**Read text from user**

#include <stdio.h>

#include <ctype.h>

#include <stdbool.h>

// Token types enumeration

enum TokenType {

KEYWORD,

IDENTIFIER,

NUMBER,

OPERATOR,

SPECIAL\_SYMBOL,

DELIMITER,

WHITESPACE,

ERROR

};

// Structure to represent a token

struct Token {

enum TokenType type;

char lexeme[100];

};

// Function to classify a character as an operator

int isOperator(char ch) {

char operators[] = "+-\*/%";

for (int i = 0; i < sizeof(operators); i++) {

if (ch == operators[i]) {

return 1; // It is an operator

}

}

return 0; // It is not an operator

}

// Function to classify a character as a special symbol

int isSpecialSymbol(char ch) {

char specialSymbols[] = "!@#$%^&\*()\_-+=<>,./?;:'\"[]{}\\|`~";

for (int i = 0; i < sizeof(specialSymbols); i++) {

if (ch == specialSymbols[i]) {

return 1; // It is a special symbol

}

}

return 0; // It is not a special symbol

}

// Function to classify a character as a delimiter

int isDelimiter(char ch) {

char delimiters[] = " \t\n\r\f\v";

for (int i = 0; i < sizeof(delimiters); i++) {

if (ch == delimiters[i]) {

return 1; // It is a delimiter

}

}

return 0; // It is not a delimiter

}

// Function to perform lexical analysis

struct Token lexer(char input[]) {

struct Token token;

int i = 0;

// Skip leading whitespaces

while (isDelimiter(input[i])) {

i++;

}

// Check for end of input

if (input[i] == '\0') {

token.type = ERROR;

strcpy(token.lexeme, "End of input");

return token;

}

// Check for identifiers or keywords

if (isalpha(input[i]) || input[i] == '\_') {

int j = 0;

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

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

}

token.lexeme[j] = '\0';

// Check if it is a keyword

if (strcmp(token.lexeme, "if") == 0 || strcmp(token.lexeme, "else") == 0 ||

strcmp(token.lexeme, "int") == 0 || strcmp(token.lexeme, "float") == 0) {

token.type = KEYWORD;

} else {

token.type = IDENTIFIER;

}

return token;

}

// Check for numbers

if (isdigit(input[i]) || (input[i] == '.' && isdigit(input[i + 1]))) {

token.type = NUMBER;

int j = 0;

while (isdigit(input[i]) || (input[i] == '.' && isdigit(input[i + 1]))) {

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

}

token.lexeme[j] = '\0';

return token;

}

// Check for operators

if (isOperator(input[i])) {

token.type = OPERATOR;

int j = 0;

while (isOperator(input[i])) {

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

}

token.lexeme[j] = '\0';

return token;

}

// Check for special symbols

if (isSpecialSymbol(input[i])) {

token.type = SPECIAL\_SYMBOL;

int j = 0;

while (isSpecialSymbol(input[i])) {

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

}

token.lexeme[j] = '\0';

return token;

}

// Check for delimiters

if (isDelimiter(input[i])) {

token.type = DELIMITER;

int j = 0;

while (isDelimiter(input[i])) {

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

}

token.lexeme[j] = '\0';

return token;

}

// If none of the above, it's an error

token.type = ERROR;

strcpy(token.lexeme, "Invalid character");

return token;

}

// Function to get the string representation of token types

const char \*getTokenTypeName(enum TokenType type) {

switch (type) {

case KEYWORD:

return "Keyword";

case IDENTIFIER:

return "Identifier";

case NUMBER:

return "Number";

case OPERATOR:

return "Operator";

case SPECIAL\_SYMBOL:

return "Special Symbol";

case DELIMITER:

return "Delimiter";

case WHITESPACE:

return "Whitespace";

case ERROR:

return "Error";

default:

return "Unknown";

}

}

int main() {

char input[100];

// Get input from the user

printf("Enter the input text:\n");

fgets(input, sizeof(input), stdin);

// Perform lexical analysis

struct Token token;

int index = 0;

while ((token = lexer(input + index)).type != ERROR) {

// Print the token information, excluding whitespace

if (token.type != WHITESPACE) {

printf("Type: %s, Lexeme: %s\n", getTokenTypeName(token.type), token.lexeme);

}

// Move to the next token

index += strlen(token.lexeme);

}

return 0;

}

**Read a file –**

#include <stdio.h>

#include <ctype.h>

#include <stdbool.h>

// Token types enumeration

enum TokenType {

KEYWORD,

IDENTIFIER,

NUMBER,

OPERATOR,

SPECIAL\_SYMBOL,

DELIMITER,

WHITESPACE,

ERROR

};

// Structure to represent a token

struct Token {

enum TokenType type;

char lexeme[100];

};

// Function to classify a character as an operator

int isOperator(char ch) {

char operators[] = "+-\*/%";

for (int i = 0; i < sizeof(operators); i++) {

if (ch == operators[i]) {

return 1; // It is an operator

}

}

return 0; // It is not an operator

}

// Function to classify a character as a special symbol

int isSpecialSymbol(char ch) {

char specialSymbols[] = "!@#$%^&\*()\_-+=<>,./?;:'\"[]{}\\|`~";

for (int i = 0; i < sizeof(specialSymbols); i++) {

if (ch == specialSymbols[i]) {

return 1; // It is a special symbol

}

}

return 0; // It is not a special symbol

}

// Function to classify a character as a delimiter

int isDelimiter(char ch) {

char delimiters[] = " \t\n\r\f\v";

for (int i = 0; i < sizeof(delimiters); i++) {

if (ch == delimiters[i]) {

return 1; // It is a delimiter

}

}

return 0; // It is not a delimiter

}

// Function to perform lexical analysis

struct Token lexer(char input[]) {

struct Token token;

int i = 0;

// Skip leading whitespaces

while (isDelimiter(input[i])) {

i++;

}

// Check for end of input

if (input[i] == '\0') {

token.type = ERROR;

strcpy(token.lexeme, "End of input");

return token;

}

// Check for identifiers or keywords

if (isalpha(input[i]) || input[i] == '\_') {

int j = 0;

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

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

}

token.lexeme[j] = '\0';

// Check if it is a keyword

if (strcmp(token.lexeme, "if") == 0 || strcmp(token.lexeme, "else") == 0 ||

strcmp(token.lexeme, "int") == 0 || strcmp(token.lexeme, "float") == 0) {

token.type = KEYWORD;

} else {

token.type = IDENTIFIER;

}

return token;

}

// Check for numbers

if (isdigit(input[i]) || (input[i] == '.' && isdigit(input[i + 1]))) {

token.type = NUMBER;

int j = 0;

while (isdigit(input[i]) || (input[i] == '.' && isdigit(input[i + 1]))) {

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

}

token.lexeme[j] = '\0';

return token;

}

// Check for operators

if (isOperator(input[i])) {

token.type = OPERATOR;

int j = 0;

while (isOperator(input[i])) {

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

}

token.lexeme[j] = '\0';

return token;

}

// Check for special symbols

if (isSpecialSymbol(input[i])) {

token.type = SPECIAL\_SYMBOL;

int j = 0;

while (isSpecialSymbol(input[i])) {

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

}

token.lexeme[j] = '\0';

return token;

}

// Check for delimiters

if (isDelimiter(input[i])) {

token.type = DELIMITER;

int j = 0;

while (isDelimiter(input[i])) {

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

}

token.lexeme[j] = '\0';

return token;

}

// If none of the above, it's an error

token.type = ERROR;

strcpy(token.lexeme, "Invalid character");

return token;

}

// Function to get the string representation of token types

const char \*getTokenTypeName(enum TokenType type) {

switch (type) {

case KEYWORD:

return "Keyword";

case IDENTIFIER:

return "Identifier";

case NUMBER:

return "Number";

case OPERATOR:

return "Operator";

case SPECIAL\_SYMBOL:

return "Special Symbol";

case DELIMITER:

return "Delimiter";

case WHITESPACE:

return "Whitespace";

case ERROR:

return "Error";

default:

return "Unknown";

}

}

int main() {

FILE \*file;

char filename[100];

char input[1000];

// Get the filename from the user

printf("Enter the filename: ");

scanf("%s", filename);

// Open the file

file = fopen(filename, "r");

if (file == NULL) {

printf("Error opening file. Exiting...\n");

return 1;

}

// Read the content of the file into the input buffer

fgets(input, sizeof(input), file);

// Perform lexical analysis

struct Token token;

int index = 0;

while ((token = lexer(input + index)).type != ERROR) {

// Print the token information, excluding whitespace

if (token.type != WHITESPACE) {

printf("Type: %s, Lexeme: %s\n", getTokenTypeName(token.type), token.lexeme);

}

// Move to the next token

index += strlen(token.lexeme);

}

// Close the file

fclose(file);

return 0;

}

**Lex program for lexical analyser –**

%{

#include <stdio.h>

#include <string.h>

%}

%option noyywrap

%{

// Define token types

enum TokenType {

KEYWORD,

IDENTIFIER,

NUMBER,

OPERATOR,

SPECIAL\_SYMBOL,

DELIMITER,

WHITESPACE

};

// Function to print token information

void printToken(enum TokenType type, char\* lexeme) {

printf("Type: %d, Lexeme: %s\n", type, lexeme);

}

%}

%%

"if"|"else"|"int"|"float" {

printToken(KEYWORD, yytext);

}

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

printToken(IDENTIFIER, yytext);

}

[0-9]+ {

printToken(NUMBER, yytext);

}

[-+\*/%] {

printToken(OPERATOR, yytext);

}

[!@#$%^&\*()\_\-+=<>,./?;:'\"[\]{}\\|`~] {

printToken(SPECIAL\_SYMBOL, yytext);

}

[ \t\n\r\f\v]+ {

printToken(WHITESPACE, yytext);

}

. {

printToken(DELIMITER, yytext);

}

%%

int main() {

yylex();

return 0;

}

lex lexer.l

gcc lex.yy.c -o lexer -ll

./lexer

**C program to match files % -**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Token types enumeration

enum TokenType {

KEYWORD,

IDENTIFIER,

NUMBER,

OPERATOR,

SPECIAL\_SYMBOL,

DELIMITER,

WHITESPACE,

ERROR

};

// Structure to represent a token

struct Token {

enum TokenType type;

char lexeme[100];

};

// Function to classify a character as an operator

int isOperator(char ch) {

char operators[] = "+-\*/%";

for (int i = 0; i < sizeof(operators); i++) {

if (ch == operators[i]) {

return 1; // It is an operator

}

}

return 0; // It is not an operator

}

// Function to classify a character as a special symbol

int isSpecialSymbol(char ch) {

char specialSymbols[] = "!@#$%^&\*()\_-+=<>,./?;:'\"[]{}\\|`~";

for (int i = 0; i < sizeof(specialSymbols); i++) {

if (ch == specialSymbols[i]) {

return 1; // It is a special symbol

}

}

return 0; // It is not a special symbol

}

// Function to classify a character as a delimiter

int isDelimiter(char ch) {

char delimiters[] = " \t\n\r\f\v";

for (int i = 0; i < sizeof(delimiters); i++) {

if (ch == delimiters[i]) {

return 1; // It is a delimiter

}

}

return 0; // It is not a delimiter

}

// Function to perform lexical analysis

struct Token lexer(char input[]) {

struct Token token;

int i = 0;

// Skip leading whitespaces

while (isDelimiter(input[i])) {

i++;

}

// Check for end of input

if (input[i] == '\0') {

token.type = ERROR;

strcpy(token.lexeme, "End of input");

return token;

}

// Check for identifiers or keywords

if (isalpha(input[i]) || input[i] == '\_') {

int j = 0;

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

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

}

token.lexeme[j] = '\0';

// Check if it is a keyword

if (strcmp(token.lexeme, "if") == 0 || strcmp(token.lexeme, "else") == 0 ||

strcmp(token.lexeme, "int") == 0 || strcmp(token.lexeme, "float") == 0) {

token.type = KEYWORD;

} else {

token.type = IDENTIFIER;

}

return token;

}

// Check for numbers

if (isdigit(input[i]) || (input[i] == '.' && isdigit(input[i + 1]))) {

token.type = NUMBER;

int j = 0;

while (isdigit(input[i]) || (input[i] == '.' && isdigit(input[i + 1]))) {

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

}

token.lexeme[j] = '\0';

return token;

}

// Check for operators

if (isOperator(input[i])) {

token.type = OPERATOR;

int j = 0;

while (isOperator(input[i])) {

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

}

token.lexeme[j] = '\0';

return token;

}

// Check for special symbols

if (isSpecialSymbol(input[i])) {

token.type = SPECIAL\_SYMBOL;

int j = 0;

while (isSpecialSymbol(input[i])) {

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

}

token.lexeme[j] = '\0';

return token;

}

// Check for delimiters

if (isDelimiter(input[i])) {

token.type = DELIMITER;

int j = 0;

while (isDelimiter(input[i])) {

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

}

token.lexeme[j] = '\0';

return token;

}

// If none of the above, it's an error

token.type = ERROR;

strcpy(token.lexeme, "Invalid character");

return token;

}

// Function to calculate the percentage of matching tokens

float calculateMatchingPercentage(struct Token tokens1[], int size1, struct Token tokens2[], int size2) {

int matchingCount = 0;

for (int i = 0; i < size1 && i < size2; i++) {

if (tokens1[i].type == tokens2[i].type && strcmp(tokens1[i].lexeme, tokens2[i].lexeme) == 0) {

matchingCount++;

}

}

float percentage = (float)matchingCount / (float)(size1 > size2 ? size1 : size2) \* 100.0;

return percentage;

}

int main() {

FILE \*file1, \*file2;

char filename1[100], filename2[100];

char input1[1000], input2[1000];

// Get the filenames from the user

printf("Enter the first filename: ");

scanf("%s", filename1);

// Open the first file

file1 = fopen(filename1, "r");

if (file1 == NULL) {

printf("Error opening file %s. Exiting...\n", filename1);

return 1;

}

// Read the content of the first file into the input buffer

fgets(input1, sizeof(input1), file1);

// Close the first file

fclose(file1);

// Get the second filename from the user

printf("Enter the second filename: ");

scanf("%s", filename2);

// Open the second file

file2 = fopen(filename2, "r");

if (file2 == NULL) {

printf("Error opening file %s. Exiting...\n", filename2);

return 1;

}

// Read the content of the second file into the input buffer

fgets(input2, sizeof(input2), file2);

// Close the second file

fclose(file2);

// Perform lexical analysis on both inputs

struct Token tokens1[1000], tokens2[1000];

int size1 = 0, size2 = 0;

int index1 = 0, index2 = 0;

while ((tokens1[size1] = lexer(input1 + index1)).type != ERROR) {

size1++;

index1 += strlen(tokens1[size1 - 1].lexeme);

}

while ((tokens2[size2] = lexer(input2 + index2)).type != ERROR) {

size2++;

index2 += strlen(tokens2[size2 - 1].lexeme);

}

// Calculate and print the percentage of matching tokens

float percentage = calculateMatchingPercentage(tokens1, size1, tokens2, size2);

printf("Percentage of matching tokens: %.2f%%\n", percentage);

return 0;

}

**Lex program to match files –**

%{

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

%}

%option noyywrap

%{

// Token types enumeration

enum TokenType {

KEYWORD,

IDENTIFIER,

NUMBER,

OPERATOR,

SPECIAL\_SYMBOL,

DELIMITER,

WHITESPACE,

ERROR

};

// Structure to represent a token

struct Token {

enum TokenType type;

char lexeme[100];

};

// Function to calculate the percentage of matching tokens

float calculateMatchingPercentage(struct Token tokens1[], int size1, struct Token tokens2[], int size2) {

int matchingCount = 0;

for (int i = 0; i < size1 && i < size2; i++) {

if (tokens1[i].type == tokens2[i].type && strcmp(tokens1[i].lexeme, tokens2[i].lexeme) == 0) {

matchingCount++;

}

}

float percentage = (float)matchingCount / (float)(size1 > size2 ? size1 : size2) \* 100.0;

return percentage;

}

%}

%%

"if"|"else"|"int"|"float" {

printf("Type: KEYWORD, Lexeme: %s\n", yytext);

}

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

printf("Type: IDENTIFIER, Lexeme: %s\n", yytext);

}

[0-9]+ {

printf("Type: NUMBER, Lexeme: %s\n", yytext);

}

[-+\*/%] {

printf("Type: OPERATOR, Lexeme: %s\n", yytext);

}

[!@#$%^&\*()\_\-+=<>,./?;:'\"[\]{}\\|`~] {

printf("Type: SPECIAL\_SYMBOL, Lexeme: %s\n", yytext);

}

[ \t\n\r\f\v]+ {

printf("Type: WHITESPACE, Lexeme: %s\n", yytext);

}

. {

printf("Type: DELIMITER, Lexeme: %s\n", yytext);

}

%%

int main() {

FILE \*file1, \*file2;

char filename1[100], filename2[100];

// Get the filenames from the user

printf("Enter the first filename: ");

scanf("%s", filename1);

// Open the first file

file1 = fopen(filename1, "r");

if (file1 == NULL) {

printf("Error opening file %s. Exiting...\n", filename1);

return 1;

}

// Set file1 as the input for Lex

yyin = file1;

// Perform lexical analysis on file1

yylex();

// Close the first file

fclose(file1);

// Get the second filename from the user

printf("Enter the second filename: ");

scanf("%s", filename2);

// Open the second file

file2 = fopen(filename2, "r");

if (file2 == NULL) {

printf("Error opening file %s. Exiting...\n", filename2);

return 1;

}

// Set file2 as the input for Lex

yyin = file2;

// Perform lexical analysis on file2

yylex();

// Close the second file

fclose(file2);

return 0;

}

lex lexer.l

gcc lex.yy.c -o lexer -ll

./lexer