1.TOKEN IDENTIFICATION

/\* 1. LEX code for identifying constants, operators, and identifiers \*/

%{

#include <stdio.h>

%}

%%

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

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

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Identifier: %s\n", yytext); }

[ \t\n] ;

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

%%

int main() {

printf("Enter an expression: ");

yylex();

return 0;

}

int yywrap() {

return 1;

}

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX\_LEN 30

void checkToken(char \*token) {

if (isdigit(token[0]))

printf("Constant: %s\n", token);

else if (strcmp(token, "+") == 0 || strcmp(token, "-") == 0 || strcmp(token, "\*") == 0 || strcmp(token, "/") == 0)

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

else

printf("Identifier: %s\n", token);

}

int main() {

char input[MAX\_LEN], token[MAX\_LEN];

int i = 0, j = 0;

printf("Enter an expression: ");

fgets(input, MAX\_LEN, stdin);

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

if (isalnum(input[i]))

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

else {

if (j > 0) {

token[j] = '\0';

checkToken(token);

j = 0;

}

if (input[i] != ' ')

checkToken((char[]){input[i], '\0'});

}

i++;

}

if (j > 0) {

token[j] = '\0';

checkToken(token);

}

return 0;

}

2.TO CHECK IF GIVEN IS COMMENT OR NOT

LEX

%{

#include <stdio.h>

%}

%%

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

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

.\* { printf("Not a comment: %s\n", yytext); }

%%

int main() {

printf("Enter a line: ");

yylex();

return 0;

}

int yywrap() {

return 1;

}

#include <stdio.h>

#include <string.h>

void checkComment(char \*line) {

if (strncmp(line, "//", 2) == 0)

printf("Single-line Comment: %s\n", line);

else if (strstr(line, "/\*") && strstr(line, "\*/"))

printf("Multi-line Comment: %s\n", line);

else

printf("Not a comment: %s\n", line);

}

int main() {

char line[100];

printf("Enter a line: ");

fgets(line, sizeof(line), stdin);

checkComment(line);

return 0;

}

3.REDUNDANT SPACES

%{

#include <stdio.h>

%}

%%

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

. { printf("Not an operator: %s\n", yytext); }

%%

int main() {

printf("Enter an expression: ");

yylex();

return 0;

}

int yywrap() {

return 1;

}

#include <stdio.h>

#include <string.h>

void checkOperator(char ch) {

if (ch == '+' || ch == '-' || ch == '\*' || ch == '/')

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

else

printf("Not an operator: %c\n", ch);

}

int main() {

char expr[50];

printf("Enter an expression: ");

fgets(expr, sizeof(expr), stdin);

for (int i = 0; expr[i] != '\0' && expr[i] != '\n'; i++) {

checkOperator(expr[i]);

}

return 0;

}

4.RECOGNIZWE THE OPERATORS

%{

#include <stdio.h>

int whitespace = 0, newline = 0;

%}

%%

[ \t] { whitespace++; }

\n { newline++; }

. ;

%%

int main() {

printf("Enter text (Ctrl+D to stop):\n");

yylex();

printf("Whitespaces: %d, Newlines: %d\n", whitespace, newline);

return 0;

}

int yywrap() {

return 1;

}

#include <stdio.h>

int main() {

char ch;

int whitespace = 0, newline = 0;

printf("Enter text (Ctrl+D to stop):\n");

while ((ch = getchar()) != EOF) {

if (ch == ' ' || ch == '\t')

whitespace++;

if (ch == '\n')

newline++;

}

printf("Whitespaces: %d, Newlines: %d\n", whitespace, newline);

return 0;

}

5.CHECK IF IT IS A IDENTIFER

%{

#include <stdio.h>

%}

%%

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Valid Identifier: %s\n", yytext); }

[^a-zA-Z0-9\_] { printf("Invalid Identifier: %s\n", yytext); }

%%

int main() {

printf("Enter an identifier: ");

yylex();

return 0;

}

int yywrap() {

return 1;

}

#include <stdio.h>

#include <ctype.h>

#include <string.h>

int isValidIdentifier(char \*id) {

if (!isalpha(id[0]) && id[0] != '\_')

return 0;

for (int i = 1; id[i] != '\0'; i++) {

if (!isalnum(id[i]) && id[i] != '\_')

return 0;

}

return 1;

}

int main() {

char id[50];

printf("Enter an identifier: ");

scanf("%s", id);

if (isValidIdentifier(id))

printf("Valid Identifier\n");

else

printf("Invalid Identifier\n");

return 0;

}

6.LEFT RECURSION detection

%{

#include <stdio.h>

%}

%%

[A-Za-z]+ -> [A-Za-z]+ { printf("Left recursion detected: %s\n", yytext); }

[A-Za-z]+ { printf("Production: %s\n", yytext); }

%%

int main() {

printf("Enter a production rule: ");

yylex();

return 0;

}

int yywrap() {

return 1;

}

#include <stdio.h>

#include <string.h>

void eliminateLeftRecursion(char \*nonTerminal, char \*productions[]) {

char alpha[10][10], beta[10][10];

int a = 0, b = 0;

for (int i = 0; productions[i] != NULL; i++) {

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

strcpy(alpha[a++], productions[i] + 1);

} else {

strcpy(beta[b++], productions[i]);

}

}

if (a == 0) {

printf("No left recursion in the given grammar.\n");

return;

}

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

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

printf("%s%s' ", beta[i], nonTerminal);

}

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

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

printf("%s%s' ", alpha[i], nonTerminal);

}

printf("| ε\n");

}

int main() {

char \*productions[] = {"Aa", "b", "Ac", "d", NULL};

char nonTerminal[] = "A";

eliminateLeftRecursion(nonTerminal, productions);

return 0;

}

7.LEFT FACTORING ELEMINATION

#include <stdio.h>

#include <string.h>

#define MAX\_PROD 10

#define MAX\_LEN 20

void eliminateLeftFactoring(char nonTerminal, char productions[MAX\_PROD][MAX\_LEN], int n) {

int i, j, prefixLen = 0;

char prefix[MAX\_LEN], alpha[MAX\_PROD][MAX\_LEN], beta[MAX\_PROD][MAX\_LEN];

int alphaCount = 0, betaCount = 0;

// Find the longest common prefix

strcpy(prefix, productions[0]);

for (i = 1; i < n; i++) {

for (j = 0; prefix[j] && productions[i][j] && prefix[j] == productions[i][j]; j++);

prefix[j] = '\0'; // Trim prefix to common part

}

prefixLen = strlen(prefix);

// If no common prefix found, return original productions

if (prefixLen == 0) {

printf("No left factoring needed for %c\n", nonTerminal);

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

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

}

return;

}

// Separate α (after common prefix) and β (remaining productions)

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

if (strncmp(productions[i], prefix, prefixLen) == 0) {

strcpy(alpha[alphaCount++], productions[i] + prefixLen);

} else {

strcpy(beta[betaCount++], productions[i]);

}

}

// Print the left-factored productions

printf("Left Factored Grammar:\n");

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

// Print α productions

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

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

printf("%s%s", alpha[i][0] ? alpha[i] : "ε", (i == alphaCount - 1) ? "\n" : " | ");

}

// Print β productions separately

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

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

}

}

int main() {

char productions[MAX\_PROD][MAX\_LEN] = {

"abcX", "abcY", "cd"

};

int n = 3;

char nonTerminal = 'A';

eliminateLeftFactoring(nonTerminal, productions, n);

return 0;

}

8.SYMBOL TABLE IMPLEMENTATION

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct Symbol {

char name[50];

char type[10];

} table[100];

int count = 0;

void insert(char \*name, char \*type) {

strcpy(table[count].name, name);

strcpy(table[count].type, type);

count++;

}

void display() {

printf("Symbol Table:\n");

printf("Name\tType\n");

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

printf("%s\t%s\n", table[i].name, table[i].type);

}

}

int main() {

insert("x", "int");

insert("y", "float");

insert("z", "char");

display();

return 0;

}

9.GRAMMAR VALIDATION

#include <stdio.h>

#include <string.h>

#include <ctype.h>

int isValidGrammar(char \*str) {

for (int i = 0; str[i]; i++) {

if (!isalnum(str[i]) && str[i] != ' ')

return 0; // Invalid character found

}

return 1;

}

int main() {

char str[100];

printf("Enter a string: ");

fgets(str, sizeof(str), stdin); // Use fgets instead of gets

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

if (isValidGrammar(str))

printf("String follows grammar.\n");

else

printf("String does not follow grammar.\n");

return 0;

}

10.RECURSIVE DESCENT PARCER

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

char \*input;

void E(); void T(); void F();

void error() { printf("Error in parsing.\n"); exit(1); }

void match(char expected) { if (\*input == expected) input++; else error(); }

void E() { T(); while (\*input == '+' || \*input == '-') { input++; T(); } }

void T() { F(); while (\*input == '\*' || \*input == '/') { input++; F(); } }

void F() { if (\*input == '(') { input++; E(); match(')'); } else if (isalnum(\*input)) input++; else error(); }

int main() {

char expr[100];

printf("Enter expression (e.g., a+b\*(c-d)): ");

scanf("%s", expr);

input = expr;

E();

if (\*input == '\0') printf("Parsing successful!\n");

else printf("Parsing failed.\n");

return 0;

}

11.OPERATOR PRECESENDENCE PARSING

#include <stdio.h>

#include <ctype.h>

int precedence(char op) {

return (op == '+' || op == '-') ? 1 : (op == '\*' || op == '/') ? 2 : 0;

}

int main() {

char expr[100];

printf("Enter an expression: ");

scanf("%s", expr);

printf("Operators in order of precedence: ");

for (int i = 0; expr[i]; i++) {

if (precedence(expr[i]))

printf("%c ", expr[i]);

}

printf("\n");

return 0;

}

12.THREE ADDRESS CODE

#include <stdio.h>

int tempVar = 1;

void genTAC(char op, char a, char b) {

printf("t%d = %c %c %c\n", tempVar++, a, op, b);

}

int main() {

char a, b, op;

printf("Enter expression (e.g., A+B): ");

scanf(" %c %c %c", &a, &op, &b);

genTAC(op, a, b);

return 0;

}

13.CHARACTER WORD AND LINE COUNTER

#include <stdio.h>

#include <string.h>

int main() {

char str[1000];

int chars = 0, words = 0, lines = 0;

printf("Enter text (Press ENTER twice to finish):\n");

while (1) {

fgets(str, sizeof(str), stdin);

if (strcmp(str, "\n") == 0) // Detect double ENTER (empty line)

break;

chars += strlen(str);

for (int i = 0; str[i]; i++) {

if (str[i] == ' ' || str[i] == '\n') words++;

if (str[i] == '\n') lines++;

}

}

printf("\nCharacters: %d\nWords: %d\nLines: %d\n", chars, words, lines);

return 0;

}

14.CODE OPTIMIZATION

#include <stdio.h>

#include <string.h>

void optimize(char expr[]) {

if (strstr(expr, "a\*b") && strstr(expr, "a\*b"))

printf("Optimized: t1 = a \* b\nUse t1 instead.\n");

else

printf("No optimization needed.\n");

}

int main() {

char expr[50];

printf("Enter expression (e.g., a\*b + a\*b): ");

scanf("%s", expr);

optimize(expr);

return 0;

}

15.INTERMIDEATE CODE GENERATOR

#include <stdio.h>

#include <string.h>

#include <ctype.h>

void generateAssembly(char expr[]) {

int len = strlen(expr);

int tempVar = 1;

printf("\nGenerated Assembly:\n");

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

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

printf("MOV R%d, %c\n", tempVar, expr[i-1]); // Load operand 1

printf("%s R%d, %c\n", (expr[i] == '+' ? "ADD" : expr[i] == '-' ? "SUB" : expr[i] == '\*' ? "MUL" : "DIV"), tempVar, expr[i+1]);

printf("MOV %c, R%d\n", expr[i+1], tempVar); // Store result

tempVar++;

}

}

}

int main() {

char expr[100];

printf("Enter an arithmetic expression (e.g., A+B\*C): ");

scanf("%s", expr);

generateAssembly(expr);

return 0;

}

16.Lexical Analysis (Counting Characters, Words, and Lines using LEX)

%{

#include <stdio.h>

int char\_count = 0, word\_count = 0, line\_count = 0;

%}

%%

[^\n\t ]+ { word\_count++; char\_count += yyleng; }

\n { line\_count++; char\_count++; }

. { char\_count++; }

%%

int main() {

yylex();

printf("Characters: %d\nWords: %d\nLines: %d\n", char\_count, word\_count, line\_count);

return 0;

}

int yywrap() { return 1; }

input

#include <stdio.h>

int main() {

int a, b;

printf("Hello World");

}

17.Lexical Analysis (Identifying Constants using LEX)

%{

#include <stdio.h>

%}

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

%%

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

.|\n ;

%%

int main() {

yylex();

return 0;

}

int yywrap() { return 1; }

input

#define PI 3.14

int a = 30;

float b = 4.5;

18.Lexical Analysis (Counting Macros and Header Files using LEX)

%{

#include <stdio.h>

int macro\_count = 0, header\_count = 0;

%}

%%

^#define { macro\_count++; }

^#include { header\_count++; }

.|\n ;

%%

int main() {

yylex();

printf("Macros: %d\nHeader Files: %d\n", macro\_count, header\_count);

return 0;

}

int yywrap() { return 1; }

input

#define PI 3.14

#include <stdio.h>

#include <conio.h>

19.Lexical Analysis (HTML Tag Recognition using LEX)

%{

#include <stdio.h>

%}

TAG <[^>]+>

%%

{TAG} { printf("HTML Tag: %s\n", yytext); }

.|\n ;

%%

int main() {

yylex();

return 0;

}

int yywrap() { return 1; }

input

<html>

<body>

<h1>Hello World</h1>

</body>

</html>

20.Lexical Analysis (Line Numbering in Source Code using LEX)

%{

#include <stdio.h>

int line\_no = 1;

%}

%%

^.\* { printf("%d %s\n", line\_no++, yytext); }

.|\n ;

%%

int main() {

yylex();

return 0;

}

int yywrap() { return 1; }

input

#include <stdio.h>

void main() {

int a = 30;

printf("Hello");

}