Name	Roll No.	Section	DoP
Vidhan Dahatkar	166	B	

Practical 2(a)

Aim- Study the LEX/Flex and YACC/Bison tool and Develop:

A. LEX program to eliminate comment lines (Single and Multiple) in a text (C program) file and copy the resulting program into a separate file.

Theory-

Lexical analysis, also known as scanning or lexing, is the initial phase of the compilation process inwhich the source code is analyzed and broken down into a sequence of tokens. Tokens are the smallest units of meaning in a programming language, such as keywords, identifiers, literals, and operators. The purpose of lexical analysis is to simplify the code by reducing it to a more manageable form for further processing by the compiler or interpreter.

In this task, the objective is to eliminate comment lines from C programs, includingbothsingle-lineand multi-line comments, using a lexical analyzer (LEX). Single-line comments in C start with `//`and continuetothe end of the line, while multi-linecommentsbeginwith`/*`andendwith`*/`.Thegoalisto remove these comments from the source code and generate a clean version of the program.

LEXProgramtoEliminateCommentLines:
The aim of the program is to remove both single-line and multi-line comments from a C program. The LEX program follows a pattern where:

- Single-line comments (those starting with `//`) are removed.
- Multi-line comments (those enclosed between `/*`and `*/`) are also eliminated.

This process is done using regular expressions that match the comment patterns and take appropriate action to ignore them. The clean version of the program is then written to a separate file without any comments.

ProgramOverview:

1. The**LEX program**will processthe inputfile, recognizingthe comment patterns.
2. **Single-line comments** (anything after `//`) will be ignored.
3. **Multi-line comments** (everything between `/*`and `*/`) will also be removed.
4. The**resulting code**will be writtento aseparate file, wherecomments are excluded.

Program-

```
 %{
#inclu
de<std
io.h>
FILE
*outpu
tFile;
%}

%%

[a-zA-Z0-9_{}();,"'\t]+     { fputs(yytext, outputFile); }
\/\[^]\+([^/][^]\+)\/ { / Ignore multi-line comments
*/ }
\/\/[^\n]*                  { /* Ignore single-line comments */ }
\n                          { fputc('\n', outputFile); }

%%

intyywrap(){
   return 1;
}

intmain(intargc,char**argv){ if
   (argc                !=              3)              {
      printf("Usage:%s<input_file><output_file>\n
      ",argv[0]); return 1;
   }

   FILE*inputFile=fopen(argv[1],"r"); if
   (!inputFile)                    {
      perror("Erroropeninginput
      file");
      return 1;
   }

   outputFile=fopen(argv[2],"w"); if
   (!outputFile)                     {
      perror("Erroropeningoutputfi
      le");
      fclose
      (input
```

```
        File);
        return
        1;
    }
    yyin=inputFile; yylex();

    fclose(inputFile);
    fclose(outputFile);

    printf("Outputwrittento%s\n",a
    rgv[2]); return 0;
}
```

Output-

Pract 2a > input2a.c > main()

```c
#include <stdio.h>
// This is a single line comment
int main()
{
    printf("Hello, World!\n"); /*This is a Multi-Line Comment*/
    return 0; // End of the program
}
```

Pract 2a > out.c > main()

```c
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

```
C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 2a>flex pract-2a.l

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 2a>gcc lex.yy.c

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 2a>a.exe input2a.c output2a.c
# <.>      !\       Output written to output2a.c

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 2a>
```

Conclusion–

With FLEX, the program can efficiently identify and remove single-lineandmulti-linecommentsfroma C program. This demonstrates that FLEX is highly capable of handling specific tasks like recognizing comment patterns and processing text in a structured way. These programs show that FLEX is versatile and can be used to simplify and clean up code by focusing on eliminating unnecessary elements, like comments. In the broader context, FLEX proves to be an excellent tool for developing programs that streamline code by targeting particular goals, such as cleaning up comments for easier readability and further processing.

Name	Roll No.	Section	DoP
Vidhan Dahatkar	166	B	

Practical 2(b)

Aim-Study the LEX/Flex and YACC/Bison tool and Develop:

B. YACC program to recognize valid identifier, operators and key words in the given text (C program) file.

Theory-

The process of **syntax analysis** or **parsing** involves checking the structure of the source code to ensure that it conforms to the rules of a programming language. This step typically follows lexical analysis, where the input is broken down into tokens by a tool like **LEX/FLEX**. **YACC (Yet Another Compiler Compiler)**, along with FLEX, is a powerful tool used for building parsers and analyzers based oncontextfree grammars. It helps in the generation of a parser for recognizing the syntactic structure of source code.

In this task, we aim to **develop a YACC program** that will recognize **valid identifiers, operators**, and **keywords** from a given C program. The program will be able to differentiate between various types of tokens and validate whether they match the appropriate patterns for identifiers, operators, or keywords.

Components of the Task:

1. **Valid Identifiers**:
   - An identifier in C is a sequence of letters (uppercase and lowercase), digits, and underscores (_) but cannot start with a digit.
   - **Example**: `int_var`, `main`, `variable123`
2. The YACC program should check for sequences of characters that fit the definition of identifiers in C.
3. **Operators**:
   - Operators are symbols that perform specific operations on variables or values. In C, operatorsincludearithmeticoperators(`+`,`-`,`*`,`/`),relationaloperators(`<`,`>`,`==`),logical operators (`&&`, `||`), assignment operators (`=`, `+=`, `-=`), and many others.
4. The YACC program will be able to recognize these operators as part of the input text.
5. **Keywords**:
   - Keywords are reserved words in a programming language that have special meaning, and they cannot be used as identifiers. In C, keywords include words like `int`, `float`, `while`, `if`, `return`, and many others.
6. The program should recognize these predefined words and treat them as distinct from user-defined identifiers.

Working with YACC:

YACC is used for syntax analysis, where grammar rules are defined to describe the valid structure of a language. The primary role of YACC in this case is to define rules for **recognizing valid identifiers**, **operators**, and **keywords** within the input text. YACC uses a set of **production rules** (grammar) and **actions** to process the input.

General Structure of a YACC Program:

A YACC program is divided into three main sections:

1. **Declarations** : This section includes definitions of token types, including those for identifiers, operators, and keywords.
2. **Grammar Rules** : This section defines the syntax rules for valid identifiers, operators, and keywords, such as the structure of a simple C expression or statement.
3. **C Code** : This section includes additional code that can handle specific actions to be taken when certain patterns are matched by the grammar.

Explanation of the Example:

1. **Tokens**:
   - We define three types of tokens : `IDENTIFIER`, `KEYWORD`, and `OPERATOR`. Each token represents different elements of the source code.
   - These tokens will be identified using regular expressions defined in the **LEX program**.
2. **Grammar Rules**:
   - The grammar rules define the structure of the program. For example, a declaration consists of a `KEYWORD` followed by an `IDENTIFIER`, and an assignment consists of an `IDENTIFIER`followed by an `=`and an `expression`.
   - The grammar is flexible and can be expanded to handle more complex statements.
3. **Actions**:
   - The actions (`{}`) are executed when a grammar rule is matched. For example, when a `KEYWORD IDENTIFIER`pair is matched, the program prints out the keyword and identifier.

YACCandFLEXIntegration:

In practice, **LEX** and **YACC** are used together to process source code. FLEX handles the lexical analysis, breaking down the input into tokens like `IDENTIFIER`, `KEYWORD`, and `OPERATOR`, which are then processed by YACC to ensure they follow the syntax of a valid C program.

Program-

LEX file:

```
%{
#include<stdio.h>
#include "y.tab.h"
%}

%option noyywrap

DIGIT     [0-9]
LETTER  [a-zA-Z_]
IDENTIFIER {LETTER}({LETTER}|{DIGIT})*

%%
"+"|"-"|""|"/"|"%"|"="|"+="|"-="|"="|"/="|"%="|"=="|"!="|"<"|">"|"&&"|"||"|"!" {
printf("Operator: %s\n",yytext);returnOPERATOR;}

"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"ext
ern"|"float"|"f
or"|"goto"|"if"|"int"|"long"|"register"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"sw
itch"|"typede f"|"union"|"unsigned"|"void"|"volatile"|"while" { printf("Keyword: %s\n",
yytext); return KEYWORD; }

{IDENTIFIER} {printf("Identifier:%s\n",yytext);returnIDENTIFIER;} [

\t\n];

. ;

%%

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

YACC file:

```
%{
#include <stdio.h>
voidyyerror(constch
ar*s); int yylex();
%}

%token IDENTIFIER KEYWORD OPERATOR

%%
```

```
program:
    program statement
    | statement
    ;

statement:
    IDENTIFIER { printf("Parsed Identifier: %s\n", yytext); }
    | KEYWORD { printf("Parsed Keyword: %s\n", yytext); }
    |OPERATOR{printf("ParsedOperator:%s\n",yytext);}
    ;

%%

void yyerror(const
char *s) {
fprintf(stderr,"Error
:%s\n",s); }

int main() {
    printf("Parsingsta
    rted...\n");
    yyparse();
    printf("Parsingcomplet
    ed.\n"); return 0;
}
```

Output-

```
C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 2b>bison -d -y parser.y

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 2b>flex practical-2b.l

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 2b>gcc parser.tab.c lex.yy.c

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 2b>a.exe
Parsing started...
+
Operator: +
Parsed Operator: (null)
return
Keyword: return
Parsed Keyword: (null)
thanks
Identifier: thanks
Parsed Identifier: (null)
Parsing completed.
```

Conclusion–

This YACC program will allow you to analyze and validate a C program by recognizing identifiers, operators, and keywords. YACC is powerful for developing parsers that can process the syntax of programming languages, making it an essential tool in building compilers and interpreters. By integrating YACC with FLEX, we can perform both lexical and syntax analysis to identify and process key language constructs in a C program.