UCS 1602 - Compiler Design

Exercise 1 - Implementation of lexical analyser and symbol table

Name:	Mahesh Bharadwaj K
Reg No:	185001089
Semester:	VI
Date:	February 8, 2021

Aim:

To create lexical analyser and generate symbol table using C programming.

Program

1. Header file

```
#define MAX 128

typedef enum token_type
{
    INVALID = 0,
    KEYWORD,
    IDENTIFIER,
    COMMENT,
    INT_CONST,
    FLOAT_CONST,
    CHAR_CONST,
    ARITH_OP,
    ARTITH_ASSIGN_OP,
    LOGIC_OP,
    REL_OP,
    BIT_OP,
    UNARY_OP,
    ASSIGN_OP,
    SPECIAL,
    FUNCTION,
    PRE_PROCESSOR,
    WHITESPACE
} token_type;

const char keywords[MAX][30] = {"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",
↪   "\0"};

const char arith_op[MAX][30] = {"+", "-", "*", "/", "%", "\0"};

const char arith_assign_op[MAX][30] = {"+=", "-=", "*=", "/=", "%=", "\0"};

const char logic_op[MAX][30] = {"&&", "||", "!", "\0"};

const char rel_op[MAX][30] = {"<", "<=", ">", ">=", "==", "!=", "\0"};
const char bit_op[MAX][30] = {"^", "&", "|", "<<", ">>", "\0"};

const char unary_op[MAX][30] = {"-", "++", "--", "\0"};

const char assign_op[MAX][30] = {"=", "\0"};

const char special[MAX][30] = {";", ".", "[", "]", "(", ")", "{", "}", "[", "]", ",",
↪  "\0"};

const char data_type[MAX][30] = {"char", "int", "float", "double", "\0"};

typedef struct entry
{
    int sno;
    char id[30];
    char type[30];
    int addr;
    char val[30];
} entry;

entry symbol_table[MAX] = {0};

int check_identifier_table(char *const id, entry symbol_table[])
{
    int found = 0;
    for (int i = 0; i < MAX && symbol_table[i].sno; i++)
    {
        if (strcmp(symbol_table[i].id, id) == 0)
        {
            found = 1;
            break;
        }
    }
    return found;
}

void distab(entry s[], int len)
{
    printf("\n+----------------+----------------+----------------+----------------+");
    printf("\n\tName\t\tType\t\tAddress\t\t Value");

    ↪  printf("\n+----------------+----------------+----------------+----------------+\n");
    for (int i = 0; i < len; i++)
    {
        printf("\t%s\t\t%s\t\t%d\t\t%s\n", s[i].id, s[i].type, s[i].addr, s[i].val);
    }
}

int check(const char *const token, const char array[][30])
{
    int found = 0;

    for (int i = 0; i < MAX && array[i][0]; i++)
    {
        if (strcmp(token, array[i]) == 0)
        {
            found = 1;
            break;
        }
    }
    return found;
}

int check_real_const(const char *token)
{
    int count = 0, digit_found = 0;
    int i = (token[0] == '-' || token[0] == '+') ? 1 : 0;
    for (; token[i]; i++)
    {
        if (isdigit(token[i]))
        {
            digit_found = 1;
            continue;
        }
        if (token[i] == '.')
            count = 1;
        else
            return -1;
    }
    if (count > 1 || !digit_found)
        return -1;
    return count;
}

int check_char_const(const char *token)
{
    if (token[0] != '\"')
        return 0;
    int count = 0;
    for (int i = 0; token[i]; i++)
        if (token[i] == '\"')
            count++;
    return (count == 2);
}

int check_indentifier(const char *token)
{
    if (isdigit(token[0]))
        return 0;
    for (int i = 0; token[i]; i++)
    {
        if (!(isalnum(token[i]) || token[i] == '_'))
            return 0;
    }
    return 1;
}

int check_pre_processor(const char *line)
{
    if (line[0] == '#')
        return 1;
    return 0;
}

int check_function(const char *token)
{
    char test[128] = {0};
    int idx = 0, i = 0, found = 0;
    while (token[idx])
    {
        if (token[idx] != '(')
        {
            test[i++] = token[idx++];
            found = 1;
        }
        else
        {
            test[i] = 0;
            break;
        }
    }
    if (found == 1)
    {
        while (token[idx])
        {
            if (token[idx] == ')')
            {
                found = 2;
                break;
            }
            idx++;
        }
        if (check_indentifier(test) && found == 2)
            return 1;
    }
    return 0;
}

int is_delimiter(const char c)
{
    for (int i = 0; special[i][0]; i++)
        if (special[i][0] == '.')
            continue;
        else if (c == special[i][0])
            return 1;
        else
            ;
    if (c == ' ' || c == '\n' || c == '\t')
        return 1;
    return 0;
}
```

2. Main Program

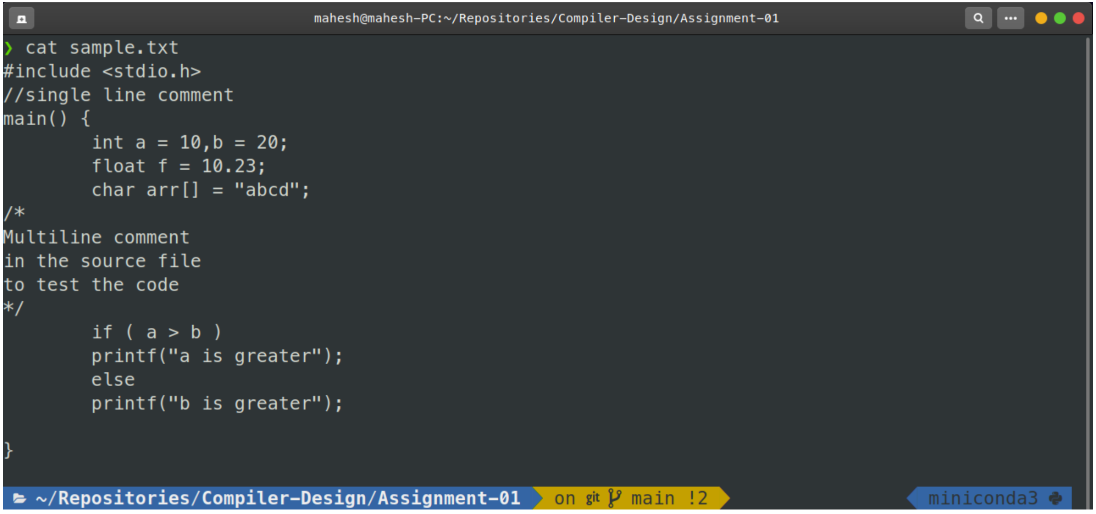
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include "lexer.h"

void remove_newline(char *const string)
{
    int i = 0;
    for (; string[i]; i++)
        if (string[i] == '\n')
            string[i] = 0;
    string[i] = 0;
}
```

```
int symtab_index = 0;
int start_sno = 1;
int start_addr = 1000;
char buffer_id[MAX] = {0};
char buffer_val[MAX] = {0};
char buffer_type[MAX] = {0};

token_type identify_token(const char *const token)
{
    int tmp;
    token_type type;
    if (token[0] == ' ' || token[0] == '\t' || token[0] == '\n')
        type = WHITESPACE;
    else if (check(token, keywords))
    {
        type = KEYWORD;
        for (int i = 0; i < MAX && data_type[i][0]; i++)
        {
            if (strcmp(token, data_type[i]) == 0)
            {
                strcpy(buffer_type, data_type[i]);
                break;
            }
        }
    }
    else if (check_char_const(token))
    {
        type = CHAR_CONST;
        strcpy(buffer_val, token);
        if (!check_identifier_table(buffer_id, symbol_table))
        {
            symbol_table[symtab_index].sno = start_sno++;
            strcpy(symbol_table[symtab_index].id, buffer_id);
            strcpy(symbol_table[symtab_index].val, buffer_val);
            strcpy(symbol_table[symtab_index].type, buffer_type);
            symbol_table[symtab_index].addr = start_addr;
            if (strcmp(symbol_table[symtab_index].type, "char") == 0)
                start_addr += strlen(symbol_table[symtab_index].val) - 2;
            else if (strcmp(symbol_table[symtab_index].type, "int") == 0)
                start_addr += 2;
            else
                start_addr += 4;
            symtab_index++;
        }
    }
    else if (check_function(token))
        type = FUNCTION;
    else if (check(token, arith_op))
        type = ARITH_OP;
    else if (check(token, arith_assign_op))
        type = ARTITH_ASSIGN_OP;
    else if (check(token, logic_op))
        type = LOGIC_OP;
    else if (check(token, bit_op))
        type = BIT_OP;
    else if (check(token, rel_op))
        type = REL_OP;
    else if (check(token, unary_op))
        type = UNARY_OP;
    else if (check(token, assign_op))
```

```
        type = ASSIGN_OP;
    else if (check(token, special))
        type = SPECIAL;
    else if ((tmp = check_real_const(token)) != -1)
    {
        if (tmp == 0)
        {
            type = INT_CONST;
            strcpy(buffer_val, token);
        }
        else
        {
            type = FLOAT_CONST;
            strcpy(buffer_val, token);
        }
        if (!check_identifier_table(buffer_id, symbol_table))
        {
            symbol_table[symtab_index].sno = start_sno++;
            strcpy(symbol_table[symtab_index].id, buffer_id);
            strcpy(symbol_table[symtab_index].val, buffer_val);
            strcpy(symbol_table[symtab_index].type, buffer_type);
            symbol_table[symtab_index].addr = start_addr;
            if (strcmp(symbol_table[symtab_index].type, "char") == 0)
                start_addr += strlen(symbol_table[symtab_index].val) - 2;
            else if (strcmp(symbol_table[symtab_index].type, "int") == 0)
                start_addr += 2;
            else
                start_addr += 4;
            symtab_index++;
        }
    }
    else if (check_indentifier(token))
    {
        type = IDENTIFIER;
        strcpy(buffer_id, token);
    }
    else
        type = INVALID;
    return type;
}

void put_token_type(char *const token, token_type type)
{
    char string[26] = {0};

    switch (type)
    {
    case WHITESPACE:
        return;
    case KEYWORD:
        strcpy(string, "Keyword");
        break;
    case FUNCTION:
        strcpy(string, "Function call");
        break;
    case ARITH_OP:
        strcpy(string, "Arithmetic Operator");
        break;
    case LOGIC_OP:
        strcpy(string, "Logical Operator");
        break;
    case BIT_OP:
        strcpy(string, "Bitwise Operator");
        break;
    case REL_OP:
        strcpy(string, "Relational Operator");
        break;
    case ARTITH_ASSIGN_OP:
        strcpy(string, "Arith Assign Operator");
        break;
    case IDENTIFIER:
        strcpy(string, "Identifier");
        break;
    case INT_CONST:
        strcpy(string, "Integer Constant");
        break;
    case FLOAT_CONST:
        strcpy(string, "Float Constant");
        break;
    case CHAR_CONST:
        strcpy(string, "Char Constant");
        break;
    case ASSIGN_OP:
        strcpy(string, "Assignment Operator");
        break;
    case SPECIAL:
        strcpy(string, "Special Character");
        break;
    default:
        strcpy(string, "Invalid Character");
    }
    printf("  | %25s | %-25s |\n", token, string);
}

void process_line(char *const line)
{
    char *start = line;
    char *tmp;
    char token[128];
    int i = 0, idx = 0;
    while (line[idx])
    {
        if (!is_delimiter(line[idx]))
        {
            token[i++] = line[idx++];
            if (line[idx] == '\"')
            {
                while (line[idx] && line[idx] != '\"')
                    token[i++] = line[idx++];
                token[i++] = '\"';
                idx++;
            }
            else if (line[idx] == '(')
            {
                while (line[idx] && line[idx] != ')')
                    token[i++] = line[idx++];
                token[i++] = ')';
                idx++;
            }
        }
        else
        {
            token[i] = 0;
            // printf("token is: %s\n", token);
            if (token[0])
                put_token_type(token, identify_token(token));
            i = 0;
            if (!(line[idx] == ' ' || line[idx] == '\n' || line[idx] == '\t'))
            {
                char temp_str[2] = "";
                temp_str[0] = line[idx];
                temp_str[1] = 0;
                printf("  | %25s | %-25s |\n", temp_str, "Special Character");
            }
            idx++;
        }
    }
}

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        fprintf(stderr, "Invalid Usage! Specify file to parse as argument");
        exit(1);
    }

    char *file_path = argv[1];
    FILE *fp = fopen(file_path, "r");
    if (fp == NULL)
    {
        perror("File not found: ");
        exit(1);
    }

    int count;
    char buffer[1024] = {0};
    bzero(buffer, sizeof(buffer));

    printf("  +---------------------------+---------------------------+\n");
    printf("  |          TOKEN            |          TOKEN TYPE       |\n");
    printf("  +---------------------------+---------------------------+\n");

    while (fgets(buffer, sizeof(buffer), fp))
    {
        remove_newline(buffer);

        if (buffer[0] == '/' && buffer[1] == '/')
        { //Line comment
            remove_newline(buffer);
            printf("  | %25s | %-25s |\n", buffer, "Single Line comment");
        }
        else if (buffer[0] == '/' && buffer[1] == '*') /* handle block comments*/
        {
            remove_newline(buffer);
            printf("  | %25s | %-25s |\n", buffer, " ");
            bzero(buffer, sizeof(buffer));

            int end_block_found = 0;

            while (fgets(buffer, sizeof(buffer), fp) && !end_block_found)
            {
                if (buffer[0] == '*' && buffer[1] == '/')
                    end_block_found = 1;
                remove_newline(buffer);

                printf("  | %25s | %-25s |\n", buffer, ((end_block_found) ? "Multiline
                ↪  Comment" : " "));
                bzero(buffer, sizeof(buffer));
            }
        }
        else if (check_pre_processor(buffer))
        {
            printf("  | %25s | %-25s |\n", buffer, "Preprocessor Directive");
        }
        else
            process_line(buffer);
        bzero(buffer, sizeof(buffer));
    }
    printf("  +---------------------------+---------------------------+\n");

    distab(symbol_table, symtab_index);
}
```

Output

Figure 1: **Sample Program file**

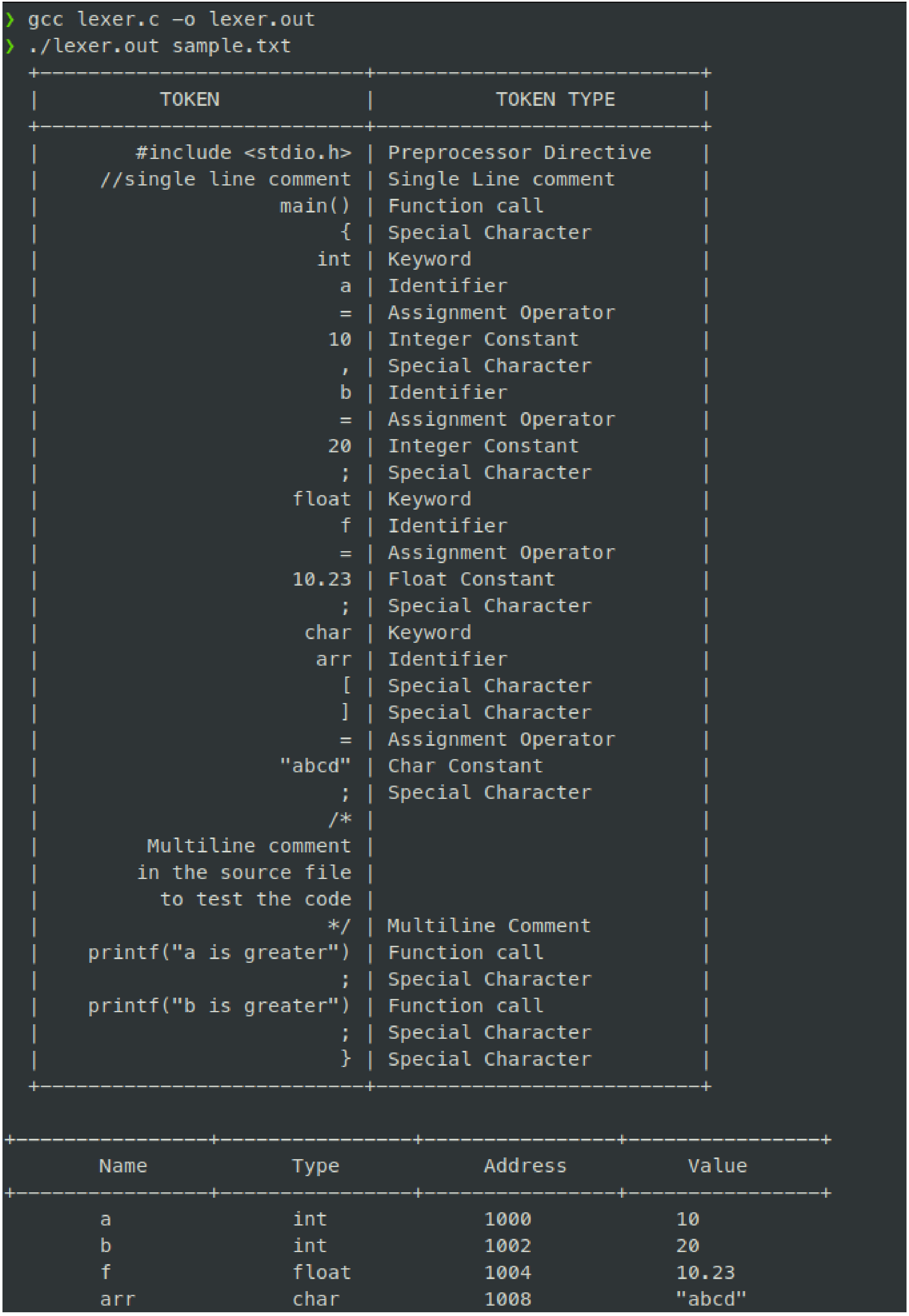


mahesh@mahesh-PC:~/Repositories/Compiler-Design/Assignment-01

```
❯ cat sample.txt
#include <stdio.h>
//single line comment
main() {
        int a = 10,b = 20;
        float f = 10.23;
        char arr[] = "abcd";
/*
Multiline comment
in the source file
to test the code
*/
        if ( a > b )
        printf("a is greater");
        else
        printf("b is greater");

}
```

~/Repositories/Compiler-Design/Assignment-01
on git main !2
miniconda3

Figure 2: **Output of Lexer and symbol table**



```
❯ gcc lexer.c -o lexer.out
❯ ./lexer.out sample.txt
  +---------------------------+---------------------------+
  |           TOKEN           |          TOKEN TYPE       |
  +---------------------------+---------------------------+
  |        #include <stdio.h> | Preprocessor Directive    |
  |     //single line comment | Single Line comment       |
  |                    main() | Function call             |
  |                         { | Special Character         |
  |                       int | Keyword                   |
  |                         a | Identifier                |
  |                         = | Assignment Operator       |
  |                        10 | Integer Constant          |
  |                         , | Special Character         |
  |                         b | Identifier                |
  |                         = | Assignment Operator       |
  |                        20 | Integer Constant          |
  |                         ; | Special Character         |
  |                     float | Keyword                   |
  |                         f | Identifier                |
  |                         = | Assignment Operator       |
  |                     10.23 | Float Constant            |
  |                         ; | Special Character         |
  |                      char | Keyword                   |
  |                       arr | Identifier                |
  |                         [ | Special Character         |
  |                         ] | Special Character         |
  |                         = | Assignment Operator       |
  |                    "abcd" | Char Constant             |
  |                         ; | Special Character         |
  |                        /* |                           |
  |         Multiline comment |                           |
  |        in the source file |                           |
  |          to test the code |                           |
  |                        */ | Multiline Comment         |
  |    printf("a is greater") | Function call             |
  |                         ; | Special Character         |
  |    printf("b is greater") | Function call             |
  |                         ; | Special Character         |
  |                         } | Special Character         |
  +---------------------------+---------------------------+


+----------------+----------------+----------------+----------------+
        Name            Type           Address          Value
+----------------+----------------+----------------+----------------+
        a               int            1000            10
        b               int            1002            20
        f               float          1004            10.23
        arr             char           1008            "abcd"
```