# ***Compiler Design LAB (CIC-351)***

# ***Practical File***


Faculty Name : Ms. Sakshi Jha

Student Name : Amit Singhal

Roll No : 11614802722 (C-6)

Semester : 5th Semester

Group : 5C6 (CSE Shift-1)




**Maharaja Agrasen Institute of Technology, PSP Area,**

**Sector – 22, Rohini, New Delhi - 110086**

# MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

## VISION

"To attain global excellence through **education, innovation, research,** and **work ethics** with the commitment to **serve humanity."**

## MISSION

**M1.** To promote diversification by adopting advancement in science, technology, management, and allied discipline through continuous learning

**M2.** To foster **moral values** in students and equip them for developing sustainable solutions to serve both national and global needs in society and industry.

**M3.** To **digitize educational resources and process** for enhanced teaching and effective learning.

**M4.** To cultivate an **environment** supporting **incubation, product development, technology transfer, capacity building and entrepreneurship**.

**M5.** To encourage **faculty-student networking with alumni, industry, institutions,** and other **stakeholders** for collective engagement.


## Department of Computer Science and Engineering

## MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

### VISION

"To attain global excellence through education, innovation, research, and work ethics in the field of Computer Science and engineering with the commitment to serve humanity."

### MISSION

**M1.** To lead in the advancement of computer science and engineering through internationally recognized research and education.

**M2.** To prepare students for full and ethical participation in a diverse society and encourage lifelong learning.

**M3.** To foster development of problem solving and communication skills as an integral component of the profession.

**M4.** To impart knowledge, skills and cultivate an environment supporting incubation, product development, technology transfer, capacity building and entrepreneurship in the field of computer science and engineering.

**M5.** To encourage faculty, student's networking with alumni, industry, institutions, and other stakeholders for collective engagement.

# Lab Assessment Sheet

| S.No | Experiment | Date | Marks | | | | | Total Marks | Signature |
|---|---|---|---|---|---|---|---|---|---|
| | | | R1 | R2 | R3 | R4 | R5 | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

# PROGRAM – 1

❖ AIM :: Basic program to practice flex, bison & lex.yy.c

Theory ::

**Flex:**

- **Purpose:** Tool for generating lexical analyzers.
- **Input:** .l files (define lexical rules and patterns).
- **Output:** C source file (lex.yy.c), which can be compiled into an executable.

**Bison:**

- **Purpose:** Parser generator for processing grammar and parsing tokenized input.
- **Works with Flex:** Bison creates a parser that processes tokens generated by the Flex lexer.

**Dev C++:**

- **Purpose:** IDE for compiling and debugging C programs.
- **Usage:** Compiles the lex.yy.c file produced by Flex into an executable (a.out).
- **Features:** Provides an integrated environment for editing, compiling, and debugging generated C code.

**Key Concepts:**

1. **.l File:** The input file for Flex, containing lexical rules and patterns.
2. **lex.yy.c:** C file generated by Flex, containing the lexical analyzer.
3. **a.out:** Default name of the compiled executable generated after compiling lex.yy.c.
4. **Basic Syntax in Lex:**
    - **Definitions Section:** %{ ... %} for including C code.
    - **Rules Section:** Pattern matching rules (regex) and actions.
    - **Main Section:** yylex() to process input, yywrap() to indicate end of input.

Lex Code ::

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

%%
```

```
[0-9]     { printf("Welcome\n"); exit(0); }
.         { printf("Wrong\n"); exit(0); }

%%

int yywrap() { return 1; }

int main() {
    printf("Enter: ");
    yylex();  // Start the lexer
    return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ls
prg1.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ flex prg1.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ls
lex.yy.c  prg1.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ gcc lex.yy.c
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ls
a.out  lex.yy.c  prg1.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ./a.out
Enter: 116
Welcome
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ./a.out
Enter: Amit
Wrong
```

# PROGRAM – 2

❖ AIM :: WAP to check whether a string includes a `keyword` or not.

Theory ::

## Keywords in C Programming

In C programming, **keywords** are predefined reserved words that are integral to the structure and syntax of the language. These words have specific meanings that define the operations and controls within a C program. Because of their predefined significance, keywords cannot be used for naming **variables**, **functions**, **arrays**, or **any other identifiers**. Doing so would conflict with their inherent roles in the language.

C was originally developed at **Bell Labs** by Dennis Ritchie between 1969 and 1973, and the keywords in C form a core part of the language's design, allowing for structured, efficient, and low-level programming. Each keyword is associated with specific functionality such as declaring data types, defining control flow, managing memory, and performing input/output operations.

### Importance of Keywords

C's keywords are essential because they define the rules and structure of the language, guiding how programmers can build applications. They serve as the backbone for:

- **Control Flow**: Keywords like if, else, for, while, switch, and case are used to create conditional statements and loops, enabling developers to control the flow of execution based on various conditions.
- **Data Management**: Keywords such as int, float, char, double, struct, and union allow programmers to define variables and data structures to store and manipulate data efficiently.
- **Memory Allocation**: Keywords like static, auto, extern, register, and volatile play a key role in memory management, controlling how memory is allocated and accessed by the program.
- **Functionality**: Keywords such as return are used to indicate the output or result of a function. The keyword void specifies that a function does not return any value.
- **Error Handling**: Keywords like goto can be used for simple jumps in the program (though generally discouraged in modern C due to readability concerns).

### Types of Keywords

1. **Data Type Keywords**: These define the type of data that can be stored in a variable. Examples include:
    - int: Represents integer data types.
    - float: Represents floating-point numbers.
    - char: Represents character data types.

    - double: Represents double-precision floating-point numbers.
2. **Control Flow Keywords**: These manage the flow of control within a program, allowing for conditional execution and looping. Examples include:
    - if, else: Conditional statements.
    - for, while, do: Looping statements.
    - switch, case: Used in switch-case statements for multi-way branching.
3. **Storage Class Keywords**: These control the scope, lifetime, and visibility of variables and functions. Examples include:
    - static: Limits the scope of a variable to the file or function where it's defined, retaining its value across function calls.
    - extern: Declares a global variable or function that is defined in another file.
    - auto: Defines automatic variables, typically the default storage class for local variables.
    - register: Requests the compiler to store the variable in a CPU register rather than RAM for faster access.
4. **Memory Management Keywords**: These are responsible for handling dynamic memory allocation and freeing memory. Examples include:
    - malloc(), free(): Functions for dynamic memory allocation, although not keywords, are integral to memory management.
5. **Return Keywords**: Keywords like return specify the value that a function should return after execution.
6. **Function Modifiers**: Keywords that modify the behavior of functions and blocks of code, such as inline for making a function inline and const for constant variables.

**Conclusion**

Keywords form the foundation of the C programming language, shaping how developers structure their programs, define data, control the flow of execution, and manage memory. The limited set of 32 keywords ensures simplicity and efficiency, both in terms of program writing and compilation. As reserved words, they embody the core logic and design philosophy of C, ensuring that C remains a powerful, flexible, and low-level language for systems programming, even decades after its creation. Understanding the proper use and function of keywords is essential for mastering C programming and writing efficient, maintainable code.

Lex Code ::

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

  // Array of C keywords
```

```
    char keyword[32][10] = {
        "auto", "double", "int", "struct", "break", "else", "long",
        "switch", "case", "enum", "register", "typedef", "char",
        "extern", "return", "union", "const", "float", "short",
        "unsigned", "continue", "for", "signed", "void", "default",
        "goto", "sizeof", "volatile", "do", "if", "static", "while"
    };

    int is_keyword(char *str) {
        for(int i = 0; i < 32; i++) {
            if(strcmp(str, keyword[i]) == 0)
                return 1;
        }
        return 0;
    }
%}

%%

[a-zA-Z]+ {
    if (is_keyword(yytext)) {
        printf("%s is a keyword\n", yytext);
    } else {
        printf("%s is not a keyword\n", yytext);
    }
}

[0-9]+ {
    printf("%s is a number\n", yytext);
}

\n { return 0; }    // Stops reading after a newline
```

```
.|\n { /* ignore any other characters */ }

%%

int yywrap() {
    return 1;
}

int main() {
    printf("Enter the string: ");
    yylex();
    return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ flex prg_2_checkKeywords.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ gcc lex.yy.c
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ./a.out
Enter the string: if Amit goto 116
if is a keyword
Amit is not a keyword
goto is a keyword
116 is a number
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ./a.out
Enter the string: do while return long double sizeof unsigned int
do is a keyword
while is a keyword
return is a keyword
long is a keyword
double is a keyword
sizeof is a keyword
unsigned is a keyword
int is a keyword
```

# PROGRAM – 3

❖ AIM :: WAP to count no. of `tokens` in a string.

Theory ::

## Tokens in C

In C programming, a **token** is the smallest unit of a program that has a meaning in the language. Tokens are categorized into several types, including:

1. **Keywords**: Reserved words with predefined meanings (e.g., int, return, if).
2. **Identifiers**: Names given to variables, functions, arrays, etc. (e.g., main, count).
3. **Constants**: Literal values like numbers and characters (e.g., 42, 'a').
4. **Operators**: Symbols representing operations (e.g., +, -, *, /).
5. **Punctuation**: Symbols that help structure the code (e.g., ;, ,, {, }).

## Examples of Tokens:

- **Keywords**: if, while, for
- **Identifiers**: variable, sum, total
- **Constants**: 10, 3.14, 'x'
- **Operators**: +, -, =
- **Punctuation**: ;, {, }

Lex Code ::

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

int token_count = 0;
%}

%%
[a-zA-Z_][a-zA-Z0-9_]*     { token_count++; }  // Identifiers
[0-9]+                     { token_count++; }  // Numbers
"+"|"-"|"*"|"/"            { token_count++; }  // Operators
";"|"{"|"}"|"("|")"        { token_count++; }  // Punctuation
```

```
[ \t\n]+                    { /* Ignore whitespace */ }
.                           { /* Ignore other characters */ }
%%

int yywrap() {
    return 1;
}

int main() {
    printf("Enter a string: ");
    yylex();                                        // Start lexing
    printf("\nNumber of tokens: %d\n", token_count);
    return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ flex prg3.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ gcc lex.yy.c
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ./a.out
Enter a string: Amit Singhal 11614802722 Compliler Design LAB
Number of tokens: 6
```

# PROGRAM – 4

❖ AIM :: WAP to check whether a string belong to the grammer or not.

Theory ::

**Grammars:**
A grammar is a formal system that defines the syntax of a language through a set of production rules. These rules determine how strings in the language can be formed from an initial symbol, often called the start symbol. The string generation process involves repeatedly replacing symbols with other symbols or terminal symbols (which represent actual characters in the language).

Grammars are essential in the field of compiler design as they define the syntactic structure of programming languages. They are used in various stages of compilation, such as parsing and syntax analysis, to ensure that the source code adheres to the correct syntax before further processing.

**Types of Grammars:**

Grammars are classified into the following types based on their complexity and generative power:

- **Regular Grammars:**
  Regular grammars define regular languages, which can be recognized by finite automata. These grammars have production rules where the left-hand side contains a single non-terminal, and the right-hand side contains a terminal, possibly followed by a non-terminal. Regular grammars are equivalent to regular expressions.
- **Context-Free Grammars (CFGs):**
  Context-Free Grammars define context-free languages, which can be recognized by pushdown automata. CFGs allow production rules where the left-hand side contains a single non-terminal and the right-hand side contains a string of terminals and/or non-terminals. CFGs are widely used to describe the syntax of programming languages, as they can model nested structures like parentheses or block statements.
- **Context-Sensitive Grammars (CSGs):**
  Context-Sensitive Grammars describe context-sensitive languages, which can be recognized by linear-bounded automata. In CSGs, production rules can have context-sensitive restrictions, meaning that the replacement of a non-terminal may depend on its surrounding symbols.
- **Unrestricted Grammars:**
  Unrestricted grammars describe recursively enumerable languages, which can be recognized by Turing machines. These grammars have the most general form of production rules, allowing for any combination of symbols on both sides of a rule.

**Checking Whether a String Belongs to a Grammar:**

To determine whether a string belongs to a grammar, follow these general steps:

1. **Identify the Production Rules.**
2. **Start with the Initial Symbol.**
3. **Apply Production Rules Recursively.**
4. **Compare with the Input String.**
5. **Recursive Descent Parsing.**

**Example:**

Consider the grammar S → aSb | ab and the string aabb:

1. **Initial Symbol:** Start with S.
2. **Apply Production Rules:**
    - Apply S → aSb, resulting in aSb.
    - Apply S → ab to S in the middle, resulting in aabb.
3. **Comparison:** The derived string aabb matches the input string exactly.
4. **Conclusion:** The string aabb belongs to the grammar.

If the input string were aaabb, no sequence of applying the rules S → aSb | ab would produce this string, so aaabb does not belong to the grammar.

## Code ::

### 1. `gram.l` Lex Program (Lexer)

```
%{
        #include "y.tab.h"
%}

%%
        a       return A;
        b       return B;
        .|\n    return yytext[0];
%%

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

2. `**gram.y**` Yacc Program (Parser)

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

        extern int yylex();
        void yyerror(char *s);
        int valid = 1;
%}

%token A B

%%
    str: S '\n' { printf("Parsed with rule ab^n\n"); return 0; };

    S: A T;
    T: B T | /* empty */;
%%

void yyerror(char *msg) {
   valid = 0;
   fprintf(stderr, "%s\n", msg);
}

int main() {
   printf("Enter the string to check: \n");
   yyparse();
   if (valid) printf("\nValid\n");
   else printf("\nInvalid\n");
   return 0;
}
```

Output ::

**1) Installing `flex` :**

```
[mait@fedora exp4]$ flex gram.l
bash: flex: command not found...
Install package 'flex' to provide command 'flex'? [N/y] y


 * Waiting in queue...
 * Loading list of packages....
The following packages have to be installed:
 flex-2.6.4-12.fc38.x86_64      A tool for generating scanners (text pattern recognizers)
 m4-1.4.19-5.fc38.x86_64        GNU macro processor
Proceed with changes? [N/y] y


 * Waiting in queue...
 * Waiting for authentication...
 * Waiting in queue...
 * Downloading packages...
 * Requesting data...
 * Testing changes...
 * Installing packages...

[mait@fedora exp4]$ flex gram.l
[mait@fedora exp4]$
```

**2) Installing `bison` :**

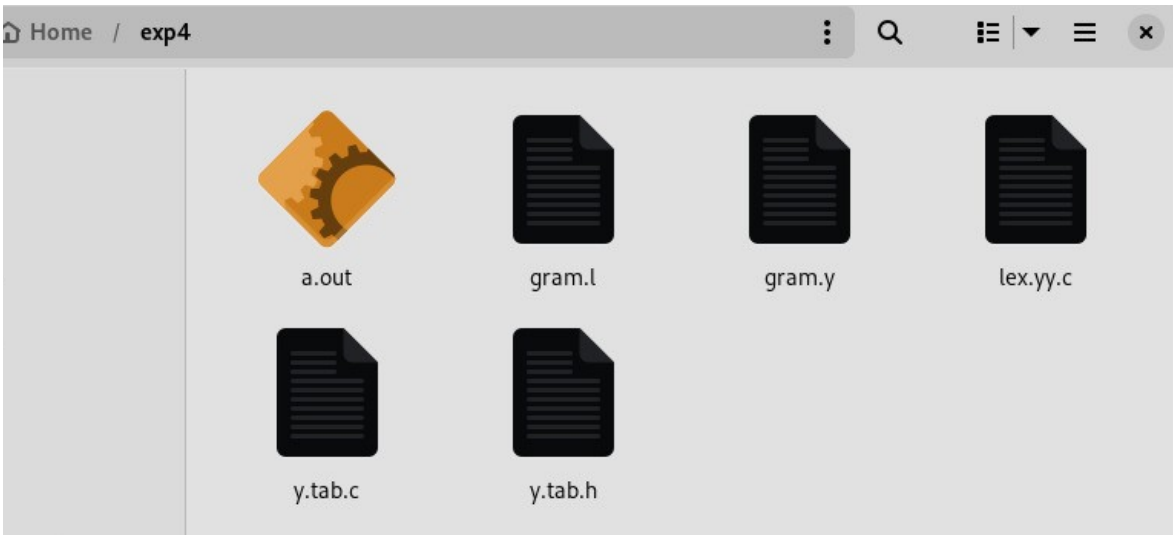
```
[mait@fedora exp4]$ bison gram.y
bash: bison: command not found...
Install package 'bison' to provide command 'bison'? [N/y] y


 * Waiting in queue...
 * Loading list of packages....
The following packages have to be installed:
 bison-3.8.2-4.fc38.x86_64      A GNU general-purpose parser generator
Proceed with changes? [N/y] y


 * Waiting in queue...
 * Waiting for authentication...
 * Waiting in queue...
 * Downloading packages...
 * Requesting data...
 * Testing changes...
 * Installing packages...

[mait@fedora exp4]$ bison gram.y
```

**3) Directory & Files :**




**4) Program Output :**

```
[mait@fedora ~]$ cd exp4/
[mait@fedora exp4]$ flex gram.l
[mait@fedora exp4]$ bison -dy gram.y
[mait@fedora exp4]$ gcc lex.yy.c y.tab.c
[mait@fedora exp4]$ ./a.out
Enter the string to check:
abb
Parsed with rule ab^n

Valid
[mait@fedora exp4]$ ./a.out
Enter the string to check:
amit
syntax error

Invalid
```