Name	Roll No.	Section	DoP
Vidhan Dahatkar	166	B	

Practical 3(a)

Aim -Develop a LEX program to recognize valid arithmetic expression. Identifiers in the expression could be only integers and operators could be + and *. Count the identifiers & operators present and print them separately.

Theory -

In the process of **lexical analysis**, the source code is divided into smaller meaningful components, or **tokens**, which are the building blocks of a program. The **LEX** tool is used to create a **lexical analyzer** (also known as a lexer or scanner) that identifies these tokens based on patterns defined by **regular expressions**. The goal of this task is to develop a LEX program that can recognize and analyze valid **arithmetic expressions**, count the **identifiers** (in this case, integers) and **operators** (+ and *), and print them separately.

Key Components of the Task:

1. **Identifiers (Integers):**
   - In this context, identifiers refer to integer values that can be part of an arithmetic expression.
   - **Pattern for Identifiers:** The regular expression to match integers is a sequence of one or more digits, e.g., 123, 4567, etc.
2. **Operators:**
   - The operators in the given arithmetic expression are restricted to the **addition (+)** and **multiplication (*)** operators.
   - **Pattern for Operators:** The operators + and * are specific symbols that will be recognized in the input expression.
3. **Lexical Analysis:**
   - The task of the **LEX program** is to read an arithmetic expression, tokenize the expression, and identify the **integers** and **operators**.
   - Each recognized token (integer or operator) will be counted and printed separately to allow an understanding of how many identifiers and operators are present in the expression.
4. **Regular Expressions in LEX:**
   - LEX uses **regular expressions** to define patterns for recognizing specific types of tokens.
     In this case, we define:
     - A **pattern for integers**: This could be a regular expression like [0-9]+, which matches one or more digits.
     - A **pattern for operators**: The regular expressions would be \+ for addition and * for multiplication.

- The LEX program will then associate actions with these patterns, such as counting the tokens and printing them.

5. **Actions:**
   - The **actions** defined in the LEX program will specify what to do when a particular pattern is matched. For example:
     - When an integer is matched, it will be counted and printed.
     - When an operator is matched, it will be counted and printed.

6. **Counting Identifiers and Operators:**
   - The program will maintain counters for the number of identifiers (integers) and operators found in the expression. The final output will display the count of each type of token, along with the tokens themselves.

How the LEX Program Works:

1. **Define Regular Expressions:**
   - In the LEX program, the regular expressions for integers and operators are defined in the **definition section**. For example:
     - Integer:
     [0-9]+ ■
     Operators:
     \+ and *
2. **Define Rules:**
   - The **rules section** will map the regular expressions to corresponding actions. When an integer is matched, it will increment the integer counter, and when an operator is matched, it will increment the operator counter.
3. **Print and Count Tokens:**
   - The program will print out each identifier (integer) and operator as it encounters them.
     Additionally, it will print the total count of identifiers and operators at the end.
   - The program ignores any other characters that don't match the defined patterns.

2. **Actions:**
   - Each time an integer or operator is matched, the program prints it and updates the count. At the end, the program prints the total counts of identifiers and operators.

Program -

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

int identifier_count = 0;  // To count identifiers (integers)
int operator_count = 0;    // To count operators (+ and *)
%}

%%
```

```
[0-9]+          { identifier_count++; }  // Match integers (Identifiers)
[+*]            { operator_count++; }    // Match operators + and *
[ \t\n]         { /* Ignore spaces, tabs, and newlines */ }
.               { /* Ignore other characters */ }
%%

int main() {
   char input[100];  // To hold the input expression
   printf("Enter arithmetic expression:\n");

   // Use fgets to read the input
   fgets(input, sizeof(input), stdin);

   // Set the input to yylex for lexical analysis
   YY_BUFFER_STATE buffer = yy_scan_string(input);
   yylex();  // Perform lexical analysis (start scanning input)

   printf("Identifiers count: %d\n", identifier_count);
   printf("Operators count: %d\n", operator_count);

   return 0;
}

int yywrap() {
   return 1;  // Return 1 to indicate the end of input
}
```

Output -

```
C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 3a>flex pract-3a.l

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 3a>gcc lex.yy.c

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 3a>a.exe
Enter arithmetic expression:
12-5+7*3-13*4
Identifiers count: 6
Operators count: 3

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 3a>
```

Conclusion –

This LEX program demonstrates the power of lexical analysis to break down a simple arithmetic expression into its constituent components—**identifiers (integers)** and **operators** (+, *). By using regular expressions and defining corresponding actions, we can efficiently recognize and count the different tokens in the expression. The ability to

count and print tokens separately allows us to gain insights into the structure of the expression.

Name	Roll No.	Section	DoP
Vidhan Dahatkar	166	B	

Practical 3(b)

Aim –
Develop a YACC program to evaluate arithmetic expression involving operators: +, -, *, and /.

Theory -

A YACC program for evaluating arithmetic expressions takes an input string containing numbers and operators (+, -, *, /) and processes it to compute the result. The process involves two main steps: lexical analysis and syntax analysis. **Lexical analysis** is the first step, where the input string is broken down into tokens (like numbers and operators) using regular expressions. These tokens are passed to the **syntax analyzer** (the YACC parser), which uses a context-free grammar to verify if the input follows a valid structure (e.g., numbers can be combined with operators to form expressions).

The syntax is defined by rules (grammar), which specify how tokens should be grouped together. Each rule is associated with a **semantic action** that is executed when the rule is applied. For example, if an expression is an addition (expr : expr '+' term), the semantic action adds the two operands together. The semantic actions evaluate the expression as the parsing process occurs, so the final result is computed as the program parses the entire input.

YACC also incorporates **operator precedence** and **associativity** rules, which ensure that operators like multiplication and division are evaluated before addition and subtraction. This is typically done using %left or %right declarations in YACC, which specify how operators of the same precedence should be grouped. For example, * and / have higher precedence than + and -, and they are left-associative. The program also includes **error handling** to catch syntax mistakes, such as mismatched parentheses or invalid operators.

Key Points:

- **Lexical Analysis**: Breaks input into tokens such as numbers, operators, and parentheses.
- **Syntax Analysis (Parsing)**: Uses grammar rules to verify if the expression is valid.
- **Semantic Actions**: Performs calculations (e.g., adds or multiplies values) during parsing.
- **Operator Precedence**: Ensures correct order of operations (e.g., multiplication before addition).
- **Associativity**: Determines how operators of the same precedence are evaluated (left or right).
- **Error Handling**: Detects invalid expressions and reports errors.

Program –

lexer.l

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

%%
[0-9]+      { yylval.num = atoi(yytext); return NUM; }
"+"         { return PLUS; }
"-"         { return MINUS; }
"*"         { return MUL; }
"/"         { return DIV; }
[ \t\n]     { /* ignore whitespace */ }
.           { return yytext[0]; }
%%

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

parser.y

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

extern int yylex();  // Lexical analyzer function
int yyparse();  // Parser function

void yyerror(const char *s);  // Error handling function

%}

%union {
   int num;  // Store the value of the numbers
}

%token <num> NUM
%token PLUS MINUS MUL DIV

%left PLUS MINUS
%left MUL DIV

%type <num> program expression
```

```
%%

program:
    expression { printf("Result: %d\n", $1); }
;

expression:
    NUM                         { $$ = $1; }
    | expression PLUS expression  { $$ = $1 + $3; }
    | expression MINUS expression { $$ = $1 - $3; }
    | expression MUL expression   { $$ = $1 * $3; }
    | expression DIV expression   {
                                  if ($3 == 0) {
                                      printf("Error: Division by zero\n");
                                      exit(1);
                                  }
                                  $$ = $1 / $3;
                                 }
    | '(' expression ')'          { $$ = $2; }  // Handle parentheses
    ;

%%

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

int main() {
    printf("Enter an arithmetic expression: ");
    yyparse();
    return 0;
}
```

Output -

```
C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 3b>bison -d -y parser.y

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 3b>flex lexer.l

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 3b>gcc parser.tab.c lex.yy.c

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 3b>calculator.exe
Enter an arithmetic expression: ((21*(17-4)+19)/10)/5*3
Result: 15

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 3b>
```

Conclusion –

In the practical implementation of a YACC program to evaluate arithmetic expressions, we demonstrated how to efficiently parse and compute mathematical operations. By using YACC for syntax analysis and semantic actions, the program ensures correct operator precedence and associativity. With lexical analysis and a clear grammar, it handles various valid expressions and performs accurate calculations. Error handling is also included to detect invalid inputs, showcasing the power of YACC in building expression evaluators.