UNIT -2

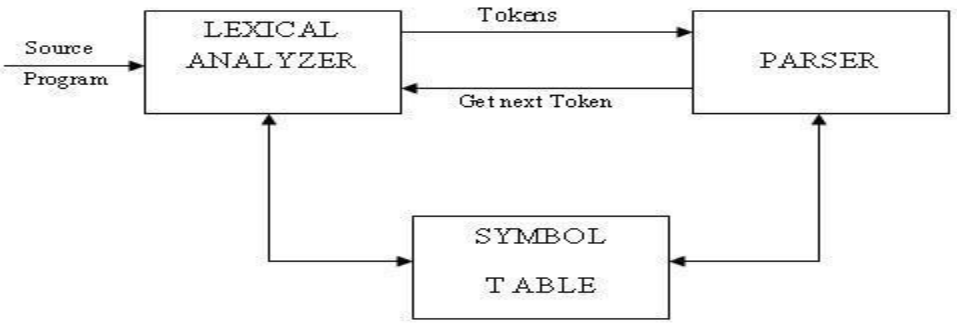
LEXICAL ANALYSIS

OVER VIEW OF 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.
- 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.

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.

LEXICAL ANALYSIS VS PARSING:

Lexical analysis	Parsing
A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc. The lexical analyzer (the "lexer") parses 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 Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence). A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual analysis).

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

LEXICAL ERRORS:

Lexical errors are the errors thrown by your 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** recognised 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:

i. Delete one character from the remaining input.
ii. Insert a missing character in to the remaining input.
iii. Replace a character by another character.
iv. Transpose two adjacent characters.

DIFFERENCE BETWEEN COMPILER AND INTERPRETER

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

REGULAR EXPRESSIONS

Regular expression is a formula that describes a possible set of string.

Component of regular expression..

X	**the character x**
.	**any character, usually accept a new line**
[x y z]	**any of the characters x, y, z, …..**
R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences…..**
R+	**one or more occurrences ……**
R1R2	**an R1 followed by an R2**
R2R1	**either an R1 or an R2.**

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as an language, we can use the regular-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters or digits. In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the regular expression over alphabet .

- is a regular expression denoting { € }, that is, the language containing only the empty string.
- For each ‘a’ in ∑, is a regular expression denoting { a }, the language with only one string consisting of the single symbol ‘a’ .
- If R and S are regular expressions, then

(R) | (S) means LrULs
R.S means Lr.Ls
R* denotes Lr*

2.8 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.

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examins the input string and finds a prefix that is a lexeme matching one of the patterns.

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting structure of lexemes. The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

digit -->[0,9]
digits -->digit+
number -->digit(.digit)?(e.[+-]?digits)?
letter -->[A-Z,a-z]
id -->letter(letter/digit)*
if --> if
then -->then
else -->else
relop --></>/<=/>=/==/< >

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

$$ws \rightarrow (\text{blank/tab/newline})^{+}$$

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

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

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.

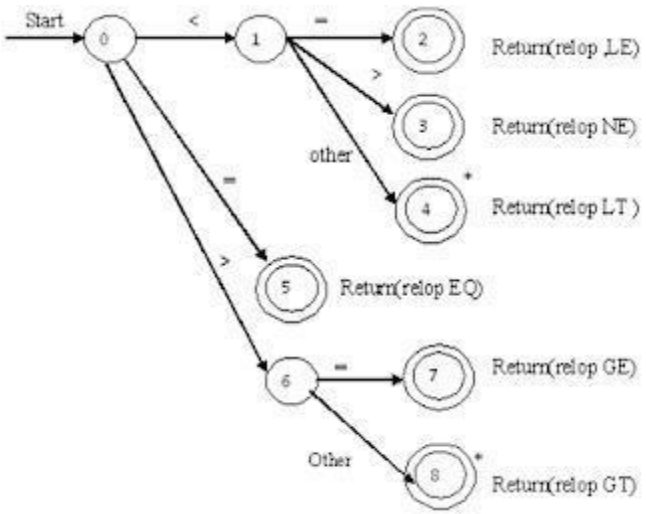
If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

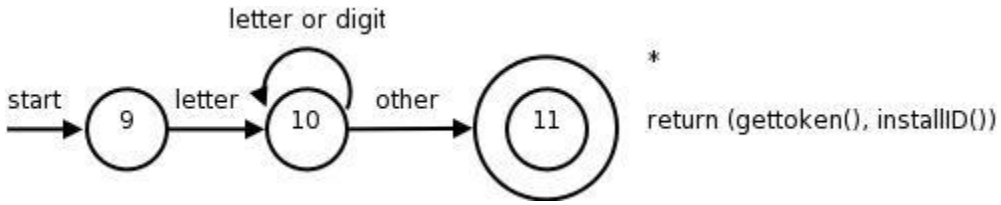
1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.

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.

If	=	if
Then	=	then
Else	=	else
Relop	=	< | <= | = | > | >=
Id	=	letter (letter | digit) *|
Num	=	digit |

AUTOMATA

An 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 an 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**.

3, an automation in which the output depends on the state and input at any instant of time is **called a mealy machine**.

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:-

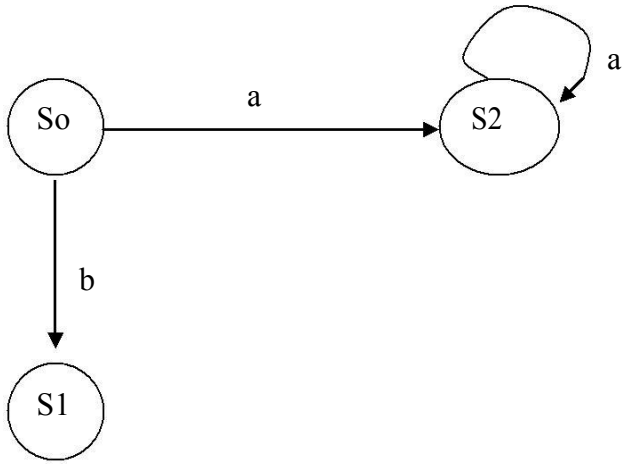
1, it has no transitions on input € ,
2, each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation M = (Q, ∑, δ, qo, F), where Q is a finite 'set of states', which is non empty.
∑ is 'input alphabets', indicates input set.
qo is an 'initial state' and qo is in Q ie,
qo, ∑, 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 → NFA → DFA → 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 S0 there is only one path for input going to S1.

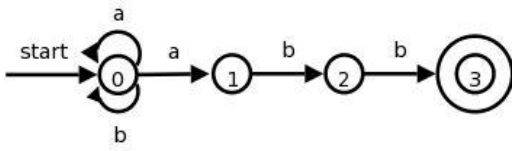
NONDETERMINISTIC AUTOMATA

- A NFA is a mathematical model that consists of
  - A set of states S.
  - A set of input symbols ∑.
  - A transition for move from one state to an other.
  - 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 € as well as by input symbols.

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



DEFINITION OF CFG

It involves four quantities.

CFG contain terminals, N-T, start symbol and production.

Terminal are basic symbols form which string are formed.

N-terminals are synthetic variables that denote sets of strings

In a Grammar, one N-T are distinguished as the start symbol, and the set of string it denotes is the language defined by the grammar.

The production of the grammar specify the manor in which the terminal and N-T can be combined to form strings.

Each production consists of a N-T, followed by an arrow, followed by a string of one terminal and terminals.

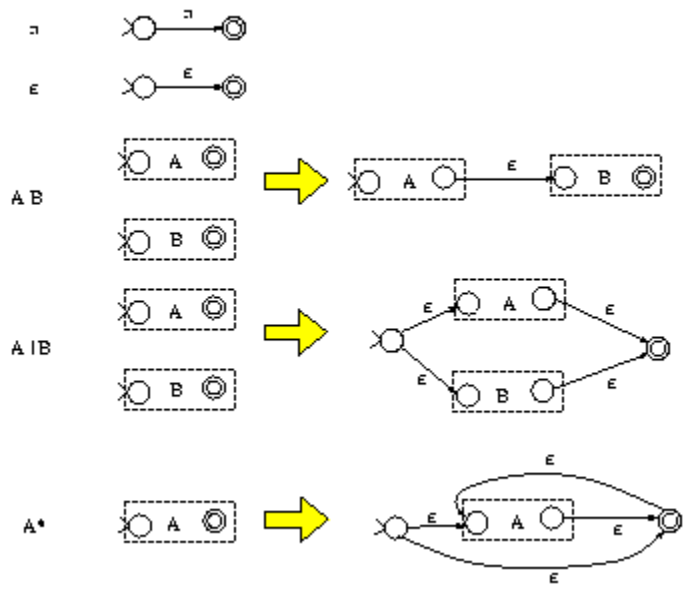
Converting a Regular Expression into a Deterministic Finite Automaton

The task of a scanner generator, such as flex, is to generate the transition tables or to synthesize the scanner program given a scanner specification (in the form of a set of REs). So it needs to convert a RE into a DFA. This is accomplished in two steps: first it converts a RE into a non-deterministic finite automaton (NFA) and then it converts the NFA into a DFA.

A NFA is similar to a DFA but it also permits multiple transitions over the same character and transitions over ε. The first type indicates that, when reading the common character associated with these transitions, we have more than one choice; the NFA succeeds if at least one of these choices succeeds. The εtransition doesn't consume any input characters, so you may jump to another state for free.

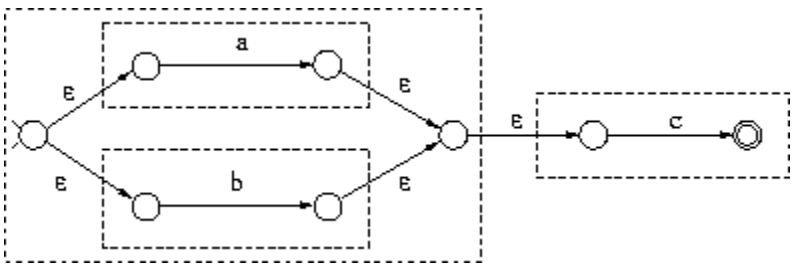
Clearly DFAs are a subset of NFAs. But it turns out that DFAs and NFAs have the same expressive power. The problem is that when converting a NFA to a DFA we may get an exponential blowup in the number of states.

We will first learn how to convert a RE into a NFA. This is the easy part. There are only 5 rules, one for each type of RE:



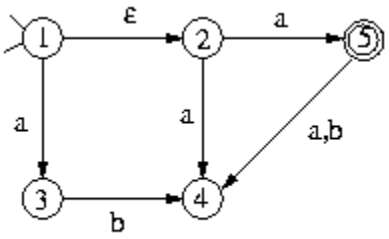
The algorithm constructs NFAs with only one final state. For example, the third rule indicates that, to construct the NFA for the RE *AB*, we construct the NFAs for *A* and *B* which are represented as two boxes with one start and one final state for each box. Then the NFA for *AB* is constructed by connecting the final state of *A* to the start state of *B* using an empty transition.

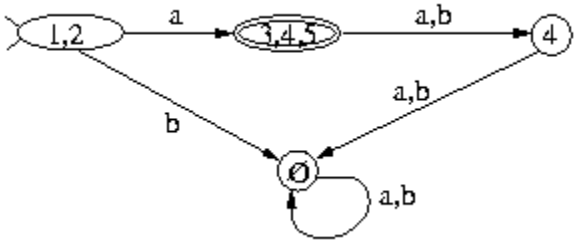
For example, the RE (*a*|b)*c* is mapped to the following NFA:



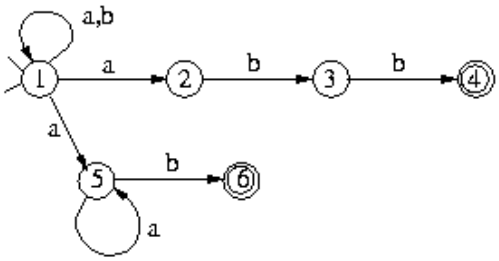
The next step is to convert a NFA to a DFA (called *subset construction*). Suppose that you assign a number to each NFA state. The DFA states generated by subset construction have sets of numbers, instead of just one number. For example, a DFA state may have been assigned the set {5,6,8}. This indicates that arriving to the state labeled {5,6,8} in the DFA is the same as arriving to the state 5, the state 6, or the state 8 in the NFA when parsing the same input. (Recall that a particular input sequence when parsed by a DFA, leads to a unique state, while when parsed by a NFA it may lead to multiple states.)

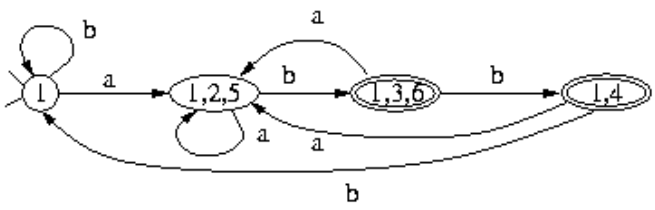
First we need to handle transitions that lead to other states for free (without consuming any input). These are the ε transitions. We define the *closure* of a NFA node as the set of all the nodes reachable by this node using zero, one, or more ε transitions. For example, The closure of node 1 in the left figure below





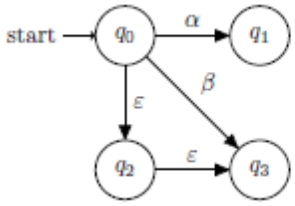
is the set {1,2}. The start state of the constructed DFA is labeled by the closure of the NFA start state. For every DFA state labeled by some set $\{s_1, \ldots, s_n\}$ and for every character c in the language alphabet, you find all the states reachable by s_1, s_2, ..., or s_n using c arrows and you union together the closures of these nodes. If this set is not the label of any other node in the DFA constructed so far, you create a new DFA node with this label. For example, node {1,2} in the DFA above has an arrow to a {3,4,5} for the character a since the NFA node 3 can be reached by 1 on a and nodes 4 and 5 can be reached by 2. The b arrow for node {1,2} goes to the error node which is associated with an empty set of NFA nodes. The following NFA recognizes $(a|b)^*(abb\,|\,a^+b)$, even though it wasn't constructed with the 5 RE-to-NFA rules. It has the following DFA:





Converting NFAs to DFAs

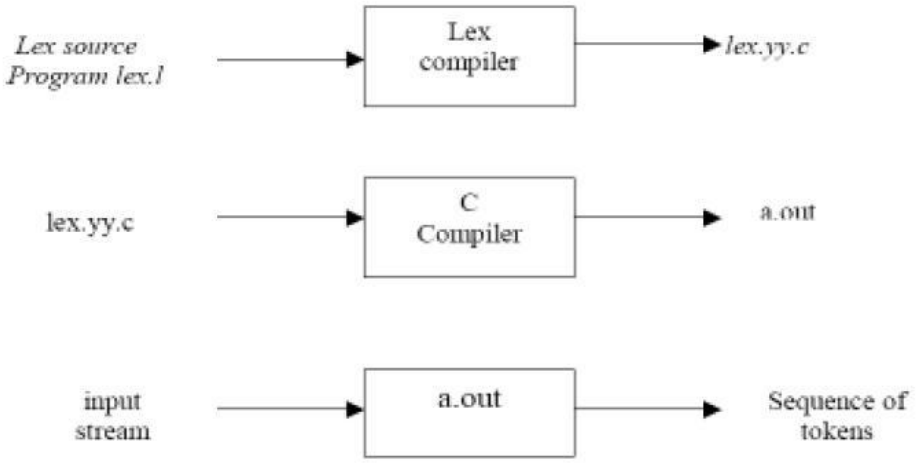
To convert an NFA to a DFA, we must and a way to remove all "-transitions and to ensure that there is one transition per symbol in each state. We do this by constructing a DFA in which each state corresponds to a set of some states from the NFA. In the DFA, transitions from a state S by some symbol go to the state S that consists of all the possible NFA-states that could be reached by from some NFA state q contained in the present DFA state S. The resulting DFA \simulates" the given NFA in the sense that a single DFA-transition represents many simultaneous NFA-transitions. The first concept we need is the "E-closure pronounced \epsilon closure". The " -closure of an NFA state q is the set containing q along with all states in the automaton that are reachable by any number of " E-transitions from q . In the following automaton, the " E-closures are given in the table to the right:



State	ε-closure
q_0	$\{q_0, q_2, q_3\}$
q_1	$\{q_1\}$
q_2	$\{q_2, q_3\}$
q_3	$\{q_3\}$

Likewise, we can done the "-closure of a set of states to be the states reachable by " -transitions from its members. In other words, this is the union of the " -closures of its elements. To convert our NFA to its DFA counterpart, we begin by taking the " –closure of the start state q of our NFA and constructing a new start state S. in our DFA corresponding to that " -closure. Next, for each symbol in our alphabet, we record the set of NFA states that we can reach from S 0on that symbol. For each such set, we make a DFA state corresponding to its "E-closure, taking care to do this only once for each set. In the case two sets are equal, we simply reuse the existing DFA state that we already constructed. This process is then repeated for each of the new DFA states (that is, set of NFA states) until we run out of DFA states to process. Finally, every DFA state whose corresponding set of NFA states contains an accepting state is itself marked as an accepting state.

Creating a lexical analyzer with Lex



2.17 Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. # *define PIE 3.14*), and regular definitions.
2. The *translation rules* of a Lex program are statements of the form :

p1	*{action 1}*
p2	*{action 2}*
p3	*{action 3}*
...	...
...	...

   where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever *auxiliary procedures* are needed by the *actions*.Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

Note: You can refer to a sample lex program given in page no. 109 of chapter 3 of the book: *Compilers: Principles, Techniques, and Tools* by Aho, Sethi & Ullman for more clarity.