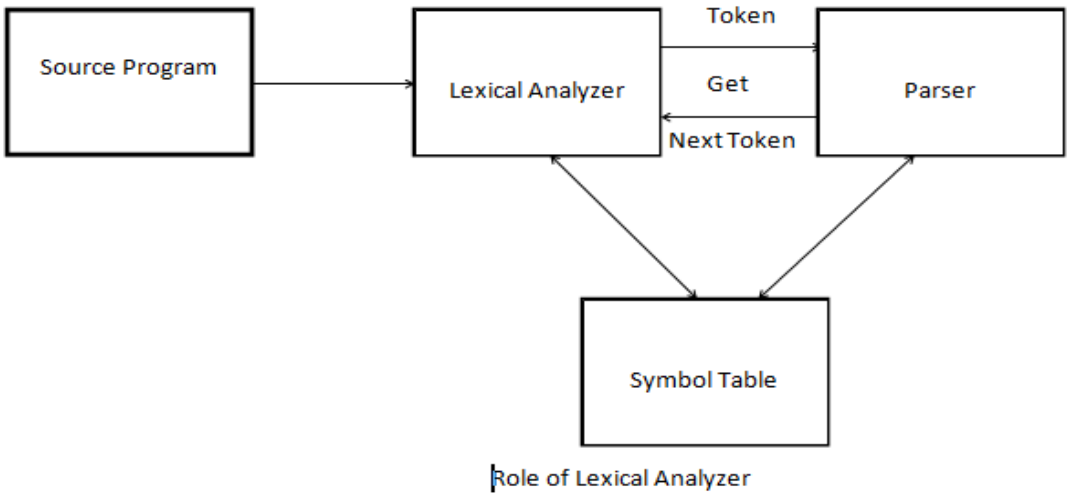
Unit 2: Lexical Analyzer

Lexical analysis is the first phase of a compiler. It takes the source code as input and generates the tokens by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer.



Role of Lexical Analyzer

Issues in Lexical analyzer: There are several issues in lexical analyzer for separating the analysis phase of compiling into lexical analysis and parsing.

1. Simpler design is the most important consideration for lexical analyzer. The separation of lexical analyzer from syntax analyzer often allows us to simplify one or more phases.
2. For improve compiler efficiency we have need a separate lexical analyzer which allows us to construct a specialized and potentially more efficient processer for the separation of tokens.
3. For enhance compiler portability input alphabet characteristics and other device-specific drawback can be restricted to the lexical analyzer.

Tokens:
Basic task of scanner is the group a character from a word is known as **lexemes**. Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules are known as **pattern**. A pattern explains what can be a token, and these patterns are defined by regular expressions. Words of source code can be separated by **delimiters**.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

```
int value = 100;
```

Contains the tokens:

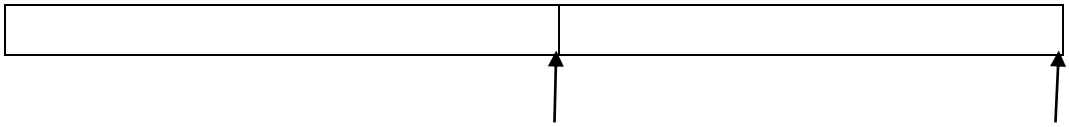
int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

Lexical errors: When scanner find out lexeme and match it with patterns it is not matched with any pattern then these word is treated as errors. Lexical analyzer performs some action on those lexical errors. This is called as error recovery action. These actions are given below:

- Deleting an extraneous character
- Inserting a missing character
- Replacing incorrect character by correct character
- Exchange two adjacent characters

Input Buffering:

The lexical analyzer scans the source program one character at a time and identifies a token. Even after reading the last character of the token, scanner needs to check many characters other than the token to identify the token. So scanner reads the input from the input buffer. Following figure shows the input buffer, which is divided into two part. There are two pointers one mark the beginning of the token and another pointer is a look ahead pointer which scans the characters from the beginning pointer until the token is identified.



As input buffer is of limited size, there is a constraint on how much look ahead can be used before the next token is identified. Solution is to select appropriate buffer size.

Specifications of Tokens:

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 alphanumeric, {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 GFCCT is 5 and is denoted by |GFCCT| = 5. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

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 Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

Language:

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

Regular Expressions:

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belongs to the language. 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 by using regular expressions. 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 {ε, 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:

Number = $(Digit)^+$ or (Digit).(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.

Finite Automata:

Finite automata are a state machine that takes a string of symbols as input and changes its state accordingly. A finite automaton is a recognized for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each alphabet. If the input string is successfully processed and the automata reach its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language.

The mathematical model of finite automata consists of: $M= (Q, \Sigma, \delta, q_0, F)$

Finite set of states (Q)

Finite set of input symbols (Σ)

One Start state (q_0)

Set of final states (F)

Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$

Finite Automata Construction:

Let L(r) be a regular language recognized by some finite automata (FA).

States : States of FA are represented by circles. State names are written inside circles.

Start state : The state from where the automata starts, is known as the start state. Start state has an arrow pointed towards it.

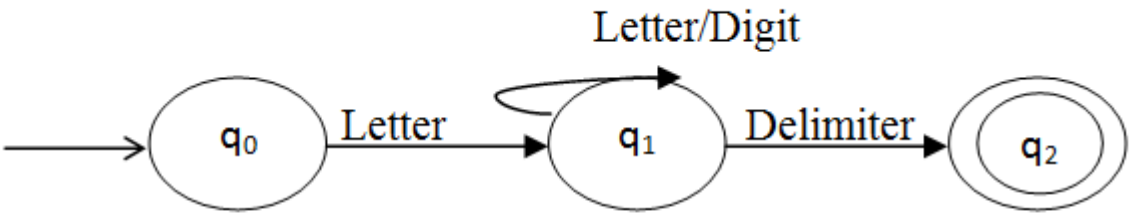
Intermediate states : All intermediate states have at least two arrows; one pointing to and another pointing out from them.

Final state : If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. **odd = even+1**.

Transition : The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state. If automata stay on the same state, an arrow pointing from a state to itself is drawn.

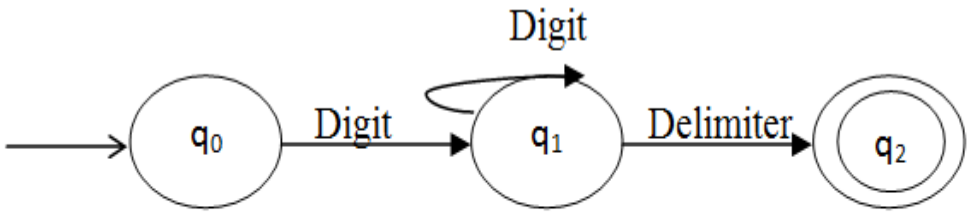
Design a lexical analyzer generator: Design of lexical analyzer can be explained with the help of finite automata.

E.g.; 1.Finite automata which recognizes an identifier, is shown in following figure where identifier starts with letter followed by letter or digit. The delimiter identifies the token as identifier.



Reorganization of Identifier

2. Finite automata which recognizes a digit, is shown in following figure where digit starts with digit followed by any number of digits. The delimiter identifies the token as identifier.



Reorganization of Digit

We can write code for each state in the transition diagram of reorganization of identifier. In this we use following functions as standard functions.

1) getchar(): It returns the input charter pointed by look ahead pointer and advances look ahead pointer to next character.
2) retract(): It is used to retract look ahead pointer one character.
3) install(): If token is not in the symbol table then we can add it using install().
4) error(): Error message can be display.

We can write code for states as follows:

State q_0: ch:= getchar ();

if letter(ch) then goto state q_1

else

error();

State q_1: ch:= getchar ();

if letter(ch) or digit(ch) then goto state q_1.

else if delimiter(ch) then goto state q_2.

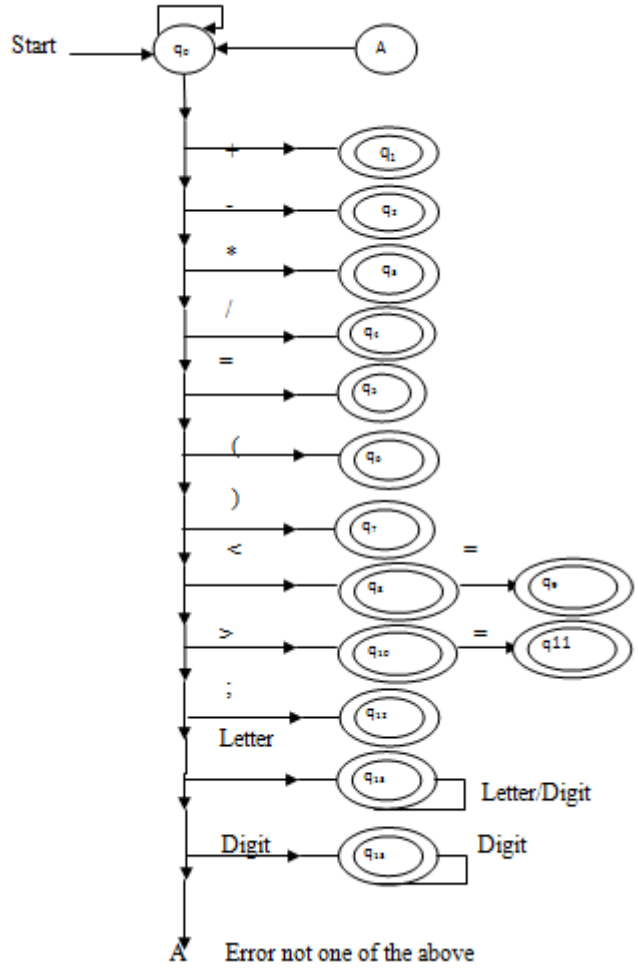
else error();

State q_2: retract();

return (id, install ( )),

Example: Consider the language L which consists of tokens +,-,*,/,=,<,>,=,<=,>=,; identifier keyword and constants.

We shall construct FA scanner for L. The scanner ignores blanks and comments. The following figure gives the required FA.



Token Returned

+

-

*

/

=

(

)

<=

>=

;

If token in keyword table then keyword else identifier ,: add to symbol table

Constant: add to symbol table

FA for reorganization of Tokens

Figure above shows if tokens +,*,/,- are matched they are returned . if there is >,scanner check the next character if it is = it then returns ,< = else < .State q_{13} identifies identifiers .If it is present in keyword table then it is returned it as keyword otherwise it identifiers. Constant compares all digits until next non numeric character is matched.

After representing then scanner as a FA, it can be easy implemented, by writing code of each state. Another method, which is used to defining tokens by regular expression and we know that we can construct FA equivalent to regular expression.

The automatic lexical analyzer generator like flex can be used to generate lexical analyzer.

Questions:

1) What is the role of Lexical Analyzer? Explain in detail.
2) Explain input buffering in detail.
3) How specify and recognize the tokens?
4) Explain issues in Lexical Analyzer.
5) How to design Lexical Analyzer generator for generation of tokens?
6) Define: 1) alphabet 2) String 3) Language 4) Lexeme

5) token 6) pattern 7) regular expression 8) Finite Automata