**11. Generate a valid pattern that recognizes all statements that begins with an Upper-case letter**

**followed by five digits or alphabets. Use a YACC tool to do the same.**

**LEX FILE**

%{

#include "y.tab.h"

%}

%%

[A-Z][A-Za-z0-9]{5} { return IDENTIFIER; }

.|\n { /\* Ignore other characters \*/ }

%%

**YACC file**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

extern char \*yytext;

%}

%token IDENTIFIER

%%

input:

| input line

;

line: IDENTIFIER '\n' { printf("Valid input: %s\n", yytext); }

| error '\n' { yyerror("Invalid input"); }

;

%%

int yyerror(char \*s) {

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

return 0;

}

int main() {

printf("Enter a statement: ");

yyparse();

return 0;

}

**OUTPUT**:

Enter a statement: A12345

Valid input: A12345

----------------------------------------------------------------------------------------------------------

**12. Design a lexical analyzer that identifies comments, operators and keywords from a given expression**

**LEX file**

%{

#include <stdio.h>

#include <string.h>

#define MAX\_KEYWORD\_LEN 16

char \*keywords[] = {"if", "else", "while", "for", "return", "int", "float"};

%}

%%

"if"|"else"|"while"|"for"|"return"|"int"|"float" { printf("KEYWORD: %s\n", yytext); }

[+\-\*/%] { printf("OPERATOR: %s\n", yytext); }

"&&"|"||" { printf("LOGICAL OPERATOR: %s\n", yytext); }

"=" { printf("ASSIGNMENT OPERATOR: %s\n", yytext); }

"//".\*\n { /\* Single-line comment - Ignore \*/ }

"/\*"[^\*]\*"\*"+([^/\*][^\*]\*"\*"+)\*"/" { /\* Multi-line comment - Ignore \*/ }

[A-Za-z\_][A-Za-z0-9\_]\* { printf("IDENTIFIER: %s\n", yytext); }

[0-9]+ { printf("INTEGER LITERAL: %s\n", yytext); }

[0-9]\*"."[0-9]+ { printf("FLOAT LITERAL: %s\n", yytext); }

[ \t\n\r]+ { /\* Ignore whitespace \*/ }

. { printf("UNKNOWN: %s\n", yytext); }

%%

int main() {

printf("Enter an expression: ");

yylex();

return 0;

}

int yyerror(char \*s) {

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

return 0;

}

**INPUT:**

int main() {

float x = 3.14;

// This is a comment

x = x + 1;

}

**OUTPUT**:

KEYWORD: int

IDENTIFIER: main

OPERATOR: (

OPERATOR: )

OPERATOR: {

KEYWORD: float

IDENTIFIER: x

ASSIGNMENT OPERATOR: =

FLOAT LITERAL: 3.14

OPERATOR: ;

COMMENT: // This is a comment

IDENTIFIER: x

ASSIGNMENT OPERATOR: =

IDENTIFIER: x

OPERATOR: +

INTEGER LITERAL: 1

OPERATOR: ;

OPERATOR: }

------------------------------------------------------------------------------------

**13. Develop a program to recognize a valid control structure syntax of c language(for loop, while loop,**

**if-else, if-else-if,switch case,etc...)**

**Lex file**

%{

#include "y.tab.h"

%}

%%

"if" { return IF; }

"else" { return ELSE; }

"while" { return WHILE; }

"for" { return FOR; }

"switch" { return SWITCH; }

"case" { return CASE; }

"break" { return BREAK; }

"return" { return RETURN; }

"{" { return LBRACE; }

"}" { return RBRACE; }

"(" { return LPAREN; }

")" { return RPAREN; }

";" { return SEMICOLON; }

"," { return COMMA; }

"==" { return EQ; }

"!=" { return NEQ; }

"<" { return LT; }

"<=" { return LTE; }

">" { return GT; }

">=" { return GTE; }

"=" { return ASSIGN; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MUL; }

"/" { return DIV; }

[0-9]+ { return NUMBER; }

[A-Za-z\_][A-Za-z0-9\_]\* { return IDENTIFIER; }

[ \t\n\r]+ { /\* Skip whitespace \*/ }

. { return UNKNOWN; }

%%

**YACC file**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

extern int yylex();

void yyerror(const char \*s);

%}

%token IF ELSE WHILE FOR SWITCH CASE BREAK RETURN

%token LPAREN RPAREN LBRACE RBRACE SEMICOLON COMMA

%token EQ NEQ LT LTE GT GTE ASSIGN PLUS MINUS MUL DIV NUMBER IDENTIFIER

%token UNKNOWN

%%

program:

| program statement

;

statement:

if\_stmt

| while\_stmt

| for\_stmt

| switch\_stmt

| RETURN expression SEMICOLON

;

if\_stmt:

IF LPAREN expression RPAREN statement

| IF LPAREN expression RPAREN statement ELSE statement

;

while\_stmt:

WHILE LPAREN expression RPAREN statement

;

for\_stmt:

FOR LPAREN expression SEMICOLON expression SEMICOLON expression RPAREN statement

;

switch\_stmt:

SWITCH LPAREN expression RPAREN LBRACE case\_list RBRACE

;

case\_list:

case\_list case\_stmt

| /\* empty \*/

;

case\_stmt:

CASE NUMBER COLON statement

| BREAK SEMICOLON

;

expression:

IDENTIFIER

| NUMBER

| expression PLUS expression

| expression MINUS expression

| expression MUL expression

| expression DIV expression

| LPAREN expression RPAREN

;

%%

void yyerror(const char \*s) {

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

}

int main() {

printf("Enter a C expression with control structures:\n");

yyparse();

return 0;

}

**INPUT:**

if (x > 0) {

return 1;

} else {

return 0;

}

while (x < 10) {

x = x + 1;

}

for (int i = 0; i < 10; i++) {

printf("%d", i);

}

switch (x) {

case 1:

return 1;

case 2:

return 2;

default:

return 0;

}

**OUTPUT:**

Enter a C expression with control structures:

if (x > 0) {

return 1;

} else {

return 0;

}

Syntax is valid.

-------------------------------------------------------------------------

from the given input string

-

**14. Develop a LEX program to find out the total number of vowels and consonants LEX FILE**

%{

#include <stdio.h>

int vowel\_count = 0;

int consonant\_count = 0;

%}

%%

[aAeEiIoOuU] { vowel\_count++; }

[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z] { consonant\_count++; }

[^a-zA-Z] { /\* Ignore non-alphabetic characters \*/ }

%%

int main() {

printf("Enter a string: ");

yylex(); // Call the lexer

printf("Vowels: %d\n", vowel\_count);

printf("Consonants: %d\n", consonant\_count);

return 0;

}

int yyerror(const char \*s) {

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

return 0;

}

**INPUT:**

Enter a string: Hello World!

**OUTPUT**:

Vowels: 3

Consonants: 7

-------------------------------------------------------------------------------------------------------------

**15. Develop a program to generate machine code from a given postfix notation**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_STACK\_SIZE 100

int stack[MAX\_STACK\_SIZE];

int top = -1;

void push(int value) {

if (top < MAX\_STACK\_SIZE - 1) {

stack[++top] = value;

} else {

printf("Stack overflow\n");

exit(1);

}

}

int pop() {

if (top >= 0) {

return stack[top--];

} else {

printf("Stack underflow\n");

exit(1);

}

}

void generateMachineCode(const char \*postfix) {

char token[20];

int i = 0, j = 0;

while (postfix[i] != '\0') {

// Skip spaces

if (postfix[i] == ' ') {

i++;

continue;

}

if (isdigit(postfix[i])) {

j = 0;

while (isdigit(postfix[i])) {

token[j++] = postfix[i++];

}

token[j] = '\0';

int num = atoi(token);

printf("PUSH %d\n", num);

push(num);

}

else if (postfix[i] == '+' || postfix[i] == '-' || postfix[i] == '\*' || postfix[i] == '/') {

char operator = postfix[i++];

int b = pop();

int a = pop();

if (operator == '+') {

printf("ADD\n");

push(a + b); // Perform the addition

} else if (operator == '-') {

printf("SUB\n");

push(a - b); // Perform the subtraction

} else if (operator == '\*') {

printf("MUL\n");

push(a \* b); // Perform the multiplication

} else if (operator == '/') {

if (b == 0) {

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

exit(1);

}

printf("DIV\n");

push(a / b); // Perform the division

}

} else {

printf("Error: Invalid character '%c'\n", postfix[i]);

exit(1);

}

}

if (top == 0) {

printf("Result: %d\n", stack[top]);

} else {

printf("Error: Invalid postfix expression\n");

exit(1);

}

}

int main() {

char postfix[100];

printf("Enter a postfix expression: ");

fgets(postfix, sizeof(postfix), stdin);

postfix[strcspn(postfix, "\n")] = 0; // Remove newline character

printf("Machine Code Instructions:\n");

generateMachineCode(postfix);

return 0;

}

**INPUT:**

Enter a postfix expression: 5 2 + 3 \*

**OUTPUT**:

Machine Code Instructions:

PUSH 5

PUSH 2

ADD

PUSH 3

MUL

Result: 21

---------------------------------------------------------------------------------------------------------------

**16. Write a LEX program to scan reserve words, variables and operators of c language**

**LEX file**

%{

#include <stdio.h>

#include <string.h>

void print\_reserved\_word(char \*word);

void print\_variable(char \*var);

void print\_operator(char \*op);

%}

%%

/int|char|if|else|while|for|return|break|continue|switch|case|default/ { print\_reserved\_word(yytext); }

[A-Za-z\_][A-Za-z0-9\_]\* { print\_variable(yytext); }

[+\-\*/%=<>!&|^] { print\_operator(yytext); }

"=="|"<="|">="|"<"|"=" { print\_operator(yytext); }

"!=" { print\_operator(yytext); }

"(" { printf("Left Parenthesis: (\n"); }

")" { printf("Right Parenthesis: )\n"); }

"{" { printf("Left Brace: {\n"); }

"}" { printf("Right Brace: }\n"); }

";" { printf("Semicolon: ;\n"); }

"," { printf("Comma: ,\n"); }

[ \t\n\r]+ { /\* Skip whitespace \*/ }

. { printf("Unknown symbol: %s\n", yytext); }

%%

void print\_reserved\_word(char \*word) {

printf("Reserved Word: %s\n", word);

}

void print\_variable(char \*var) {

printf("Variable: %s\n", var);

}

void print\_operator(char \*op) {

printf("Operator: %s\n", op);

}

int main() {

printf("Enter C code:\n");

yylex();

return 0;

}

**INPUT:**

int main() {

int a = 10;

float b = 20.5;

if (a > b) {

printf("Hello\n");

}

}

**OUTPUT**:

Reserved Word: int

Reserved Word: main

Left Parenthesis: (

Right Parenthesis: )

Left Brace: {

Reserved Word: int

Variable: a

Operator: =

Number: 10

Semicolon: ;

Reserved Word: float

Variable: b

Operator: =

Number: 20

Operator: .

Number: 5

Semicolon: ;

Reserved Word: if

Left Parenthesis: (

Variable: a

Operator: >

Variable: b

Right Parenthesis: )

Left Brace: {

Reserved Word: printf

Left Parenthesis: (

String literal: "Hello\n"

Right Parenthesis: )

Semicolon: ;

Right Brace: }

Right Brace: }

---------------------------------------------------------------------------------------------------

**17. Develop a program in c that converts the given three address code into assembly language statements.**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

int is\_number(char \*str) {

for (int i = 0; i < strlen(str); i++) {

if (!isdigit(str[i]) && str[i] != '.') {

return 0;

}

}

return 1;

}

void generate\_assembly(char \*tac) {

char result[10], op1[10], op2[10], operator;

sscanf(tac, "%s = %s %c %s", result, op1, &operator, op2);

if (operator == '+') {

printf("MOV %s, %s\n", result, op1); // MOV result, op1

if (!is\_number(op2)) {

printf("ADD %s, %s\n", result, op2); // ADD result, op2

} else {

printf("ADD %s, %s\n", result, op2); // ADD result, constant

}

}

else if (operator == '-') {

printf("MOV %s, %s\n", result, op1); // MOV result, op1

if (!is\_number(op2)) {

printf("SUB %s, %s\n", result, op2); // SUB result, op2

} else {

printf("SUB %s, %s\n", result, op2); // SUB result, constant

}

}

else if (operator == '\*') {

printf("MOV %s, %s\n", result, op1); // MOV result, op1

printf("MUL %s, %s\n", result, op2); // MUL result, op2

}

else if (operator == '/') {

printf("MOV %s, %s\n", result, op1); // MOV result, op1

printf("DIV %s, %s\n", result, op2); // DIV result, op2

}

else {

printf("Invalid operator!\n");

}

}

int main() {

char tac[100];

printf("Enter the three-address code (enter 'end' to stop):\n");

while (1) {

fgets(tac, sizeof(tac), stdin);

if (strncmp(tac, "end", 3) == 0) {

break;

}

generate\_assembly(tac);

}

return 0;

}

**INPUT**:

Enter the three-address code (enter 'end' to stop):

t1 = a + b

t2 = t1 \* c

t3 = t2 - d

t4 = t3 / e

end

**OUTPUT:**

MOV t1, a

ADD t1, b

MOV t2, t1

MUL t2, c

MOV t3, t2

SUB t3, d

MOV t4, t3

DIV t4, e

----------------------------------------------------------------------------------------------------------

**18. Develop a c program to eliminate left recursion from a grammar.**

#include <stdio.h>

#include <string.h>

#define MAX\_PROD 10

#define MAX\_LEN 100

void removeLeftRecursion(char \*nonTerminal, char productions[MAX\_PROD][MAX\_LEN], int prodCount) {

char newNonTerminal[MAX\_LEN];

char newProductions[MAX\_PROD][MAX\_LEN];

char temp[MAX\_LEN];

int i, j, k;

sprintf(newNonTerminal, "%s'", nonTerminal);

printf("%s -> ", nonTerminal);

for (i = 0; i < prodCount; i++) {

if (productions[i][0] == nonTerminal[0]) {

sprintf(newProductions[i], "%s%s", productions[i] + 1, newNonTerminal);

} else {

// Else use it as is

printf("%s", productions[i]);

if (i != prodCount - 1)

printf(" | ");

}

}

printf("\n");

printf("%s -> ", newNonTerminal);

for (i = 0; i < prodCount; i++) {

if (productions[i][0] == nonTerminal[0]) {

sprintf(newProductions[i], "%s%s", productions[i] + 1, newNonTerminal);

} else {

printf("%s", productions[i]);

if (i != prodCount - 1)

printf(" | ");

}

}

printf("\n");

}

int main() {

char nonTerminal[MAX\_LEN];

char productions[MAX\_PROD][MAX\_LEN];

int prodCount, i;

printf("Enter the non-terminal (e.g., A): ");

scanf("%s", nonTerminal);

printf("Enter the number of productions for %s: ", nonTerminal);

scanf("%d", &prodCount);

printf("Enter the productions for %s (one per line):\n", nonTerminal);

for (i = 0; i < prodCount; i++) {

printf("Production %d: ", i + 1);

scanf("%s", productions[i]);

}

printf("\nOriginal Grammar:\n");

for (i = 0; i < prodCount; i++) {

printf("%s -> %s\n", nonTerminal, productions[i]);

}

printf("\nAfter Eliminating Left Recursion:\n");

removeLeftRecursion(nonTerminal, productions, prodCount);

return 0;

}

**INPUT:**

Enter the non-terminal (e.g., A): A

Enter the number of productions for A: 2

Enter the productions for A (one per line):

A -> Aα

A -> β

**OUTPUT:**

Original Grammar:

A -> Aα

A -> β

After Eliminating Left Recursion:

A -> βA'

A' -> αA' | ε

-------------------------------------------------------------------------------------------------------

**19. Develop a program in c that generates an abstract syntax tree from a given arithmetic expression**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

typedef struct ASTNode {

char value; // To store operators or operands

struct ASTNode \*left; // Left child (for operands or operators)

struct ASTNode \*right; // Right child (for operands or operators)

} ASTNode;

// Function to create a new AST node

ASTNode\* createNode(char value) {

ASTNode\* newNode = (ASTNode\*)malloc(sizeof(ASTNode));

newNode->value = value;

newNode->left = newNode->right = NULL;

return newNode;

}

// Function to parse a factor (a number or a parenthesized expression)

ASTNode\* parseFactor(char\* expression, int\* index) {

ASTNode\* node;

if (isdigit(expression[\*index])) {

node = createNode(expression[\*index]);

(\*index)++;

} else if (expression[\*index] == '(') {

(\*index)++; // Skip '('

node = parseExpression(expression, index);

if (expression[\*index] == ')') {

(\*index)++; // Skip ')'

}

} else {

node = NULL;

}

return node;

}

// Function to parse terms (handles '\*' and '/' operators)

ASTNode\* parseTerm(char\* expression, int\* index) {

ASTNode\* left = parseFactor(expression, index);

while (expression[\*index] == '\*' || expression[\*index] == '/') {

char operator = expression[\*index];

(\*index)++;

ASTNode\* right = parseFactor(expression, index);

ASTNode\* operatorNode = createNode(operator);

operatorNode->left = left;

operatorNode->right = right;

left = operatorNode; // Update the left node

}

return left;

}

// Function to parse expressions (handles '+' and '-' operators)

ASTNode\* parseExpression(char\* expression, int\* index) {

ASTNode\* left = parseTerm(expression, index);

while (expression[\*index] == '+' || expression[\*index] == '-') {

char operator = expression[\*index];

(\*index)++;

ASTNode\* right = parseTerm(expression, index);

ASTNode\* operatorNode = createNode(operator);

operatorNode->left = left;

operatorNode->right = right;

left = operatorNode; // Update the left node

}

return left;

}

// Function to print the AST (in-order traversal)

void printAST(ASTNode\* root) {

if (root != NULL) {

if (root->left != NULL) {

printAST(root->left);

}

printf("%c ", root->value);

if (root->right != NULL) {

printAST(root->right);

}

}

}

// Function to evaluate the AST (basic implementation for arithmetic expressions)

int evaluateAST(ASTNode\* root) {

if (root == NULL) {

return 0;

}

// If the node is a digit, return its integer value

if (isdigit(root->value)) {

return root->value - '0';

}

// Otherwise, evaluate the left and right subtrees

int left = evaluateAST(root->left);

int right = evaluateAST(root->right);

// Perform the operation based on the operator

if (root->value == '+') {

return left + right;

} else if (root->value == '-') {

return left - right;

} else if (root->value == '\*') {

return left \* right;

} else if (root->value == '/') {

return left / right;

}

return 0;

}

int main() {

char expression[100];

printf("Enter an arithmetic expression: ");

scanf("%s", expression);

int index = 0;

ASTNode\* root = parseExpression(expression, &index);

printf("\nGenerated AST (In-Order Traversal):\n");

printAST(root);

printf("\n");

int result = evaluateAST(root);

printf("\nResult of the expression: %d\n", result);

return 0;

}

**INPUT**:

Enter an arithmetic expression: a+b\*c

**OUTPUT:**

Generated AST (In-Order Traversal):

a + b \* c

Result of the expression: 10

-------------------------------------------------------------------------------------------------------------

**20. Design a top-down parser which generates a pursing table with no backtracking**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_INPUT\_LEN 100

#define MAX\_STACK\_SIZE 100

// Define non-terminal symbols and terminal symbols

#define S 0

#define E 1

#define E\_prime 2

#define T 3

#define T\_prime 4

#define F 5

#define id 6

#define lparen 7

#define rparen 8

#define plus 9

#define star 10

#define dollar 11

// Parsing Table

int parseTable[6][12] = {

{1, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, // S -> E

{3, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, // E -> T E'

{-1, -1, 4, 5, -1, 5, -1, -1, -1, -1, -1, -1}, // E' -> + T E' | ε

{6, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}, // T -> F T'

{-1, -1, -1, -1, 7, 5, -1, -1, -1, -1, -1, -1}, // T' -> \* F T' | ε

{8, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1} // F -> id | ( E )

};

// Stack for top-down parsing

int stack[MAX\_STACK\_SIZE];

int top = -1;

void push(int symbol) {

if (top < MAX\_STACK\_SIZE - 1) {

stack[++top] = symbol;

}

}

int pop() {

if (top == -1) {

return -1;

}

return stack[top--];

}

void parse(char\* input) {

int index = 0;

int inputLength = strlen(input);

// Push start symbol S onto the stack

push(S);

// Process the input string

while (top != -1) {

int topSymbol = pop();

// Match terminal symbol

if (topSymbol >= id && topSymbol <= dollar) {

if (index < inputLength && input[index] == topSymbol) {

index++;

} else {

printf("Error: Expected %c but found %c\n", topSymbol, input[index]);

return;

}

}

// Match non-terminal symbol

else if (topSymbol >= S && topSymbol <= F) {

int rule = parseTable[topSymbol][input[index]];

if (rule == -1) {

printf("Error: No rule for %d with input %c\n", topSymbol, input[index]);

return;

} else {

// Apply rule (push right-hand side of the rule in reverse order)

if (rule == 5) continue; // ε rule

push(rule); // Continue the rule application

}

}

}

if (index == inputLength) {

printf("Parsing completed successfully.\n");

} else {

printf("Error: Input not completely parsed.\n");

}

}

int main() {

char input[MAX\_INPUT\_LEN];

printf("Enter an arithmetic expression: ");

fgets(input, MAX\_INPUT\_LEN, stdin);

input[strcspn(input, "\n")] = 0; // Remove trailing newline

parse(input);

return 0;

}

**INPUT:**

Enter an arithmetic expression: id + id \* id

**OUTPUT**:

Parsing completed successfully.

------------------------------------------------------------------------------------------------------------------