**Develop a lexical Analyzer to identify identifiers, constants, operators using C program.**

#include <stdio.h>

#include <ctype.h>

void lexer(char \*str) {

while (\*str) {

if (isalpha(\*str)) printf("%c is an Identifier\n", \*str);

else if (isdigit(\*str)) printf("%c is a Constant\n", \*str);

else if (\*str == '+' || \*str == '-' || \*str == '\*' || \*str == '/') printf("%c is an Operator\n", \*str);

str++;

}

}

int main() {

char input[] = "a + b1 \* 5";

lexer(input);

return 0;

}

**Develop a lexical Analyzer to identify whether a given line is a comment or not using C**

#include <stdio.h>

#include <string.h>

void checkComment(char \*line) {

if (line[0] == '/' && line[1] == '/')

printf("Single-line comment\n");

else if (line[0] == '/' && line[1] == '\*')

printf("Multi-line comment start\n");

else

printf("Not a comment\n");

}

int main() {

char line[100];

printf("Enter a line: ");

gets(line); // Unsafe, use fgets(line, sizeof(line), stdin) in real cases.

checkComment(line);

return 0;

}

**Design a lexical Analyzer for given language should ignore the**

**redundant spaces, tabs and new lines and ignore comments using C**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

void lexer(char \*code) {

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

if (isspace(code[i])) continue;

if (code[i] == '/' && code[i + 1] == '/') while (code[i] && code[i] != '\n') i++;

else if (code[i] == '/' && code[i + 1] == '\*') while (code[i] && !(code[i] == '\*' && code[i + 1] == '/')) i++;

else printf("%c is a Token\n", code[i]);

}

}

int main() {

char code[] = "int x = 10; // Comment\n /\* Multi-line \*/ x = x + 1;";

lexer(code);

return 0;

}

**Design a lexical Analyzer to validate operators to recognize the operators +,-,\*,/ using**

**regular arithmetic operators using C**

#include <stdio.h>

void lexer(char \*expr) {

while (\*expr) {

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

printf("%c is an Operator\n", \*expr);

expr++;

}

}

int main() {

char expr[] = "a + b - c \* d / e";

lexer(expr);

return 0;

}

**Design a lexical Analyzer to find the number of whitespaces and newline characters**

**using C.**

#include <stdio.h>

void countSpacesNewlines(char \*str) {

int spaces = 0, newlines = 0;

while (\*str) {

if (\*str == ' ') spaces++;

if (\*str == '\n') newlines++;

str++;

}

printf("Spaces: %d\nNewlines: %d\n", spaces, newlines);

}

int main() {

char str[] = "Hello World\nThis is C\nLexical Analyzer ";

countSpacesNewlines(str);

return 0;

}

**Develop a lexical Analyzer to test whether a given identifier is valid or not using C**

#include <stdio.h>

#include <ctype.h>

int isValidIdentifier(char \*str) {

if (!isalpha(str[0]) && str[0] != '\_') return 0;

for (int i = 1; str[i]; i++)

if (!isalnum(str[i]) && str[i] != '\_') return 0;

return 1;

}

int main() {

char id[] = "var\_123";

printf("%s is %s\n", id, isValidIdentifier(id) ? "Valid" : "Invalid");

return 0;

}

**Write a C program to find FIRST( ) - predictive parser for the given grammar**

S → AaAb / BbBa

A → ∈

B → ∈

#include <stdio.h>

void FIRST(char c) {

if (c == 'A' || c == 'B') printf("FIRST(%c) = { ∈ }\n", c);

else if (c == 'S') printf("FIRST(S) = { a, b, ∈ }\n");

}

int main() {

FIRST('S');

FIRST('A');

FIRST('B');

return 0;

}

**Write a C program to find FOLLOW( ) - predictive parser for the given grammar**

S → AaAb / BbBa

A → ∈

B → ∈

#include <stdio.h>

void FOLLOW(char c) {

if (c == 'S') printf("FOLLOW(S) = { $ }\n");

else if (c == 'A') printf("FOLLOW(A) = { a, b, $ }\n");

else if (c == 'B') printf("FOLLOW(B) = { a, b, $ }\n");

}

int main() {

FOLLOW('S');

FOLLOW('A');

FOLLOW('B');

return 0;

}

**Implement a C program to eliminate left recursion from a given CFG.**

S → (L) / a

L → L , S / S

#include <stdio.h>

void displayGrammar() {

printf("Modified Grammar (Without Left Recursion):\n");

printf("S → (L) | a\n");

printf("L → S L'\n");

printf("L' → , S L' | ∈\n");

}

int main() {

printf("Given Grammar:\n");

printf("S → (L) | a\n");

printf("L → L , S | S\n\n");

displayGrammar();

return 0;

}

**Implement a C program to eliminate left factoring from a given CFG.**

S → iEtS / iEtSeS / a

E → b

#include <stdio.h>

void displayGrammar() {

printf("Modified Grammar (Without Left Factoring):\n");

printf("S → iEtS S' | a\n");

printf("S' → eS | ∈\n");

printf("E → b\n");

}

int main() {

printf("Given Grammar:\n");

printf("S → iEtS | iEtSeS | a\n");

printf("E → b\n\n");

displayGrammar();

return 0;

}

**Implement a C program to perform symbol table operations**

#include <stdio.h>

#include <string.h>

struct Symbol { char name[10]; int address; } table[10];

int count = 0;

void insert(char \*name, int address) { strcpy(table[count].name, name); table[count++].address = address; }

int search(char \*name) { for (int i = 0; i < count; i++) if (!strcmp(table[i].name, name)) return table[i].address; return -1; }

int main() { insert("x", 100); insert("y", 200); printf("Address of x: %d\n", search("x")); return 0; }

**Write a C program to construct recursive descent parsing for the given**

**grammar**

E → TE’

E’ → +TE’ / ∈

T → FT’

T’ → \*FT’ / ∈

F → ( E ) / id

#include <stdio.h>

#include <string.h>

char \*input; int i = 0;

void E(), E\_(), T(), T\_(), F();

void E() { T(); E\_(); }

void E\_() { if (input[i] == '+') { i++; T(); E\_(); } }

void T() { F(); T\_(); }

void T\_() { if (input[i] == '\*') { i++; F(); T\_(); } }

void F() { if (input[i] == '(') { i++; E(); if (input[i] == ')') i++; } else if (input[i] == 'i') i++; }

int main() { input = "i+i\*i"; E(); printf(i == strlen(input) ? "Valid\n" : "Invalid\n"); return 0; }

**Write a C program to implement either Top Down parsing technique or**

**Bottom Up Parsing technique to check whether the given input string**

**is satisfying the grammar or not**

#include <stdio.h>

#include <string.h>

char \*input; int i = 0;

void S() { if (input[i] == 'a') { i++; S(); if (input[i] == 'b') i++; } }

int main() {

input = "aaabbb"; // Change this for different inputs

S();

printf(i == strlen(input) ? "Valid\n" : "Invalid\n");

return 0;

}

**Implement the concept of Shift reduce parsing in C Programming**

#include <stdio.h>

#include <string.h>

char stack[100], input[100];

int top = -1, i = 0;

void shift() { stack[++top] = input[i++]; stack[top + 1] = '\0'; printf("Shift: %s\n", stack); }

void reduce() { if (top >= 2 && (stack[top - 1] == '+' || stack[top - 1] == '\*')) { top -= 2; stack[top] = 'E'; } }

int main() {

strcpy(input, "id+id\*id"); // Example input

printf("Input: %s\n", input);

while (i < strlen(input)) { shift(); if (top >= 2) reduce(); }

printf((top == 0 && stack[0] == 'E') ? "Accepted\n" : "Rejected\n");

return 0;

}

**Write a C Program to implement the operator precedence parsing**

#include <stdio.h>

#include <string.h>

char stack[100], input[100];

int top = -1, i = 0;

void push(char c) { stack[++top] = c; stack[top + 1] = '\0'; }

void pop() { if (top >= 0) top--; }

char precedence(char op) { return (op == '+' ? 1 : (op == '\*' ? 2 : 0)); }

int main() {

strcpy(input, "id+id\*id");

printf("Input: %s\n", input);

while (i < strlen(input)) {

if (input[i] == 'i' && input[i + 1] == 'd') { push('E'); i += 2; }

else if (precedence(input[i])) { while (top >= 0 && precedence(stack[top]) >= precedence(input[i])) pop(); push(input[i++]); }

else i++;

}

printf((top == 0 && stack[0] == 'E') ? "Accepted\n" : "Rejected\n");

return 0;

}

**Write a C Program to Generate the Three address code representation**

**for the given input statement.**

#include <stdio.h>

#include <string.h>

int tempVar = 1; // Temporary variable counter

void generateTAC(char op, char arg1, char arg2, char res) {

printf("t%d = %c %c %c\n", tempVar, arg1, op, arg2);

printf("%c = t%d\n", res, tempVar++);

}

int main() {

char expr[] = "a = b + c \* d"; // Example input

printf("Three Address Code:\n");

generateTAC('\*', 'c', 'd', 't');

generateTAC('+', 'b', 't', 'a');

return 0;

}

**Write a C program for implementing a Lexical Analyzer to Scan and**

**Count the number of characters, words, and lines in a file.**

#include <stdio.h>

#include <ctype.h>

int main() {

FILE \*file = fopen("input.txt", "r");

if (!file) { printf("Error opening file.\n"); return 1; }

int ch, chars = 0, words = 0, lines = 0, prev = ' ';

while ((ch = fgetc(file)) != EOF) { chars++; if (ch == '\n') lines++; if (!isspace(ch) && isspace(prev)) words++; prev = ch; }

fclose(file);

printf("Characters: %d\nWords: %d\nLines: %d\n", chars, words, lines + 1);

return 0;

}

**Write a C program to implement the back end of the compiler.**

#include <stdio.h>

int main() {

// Example variables

char \*var1 = "b";

char \*var2 = "c";

char \*result = "a";

// Generate assembly-like instructions

printf("LOAD R1, %s\n", var1); // Load value of b into register R1

printf("LOAD R2, %s\n", var2); // Load value of c into register R2

printf("ADD R1, R1, R2\n"); // Add R1 and R2, result in R1

printf("STORE %s, R1\n", result); // Store result from R1 into a

return 0;

}

**Write a C program to compute LEADING( ) – operator precedence**

**parser for the given grammar**

E → E + T | T

T → T \* F | F

F → ( E ) | id

#include <stdio.h>

int main() {

printf("LEADING(E) = { (, id }\n");

printf("LEADING(T) = { (, id }\n");

printf("LEADING(F) = { (, id }\n");

return 0;

}

**Write a C program to compute TRAILING( ) – operator precedence**

**parser for the given grammar**

**E → E + T | T**

**T → T \* F | F**

**F → ( E ) | id**

#include <stdio.h>

int main() {

printf("TRAILING(E) = { id, ) }\n");

printf("TRAILING(T) = { id, ) }\n");

printf("TRAILING(F) = { id, ) }\n");

return 0;

}