Objectives:

- To acquire knowledge about Lexical Analysis and Syntax Analysis in the context of compiler construction.
- To gain hands-on experience with FLEX (Fast Lexical Analysis Generator) for performing lexical analysis.
- To identify, categorize, and analyze tokens, keywords, variables, and other language components using FLEX.
- To design a customized language syntax and implement a syntax analyzer using BISON for parsing.

Introduction:

Lexical analysis and syntax analysis are fundamental phases in the process of compiler Design. Lexical analysis generates tokens from source code, whereas syntax analysis focuses on the hierarchical structure of the code using bison for ensuring syntax and semantics validity.

1. Tokenization: Lexical analysis, while reading a sentence and splitting it into individual words, involves breaking down the source code into tokens. Tokens are the smallest meaningful units, encompassing keywords, identifiers, operators, and literals.

2. Whitespace and Comments: Lexical analysis, carried out using FLEX, disregards whitespace and comments, as they hold no significance in most programming languages. The focus is on extracting meaningful code elements using regular expressions defined in FLEX.

3. Error Detection: FLEX aids in identifying and reporting errors, such as syntax errors, when encountering invalid or unexpected tokens. This process assists programmers in locating and rectifying issues in their code.

4. Symbol Table: Lexical analysis, facilitated by FLEX, often maintains a symbol table or dictionary, tracking identifiers like variable and function names. This information is utilized in subsequent phases of the compilation process.

5. FLEX and BISON Integration: FLEX is employed for lexical analysis, employing regular expressions to recognize and categorize tokens. BISON, on the other hand, is utilized for parsing the syntax of the language, ensuring adherence to predefined grammar rules. Regular expressions in FLEX are

complemented by Bison's parsing capabilities to create a comprehensive language processing system.

6. Output: The output of the combined FLEX and BISON approach is a structured representation of the program, comprising a token stream and a parsed syntax tree. This output serves as input for subsequent phases in the compilation or interpretation process.

In summary, the integration of FLEX for lexical analysis and BISON for syntax analysis allows for the creation of a comprehensive language processing system, encompassing tokenization, error detection, and syntax parsing in the compilation or interpretation of a programming language.

Working of Flex and Bison:

Flex and Bison often work together in the process of creating a compiler or interpreter. Flex is used to generate a lexical analyzer that recognizes patterns in the input code, producing a stream of tokens. Bison, on the other hand, generates a parser that analyzes the syntax of the code based on a context-free grammar. The output of the Flex lexer is fed into the Bison parser, creating a cohesive system that handles both lexical analysis and parsing, essential steps in the compilation process. This combination allows for the efficient development of language processors for custom programming languages.

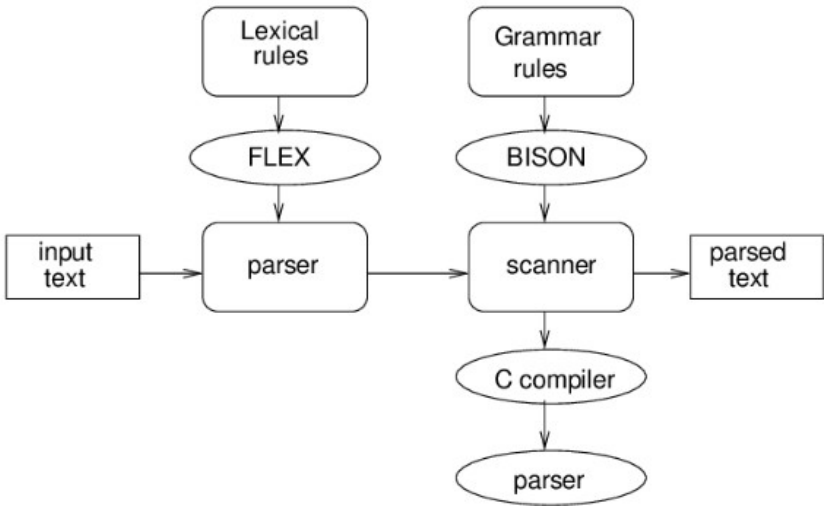

Figure-1: Flex and bison working together

Basic Structure of Flex:

1. Definitions Section:

This section contains any necessary definitions and declarations, such as header files, constants, and global variables.

```
%{
#include <stdio.h>
int wordCount = 0;
%}
```

2. Rules Section:

This is the heart of the flex file, where you define patterns and corresponding actions. Each rule consists of a regular expression and the associated action to be taken when the pattern is matched.

```
%%
[a-zA-Z]+     { wordCount++; printf("Found a word: %s\n", yytext); }
[ \t\n]+      { /* Ignore whitespace and newlines */ }
.             { /* Ignore any other characters */ }
%%
```

3. User Code Section:

This section contains any additional C code that needs to be included in the generated lexer. It typically includes functions or code snippets that are called from the rules section.

```
int main() {
    yylex();
    printf("Total words: %d\n", wordCount);
    return 0;
}
```

Basic Structure of Bison:

1. Definitions Section:

Similar to Flex, this section includes any necessary definitions, declarations, and global variables.

```
%{
#include <stdio.h>
int sentenceCount = 0;
%}
```

2. Rules Section:

In Bison, this section defines the grammar rules using a context-free grammar. Each rule specifies a production and the associated action to be executed.

```
%%
words: word | words word;
word: [a-zA-Z]+;
%%
```

3. User Code Section:

Similar to Flex, this section contains any additional C code needed for the generated parser. It often includes functions or code snippets called from the rules section.

```
int main() {
    yyparse();
    printf("Total sentences: %d\n", sentenceCount);
    return 0;
}
```

Syntax for my Programming language:

Basic Syntax:

Keyword	Operation	Example
;	Ending of a statement	purno a;
shuru	Marks Starting of main	shuru
sesh	Marks Ending of main	sesh
addHeader	Including Header File	addHeader “math.h”
eval	Evaluating Expression	eval 9+3-2*2%5;
show	Printing Variable	show a;
scan	Input from user	scan a;
#comment#	Marks as comment	# multi/single comment #

Data type:

purno	Integer type	purno a = 3;
vogno	Double type	vogno a = 3.2;
shobdo	String type	shobdo a = `65`;

Operational:

inc	Increment	loop(i<4;inc 4)
dec	Decrement	loop(i>100;dec 4)
+ - * / %	Mathematical Operation	eval 9+3-2*2%5;

Conditional:

syntax: if(condition){statements} else{statements}

example:

```
if(i>4){
    show i;
}
else{
    eval 7+8;
}
```

Loop:

syntax: loop(condition;changer){statements}

example:

```
 loop(i>4;inc i){
    show i;
}
```

Function:

1. User Defined function:

Syntax: func function_name(parameters)=>(returnType){statements}

Example:

```
func sum(purno a,vogno b)=>(purno c){
    c = a+b;
}
```

N.B: if function parameters not given => act as void parameters

if function return type with variable not given => act as void return

2. Built in Funtion:

Function Name	Operation	Example
sin	Return float of sine operation	a = sin(30);
cos	Return float of cosine operation	a = cos(30);
tan	Return float of tangent operation	a = tan(30);
log	Return float of e based logarithm	a = log(30);
log10	Return float of 10 based logarithm	a = log10(30);
pow	Return power of a number	a = pow(2,4);
ceil	Return ceil value of a number	a = ceil(30.9);
floor	Return floor value of a number	a = sin(30.9);
len	Return length of a string	a = len(`hello`);
compare	Return true/false of comparison	a=compare(`ab`,`ab`);
copy	Copy string	a =copy(`ab`);
concat	Concatenate string	a=concat(`ab`,b);

Token:

```
%token headerStart comment purno EOL vogno shobdo eval mod show shuru
sesh IF ELSE concat
%token isEqual isLarge isLargeEqual isSmaller isSmallerEqual
isNotEqual qt compare
%token LOOP INC DEC FUNC copy len FLOOR CEIL SIN COS TAN LOG POW
LOG10 scan GCD
%token <txt> headerName varName
%token <num> number
%token <numd> numberd

%type <numd> expr val numberParameter
%type <txt> changer funcShuru condition loopOP header dataType
stringParameter
```

Union:

```
%union {
    int num;
    char* txt;
    double numd;
}
```

Symbol Table:

```
typedef struct {
    char* name;
    char* type;
    int intValue;
    double doubleValue;
    char* strValue;
} SymbolEntry;
```

Grammar Rules:

```
input:headers program
    | headers functions program
    | cmnt input

functions: function functions
        |function

function: funcShuru '(' parameters ')' '=' isLarge '(' Ret ')' '{'statements'}'
parameters: {printf("No parameters");}
        | onePar
        | onePar ',' parameters

Ret:    {…
onePar: dataType varName    {…
funcShuru: FUNC varName{…
cmnt: comment {printf("This is a comment\n");}


headers: header headers
        | header
        | header cmnt

header: headerStart headerName
{…
program:start statements end

start : shuru
{…
end : sesh
{…
statements : statement statements
        | statement


val: number {$$ = $1*1.0;}
    | numberd {$$ = $1;}
    | qt varName qt
        {…
    | varName
        {…
        }
```

```
statement:
        | scan varName EOL{…
        }
        | cmnt
        | multiVariable
        | variableValueAssign
        | eval expr EOL {printf("Evaluated result is : %f\n",$2);}
        | show value EOL
        | ifshuru '(' condition ')' '{' statements '}' ELSE '{' statements '}' {…
        | LOOP '(' varName loopOP expr EOL changer expr ')' '{' statements '}'  {…
        | varName '(' ')' EOL {…
        | varName '(' callPar ')' EOL {…
        | varName '=' concat '(' stringParameter ',' stringParameter ')' EOL     {…
        | varName '=' copy '(' stringParameter ')' EOL     {…
        | varName '=' compare '(' stringParameter ',' stringParameter ')' EOL     {…
        | varName '=' len '(' stringParameter  ')' EOL     {…
        | varName '=' FLOOR '(' numberParameter ')' EOL {…
        | varName '=' CEIL '(' numberParameter ')' EOL {…
        | varName '=' SIN '(' numberParameter ')' EOL {…
        | varName '=' COS '(' numberParameter ')' EOL {…
        | varName '=' TAN '(' numberParameter ')' EOL {…
        | varName '=' LOG '(' numberParameter ')' EOL {…
        | varName '=' LOG10 '(' numberParameter ')' EOL {…
        | varName '=' POW '(' numberParameter ',' numberParameter')' EOL {…
        | varName '=' GCD '(' numberParameter ',' numberParameter')' EOL {…
```

```
multiVariable: dataType varNames

dataType : purno {isPurno = 1; $$ = "purno";}
         | vogno {isPurno = 0;$$ = "vogno";}
         | shobdo {isPurno = -1;$$ = "shobdo";}

varNames: oneVar ',' varNames
        |oneVar EOL

oneVar: varName
        {…
        }

        | varName '=' number {…
        | varName '=' numberd {…
        | varName '=' qt varName qt {…
        }

variableValueAssign : varName '=' number EOL
                        {…
                    |varName '=' numberd EOL {…
                    |varName '=' expr EOL {…
                    |varName '=' qt varName qt EOL{…
```

```
numberParameter : number {$$ = $1*1.0;}…
stringParameter : qt varName qt {$$ = $2;}…
callPar: oneCall
       | oneCall ',' callPar

oneCall : dataType varName {…
changer: INC {$$ = "inc";}…
loopOP: isLarge {$$ = ">";}
    | isLargeEqual {$$ = ">=";}
    | isSmaller {$$ = "<";}
    | isSmallerEqual {$$ = "<=";}


ifshuru: IF {printf("Started: IF BLOCK\n");}
condition : expr isEqual expr
    {…
    }
    | expr isLarge expr {…
    }
    | expr isLargeEqual expr {…
    }
    | expr isSmaller expr {…
    }
    | expr isSmallerEqual expr {…
    }
    | expr isNotEqual expr {…
    }
    |expr {…
    }
```

Discussion:

In this assignment, I learned flex and bison software. I created my own syntax for my language. I did Lexical Analysis with flex and generated token. I used regular expression to detect token. I passed the token to bison file for parsing. All the syntax and rules for my language parsed using cfg defined with bison file. Although there are some shift/reduce and reduce/reduce conflict , my parser works for my language and give valid error when occurs with line number. I learned the idea how a compiler checks the given input is syntactically matched with the language's syntax or not. It's been a fascinating journey into the world of language design and compiler construction.

Conclusion:

In conclusion, the exploration of flex and bison for the creation of my own language was found to be quite intriguing. Flex was employed for identifying words in the code, while bison was utilized for determining the structure. Despite encountering some complexities with conflicts, mistakes are effectively captured by the code, and their respective locations are communicated. Valuable insights were gained into the process of code sense-checking by compilers. This project contributed to a deeper understanding of language creation, and a continued enthusiasm for further learning in the domain of compiler development has been fostered. The journey proved to be an engaging experience.

Reference: