LECTURE NOTES

ON

COMPILER DESIGN

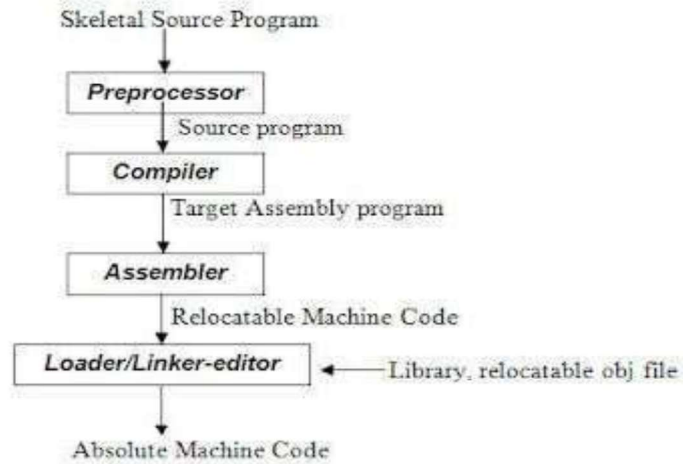# UNIT -I

1. OVERVIEW OF LANGUAGE PROCESSING SYSTEM


Fig 1.1 Language –processing System

1.1 Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

1. **Macro processing:** A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. **File inclusion:** A preprocessor may include header files into the program text.
3. **Rational preprocessor:** these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. **Language Extensions:** These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

1.2 Compiler

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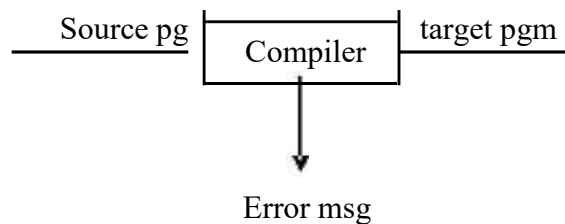
Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

1.3 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).

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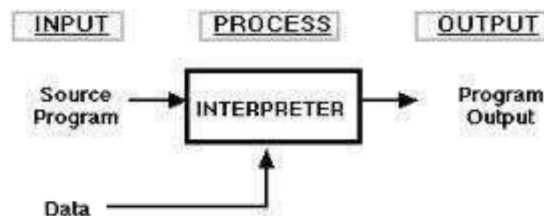
Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.
1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Direct Execution

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 theprogramis slower. • Memory consumption is more.

1.5 Loader and Link-editor:

Once the assembler procedures an object program, that program must be p laced into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome this problems of wasted translation time and memory. System programmers developed another component called loader.

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could "relocate" directly behind the user's program. The task of adjusting programs o they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

1.6 TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected and reported to the programmers. Important role of translator are:

1 Translating the hll program input into an equivalent ml program.
2 Providing diagnostic messages wherever the programmer violates specification of the hll.

1.6.1 TYPE OF TRANSLATORS:-
- Interpreter
- Compiler
- preprocessor

1.6.2 LIST OF COMPILERS

1. Ada compilers
2. ALGOL compilers
3. BASIC compilers
4. C# compilers
5. C compilers
6. C++ compilers
7. COBOL compilers
8. Java compilers

2. 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. The phases of a compiler are shown in below
There are two phases of compilation.
  a. Analysis (Machine Independent/Language Dependent)
  b. Synthesis (Machine Dependent/Language independent) Compilation process is partitioned into no-of-sub processes called **'phases'**.
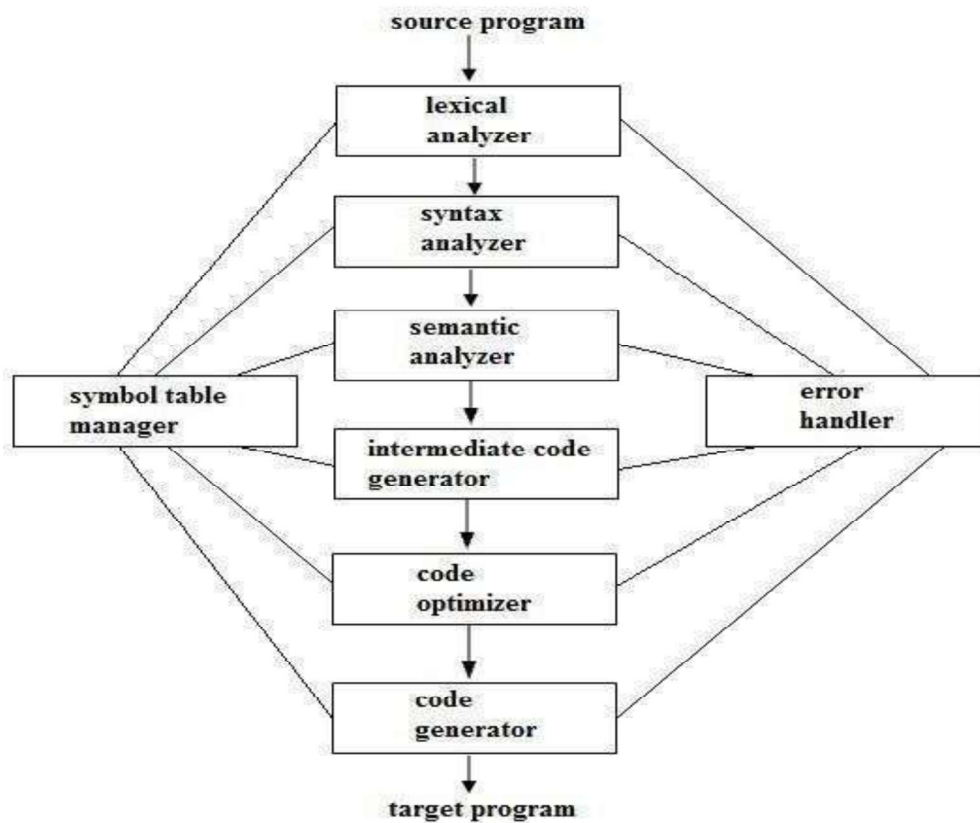

Fig 1.5 Phases of a compiler

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

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.

Table Management (or) Book-keeping:-

This is the portion to **keep the names** used by the program and records essential information about each. The data structure used to record this information called a **'Symbol Table'.**

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.

The parser has two functions. It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language. It also imposes on tokens a tree-like structure that is used by the sub-sequent phases of the compiler.

Example, if a program contains the expression **A+/B** after lexical analysis this expression might appear to the syntax analyzer as the token sequence **id+/id.** On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formulations rule of an expression.

Syntax analysis is to make explicit the hierarchical structure of the incoming token stream by **identifying which parts of the token stream should be grouped**.

Example, (A/B*C has two possible interpretations.)
1- divide A by B and then multiply by C or
2- multiply B by C and then use the result to divide A.
Each of these two interpretations can be represented in terms of a parse tree.

Intermediate Code Generation:-

The intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are

possible. One common style uses instruction with one operator and a small number of operands. The output of the syntax analyzer is some representation of a parse tree. The intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization:-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Its output is another intermediate code program that does the same job as the original, but in a way that saves time and / or spaces.

/* 1, Local Optimization:-

There are local transformations that can be applied to a program to make an improvement. For example,

If **A** > **B** goto **L2**

Goto **L3 L2 :**

This can be replaced by a single statement If **A < B** goto   **L3**

Another important local optimization is the   elimination of common sub-expressions

A := B + C + D

E := B + C + F

Might be evaluated as

T1 := B +   C

A   := T1 +   D

E   := T1 +   F

Take this advantage of the common sub-expressions **B + C.**

Loop Optimization:-

Another important source of optimization concerns about **increasing the speed of loops**. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point, in the program just before the loop is entered.*/

Code generator :-

C produces the object code by deciding on the memory locations for data, selecting code to access each data and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed quickly. A good code generator would attempt to utilize registers as efficiently as possible.
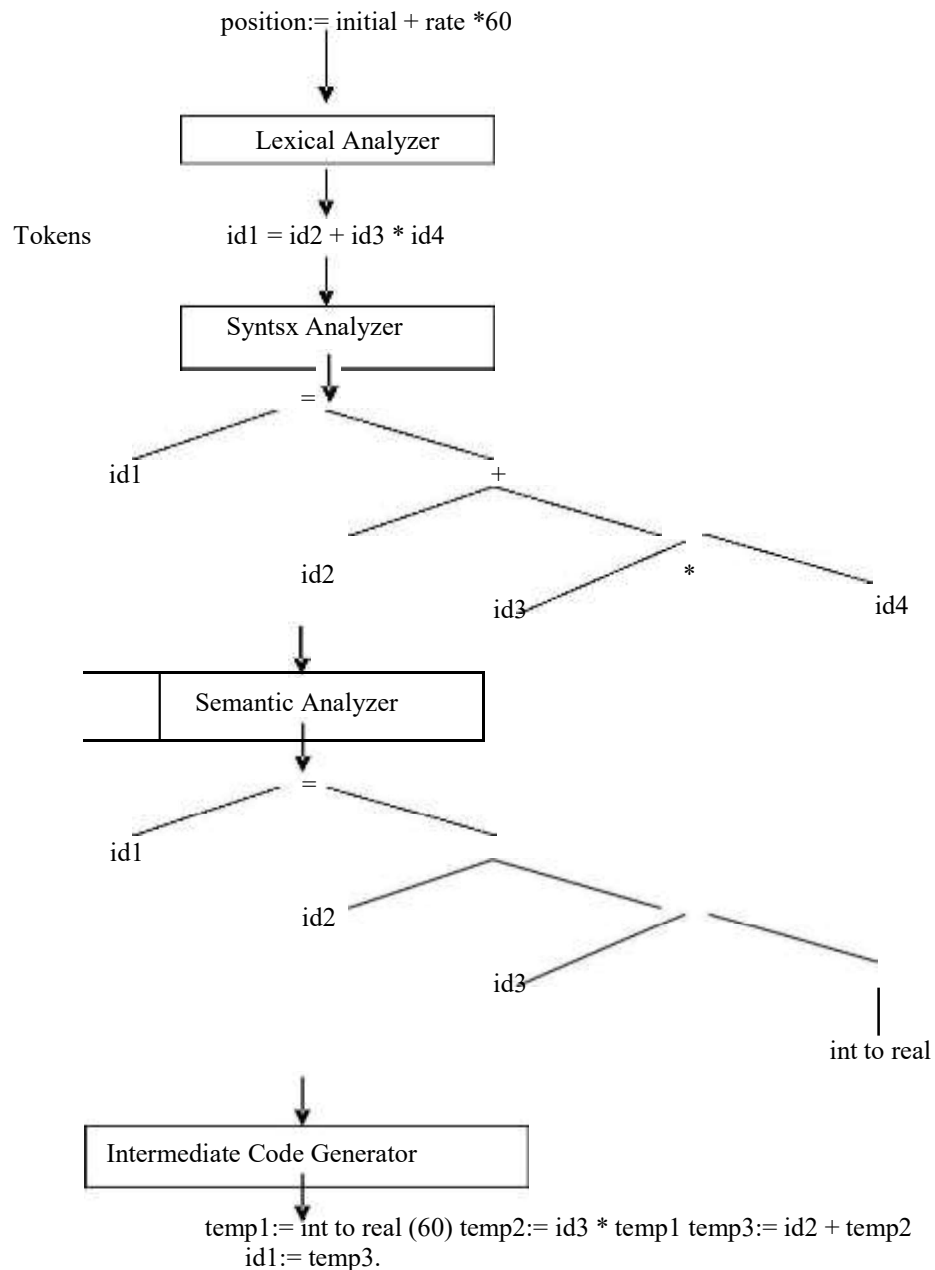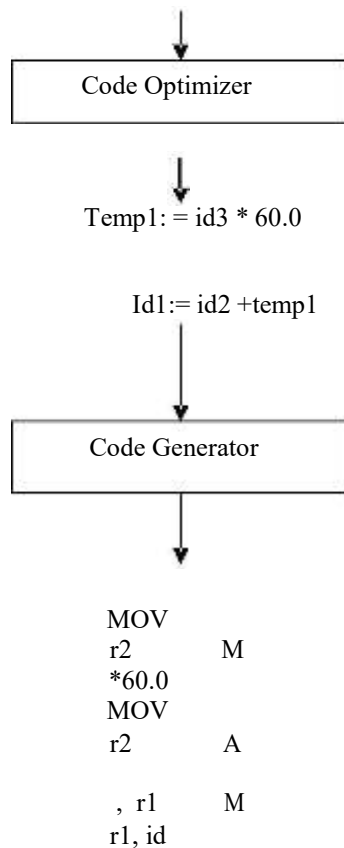
Error Handing :-

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. Errors may occur in all or the phases of a compiler.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Example:

position:= initial + rate *60

↓

```
┌─────────────────────────┐
│     Lexical Analyzer     │
└─────────────────────────┘
```

↓

Tokens   id1 = id2 + id3 * id4

↓

```
┌─────────────────────────┐
│     Syntsx Analyzer      │
└─────────────────────────┘
```

↓

```
        =
      /   \
    id1    +
         /   \
       id2    *
            /   \
          id3    id4
```

↓

```
┌─────────────────────────┐
│    Semantic Analyzer     │
└─────────────────────────┘
```

↓

```
        =
      /   \
    id1    +
         /   \
       id2    *
            /   \
          id3   int to real
```

↓

```
┌──────────────────────────────────┐
│    Intermediate Code Generator    │
└──────────────────────────────────┘
```

↓

temp1:= int to real (60) temp2:= id3 * temp1 temp3:= id2 + temp2
id1:= temp3.

```
                    ┌─────────────────────┐
                    │   Code Optimizer    │
                    └─────────────────────┘
                              │
                              ▼

                    Temp1: = id3 * 60.0

                    Id1:= id2 +temp1

                              │
                              ▼
                    ┌─────────────────────┐
                    │   Code Generator    │
                    └─────────────────────┘
                              │
                              ▼
```

MOV
r2   M
*60.0
MOV
r2   A

, r1   M
r1, id

2.1 LEXICAL ANALYZER:

The LA is the first phase of a compiler. Lexical analysis is called as linear analysis or scanning. In this phase the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.
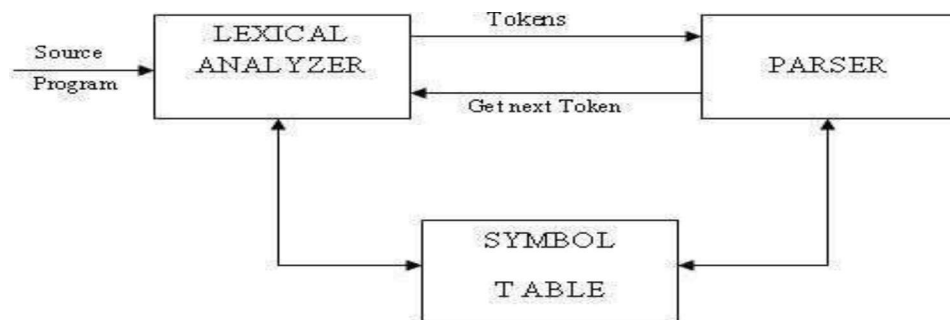
Upon receiving a 'get next token' command form the parser, the lexical analyzer

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.

2.1.1 Lexical Analysis Vs Parsing:

Lexical analysis	Parsing
A Scanner simply turns an input String (say a file) into a list of tokens. These tokens repr things like identifiers, parentheses, operators etc. The lexical analyzer (the "lexer") p individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar	A parser converts this list of tokens into a like object to represent how the toke together to form a cohesive (sometimes referred to as a sentence). A parser does not give the nodes meaning beyond structural cohesion. The thing to do is extract meaning from this structure (sometimes called conte analysis).

2.1.2 Token, Lexeme, Pattern:

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.

Example:
            Description of token

Token	lexeme	pattern
const	const	const
if	if	If

relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w "and "except"
literal	"core"	pattern

 A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

2.1.3 Lexical Errors:

Lexical errors are the errors thrown by the lexer when unable to continue. Which means that there's no way to recognise 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:
- Delete one character from the remaining input.
- Insert a missing character in to the remaining input.
- Replace a character by another character.
- Transpose two adjacent characters.

3. Difference Between Compiler And Interpreter:

1. A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.
■ Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.
3. List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.
■ An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.
■ The compiler produce object code whereas interpreter does not produce object code. In
■ the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. Hence interpreter is less efficient than compiler.

■ **Examples of interpreter:** A UPS Debugger is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files.

7. **Example of compiler:** Borland c compiler or Turbo C compiler compiles the programs written in C or C++.

4. REGULAR EXPRESSIONS:

4.1 : SPECIFICATION OF TOKENS

There are 3 specifications of tokens:
1) Strings
2) Language
3) Regular expression

Strings and Languages

An **alphabet** or character class is a finite set of symbols.

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet. A **language** is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s.

For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

Operations on strings
The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of strings.
    For example, ban is a prefix of banana.

2. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s.
    For example, nana is a suffix of banana.

3. A **substring** of s is obtained by deleting any prefix and any suffix from s.
    For example, nan is a substring of banana.

4. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.

5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s

For example, baan is a subsequence of banana.

Operations on languages:
The following are the operations that can be applied to languages:
1. Union
2. Concatenation
3. Kleene closure
4.Positive closure

The following example shows the operations on strings:

Let L={0,1} and S={a,b,c}
    Union   : L U S={0,1,a,b,c}
    Concatenation   : L.S={0a,1a,0b,1b,0c,1c}
    Kleene closure   : L^{*}={ ε,0,1,00….}

    Positive closure : L^{+}={0,1,00….}

4.2 Regular Expressions:

Each regular expression r denotes a language L(r).

Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.

2. If'a'is a symbol in Σ, then 'a'is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with 'a'in its one position.

3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,

> o (r)|(s) is a regular expression denoting the language L(r) U L(s). o (r)(s) is a regular expression denoting the language L(r)L(s).
> o   (r)* is a regular expression denoting (L(r))*.
> o   (r) is a regular expression denoting L(r).

4. The unary operator * has highest precedence and is left associative.

5. Concatenation has second highest precedence and is left associative. has lowest precedence and is left associative.

4.2.1 REGULAR DEFINITIONS:

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?)) Pascal identifier

Letter - A | B | ……| Z | a | b |……| z| Digits - 0 | 1 | 2 | …. | 9

Id - letter (letter / digit)*

Shorthand's

Certain constructs occur so frequently in regular   expressions that it is convenient to introduce notational shorthands for them.

1. One or more instances (+):

- o The unary postfix operator + means " one or more instances of" .

- o If r is a regular expression that denotes the language L(r), then $(r)^+$ is a regular expression that denotes the language $(L(r))^+$

- o Thus the regular expression a^+ denotes the set of all strings of one or more a's.

- o The operator $^+$ has the same precedence and associativity as the operator *.

2. Zero or one instance (?):

- The unary postfix operator ? means "zero or one instance of".

- The notation r? is a shorthand for r | ε.

- If 'r' is a regular expression, then (r)? is a regular expression that denotes the language L(r) U { ε }.

3. Character Classes:

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
- We can describe identifiers as being strings generated by the regular expression,
   [A–Za–z][A–Za–z0–9]*

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

4.2.2 RECOGNITION OF TOKENS:

Consider the following grammar fragment: stmt → if expr then stmt
       |if expr then stmt else stmt |ε

expr → term relop term |term term → id |num

where the terminals if , then, else, relop, id and num generate sets of strings given by the following regular definitions:

If   → if
then   → then
else   → else
relop → <|<=|=|<>|>|>=
id   → letter(letter|digit)*

num   → digit$^+$ (.digit$^+$)?(E(+|-)?digit$^+$)?

For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers.

| Lexeme	Token Name	Attribute Value
Any ws		_
if	if	_
then	then	_
else		_
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
<>	relop	NE

4.3 TRANSITION DIAGRAM:

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .Edges are

directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols.If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.

2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.

3. One state is designed the state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.
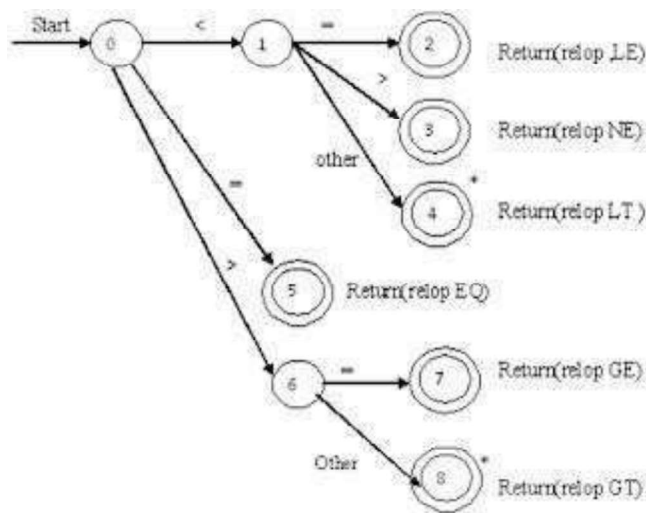
As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.
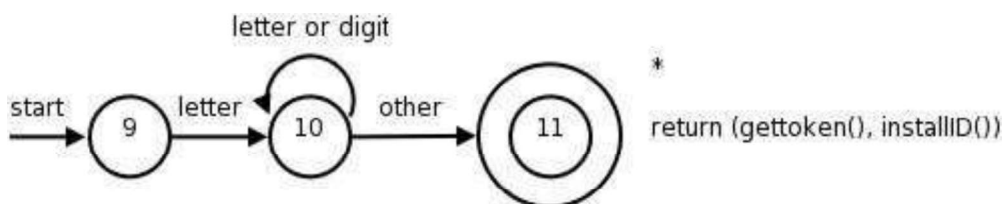
The above TD for an identifier, defined to be a letter followed by any no of letters or digits.A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

4.4 Automata:

Automation is defined as a system where information is transmitted and used for performing some functions without direct participation of man.

1. An automation in which the output depends only on the input is **called automation without memory.**
2. An automation in which the output depends on the input and state also is **called as automation with memory**.
3. An automation in which the output depends only on the state of the machine is **called a Moore machine**.
4. An automation in which the output depends on the state and input at any instant of time is **called a mealy machine**.

4.4.1 DESCRIPTION OF AUTOMATA

1. An automata has a mechanism to read input from input tape,
2. Any language is recognized by some automation, Hence these automation are basically language 'acceptors' or 'language recognizers'.

Types of Finite Automata

Deterministic Automata
Non-Deterministic Automata.

Deterministic Automata:

A deterministic finite automata has at most one transition from each state on any input. A DFA is a special case of a NFA in which:-

1. it has no transitions on input ϵ ,
2. Each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation M = (Q, $\sum$, δ, qo, F), where Q is a finite 'set of states', which is non empty.

$\sum$ is 'input alphabets', indicates input set.

qo is an 'initial state' and qo is in Q ie, qo, $\sum$, Q F is a set of 'Final states',

δ is a 'transmission function' or mapping function, using this function the next state can be determined.

The regular expression is converted into minimized DFA by the following procedure:

Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Minimized DFA

The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.
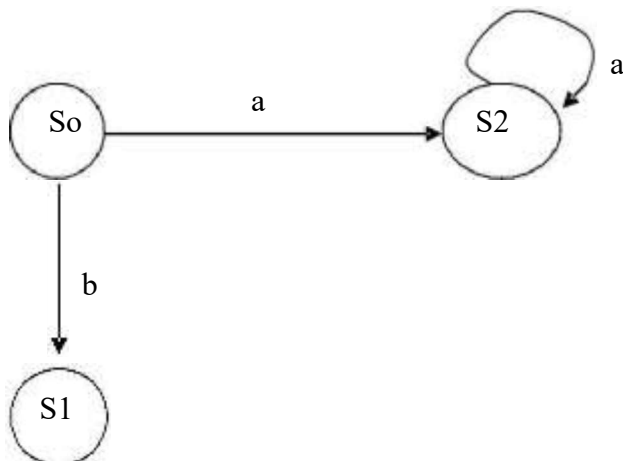
From state S0 for input 'a' there is only one path going to S2. similarly from so there is only one path for input going to S1.

Nondeterministic Automata:

A NFA ia A mathematical model consists of

- A set of states S.
- A set of input symbols $\sum$.
- A transition is a move from one state to another.
- A state so that is distinguished as the start (or initial) state
- A set of states F distinguished as accepting (or final) state.
- A number of transition to a single symbol.

A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, in which the nodes are the states and the labeled edges represent the transition function.

This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol $\in$ as well as input symbols.

The transition graph for an NFA that recognizes the language (a|b)*abb is shown
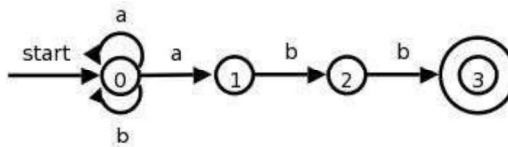
5. Bootstrapping:

When a computer is first turned on or restarted, a special type of absolute loader, called as bootstrap loader is executed. This bootstrap loads the first program to be run by the computer usually an operating system. The bootstrap itself begins at address O in the memory of the machine. It loads the operating system (or some other program) starting at address 80. After all of the object code from device has been loaded, the bootstrap program jumps to address 80, which begins the execution of the program that was loaded.

Such loaders can be used to run stand-alone programs independent of the operating system or the system loader. They can also be used to load the operating system or the loader itself into memory.

Loaders are of two types:

- Linking loader.
- Linkage editor.

Linkage loaders, perform all linking and relocation at load time.

Linkage editors, perform linking prior to load time and dynamic linking, in which the linking function is performed at execution time.

A linkage editor performs linking and some relocation; however, the linkaged program is written to a file or library instead of being immediately loaded into memory. This approach reduces the overhead when the program is executed. All that is required at load time is a very simple form of relocation.
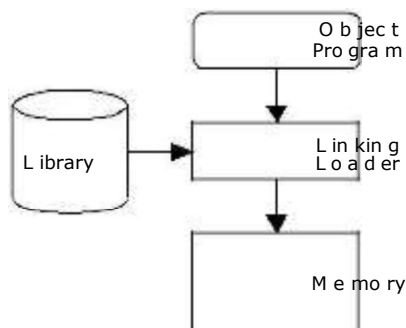
Linking Loader
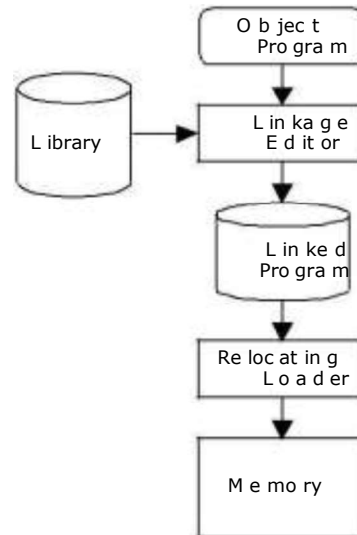
Linkage Editor

6. Pass And Phases Of Translation:

Phases: (Phases are collected into a front end and back end)

Frontend:

      The front end consists of those phases, or parts of phase, that depends primarily on the source language and is largely independent of the target machine. These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code.

      A certain amount of code optimization can be done by front end as well. the front end also includes the error handling tha goes along with each of these phases.
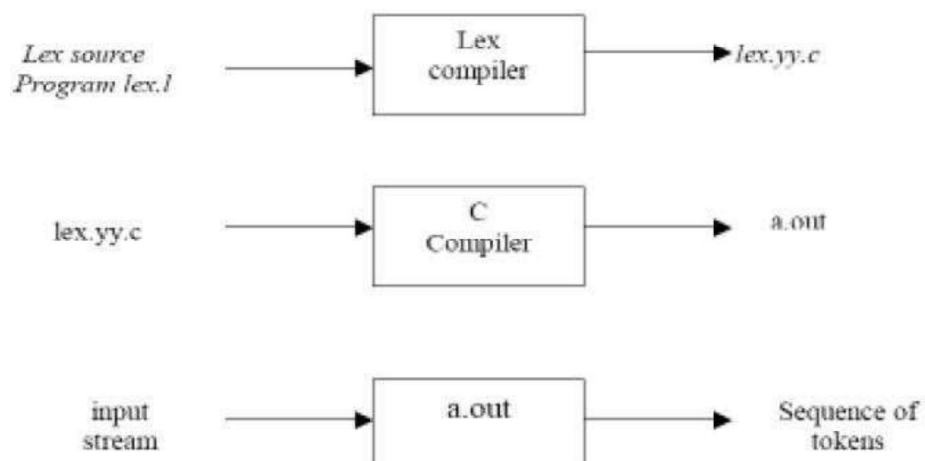
Back end:

      The back end includes those portions of the compiler that depend on the target machine and generally, these portions do not depend on the source language .

7. Lexical Analyzer Generator:

7.1 Creating a lexical analyzer with Lex:

- First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.

- Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

```
Lex source                          Lex
Program lex.l  ────────►        compiler        ────►  lex.yy.c


lex.yy.c       ────────►           C            ────►   a.out
                               Compiler


input          ────────►         a.out          ────►  Sequence of
stream                                                    tokens
```

Lex Specification

A Lex program consists of three parts:

{ definitions }
%%
{ rules }
%%
{ user subroutines }

- o **Definitions** include declarations of variables, constants, and regular definitions
- o **Rules** are statements of the form p1 {action1}p2 {action2} ... pn {action}
- o where pi is regular expression and actioni describes what action the lexical analyzer should take when pattern pi matches a lexeme. Actions are written in C code.
- o **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

8. INPUT BUFFERING

The LA scans the characters of the source program one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffering techniques:
1. Buffer pairs
2. Sentinels

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. Figure 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.

Token beginnings look ahead pointer, 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: DECALRE (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.