1]0.33\*12-4-4+(3\*2)

1]Install-🡪sudo apt update

sudo apt install flex bison

2]Save ycc file--🡪arith.y

3]Yacc code-🡪

/\* YACC Program to evaluate arithmetic expressions with precedence and associativity \*/

%{

#include <stdio.h> /\* Standard Input/Output Library \*/

#include <stdlib.h> /\* Standard Library for exit() and atof() \*/

%}

%token NUMBER /\* Define token for numeric values \*/

/\* Operator precedence and associativity \*/

%left '-' '+' /\* Addition and Subtraction have equal precedence \*/

%left '\*' '/' /\* Multiplication and Division have higher precedence \*/

%%

/\* Grammar rules and semantic actions \*/

expression:

expression '+' expression { /\* Addition \*/

$$ = $1 + $3;

printf("Result: %f\n", $$);

}

| expression '-' expression { /\* Subtraction \*/

$$ = $1 - $3;

printf("Result: %f\n", $$);

}

| expression '\*' expression { /\* Multiplication \*/

$$ = $1 \* $3;

printf("Result: %f\n", $$);

}

| expression '/' expression { /\* Division with error handling \*/

if ($3 == 0) {

printf("Error: Division by zero\n");

exit(1);

}

$$ = $1 / $3;

printf("Result: %f\n", $$);

}

| '(' expression ')' { /\* Parenthesized expression \*/

$$ = $2;

}

| NUMBER { /\* Numeric value \*/

$$ = atof(yytext);

}

;

%%

/\* Include the Lex file \*/

#include "lex.yy.c"

/\* Main function to start parsing \*/

int main() {

printf("Enter an arithmetic expression:\n");

yyparse(); /\* Call the parser \*/

return 0;

}

/\* Error handling function \*/

int yyerror(char \*s) {

fprintf(stderr, "Error: %s\n", s);

return 0;

}

4] save lex program --🡪arith.l

/\* Lexical Analyzer for Arithmetic Expressions \*/

%{

#include "y.tab.h" /\* Include the header file generated by YACC \*/

%}

%%

/\* Match numeric values, including integers and floating-point numbers \*/

[0-9]+(\.[0-9]+)? {

yylval = atof(yytext); /\* Convert matched text to a floating-point number \*/

return NUMBER; /\* Return the NUMBER token to YACC \*/

}

/\* Ignore whitespace characters (spaces, tabs, and newlines) \*/

[ \t\n]+ { /\* No action, just ignore \*/ }

/\* Match any single character and return it to YACC for further processing \*/

. { return yytext[0]; }

%%

5] Compile the Lex and YACC Programs:

1) Generate the YACC source file (y.tab.c and y.tab.h--🡪yacc -d arith.y[-d generates the header file y.tab.h.]

2) Generate the Lex source file-🡪flex arith.l

3)Compile the generated C files🡪gcc y.tab.c lex.yy.c -o arith -lm[-lm is used to link the math library.]

6] Run the Program:

---🡪./arith

Now, enter the input expression---🡪0.33\*12-4-4+(3\*2)

2] upper to lower and vice versa

Lex File (lexer.l)

%{

#include "y.tab.h" // Include the header file generated by YACC. This contains definitions of the tokens used by YACC.

%}

%%

/\*

\* This section defines the rules for tokenizing input characters.

\* Each rule tells the lexer how to identify and classify characters

\* from the input stream.

\*/

// Rule to identify lowercase letters (a-z)

[a-z] {

// When a lowercase letter is found, return the token 'LOWERCASE' to YACC.

return LOWERCASE;

}

// Rule to identify uppercase letters (A-Z)

[A-Z] {

// When an uppercase letter is found, return the token 'UPPERCASE' to YACC.

return UPPERCASE;

}

// Rule to handle the newline character '\n' (end of a line)

\n {

// When a newline is encountered, return 0 (this signals the end of input to YACC).

return 0;

}

// Rule to ignore any other characters (such as spaces, punctuation, etc.)

. {

// Ignore any non-alphabet characters by returning 0 (no action for other characters).

return 0;

}

%%

/\*

\* This function is required by Lex. It is called when the lexer

\* reaches the end of the input. Returning 1 indicates the end

\* of the input stream.

\*/

int yywrap() {

return 1; // Indicating that there's no more input to process.

}

2]YACC FILE(parser.y)

%{

#include <stdio.h> // Standard I/O functions

#include <ctype.h> // Provides functions like toupper() and tolower() for character case conversion

// Define tokens that will be returned by the Lex scanner

#define LOWERCASE 1

#define UPPERCASE 2

%}

// Declare the tokens to YACC

%token LOWERCASE

%token UPPERCASE

%%

/\*

\* Grammar Rules Section

\* This section defines how sequences of tokens are interpreted.

\*/

// 'input' is the start symbol. It allows multiple lines of input.

input:

/\* empty \*/ // Allow empty input

| input line // Process multiple lines by recursively calling 'input'

;

// Each 'line' consists of either a LOWERCASE or an UPPERCASE character

line:

LOWERCASE {

// Convert lowercase character to uppercase and print

printf("%c", toupper(yytext[0]));

}

| UPPERCASE {

// Convert uppercase character to lowercase and print

printf("%c", tolower(yytext[0]));

}

;

%%

/\*

\* C Code Section

\* This section contains the main function and error handling.

\*/

int main() {

printf("Enter a string: ");

yyparse(); // Begin parsing input using the rules defined above

return 0;

}

// Error handling function called by YACC on parse errors

int yyerror(char \*s) {

printf("Error: %s

", s);

return 0;

}

(A)11.Write a program to evaluate a given variable name using YACC specification. SAMPLE INPUT 1) pune 2) PUNE 3) Pune1 4) pUNE\_2

**Lex File (lex.yy.c)**

This Lex program will scan the input and check if the given variable name is valid according to C/C++ rules (starting with a letter or underscore, followed by letters, digits, or underscores).

/\* Lexical Analyzer for Variable Names \*/

%{

#include "y.tab.h" /\* Include the header file generated by YACC \*/

%}

%%

/\* Identifier rule: Start with a letter or underscore, followed by letters, digits, or underscores \*/

[a-zA-Z\_][a-zA-Z0-9\_]\* {

return IDENTIFIER; /\* Return the IDENTIFIER token to YACC \*/

}

/\* Ignore whitespace characters (spaces, tabs, and newlines) \*/

[ \t\n]+ { /\* No action, just ignore \*/ }

/\* Handle any invalid characters (any character that doesn't match the above rules) \*/

. { return yytext[0]; }

%%

int main() {

yylex(); /\* Start Lex analysis \*/

return 0;

}

**YACC File (yacc.y)**

This YACC program will evaluate the variable name and print whether it's valid or invalid

/\* YACC Program to evaluate variable names based on C/C++ identifier rules \*/

%{

#include <stdio.h> /\* Standard Input/Output library \*/

#include <stdlib.h> /\* Standard Library for exit() \*/

%}

%token IDENTIFIER

%%

/\* Grammar rule for a valid identifier \*/

identifier:

IDENTIFIER {

printf("Valid identifier: %s\n", yytext); /\* Print if valid \*/

}

| error {

printf("Invalid identifier: %s\n", yytext); /\* Print if invalid \*/

}

;

%%

#include "lex.yy.c" /\* Include the Lex file generated by Lex \*/

/\* Main function to parse the input \*/

int main() {

printf("Enter a variable name to evaluate:\n");

yyparse(); /\* Call the YACC parser \*/

return 0;

}

/\* Error handling function \*/

int yyerror(char \*s) {

fprintf(stderr, "%s\n", s); /\* Print any errors \*/

return 0;

}

C)11.Write a program to evaluate a given built-in functions using YACC specification. INPUT 1.u= sqrt(36) 2. v = strlen(“pune”)

**Lex File (lex.yy.c)**

This Lex program scans for function calls such as sqrt() and strlen() and handles identifiers, function names, numbers, and strings.

/\* Lexical Analyzer for Built-in Functions (sqrt, strlen) \*/

%{

#include "y.tab.h" /\* Include the header file generated by YACC \*/

#include <math.h> /\* For sqrt() function \*/

#include <string.h> /\* For strlen() function \*/

#include <stdio.h>

%}

%%

/\* Match identifiers (variable names, function names) \*/

[a-zA-Z\_][a-zA-Z0-9\_]\* {

yylval = strdup(yytext); /\* Store function name or variable as a string \*/

return IDENTIFIER; /\* Return IDENTIFIER token to YACC \*/

}

/\* Match numbers \*/

[0-9]+(\.[0-9]+)? {

yylval = atof(yytext); /\* Convert the matched number to a floating-point value \*/

return NUMBER; /\* Return the NUMBER token to YACC \*/

}

/\* Match strings (for the second input, e.g., "pune") \*/

\"[^\"]\*\" {

yylval = strdup(yytext); /\* Store the string \*/

return STRING; /\* Return the STRING token to YACC \*/

}

/\* Ignore whitespace characters (spaces, tabs, and newlines) \*/

[ \t\n]+ { /\* No action, just ignore \*/ }

/\* Handle any invalid characters \*/

. { return yytext[0]; }

%%

**YACC File (yacc.y)**

This YACC program defines the grammar rules for handling built-in functions like sqrt() and strlen() and evaluates their results.

/\* YACC Program to evaluate built-in functions like sqrt() and strlen() \*/

%{

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

#include <ctype.h>

#include "lex.yy.c" /\* Include the Lex file generated by Lex \*/

%}

%token IDENTIFIER NUMBER STRING

%%

/\* Grammar rules for evaluating built-in functions \*/

statement:

IDENTIFIER '=' function\_call { /\* Handle assignment to variables \*/

printf("%s = %f\n", $1, $3); /\* Print the result of the function call \*/

}

;

function\_call:

IDENTIFIER '(' expression ')' { /\* Match function calls like sqrt(x), strlen("str") \*/

if (strcmp($1, "sqrt") == 0) { /\* Handle the sqrt function \*/

$$ = sqrt($3); /\* Call sqrt function \*/

} else if (strcmp($1, "strlen") == 0) { /\* Handle the strlen function \*/

$$ = strlen($3); /\* Call strlen function \*/

} else {

printf("Unknown function: %s\n", $1); /\* Handle unknown functions \*/

$$ = 0;

}

}

;

expression:

NUMBER { $$ = $1; } /\* Return the number \*/

| STRING { $$ = strlen($1); } /\* Return the length of the string \*/

;

%%

/\* Main function to start parsing \*/

int main() {

printf("Enter an expression:\n");

yyparse(); /\* Call the YACC parser \*/

return 0;

}

/\* Error handling function \*/

int yyerror(char \*s) {

fprintf(stderr, "Error: %s\n", s); /\* Print errors \*/

return 0;

}

(D)11.Write a program to evaluate a given built-in functions using YACC specification. INPUT u= sin(12)+cos(12)

YACC File(math\_eval.y)

%{

#include <stdio.h> // Standard I/O library for printing results

#include <stdlib.h> // For general utilities like atoi (converting strings to integers)

#include <math.h> // For mathematical functions like sin(), cos(), etc.

// Define the token types for the parser

#define NUM 1 // Token for numbers (e.g., 12)

#define VAR 2 // Token for variables (e.g., u)

#define ASSIGN 3 // Token for assignment operator '='

#define FUNC 4 // Token for function (e.g., sin, cos)

// Variable to store the result of the evaluation

double result; // Will hold the result of the computation

%}

// Declare the tokens used by the lexer

%token NUM VAR ASSIGN FUNC

// Declare the types of the values to be passed between lexer and parser

%union {

double num; // Holds a numeric value

char var; // Holds a single character for a variable (like 'u')

}

%%

// Start symbol: This represents the assignment of a value to a variable.

program:

VAR ASSIGN expression {

// Store the result of the expression in the variable 'result'

result = $3;

printf("%s = %lf\n", yytext, result); // Print the variable name and its computed value

};

// Expression can be a number, a function, or an operation involving numbers.

expression:

NUM {

// A number is directly returned

$$ = $1;

}

| FUNC '(' expression ')' {

// Function (e.g., sin or cos) applied to an expression

if (strcmp(yytext, "sin") == 0)

$$ = sin($3); // Evaluate sine

else if (strcmp(yytext, "cos") == 0)

$$ = cos($3); // Evaluate cosine

else

printf("Unsupported function: %s\n", yytext);

}

| expression '+' expression {

// Addition: add two expressions

$$ = $1 + $3;

}

| expression '-' expression {

// Subtraction: subtract the second expression from the first

$$ = $1 - $3;

}

| expression '\*' expression {

// Multiplication: multiply two expressions

$$ = $1 \* $3;

}

| expression '/' expression {

// Division: divide the first expression by the second

if ($3 == 0) {

printf("Error: Division by zero\n");

$$ = 0; // Handle division by zero

} else {

$$ = $1 / $3;

}

}

| '(' expression ')' {

// Parentheses: evaluate the expression inside parentheses

$$ = $2;

}

;

// Lexical rules

%%

// Error handling function in case of syntax errors

int yyerror(const char \*s) {

printf("Error: %s\n", s);

return 0;

}

// Main driver for the parser

int main() {

printf("Enter an expression (e.g., u = sin(12) + cos(12)): ");

yyparse(); // Start parsing the input

return 0;

}

Lexical Analysis(lexer.l)

%{

#include "y.tab.h"

%}

%%

[0-9]+(\.[0-9]+)? { yylval.num = atof(yytext); return NUM; } // Matches numbers (e.g., 12 or 3.14)

[a-zA-Z] { yylval.var = yytext[0]; return VAR; } // Matches variable names (e.g., 'u')

"=" { return ASSIGN; } // Matches the assignment '='

"sin" { return FUNC; } // Matches 'sin'

"cos" { return FUNC; } // Matches 'cos'

[ \t\n] { /\* Ignore spaces, tabs, and newlines \*/ }

%%

int yywrap() {

return 1;

}

E

)11.Write a program to evaluate a given built-in functions using YACC specification. INPUT p= pow(3,2) / log (24)

Lex File (lex.yy.c)

/\* Lexical Analyzer for Built-in Functions (pow, log) \*/

%{

#include "y.tab.h" /\* Include the header file generated by YACC \*/

#include <math.h> /\* For pow() and log() functions \*/

#include <stdio.h>

%}

%%

/\* Match function names like pow, log \*/

[a-zA-Z\_][a-zA-Z0-9\_]\* {

yylval = strdup(yytext); /\* Store function name as a string \*/

return IDENTIFIER; /\* Return IDENTIFIER token to YACC \*/

}

/\* Match numbers \*/

[0-9]+(\.[0-9]+)? {

yylval = atof(yytext); /\* Convert the matched number to a floating-point value \*/

return NUMBER; /\* Return the NUMBER token to YACC \*/

}

/\* Match operators like +, -, \*, / \*/

[\+\-\\*/] {

return yytext[0]; /\* Return the operator character to YACC \*/

}

/\* Match parentheses \*/

\(|\) {

return yytext[0]; /\* Return parentheses to YACC \*/

}

/\* Ignore whitespace characters (spaces, tabs, and newlines) \*/

[ \t\n]+ { /\* No action, just ignore \*/ }

/\* Handle any invalid characters \*/

. { return yytext[0]; }

%%

YACC File (yacc.y)

/\* YACC Program to evaluate built-in functions like pow() and log() \*/

%{

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

#include <ctype.h>

#include "lex.yy.c" /\* Include the Lex file generated by Lex \*/

%}

%token IDENTIFIER NUMBER

%%

/\* Grammar rules for evaluating expressions involving built-in functions \*/

/\* Statement rule: handle assignments (e.g., p = pow(3, 2) / log(24)) \*/

statement:

IDENTIFIER '=' expression { /\* Assign expression result to the identifier \*/

printf("%s = %f\n", $1, $3); /\* Print the result of the expression \*/

}

;

/\* Expression rule: handle function calls, numbers, and operations \*/

expression:

expression '+' term { $$ = $1 + $3; } /\* Addition \*/

| expression '-' term { $$ = $1 - $3; } /\* Subtraction \*/

| term { $$ = $1; } /\* Single term (function call or number) \*/

;

/\* Term rule: handle multiplication, division \*/

term:

term '\*' factor { $$ = $1 \* $3; } /\* Multiplication \*/

| term '/' factor { $$ = $1 / $3; } /\* Division \*/

| factor { $$ = $1; } /\* Single factor (function call or number) \*/

;

/\* Factor rule: handle numbers, function calls, and parentheses \*/

factor:

NUMBER { $$ = $1; } /\* A number \*/

| IDENTIFIER '(' expression ',' expression ')' { /\* Function call like pow(x, y) \*/

if (strcmp($1, "pow") == 0) { /\* Check for pow function \*/

$$ = pow($3, $5); /\* Call pow(x, y) \*/

} else if (strcmp($1, "log") == 0) { /\* Check for log function \*/

$$ = log($3); /\* Call log(x) \*/

} else {

printf("Unknown function: %s\n", $1); /\* Handle unknown functions \*/

$$ = 0;

}

}

| '(' expression ')' { $$ = $2; } /\* Parenthesized expression \*/

;

%%

/\* Main function to start parsing \*/

int main() {

printf("Enter an expression:\n");

yyparse(); /\* Call the YACC parser \*/

return 0;

}

/\* Error handling function \*/

int yyerror(char \*s) {

fprintf(stderr, "Error: %s\n", s); /\* Print errors \*/

return 0;

}