**Develop a program in C to design a lexical analyzer that recognizes identifiers and constants**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX\_LENGTH 100

// Function to check if a string is a keyword

int isKeyword(char \*token) {

char keywords[8][10] = {"int", "float", "char", "if", "else", "while", "for", "break"};

int i, flag = 0;

for (i = 0; i < 8; ++i) {

if (strcmp(keywords[i], token) == 0) {

flag = 1;

break;

}

}

return flag;

}

// Function to recognize identifiers and constants

void lexer(char \*input) {

int i = 0;

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

// If the character is a letter (identifier)

if (isalpha(input[i])) {

char token[MAX\_LENGTH] = "";

int j = 0;

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

while (isalnum(input[i]) || input[i] == '\_') {

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

}

token[j] = '\0';

if (isKeyword(token)) {

printf("'%s' is a keyword\n", token);

} else {

printf("'%s' is an identifier\n", token);

}

}

// If the character is a digit (constant)

else if (isdigit(input[i])) {

char token[MAX\_LENGTH] = "";

int j = 0;

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

while (isdigit(input[i]) || input[i] == '.') {

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

}

token[j] = '\0';

printf("'%s' is a constant\n", token);

}

// Ignore spaces

else if (isspace(input[i])) {

i++;

}

else {

// For other characters, simply print as a special symbol

printf("'%c' is a special symbol\n", input[i++]);

}

}

}

int main() {

char input[MAX\_LENGTH];

printf("Enter the input string: ");

fgets(input, MAX\_LENGTH, stdin);

printf("Lexical analysis:\n");

lexer(input);

return 0;

}

**OUTPUT**

Enter the input string: hello if how are

Lexical analysis:

'hello' is an identifier

'if' is a keyword

'how' is an identifier

'are' is an identifier

—-----------------------------------------------------------------------------------------------------------------------------------------

**Implement a symbol table that involves insertion, deletion, search and modify operations using C language**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 100

// Structure for a symbol table entry

typedef struct {

char name[30];

int value;

// You can add more fields as needed for your specific use case

} SymbolEntry;

// Structure for the symbol table

**typedef struct** {

SymbolEntry entries[MAX\_SIZE];

int count;

} SymbolTable;

// Function to initialize the symbol table

void initializeSymbolTable(SymbolTable \*table) {

table->count = 0;

}

// Function to insert an entry into the symbol table

void **insertEntry**(SymbolTable \*table, char \*name, int value) {

if (table->count >= MAX\_SIZE) {

printf("Symbol table overflow!\n");

return;

}

SymbolEntry newEntry;

strcpy(newEntry.name, name);

newEntry.value = value;

table->entries[table->count++] = newEntry;

}

// Function to search for an entry in the symbol table

int **searchEntry**(SymbolTable \*table, char \*name) {

int i;

for (i = 0; i < table->count; ++i) {

if (strcmp(table->entries[i].name, name) == 0) {

return i; // Return the index if found

}

}

return -1; // Return -1 if not found

}

// Function to delete an entry from the symbol table

void **deleteEntry**(SymbolTable \*table, char \*name) {

int index = searchEntry(table, name);

if (index == -1) {

printf("Entry '%s' not found in the symbol table.\n", name);

return;

}

// Replace the entry to be deleted with the last entry and decrement count

table->entries[index] = table->entries[table->count - 1];

table->count--;

printf("Entry '%s' deleted from the symbol table.\n", name);

}

// Function to modify the value of an entry in the symbol table

void **modifyEntry**(SymbolTable \*table, char \*name, int newValue) {

int index = searchEntry(table, name);

if (index == -1) {

printf("Entry '%s' not found in the symbol table.\n", name);

return;

}

table->entries[index].value = newValue;

printf("Value of entry '%s' modified in the symbol table.\n", name);

}

// Function to display the symbol table

void **displaySymbolTable**(SymbolTable \*table) {

printf("Symbol Table:\n");

printf("Name\tValue\n");

int i;

for (i = 0; i < table->count; ++i) {

printf("%s\t%d\n", table->entries[i].name, table->entries[i].value);

}

}

int main() {

SymbolTable symbolTable;

initializeSymbolTable(&symbolTable);

// Inserting entries into the symbol table

insertEntry(&symbolTable, "x", 10);

insertEntry(&symbolTable, "y", 20);

insertEntry(&symbolTable, "z", 30);

displaySymbolTable(&symbolTable);

// Modifying an entry in the symbol table

modifyEntry(&symbolTable, "y", 50);

displaySymbolTable(&symbolTable);

// Deleting an entry from the symbol table

deleteEntry(&symbolTable, "x");

displaySymbolTable(&symbolTable);

return 0;

}

**OUTPUT**

Symbol Table:

Name Value

x 10

y 20

z 30

Value of entry 'y' modified in the symbol table.

Symbol Table:

Name Value

x 10

y 50

z 30

Entry 'x' deleted from the symbol table.

Symbol Table:

Name Value

z 30

y 50

—-----------------------------------------------------------------------------------------------------------------------------------------

**Design a program that implements a lexical analyzer that separates token using a LEX tool**

%{

#include <stdio.h>

%}

/\* Definitions \*/

DIGIT [0-9]

LETTER [a-zA-Z]

IDENTIFIER {LETTER}({LETTER}|{DIGIT})\*

CONSTANT {DIGIT}+(\.{DIGIT}+)?([eE][+-]?{DIGIT}+)?

/\* Rules \*/

%%

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

{CONSTANT} { printf("Constant: %s\n", yytext); }

[()+-/\*=] { printf("Operator: %c\n", yytext[0]); }

[ \t\n] ; /\* ignore whitespace characters \*/

. { printf("Invalid character: %c\n", yytext[0]); }

%%

int yywrap() {

return 1;

}

int main() {

yylex();

return 0;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

**Use YACC tool to recognize a valid arithmetic expression that uses basic arithmetic operators[+,-,\*,/].**

%{

#include <stdio.h>

#include <stdlib.h>

%}

%token NUMBER

%left '+' '-'

%left '\*' '/'

%%

expression : expression '+' expression

| expression '-' expression

| expression '\*' expression

| expression '/' expression

| '(' expression ')'

| NUMBER

;

%%

int yylex(); // The lexer function is provided externally

int yyerror(const char \*s) {

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

return 0;

}

int main() {

yyparse();

return 0;

}

—---------------------------

%{

#include "y.tab.h"

#include <stdio.h>

%}

DIGIT [0-9]

NUMBER {DIGIT}+

WS [ \t\n]

%%

"+" { return '+'; }

"-" { return '-'; }

"\*" { return '\*'; }

"/" { return '/'; }

"(" { return '('; }

")" { return ')'; }

{NUMBER} { yylval = atoi(yytext); return NUMBER; }

{WS} ; // Ignore whitespace

. { printf("Invalid character: %c\n", yytext[0]); return yytext[0]; }

%%

int yywrap() {

return 1;

}

**yacc -d parser.y # Generates y.tab.c and y.tab.h**

**flex lexer.l # Generates lex.yy.c**

**gcc -o parser y.tab.c lex.yy.c -ll**

**—-----------------------------------------------------------------------------------------------------------------------------------------**

**Design a program to recognize a valid variable which starts with an alphabet followed by any number of digits or alphabets using YACC tool.**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

%}

%token IDENTIFIER

%%

program : /\* empty \*/

| program statement '\n'

;

statement : IDENTIFIER { printf("Valid variable: %s\n", $1); }

;

IDENTIFIER : [a-zA-Z][a-zA-Z0-9]\*

;

%%

int yylex(); // The lexer function is provided externally

int yyerror(const char \*s) {

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

return 0;

}

int main() {

printf("Enter a sequence of variables:\n");

yyparse();

return 0;

}

—-------------

%{

#include "y.tab.h"

#include <stdio.h>

%}

%%

[a-zA-Z][a-zA-Z0-9]\* { yylval = strdup(yytext); return IDENTIFIER; }

\n return '\n';

. return yytext[0];

%%

int yywrap() {

return 1;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

**Use LEX and YACC tools to implement a native calculator**

%{

#include<stdio.h>

int flag=0;

%}

%token NUMBER

%left '+' '-'

%left '\*' '/' '%'

%left '(' ')'

%%

ArithmeticExpression: E

{

printf("\nResult=%d\n",$$);

return 0;

};

E:E'+'E {$$=$1+$3;}

|E'-'E {$$=$1-$3;}

|E'\*'E {$$=$1\*$3;}

|E'/'E {$$=$1/$3;}

|E'%'E {$$=$1%$3;}

|'('E')' {$$=$2;}

|NUMBER {$$=$1;};

%%

void main()

{

printf("\nEnter Any Arithmetic Expression which have operations:\n");

yyparse();

if(flag==0)

printf("\nEntered arithmetic expression is Valid\n\n");

}

void yyerror()

{

printf("\nEntered arithmetic expression is Invalid\n\n");

flag=1;

}

—-------------------------

%{

#include<stdio.h>

#include "y.tab.h"

extern int yylval;

%}

%%

[0-9]+ {

yylval=atoi(yytext);

return NUMBER;

}

[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()

{

return 1;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

**Design a program to generate a three-address code from a given arithmetic expression.**

%{

#include <math.h>

#include<ctype.h>

#include<stdio.h>

int var\_cnt=0;

char iden[20];

%}

%token digit

%token id

%%

S:id '=' E { printf("%s = t%d\n",iden, var\_cnt-1); }

E:E '+' T { $$=var\_cnt; var\_cnt++; printf("t%d = t%d + t%d;\n", $$, $1, $3 ); }

|E '-' T { $$=var\_cnt; var\_cnt++; printf("t%d = t%d - t%d;\n", $$, $1, $3 ); }

|T { $$=$1; }

;

T:T '\*' F { $$=var\_cnt; var\_cnt++; printf("t%d = t%d \* t%d;\n", $$, $1, $3 ); }

|T '/' F { $$=var\_cnt; var\_cnt++; printf("t%d = t%d / t%d;\n", $$, $1, $3 ); }

|F {$$=$1 ; }

;

F:P '^' F { $$=var\_cnt; var\_cnt++; printf("t%d = t%d ^ t%d;\n", $$, $1, $3 );}

| P { $$ = $1;}

;

P: '(' E ')' { $$=$2; }

|digit { $$=var\_cnt; var\_cnt++; printf("t%d = %d;\n",$$,$1); }

;

\

%%

int main()

{

var\_cnt=0;

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

yyparse();

return 0;

}

yyerror()

{

printf("Error\n");

}

—------------------------------------------

%{

#include<stdio.h>

#include<string.h>

#include"y.tab.h"

extern char iden[20];

extern int yylval;

%}

%%

[0-9]+ {yylval=atoi(yytext);

return digit;

}

[a-zA-Z]+ {strcpy(iden,yytext);

yylval=1;

return id;}

[\t] {;}

[\n] return 0;

. return yytext[0];

%%

int yywrap()

{

return 1;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

**Implement a simple type checker that checks the scope of the variables and semantic errors from the given statement**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_VARIABLES 50

struct Variable {

char name[20];

int scope;

};

struct Variable symbolTable[MAX\_VARIABLES];

int currentScope = 0;

int symbolCount = 0;

int isVariableDeclared(char \*name, int scope) {

int i;

for (i = symbolCount - 1; i >= 0; i--) {

if (strcmp(symbolTable[i].name, name) == 0 && symbolTable[i].scope == scope) {

return 1;

}

}

return 0;

}

%}

DIGIT [0-9]

LETTER [a-zA-Z]

WS [ \t\n]

%%

"int" { /\* Code to handle variable declarations \*/ }

{LETTER}({LETTER}|{DIGIT})\* {

if (!isVariableDeclared(yytext, currentScope)) {

printf("Semantic Error: Variable '%s' used without declaration in scope %d\n", yytext, currentScope);

}

}

"{" { currentScope++; }

"}" { currentScope--; }

{WS} ; // Ignore whitespace

. { /\* Other tokens or characters, if needed \*/ }

%%

int main() {

printf("Enter a statement:\n");

yylex();

return 0;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

**Develop a program that optimizes the given input block using Code Optimization Techniques.**

%{

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

%}

%token NUMBER ADD SUB MUL DIV

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

"+" { return ADD; }

"-" { return SUB; }

"\*" { return MUL; }

"/" { return DIV; }

"(" { return '('; }

")" { return ')'; }

[ \t\n] ; /\* Ignore whitespace \*/

%%

bool isOperator(int token) {

return (token == ADD || token == SUB || token == MUL || token == DIV);

}

int evaluate(int op, int left, int right) {

switch (op) {

case ADD: return left + right;

case SUB: return left - right;

case MUL: return left \* right;

case DIV: return right != 0 ? left / right : 0; // Handle division by zero

default: return 0;

}

}

int optimize(int token) {

int left, right;

switch (token) {

case ADD:

case SUB:

case MUL:

case DIV:

right = yylex();

if (isOperator(right)) {

right = optimize(right);

}

left = yylex();

if (isOperator(left)) {

left = optimize(left);

}

if (!isOperator(left) && !isOperator(right)) {

return evaluate(token, left, right);

} else {

unput(right);

unput(left);

return token;

}

default: return token;

}

}

int main() {

printf("Enter an arithmetic expression: ");

int token;

while ((token = yylex()) != 0) {

int result = optimize(token);

if (!isOperator(result)) {

printf("%d\n", result);

} else {

printf("Operator: %c\n", result);

}

}

return 0;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

**Given an intermediate code as an input. Develop a program that generates the machine code from the given input**

#include<stdio.h>  
#include<string.h>  
int main()  
{  
char icode[20][30];  
char str[20],opr[10];  
int i =0;  
printf("\n enter the set of intermediate code instructions:\n");  
  
do  
{  
scanf("%s",icode[i]);  
}while(strcmp(icode[i++],"exit") != 0);  
i = 0;  
do{  
strcpy(str, icode[i]);  
switch(str[3])  
{  
case '+':  
strcpy(opr,"ADD");  
break;  
case '-':  
strcpy(opr,"SUB");  
break;  
case '\*':  
strcpy(opr,"MUL");  
break;  
case '/':  
strcpy(opr,"DIV");  
break;  
}  
printf("\n MOV %c,R%d",str[2],i);  
printf("\n %s %c,R%d",opr,str[4],i);  
printf("\n MOV R%d,%c",i,str[0]);  
}while(strcmp(icode[++i],"exit") !=0);  
  
return 0;  
}

—-----------------------------------------------------------------------------------------------------------------------------------------

**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**

%{

#include <stdio.h>

%}

%token STATEMENT

%%

start : STATEMENT '\n' { printf("Valid statement: %s\n", $1); }

;

STATEMENT : [A-Z][A-Za-z0-9]{5}

;

%%

int yylex();

int yyerror(const char \*s) {

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

return 0;

}

int main() {

printf("Enter a statement: ");

yyparse();

return 0;

}

—-------------------------------------------------------------

%{

#include "y.tab.h"

#include <stdio.h>

%}

%%

[A-Z][A-Za-z0-9]{5} { yylval = strdup(yytext); return STATEMENT; }

\n return '\n';

. return yytext[0];

%%

int yywrap() {

return 1;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

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

%{

#include <stdio.h>

#include <string.h>

%}

%%

\/\/.\* { printf("Comment: %s\n", yytext); }

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

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

[+\-\*/%=<>!] { printf("Operator: %c\n", yytext[0]); }

[ \t\n]+ ; /\* Skip whitespace \*/

. { printf("Identifier or constant: %s\n", yytext); }

%%

int main() {

printf("Enter an expression: ");

yylex();

return 0;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

**Develop a Program to recognize a valid control structures syntax of C language (For loop, while loop, if else, if-else-if, switch-case, etc.).**

%{

#include <stdio.h>

#include <string.h>

%}

%token FOR WHILE IF ELSE SWITCH CASE BREAK CONTINUE

%%

program : /\* empty \*/

| program statement '\n'

;

statement : loop

| conditional

| BREAK '\n'

| CONTINUE '\n'

;

loop : for\_loop

| while\_loop

;

for\_loop : FOR '(' expression ';' expression ';' expression ')' '{' program '}'

;

while\_loop : WHILE '(' expression ')' '{' program '}'

;

conditional : IF '(' expression ')' '{' program '}'

| IF '(' expression ')' '{' program '}' ELSE '{' program '}'

| IF '(' expression ')' '{' program '}' ELSE\_IF '{' program '}'

| SWITCH '(' expression ')' '{' cases '}'

;

ELSE\_IF : ELSE IF '(' expression ')'

;

cases : /\* empty \*/

| cases CASE expression ':' program

;

expression : /\* Placeholder for expression parsing \*/

;

%%

int yylex();

int yyerror(const char \*s) {

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

return 0;

}

int main() {

printf("Enter the C control structure code:\n");

yyparse();

return 0;

}

—-------------------------------------------------------------

%{

#include "y.tab.h"

#include <stdio.h>

%}

%%

"for" { return FOR; }

"while" { return WHILE; }

"if" { return IF; }

"else" { return ELSE; }

"switch" { return SWITCH; }

"case" { return CASE; }

"break" { return BREAK; }

"continue" { return CONTINUE; }

"&&" /\* Ignored \*/

"||" /\* Ignored \*/

";" /\* Ignored \*/

"+"|"-"|"\*"|"/"|"=" /\* Ignored \*/

"=="|"!="|">="|"<=" /\* Ignored \*/

"(" /\* Ignored \*/

")" /\* Ignored \*/

"{" /\* Ignored \*/

"}" /\* Ignored \*/

":" /\* Ignored \*/

"," /\* Ignored \*/

[a-zA-Z\_][a-zA-Z0-9\_]\* /\* Ignored \*/

[ \t\n] /\* Ignored \*/

. /\* Ignored \*/

%%

int yywrap() {

return 1;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

**Develop a Lex program to find out the total number of vowels and consonants from the given input string**

%{

#include <stdio.h>

int vowels = 0;

int consonants = 0;

%}

%%

[a-zA-Z] {

if (yytext[0] == 'a' || yytext[0] == 'e' || yytext[0] == 'i' || yytext[0] == 'o' || yytext[0] == 'u' ||

yytext[0] == 'A' || yytext[0] == 'E' || yytext[0] == 'I' || yytext[0] == 'O' || yytext[0] == 'U') {

vowels++;

} else {

consonants++;

}

}

\n {

printf("Total vowels: %d\n", vowels);

printf("Total consonants: %d\n", consonants);

vowels = 0;

consonants = 0;

}

. /\* Ignore any other characters \*/

%%

int main() {

yylex();

return 0;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

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

%{

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

%}

%option noyywrap

%%

[0-9]+ {

printf("PUSH %d\n", atoi(yytext));

}

[+\-\*/] {

printf("%s\n", yytext);

}

\n {

printf("END\n");

}

[ \t] {

/\* Ignore whitespace \*/

}

. {

printf("Invalid character: %s\n", yytext);

}

%%

int main() {

yylex();

return 0;

}

**OUTPUT**

PUSH 3

PUSH 4

+

PUSH 5

\*

END

**Write a LEX program to scan reserved words, variables and operators of C language**

%{

#include <stdio.h>

#include <string.h>

%}

%%

"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"register"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|"typedef"|"union"|"unsigned"|"void"|"volatile"|"while" {

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

}

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

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

}

"+"|"-"|"\*"|"/"|"="|"=="|"!="|">"|"<"|">="|"<="|"&&"|"||"|"!"|"&"|"|"|"%"|"<<|">>"|"&="|"|="|"^="|"<<="|">>="|"++"|"--"|"+="|"-="|"\*="|"/="|"->"|"."|"->\*"|","|";"|"?"|":" {

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

}

[ \t\n] {

/\* Ignore whitespace \*/

}

. {

printf("Invalid Character: %s\n", yytext);

}

%%

int main() {

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

yylex();

return 0;

}

—-----------------------------------------------------------------------------------------------------------------------------------------

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

**SEE ABOVE**

**—--------------------------------------------------------------------------------------------------------------------------**

**Develop a C program to eliminate left recursion from a grammar**

#include <stdio.h>

#include <string.h>

void eliminateLeftRecursion(char nonTerminal, char \*productionRules[], int numOfRules) {

printf("Original Grammar for %c:\n", nonTerminal);

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

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

}

printf("\nAfter Eliminating Left Recursion for %c:\n", nonTerminal);

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

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

char newRule[50] = "";

strcat(newRule, &productionRules[i][1]);

strcat(newRule, " ");

strcat(newRule, nonTerminal == 'A' ? "'" : "");

printf("%c' -> %s%c'\n", nonTerminal, newRule, nonTerminal);

} else {

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

}

}

printf("%c' -> epsilon\n\n", nonTerminal);

}

int main() {

char \*productionRulesA[] = {"A alpha", "beta"};

eliminateLeftRecursion('A', productionRulesA, sizeof(productionRulesA) / sizeof(productionRulesA[0]));

return 0;

}

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

#include <stdio.h>

#include <stdlib.h>

// Node structure for the AST

**typedef struct Node** {

char data;

struct Node \*left;

struct Node \*right;

} Node;

// Create a new node for the AST

Node \***createNode**(char data) {

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

newNode->data = data;

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

return newNode;

}

// Build AST recursively from the given expression

Node \***buildAST**(char expression[], int \*index) {

if (expression[\*index] == '\0') {

return NULL;

}

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

(\*index)++;

if (expression[\*index] == '+' || expression[\*index] == '-' ||

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

(\*index)++;

node->left = buildAST(expression, index);

(\*index)++;

node->right = buildAST(expression, index);

(\*index)++;

}

return node;

}

// Print the AST in an in-order traversal

void **printAST**(Node \*root) {

if (root == NULL) {

return;

}

printAST(root->left);

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

printAST(root->right);

}

int main() {

char expression[] = "a+b\*c"; // Replace with your arithmetic expression

int index = 0;

Node \*root = buildAST(expression, &index);

printf("Abstract Syntax Tree (In-order traversal): ");

printAST(root);

printf("\n");

return 0;

}

—----------------------------------------------------------------------------------------------------------------------------

**Design a top-down parser which generates a parsing table with no backtracking**

%{

#include <stdio.h>

int lookahead = -1;

int getToken();

void E();

void T();

%}

%token int\_token lparen rparen plus eof

%%

expr : E eof { printf("Valid Expression!\n"); }

;

E : T plus E | T

;

T : int\_token | lparen E rparen

;

%%

int getToken() {

return yylex();

}

void match(int token) {

if (lookahead == token) {

lookahead = getToken();

} else {

fprintf(stderr, "Syntax Error!\n");

exit(1);

}

}

void E() {

T();

if (lookahead == plus) {

match(plus);

E();

}

}

void T() {

if (lookahead == int\_token) {

match(int\_token);

} else if (lookahead == lparen) {

match(lparen);

E();

match(rparen);

} else {

fprintf(stderr, "Syntax Error!\n");

exit(1);

}

}

int main() {

printf("Enter an arithmetic expression: ");

lookahead = getToken();

expr();

return 0;

}