**Name** –Bhargav Shamuvel Gurav

**Class** – L.Y. B-Tech (Computer)

**Course Code** – CO406U

**PRN** – 2041009

**Batch** – B1

**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;

}

}

}

if(i!=-1)

{

str[i]='\0'; keyw(str);

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)

{

}

}

if(flag==0)

{

printf("%s is a keyword\n",p); kw++; flag=1;

break;

if(isdigit(p[0]))

{

}

else

{

printf("%s is a number\n",p); num++;

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;

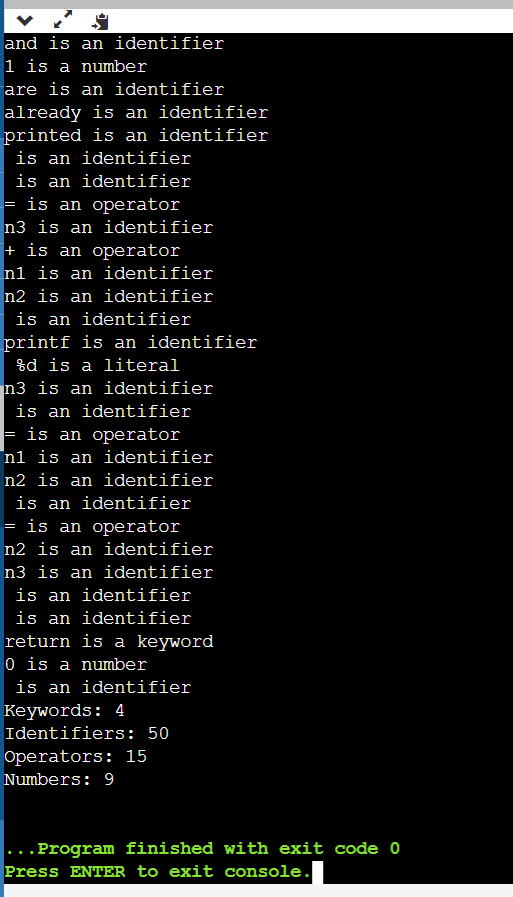
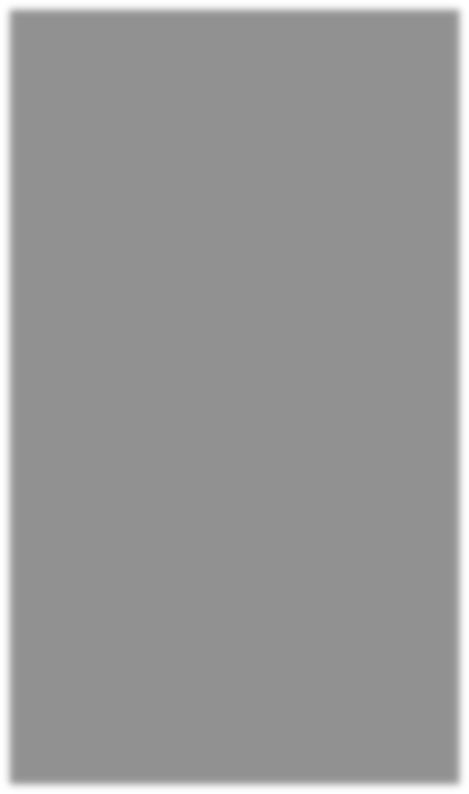
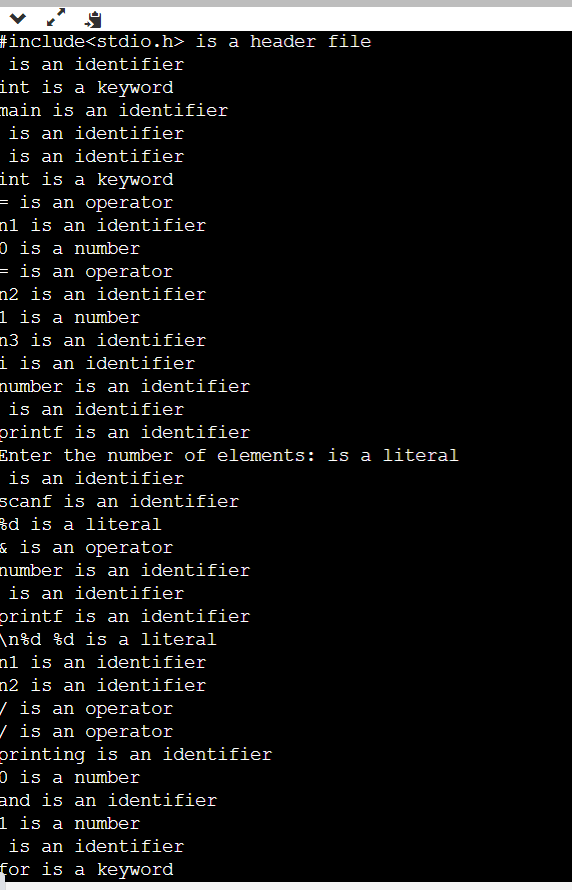
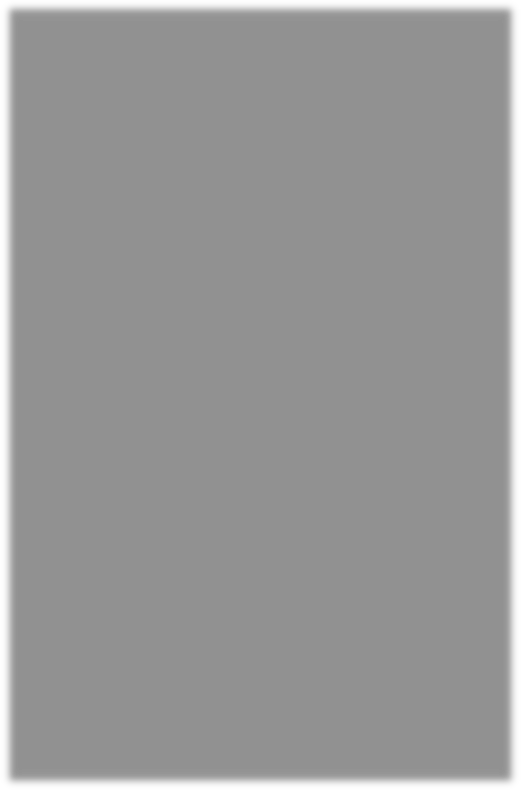
n2=n3;

}

return 0;

}

# Output:



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