Development of <Lexical Analyzer> for <Compiler>

Course Name: Compiler Design
Course Code: CSC 437
Section: A

A Lexical Analysis and Design Report Submitted by:

SL	ID	Name
29	20103167	Md. Min Khayer

Submitted to:

Mr. Krishna Das,

Asst. Professor,

Department of Computer Science and Engineering

College of Engineering and Technology

IUBAT– International University of Business Agriculture and Technology

Summer 2022

ACKNOWLEDGMENTS

First and foremost, praise and thanks to the Almighty, for His showers of blessings throughout my work to complete the Compiler Design course successfully.

I would like to express deep and sincere gratitude to my Course Instructor, Assistant Prof. Krishna Das for giving me the opportunity to do this assignment and for the continuous support, motivation, enthusiasm, and immense knowledge. His guidance helped me throughout the course and writing of this report. I could not have imagined having better course instructor and mentor for doing this course.

Last but not least, I would like to thank our parents, friends and respondents who have helped me with their valuable suggestions and guidance has been very helpful in various phases of the completion of this course.

ABSTRACT

Lexical analysis is a critical process in the initial phase of program compilation, where the source code is parsed into a sequence of meaningful tokens. This process involves reading the stream of characters from the source code, grouping them into lexemes, and producing tokens that serve as input for the syntax analyzer. Essential to this process is the ability to recognize various types of tokens, including keywords, identifiers, numbers, operators, and delimiters. The development of a lexical analyzer requires the construction of finite automata (FA) or transition diagrams that can identify these tokens. By merging individual FAs for each token type, a comprehensive FA is created to handle the diverse elements of a programming language. Keywords are distinguished using data structures such as hash maps or string arrays, ensuring accurate classification. The design of a lexical analyzer also involves strict adherence to token specifications, where identifiers and numbers must meet specific criteria based on predefined character sets. The implementation phase translates the FA design into a working program, capable of reading input code, recognizing tokens, and handling errors effectively. This process not only demands a deep understanding of lexical analysis and finite automata but also showcases the practical application of these concepts in building a functional compiler component. The end result is a detailed sequence of tokens that accurately represents the input code, laying a solid foundation for subsequent phases of compilation.

Table of Contents

Chapter I.

1.1 LEX

Lex is a lexical analyzer generator tool also known as tokenizers, which recognize lexical patterns in text. It is a program that generates lexical analyzer (scanner) used in the lexical analysis phase of a compiler. It is used with YACC parser generator.

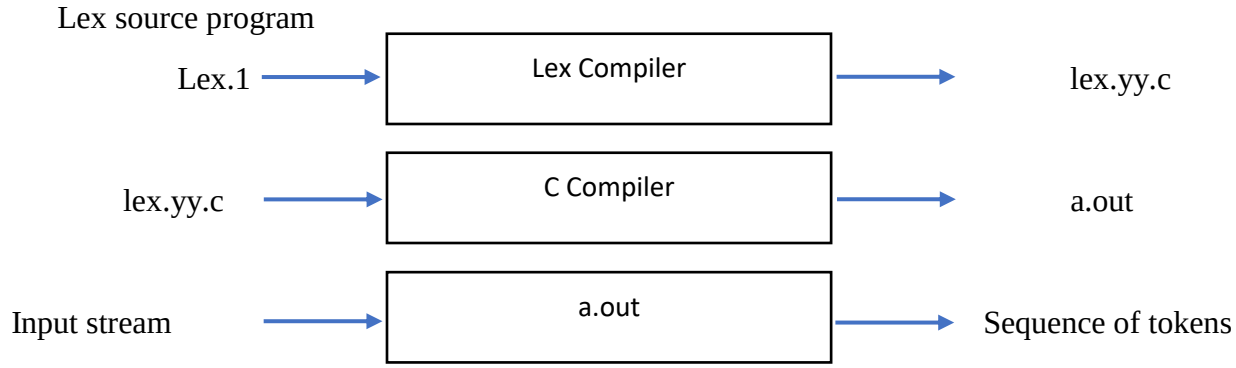
The lexical analyzer is a program that transforms an input stream into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code. If the lexical analyzer detects that the token is invalid, it generates an error. Lexical Analysis can be implemented with the Deterministic Finite Automata. The role of Lexical Analyzer in compiler design is to read character streams from the source code, check for legal tokens, and pass the data to the syntax analyzer when it demands.

1.1.1 The Process of Lexical Analysis

The lexical analyzers are divided into two process:

1) **Scanning**: It is a scanning process that take modified source code as an input. Through the input buffering and with the help of sentinels efficiently scan the source code. This process that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
2) **Analysis**: Lexical analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Lex process: It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.



Firstly, lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.

lex.yy.c is compiled by the C compiler to a file called a.out. The normal use of the compiled C program, referred to as a.out.

The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

Chapter II.

2.1 Problem Analysis

2.1.1 Token and its validity

The input file source code contains some of the code string. Each of the string need to be identifying as a token. Some of the tokens are valid and some of the tokens are not valid. For this validity purpose we follow the grammar rules that we have been set.

In the input file source code program, we use only while, if, else, return, break, continue, int, float, void and for as a reserved word. Other string or letter consider as an identifier. But we also set the rules of grammar for the identifier. Only valid identifier must contain {a, b, e, g, h, i, k, m, n, o, r, u, y, A, B, E, G, H, I, K, M, N, O, R, U, Y} these letters. Identifier must start with letter. After starting with letter the digit may come that is valid. The valid digit only {0, 1, 2, 3, 6, 7} these numbers. And without satisfying all these condition the identifier consider as an error identifier (invalid). In case of the number the digit must come from {0, 1, 2, 3, 6, 7} these. The number must start with these digits. If the number is starting with dot (.), 4, 5, 8, 9 then it is not valid number. The number can be formed as: (some of the example shown in below)

digits→ digit digit∗ 22

optional-fraction → (. digits) | s 22.22

optional-exponent → (E (+ | − | s) digits) | s

22E+22, 22E-22, 22E22, 22.22E+22, 22.22E-22, 22.22E22

num→ digits optional-fraction optional-exponent

22, 22.22, 22.22E22, 22.22E+22, 22.22E-22, 22E22, 22E+22, 22E-22

So, this types of formation of number is valid for this grammar rules. The number combination must contain only the valid digit 0, 1, 2, 3, 6, 7.

If the input file source code contains the operators, then the operators must be identified as a token. The token name will be the types of the operator's name.

Finally, the brackets and semicolon identify as a token. The token name is similar as like the brackets & semicolon symbol.

Input file source code Token specification: (for valid)

Token Type	Lexical Specification
keyword	void, float, int, while, for, if, break, else, continue, return,
identifier	main, min, khayer, i, BOGURA, bogura,
num	0, 20103167, 10, 1, 10.33, 3
(	(
)	)
{	{
Assignop	=
;	;
Relop	<, ==, !=, >
Addop	+
Subop	-
Mulop	*
Divop	/
}	}
And	&&
Or	||
not	!

Input file source code Token specification: (for invalid)

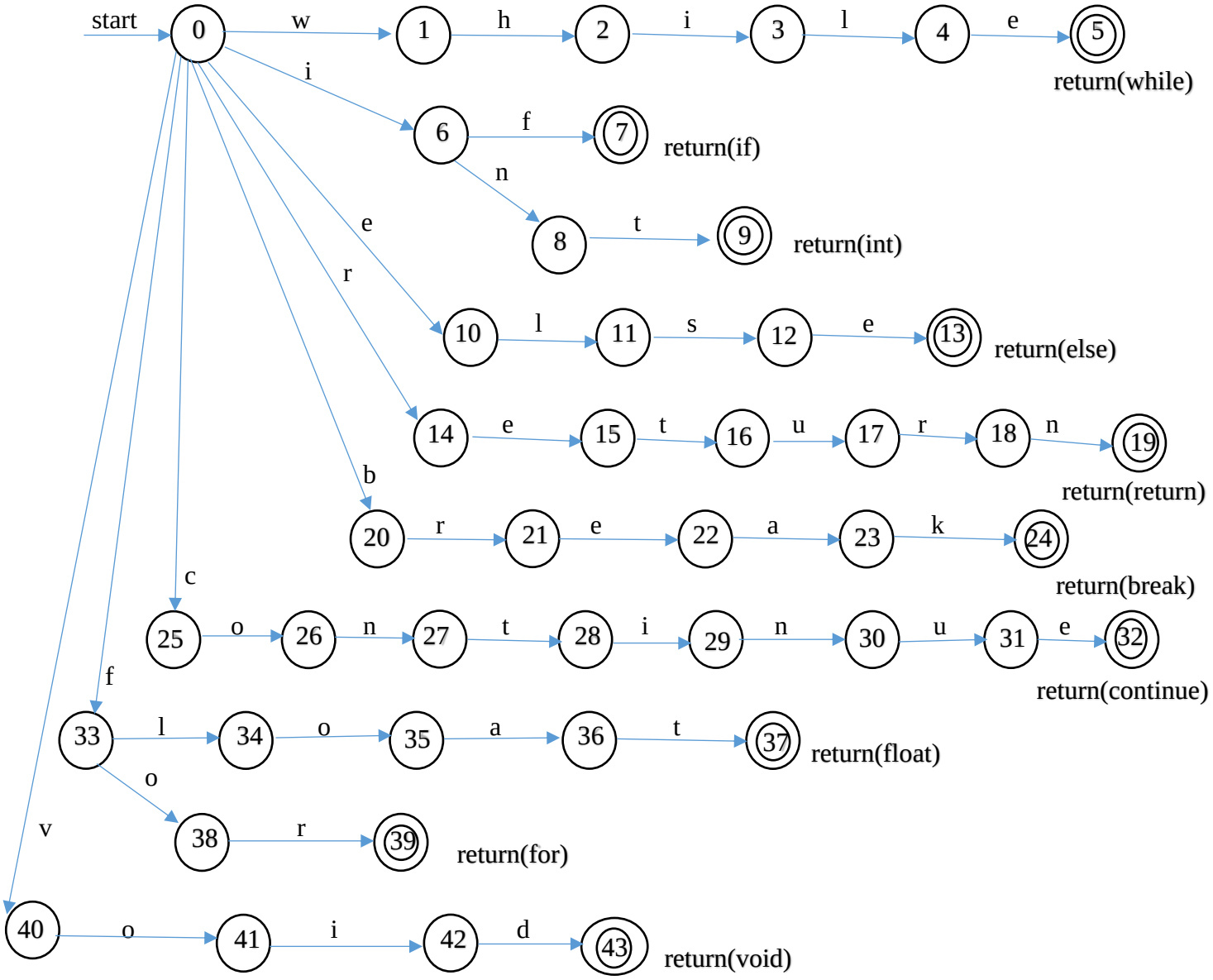
Error identifier	min.khayer, faridpur, result,
Error number	5, 34E4, 45.34E-4, 20103168, .34,

Chapter III.

3.1 Design the Transition Diagram

Transition Diagram for Keyword (Reserved words):

Keyword = {while, if, else, return, break, continue, int, float, void, for}



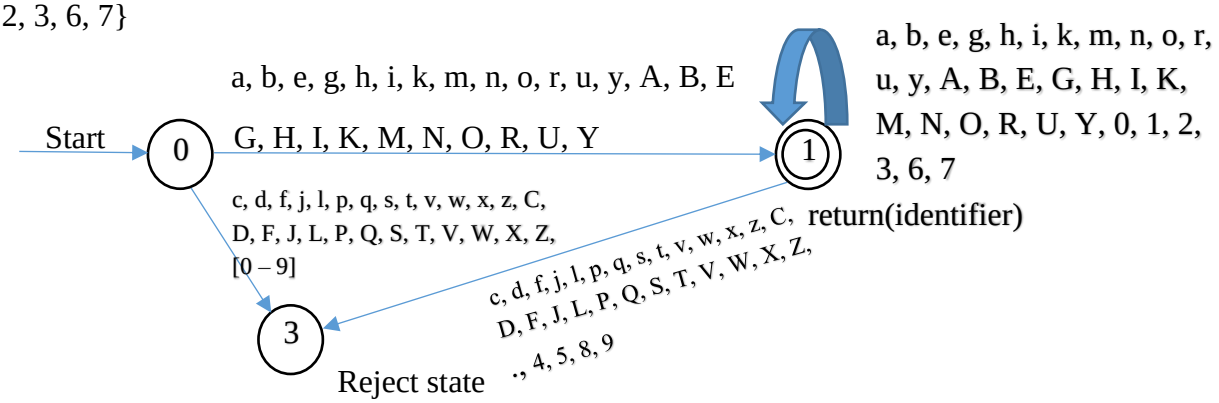
Single FA (Transition Diagram) for Identifier (ID):

My full name: Minkhayer & Home district: Bogura

My ID: 20103167 & DOB: 31-12-2001

Letter = {a, b, e, g, h, i, k, m, n, o, r, u, y, A, B, E, G, H, I, K, M, N, O, R, U, Y}

Digit = {0, 1, 2, 3, 6, 7}



Single FA (Transition Diagram) for unsigned Numbers:

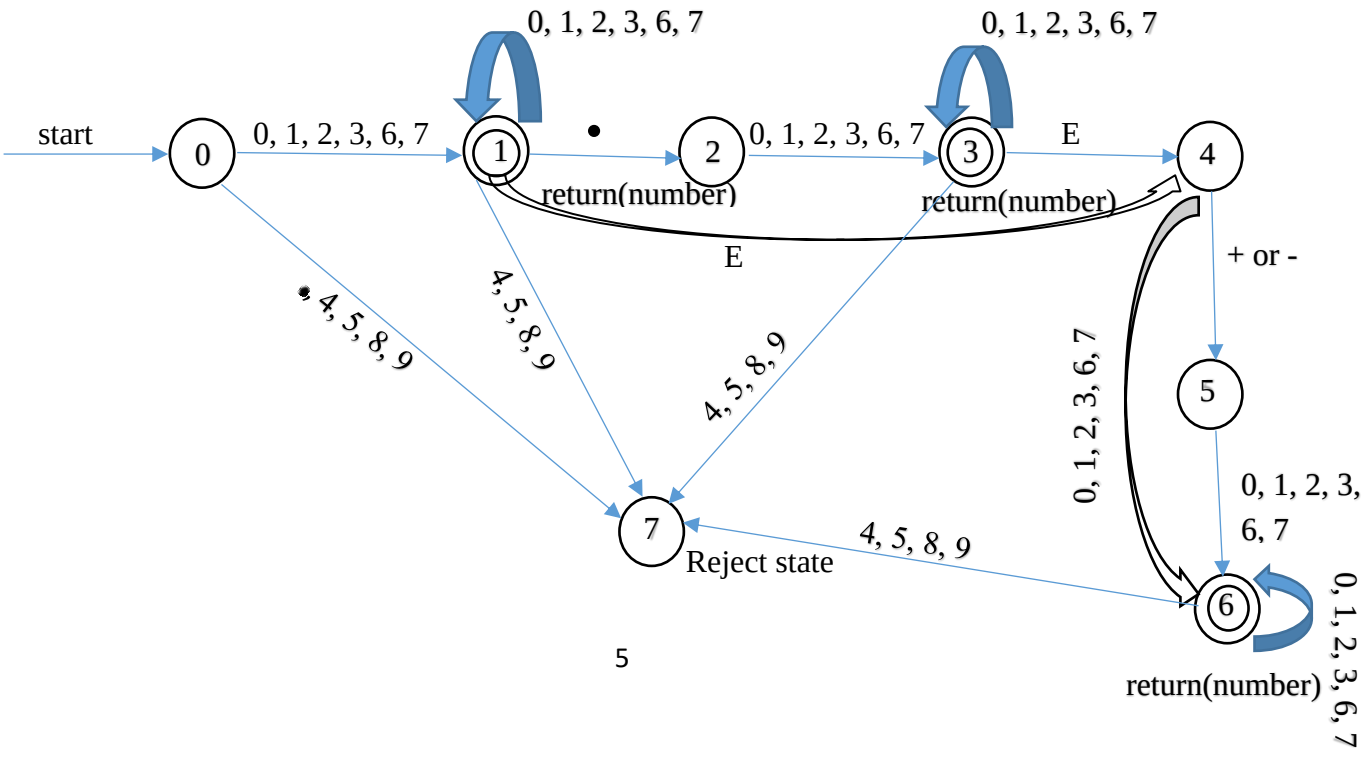
Digit = {0, 1, 2, 3, 6, 7}

digits→ digit digit∗

optional-fraction → (. digits) | s

optional-exponent → (E (+ | − | s) digits) | s

num→ digits optional-fraction optional-exponent

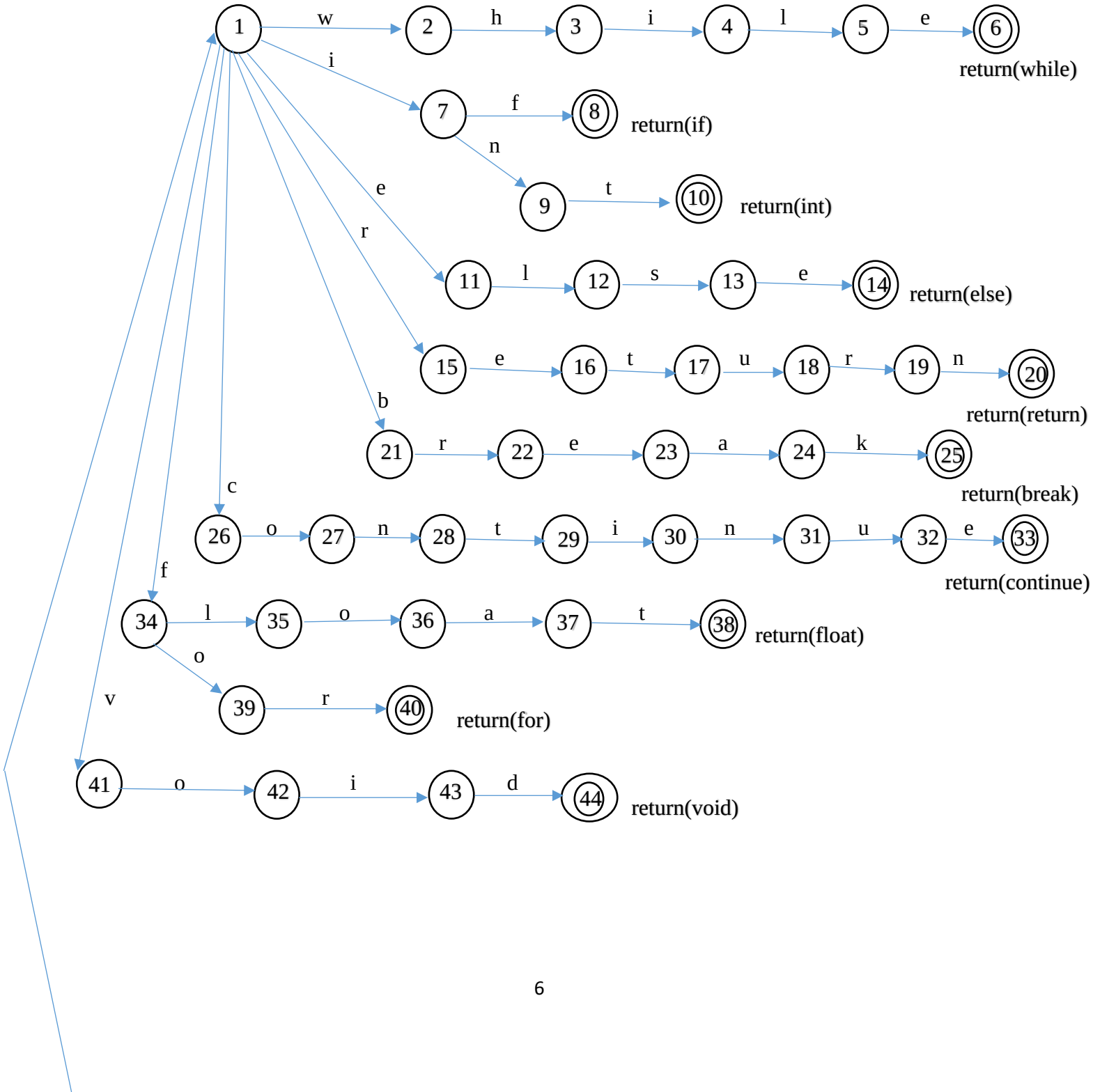


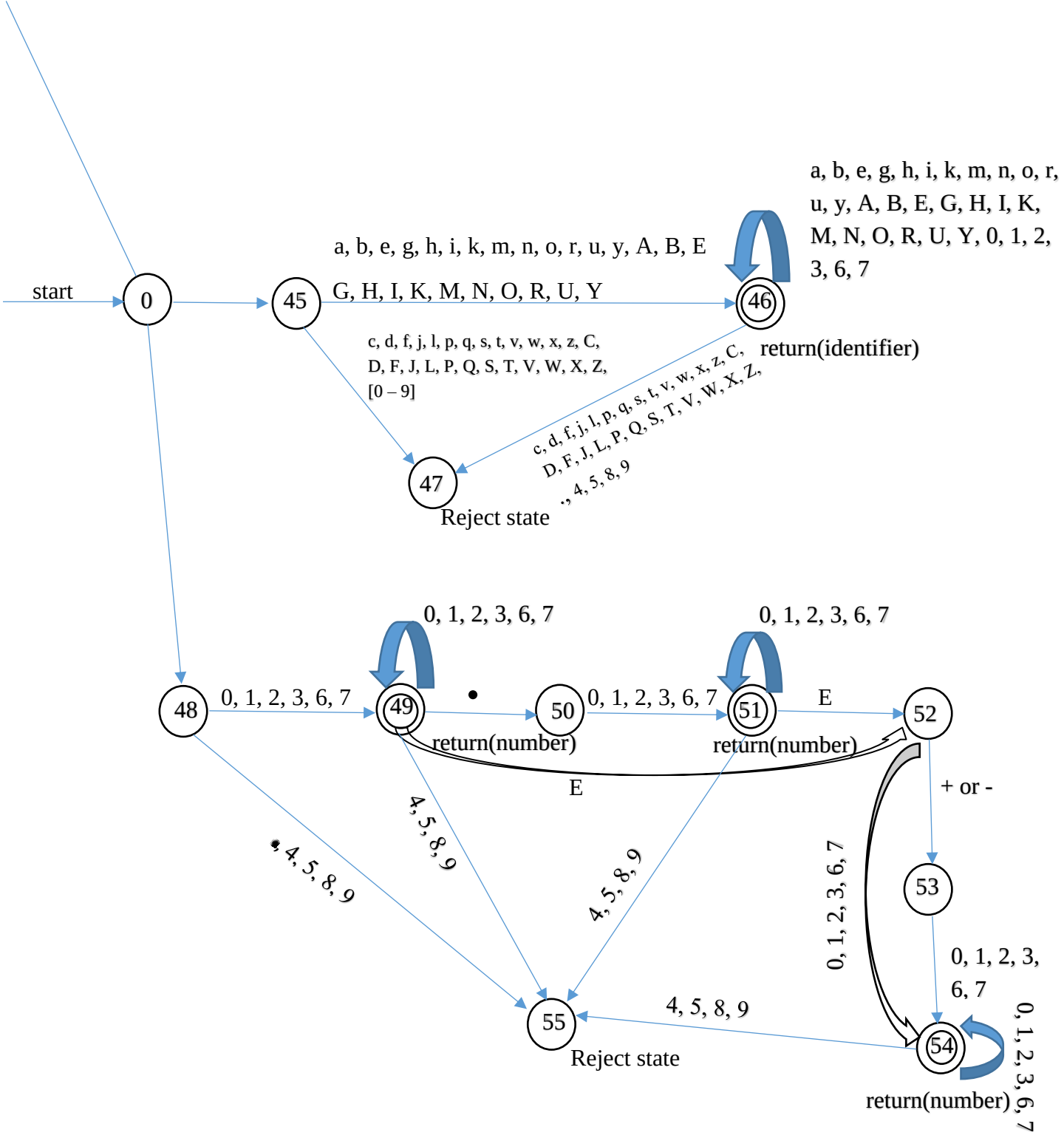
Combined all single FA (Transition Diagram) for Keyword (Reserved words), Identifier (ID) & Unsinged Numbers:

Keyword = {while, if, else, return, break, continue, int, float, void, for}

Letter = {a, b, e, g, h, i, k, m, n, o, r, u, y, A, B, E, G, H, I, K, M, N, O, R, U, Y}

Digit = {0, 1, 2, 3, 6, 7}




start
0
45
a, b, e, g, h, i, k, m, n, o, r, u, y, A, B, E
G, H, I, K, M, N, O, R, U, Y
46
a, b, e, g, h, i, k, m, n, o, r, u, y, A, B, E, G, H, I, K, M, N, O, R, U, Y, 0, 1, 2, 3, 6, 7
return(identifier)
c, d, f, j, l, p, q, s, t, v, w, x, z, C, D, F, J, L, P, Q, S, T, V, W, X, Z, [0 – 9]
c, d, f, j, l, p, q, s, t, v, w, x, z, C, D, F, J, L, P, Q, S, T, V, W, X, Z, ., 4, 5, 8, 9
47
Reject state
48
0, 1, 2, 3, 6, 7
49
0, 1, 2, 3, 6, 7
50
return(number)
0, 1, 2, 3, 6, 7
51
0, 1, 2, 3, 6, 7
return(number)
E
52
E
+ or -
53
0, 1, 2, 3, 6, 7
54
0, 1, 2, 3, 6, 7
return(number)
55
Reject state
4, 5, 8, 9
4, 5, 8, 9
4, 5, 8, 9
4, 5, 8, 9
0, 1, 2, 3, 6, 7

3.2 Working Process of the Designed Transition Diagram

The regular expression follows some pattern or sequence or rules. For recognizing or identifying the token, design the stylized flowcharts of the regular expression. The lexical analyzer, first convert patterns into stylized flowcharts, called "transition diagrams." Through the transition diagram represent the symbol logically. Depending on the string or symbol what type of token it is that could be identify by the Finite automata machine. It is a decision making process.

The two main components are circles representing states (think of them as decision points of the lexer) and arrows representing edges (think of them as the decisions made). One state is designated the start state, or initial state it is indicated by an edge, labeled "start". The other state of this design one is non accepting state & another one is accepting state or final state. The accepting state represent by a double circle. This is returning the token keyword, identifier & Unsigned numbers.

This could be done by the lexical analyzer. The lexical analyzer will recognize the keyword, identifier & unsigned number if the lexemes match with that pattern.

At first We draw the single FA (transition diagram) for identifying the keyword (Reserved word). Then draw the transition diagram for identifier & Unsigned number. All of them draw separately. After that merge their start states of three diagrams to create a single transition diagram. So, the finite automata machine can easily match the pattern for keyword, identifier, unsigned number and recognize the token. We begin in state 0, the start state. If the input file source code contains the keyword {while, if, else, return, break, continue, int, float, void, for} that match the pattern for keyword, then it returns the token type keyword. If the input file source code contains the identifier that comes from my full name and home district letter only {a, b, e, g, h, i, k, m, n, o, r, u, y, A, B, E, G, H, I, K, M, N, O, R, U, Y} match the pattern for identifier, then it returns the token type identifier. Without these letters, the other letters have to be rejected state.

If the input file source code contains the digit that comes from my id and DOB number {0, 1, 2, 3, 6, 7} only match the pattern for number, then it returns the token type num. Without these digits, the other digits have to be rejected state. Also if the dot (.) come in the first position, then this one also goes to the rejected state. It is showing an error num.

Chapter IV.

4.1 Implementing the design through a programming language

Implementing the design of a lexical analyzer through a programming language involves several key steps. First, the input file containing source code is read using standard file I/O operations. Tokens are then defined and validated based on lexical specifications, such as keywords, identifiers, and numbers, often using regular expressions or state machines. Finite automata (FA) are constructed for each token type, handling state transitions and recognizing valid tokens. The program processes the input character by character, updating the FA's state and generating tokens upon reaching terminal states. Special cases and errors are managed to ensure robustness, and the tokens are output in the required format. This implementation translates the theoretical principles of lexical analysis into a practical tool that effectively tokenizes source code, providing a foundation for subsequent compilation phases.

Chapter V.

5.1 Source code of implementing Lexical Analyzer

Input File: (C++ Code)

Program file name: inputfile.cpp

```
void main ( )
{
    min.khayer ;
    float min = 0 ;
    int khayer = 20103167 ;
    while ( )
    {
    for ( int i = 0 ; i < 10 ; i + 1 )
    {
        if ( i == 5 )
        break ;
        min = khayer + i - 10.33 + 34E4 / 45.34E-4 + 20103168 *
.34 ;
        BOGURA ;
        bogura ;
        faridpur ;
        20103167 ;
        20103168 ;
        else { continue ; }
    }
    return min ;
    }
    result = ( 3 != 5 ) && ( 3 < 5 ) ;
    result = ( 3 == 5 ) || ( 3 > 5 ) ;
    result = ! ( 5 == 5 ) ;
}
```

Program that reads an input text file: (JavaScript code)

Program file name: readfile.js

```
const fs = require("fs");
const keywords = ["while", "if", "else", "return", "break",
"continue", "int", "float", "void", "for"]
const relOps = ["<", ">", "<=", ">=", "==", "!="]
const logicalOp = ["&&", "||", "!"]
const operators = ["+", "-", "*", "/", "="]
const brackets = ["{", "}", "(", ")", "[", "]"]
const numbers = ["0", "1", "2", "3", "6", "7", ".", "E", "-"]
const identifier = ['a', 'b', 'e', 'g', 'h', 'i', 'k', 'm',
'n', 'o', 'r', "u", 'y', 'A', 'B', 'E', 'G', 'H', 'I', 'K',
'M', 'N', 'O', 'R', 'U', 'Y'];
//My name is minkhayer and home district: Bogura
//My ID is 20103167 and DOB: 31/12/2001
// 012367
// abeghikmnoruy
// ABEGHIKMNORUY
const lexical = () => {
    let data = fs.readFileSync("./inputfile.cpp", { encoding:
"utf8" });
    //console.log(data); //load the input file
    let dataArray = data.split("\n");
    //console.log(dataArray)
    dataArray = dataArray.map((each) => {
        return each.replace('\r', '').replace('\t',
'').replace('\t', '')
    }, [])
    //console.log(dataArray)
    for (each of dataArray) {
        if (each[0] === "#") {
            continue;
        }
        line = each.split(" ")
        for (word of line) {
            if (word === '') continue;
            else if (keywords.includes(word)) {
                console.log("keyword: " + word);
```

```
        }
        else if (relOps.includes(word)) {
            console.log(word + ":Relop, " + word);
        }
        else if (operators.includes(word)) {
            if (word == "+") {
                console.log(word + ": Addop, " + word);
            }
            else if (word == "-") {
                console.log(word + ": Subop, " + word);
            }
            else if (word === "=") {
                console.log("=: Assignop, " + word);
            }
            else if (word == "/") {
                console.log(word + ": Divop, " + word);
            }
            else if (word === "*") {
                console.log("=: Mulop, " + word);
            }
        }
        else if (logicalOp.includes(word)) {
            if (word == "&&") {
                console.log(word + ": And, " + word);
            }
            else if (word == "||") {
                console.log(word + ": Or, " + word);
            }
            else if (word === "!") {
                console.log("=: not, " + word);
            }
        }
        else if (brackets.includes(word)) {
            console.log(word + ":" + word);
        }
```

```
            else if (word === ";") {
                console.log(word + ":" + word);
            }
            else if (word[0] == ".") {
                console.log("Error: .num:" + word);
            }
            else if ((+word[0]) >= 0 && (+word[0]) <= 9) {

                let cnt = 0;
                for (let i = 0; i < word.length; i++) {
                    if (numbers.includes(word[i])) cnt++;
                }
                //console.log("The count number: " + cnt);
                if (word.length > 0 && word.length == cnt) {
                    console.log("num :" + word);
                }
                else {
                    console.log('Error number:' + word);
                }
            }
            else {
                //word = word.toLowerCase();
                let cnt = 0;
                for (let i = 0; i < word.length; i++) {
                    if (identifier.includes(word[i])) cnt++;
                }
                if (word.length > 0 && word.length == cnt) {
                    console.log("identifier: " + word);
                }
                else {
                    console.log("Error Idenfifier: " + word);
                }
            }
        }
    }
}
```

Chapter VI.

6.1 The output of the program of Lexical Analyzer

```
The keyword: void
identifier: main
(:(
):)
{:{
Error Idenfifier: min.khayer
;:;
keyword: float
identifier: min
=: Assignop, =
num :0
;:;
keyword: int
identifier: khayer
=: Assignop, =
num :20103167
;:;
keyword: while
(:(
):)
{:{
keyword: for
(:(
keyword: int
identifier: i
=: Assignop, =
num :0
;:;
identifier: i
<:Relop, <
num :10
;:;
identifier: i
```

```
+: Addop, +
num :1
):)
{:{
keyword: if
(:(
identifier: i
==:Relop, ==
Error number:5
):)
keyword: break
;:;
identifier: min
=: Assignop, =
identifier: khayer
+: Addop, +
identifier: i
-: Subop, -
num :10.33
+: Addop, +
Error number:34E4
/: Divop, /
Error number:45.34E-4
+: Addop, +
Error number:20103168
=: Mulop, *
Error: .num:.34
;:;
identifier: BOGURA
;:;
identifier: bogura
;:;
Error Idenfifier: faridpur
;:;
num :20103167
;:;
```

```
Error number:20103168
;:;
keyword: else
{:{
keyword: continue
;:;
}:}
}:}
keyword: return
identifier: min
;:;
}:}
Error Idenfifier: result
=: Assignop, =
(:(
num :3
!=:Relop, !=
Error number:5
):)
&&: And, &&
(:(
num :3
<:Relop, <
Error number:5
):)
;:;
Error Idenfifier: result
=: Assignop, =
(:(
num :3
==:Relop, ==
Error number:5
):)
||: Or, ||
(:(
num :3
```

```
>:Relop, >
Error number:5
):)
;:;
Error Idenfifier: result
=: Assignop, =
=: not, !
(:(
Error number:5
==:Relop, ==
Error number:5
):)
;:;
}:}
```

The output of this program we have seen each and every symbol or string has been detected as a stream of token. The input file source code all the string recognizes as a token. This program output we found keyword, identifier, and number, all types of operators, brackets and semicolon. If the string not match with the identifier & number pattern then, it will show the error identifier and error number. Like: min, khayer, bogura are detecting as an identifier but faridpur is not detect as an identifier. It is detecting as an error identifier. The same thing will be happened when detect the number. 0, 20103167, 10 are detecting as a number. But 20103168, .34, 5 are detecting as an error number.