Objectives:

1. To write a program in flex to tokenize a source file
2. To design own programming language for the source file
3. To verify the efficiency and accuracy of the lexical analyzer's output with expected results.

Introduction:

Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. It is a computer program that generates lexical analyzers (also known as "scanners" or "lexers"). Flex takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. Scanners perform lexical analysis by dividing the input into meaningful units. For a C or C++ program the units are *variables, constants, keywords* etc. These units also called as tokens. Flex can only generate code for C and C++. To use the scanner code generated by flex from other languages a language binding tool such as SWIG can be used.

The command prompt script for running flex program :

1. flex 1907011.l
2. gcc lex.yy.c -o output.exe
3. ./output.exe

Directives:

1. **##importing<stdio.h>**
   - To include a header file, we need to use the keyword "##**importing**" This is the header file of My own programming language
2. **##importing<math.h>**
   This is the header file for mathematical calculations

Data Types:

1. **integer:**

integer variableName $ 10

2. **Double:**

doublefloating variableName $ 10.1

3. **String:**
word variableName $ str
5. **Char:**
letter variableName $ ‘a’

Variable Declaration:

The Rules for declaring variables are,

1. The variable must begin with ‘**varr_**’.
2. Immediate After the word ‘varr_’ there must be at least one character.
3. Numbers maybe included while naming the variables.

For Example,

integer abc //wrong

integer var_a_bc //wrong

integer varr_a_bc //right

Operations & Operators:

Additon:
Operator: +
Syntax: varr_1 $ varr_2 + varr_3

Subtracton:
Operator: -
Syntax :varr_1 $ varr_2 + varr_3

Multiplication:
Operator: *
Syntax: varr_1 $ varr_2 * varr_3

Division:
Operator: |
Syntax: varr_1 $ varr_2 | varr_3

Increment:
Operator: ++

Decrement:
Operator: --

Assignment:
Operator: $

Syntax: varr_1 = varr_2

varr_1 = 10

Comment:

To for single line comment "**!**" is used and for multiline comment "**!> <!**" is used.

For Example:

! This is single line comment

!> this is

multiline comment<!

Conditons:

If:

Syntax: iff [**Condition]**
(
any number of statements
)

If-Else:

Syntax: iff [**Condition]**
(
any number of statements
)
else
(
any number of statements
)

Switch Case:

Syntax: switch **Expression**
[
case: **Constant**
……..
default:

……..
]

Loops:
For Loop:

Syntax: loopfor **variable_initialize : condition : increment/decrement**
[
statements
]

Discussuions: In this assignment ,I have designed my own programming language and test the language by flex.I have included customize variable pattern ,datatype pattern,if else ,loops,single line and multiple line comment.I tested every pattern and counted the number of variables,keywords,statement .I have got some errors while matching the conditional statement and loop statements. Creating our own language and testing its accuracy has revealed the difficulties of finding the right balance between language expressiveness and simplicity, particularly in the context of lexer development.

Conclusion: In summary, our experiment has showcased the viability of crafting a fresh programming language with the assistance of Flex. We successfully reached our goals, which encompassed the creation of distinctive language attributes and the establishment of a dependable lexer.

This undertaking lays a robust groundwork for future ventures into language design and compiler construction. The knowledge gained from this endeavor will guide forthcoming endeavors in software engineering and tool development. All in all, the process of building a new language with the aid of Flex has proven to be a valuable educational journey.

Source Code:(1907011.l)

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

        int cntvariable=0;
```

```
        int cntkeyword=0;
        int cntstatement=0;
%}

mainfunc ("main")["()"][\n]?[\{].[\}]
variable ("varr_")[a-zA-Z0-9_]{1,32}
datatype ("integer"|"floating"|"longlonginteger"|"letter"|"word"|"doublefloating")
keyword
("iff"|"elsee"|"loopfor"|"loopwhile"|"show"|"take"|"switch"|"case"|"continue")
integer [+-]?[0-9]{1,5}
floating [+-]?[0-9][0-9]*[.][0-9]+
longlonginteger [+-]?[1-9]{8,12}
numbers ("integer"|"floating"|"longlonginteger")
nulll ["void"]
eofstmnt (";"[\n])
singlelinecmnt [!][a-zA-Z0-9_.,;]+
multilinecmnt "!>"([\n]|[a-zA-Z0-9_.,;]+[\n][a-zA-Z0-9_.,;]+|[a-zA-Z0-9_.,;]+)"<!"
assign_op $
arithmatic_op (+|-|*|/)
relational_op (==|>|<|>=|<=)
increment (++)
decrement (--)
condition ({variable}){[ ]}*{relational_op}{[ ]}*({variable}|{numbers})
ifi ("iff")[ ]*[[].*[]][ ]*[\(].*[\)]
elseifi ("elsee iff")[ ]*[\[]].*[\]][ ]*[\(].*[\)]
elsei  ("elsee")[ ]*[\()].*[\)]
condi_if                {ifi}[ ]*{elseifi}*?{elsei}?
integer_declare {variable}([ ]*{assign_op}[ ]*{numbers})
forloop1 ("loopfor")[ ]{variable}[ ](":")[ ]{variable}[ ]*[\[].*[\]]
forloop (("loopfor")["->"]["([ ]*:[ ]*:[ ]*)"]["->"][\n][\{][\n][\}])|(("loopfor")("-
>")]((integer){[ ]}+)?{integer_declare}{[ ]}*[:]{[ ]}*{condition}{[ ]}*[:]{[
]}*{({variable}{increment})|({variable}{decrement})}{[ ]}*("->"){[ ]}*[\{][\n][\}])



%%
"##importing<stdio.h>" {printf("Header File included -> %s\n",yytext);}
"##importing<math.h>" {printf("Header file for mathematical calculations ->
%s\n",yytext);}
{singlelinecmnt} {printf("Single Line Comment\n");}
```

```
{multilinecmnt} {printf("Multi Line Comment\n");}
{eofstmnt}     {printf("Statement ends\n"); cntstatement++;}
{datatype} {printf("Datatype -> %s\n",yytext);cntkeyword++;}
{keyword} {printf("Keyword -> %s\n",yytext);cntkeyword++;}
{variable} {printf("Identifier -> %s\n",yytext);cntvariable++;}

{ifi} {printf("if block");}

{forloop1} {printf("for loop\n");}

.
%%

int yywrap()
{
        return 1;
}

int main()
{
        yyin=fopen("input.txt","r");
        yylex();
        printf("Total variables :%d\n",cntvariable);
        printf("Total keywords :%d\n",cntkeyword);
        printf("Total Statements :%d\n",cntstatement);
        return 0;
}
```

Input file(input.txt):

```
##importing<stdio.h>
##importing<math.h>

!this is single line comment
!>this is multi
line comment<!

main(){
  integer varr_a $ 0;
  floating varr_ff;
```

```
  iff[varr_a]
  (
    show varr_a;
  )

}
```