**Name**: D. Ashwanth Ram

**Reg No**: 3122 21 5001 013

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

**UCS2702 - Compiler Design Programming Assignment-1** -------------------------------------------------------------------------------------------------------------------------------

**Aim:**

Implementation of lexical analyser and symbol table

**Code:**

**Main.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_TOKENS 100

#define MAX\_LEXEME\_LEN 50

// Token structure

typedef struct {

char lexeme[MAX\_LEXEME\_LEN];

char type[30];

} Token;

// Symbol table entry structure

typedef struct {

char identifier[MAX\_LEXEME\_LEN];

char type[10];

int no\_of\_bytes;

int address;

int value;

} SymbolTableEntry;

Token tokens[MAX\_TOKENS];

int token\_count = 0;

SymbolTableEntry symbol\_table[MAX\_TOKENS];

int symbol\_count = 0;

void addToken(const char \*lexeme, const char \*type) {

if (token\_count < MAX\_TOKENS) {

strncpy(tokens[token\_count].lexeme, lexeme, MAX\_LEXEME\_LEN - 1);

tokens[token\_count].lexeme[MAX\_LEXEME\_LEN - 1] = '\0';

strncpy(tokens[token\_count].type, type, 29);

tokens[token\_count].type[29] = '\0';

token\_count++;

}

}

void addSymbol(const char \*identifier, const char \*type, int no\_of\_bytes, int address, int value) {

if (symbol\_count < MAX\_TOKENS) {

strncpy(symbol\_table[symbol\_count].identifier, identifier, MAX\_LEXEME\_LEN - 1);

symbol\_table[symbol\_count].identifier[MAX\_LEXEME\_LEN - 1] = '\0';

strncpy(symbol\_table[symbol\_count].type, type, 9);

symbol\_table[symbol\_count].type[9] = '\0';

symbol\_table[symbol\_count].no\_of\_bytes = no\_of\_bytes;

symbol\_table[symbol\_count].address = address;

symbol\_table[symbol\_count].value = value;

symbol\_count++;

}

}

const char \*keywords[] = {

"auto", "break", "case", "char", "const", "continue", "default", "do", "double", "else", "enum",

"extern", "float", "for", "goto", "if", "inline", "int", "long", "register", "restrict", "return",

"short", "signed", "sizeof", "static", "struct", "switch", "typedef", "union", "unsigned", "void",

"volatile", "while", "\_Alignas", "\_Alignof", "\_Atomic", "\_Bool", "\_Complex", "\_Generic", "\_Imaginary",

"\_Noreturn", "\_Static\_assert", "\_Thread\_local"

};

const char \*operators[] = {

"+", "-", "\*", "/", "%", "++", "--", "==", "!=", ">", "<", ">=", "<=", "&&", "||", "!", "&", "|", "^",

"~", "<<", ">>", "=", "+=", "-=", "\*=", "/=", "%=", "&=", "|=", "^=", "<<=", ">>=", "->", ".", "?"

};

const char \*specials[] = {

"{", "}", "[", "]", "(", ")", ",", ";", ":", ".", "&", "\*", "+", "-", "~", "!", "/", "%", "<", ">", "^",

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

"++", "--", "->", "."

};

void tokenize(const char \*code) {

const char \*ptr = code;

char buffer[MAX\_LEXEME\_LEN];

int i = 0;

while (\*ptr) {

if (isspace(\*ptr)) {

ptr++;

continue;

}

if (\*ptr == '#') {

while (\*ptr && \*ptr != '\n') {

buffer[i++] = \*ptr++;

}

buffer[i] = '\0';

addToken(buffer, "preprocessor directive");

i = 0;

continue;

}

if (isalpha(\*ptr) || \*ptr == '\_') {

while (isalnum(\*ptr) || \*ptr == '\_') {

buffer[i++] = \*ptr++;

}

buffer[i] = '\0';

int is\_keyword = 0;

for (int j = 0; j < sizeof(keywords) / sizeof(keywords[0]); j++) {

if (strcmp(buffer, keywords[j]) == 0) {

addToken(buffer, "keyword");

is\_keyword = 1;

break;

}

}

if (!is\_keyword) {

addToken(buffer, strcmp(buffer, "printf") == 0 ? "function call" : "identifier");

}

i = 0;

continue;

}

if (isdigit(\*ptr)) {

while (isdigit(\*ptr)) {

buffer[i++] = \*ptr++;

}

buffer[i] = '\0';

addToken(buffer, "integer constant");

i = 0;

continue;

}

int is\_operator = 0;

for (int j = 0; j < sizeof(operators) / sizeof(operators[0]); j++) {

if (strncmp(ptr, operators[j], strlen(operators[j])) == 0) {

addToken(operators[j], "operator");

ptr += strlen(operators[j]);

is\_operator = 1;

break;

}

}

if (is\_operator) continue;

int is\_special = 0;

for (int j = 0; j < sizeof(specials) / sizeof(specials[0]); j++) {

if (strncmp(ptr, specials[j], strlen(specials[j])) == 0) {

addToken(specials[j], "special character");

ptr += strlen(specials[j]);

is\_special = 1;

break;

}

}

if (is\_special) continue;

if (\*ptr == '\"') {

buffer[i++] = \*ptr++;

while (\*ptr && \*ptr != '\"') {

buffer[i++] = \*ptr++;

}

if (\*ptr == '\"') {

buffer[i++] = \*ptr++;

}

buffer[i] = '\0';

addToken(buffer, "string literal");

i = 0;

continue;

}

ptr++;

}

}

void buildSymbolTable() {

int address\_counter = 1000;

for (int i = 0; i < token\_count; i++) {

if (strcmp(tokens[i].type, "keyword") == 0 && strcmp(tokens[i].lexeme, "int") == 0) {

while (i < token\_count && strcmp(tokens[i].lexeme, ";") != 0) {

if (strcmp(tokens[i].type, "identifier") == 0) {

char \*identifier = tokens[i].lexeme;

int value = 0;

if (i + 1 < token\_count && strcmp(tokens[i + 1].lexeme, "=") == 0) {

value = atoi(tokens[i + 2].lexeme);

i += 2;

}

addSymbol(identifier, "int", 2, address\_counter, value);

address\_counter += 2;

}

i++;

}

}

}

}

void printTokens() {

for (int i = 0; i < token\_count; i++) {

printf("%s - %s\n", tokens[i].lexeme, tokens[i].type);

}

}

void printSymbolTable() {

printf("Content of Symbol Table\n");

printf("--------------------------------------------------\n");

printf("|Identifier| Type | No of bytes | Address | Value |\n");

printf("--------------------------------------------------\n");

for (int i = 0; i < symbol\_count; i++) {

printf("| %-9s | %-4s | %-11d | %-7d | %-5d |\n",

symbol\_table[i].identifier,

symbol\_table[i].type,

symbol\_table[i].no\_of\_bytes,

symbol\_table[i].address,

symbol\_table[i].value);

}

printf("--------------------------------------------------\n");

}

int main() {

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

if (file == NULL) {

perror("Error opening file");

return EXIT\_FAILURE;

}

fseek(file, 0, SEEK\_END);

long length = ftell(file);

fseek(file, 0, SEEK\_SET);

char \*code = malloc(length + 1);

if (code == NULL) {

perror("Error allocating memory");

fclose(file);

return EXIT\_FAILURE;

}

fread(code, 1, length, file);

code[length] = '\0';

fclose(file);

tokenize(code);

buildSymbolTable();

printTokens();

printSymbolTable();

free(code);

return 0;

}

**Input.c**

#include <stdio.h>

main()

{

int a = 10, b = 20;

if (a > b)

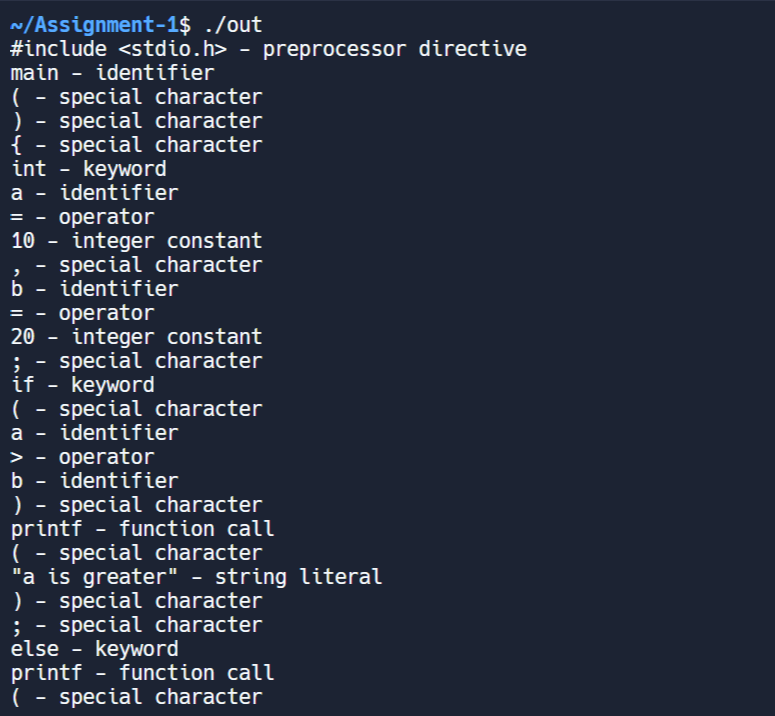
printf("a is greater");

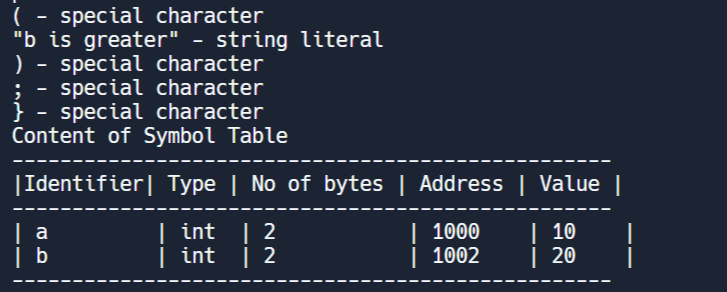
else

printf("b is greater");

}

**Output:**





**Learning Outcome**:

1. **Tokenization and Lexical Analysis:** Learn how source code is broken into tokens, identifying keywords, operators, and other elements.
2. **Building a Symbol Table:** Understand how to store and manage information about variables, functions, and other symbols.
3. **Memory Management and File I/O:** Learn dynamic memory allocation and reading source code from a file using C.