**Compiler Design Project Report**

Submitted by,

Honrao Akshay Anandrao (15CO218)

Tarun Karthik (15CO155)

**Introduction to Compiler :**

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

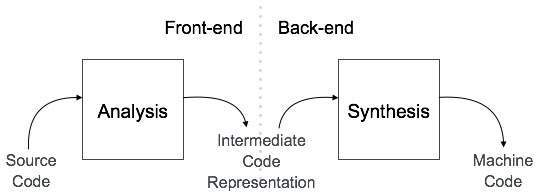
A compiler can broadly be divided into two phases based on the way they compile -

### **1] Analysis Phase :**

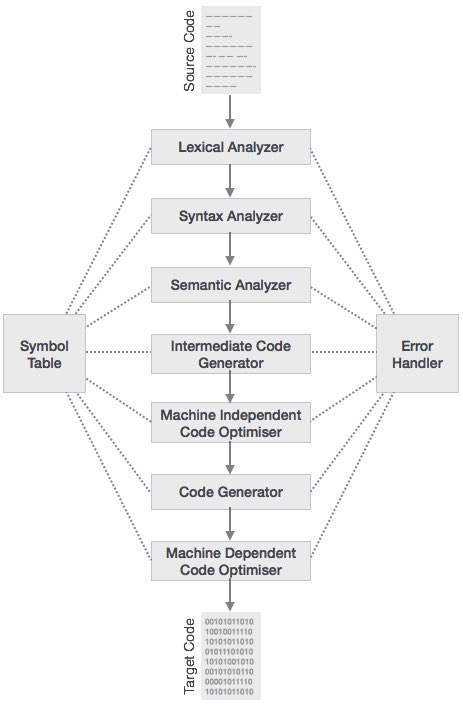
Known as the front-end of the compiler, the **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

### **2] Synthesis Phase :**

Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.



The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



### 

### **Syntax Analysis**

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

### **Semantic Analysis**

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

### **Intermediate Code Generation**

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

### **Code Optimization**

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

### **Code Generation**

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

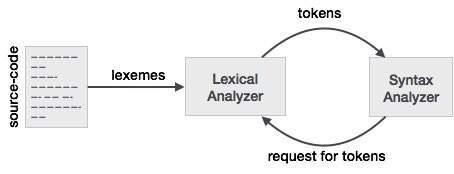
**Symbol Table :** It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

### 

### **Lexical Analysis Phase :**

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



## **Tokens :**

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

**Lexical Analyzer Code :**

=====================================================================

**CODE**

=====================================================================

%{

#include<stdio.h>

#include <math.h>

#include <ctype.h>

#include <stdio.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

int keywordcount = 0;

int linecount = 1;

int mcomment = 0, max=0;

#define SIZE 100

%}

preprocessor #include

header <[a-zA-Z]+\.h>

keywords while|for|int|float|char|auto|break|case|const|continue|default|do|double|else|enum|extern|goto|if|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|printf|scanf|main

line [\n]

space [ ]

startc "/\*"

endc "\*/"

varchar [a-zA-Z\_][a-zA-Z0-9\_]\*

number [0-9]\*

string \".+\"

openbc [(]

closebc [)]

digit [0-9]

pointer "\*"{varchar}

array {varchar}"["{number}+"]"

invalidid {digit}+[\_a-zA-Z]+[\_a-zA-Z0-9]\*

all ({varchar}|{digit}|{operator}|{array}|{keywords})\*

%s comment

%s function

%%

{preprocessor} {printf("\nline number : %d : %s is a preprocessor\n",linecount,yytext);if(mcomment==0)

Insert("preprocessor",yytext,Search(yytext));}

{header} {printf("\nline number : %d : %s is a header\n",linecount,yytext);if(mcomment==0)Insert("header",yytext,Search(yytext));}

{invalidid} {printf("\nERROR 1 : Line number %d : Invalid Identifier :%s",linecount,yytext);exit(1);}

[/]{1}[/]{1}.\* { printf("\nline number : %d : %s : is a comment\n",linecount,yytext); }

{keywords} { keywordcount++; printf("\nline number : %d : C Keyword(%d) : %s\n",linecount,keywordcount,yytext); if(mcomment==0)Insert("keywords",yytext,Search(yytext)); }

{startc} {BEGIN comment; if(mcomment==0){printf("\nmultiline comment starts at line %d\n",linecount); mcomment = 1;}else{printf("\nERROR 2 : Line number %d : Comment Error ",linecount);exit(1);}}

<comment>\n {BEGIN comment; linecount++;}

<comment>[a-zA-Z0-9 ]\* ;

<comment>{endc} {BEGIN 0; if(mcomment == 1)printf("\nmultiline comment end at line %d\n",linecount); mcomment = 0;}

{line} { linecount++; }

{pointer} {printf("\nline number : %d : %s is a (pointer) variable name\n",linecount,yytext);if(mcomment==0)Insert("identifier",yytext,Search(yytext));}

{array} {printf("\nline number : %d : %s is a (array) variable name\n",linecount,yytext);if(mcomment==0)Insert("identifier",yytext,Search(yytext));}

{varchar} {printf("\nline number : %d : %s is a variable name\n",linecount,yytext);if(mcomment==0)Insert("identifier",yytext,Search(yytext));}

{number} {printf("\nline number : %d : %s is a number\n",linecount,yytext);if(mcomment==0)Insert("Number",yytext,Search(yytext));}

{string} {printf("\nline number : %d : %s , is a string\n",linecount,yytext);if(mcomment==0)Insert("String",yytext,Search(yytext));}

"=" {printf("\nline number : %d : equal to operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"+" {printf("\nline number : %d : addition operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"-" {printf("\nline number : %d : subtraction operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"\*" {printf("\nline number : %d : multiplication operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"/" {printf("\nline number : %d : division operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"%" {printf("\nline number : %d : modulo operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"++" {printf("\nline number : %d : increment operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"--" {printf("\nline number : %d : decrement operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"+=" {printf("\nline number : %d : add & assign operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"-=" {printf("\nline number : %d : subtract & assign operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"\*=" {printf("\nline number : %d : multiply & assign operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"/=" {printf("\nline number : %d : divide & assign operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"==" {printf("\nline number : %d : equal to operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"%=" {printf("\nline number : %d : modulo & assign to operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

">=" {printf("\nline number : %d : greater than or equal to operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"<=" {printf("\nline number : %d : less than or equal to operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"!=" {printf("\nline number : %d : not equal to operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

">" {printf("\nline number : %d : greater than operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"<" {printf("\nline number : %d : less than operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"?" {printf("\nline number : %d : conditional operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"&" {printf("\nline number : %d : bit wise AND operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"|" {printf("\nline number : %d : bit wise OR operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"&&" {printf("\nline number : %d : AND operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"||" {printf("\nline number : %d : OR operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"!" {printf("\nline number : %d : NOT operator %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

":" {printf("\nline number : %d : Colon %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

";" {printf("\nline number : %d : semi colon %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"," {printf("\nline number : %d : comma %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"{" {printf("\nline number : %d : open curly brace %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

"}" {printf("\nline number : %d : closing curly brace %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

{openbc} {printf("\nline number : %d : open round brace %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

{closebc} {printf("\nline number : %d : close round brace %s\n",linecount,yytext);if(mcomment==0)Insert("Operator",yytext,Search(yytext));}

%%

int size=0;

void Insert(char[],char[],int);

void Display();

int Search(char[]);

struct SymbTab

{

char label[100],symbol[100];

int addr;

struct SymbTab \*next;};

struct SymbTab \*first,\*last;

int main()

{

yyin = fopen("Program.txt","r");

FILE \*file= fopen("Symbol\_Table.txt", "w");

yyout= file;

yylex();

fclose(yyin);

Display();

fclose(yyout);

return(0);

}

void Insert(char l[100],char a[100],int op)

{

int n;

struct SymbTab \*p;

p=malloc(sizeof(struct SymbTab));

strcpy(p->label,l);

strcpy(p->symbol,a);

p->addr=op;

p->next=NULL;

if(size==0){

first=p;

last=p;

}

else{

last->next=p;

last=p;

}

size++;

}

int Search(char l[100])

{

int abc=0;

struct SymbTab \*qw;

if(size==0)

{

return 0;

}

qw=first;

int i;

for(i=0;i<size;i++)

{

if(strcmp(qw->symbol,l)==0)

{

abc=qw->addr;

return abc;

}

else

{

qw=qw->next;

}

}

max++;

return max;

}

void Display()

{

int i;

struct SymbTab \*p;

p=first;

fprintf(yyout,"\n\tSYMBOL\t\t\t\tTOKEN\t\t\t\tAttribute Number\n");

for(i=0;i<size;i++)

{

fprintf(yyout,"\t%s\t\t\t\t%s\t\t\t\t%d\n",p->symbol,p->label,p->addr);

p=p->next;

}

}

int yywrap()

{

return (1);

}

=====================================================================

**Function Detection Part :**

The function detection part is implemented in the above code. The while loop detection part is not implemented because the logic used for detection of function is same as that of logic required for loop detection, which can produce ambiguity. The opening and closing of the curly braces is the logic used for function detection.

**TEST CASES** **:**

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**TEST CASE - 1 [ Without Errors]**

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include<stdio.h>

void func(){

//Single line comment

}

int main(){

int p, a, integer;

char c;

float b;

/\*

Here is the

multiline comment error

\*/

scanf("%d",&a );

scanf("%f",&b );

if(a == 0){

printf("a is zero\n" );

}

else

{

printf("a is non zero\n");

}

while(a){

while(b){

printf("Inner while loop\n");

b=b-1.0;

}

printf("Outer while loop\n");

a/=2;

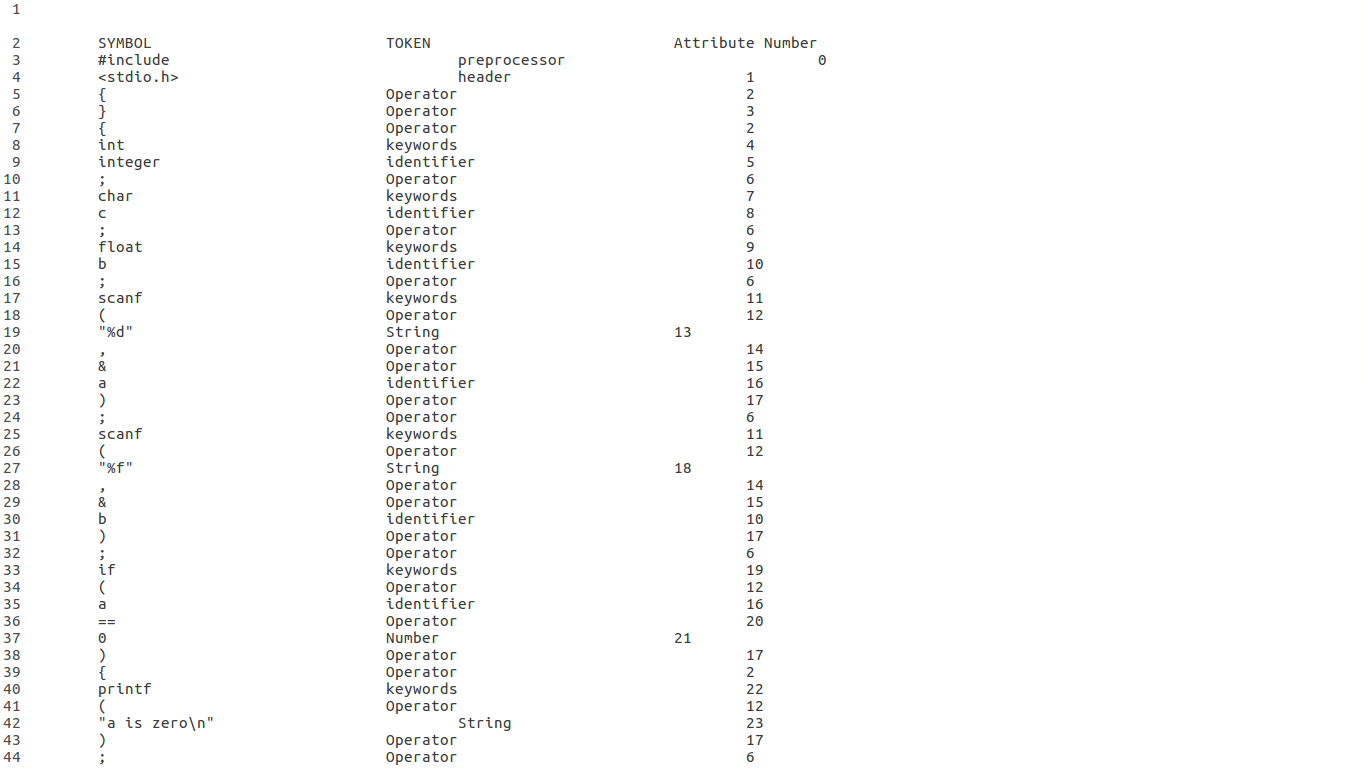
}

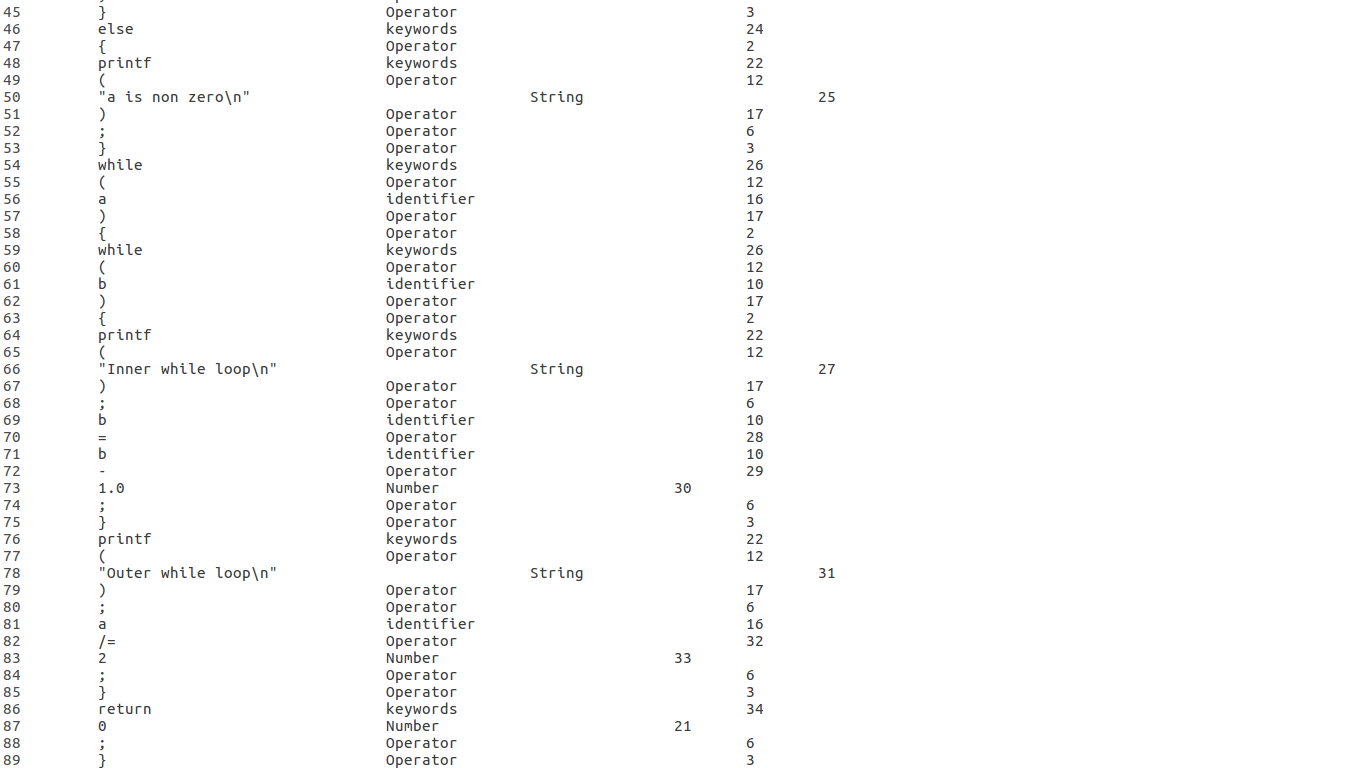
return 0;

}

}

**Output -** Symbol Table





+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**TEST CASE - 2 [ Illegal Token Error]**

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include<stdio.h>

int func();

int main(){

$int integer; //ERROR -Illegal Token

char c;

float b;

/\*

Here is the

multiline comment

\*/

printf("Give a(int) and b(float) values:\n");

scanf("%d", &a);

scanf("%f", &b);

printf("q value= %d\n", q);

if(a == q){

printf("a is equal to q\n" );

}

else

{

printf("a is not equal to q\n");

}

while{

while(b){

printf("Inner while loop\n");

b=b-1.0;

}

printf("Outer while loop\n");

a/=2;

}

return 0;

}

}

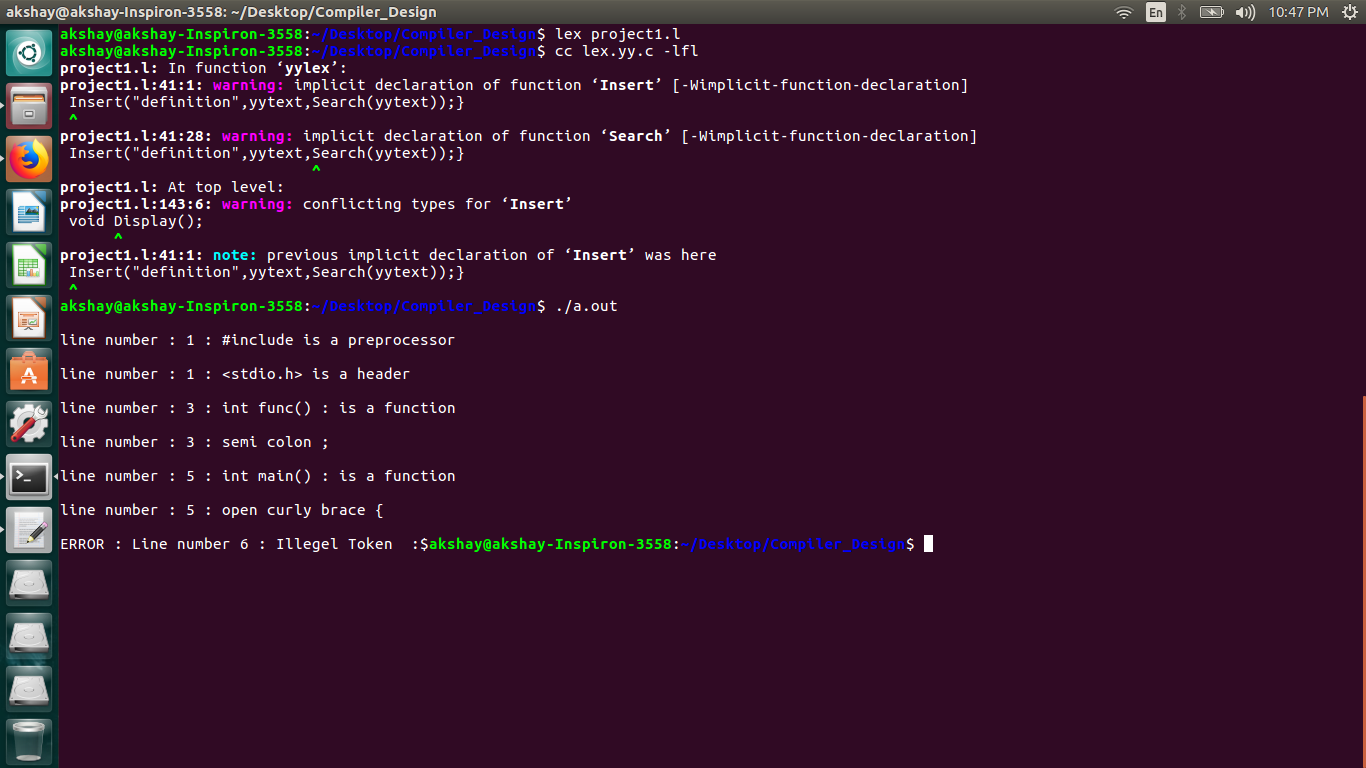
int func(){

printf("Function implementation");

return(1);

}

**Output -**



+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**TEST CASE - 3 [ Comment Error]**

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include<stdio.h>

#include<string.h>

#include<ctype.h>

#define X 10

// first program ever!!!!

void main()

{

printf("\nHello World");

do

{

/\* do something random/\*

i said do something random

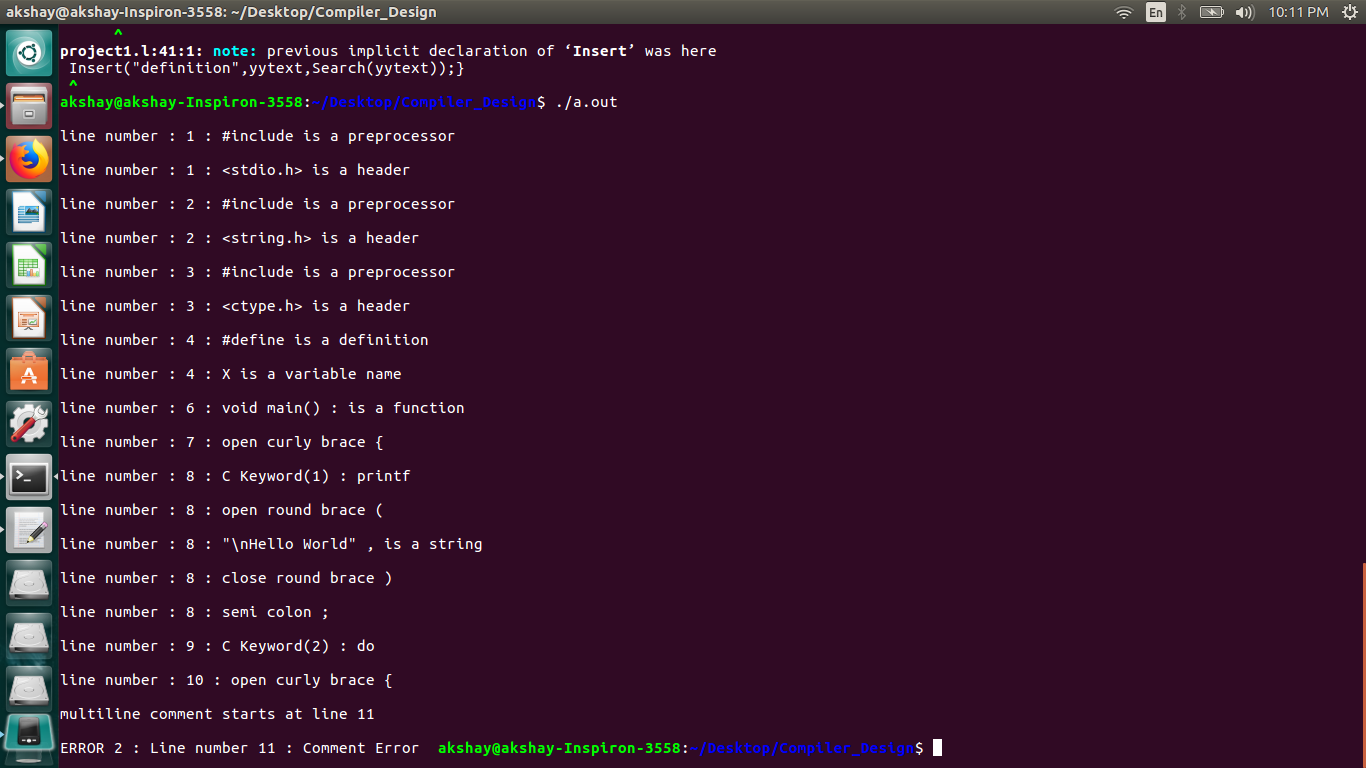
nevermind …

COMMENT ERROR \*/

}while(1);

}

**Output -**



+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**TEST CASE - 4 [ Invalid Identifier Error]**

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#include<stdio.h>

int x1;

void recurse()

{

recurse();

}

void main()

{

int c, t;

1char p; //ERROR- Invalid Identifier

recurse();

// recurse();

recurse();

}

**Output -**

