Name –Bhargav Shamuvel Gurav **PRN** – 2041009

Class – L.Y. B-Tech (Computer) **Batch** – B1

Course Code – CO406U **Course Name** - CDL

Practical no. 1

Aim: Design a lexical analyzer for a given language and the lexical analyzer should ignore redundant spaces, tabs and newlines.

Theory :

A lexical analyzer, also known as a lexer or scanner, is a fundamental component of a compiler or interpreter that breaks down the input source code into a sequence of tokens. The primary purpose of the lexer is to remove whitespace (spaces, newlines, tabs) and produce a stream of tokens that can be processed by the parser or other parts of the compiler.

Here's a simplified example of how a lexical analyzer might work to remove spaces, newlines, and tabs from the input source code in a high-level language like Python. We'll use Python-like pseudocode for illustration:

```
def lexer(source_code):
    tokens = []
    current_token = ""

    for char in source_code:
        if char in [' ', '\n', '\t']:
            # Ignore whitespace characters
            continue
        else:
            current_token += char

    return tokens

# Example usage:
source_code = """
for i in range(10):
    if i % 2 == 0:
        print(i)
"""

tokens = lexer(source_code)
```

print(tokens)

In this pseudocode:

1. We define a lexer function that takes the source code as input.
2. We initialize an empty list called tokens to store the resulting tokens and an empty string current_token to build the current token.
3. We iterate over each character in the source code.
4. If the character is a space, newline, or tab, we simply continue to the next character, effectively ignoring it.
5. If the character is not whitespace, we append it to the current_token.
6. Finally, we return the list of tokens.

The example usage demonstrates how the lexer removes spaces, newlines, and tabs from the source code. The tokens list will contain the source code without these whitespace characters. Note that this is a simplified example, and real-world lexers are more complex, as they need to handle a variety of programming language constructs, keywords, and symbols. Additionally, they often generate tokens with associated types, such as identifiers, keywords, literals, and operators.

Program Code:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
void keyw(char *p);
int i=0,id=0,kw=0,num=0,op=0;
char	keys[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","un
signed","void","volatile","while"};
void main()
{
    char ch,str[25],seps[15]=" \t\n,;(){}[]#\"<>",oper[]="!%^&*-+=~|.<>/?"; int j;
    FILE *f1;
    f1 = fopen("input.txt","r");
    while((ch=fgetc(f1))!=EOF)
    {
            for(j=0;j<=14;j++)
            {
```

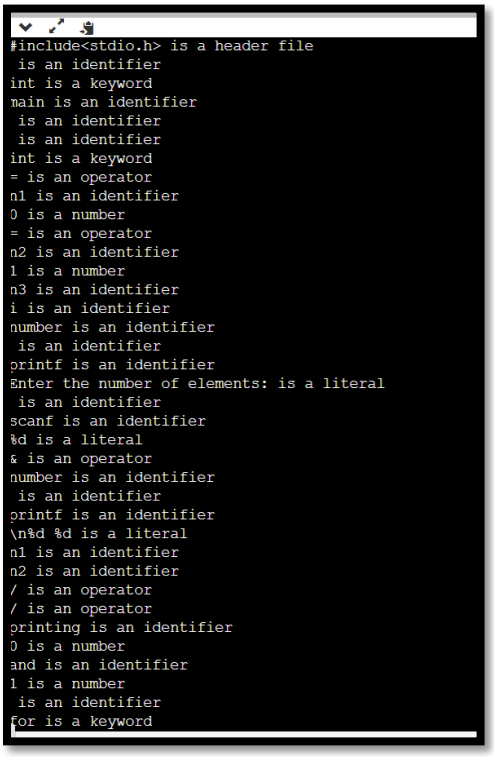
```
		if(ch==oper[j])
		{
			printf("%c is an operator\n",ch); op++;
			str[i]='\0';
			keyw(str);
		}
	}
	for(j=0;j<=14;j++)
	{
		if(i==-1) break;
		if(ch==seps[j])
		{
			if(ch=='#')
			{
				while(ch!='>')
				{
					printf("%c",ch);
					ch=fgetc(f1);
				}
				printf("%c is a header file\n",ch); i=-1;
				break;
			}
			if(ch=='")
			{
				do
				{
					ch=fgetc(f1);
					printf("%c",ch);
				}
				while(ch!='");
				printf("\b is a literal\n"); i=-1;
				break;
			}
			str[i]='\0';
			keyw(str);
		}

	}
	if(i!=-1)
	{
```

```
                str[i]=ch;
                i++;
        }
        else i=0;
    }
    printf("Keywords:    %d\nIdentifiers:    %d\nOperators:    %d\nNumbers:
%d\n",kw,id,op,num);
}

void keyw(char *p)
{
    int k,flag=0; for(k=0;k<=31;k++)
    {
        if(strcmp(keys[k],p)==0)
        {
            printf("%s is a keyword\n",p); kw++;
            flag=1;
            break;
        }

    }
    if(flag==0)
    {
        if(isdigit(p[0]))
        {
            printf("%s is a number\n",p); num++;
        }
        else
        {
            if(p[0]!='\0')
            {
                printf("%s is an identifier\n",p); id++;
            }

        }

    }
    i=-1;
}
```

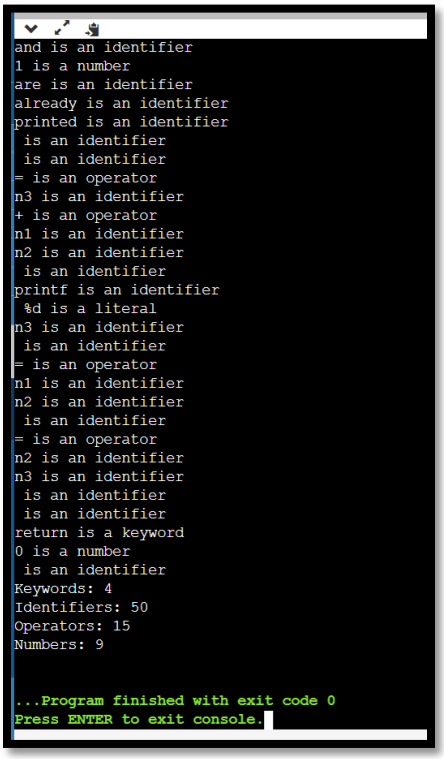
Input Code (input.txt):

```
#include<stdio.h>
int main()
{
 int n1=0,n2=1,n3,i,number;
 printf("Enter the number of elements:");
 scanf("%d",&number);
 printf("\n%d %d",n1,n2);//printing 0 and 1
 for(i=2;i<number;++i)//loop starts from 2 because 0 and 1 are already printed
 {
  n3=n1+n2;
  printf(" %d",n3);
  n1=n2;
  n2=n3;
 }
  return 0;
 }
```

Output:



```
#include<stdio.h> is a header file
 is an identifier
int is a keyword
main is an identifier
 is an identifier
 is an identifier
int is a keyword
= is an operator
n1 is an identifier
0 is a number
= is an operator
n2 is an identifier
1 is a number
n3 is an identifier
i is an identifier
number is an identifier
 is an identifier
printf is an identifier
Enter the number of elements: is a literal
 is an identifier
scanf is an identifier
%d is a literal
& is an operator
number is an identifier
 is an identifier
printf is an identifier
\n%d %d is a literal
n1 is an identifier
n2 is an identifier
/ is an operator
/ is an operator
printing is an identifier
0 is a number
and is an identifier
1 is a number
 is an identifier
for is a keyword
```



```
and is an identifier
1 is a number
are is an identifier
already is an identifier
printed is an identifier
 is an identifier
 is an identifier
= is an operator
n3 is an identifier
+ is an operator
n1 is an identifier
n2 is an identifier
 is an identifier
printf is an identifier
 %d is a literal
n3 is an identifier
 is an identifier
= is an operator
n1 is an identifier
n2 is an identifier
 is an identifier
= is an operator
n2 is an identifier
n3 is an identifier
 is an identifier
 is an identifier
return is a keyword
0 is a number
 is an identifier
Keywords: 4
Identifiers: 50
Operators: 15
Numbers: 9


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion : In this practical we learnt how the lexical analyzer identifies tokens.