# SSN COLLEGE OF ENGINEERING, KALAVAKKAM

## DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

## UCS1602 - Compiler Design Programming

## Assignment-1

## Implementation of lexical analyzer and symbol table

**Name:** Rahul Ram M

**Class:** CSE - B

**Reg No:** 185001121

**CODE:**

**list.h**

```
struct table
{
  char variable[30];
  char type[10];
  int bytes;
  int address;
  char value[10];
  struct table *next;
};


void printTable(struct table *ptr);
void addRow(struct table** head_ref, char variable[], char type[], int bytes, int address, char value[]);
```

```
bool isEmpty(struct table *head);

bool search(struct table* head, char variable[]);

void addValue(struct table* head, char variable[], char value[]);


void printTable(struct table *ptr)

{

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

printf("Identifier | Type      | Number of  | Address    | Value\n");

printf("Name       |           | Bytes      |            |\n");

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


while(ptr != NULL)

        {

        printf("%-10s | %-10s | %-10d | %-10d | %s\n", ptr->variable, ptr->type,
ptr->bytes, ptr->address, ptr->value);

        ptr = ptr->next;

        }

        printf("\n");

}


void addRow(struct table** head_ref, char variable[], char type[], int bytes, int
address, char value[])

{

   struct table* new_node = (struct table*) malloc(sizeof(struct table));
```

```
    struct table *last = *head_ref;

    new_node->address = address;
    new_node->bytes = bytes;
    strcpy(new_node->variable, variable);
    strcpy(new_node->type, type);
    strcpy(new_node->value, value);

    new_node->next = NULL;

    if (*head_ref == NULL)
    {
       *head_ref = new_node;
       return;
    }

    while (last->next != NULL)
        last = last->next;

    last->next = new_node;
    return;
}

bool isEmpty(struct table *head)
```

```
{
  return head == NULL;
}

bool search(struct table* head, char variable[])
{
   struct table* current = head;

   while (current != NULL)
   {
      if (strcmp(current->variable, variable) == 0)
         return true;
      current = current->next;
   }
   return false;
}

void addValue(struct table* head, char variable[], char value[])
{
   struct table* current = head;

   while (current != NULL)
   {
      if (strcmp(current->variable, variable) == 0)
```

```
      {
        strcpy(current->value, value);
        break;
}

      current = current->next;
   }
}
```

**token_list.h**

```
void addTokens(char token[], char type[]);
void printTokens();
char* getSubString(char line[], int start, int stop);

char tokens[30][50];
char types[30][50];

int tok_size = 0;

void addTokens(char token[], char type[])
{
strcpy(tokens[tok_size], token);
strcpy(types[tok_size], type);
tok_size += 1;
}
```

```
void printTokens()
{
for(int i = 0; i < tok_size; i++)
{
printf("%-50s - %s\n", tokens[i], types[i]);
}
}

char* getSubString(char line[], int start, int stop)
{
int length = stop - start + 1;
char *sub;
int c = 0;
while (c < length) {
     sub[c] = line[start+c];
     c++;
  }
  sub[c] = '\0';
  return sub;
}
```

**data.h**

```
bool checkKeywords(char string[]);
```

```
bool checkDatatypes(char string[]);
bool checkArithmetic_op(char string[]);
bool checkArith_assign_op(char string[]);
bool checkLogical_op(char string[]);
bool checkRelational_op(char string[]);
bool checkBitwise_op(char string[]);
bool checkUnary_op(char string[]);
bool checkSpecial_char(char string[]);
bool checkAssign(char string[]);
bool checkHash(char string[]);
bool checkDelimiters(char character);


char keywords[][10] = {"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"};

char datatypes[][7] = {"int", "char", "float", "double"};
int datatypeVal[] = {2, 1, 4, 8};

char arithmetic_op[][3] = {"+", "-", "*", "/", "%"};

char arith_assign_op[][3] = {"+=", "-=", "*=", "/=", "%="};
```

```
char logical_op[][3] = {"&&", "||", "!"};

char relational_op[][3] = {"<", "<=", ">", ">=", "==", "!="};

char bitwise_op[][3] = {"^", "&", "|", "<<", ">>"};

char unary_op[][3] = {"-", "++", "--"};

char special_char[][3] = {";", ",", ".", "[", "]", "{", "}", "(", ")"};

char delimiters[] = {'+', '-', '*', '%', '&', '|', '!', '<', '>', ' ', ';', ',' ,'{', '}', '[', ']',')', '^', '='};

bool checkKeywords(char string[])
{
for(int i = 0; i < sizeof(keywords)/sizeof(keywords[0]); i++)
{
if(strcmp(keywords[i], string) == 0)
{
return true;
}
}
return false;
}

bool checkDatatypes(char string[])
```

```
{
for(int i = 0; i < 4; i++)
{
if(strcmp(datatypes[i], string) == 0)
{
return true;
}
}
return false;
}

bool checkArithmetic_op(char string[])
{
for(int i = 0; i < 5; i++)
{
if(strcmp(arithmetic_op[i], string) == 0)
{
return true;
}
}
return false;
}

bool checkArith_assign_op(char string[])
{
```

```
for(int i = 0; i < 5; i++)
{
if(strcmp(arith_assign_op[i], string) == 0)
{
return true;
}
}
return false;
}

bool checkLogical_op(char string[])
{
for(int i = 0; i < 3; i++)
{
if(strcmp(logical_op[i], string) == 0)
{
return true;
}
}
return false;
}

bool checkRelational_op(char string[])
{
for(int i = 0; i < 6; i++)
```

```
{
if(strcmp(relational_op[i], string) == 0)
{
return true;
}
}
return false;
}

bool checkBitwise_op(char string[])
{
for(int i = 0; i < 5; i++)
{
if(strcmp(bitwise_op[i], string) == 0)
{
return true;
}
}
return false;
}

bool checkUnary_op(char string[])
{
for(int i = 0; i < 3; i++)
{
```

```
if(strcmp(unary_op[i], string) == 0)
{
return true;
}
}
return false;
}

bool checkSpecial_char(char string[])
{
for(int i = 0; i < 9; i++)
{
if(strcmp(special_char[i], string) == 0)
{
return true;
}
}
return false;
}

bool checkAssign(char string[])
{
return strcmp(string, "=") == 0;
}
```

```
bool checkHash(char string[])
{
return strcmp(string, "#") == 0;
}

bool checkDelimiters(char character)
{
for(int i = 0; i < strlen(delimiters); i++)
{
if(character == delimiters[i])
{
return true;
}
}
return false;
}
```

**main.c**

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

#include "list.h"
#include "data.h"
```

```
#include "token_list.h"

#define MAX_LEN 100

int main()
{
struct table *head = NULL;
char line[100];
char temp_line[100];
char temp[MAX_LEN];

int left, right, v_index, len;
int address = 1000;

   char dt[10];                  // datatype
   char var[5][15];  // variables
   char val[5][10];   // its value

   char mCom[MAX_LEN];          // multi line comment
   strcpy(mCom, "");

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

```
while (fgets(line, MAX_LEN, file))
   {
        left = 0, right = 0, v_index = -1;
        len = strlen(line);

        strcpy(dt, "");

        for(int i = 0; i < 5; i++)
        {
                strcpy(var[i], "");
                strcpy(val[i], "");
}

        strcpy(temp_line, line);

        // multi line comment
        if(strcmp(mCom, "") != 0)
        {
                if(strstr(line, "*/") != NULL)
                {
                        temp[strcspn(temp, "\n")] = 0;
strcpy(temp, getSubString(temp_line, 0, len-4));
strcpy(temp_line, line);
addTokens(temp, "multi line comment");
```

```
strcpy(mCom, "");
}
else
{
line[strcspn(line, "\n")] = 0;
                        addTokens(line, "multi line comment");
}
left = len;
}

        while(left < len)
        {
                right = left;
                while(right < len)
                {
                        // get substring from left to right
                        strcpy(temp, getSubString(temp_line, left, right));
                        strcpy(temp_line, line);

                        // leading spaces
                        // move right to len, to move left towards right
                        if(left == right && line[left] == ' ')
                        {
                                right = len;
```

```
}
// single line comment
else if(strcmp(temp, "//") == 0)
{
strcpy(temp, getSubString(temp_line, right+1, len-1));
strcpy(temp_line, line);
temp[strcspn(temp, "\n")] = 0;
addTokens(temp, "single line comment");
left = len, right = len;
}
// mutli line comment
else if(strcmp(temp, "/*") == 0)
{
strcpy(temp, getSubString(temp_line, right+1, len-1));
strcpy(temp_line, line);
temp[strcspn(temp, "\n")] = 0;
addTokens(temp, "multi line comment");
strcpy(mCom, temp);
left = len, right = len;
}
// for identifiers, keywords and constants
// if right == delimiters and left not equal to right(not a character)
else if(checkDelimiters(line[right]) && left != right)
{
```

```
// get substring before the delimiter
strcpy(temp, getSubString(temp_line, left, right-1));
strcpy(temp_line, line);
// check if it is a keyword
if(checkKeywords(temp))
{
// add token
addTokens(temp, "keyword");
// check if it is a datatype
if(checkDatatypes(temp))
{
// copy the datatype to dt
strcpy(dt, temp);
}
}
// not a keyword - could be identifier or constants
// starting alphabets - identifier (except string or char)
else if(isalpha(temp[0]))
{
// copy the identifer and add token
strcpy(var[++v_index], temp);
addTokens(temp, "identifier");
}
// else it has to be constants(not detailed)
```

```
else
{
// copy the value and add appropriate token
strcpy(val[v_index], temp);
if(strcmp(dt, "int") == 0)
{
addTokens(temp, "integer constant");
}
else if(strcmp(dt, "float") == 0)
{
addTokens(temp, "float constant");
}
else
{
addTokens(temp, "double constant");
}


}
// moving left, next to the current word using below 2 assignments
left = right-1;
right = len;
}
// for single characters
else if(checkDelimiters(line[right]) && right == left)
```

```
{
// if it is not the last character of the line (2 character ops)
if(right != len-1)
{
// get the next character along with the current one in the substring
strcpy(temp, getSubString(temp_line, left, right+1));
strcpy(temp_line, line);
// check if it matches any ops
// if yes - move left next to the op
if(checkArith_assign_op(temp))
{
addTokens(temp, "arithmetic assignment operator");
left = right+1;
right = len;
}
else if(checkLogical_op(temp))
{
addTokens(temp, "logical operator");
left = right+1;
right = len;
}
else if(checkRelational_op(temp))
{
addTokens(temp, "relational operator");
```

```
left = right+1;
right = len;
}
else if(checkBitwise_op(temp))
{
addTokens(temp, "bitwise operator");
left = right+1;
right = len;
}
else if(checkUnary_op(temp))
{
addTokens(temp, "unary operator");
left = right+1;
right = len;
}
// one character ops
else
{
// get only the current character
strcpy(temp, getSubString(temp_line, left, right));
strcpy(temp_line, line);
// check for the ops and add token
// if yes - move left next to the op
if(checkArithmetic_op(temp))
```

```
{
addTokens(temp, "arithmetic operator");
left = right;
right = len;
}
else if(strcmp(temp, "=") == 0)
{
addTokens(temp, "assignment operator");
left = right;
right = len;
}
else if(checkRelational_op(temp))
{
addTokens(temp, "relational operator");
left = right;
right = len;
}
else if(checkBitwise_op(temp))
{
addTokens(temp, "bitwise operator");
left = right;
right = len;
}
else if(checkUnary_op(temp))
```

```
{
addTokens(temp, "unary operator");
left = right;
right = len;
}
else if(checkSpecial_char(temp))
{
addTokens(temp, "special character");
left = right;
right = len;
}
// if none matches -  error
else
{
strcpy(temp, getSubString(temp_line, left, len-1));
strcpy(temp_line, line);
temp[strcspn(temp, "\n")] = 0;
addTokens(temp, "error");
// error skip the current line
left = len;
right = len;
}
}
}
```

```
// single character op
else
{
// same as else part
strcpy(temp, getSubString(temp_line, left, right));
strcpy(temp_line, line);
if(checkArithmetic_op(temp))
{
addTokens(temp, "arithmetic operator");
left = right;
right = len;
}
else if(strcmp(temp, "=") == 0)
{
addTokens(temp, "assignment operator");
left = right;
right = len;
}
else if(checkRelational_op(temp))
{
addTokens(temp, "relational operator");
left = right;
right = len;
}
```

```
else if(checkBitwise_op(temp))
{
addTokens(temp, "bitwise operator");
left = right;
right = len;
}
else if(checkUnary_op(temp))
{
addTokens(temp, "unary operator");
left = right;
right = len;
}
else if(checkSpecial_char(temp))
{
addTokens(temp, "special character");
left = right;
right = len;
}
else
{
strcpy(temp, getSubString(temp_line, left, len-1));
strcpy(temp_line, line);
temp[strcspn(temp, "\n")] = 0;
addTokens(temp, "error");
```

```
left = len;
right = len;
}
}
}
// char constant
else if(line[right] == '\'' && left == right)
{
strcpy(temp, getSubString(temp_line, right+1, right+1));
strcpy(temp_line, line);
strcpy(val[v_index], temp);
addTokens(temp, "char constant");
left = right+2;
right = len;
}
// string constant
else if(line[right] == '"' && left == right)
{
int lst = -1;
for(int i = right+1; i < len; i++)
{
if(line[i] == '"')
{
lst = i;
```

```
break;
}
}
if(lst != -1)
{
strcpy(temp, getSubString(temp_line, left+1, lst-1));
strcpy(temp_line, line);
addTokens(temp, "string constant");
left = lst;
right = len;
}
else
{
strcpy(temp, getSubString(temp_line, left, len-1));
strcpy(temp_line, line);
temp[strcspn(temp, "\n")] = 0;
addTokens(temp, "error");
// error skip the current line
left = len;
right = len;
}
}
// preprocessor directive
                    else if(checkHash(temp) && left == right)
```

```
                        {
                                line[strcspn(line, "\n")] = 0;
                                addTokens(line, "preprocessor directive");
                                left = len, right = len;
}

// function call and conditions
else if(line[right] == '(')
{
strcpy(temp, getSubString(temp_line, left, right-1));
strcpy(temp_line, line);

bool isfun = true;
int last = len;

// inbuilt
if(checkKeywords(temp))
{
addTokens(temp, "keyword");
addTokens("(", "special character");
left = right;
right = len;
break;
}
```

```
if(isalpha(temp[0]))
{
for(int i = 1; i < strlen(temp); i++)
{
if(!isalpha(temp[i]) && !isdigit(temp[i]) && temp[i] != '_')
{
isfun = false;
break;
}
}
}
else
{
isfun = false;
}

if(isfun)
{
for(int i = right; i < len; i++)
{
if(line[i] == ')')
{
last = i+1;
```

```
break;
}
}
strcpy(temp, getSubString(temp_line, left, last-1));
strcpy(temp_line, line);
temp[strcspn(temp, "\n")] = 0;
addTokens(temp, "function call");

left = last-1;
right = len;
}
else
{
strcpy(temp, getSubString(temp_line, left, len-1));
strcpy(temp_line, line);
temp[strcspn(temp, "\n")] = 0;
addTokens(temp, "error in function call");
left = len, right = len;
}

}

                        right++;
}
```

```
left++;
}
int bytes = 0;
for (int j = 0; j < 4; j++)
{
if(strcmp(dt, datatypes[j]) == 0)
{
bytes = datatypeVal[j];
break;
}
}
for(int i = 0; i <= v_index; i++)
{
if(strcmp(dt, "") != 0)
{
addRow(&head, var[i], dt, bytes, address, val[i]);
                address += bytes;
}
   }
}
printTokens();
printTable(head);
return 0;
}
```

**input.txt**

```
#include<stdio.h>
main()
{
float a=10,b=20;
if(a>b)
printf("a is greater");
else
printf("b is greater");
// calling hello()
hello();
}
```

**Sample Output:**

```
#include<stdio.h>                    - preprocessor directive
main()                               - function call
{                                    - special character
float                                - keyword
a                                    - identifier
=                                    - assignment operator
10                                   - float constant
,                                    - special character
b                                    - identifier
=                                    - assignment operator
20                                   - float constant
```

; - special character

if - keyword

( - special character

a - identifier

> - relational operator

b - identifier

) - special character

printf("a is greater") - function call

; - special character

else - keyword

printf("b is greater") - function call

; - special character

calling hello() - single line comment

hello() - function call

; - special character

} - special character

Content of Symbol Table:

| Identifier Name | Type | Number of Bytes | Address | Value |
|---|---|---|---|---|
| a | float | 4 | 1000 | 10 |
| b | float | 4 | 1004 | 20 |