SSN COLLEGE OF ENGINEERING, KALAVAKKAM

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UCS1602 - Compiler Design

EX - 1 : Implementation of lexical analyser and symbol table

NAME : Gayathri M
REG NO: 185001050
DATE : 02/02/2021

Program Code:

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

//Symbol table
struct symtab
{
    char id[20];
    char type[20];
    int numb;
    int add;
    char val[20];

};

//check if a word is a keyword
int isKeyword(char word[])
{
    char keywords[32][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"};
```

```
    for(int i = 0; i < 32; ++i)
    {
        if(strcmp(keywords[i],word) == 0)
        {
            return 1;
        }
    }
    return 0;
}
//check if a word is a decimal integer constant
int isanumber(char word[])
{
    int i =0;
    for (i =0; i<strlen(word); i++)
    {
        if(isdigit(word[i])==0)
            return 0;
    }
    return 1;
}
//check if a word is a hexadecimal  constant
int ishex(char word[]){
    int i=0;
    for (i =0; i<strlen(word); i++)
    {
        if(isdigit(word[i])==0 && !(word[i]>='A' && word[i]<='F')){
            return 0;
        }
    }
    return 1;
}
//identify the type of the identifier
int istype(char buff[])
{
    char type[][10]={"int","double","char"};
    int x=0;
    for(x=0; x<4; x++)
    {
        if(strcmp(buff,type[x])==0)
        {
            return x+1;
        }
    }
    return 0;
}
```

```c
//Displaying the symbol table
void distab(struct symtab s[],int len)
{

printf("\n+----------------+----------------+----------------+------------
---+---------------+");
    printf("\n\tName\t\tType\t\tBytes\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%d\t\t%s\n",s[i].id,s[i].type,s[i].numb,s[i].ad
d,s[i].val);
    }
}
//check if the identifier is already present in the symbol table
int chk_id(char id[],struct symtab s[],int len)
{
    for(int i=0; i<len; i++)
    {
        if(strcmp(s[i].id,id)==0)
        {
            return 1;
        }
    }
    return 0;
}

int main()
{
    char statement[200];
    char word[200];
    char ch;
    int len=0,i,j,k=0,l,f=0,p=0,z=0,t=0,c=0,e=0,n=0;
    char op[]="+-*/%";
    char relop[]="<>=!";
    char logop[]="&|";
    char spch[]=";,.[](){}[]";
    char numc[20];
    int nb=0,temp=0,ct=0;
    FILE *fp;
    struct symtab s[50];
    int typect = 0;
    int ltyp = 0;
```

```
int add=1000;

//open file that has source code
fp = fopen("source.txt","r");
if(fp == NULL)
{
    printf("ERROR: Unable to open file\n");
    exit(0);
}
while((ch = fgetc(fp)) != EOF)
{
    statement[z++] = ch;
}
statement[z] = '\0';
fclose(fp);
len = strlen(statement);
printf("\nThe tokens and their types: \n");

for(i=0; i<len; i++)
{
    //preprocessor directive
    if(statement[i]=='#')
    {
        word[k++] = statement[i];
        l=i+1;
        while(statement[l]!='\n')
        {
            word[k++] = statement[l];
            l++;
        }
        i = l;
        word[k] = '\0';
        k = 0;
        printf("\n%s \t\t-\tpreprocessor directive",word);
        p = 1;
        continue;

    }
    //check for single Comment statement
    if(statement[i]=='/' && statement[i+1]=='/')
    {
        word[k++] = statement[i];
        l=i+1;
        while(statement[l]!='\n')
        {
            word[k++] = statement[l];
```

```
            l++;
        }
        i = l;
        word[k] = '\0';
        k = 0;
        printf("\n%s \t\t-\t\t single comment line",word);
        ct = 1;
        continue;
    }
    //check for multiline comment statement
    else if(statement[i]=='/' && statement[i+1]=='*')
    {
        word[k++] = statement[i];
        l=i+1;
        while(statement[l]!='/')
        {
            if(statement[l]!='\n')
            {
                word[k++] = statement[l];
            }
            l++;
            if(l == len)
            {
                printf("\nERROR : Multiline comment unterminated \n");
                return 0;
            }
        }
        word[k++] = statement[l];
        i = l;
        word[k] = '\0';
        k = 0;
        printf("\n%s - multiple comment line",word);
        ct = 1;
        continue;

    }
    // identifying identifier , function calls , keywords , constants
    if(statement[i]=='\'' || statement[i]=='\"')
    {
        l = i+1;
        while(statement[l]!='\'' && statement[l]!='\"')
        {
            word[k++] = statement[l];
            l++;
        }
        i = l+1;
```

```
            word[k] = '\0';
            printf("\n%s \t\t-\t\t string constant",word);
            strcpy(s[t].val,word);
            t++;
            c = 1;
        }
        if(isalnum(statement[i]))
        {
            word[k++] = statement[i];
            //check for value of int/double
            if(isdigit(statement[i]) || (statement[i]>='A' &&
statement[i]<='F'))
            {
                l = i+1;
                while(statement[l]!=' ')
                {

                    word[k++] = statement[l];
                    l++;
                }
                word[k] = '\0';
                i = l+1;
                if(strchr(word,'.'))
                {
                    printf("\n%s \t\t-\t\t Double constant",word);
                }
                else{
                    if(isanumber(word))
                        printf("\n%s \t\t-\t\t Decimal Integer
constant",word);
                    else if(ishex(word))
                        printf("\n%s \t\t-\t\t Hexadecimal Integer
constant", word);
                }
                strcpy(s[t].val,word);
                t++;
                n=1;
            }
        }
        else if((statement[i] == '(') && (k != 0))
        {
            word[k++] = statement[i];
            l=i+1;
            while(statement[l]!=')')
            {
                word[k++] = statement[l];
```

```
            l++;
        }
        word[k++] = statement[l];
        i = l++;
        word[k] = '\0';
        printf("\n%s \t\t-\t\t function call",word);
        f = 1;
        continue;

    }
    else if((statement[i] == ' ' || statement[i] == '\n') && (k != 0))
    {
        word[k] = '\0';
        k = 0;
        if(isKeyword(word) == 1)
        {
            printf("\n%s \t\t-\t\t keyword", word);
            if(istype(word)>0)
            {
                if(istype(word)==1)
                {
                    typect+=1;
                    s[t].numb=2;
                }
                else if(istype(word)==2)
                {
                    typect+=1;
                    s[t].numb=8;
                }
                else
                {
                    typect+=1;
                    s[t].numb=1;
                }
                strcpy(s[t].type,word);
                if(t==0)
                {
                    s[t].add=add;
                }
                else
                    s[t].add = add + s[t].numb;
                add = s[t].add;
            }
        }
        else if(f == 0 && p==0 && c==0 && n==0 && ct==0)
        {
```

```
            printf("\n%s \t\t-\t\t identifier", word);
            if(chk_id(word,s,t)==0)
            {
                strcpy(s[t].id,word);
            }
            else
            {
                continue;
            }
            if(ltyp-typect==0 && t>0)
            {
                strcpy(s[t].type,s[t-1].type);
                s[t].numb=s[t-1].numb;
                s[t].add = s[t-1].add + s[t].numb;
                add = s[t].add;
            }
            if(statement[i+1]!='=')
            {
                strcpy(s[t].val,"-");
                t++;
            }
            ltyp=typect;
        }
        f=0;
        p=0;
        c=0;
        n=0;
        ct=0;

    }
    else if((statement[i] == '[') && (k != 0))
    {
        l=i+1;
        while(statement[l]!=']')
        {
            numc[e++] = statement[l];
            l++;
        }
        numc[e] = '\0';
        nb = atoi(numc);
        e = 0;
        i = l++;
        word[k] = '\0';
        printf("\n%s \t\t-\t\t identifier of size %s",word,numc);
        if(chk_id(word,s,t)==0)
        {
```

```
                strcpy(s[t].id,word);
            }
            else
            {
                continue;
            }
            if(ltyp-typect==0 && t>0)
            {
                strcpy(s[t].type,s[t-1].type);
                s[t].numb=s[t-1].numb;
                s[t].add = s[t-1].add + s[t].numb;
                add = s[t].add;
            }
            if(nb>0)
            {
                temp = (nb-1)*s[t].numb;
                s[t].numb = nb*s[t].numb;
                s[t].add+=temp;
                add = s[t].add;
                nb = 0;
            }
            if(statement[i+2]!='=')
            {
                strcpy(s[t].val,"-");
                t++;
            }
            ltyp=typect;
            f = 1;
            continue;
        }
        // arithmetic operators and assignment op , unary op
        for(j=0; j<5; j++)
        {
            if(statement[i]==op[j])
            {
                if(statement[i+1]=='=')
                {
                    printf("\n%c%c \t\t-\t\t assignment
operator",statement[i],statement[i+1]);
                    i+=1;
                }
                else if((statement[i]=='+' &&
statement[i+1]=='+')||(statement[i]=='-' && statement[i+1]=='-'))
                {
                    printf("\n%c%c \t\t-\t\t unary
operator",statement[i],statement[i+1]);
```

```
                    i+=1;
                }
                else
                    printf("\n%c \t\t-\t\t operator",statement[i]);
            }
        }
        // Relational operators , logical , bit,assignment
        for(j=0; j<4; j++)
        {
            if(statement[i]==relop[j])
            {
                if((statement[i]=='<' &&
statement[i+1]=='<')||(statement[i]=='>' && statement[i+1]=='>'))
                {
                    printf("\n%c%c \t\t-\t\t bit
operator",statement[i],statement[i+1]);
                    i++;
                }
                else if(statement[i+1]=='=')
                {
                    printf("\n%c%c \t\t-\t\t relational
operator",statement[i],statement[i+1]);
                    i++;
                }
                else if(statement[i]=='!' && statement[i+1]==' ')
                {
                    printf("\n%c \t\t-\t\t logical operator",statement[i]);
                }
                else if(statement[i]=='=' && statement[i+1]==' ')
                {
                    printf("\n%c \t\t-\t\t assignment
operator",statement[i]);
                }
                else
                    printf("\n%c \t\t-\t\t relational
operator",statement[i]);
            }
        }
        // logical
        for(j=0; j<2; j++)
        {
            if(statement[i]==logop[j])
            {
                if(statement[i+1]=='&'||statement[i+1]=='|')
                {
                    printf("\n%c%c \t\t-\t\t logical
```

```
operator",statement[i],statement[i+1]);
                    i++;
                }
                else
                    printf("\n%c \t\t-\t\t bit operator",statement[i]);
            }
        }
        //special characters
        for(j=0; j<11; j++)
        {
            if(statement[i]==spch[j])
            {
                printf("\n%c \t\t-\t\t special character",statement[i]);
            }
        }
        if(statement[i]=='^')
        {
            printf("\n%c \t\t-\t\t bit operator",statement[i]);
        }
    }
    printf("\n\nContents of the symbol table: \n");
    distab(s,t);
    return 0;
}
```

SOURCE CODE :

```
#include <stdio.h>
// Main code
/* this is a multiline comment
it spans 2 lines of code */
int add(int a , int b) ;
main()
{
int a = 10AE , b = 20 ;
char c = 'h' , str[6] = "hello" ;
double d = 88.888 ;
if ( ! ( a > b ) )
printf("a is greater");
else
printf("b is greater");
}
```

SAMPLE OUTPUT :

msml@MSMLs-MacBook-Pro ex1 % gcc scanner.c -o s
msml@MSMLs-MacBook-Pro ex1 % ./s

The tokens and their types:

#include <stdio.h> - preprocessor directive
// Main code - single comment line
/* this is a multiline commentit spans 2 lines of code */ - multiple comment line
int - keyword
add(int a , int b) - function call
; - special character
main() - function call
{ - special character
int - keyword
a - identifier
= - assignment operator
10AE - Hexadecimal Integer constant
, - special character
b - identifier
= - assignment operator
20 - Decimal Integer constant
; - special character
char - keyword
c - identifier
= - assignment operator
h - string constant
, - special character
str - identifier of size 6
= - assignment operator
hello - string constant
; - special character
double - keyword
d - identifier
= - assignment operator
88.888 - Double constant
; - special character
if - keyword
(- special character

```
!                       -       logical operator
(                       -       special character
a                       -       identifier
>                       -       relational operator
b                       -       identifier
)                       -       special character
)                       -       special character
printf("a is greater")          -       function call
;                       -       special character
else                    -       keyword
printf("b is greater")                  -       function call
;                       -       special character
}                       -       special character
```

Contents of the symbol table:

Name	Type	Bytes	Address	Value
a	int	2	1000	10AE
b	int	2	1002	20
c	char	1	1003	h
str	char	6	1009	hello
d	double	8	1017	88.888

Learning Outcomes :

- I understood the use and necessity of lexical analyser in a compiler.
- I learnt to design a basic lexical analyser.
- I learnt to separate tokens in C given its source code.
- I learnt to maintain the symbol table and update its contents.