PART A

(PART A : TO BE REFFERED BY STUDENTS)

Experiment No. 04(A)

Aim: Write a program to implement Lexical Analyzer for given language using Finite Automata.

Objective: Develop a program to implement lexical analyzer using Finite Automata

Outcome: Students are able to design and implement lexical analyzer for given language.

Theory:

The very first phase of compiler is lexical analysis. The lexical analyzer read the input characters and generates a sequence of tokens that are used by parser for syntax analysis. The figure below summarizes the interaction between lexical analyzer and parser.

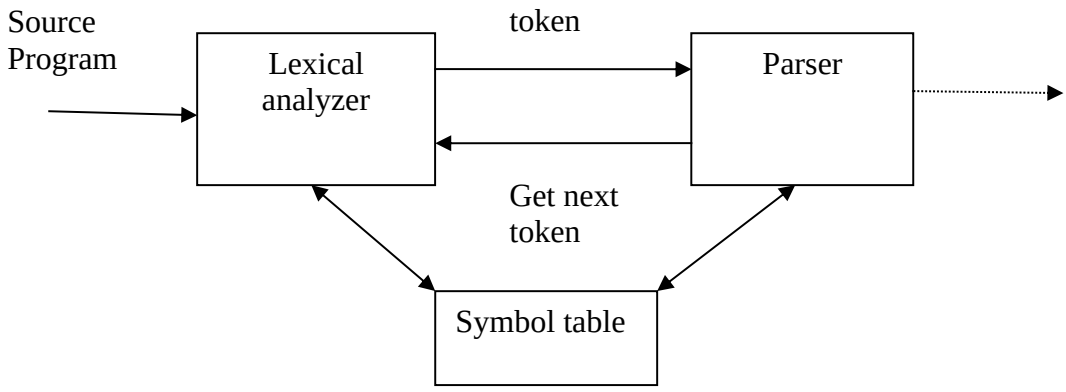


FIGURE: Interaction between lexical analyzer and parser

The lexical analyzer is usually implemented as subroutine or co-routine of the parser. When the "get next token" command received from parser, the lexical analyzers read input characters until it identifies next token.

Lexical analyzer also performs some secondary tasks at the user interface, such as stripping out comments and white spaces in the form of blank, tab and newline characters. It also correlates error messages from compiler to source program. For example lexical analyzer may keep track of number of newline characters and correlate the line number with an error message.

In some compilers, a lexical analyzer may create a copy of source program with error messages marked in it. An important notation used to specify patterns is a regular expression.Each pattern matches a set of strings. Regular expression will serve as a name for set of strings.

A *recognizer* for a language is a program that takes as input a string *x* and answer "yes" if *x* is a sentence of the language and "no" otherwise.

We compile a regular expression into a recognizer by constructing a generalized transition diagram called a *finite automaton*.

We will design lexical analysis for the language which consists of strings containing "abb" as substring.

Thus, L={"abb", "babb", "abbabba",…}

The DFA for this language is represented in transition table below:

Q / Σ	a	b
q0	q1	q0
q1	q1	q2
q2	q1	q3
q3	q3	q3

Now we will simulate this DFA to generate lexical analyzer for the language L.

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

The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q0)
- Set of final states (qf)
- Transition function (δ)

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

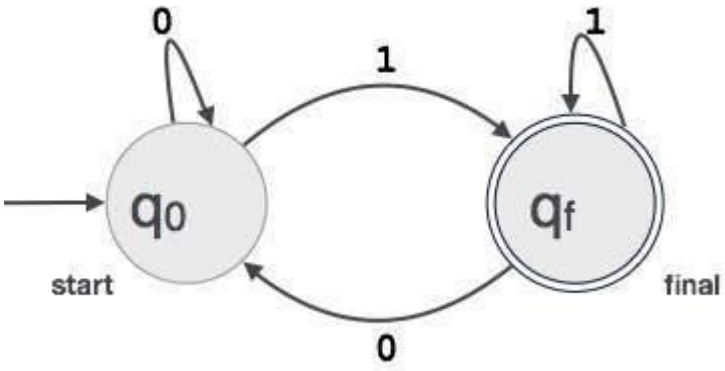
Finite Automata Construction

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 stays on the same state, an arrow pointing from a state to itself is drawn.

Example : We assume FA accepts any three digit binary value ending in digit 1. FA = {Q(q_0, q_f), Σ(0,1), q_0, q_f, δ}



PART B

(PART B : TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded at the end of the practical)

Roll. No. B48	Name: Aryan Unhale
Class: TE COMPS B	Batch:B3
Date of Experiment:25/2/25	Date of Submission:25/2/25
Grade:	

B.1 Software Code written by student:
Steps before running:
1.Download Flex (https://gnuwin32.sourceforge.net/packages/flex.htm)
2.Search for GnuWin32\bin and Path in System Environment variables (C:\Program Files (x86)\GnuWin32\bin)
3.Make sure MinGW is already there in environment variables(C:\MinGW\bin)

Exp4.l (General lexical analyzer):

```
%{
#include <stdio.h>
#include <stdlib.h>

int COMMENT=0;
%}

identifier [a-zA-Z_][a-zA-Z0-9_]*
number [0-9]+
operator [\+\-\*/=<>]
punctuation [;,\(\)\{\}]

%%

#.* { printf("\n%s is a preprocessor directive", yytext); }

int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
```

```
continue |
void |
switch |
return |
else |
goto { printf("\n%s is a keyword", yytext); }

"/*" { COMMENT=1; printf("\n%s is a COMMENT", yytext); }

{identifier}\( { if (!COMMENT) printf("\nFUNCTION: %s", yytext); }
\{ { if (!COMMENT) printf("\nBLOCK BEGINS"); }
\} { if (!COMMENT) printf("\nBLOCK ENDS"); }

{identifier}(\[[0-9]*\])? { if (!COMMENT) printf("\n%s is an IDENTIFIER", yytext); }

\".*\" { if (!COMMENT) printf("\n%s is a STRING", yytext); }

{number} { if (!COMMENT) printf("\n%s is a NUMBER", yytext); }

{operator} { if (!COMMENT) printf("\n%s is an OPERATOR", yytext); }

{punctuation} { if (!COMMENT) printf("\n%s is a PUNCTUATION", yytext); }

%%

int main(int argc, char **argv)
{
   FILE *file;

   if (argc < 2) {
      printf("Usage: %s <input_file>\n", argv[0]);
      return 1;
   }

   file = fopen(argv[1], "r");
   if (!file) {
      printf("Could not open the file: %s\n", argv[1]);
      return 1;
   }

   yyin = file;
   yylex();
   fclose(file);

   printf("\n");
   return 0;
}

int yywrap()
{
```

```
    return 1;
}
```

DFA.l(DFA Lexical Analyzer):

```
%{
#include <stdio.h>
%}

%%
.*abb.*   { printf("ACCEPTED: String contains 'abb'\n"); }
[a-b]+    { printf("NOT ACCEPTED: String does not contain 'abb'\n"); }

%%

int main() {
    yylex();
    return 0;
}

int yywrap() {
    return 1;
}
```

B.2 Input and Output:

Exp4.c(For General Lexical Analyzer):

```
#include <stdio.h>

int main()
{
    int a = 2, b = 3, c, d = 4;

    c = a + b * d;

    printf("Result: %d\n", c);

    return 0;
}
```

Output(for General Lexical Analyzer):

```
ction SEM6\Lex Programs>flex Exp4.l

ction SEM6\Lex Programs>gcc -o Exp4 lex.yy.c

ction SEM6\Lex Programs>Exp4.exe Exp4.c

#include <stdio.h> is a preprocessor directive


int is a keyword
FUNCTION: main(
) is a PUNCTUATION

BLOCK BEGINS

int is a keyword
a is an IDENTIFIER
= is an OPERATOR
2 is a NUMBER
, is a PUNCTUATION
b is an IDENTIFIER
= is an OPERATOR
3 is a NUMBER
, is a PUNCTUATION
c is an IDENTIFIER
, is a PUNCTUATION
d is an IDENTIFIER
= is an OPERATOR
4 is a NUMBER
; is a PUNCTUATION


c is an IDENTIFIER
= is an OPERATOR
a is an IDENTIFIER
+ is an OPERATOR
b is an IDENTIFIER
* is an OPERATOR
d is an IDENTIFIER

FUNCTION: printf(
"Result: %d\n" is a STRING
, is a PUNCTUATION
c is an IDENTIFIER
) is a PUNCTUATION
; is a PUNCTUATION


return is a keyword
0 is a NUMBER
; is a PUNCTUATION

BLOCK ENDS
```

Output(for DFA lexical Analyzer):

```
uction SEM6\Lex Programs>flex DFA.l

uction SEM6\Lex Programs>gcc lex.yy.c -o DFA

uction SEM6\Lex Programs>DFA.exe
abb
ACCEPTED: String contains 'abb'

bab
NOT ACCEPTED: String does not contain 'abb'

aab
NOT ACCEPTED: String does not contain 'abb'

ababb
ACCEPTED: String contains 'abb'

aaabb
ACCEPTED: String contains 'abb'

abaabbab
ACCEPTED: String contains 'abb'
```

B.3 Observations and learning:

In this experiment, we designed and implemented a lexical analyzer using Finite Automata to recognize strings that contain the substring "abb". The program was developed using Flex (Lex) and C, where the DFA transitions through different states based on input characters. After compiling and running the program, it successfully identified strings that contained "abb" as a substring while rejecting others. The results demonstrated the correct working of the DFA, ensuring proper state transitions and acceptance conditions.

B.4 Conclusion:

The experiment effectively demonstrated how Finite Automata can be used for lexical analysis by recognizing patterns within a given language. Using regular expressions in Lex, we simplified the process of token recognition, making it more efficient. This approach can be extended to develop more complex lexical analyzers for programming languages, contributing to compiler design and text processing applications.

B.5 Question of Curiosity

1. What is token?

In general, a token is an object that represents something else, such as another object (either physical or virtual), or an abstract concept as, for example, a gift is sometimes referred to as a token of the giver's esteem for the recipient. In computers, there are a number of types of tokens.

2. What is the role of lexical analyzer?

The lexical analyzer is responsible for removing the white spaces and comments from the source program. It corresponds to the error messages with the source program. It helps to identify the tokens. The input characters are read by the lexical analyzer from the source code.

3. What is the output of Lexical analyzer?

The output of the lexical analyzer is tokens.