This process is continued until the compilation is over.

The allocation of passes depends on programming language. It can be categorized into two types:

1) **Single Pass Compiler:** A single pass compiler makes a single pass over the source text, parsing, analysing, and generating code all at once. **Figure 1.2** shows dependency diagram of a typical single pass compiler.

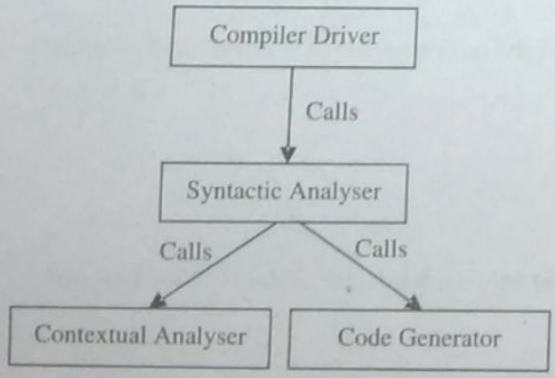


Figure 1.2: Single Pass Compiler

2) **Multi-Pass Compiler:** A multi pass compiler makes several passes over the program. The output of a preceding phase is stored in a data structure and used by subsequent phases.

Figure 1.3 shows dependency diagram of a typical multi pass compiler.

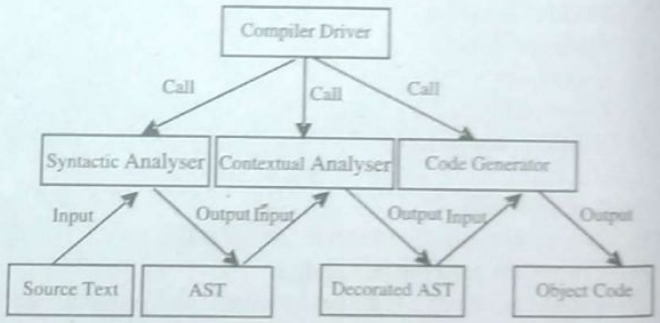


Figure 1.3: Multiple Pass Compiler

Comparison of Single-Pass and Multi-Pass Compiler

1) More than one pass or multi-pass compiler required less space than single pass compiler, because in multi-pass compiler same space is used again, that is used by one pass. It means the subsequent passes of compiler use same space.
2) A single pass compiler is fast as compare to multi-pass compiler and requires more space, because multi-pass compiler produces intermediate result during each pass and it takes time to read and write the intermediate results.

Ques 4) Discuss the steps involved in the analysis of a source program with the help of a block diagram.

Ans: Analysis of the Source Program

The steps involved in the analysis of source program are given below:

Step 1) Source program acts as an input to the preprocessor. Preprocessor modifies the source code by replacing the header files with the suitable content. Output (modified source program) of the preprocessor acts as an input for the compiler.

Step 2) Compiler translates the; modified source program of high-level language into the target program. If the target program is in machine language, then it can be executed directly. If the target program is in assembly language, then that code is given to the assembler for translation. Assembler translates the assembly language code into the relocatable machine language code.

Step 3) Relocatable machine language code acts as an input for the linker and loader. Linker links the relocatable code with the library files and the relocatable objects, and loader loads the integrated code into memory for the execution. The output of the linker and loader is the equivalent machine language code for the source code.

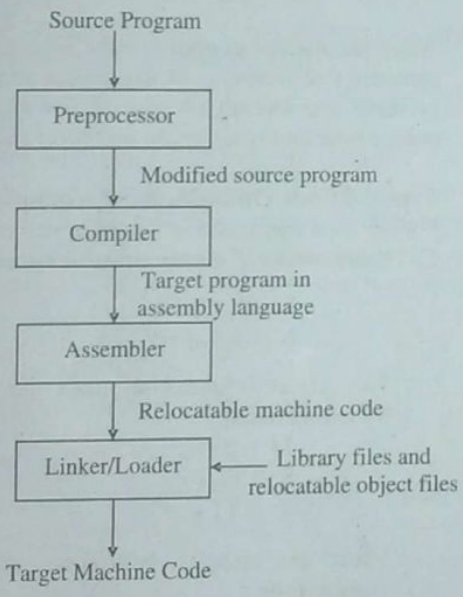


Figure 1.4: Block Diagram of Source Program Analysis

Ques 5) What are the basic functions of compiler?

Or

What are the various phases of compiler?

Or

Explain the error handling function of compiler.

Ans: Basic Compiler Functions/Phases of Compiler

To compiling a high level language program, there are some fundamental operations that are necessary such as lexical analysis, syntactic analysis, and code generation.

For the purpose of compiler construction, a high-level programming language is usually described in term of a grammar.

This grammar specifies the form, or syntax of legal statements in the language.

So the process of writing a compiler is not easy task, either logical point of view or implementation point of view. So we can split the overall design process of compiler into modules. These modules are called **phases**.

Hence, the implementation of compiler consists following phases as shown in **figure 1.5**.

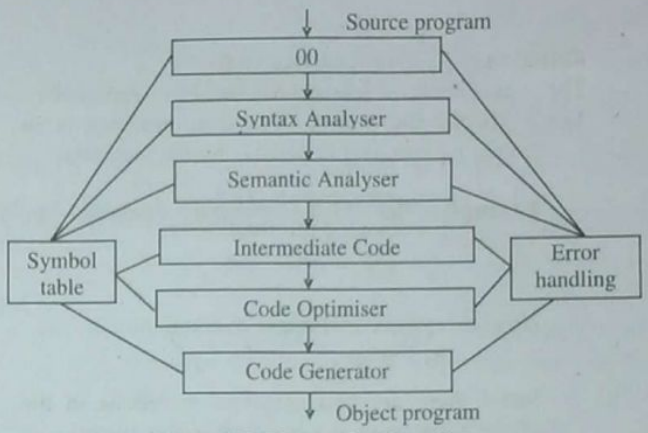


Figure 1.5: Phases of a Compiler

So the implementation of compiler consists following phases as shown in **figure 1.5**.

1) **Lexical Analysis (Scanning):** Lexical analyser plays an important role in compilation process of a program. It takes the source program as input and reads it one character at a time and produces equivalent token stream of a program.

   For example, A = B + C*50 (source program) statement. The corresponding tokens stream after lexical analyser phase are $x_1 = x_2 + x_3 * 50$, where x_1, x_2 and x_3 are tokens.

   Token is simply group of characters that logically belongs together. **For example,** Do, while, if, then, else, etc. are called **tokens**.

   Example:

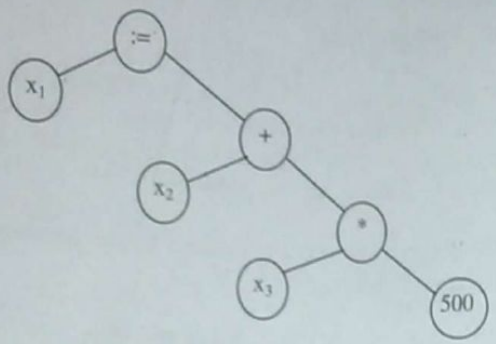
   1) If (a < b) goto xx

      There are eight tokens that are:
      If ; (; a; <; b;); goto; and xx.

   2) 2S = B
      2S; =; B (three tokens, no error will be generated at this stage.)

   3) if. A = = B
      Spaces or some characters may work as delimiter to identify a word.

2) **Syntax Analysis (Parsing):** Syntax analyser checks output of the lexical analyser is according to language specification rule, and it also imposes the tree-view structure of token streams.



Function of System Analyser (parser)
The syntax analyser (parser) has two basic functions:

1) It checks the output of lexical analyser is in proper pattern, that is, specify by the language.

   Example: In PL/I program contains the statement,

   X * + Y

   Now the **syntax** analyser converts it into,

   $x_1 * + x_2$

   Since there are two adjacent operations in the above expression, it violates the language rules of PL/I statement.

2) It also imposes the tokens a tree view structure that is used by further phases of the compiler. Second function of syntax analyser is to group tokens, in tree view structure.

   Example:

   X * Y/ Z

   Above statement have two interpretations:
   i) Multiply X by Y and then divides it to Z.
   ii) Divide Y by Z and then multiply it to X.

   The interpretations are based on the operator priority of language rules.

3) **Intermediate Code Generation:** It converts the syntactic structure into equivalent intermediate code.

   Various techniques are used to write the intermediate code.

   The intermediate code of the above syntactic tree is as follows:

   temp1 = int to real (50)
   temp2 = x_3 * temp1
   temp3 = x_2 + temp2
   x_1:= temp3

   Where temp1, temp2, and temp3 are compiler generated temporary variables.

Three-Address Code
One popular type of intermediate language is called "three-address code". **For example,** a typical three-address code statement is shown below:

A: = B op C

Where A, B, and C are operands and 'op' is a binary operator.

For example, the three-address code sequence is:

T1:= A/B
T2:= T1*C

Where T1 and T2 are names of temporary variables

4) **Code Optimization:** In this phase intermediate codes are optimized, so that we can pass optimized information from this phase to next phase (i.e., code generation).

The main object of this phase is to improve the performance of running process of computer program.

The main aim is to design a compiler in such a way that produces efficient target code.

When we attempt to optimize the source program, we consider that meaning of the source program should preserve and overall process of optimization should reduce time and space taken by object code.

Types of Code Optimization Techniques
For efficient optimization there are two techniques:

1) **Elimination of common sub-expression**

   S: = p + q + r
   X: = p + q + γ

   The above expression must be reduced by following mean:

   T1:= p + q
   S: = T1 + r
   X: = T1 + γ

   This is called **common sub-expression elimination**.

2) **Elimination of loop:** This type of conditional and unconditional statements could save space as well as time of program execution.

   X = 1
   Y = 2
   If Y < X goto (9)
   If Y ≥ X goto (5)
   X = X/Y
   T1 = Y +1
   Y = T1
   goto (3)
   goto (1)

5) **Code Generation:** Code generation is the final phase of compilation process, in which following points are considered:
   i) Specification of memory for data items.
   ii) Allocation of codes to access data items and usage of registers for computation purpose of expressions.

   This transforms intermediate code into equivalent machine code.

   For example,
   Statement X = Y/Z + w

   The equivalent machine code is:
   Load Y
   Div Z
   Add W
   Store X

   Problems in Code Generation
   There are three problems that arise when we attempt to generate an efficient target code.
   i) Selection of instructions to represent the computation that is specified by the three-address statement.
   ii) To decide the order of computation that leads to generate more efficient code.
   iii) Deciding the registers for computation purpose.

6) **Symbol Table (Book-Keeping):** A Symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.

   It keeps track of various data items used in program.

   For example, data items used in program are real, char, date, etc. Symbol table keeps track of data object/ data structure that is used in the source program.

   For example, in the source program, the declare variable is an integer, real or float and what is size of that variable, how many parameters we are passing through the function and so on.

   Procedure q
   Var x, y, z; real

100	
105	Var, real, val = x
106	Var, real, val = y
107	Var, real, val = z

7) **Error Handler:** One of the most important functions of a compiler is the detection and reporting of errors in the source program.

   The error message should allow the programmer to determine exactly where the errors have occurred.

   Error handler keeps records of errors during the compilation process of program and provides appropriate diagnosis for those errors.

   Errors can be encountered by virtually all of the phase of a compiler.

   Some situations are like:
   i) Lexical analyser may be unable to proceed because the next token in the source program is misspelled.
   ii) Syntax analyser may be unable to infer a structure for its input because a syntactic error such as a missing parenthesis has occurred.
   iii) Intermediate code generator may detect an operator whose operands have incompatible types.
   iv) Code optimizer, doing control flow analysis may detect that certain statements can never be reached.
   v) Code generator may find a compile-created constant that is too large to fit in a word of the target machine.
   vi) While entering information into the symbol table, the book keeping routing may discover an identifier that has been multiply declared with contradictory attributes.

   Whenever a phase of the compiler discovers an error, it must export the error to the error handler, which issues an appropriate diagnostic message.

   Once the error has been noted, the compiler must modify the input to the phase detecting the error, so that the latter can continue processing its input, looking for subsequent errors.

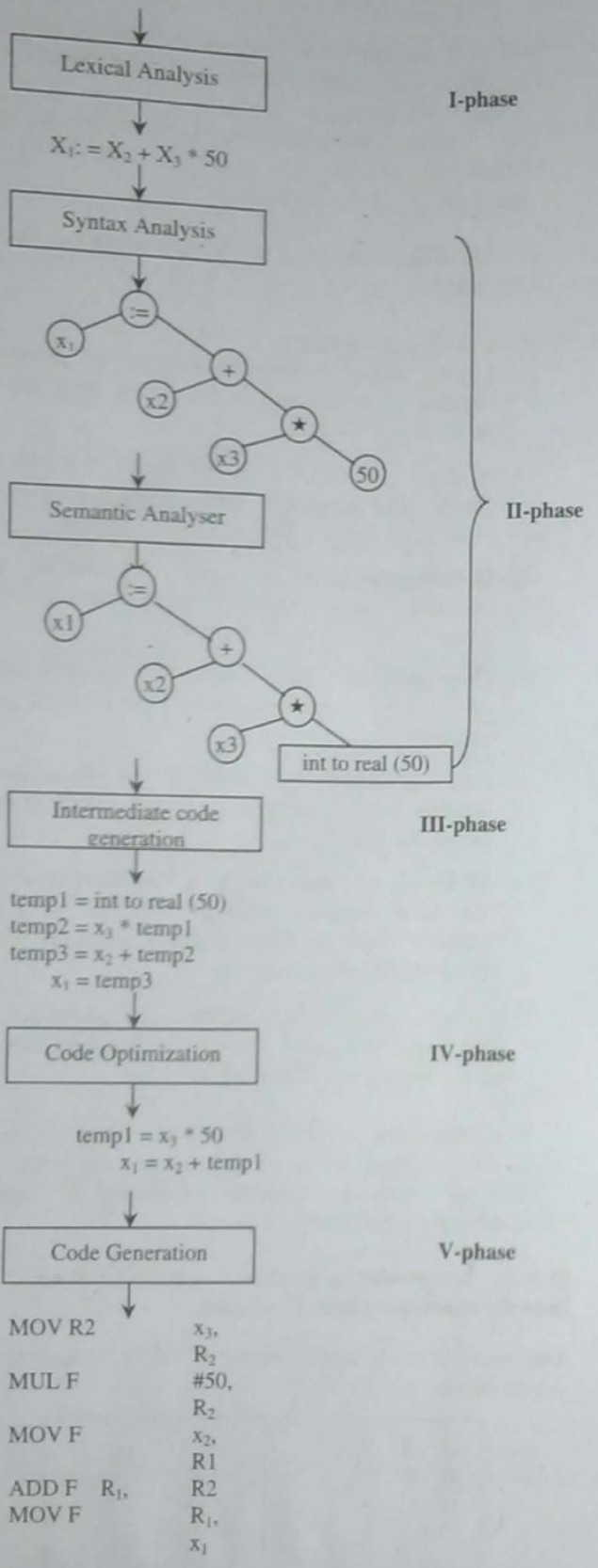
Ques 6) Let consider the program statement A = B + C * 50. Draw the subsequent phases of compiler.

Ans: Symbol table management of above statement is **shown below**:

1) A 2) B 3) C	- -

Figure 1.6: Compilation Process of Source Program Statement (A = B + C *50)

Following **figure** shows the subsequent phases of compiler:



Ques 7) Explain how the phases of compilers are grouped?

Or

Discuss about the grouping of phases.

Ans: Grouping of Phases

The phases of a compiler can be grouped as:

1) **Front End:** Front end comprises of phases which are dependent on the input (source language) and independent on the target machine (target language).

   Front end of a compiler consists of the phases:
   i) Lexical analysis,
   ii) Syntax analysis,
   iii) Semantic analysis,
   iv) Intermediate code generation.

2) **Back End:** Back end comprises of those phases of the compiler that are dependent on the target machine and independent on the source language.

   Back end of a compiler contains:
   i) Code optimization,
   ii) Code generation.

3) **Passes:** The phases of compiler can be implemented in a single pass by marking the primary actions viz. reading of input file and writing to the output file. Several phases of compiler are grouped into one pass in such a way that the operations in each and every phase are incorporated during the pass.

   For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass. If so, the token stream after lexical analysis may be translated directly into intermediate code.

4) **Reducing the Number of Passes:** Minimising the numbers of passes improve the time efficiency as reading from and writing to intermediate files can be reduced.

   When grouping phases into one pass, the entire program has to be kept in memory to ensure proper information flow to each phase because one phase may need information in a different order than the information produced in previous phase.

   The source program or target program differs from its internal representation. So, the memory for internal form may be larger than that of input and output.

Ques 8) What are the different compiler writing tools?

Ans: Compiler Writing Tools

A number of tools have been developed specifically to help construct compilers. These tools range from scanner and parser generators to complex systems, variously called compiler-compiler, compiler-generators, translator-writing systems, which produce a compiler from some form of specification of a source language and target machine.

The input specification for these systems may contain:

1) A description of the lexical and syntactic structures of the source language (**for example** A + B * C → (A + B)* C or A + (B * C)).

2) A description of what output is to be generated for each source language, and
3) A description of the object machine.

More general approaches to the automatic generation of lexical analyzers exists, but these require the user to supply more input to the compiler-compiler, i.e., to do more work.

The following is a list of some useful **compiler-constructions tools**:

1) **Parser Generators:** these produce syntax analysis, normally from input that is based on a context free grammar.

2) **Scanner Generators:** these automatically generate lexical analyzer.

3) **Syntax-Directed Translation Engines:** In this tool, the parse tree is scanned completely to generate an intermediate code. The translation is done for each node of the tree.

4) **Automatic Code Generator:** These generators take an intermediate code as input and converts each rule of intermediate language into equivalent machine language. The template matching technique is used.

   The intermediate code statements are replaced templates that represent the corresponding sequence of machine instructions.

5) **Data-Flow Analysis Engines:** 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:** Compiler-construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

Ques 9) What is bootstrapping? What are the advantages of bootstrapping?

Ans: Bootstrapping

The process by which a simple language is used to translate a more complicated program, which in turn may handle an even more complicated program and so on, is known as **bootstrapping**.

In other words, one wants to write a compiler for a language A, targeting language B (the machine language) and written in language B.

The most obvious approach is to write the compiler in language B. But if B is machine language, it is a horrible job to write any non-trivial compiler in this language.

Instead, it is customary to use a process called "bootstrapping", referring to the seemingly impossible task of pulling oneself up by the bootstraps.

Bootstrapping is an important concept in building new compilers. A compiler is characterized by three languages:

1) Its source language,
2) Its object language and
3) The language in which it is written.

For example, a compiler may run on one machine and produce object code for another machine. Such a compiler is often called cross-compiler.

Many mini-computer and microprocessor compilers run on a bigger machine and produce object code for the smaller machine.

We write a compiler C_z^{xy} in the simple language Z. This program when run through C_y^{zy}, becomes C_y^{xy}, the compiler for the language x, running on machine y, and producing object code for y.

The above process is represented by **figure 1.7**:

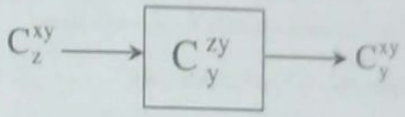


Figure 1.7: Bootstrapping of Compiler

Advantages of Bootstrapping

Bootstrapping a compiler has the following advantages:

1) It is a non-trivial test of the language being compiled, and as such is a form of dogfooding.
2) Compiler developers and bug reporting part of the community only need to know the language being compiled.
3) Compiler development can be done in the higher level language being compiled.
4) Improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself.
5) It is a comprehensive consistency check as it should be able to reproduce its own object code.

LEXICAL ANALYSIS

Ques 10) What is Lexical Analysis or Scanning?

Ans: Lexical Analysis (Scanning)

The first phase of Compiler is called **lexical analysis** or **scanning**.

Scanning is the process of recognizing the lexical components in a source string. Scanning is the first stage of a three-part process that the compiler uses to understand the input program.

The **scanner**, or **lexical analyser**, takes as input a stream of characters and produces as output a stream of words along with their associated syntactic categories. It aggregates symbols to form words and applies a set of rules to determine whether or not each word is legal in the source language. If the word is valid, the scanner assigns it a syntactic category, or part of speech. To make this process efficient, compilers use specialized recognizers.

Most of the work in scanner construction has been automated; indeed, this is a classic example of the application of theoretical results to solve an important practical problem-that is, specifying and recognizing patterns in strings.

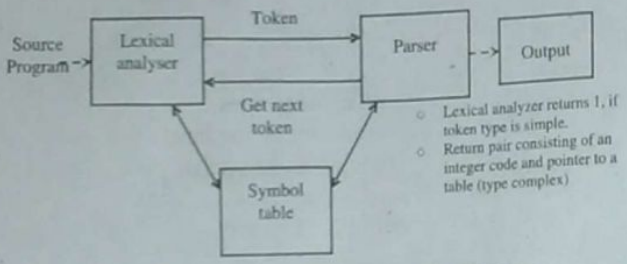


Figure 1.8: Interaction of Lexical Analyzer with Parser

Ques 11) What is the role of Lexical Analyser?

Ans: Role of Lexical Analyser

Lexical analyser performs the following tasks:

1) Reads the source program, scans the input characters, group them into lexemes and produce the token as output.
2) Enters the identified token into the symbol table.
3) Strips out white spaces and comments from source program.
4) Correlates error messages with the source program i.e., displays error message with its occurrence by specifying the line number.
5) Expands the macros if it is found in the source program.

Tasks of lexical analyser can be divided into two processes:

1) **Scanning:** Performs reading of input characters, removal of white spaces and comments.
2) **Lexical Analysis:** Produce tokens as the output.

Ques 12) Describe the following terms:

1) Tokens
2) Patterns
3) Lexemes

Ans: In lexical analysis normally uses the following terms:

1) **Token:** Token is a valid sequence of characters which are given by lexeme. In a programming language, the following are the possible tokens to be identified:
   i) Keywords,
   ii) Constant,
   iii) Identifiers,
   iv) Numbers,
   v) Operators and
   vi) Punctuations symbols
2) **Pattern:** Pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token. It can be defined by regular expressions or grammar rules.
3) **Lexeme:** Lexeme is a sequence of characters that matches the pattern for a token i.e., instance of a token.

For example, consider the following statement:

$z = x + y*7;$

Token and Lexeme of this statement is shown in **table below:**

Lexemes	Tokens
z	Identifier
=	Assignment symbol
x	Identifier
+	+ (Addition Symbol)
y	Identifier
*	* (Multiplication Symbol)
7	7 (Number)

Ques 13) What are the different approaches of building lexical analyser?

Ans: Approaches of Building Lexical Analyser

There are three approaches to building a lexical analyser:

1) Write a formal description of the token patterns of the language using a descriptive language related to regular expressions.

   These descriptions are used as input to a software tool (**for example,** Lex) that automatically generates a lexical analyser.
2) Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
3) Design a **state transition diagram** that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

Ques 14) What is the role of input buffering in lexical analyser?

Ans: Role of Input Buffering in Lexical Analyser

The lexical analyser scans the characters of source program one by one to find the tokens. Moreover, it needs to look ahead several characters beyond the next token to determine the next token itself. So, an input buffer is needed by the lexical analyser to read its input.

In a case of large source program, significant amount of time is required to process the characters during the compilation. To reduce the amount of overhead needed to

process a single character from input character stream, specialized buffering techniques have been developed. An important technique that uses two input buffers that are reloaded alternately is shown in **figure 1.9**.

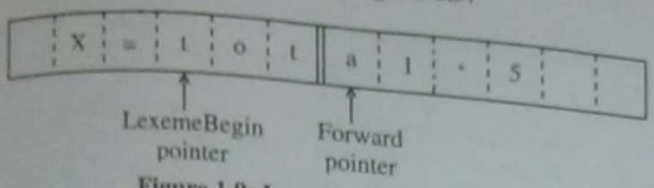


Figure 1.9: Input Buffer

Each buffer is of the same size N, where N is the size of a disk block, e.g., 1024 bytes. Thus, instead of one character, N characters can be read at a time. The pointers used in the input buffer for recognising the lexeme are as follows:

1) Pointer lexemeBegin points the beginning of the current lexeme being discovered.
2) Pointer forward scans ahead until a pattern match is found for lexeme.

Initially, both pointers point to the first character of the next lexeme to be found. The forward pointer is scanned ahead until a match for a pattern is found.

After the lexeme is processed, both the pointers are set to the character following that processed lexeme.

For example, in **figure 1.10** the lexemeBegin pointer is at character t and forward pointer is at character a. The forward pointer is scanned until the lexeme total is found.

Once it is found, both these pointers point to *, which is the next lexeme to be discovered.

SPECIFICATION OF TOKENS

Ques 15) What are strings and languages in lexical analysis?

Or

What are the different terminologies related to languages?

Ans: Strings

A "string" over an alphabet is a finite sequence of symbols from that alphabet, which is usually written next to one another and not separated by commas.

For example,

1) If $\Sigma_a = \{0, 1\}$ then 001001 is a string over Σ_a.
2) If $\Sigma_b = \{a, b, \ldots. Z\}$ then axyrpqsted is a string over Σ_b.
3) 00110, 01, 1, 0 are all strings over an alphabet $\Sigma = \{0, 1\}$
4) abab, aabb, ab, ba, a, are all strings over an alphabet $\Sigma = \{a, b\}$.

Languages

Language is simply a set of strings involving symbols from some alphabet.

Any set of strings over an alphabet Σ is called a language. The set of all strings, including the empty string over an alphabet Σ is denoted as Σ^*.

Infinite languages L are denoted as:

$L = \{w \in \Sigma^* : w \text{ has property P}\}$

For example,

1) $L_1 = \{w \in (0, 1)^* : w$ has an equal number of 0's and 1's$\}$
2) $L_2 = \{w \in \Sigma^* : w = w^R\}$ where w^R is the reverse string of w.

Terminology Related to Languages

Some important definitions regarding languages include,

1) **Alphabet:** It is defined as a finite set of symbols. **For example,** Roman alphabet {a, b, ... z}. "Binary Alphabet" {0, 1} is pertinent to the theory of computation.
2) **Length of String:** The "length" of a string is its length as a sequence. The length of a string w is written as |w|. **For example,** 1100111= 5
3) **Empty String:** The string of zero length is called the "empty string". This is denoted by ϵ.

   The empty string plays the role of 0 in a number system.
4) **Reverse String:** If $w = w_1 w_2 \ldots w_n$ where each $w_i \in \Sigma$, the reverse of w is $w_n w_{n-1} \ldots w_1$
5) **Substring:** z is a substring of w if z appears consecutively within w. **For example,** 'deck' is a substring of 'abcdeckabcjkl'.
6) **Suffix:** If w = xv for some x, then v is a suffix of w.
7) **Prefix:** If w = vy for some y, then v is a prefix of w.
8) **Lexicographic Ordering:** The Lexicographic ordering of strings is the same as the dictionary ordering, except that shorter strings precede longer strings.

   The lexicographic ordering of all strings over the alphabet {0, 1} is
   (ϵ, 0, 1, 00, 01, 10, 11, 000, ...).

Ques 16) What are the operations performed on the languages?

Ans: Operations Performed on Languages

1) **Concatenation:** If L_1 and L_2 are languages over Σ, their concatenation is $L = L_1 . L_2$, or simply $L = L_1 L_2$ where, $L = \{w \in \Sigma^* : w = x . y$ for some $x \in L_1$, and $y \in L_2\}$

For example, Given $\Sigma = \{0,1\}$

$L_1 = \{w \in \Sigma^* : w \text{ has an even number of } 0's\}$

$L_2 = \{w : w \text{ starts with a 0 and the rest of the symbols are 1's}\}$

Then
$L_1L_2 = \{w : w \text{ has an odd number of } 0's\}$

2) **Kleene Star:** Another language operation is the "Kleene Star" of a language L, which is denoted by L^*.

L^* is the set of all strings obtained by concatenating zero or more strings from L.

$$L^* = \left\{ w \in \Sigma^* : w = w_1 \bullet \cdots \bullet w_k \text{ for some } k \geq 0 \text{ and some } w_1, w_2, \ldots, w_k \in L \right\}$$

For example, if $L = \{01, 1, 100\}$ then $110001110011 \in L^*$, since $110001110011 = 1 \bullet 100 \bullet 01 \bullet 1 \bullet 100 \bullet 1 \bullet 1$, each of these strings is in L.

Ques 17) Given $\Sigma = \{a,b\}$ obtain Σ^*.

i) **Give an example of a finite language in Σ.**

ii) **Given $L = \{a^n b^n : n \geq 0\}$, check if the strings aabb, aaaabbbb, abb are in the language L.**

Ans: $\Sigma = \{a,b\}$. Therefore, we have $\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa,\}$

i) {a, aa, aab} is an example of a finite language in Σ.

ii) a) aa bb → a string in L. (n = 2)
b) aaaa bbbb → a string in L. (n = 4)
c) abb → not a string in L (since there is no n satisfying this).

Ques 18) Let L = {ab, aa, baa}. Which of the following strings are in L^*?

i) abaabaaabaa
ii) aaaabaaaa
iii) baaaaabaaaab
iv) baaaaabaa

Ans: Note that L^* is the "star-closure" of a language L, given by

$L^* = L^0 \cup L^1 \cup L^2 \ldots\ldots$
and L^+ is the "positive closure" defined by

$L^* = L^1 \cup L^2 \ldots\ldots$

i) ab aa baa ab aa → This string is in L^*
ii) aa aa baa aa → This string is in L^*
iii) baa aa ab aa aa b or baa aa ab aa a ab
↑ (undefined) ↑ (undefined)
This string is not in L^*.
iv) baa aa ab aa → This string is in L^*.

Ques 19) What is regular expression? Discuss the operations performed on regular expression.

Or

Explain the specification of tokens using regular expression.

Ans: Regular Expressions
A regular expression is a description of a set of strings.

We define regular expression as a means of representing certain subsets of strings over the alphabet and prove that regular sets are precisely those accepted by finite automata or transition diagrams.

Regular expressions are useful for representing certain sets of stings in an algebraic fashion. Actually these describe the language accepted by finite state automata.

Compiler uses this program of lexical analyser in the process of compilation. The task of lexical analyser is to scan the input program and separate out the 'tokens'.

For example, identifier is a category of token in the source language and it can be identified by regular expression as:
(letter) (letter + digit)*

If anything in the source language matches with this regular expression then it is recognized as identifier. The letter is nothing but a set {A, B, ... Z, a, b, ...z}* and digit is {0, 1, ... 9}*. The regular expression is effective way for identifying token from a language.

For example,

1) The regular expression a* denotes ε, a, aa, aaa, aaaa ……..
(Remember the operation of * to be zero or more occurrences, hence, the string 'a' can occur in zero or more cases, so ε is also included)
2) The regular expression a* b denotes b, ab, aab, aaab, ……
(a can occur in zero or more cases along with b)
3) The regular expression a* b* denotes ε, ab, aab, abb, aabb
(a and b can occur in zero or more cases)

Operations of Regular Expression
For a give regular expression, r, one denote the language that it specifies as L(r). A regular expression is built up from three basic operations:

1) **Alternation:** The alternation, or union, of two sets R and S, denoted R | S, is
$\{s \mid s \in R \text{ or } s \in S\}$

2) **Concatenation:** The concatenation of two sets R and S, denoted RS, is $\{st \mid s \in R \text{ and } t \in S\}$. One will sometimes write R^2 for RR, the concatenation of R with itself, and R^3 for RRR (or RR^2).

3) **Closure:** The Kleene closure of a set R, denoted as R^*, is $\bigcup_0^\infty R^i$. This is just the union of the concatenations of R with itself, zero or more times.

Ques 20) What are the different rules of regular expression?

Ans: Rules of Regular Expression

A regular expression is said to denote a formal language L over the alphabet and is defined by the following rules:

1) $\in$ is a regular expression denoting the language which consists of null string.
2) If 'a' is a symbol in the alphabet then 'a' is also a regular expression consists of {a}, i.e., any terminal symbol is also a regular expression.
3) If R and S are regular expressions over the alphabet then (R)/(S) is also a regular expression, which is the Union of the languages represented by R and S, i.e., $L(R) \cup L(S)$.
4) If R and S are regular expressions over the alphabet then (R)(S) is also a regular expression, denoting L(R)L(S)
5) R^* is a regular expression representing the language L(R) with zero or more occurrences of the string.
6) If R is regular expression then (R) is also a regular expression.

REVIEW OF FINITE AUTOMATA

Ques 21) Describe finite state automata.

Or

What do you mean by finite automata? Also write its mathematical definition.

Ans: Finite State Automata / Finite Automata

A Finite Automata or finite-State Automaton (FSA) is an abstract machine having:

1) A **finite set** of states. These carry no further structure and provide a simple form of memory.
2) A **start state** and a set of **final states**.
3) A finite set of **input symbols** (alphabet)
4) A finite set of **transition rules** which specify how the machine, when in a particular state, responds to a particular input symbol. The response may be to change state and/or produce an output (action).

A FSA processes an input string over its alphabet. Each symbol is processed in turn. The FSA starts in its initial state and uses the transition rules to determine the next state and output (if any) from its current state and the symbol just read.

If there is no rule defined for the current state and input symbol, the machine halts. If the entire string has been processed and the machine is in a final state, the FSA is said to accept the string.

Mathematical Definition

Mathematically, a finite automaton can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

1) Q is a finite non-empty set of states.
2) is a finite non-empty set of Inputs called input alphabet.
3) is a function which maps $Q \times \Sigma$ into Q and is usually called direct transition function. This is the function which describes the change of states during the transition. This mapping is usually represented by a transition table or a transition diagram.
4) $q_0 \in Q$ is the initial state.
5) $F \subseteq Q$ is the set of final states. It is assumed here that there may be more than one final state.

The transition function which maps $Q \times \Sigma^*$ into Q (i.e., maps a state and a string of input symbols including the empty string into a state) is called **indirect transition function**. We shall use the same symbol δ to represent both types of transition functions and the difference can be easily identified by nature of mapping (symbol or a string), i.e., by the argument. δ is also called **next state function**.

Ques 22) What are the elements of finite automata.

Ans: Elements of Finite Automata

The various components of finite automata are shown in **figure 1.10**:

1) **Input Tape:** Input tape is divided into cells (squares), which can hold one symbol from input alphabet. Input tape is a linear tape having some cells which can hold an input symbol from Σ.
2) **Finite Control:** It indicates the current state and decides the next state on receiving a particular input from the input tape. The tape reader reads the cells one by one from left to right and at any instance only one input symbol is read.

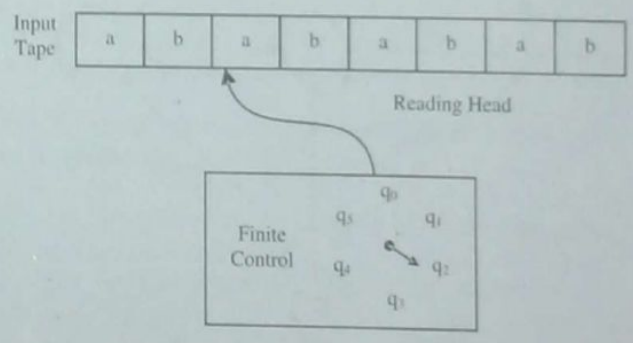


Figure 1.10: Block Diagram of Finite Automata

3) **Reading Head:** The reading head examines read symbol and the head moves to the right side with or without changing the state. When the entire string is read and if finite control is in final state then the string is accepted otherwise rejected.

Ques 23) What is transition diagram?

Ans: Transition Diagram

A transition graph or a transition system is a finite directed labeled graph in which each vertex (or node) represents a state and the directed edges indicate the transition of a state and the edges are labeled with input/output.

A typical transition system is shown in **figure 1.11**. In the **figure 1.11**, the initial state is represented by a circle with an arrow pointing towards it, the final state by two concentric circles, and the other states are represented by just a circle. The edges are labeled by input/output (e.g., by 1/0 or 1/1).

For example, if the system is in state q_0 and the input 1 is applied, the system moves to state q_1 as there is a directed edge from q_0 to q_1 with label 1/0. It outputs 0.

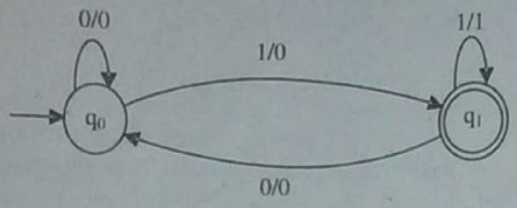


Figure 1.11: Transition System

Definition: A transition system is a 5-tuple (Q, Σ, δ, Q_0, F), where

1) Q, Σ and F are the finite non-empty set of states, the input alphabet, and the set of final states, respectively, as in the case of finite automata;
2) $Q_0 \subseteq Q$, and Q_0 is non-empty; and
3) δ is a finite subset of $Q \times \Sigma^* \times Q$.

In other words, if (q_1, w, q_2) is in δ, it means the following: The graph starts at the vertex q_1, goes along a set of edges, and reaches the vertex q_2. The concatenation of the label of all the edges thus encountered is w.

Ques 24) Consider the transition system given in figure 1.12.

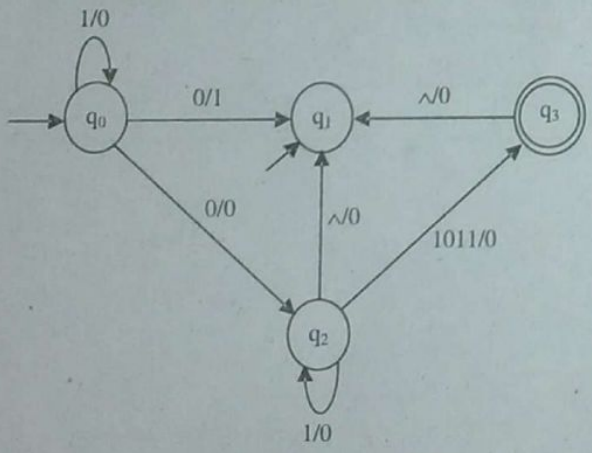


Figure 1.12: Transition System

Determine the initial states, final states, and the acceptability of 101011, 111010.

Ans: The initial states are q_0 and q_1. There is only one final state, *viz.*, q_3

The path-value of $q_0q_1q_2q_3$ is 101011. As q_3 is the final state, 101011 is accepted by the transition system. But, 111010 is not accepted by the transition system as there is no path with path value 111010.

Ques 25) Discuss various types of finite state automata with example.

Or

Write the formal definition of deterministic finite automata and non-deterministic finite automata.

Ans: Types of Finite State Automata

There are two types of finite automata as shown in **figure below**:

1) **Deterministic Finite State Automata (DFA):** The finite automaton is called deterministic finite automata if there is only one path for a specific input from current state to next state.

Definition

A deterministic finite automaton is a collection of following things and notation denoted as A = (Q, ∑, δ, q_0, F):

i) The finite set of states which can be denoted by Q,
ii) The finite set of input symbols ∑,
iii) The start state q_0 such that $q_0 \in Q$,
iv) A set of final states F such that $F \in Q$, and
v) The mapping function or transition function denoted by δ. Two parameters are passed to this transition function – one is current state and other is input symbol. The transition function returns a state which can be called as next state. **For example,** $q_1 = \delta(q_0,a)$ means from current state q_0, with input a the next state transition is q_1.

For example, all strings containing a 1 in third position from the end.

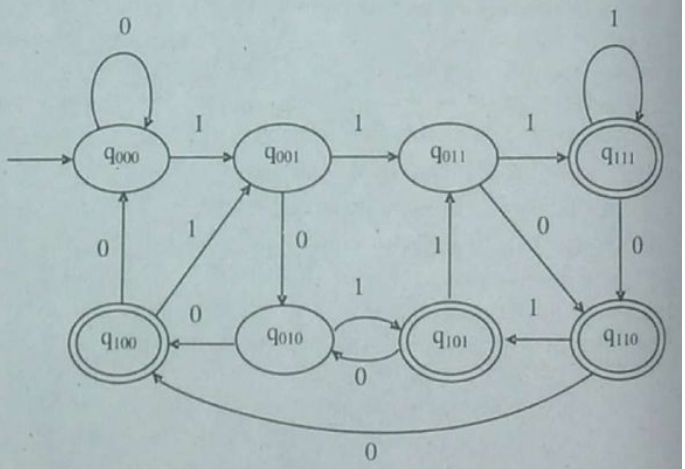


Figure 1.13: DFA

2) **Non-Deterministic Finite Automaton (NFA):** Non determinism is the ability to change states in a way that is only partially determined by current state and input symbol. That is, several possible "next states" are possible for a given combination of current symbol and input symbol.

The automaton, as it reads input string may choose at each step to go into any one of legal next states; the choice is not determined by anything and is therefore

called **nondeterministic**. Hence, if some moves of the machine cannot be determined uniquely by the input symbol and the present state. Such machines are called non-deterministic finite automata.

A Non-deterministic Finite Automaton (NFA) is very similar to Deterministic Finite Automaton. Just like a DFA, a NFA has states, transitions, an initial state and a set of final states.

The differences is that in a DFA, each state has at most one transition for a given symbol, while a NFA can have any number of transitions for a given symbol.

Definition

A Non-Deterministic Finite Automaton (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where,

i) Q is a finite non-empty set of states;

ii) Σ is a finite non-empty set of inputs;

iii) δ is the transition function mapping from $Q \times \Sigma$ into 2^Q which is the power set of Q, the set of all subsets of Q;

iv) $q_0 \in Q$ is the initial state, and

v) $F \subseteq Q$ is the set of final states.

For example, all strings containing a 1 in third position from the end.

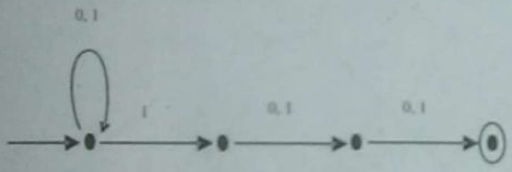


Figure 1.14: NFA

Ques 26) What is differnece between DFA and NFA.

Ans: Difference between Deterministic and Non-Deterministic Finite Automata

The main difference between the deterministic and non-deterministic automata is in δ. The other differences are given in the following table:

Deterministic Finite Automata	Non – Deterministic Finite Automata
For Every symbol of the alphabet, there is only one state transition in DFA.	There may be more than one state transition in NFA.
DFA cannot use empty string transition.	NFA can use empty string transition.
DFA can be understood as one machine.	NFA can be understood as multiple little machines computing at the same time.
DFA will reject the string if it ends at other than accepting state.	NFA is easier to construct.
If all of the branches of NFA dies or rejects the string, we can say that NFA reject the string.	It is more difficult to construct DFA.
DFA requires more space.	NFA requires less space.

Ques 27) Construct a finite state automata that accept the set of natural numbers x which are divisible by 3.

Ans: Let $M = (Q, \Sigma, q_0, \delta, F)$ be a machine with $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ $F = \{q_0\}$ and δ is defined as:

$\delta(q_0, a) = q_0$; $\delta(q_1, a) = q_1$; $\delta(q_2, a) = q_2$ for $a \in \{0, 3, 6, 9\}$

$\delta(q_0, b) = q_1$; $\delta(q_1, b) = q_2$; $\delta(q_2, b) = q_0$ for $b \in \{1, 4, 7\}$

$\delta(q_0, c) = q_2$; $\delta(q_1, c) = q_0$; $\delta(q_2, c) = q_1$ for $c \in \{2, 5, 8\}$

The transition diagram for this machine is given in **figure 1.15**.

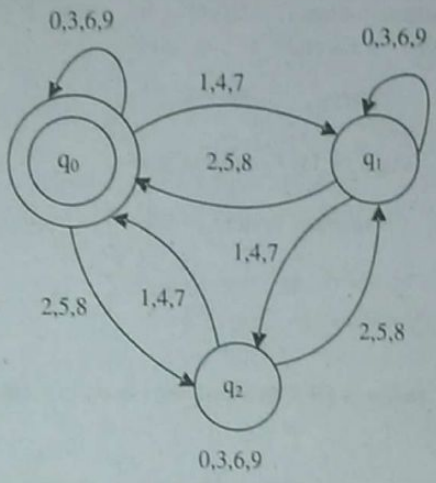


Figure 1.15

The computation of the string w = 150 is:

$\delta(q_0, 150) = \delta(\delta(\delta(q_0; 1), 5, 0)) = \delta(\delta(q_1, 5), 0)$

$= \delta(q_0, 0)$

$= q_0$ (accepting state) and

For w = 116,

$\delta(q_0, 116) = \delta(\delta(\delta(q_0, 1), 1), 6) = \delta(\delta(q_1, 1), 6)$

$= \delta(q_2, 6) = q_2$ (non-accepting state).

Ques 28) Design DFA which accepts odd number of 1's and any number of 0's.

Ans: In the problem statement, it is indicated that there will be a state which is meant for odd number of 1's and that will be the final state. There is no condition on number of 0's.

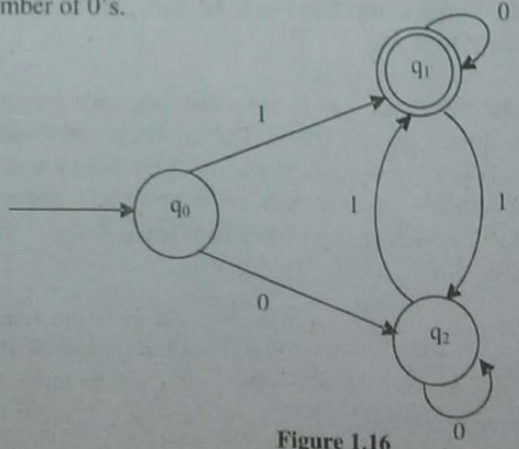
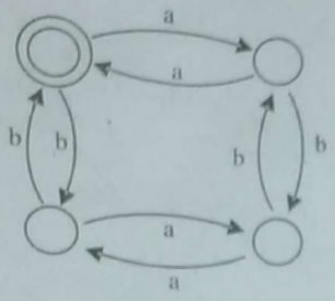


Figure 1.16

At the start, if we read input 1 then we will go to state q_1 which is a final state as we have read odd number of 1's. There can be any number of zeros at any state and therefore the self-loop is applied to state q_2 as well as to state q_1.

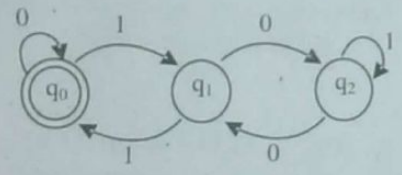
Ques 29) Design the DFA that accepts an even number of a's and even number of b's.

Ans: The Finite automate accept the string that contains even number of b's which means that the string should contain both even number of a's and even number of b's. The finite automaton that accepts the string that contain even number of a's and b's is as follows:



Ques 30) Design a DFA to accept the binary number divisible by 3.

Ans: The regular expression for binary numbers which are divisible by three is (0 | 1 (01*0)*1)*. The examples of binary number divisible by 3 are 0, 011, 110, 1001, 1100, 1111, 10010, etc. The DFA corresponding to binary number divisible by 3 can be shown in following **figure**.



The above automata will accept all binary numbers divisible by 3. For 1001, the automata will go from q_0 to q_1, then q_1 to q_2, then q_2 to q_1 and finally q_2 to q_0, hence accepted. For 0111, the automata will go from q_0 to q_0, then q_0 to q_1, then q_1 to q_0 and finally q_0 to q_1, hence rejected.

Ques 31) Design a DFA that accept the language
$L = \{a^n b: n \geq 0\}$

Ans: The state in DFA is used to remember many number of a's followed by a single b. The minimum requirement of states is 2, since if we start with q_0 if the machine see b, it changes its state to q_1 (an accepting state), and if sees 'a' it should stay back in same state q_0. That is $\delta(q_0, a) = q_0$. Hence we have

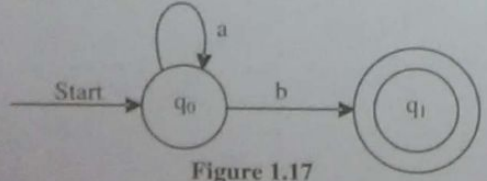


Figure 1.17

If the next input is a, b we define another state q_2, non-accepting state, such that

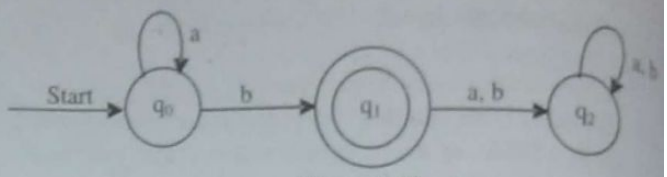


Figure 1.18

If abab is an input, then
$\delta(q_0, abab) = \delta(q_0, bab) = \delta(q_1, ab) = \delta(q_2, b) = q_2$

A non-accepting state. Hence abab $\notin$ L(M). If the input is a^3b, then
$\delta(q_0, a^3b) = \delta(q_0, a^2b) = \delta(q_0, ab) = \delta(q_0, b) = q_1$

Hence $a^3b \in L(M)$.

Ques 32) Given $\Sigma = \{a, b\}$. Construct a DFA that recognize the language $L = \{b^m ab^n : m, n > 0\}$.

Ans: Given language has a string with exactly one 'a' in between b's. So minimum requirement of states required to accept the string in L is 4. DFA with dead state q_4 (non-accepting state) is given by **figure 1.19**.

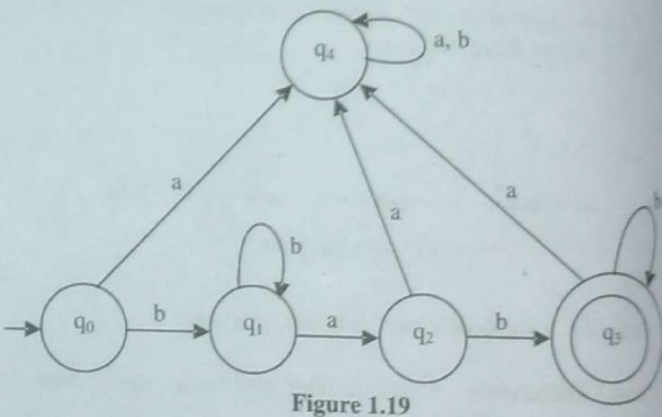
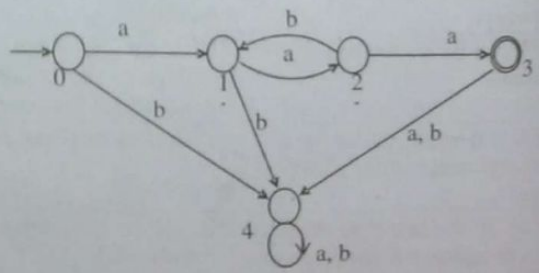


Figure 1.19

For an input $b^2a\,b^3$, we have
$\delta(q_0, b^2ab^3) = \delta(q_1, bab^3) = \delta(q_2, ab^3) = \delta(q_2, b^3) = \delta(q_3, b^2)$
$= \delta(q_3, b) = q_3$, an accepting state.

Hence $b^2\,ab^3 \in L(M)$.

Ques 33) Consider the DFA given below and identify the L accepted by the machine.



Ans: The transition table for above DFA is given by:

Table

Status	Inputs	
	a	b
S_0	S_1	S_4
S_1	S_2	S_4
S_2	S_3	S_1
S_3	S_4	S_4
S_4	S_4	S_4

Let S_0 as starting states and S_4 as final states then, 'aaa' string is accepted

Similarly aa ba a, aa b aba a, aa b aba ba a, ... accepted.

i.e., $a^2(ba)^*a$ accepted by the given DFA.

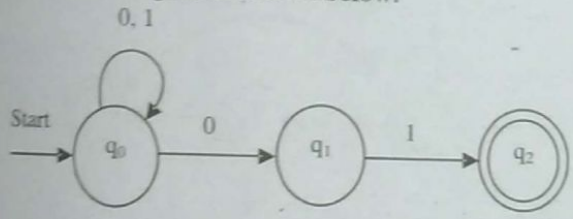
Ques 34) Design the NFA accepting the string that end in 01.

Ans: An NFA accepting strings that end in 01 is given by:
$A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$

Where the transition function δ is given by the **table below:**

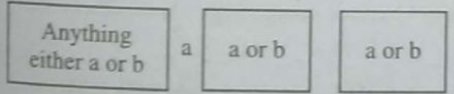
	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	ϕ	$\{q_2\}$
$*q_2$	ϕ	ϕ

The transition diagram is **shown below:**



Ques 35) Construct a NFA for the language L which accept all the strings in which the third symbol from right end is always 'a' over $\Sigma = \{a, b\}$.

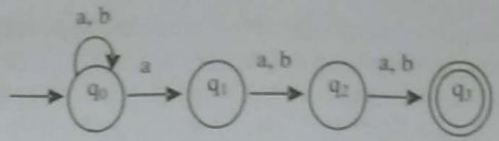
Ans: The strings in such a language are of the form



Where the transition function δ is given by the **table below:**

	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\{q_2\}$
q_2	$\{q_3\}$	$\{q_3\}$
$*q_3$	ϕ	ϕ

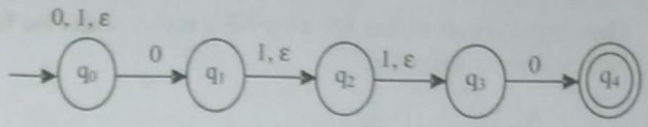
Thus we get third symbol from right end as 'a' always, The NFA then can be



The above figure is a NFA because in state q_0 with input 'a' we can go to either state q_0 or state q_1.

Ques 36) Design an NFA for the language of all strings over {0, 1} that end with one of 0110, 010, and 00.

Ans: An NFA for the language of all strings over {0, 1} that end with one of 0110, 010, and 00 is **shown below:**



Ques 37) Construct NFA for the language
$L = \{0101^n \cup 0100 \mid n \geq 0\}$ Over $\Sigma = \{0, 1\}$.

Ans: Here in language L, first three symbols are common, i.e., 010. Hence the NFA can be drawn as shown in **figure 1.20:**

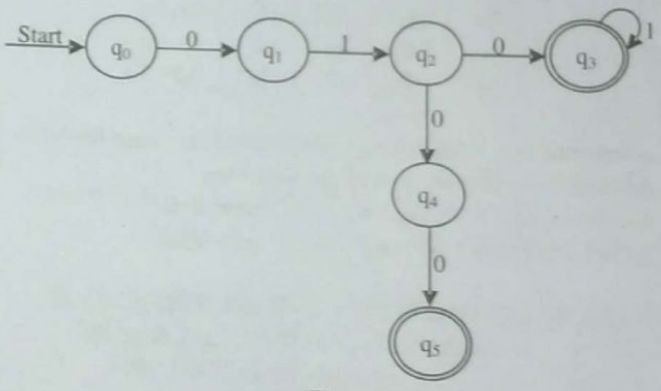


Figure 1.20

The states q_3 and q_5 are final states accepting 0101^n and 0100 respectively. The NFA then can be denoted by,
$M = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \delta, q_0, \{q_3, q_5\})$

Ques 38) Write the process of eliminating epsilon transition.

Or

Describe the process of converting NFA with epsilon into NFA without epsilon.

Ans: Process of Eliminating Epsilon Transition /Conversion of NFA with Epsilon to NFA without Epsilon

Let $M = (Q, \delta, q_0, F)$ be an $\in$-NFA. There are some steps for the conversion of NFA with $\in$-transition to NFA without $\in$-transition:

Step 1: Find the state of NFA without $\in$-transition including initial states and final states.

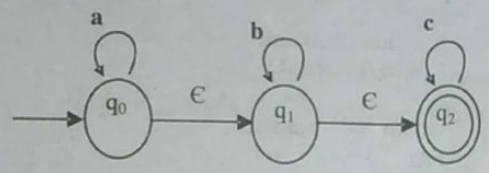
Step 2: There will be same number of states. The initial state NFA without $\in$-transition will be $\in$-closure of initial state of $\in$-NFA i.e., in the figure given below, the

ϵ-closure $(q_0) = \{q_0, q_1, q_2\}$ initial state for NFA without ϵ-transition, rest of the state are:
ϵ-closure $(q_1) = \{q_1, q_2\}$
ϵ-closure $(q_2) = \{q_2\}$

Step 3: The final states of NFA without ϵ-transitions are all those new states which contains final state of ϵ-NFA as member.

Step 4: Now find out δ' to find out the transitions for NFA without ϵ-transition. Ignore Φ entries and ϵ-transitions column.

Ques 39) Convert of the following NFA with ϵ-transition to NFA without ϵ-transition.



Ans: From the above transition diagram,

δ/ϵ	a	b	c	ϵ
q_0	$\{q_0\}$	$\{\Phi\}$	$\{\Phi\}$	$\{q_1\}$
q_1	$\{\Phi\}$	$\{q_1\}$	$\{\Phi\}$	$\{q_2\}$
$*q_2$	$\{\Phi\}$	$\{\Phi\}$	$\{q_2\}$	$\{\Phi\}$

ϵ-closure $(q_0) = \{q_0, q_1, q_2\} = q_a$ new initial state for NFA without ϵ-transition, rest of the states are
ϵ-closure $(q_1) = \{q_1, q_2\} = q_b$ new state
ϵ-closure $(q_2) = \{q_2\} = q_c$ new state

$\delta'(\{q_0, q_1, q_2\}, a) = \delta'(q_a, a) = \epsilon$-closure $(\delta(q_0, q_1, q_2), a)$
$= \epsilon$-closure $(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a))$
$= \epsilon$-closure $(q_0 \cup \Phi \cup \Phi)$
$= \epsilon$-closure (q_0)
$= \{q_0, q_1, q_2\}$

$\delta'(\{q_0, q_1, q_2\}, b) = \delta'(q_a, b) = \epsilon$-closure $(\delta(q_0, q_1, q_2), b)$
$= \epsilon$-closure $(\delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b))$
$= \epsilon$-closure $(\Phi \cup \{q_1\} \cup \Phi)$
$= \epsilon$-closure (q_1)
$= \{q_1, q_2\}$

$\delta'(\{q_0, q_1, q_2\}, c) = \delta'(q_a, c) = \epsilon$-closure $(\delta(q_0, q_1, q_2), c)$
$= \epsilon$-closure $(\delta(q_0, c) \cup \delta(q_1, c) \cup \delta(q_2, c))$
$= \epsilon$-closure $(\Phi \cup \Phi \cup q_2)$
$= \epsilon$-closure (q_2)
$= \{q_2\}$

$\delta'(\{q_1, q_2\}, a) = \delta'(q_b, a) = \epsilon$-closure $(\delta(q_1, q_2), a)$
$= \epsilon$-closure $(\delta(q_1, a) \cup \delta(q_2, a))$
$= \epsilon$-closure $(\Phi \cup \Phi)$
$= \epsilon$-closure (Φ)
$= \Phi$

$\delta'(\{q_1, q_2\}, b) = \delta'(q_b, b) = \epsilon$-closure $(\delta(q_1, q_2), b)$
$= \epsilon$-closure $(\delta(q_1, b) \cup \delta(q_2, b))$
$= \epsilon$-closure $(q_1 \cup \Phi)$
$= \epsilon$-closure (q_1)
$= \{q_1, q_2\}$

$\delta'(\{q_1, q_2\}, c) = \delta'(q_b, c) = \epsilon$-closure $(\delta(q_1, q_2), c)$
$= \epsilon$-closure $(\delta(q_1, c) \cup \delta(q_2, c))$
$= \epsilon$-closure $(\Phi \cup q_2)$
$= \epsilon$-closure (q_2)
$= \{q_2\}$

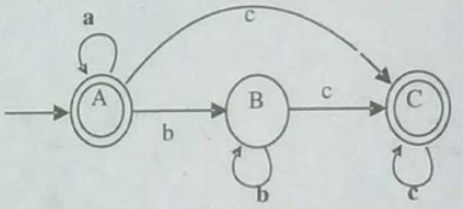
$\delta'(\{q_2\}, a) = \delta'(q_c, a) = \epsilon$-closure $(\delta(q_2), a)$
$= \Phi$

$\delta'(\{q_2\}, b) = \delta'(q_c, b) = \epsilon$-closure $(\delta(q_2), b)$
$= \Phi$

$\delta'(\{q_2\}, c) = \delta'(q_c, c) = \epsilon$-closure $(\delta(q_2), c)$
$= \epsilon$-closure (q_2)
$= \{q_2\}$

Now we can draw the transition table for NFA without ϵ-transition.

	δ/ϵ	a	b	c
A	$\{q_0, q_1, q_2\}*$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
B	$*\{q_1, q_2\}$	$\{\Phi\}$	$\{q_1, q_2\}$	$\{q_2\}$
C	$*\{q_2\}$	$\{\Phi\}$	$\{\Phi\}$	$\{q_2\}$



Ques 40) Change the following NFA with epsilon into NFA without epsilon.

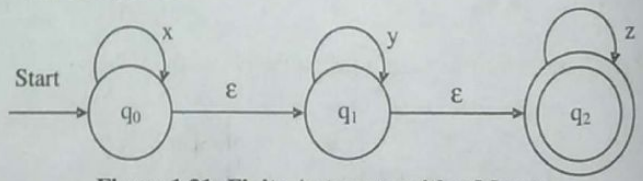


Figure 1.21: Finite Automata with ε Moves

Ans: The important aspect for building the transition table is to compute the δ function. The δ the transition function, maps $Q \times (\Sigma \cup \{\varepsilon\})$ to 2^Q. The intention is that $\delta(q, a)$ will consist of all states p such that there is a transition labeled a from q to p where a is either ε or the symbol in ε. Let us design the transition table with δ function for **figure 1.21**.

	x	y	z	ε
q_0	$\{q_0\}$	ϕ	ϕ	$\{q_1\}$
q_1	ϕ	$\{q_1\}$	ϕ	$\{q_1\}$
q_2	ϕ	ϕ	$\{q_2\}$	ϕ

Let $\delta''(q, w)$ will be all states p such that one can go from q to p along a path labeled w, sometime including edges labeled ε. While constructing δ'' we have to compute the set of states reachable from a given state q using ε transitions only. Let us use ε CLOSURE (q) to denote the set of all vertices p such that there is a path from q to p labeled ε.

The δ'' can be interpreted as follows:

$\delta''(q, \varepsilon) = \varepsilon$-CLOSURE (q)

For win Σ^* and a in Σ, $\delta''(q, wa) = \varepsilon$ CLOSURE (p)
Where p = [p | for some r in $\delta''(q, w)$p is in $\delta(r, a)$]

For the transition table

$\delta''(q_0, \varepsilon) = \varepsilon$-CLOSURE $\{q_0\}$
$= \{q_0, q_1, q_2\}$

Thus,

$\delta''(q_0, x) = \varepsilon$-CLOSURE $(\delta(\delta''(q_0, \varepsilon), x))$
$= \varepsilon$-CLOSURE $(\delta(\{q_0, q_1, q_2\}, x))$
$= \varepsilon$-CLOSURE $(\delta(q_0, x) \cup \delta(q_1, x) \cup \delta(q_2, x))$
$= \varepsilon$-CLOSURE $(\{q_0\} \cup \phi \cup \phi$
$= \varepsilon$-CLOSURE $[\{q_0\}]$
$\delta(((q0, x) = \{q_0, q_1, q_2\}$
$\delta(((q0, xy) = \varepsilon$-CLOSURE $(\delta(\delta(((q0, x), y)$
$= \varepsilon$-CLOSURE $(\delta(\{q0, q1, q2\}, y))$
$= \varepsilon$-CLOSURE $(\{q1, q2\})$
$= \{q_1, q_2\}$

Thus, the transition table can be drawn as:

	x	y	z
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	ϕ	$\{q_1, q_2\}$	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_2\}$

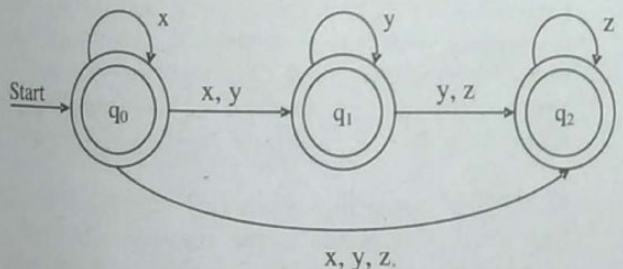


Figure 1.22: NFA without ε

Ques 41) Obtain an equivalent NFA without ε-transitions from the NFA shown in figure 1.23:

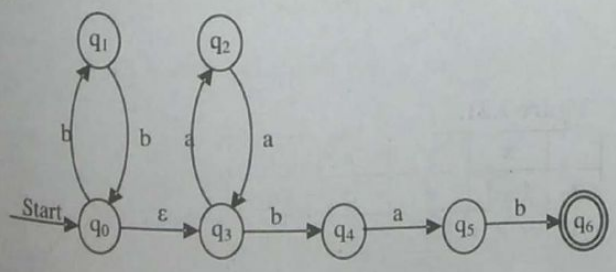


Figure 1.23

Ans: One just needs to eliminate the ε-transition between q_0 and q_3. This will add two new edges from q_0 to q_2 on a and q_0 to q_4 on b. Moreover q_3 will become initial state as q_0 is an initial state. The NFA without ε-transition is as shown in **figure 1.24**:

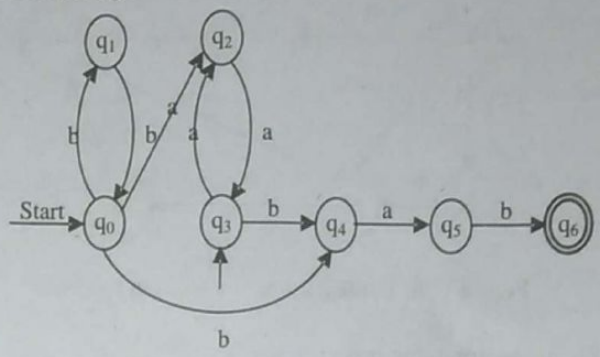


Figure 1.24: NFA without ε-Transitions for NFA given in figure 1.23

Ques 42) Construct an NFA without ε -moves corresponding to the following NFA:

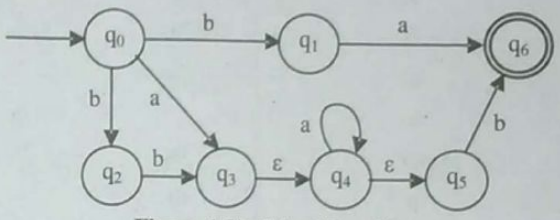


Figure 1.25: FA with ε -Moves

Ans: Constructed NDFA with ε -moves is:

$M = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \{a, b\}, \delta, q_0, \{q_6\})$

Let us determine ε-closure of all the states of M, i.e., q_0, q_1, q_2, q_3, q_4, q_5, and q_6 as below:

ε-closure $(q_0) = \{q_0\}$
ε-closure $(q_1) = \{q_1\}$
ε-closure $(q_2) = \{q_2\}$
ε-closure $(q_3) = \{q_3, q_4, q_5\}$
ε-closure $(q_4) = \{q_4, q_5\}$
ε-closure $(q_5) = \{q_5\}$
ε-closure $(q_6) = \{q_6\}$

Let the NDFA without ε-moves equivalent to NDFA having ε -moves be defined as:

$M' = (\{\{q_0\}, \{q_1\}, \{q_2\}, \{q_3, q_4, q_5\}, \{q_4, q_5\}, \{q_5\}, \{q_6\}\}, \{a, b\}, \delta', q_0', F')$,

Where, δ' is defined as:

$\delta'(\{q_0\}, a) = \{q_3, q_4, q_5\}$, $\delta'(\{q_0\}, b) = \{q_1\} \cup \{q_2\} = \{q_1\}, \{q_2\}$
$\delta'(\{q_1\}, a) = \{q_6\}$, $\delta'(\{q_1\}, b) = \{\phi\}$
$\delta'(\{q_2\}, a) = \{\phi\}$, $\delta'(\{q_2\}, b) = \{q_3, q_4, q_5\}$
$\delta'(\{q_3, q_4, q_5\}, a) = \{q_4, q_5\}$, $\delta'(\{q_3, q_4, q_5\}, b) = \{q_6\}$
$\delta'(\{q_4, q_5\}, a) = \{q_4, q_5\}$, $\delta'(\{q_4, q_5\}, b) = \{q_6\}$
$\delta'(\{q_5\}, a) = \{\phi\}$, $\delta'(\{q_5\}, b) = \{q_6\}$
$\delta'(\{q_6\}, a) = \{\phi\}$, $\delta'(\{q_6\}, b) = \{\phi\}$

Let us assume, $\{q_0\} = q_A$, $\{q_1\} = q_B$, $\{q_2\} = q_C$, $\{q_3, q_4, q_5\} = q_D$, $\{q_4, q_5\} = q_E$, $\{q_5\} = q_F$, $\{q_6\} = q_G$.

By using above transitions δ', following finite automaton (NDFA without ε-moves) can be drawn:

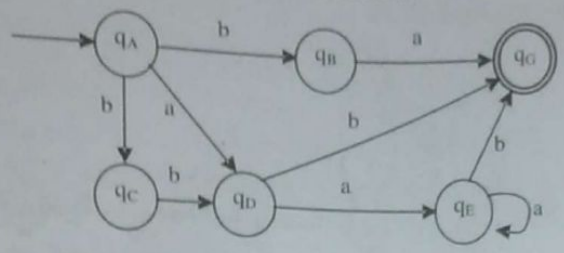


Figure 1.26: Finite Automaton without ε -Moves

The transition table is shown below:

Table 1.1: Transition Table of NDFA without ε- Moves

Present State	Inputs	
	a	b
→q_A	q_D	q_B, q_C
q_B	q_F	–
q_C	–	q_D
q_D	q_E	q_G
q_E	q_E	q_G
(q_G)	–	–

Ques 43) Give Thompson's construction algorithm.

Or

Write the rules to convert regular expression into DFAs

Ans: From Regular Expression to DFAs

The simplest algorithm for translating a regular expression into a DFA proceeds via an intermediate construction, in which an NFA is derived from the regular expression, and then the NFA is used to construct an equivalent DFA.

There exist algorithms that can translate a regular expression directly into a DFA, but they are more complex, and the intermediate construction is also of some interest.

Thus, we concentrate on describing two algorithms:

1) One that translates a regular expression into an NFA and
2) Second that translates an NFA into a DFA.

The process of constructing a scanner can be automated in three steps, as illustrated by the following diagram:

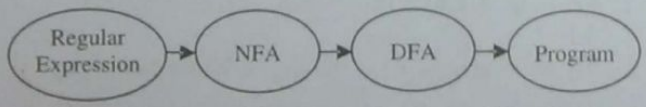


Figure 1.27: Conversion of Regular Expression to DFA

From Regular Expression to NFA(Thompson's Construction)

The set of rules for this construction are as follows:

1) For ε, the NFA is as shown in **figure 1.28(a)** consisting of two states – a start state and a final state. The transition is labelled by ε.
2) For any alphabet symbol α in the alphabet set ∑, the NFA is constructed as in **figure 1.28(b)**. It is similar to the ε-case, with the labelling done with α instead of ε.
3) For the regular expression $r_1|r_2$, if $N(r_1)$ be the NFA for r_1 and $N(r_2)$ be the NFA for r_2, then the NFA $N(r_1|r_2)$ is constructed as in **figure 1.28 (c)**. In this case, a new start state and final state are added. ε-transitions are introduced from the new start state to the start states of $N(r_1)$ and $N(r_2)$.

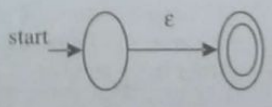
   Similarly, ε-transitions are added from the final state of $N(r_1)$ and $N(r_2)$ to the newly created final state. The final states of $N(r_1)$ and $N(r_2)$ cease to be final states any more.
4) For the regular expression r_1r_2, the NFA $N(r_1r_2)$ is constructed by merging the NFAs $N(r_1)$ and $N(r_2)$. The final state of $N(r_1)$ is merged with the start state of $N(r_2)$. It is shown in **figure 1.28 (d)**.

   The transitions emanating from the start state of $N(r_2)$ now originates at the final state of $N(r_1)$. The start state of $N(r_1)$ becomes the start state of the new NFA and the final state of $N(r_2)$ is the final state of new NFA. Final state of $N(r_1)$ ceases to be the final state any more.
5) For the regular expression r*, NFA $N(r^*)$ is constructed from the NFA N(r) of r as follows:

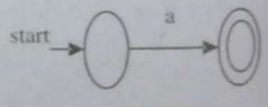
   First, a new start state and a new final state are introduced. Then, ε-transitions are added from the new start state to the start state of N(r), from final state of N(r) to the new final state, from final state of N(r) back to the start state of N(r), and from the new start state to the new final state.

   Addition of ε-transition from new start state to new final state corresponds to the zero-occurrence of r, whereas, the ε-transition from the final to initial state of N(r) corresponds to the repeated occurrence of r. The situation has been shown in **figure 1.28 (e)**.
6) If N(r) be the NFA for a regular expression r, it is also the NFA for the parenthesised expression (r).

   The rules are also shown in **figure 1.28**:



(a) NFA for ε



(b) NFA for a

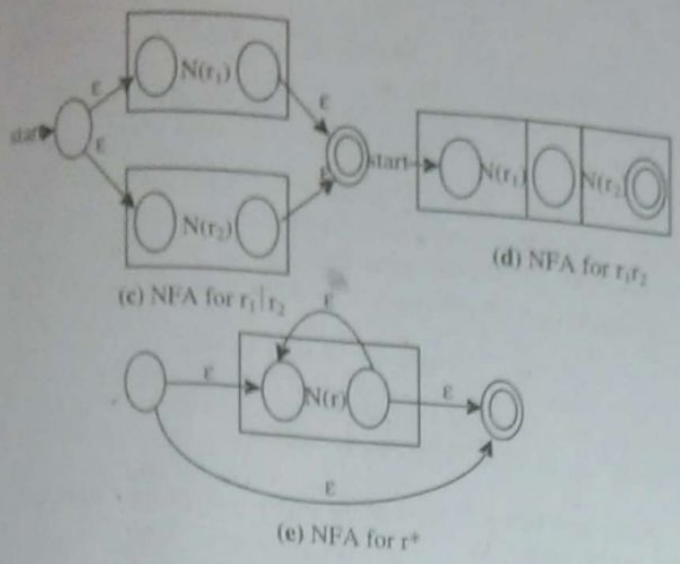
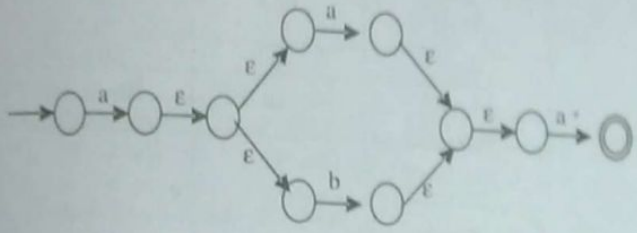
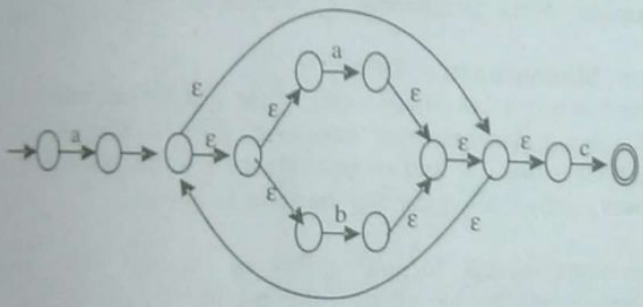


Figure 1.28: Regular Expression to NFA

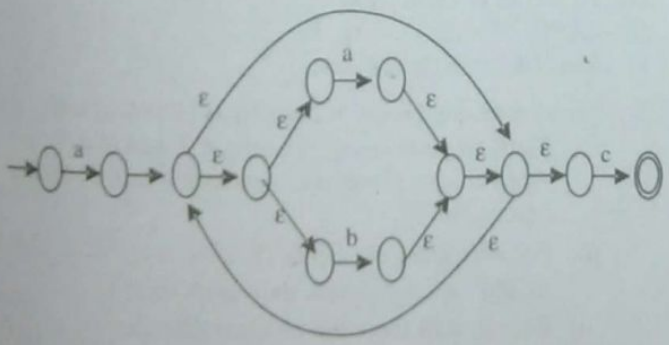
For example, the language generated by the regular expression a(a + b)a is accepted by the following NFA:



Likewise, the language generated by the regular expressions a(a + b)*c is accepted by the following NFA:



Finally, the language generated by the regular expression (a + b)a*b is accepted by the following NFA:



One may notice that these non-deterministic finite automata are probably not the same machines that one would make by hand – they have a large number of states that could be combined to create a smaller machine. The machines do, however, accept the correct languages.

Ques 44) Let M = ({q_0, q_1}, {0, 1}, δ, q_0, {q_1}) be NFA where δ(q_0, 0) = {q_0, q_1}, δ(q_0,1) = {q_1} δ(q_1, 0) = ϕ, δ(q_1, 1) = {q_0, q_1}. Construct its equivalent DFA.

Ans: Let the DFA $M' = (Q, \Sigma, \delta', q'_0, F')$. Now, the δ' function will be computed as follows:

As $\delta(q_0, 0) = \{q_0, q_1\}$ $\delta'([q_0], 0) = [q_0, q_1]$

As in NFA the initial state is q_0, the DFA will also contain the initial state $[q_0]$

Let us draw the transition table for δ function for a given NFA:0

Table: δ Function for NFA

		0	1	
δ(→q_0, 0) ⇒	→ q_0	{q_0, q_1}	{q_1}	⇐ δ(q_0, 1)
δ(q_1, 0) ⇒	(q_1)	ϕ	{q_0, q_1}	⇐ δ(q_1, 1)

From the transition table we can compute that there are $[q_0]$, $[q_1]$, $[q_0, q_1]$ states for its equivalent DFA. We need to compute the transition from state $[q_0, q_1]$

$\delta([q_0, q_1], 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$

$= \{q_0, q_1\} \cup \phi$

$= \{q_0, q_1\}$

So, $\delta'([q_0, q_1], 0) = [q_0, q_1]$

Similarly,

$\delta(\{q_0, q_1\}, 1) = \delta(q_0, 1) \cup \delta(q_1, 1)$

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

$= \{q_0, q_1\}$

So, $\delta'([q_0, q_1], 1) = [q_0, q_1]$

As in the given NFA q_1 a is final state, then in DFA where ever q_1 exists that state becomes a final state. Hence in the DFA final states are $[q_1]$ and $[q_0, q_1]$. Therefore set of final states F = {$[q_1]$, $[q_0, q_1]$}

The equivalent DFA is,

Transition Table for Equivalent DFA

	0	1
→ q_0	$[q_0, q_1]$	$[q_1]$
$[q_1]$	ϕ	$[q_0, q_1]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$

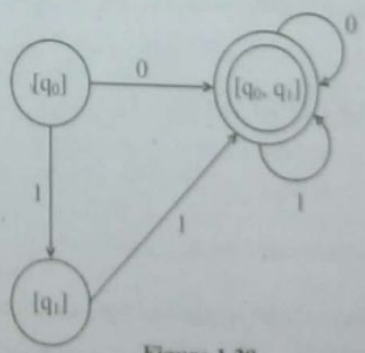


Figure 1.29

Even, we can change the names of the states of DFA
A = $[q_0]$
B = $[q_1]$
C = $[q_0, q_1]$

With these new names the DFA will be as follows:

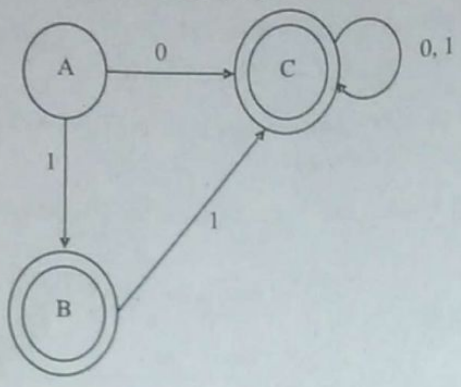


Figure 1.30: Equivalent DFA

Ques 45) Convert the given NFA to DFA.

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	q_0
q_1	q_2	q_1
q_2	q_3	q_3
(q_3)	ϕ	q_2

Ans: As we know, first we will compute δ' function
$\delta(\{q_0\}, 0) = \{q_0, q_1\}$

Hence $\delta'([q_0], 0) = [q_0, q_1]$

Similarly, $\delta(\{q_0\}, 1) = \{q_0\}$

Hence $\delta'([q_0], 1) = [q_0]$

Thus we have got a new state $[q_0, q_1]$.

Let us check how it behaves on input 0 and 1.

$$\delta'([q_0, q_1], 0) = \delta([q_0], 0) \cup \delta([q_1], 0)$$
$$= \{q_0, q_1\} \cup \{q_2\}$$
$$= \{q_0, q_1, q_2\}$$

Hence a new state is generated, i.e., $[q_0, q_1, q_2]$

Similarly
$$\delta'([q_0, q_1], 1) = \delta([q_0], 1) \cup \delta([q_1], 1)$$
$$= \{q_0\} \cup \{q_1\}$$
$$= \{q_0, q_1\}$$

No new state is generated here.

Again δ' function will be computed for $[q_0, q_1, q_2]$, the new state being generated:

State	0	1
$[q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_0, q_1]$
$[q_0, q_1, q_2]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_3]$

As, you have observed in the above table for a new state $[q_0, q_1, q_2]$ the input 0 will give a new state $[q_0, q_1, q_2, q_3]$ and input 1 will give a new state $[q_0, q_1, q_3]$ because

$$\delta'([q_0, q_1, q_2], 0) = \delta'([q_0], 0) \cup \delta'([q_1], 0) \cup \delta'([q_2], 0)$$
$$= \{q_0, q_1\} \cup \{q_2\} \cup \{q_3\}$$
$$= \{q0, q_1, q_2, q_3\}$$
$$= [q_0, q_1, q_2, q_3]$$

Same procedure is applied for input 1. Thus the final DFA is as given below:

Table: DFA for Example

State	0	1
$\rightarrow [q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_1]$	$[q_2]$	$[q_1]$
(q_2)	$[q_3]$	$[q_3]$
$[q_3]$	ϕ	$[q_2]$
$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_0, q_1]$
$[q_0, q_1, q_2]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_3]$
(q_0, q_1, q_3)	$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$
(q_0, q_1, q_2, q_3)	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3]$

Ques 46) Write the algorithm to minimise the DFA.

Ans: Minimization of DFA
Another important consequence of the test for equivalence of states is that one can "minimise" DFA's. That is, for each DFA one 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 one choose, this minimum-state DFA is unique for the language.

The algorithm is as follows:
Given DFA M = $(\Sigma, Q, \delta, q_0, F)$.

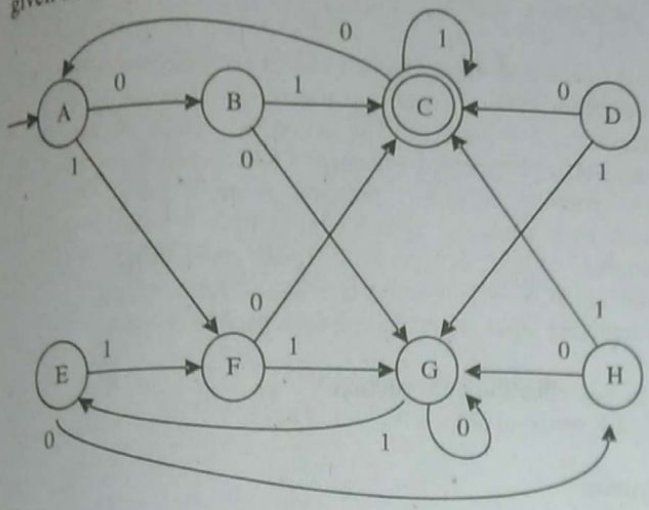
1) Remove unreachable states.
2) Set-up marking tables of pairs (q, q') where $q \neq q'$.
   i) Mark all pairs (q, q') where $q \in F$ and $q' \notin F$ (and *vice versa*). (These are the states which cannot be equivalent.)
   ii) For each unmarked pair (q, q') and $a \in \Sigma$ if $(\delta(q, a), \delta(q', a))$ is marked, then mark (q, q').
   iii) Repeat until there are no more changes.
3) Combine states.
   For each unmarked (q, q')
   i) If $pa \rightarrow q'$ then add $pa \rightarrow q$.
   ii) If $q'a \rightarrow p$ then add $qa \rightarrow p$.

iii) Remove q′.

iv) Remove $pa \to q'$, $q'a \to p$ for all $p \in Q$ and $a \in \Sigma$ (i.e., remove q′ and all transitions leading to and from q′).

4) Resulting DFA is minimal.

For example, one can apply algorithm to the automaton given below:



Step 1: We have one unreachable state, state D. We remove this state and proceed to Step 2.

Step 2: We first mark pairs of final and non-final states, getting the following tableau:

B						
C	X	X				
E			X			
F			X			
G			X			
H			X			
	A	B	C	E	F	G

In the first iteration we examine all unmarked pairs. **For example,** for pair A, B, we get $\delta(A, 1) = F$ and $\delta(B, 1) = C$, and the pair C, F is marked, so we mark A, B too. After doing it for all pairs, we get the following tableau:

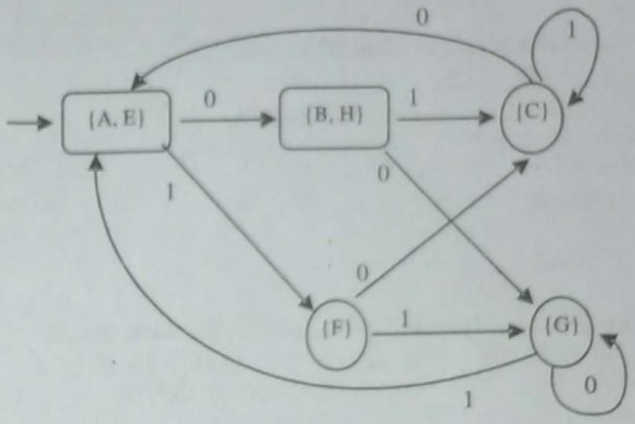
B	X					
C	X	X				
E		X	X			
F	X	X	X	X		
G		X	X		X	
H	X		X	X	X	X
	A	B	C	E	F	G

In the next iteration, we examine the remaining pairs. **For example,** we will mark pair A, G because $\delta(A, 0) = B$ and $\delta(G, 0) = G$, and the pair B, G is marked. When we are done we get the following tableau:

B	X					
C	X	X				
E		X	X			
F	X	X	X	X		
G	X	X	X	X	X	
H	X		X	X	X	X
	A	B	C	E	F	G

One more iteration will be executed now, but no new distinguishable pairs will be discovered. So we are done with Step 2 now.

Step 3: We group the states into the equivalence classes. Since A, E are equivalent and B, H are equivalent, the classes are – {A, E}, {B, H}, {C}, {F}, {G}. The minimal automaton $\hat{A}$ is:



Correctness: To justify correctness, one needed to prove a couple of things. First, we need to show that we compute the equivalence classes correctly and that $\hat{A}$ is a well-defined DFA. Then, we need to show that it accepts the same language as A, Finally, we need to show that there is no DFA A' equivalent to A with fewer states.

Ques 47) Define recognition of token with example.

Ans: Recognition of Token

For a programming language there are various types of tokens such as identifier, keywords, constants, and operators and so on. The token is usually represented by a pair token type and token value.

Token type	Token value

Token Representation

The token type tells us the category of token and token value gives us the information regarding token. The token value is also called token attribute. During lexical analysis process the symbol table is maintained. The token value can be a pointer to symbol table in case-of identifier and constants. The lexical analyser reads the input program and generates a symbol table for tokens.

For example, we will consider some encoding of tokens as follows:

Token	Code	Value
if	1	–
else	2	–
while	3	–
for	4	–
identifier	5	Ptr to symbol table
constant	6	Ptr to symbol table
<	7	1
<=	7	2
>	7	3
>=	7	4
!=	7	5
(	8	1
)	8	2
+	9	1
–	9	2
=	10	–

Consider a program code as:

```
if (a<10)
    i=i+2;
else
    i=i=2;
```

Our lexical analyser will generate following token stream.
1, (8,1), (5,100), (7,1), (6,105), (8,2), (5,107), 10, (5,107), (9,1), (6,110), 2,(5,107), 10, (5,107), (9,2), (6,110).

The corresponding symbol table for identifiers and constants will be,

Location Counter	Type	Value
100	identifier	a
⋮	⋮	
105	constant	10
⋮	⋮	
107	identifier	i
⋮	⋮	
110	constant	2

In above example scanner scans the input string and recognizes "if" as a keyword and returns token type as 1 since in given encoding code 1 indicates keyword "if" and hence 1 is at the beginning of token stream.

Next is a pair (8,1) where 8 indicates parenthesis and 1 indicates opening parenthesis '('. Then we scan the input 'a' it recognizes it as identifier and searches the symbol table to check whether the same entry is present. If not it inserts the information about this identifier in symbol table and returns 100.

If the same identifier or variable is already present in symbol table then lexical analyser does not insert it into the table instead it returns the location where it is present.

Ques 48) Explain lexical analyser generator(LEX) and its structure. **Or**

What is a lex compiler? Write its specification.

Ans: LEX

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). Lex is commonly used with the **Yacc** parser generator.

Lex, tool is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression. The action is a piece of code, which is to be executed whenever a token specified by the corresponding regular expression is recognized. LEX helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex Specification or Structure

A LEX program has the following form:

Syntax:

```
Declaration
% %
Translation Rules
% %
Auxiliary Functions Definitions
```

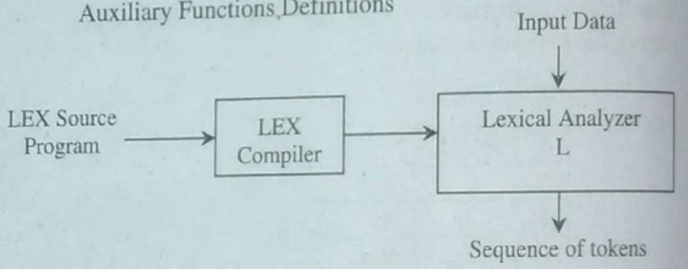
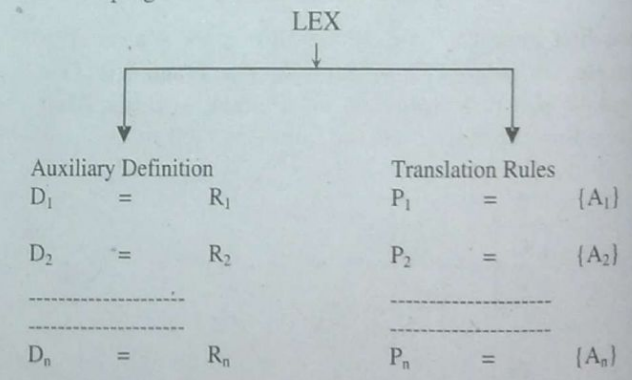


Figure 1.31: LEX Generating Tokens

The LEX program structure is **shown below**.



Parts of Lex Program

LEX program consist two parts:

1) **Auxiliary Definition:** Each D_i is a distinct name and each R_i is a regular expression, whose symbols are chosen from $\sum \cup \{D_1, D_2, \ldots\ldots D_{i-1}\}$, i.e., characters or previously defined names. The D_i's are shorthand names for Regular expressions.

For example,

(D_1) : letter = A | B | .....|Z.
(D_2) : digit = 0 | 1 |.....|9.
(D_3) : identifier = letter (letter/digit)*

2) **Translation Rules:** Where each P_i is a regular expression called a token pattern, over the alphabet consisting of Σ and auxiliary definitions names.

The token patterns describe the form of the tokens. The token patterns describe the form of the tokens.

For example,

ab* (for input symbol a, b)
if, then, else (for keywords etc.)

Each A_i is a program fragment describing what action the lexical analyser should take when token P_i is found.

For example, let us consider the LEX program that defines the set of token patterns.

Auxiliary Definition

Letter = A |B|.....|Z
Digit = 0 |1|.....|9

Transition Rules

LEX Program

	Token Patterns	Action
P_1	BEGIN	{return 1}
	END	{return 2}
	IF	{return 3}
	THEN	{return 4}
	ELSE	{return 5}
P_2	Letter (letter/ digit)*	{Install (), {return 6}
P_3	digit$^+$	{install (), {return 6}
P_4	<	{return (1, 8)}
	<=	{return (2, 8)}
	=	{return (3, 8)}
	<>	{return (4, 8)}
	>	{return (5, 8)}
	>=	{return (6, 8)}

Note: BEGIN is identified in two patterns: "keyword" and "identifier" for resolving these conflicts, priority should be set.

Since the token pattern for keyword "BEGIN" precedes the token pattern for "identifier" in the above list of token pattern, conflict is resolved in favor of the keyword.