Subject Name: **Compiler Design**

Subject Code: **CS-7002**

Semester: **7th**

CS701 Compiler Design

UNIT- 1:

1. INTRODUCTION OF COMPILER

A Compiler is a translator from one language, the input or source language, to another language, the output or target language. Often, but not always, the target language is an assembler language or the machine language for a computer processor.

Note that using a compiler requires a two step process to run a program.

1. Execute the compiler (and possibly an assembler) to translate the source program into a machine language program.
2. Execute the resulting machine language program, supplying appropriate input.

Compiler is a translator program that translates a program written in (HLL) the source program and translates it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.

Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

1.1 Language Processing System

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.

The high-level language is converted into binary language in various phases. A compiler is a program that converts high-level language to assembly language. Similarly, an assembler is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

- User writes a program in C language (high-level language).
- The C compiler compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).
- A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

Preprocessors

Preprocessors are normally fairly simple as in the C language, providing primarily the ability to include files and expand macros. There are exceptions, however. IBM's PL/I, another Algol-like language had quite an extensive preprocessor, which made available at preprocessor time, much of the PL/I language itself (e.g., loops and I believe procedure calls).

Assemblers

Assembly code is a mnemonic version of machine code in which names, rather than binary values, are used for machine instructions, and memory addresses.

Some processors have fairly regular operations and as a result assembly code for them can be fairly natural and not-too-hard to understand. Other processors, in particular Intel's x86 line, have let us charitably say more interesting instructions with certain registers used for certain things.

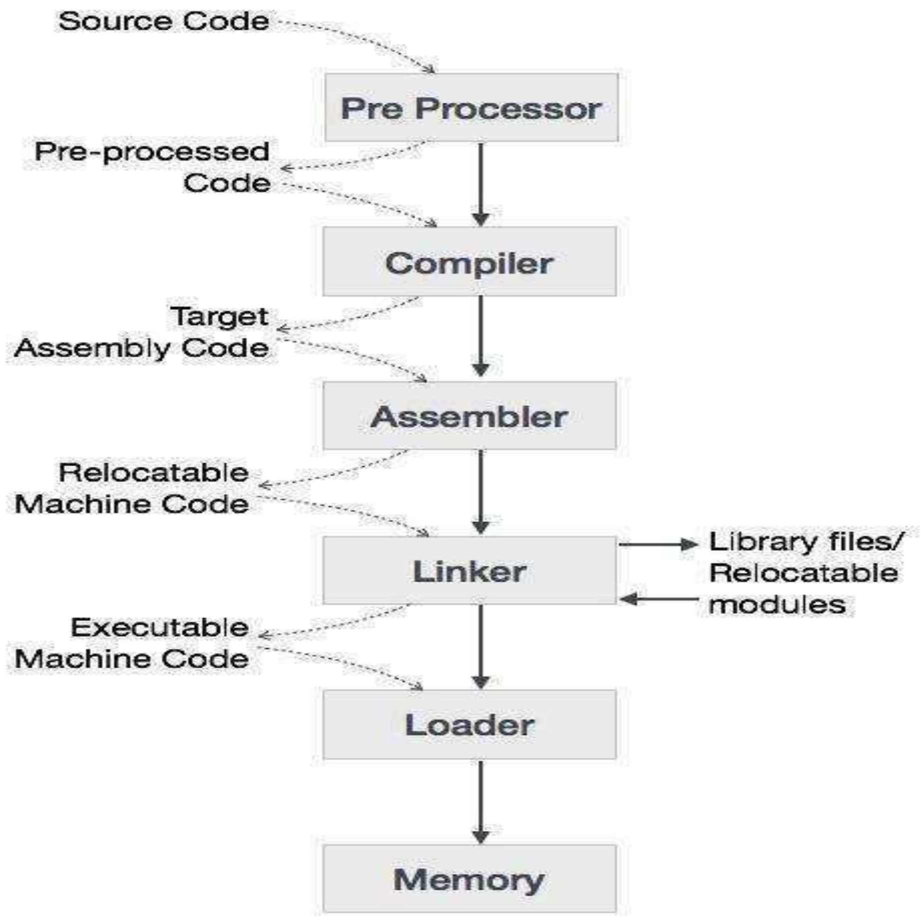

Figure 1.1: Language Processing System

Linkers

The linker has another input, namely libraries, but to the linker the libraries look like other programs compiled and assembled. The two primary tasks of the linker are

- Relocating relative addresses.
- Resolving external references (such as the procedure xor() above).

Loaders

After the linker has done its work, the resulting "executable file" can be loaded by the operating system into central memory. The details are OS dependent. With early single-user operating systems all programs would be loaded into a fixed address (say 0) and the loader simply copies the file to memory. Today it is much more complicated since (parts of) many programs reside in memory at the same time. Hence the compiler/assembler/linker cannot know the real location for an identifier. Indeed, this real location can change.

Compiler

A Compiler is a translator from one language, the input or *source* language, to another language, the output or *target* language. Often, but not always, the target language is an assembler language or the machine language for a computer processor.
Executing a program written n HLL programming language is basically of two parts. The source program must first be compiled translated into an object program. Then the results object program is loaded into a memory executed.

Assembler
Programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

Interpreter
An interpreter is a program that appears to execute a source program as if it were machine language. Execution in Interpreter Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter.
The process of interpretation can be carried out in following phases.
1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Direct Execution

Advantages:
Modification of user program can be easily made and implemented as execution proceeds.
Type of object that denotes various may change dynamically.
Debugging a program and finding errors is simplified task for a program used for interpretation.
The interpreter for the language makes it machine independent.

Disadvantages:
The execution of the program is slower.
Memory consumption is more.

2. MAJOR DATA STRUCTURE IN COMPILER

Symbol Tables
Symbol Tables are organized for fast lookup. Items are typically entered once and then looked up several times. Hash Tables and Balanced Binary Search Trees are commonly used. Each record contains a “name” (symbol) and information describing it.

Simple Hash Table
Hasher translates “name” into an integer in a fixed range- the hash value. Hash Value indexes into an array of lists.
Entry with that symbol is in that list or is not stored at all. Items with same hash value = bucket.

Balanced Binary Search Tree
Binary search trees work if they are kept balanced. Can achieve logarithmic lookup time. Algorithms are somewhat complex. Red-black trees and AVL trees are used. No leaf is much farther from root than any other.

Parse Tree

The structure of a modern computer language is tree-like. Trees represent recursion well. A grammatical structure is a node with its parts as child nodes. Interior nodes are non terminals. The tokens of the language are leaves.

3. BOOTSTRAPPING & PORTING

Bootstrapping is a technique that is widely used in compiler development. It has four main uses:

- It enables new programming languages and compilers to be developed starting from existing ones.
- It enables new features to be added to a programming language and its compiler.
- It also allows new optimizations to be added to compilers.
- It allows languages and compilers to be transferred between processors with different instruction sets

A compiler is characterized by three languages:

- Source Language
- Target Language
- Implementation Language

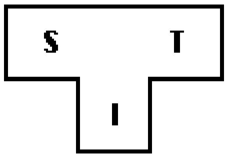

Figure 1.2: T-Diagram

Notation:

${}^{S}C_{I}^{T}$ represents a compiler for Source *S*, Target *T*, implemented in *I*. The *T-diagram* shown above is also used to depict the same compiler.

To create a new language, L, for machine A:

1. Create ${}^{S}C_{A}^{A}$, a compiler for a subset, S, of the desired language, L, using language A, which runs on machine A. (Language A may be assembly language.)

2. Create ${}^{L}C_{S}^{A}$, a compiler for language L written in a subset of L.

3. Compile ${}^{L}C_{S}^{A}$ using ${}^{S}C_{A}^{A}$ to obtain ${}^{L}C_{A}^{A}$, a compiler for language L, which runs on machine A and produces code for machine A.

$${}^{L}C_{S}^{A} \rightarrow {}^{S}C_{A}^{A} \rightarrow {}^{L}C_{A}^{A}$$

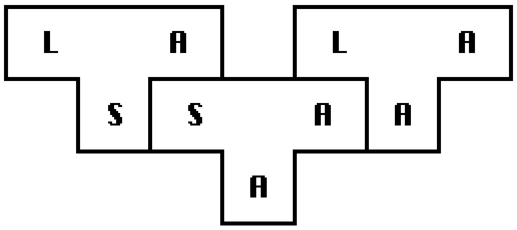

Figure 1.3: Bootstrapping of Compiler

The process illustrated by the T-diagrams is called *bootstrapping* and can be summarized by the equation:

$$L_S A + S_A A = L_A A$$

To produce a compiler for a different machine B:

1. Convert ${}^{L}C_{S}^{A}$ into ${}^{L}C_{L}^{B}$ (by hand, if necessary). Recall that language S is a subset of language L.
2. Compile ${}^{L}C_{L}^{B}$ to produce ${}^{L}C_{A}^{B}$, a *cross-compiler* for L which runs on machine A and produces code for machine B.
3. Compile ${}^{L}C_{L}^{B}$ with the cross-compiler to produce ${}^{L}C_{B}^{B}$, a compiler for language L which runs on machine B.

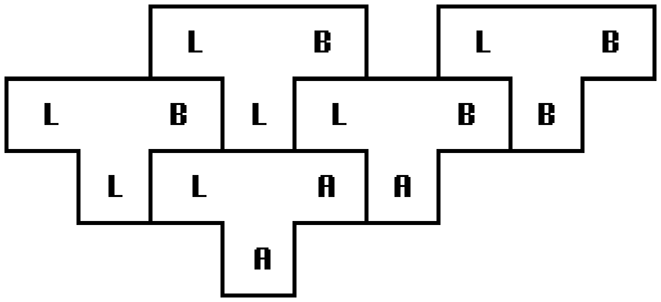

Figure 1.4: Porting of Compiler

4. STRUCTURE OF THE COMPILER: ANALYSIS AND SYNTHESIS MODEL OF COMPILATION

A compiler can broadly be divided into two phases based on the way they compile.

Analysis Model

Known as the front-end of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

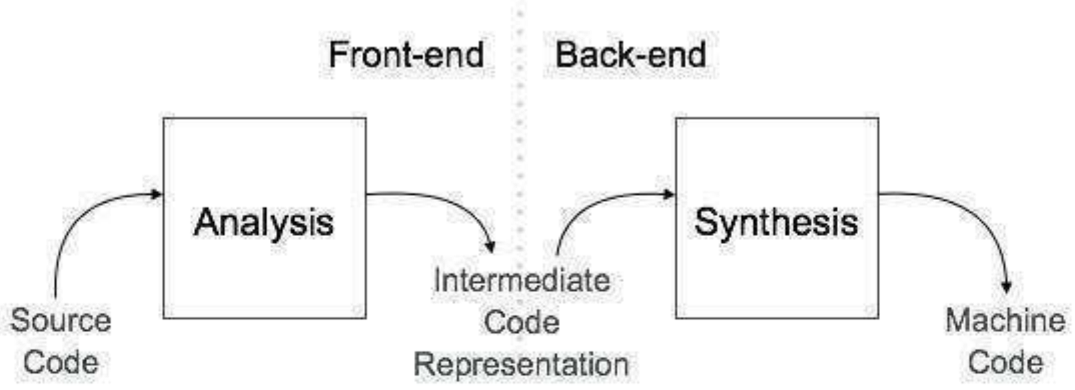

Figure 1.5: Analysis and Synthesis phase of Compiler

Synthesis Model

Known as the back-end of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table.

A compiler can have many phases and passes.

- **Pass** : A pass refers to the traversal of a compiler through the entire program.

- **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

4.1 Various Phase of Compiler

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. Compilation process is partitioned into no-of-sub processes called 'phases'. The phases of a compiler are shown in below.

Lexical Analysis:-
Lexical Analysis or Scanners reads the source program one character at a time, carving the source program into a sequence of character units called tokens.

Syntax Analysis:-
The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

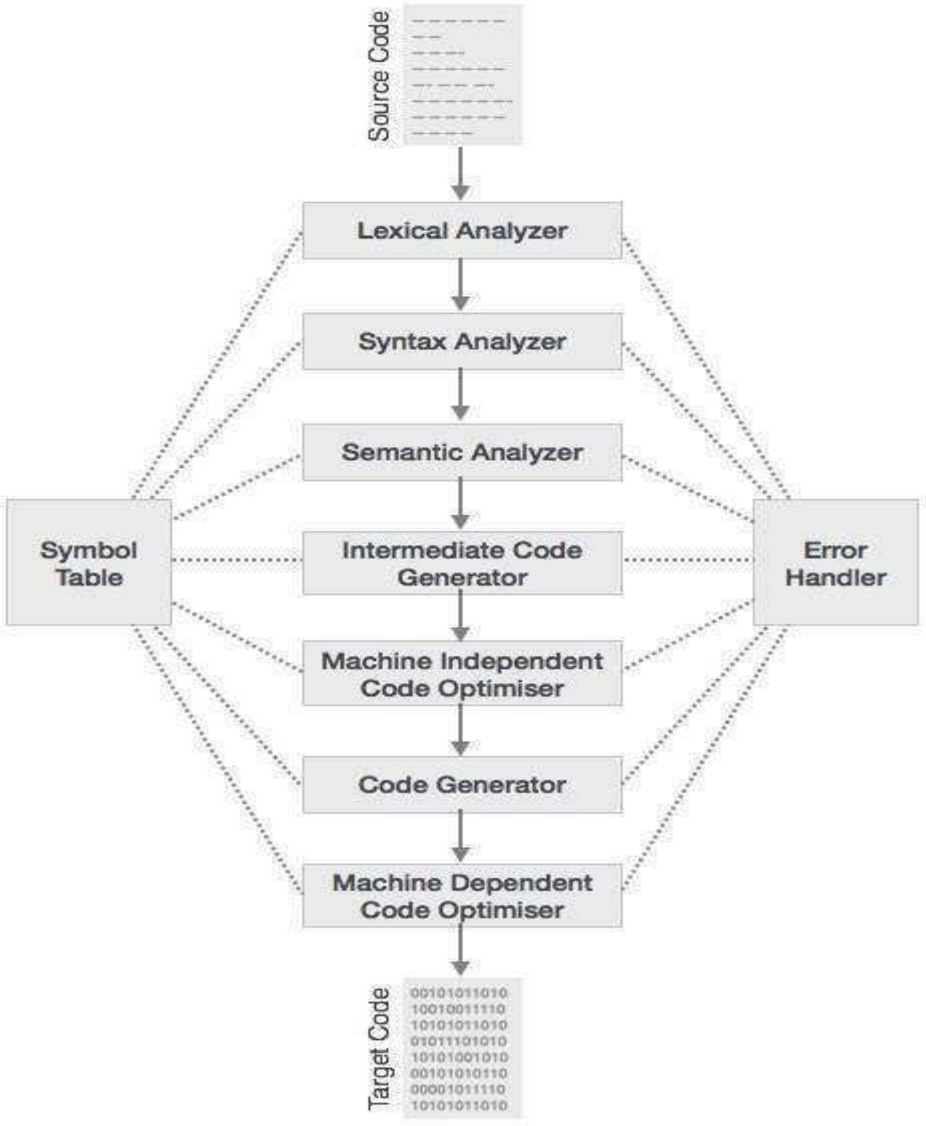

Figure 1.6: Phases of Compiler

Semantic Analysis

There is more to a front end than simply syntax. The compiler needs semantic information, e.g., the types (integer, real, pointer to array of integers, etc) of the objects involved. This enables checking for semantic errors and inserting type conversion where necessary.

Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

Code Optimization :-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-

The last phase of translation is code generation. A number of optimizations to reduce the length of machine language program are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Symbol-Table Management

The symbol table stores information about program variables that will be used across phases. Typically, this includes type information and storage location.

A possible point of confusion: the storage location does **not** give the location where the compiler has stored the variable. Instead, it gives the location where the compiled program will store the variable.

Error Handlers

It is invoked when a flaw error in the source program is detected. The output of LA is a stream of tokens,which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as expression. Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

5. LEXICAL ANALYSIS

To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language. o Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

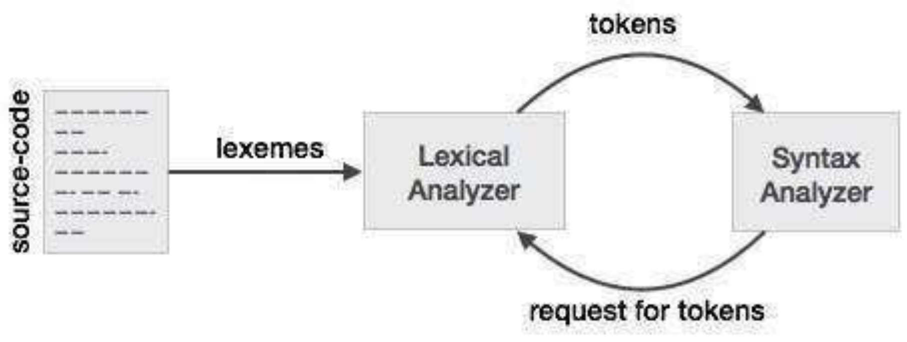

Figure 1.7: Lexical Analysis phase

ROLE OF LEXICAL ANALYZER

The LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.

Upon receiving a 'get next token' command form the parser, the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.

LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

Token

Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are, 1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

Pattern

A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme

A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

5.1 LEXICAL ERRORS

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognize a lexeme as a valid token for you lexer? Syntax errors, on the other side, will be thrown by your scanner when a given set of already recognized valid tokens don't match any of the right sides of your grammar rules. Simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error-recovery actions are:

a. Delete one character from the remaining input.
b. Insert a missing character in to the remaining input.
c. Replace a character by another character.
d. Transpose two adjacent characters.

5.2 LEXICAL ANALYSIS: INPUT BUFFER

The lexical analyzer scans the characters of the source program one a t a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer. Fig. 1.8 shows a buffer divided into two halves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered .we view the position of each pointer as being between the character last read and the character next to be read. In practice each buffering scheme adopts one convention either a pointer is at the symbol last read or the symbol it is ready to read.

The distance which the look ahead pointer may have to travel past the actual token may be large. For example, in a PL/I program we may see:

DECLARE (ARG1, ARG2… ARG *n*)

Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file.

Since the buffer shown in figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, if the look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been grouped into tokens. While we can make the buffer larger if we chose or use another buffering scheme, we cannot ignore the fact that overhead is limited.

BUFFER PAIRS

A buffer is divided into two N-character halves, as shown below

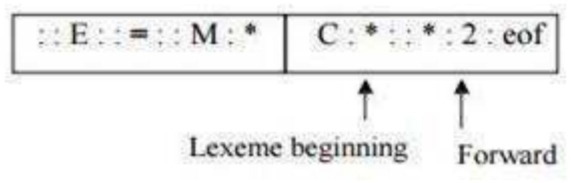

Figure 1.8: An input buffer in two halves

Each buffer is of the same size N, and N is usually the number of characters on one block. E.g., 1024 or 4096 bytes. Using one system read command we can read N characters into a buffer.

If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file.

Two pointers to the input are maintained:

- Pointer lexeme beginning marks the beginning of the current lexeme, whose extent we are attempting to determine.
- Pointer forward scans ahead until a pattern match is found.

Once the next lexeme is determined, forward is set to the character at its right end.

- The string of characters between the two pointers is the current lexeme.
- After the lexeme is recorded as an attribute value of a token returned to the parser, lexeme_beginning is set to the character immediately after the lexeme just found.

Advancing forward pointer:

Advancing forward pointer requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.

Code to advance forward pointer:

if forward at end
of first half then begin reload second half;

```
forward := forward + 1 end
else if forward at end of second half then begin reload second half;
move forward to beginning of first half end
else forward := forward + 1;
```

Sentinels

For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.

The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.

The sentinel arrangement is as shown below:

: : E : : = : : M : * : eof C : * : : * : 2 : eof : : : eof

Figure. 1.9: Sentinels at end of each buffer half

lexeme beginning forward

Note that eof retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end.

Code to advance forward pointer:

```
 forward : = forward + 1;
if forward ↑ = eof then begin
if forward at end of first half then begin reload second half; forward := forward +1
end
else if forward at end of second half then begin reload first half;
move forward to beginning of first half end
else /* eof within a buffer signifying end of input */ terminate lexical analysis
end
```

5.3. SPECIFICATION & RECOGNITION OF TOKEN

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string Compiler is 8.

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)

Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifies	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

Table1.1: Lex Symbol Description

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

int intvalue;

While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows rule priority where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

Regular Expressions in Compiler design

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as regular grammar. The language defined by regular grammar is known as regular language.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations

The various operations on languages are:

- Union of two languages L and M is written as
  L U M = {s | s is in L or s is in M}
- Concatenation of two languages L and M is written as
  LM = {st | s is in L and t is in M}
- The Kleene Closure of a language L is written as
  L* = Zero or more occurrence of language L.

Notations

If r and s are regular expressions denoting the languages L(r) and L(s), then

- **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
- **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
- **Kleene closure** : (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r) Precedence and Associativity
- *, concatenation (.), and | (pipe sign) are left associative
- * has the highest precedence
- Concatenation (.) has the second highest precedence.
- | (pipe sign) has the lowest precedence of all.

Representing valid tokens of a language in regular expression

If x is a regular expression, then:

- x* means zero or more occurrence of x.
  i.e., it can generate { e, x, xx, xxx, xxxx, … }
- x+ means one or more occurrence of x.
  i.e., it can generate { x, xx, xxx, xxxx … } or x.x*
- x? means at most one occurrence of x
  i.e., it can generate either {x} or {e}.

[a-z] is all lower-case alphabets of English language.
[A-Z] is all upper-case alphabets of English language.
[0-9] is all natural digits used in mathematics.
Representing occurrence of symbols using regular expressions
Letter = [a – z] or [A – Z]
Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]
Sign = [+ | -]
Representing language tokens using regular expressions
Decimal = (sign)$^?$(digit)$^+$
Identifier = (letter)(letter | digit)*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

Compiler-construction tools

Originally, compilers were written “from scratch”, but now the situation is quite different. A number of tools are available to ease the burden.
We will study tools that generate scanners and parsers. This will involve us in some theory, regular expressions for scanners and various grammars for parsers. These techniques are fairly successful. One drawback can be that they do not execute as fast as “hand-crafted” scanners and parsers.
We will also see tools for syntax-directed translation and automatic code generation. The automation in these cases is not as complete.
Finally, there is the large area of optimization. This is not automated; however, a basic component of optimization is “data-flow analysis” (how values are transmitted between parts of a program) and there are tools to help with this task.

5.4.LEX

Lex is a tool in lexical analysis phase to recognize tokens using regular expression.
Lex tool itself is a lex compiler.

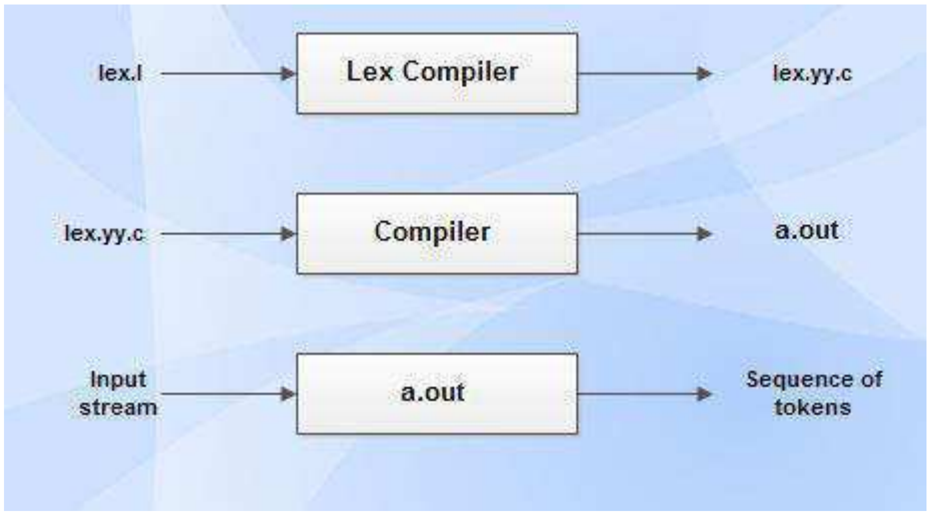

Figure 1.10 LEX Diagram

• ***lex.l*** is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms *lex.l* to a C program known as *lex.yy.c.*
• ***lex.yy.c*** is compiled by the C compiler to a file called *a.out.*
• The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.
• ***yylval*** is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.
• The attribute value can be numeric code, pointer to symbol table or nothing.
• Another tool for lexical analyzer generation is Flex.

Structure of Lex Programs
Lex program will be in following form
declarations
%%
translation rules
%%

auxiliary functions
Declarations This section includes declaration of variables, constants and regular definitions.
Translation rules It contains regular expressions and code segments.
Form : Pattern {Action}
Pattern is a regular expression or regular definition.
Action refers to segments of code.
Auxiliary functions This section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer.
Lexical analyzer produced by Lex starts its process by reading one character at a time until a valid match for a pattern is found.
Once a match is found, the associated action takes place to produce token.
The token is then given to parser for further processing.

Conflict Resolution in Lex

Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following:

- Always prefer a longer prefix than a shorter prefix.
- If two or more patterns are matched for the longest prefix, then the first pattern listed in lex program is preferred.

Lookahead Operator

- Lookahead operator is the additional operator that is read by lex in order to distinguish additional pattern for a token.
- Lexical analyzer is used to read one character ahead of valid lexeme and then retracts to produce token.
- At times, it is needed to have certain characters at the end of input to match with a pattern. In such cases, slash (/) is used to indicate end of part of pattern that matches the lexeme.

(eg.) In some languages keywords are not reserved. So the statements
IF (I, J) = 5 and IF(condition) THEN
results in conflict whether to produce IF as an array name or a keyword. To resolve this the lex rule for keyword IF can be written as,
IF/\ (.* \) {letter}

Design of Lexical Analyzer

- Lexical analyzer can either be generated by NFA or by DFA.
- DFA is preferable in the implementation of lex.

Structure of Generated Analyzer

Architecture of lexical analyzer generated by lex is given in Figure:

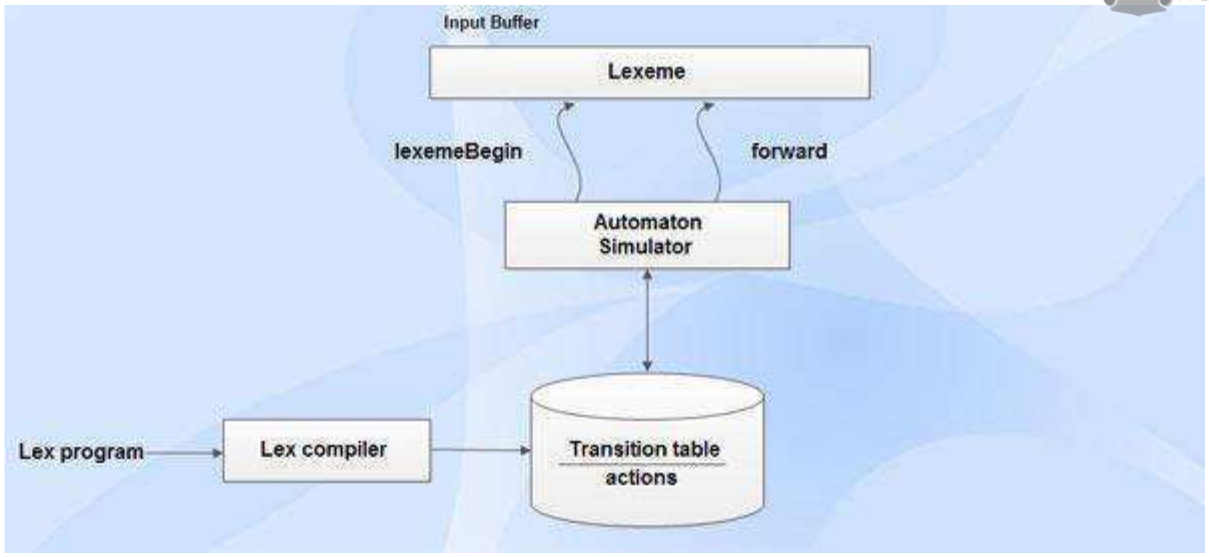

Figure 1.11 Architecture of lexical analyzer

Lexical analyzer program includes:

- Program to simulate automata.
- Components created from lex program by lex itself which are listed as follows:
  - A transition table for automaton.
  - Functions that are passed directly through lex to the output.
  - Actions from input program (fragments of code) which are invoked by automaton simulator when needed.

===