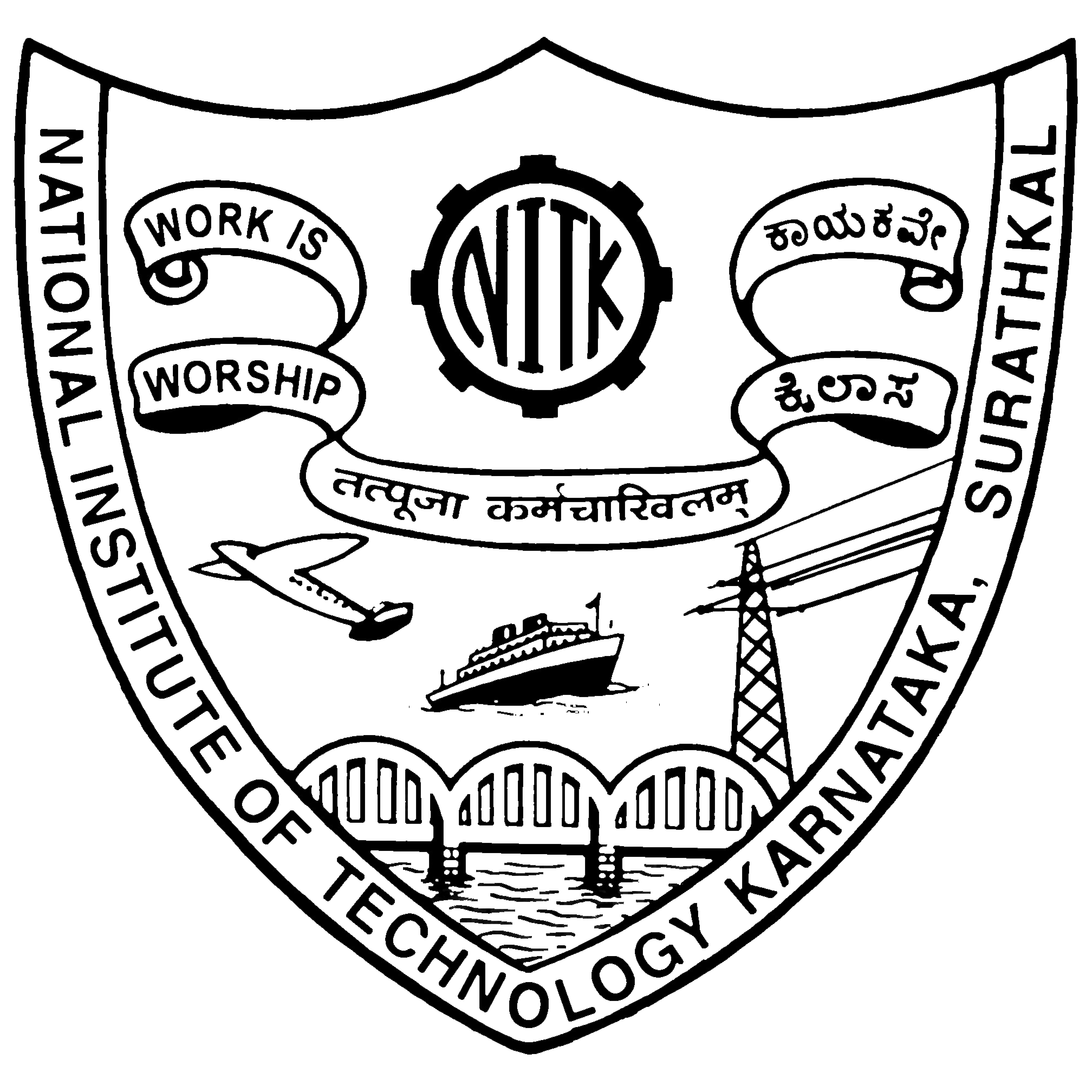
**Lexical Analyzer for the C Language**



National Institute of Technology Karnataka Surathkal

**Date:** 25/01/2017

**Submitted To:** Uma Priya

**Group Members:**

Sheetal Shalini (14CO142)

Navya R S (14CO126)

**Abstract**

*A compiler is a* [*computer program*](https://en.wikipedia.org/wiki/Computer_program) *that transforms* [*source code*](https://en.wikipedia.org/wiki/Source_code) *written in a* [*programming*](https://en.wikipedia.org/wiki/Programming_language) *language (the source language) into another computer language (the target language), with the latter often having a binary form known as* [*object code*](https://en.wikipedia.org/wiki/Object_code)*. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages or passes, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. 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. Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler. We have designed a lexical analyzer for the C language using lex. It takes as input a C code and outputs a stream of tokens. The tokens displayed as part of output include Keyword, Identifier, Constant, Operator, Special Character, Header, Format Specifier, Array, Single Line Comment, Multi Line Comment, Preprocessor Directive, Pre Defined Function, User Defined Function and Main Function. The token, the type of token and the line number of the token in the C code are being displayed. The line number is displayed so that it is easier to debug the code for errors. Errors in single line comments, multi line comments, nested comments, unmatched parenthesis and incomplete strings are displayed along with line numbers. We have also coded the symbol table, which has the columns Serial Number, Token, <Attribute, Attribute Number> and Memory Location.*

i

# **Contents**:

* Introduction --------------------------------------------------------------------- 1
  + Lexical Analyzer -------------------------------------------------------------------- 1
  + Flex Script --------------------------------------------------------------------- 1
  + C Program --------------------------------------------------------------------- 1
* Design of Programs --------------------------------------------------------------------- 2
  + Code --------------------------------------------------------------------- 2
  + Explanation --------------------------------------------------------------------- 18
* Test Cases ---------------------------------------------------------------------- 20
  + Without Errors --------------------------------------------------------------------- 20
  + With Errors --------------------------------------------------------------------- 22
* Implementation ----------------------------------------------------------------------- 25
* Results ----------------------------------------------------------------------- 27
* Future work ----------------------------------------------------------------------- 27
* References ------------------------------------------------------------------------ 27

**Introduction**

**Lexical Analysis**

In computer science, lexical analysis is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an identified "meaning"). A program that performs lexical analysis may be called a lexer, tokenizer, or scanner (though "scanner" is also used to refer to the first stage of a lexer). Such a lexer is generally combined with a parser, which together analyze the syntax of programming languages, web pages, and so forth.

**Flex Script**

The script written by us is a program that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

The structure of our flex script is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

*Definition section*

*%%*

*Rules section*

*%%*

*C code section*

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

**C Program**

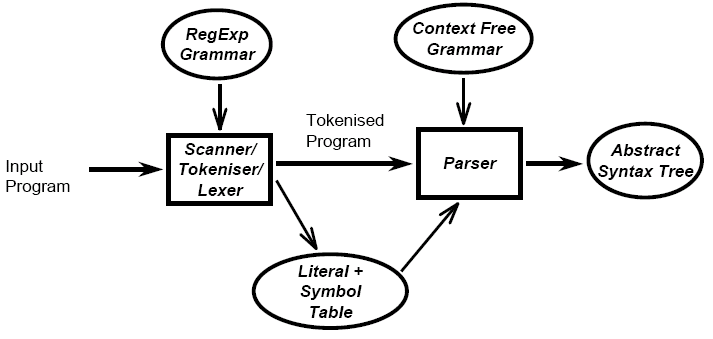
This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned in account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input.

Lexical analysis only takes care of parsing the tokens and identifying their type. For this reason, we have assumed the C program to be syntactically correct and we generate the stream of tokens as well as the symbol table from it.

1

**Design of Programs**

**Flow :**

****

Lexical Analyzer generates the tokenized program and symbol table for the input C program.

**Code:**

**Lex Code : (parser.l file)**

*alpha [A-Za-z]*

*digit [0-9]*

*und [\_]*

*space [ ]*

*%{*

*int yylineno;*

*#define IF 1*

*#define ELSE 2*

*#define INC 3*

*#define WHILE 4*

*#define INT 5*

*#define FLOAT 6*

*2*

*#define VOID 9*

*#define RETURN 10*

*#define MAIN 11*

*#define BREAK 56*

*#define ID 12*

*#define NUM 13*

*#define LE 14*

*#define GE 15*

*#define EQ 16*

*#define NE 17*

*#define OR 18*

*#define AND 19*

*#define ASS 20*

*#define PLUS 21*

*#define SUB 22*

*#define MULT 23*

*#define DIV 24*

*#define MOD 25*

*#define BA 26*

*#define BO 27*

*#define BN 28*

*#define L 29*

*#define G 30*

*#define PP 31*

*#define MM 32*

*#define SEMICOLON 33*

*#define B1 34*

*#define B2 35*

*3*

*#define C1 36*

*#define C2 37*

*#define D1 38*

*#define D2 39*

*#define COMMA 40*

*#define Q 41*

*#define COLON 57*

*#define FULLSTOP 58*

*#define HEAD 42*

*#define D 43*

*#define F 44*

*#define CA 45*

*#define CSA 46*

*#define ARR 47*

*#define SLC 48*

*#define MLCO 49*

*#define MLCC 50*

*#define DEF 51*

*#define PRINTF 52*

*#define SCANF 53*

*#define FUNC 54*

*#define MAINFUNC 55*

*%}*

*%%*

*4*

*\n {yylineno++;}*

*"main(void)" return MAINFUNC;*

*"main()" return MAINFUNC;*

*"main(int argc, char \*\*argv)" return MAINFUNC;*

*"main(int argc, char \*argv[])" return MAINFUNC;*

*"return" return RETURN;*

*void return VOID;*

*break return BREAK;*

*if return IF;*

*else return ELSE;*

*printf return PRINTF;*

*scanf return SCANF;*

*#include<stdio.h> return HEAD;*

*#include<string.h> return HEAD;*

*"#define {alpha}({alpha}|{digit}|{und})\* {digit}+" return DEF;*

*"#define {alpha}({alpha}|{digit}|{und})\* ({digit}+)\.({digit}+)" return DEF;*

*"#define {alpha}({alpha}|{digit}|{und})\* {alpha}({alpha}|{digit}|{und})\*" return DEF;*

*"//" return SLC;*

*"/\*" return MLCO;*

*"\*/" return MLCC;*

*{digit}+ return NUM;*

*({digit}+)\.({digit}+) return NUM;*

*"<=" return LE;*

*">=" return GE;*

*"==" return EQ;*

*"!=" return NE;*

*"||" return OR;*

*"&&" return AND;*

*while return WHILE;*

*";" return SEMICOLON;*

*":" return COLON;*

*"." return FULLSTOP;*

*5*

*int return INT;*

*float return FLOAT;*

*"(" return B1;*

*")" return B2;*

*"{" return C1;*

*"}" return C2;*

*"=" return ASS;*

*"+" return PLUS;*

*"-" return SUB;*

*"\*" return MULT;*

*"/" return DIV;*

*"%" return MOD;*

*"%d" return D;*

*"%f" return F;*

*"\"" return Q;*

*"," return COMMA;*

*",&" return CA;*

*", &" return CSA;*

*"&" return BA;*

*"|" return BO;*

*"!" return BN;*

*"<" return L;*

*">" return G;*

*"++" return PP;*

*"--" return MM;*

*[ \t\n] ;*

*{alpha}({alpha}|{digit}|{und})\* return ID;*

*{alpha}({alpha}|{digit}|{und})\*\[{digit}\*\] return ARR;*

*{alpha}({alpha}|{digit}|{und})\*\(({alpha}|{digit}|{und}|{space})\*\) return FUNC;*

*"[" return D1;*

*"]" return D2;*

*. printf("NA\n");*

*6*

*%%*

*struct row*

*{*

*int sno;*

*char token[100];*

*char attr[100];*

*int attrno;*

*int addr;*

*}r[100000];*

*int main(void)*

*{*

*int i=0,ptoken,h[20]={0},qline=0,qcnt=0,x=1000,j,k,y=0,b\_c=0,b\_o=0,c\_o=0,c\_c=0,d\_o=0,d\_c=0,ntoken, vtoken,rep=0;*

*ntoken = yylex();*

*printf("\n");*

*int mlc=0,slcline=0,mlcline;*

*while(ntoken) {*

*rep = 0;*

*if(yylineno==slcline)*

*{*

*ntoken=yylex();*

*continue;*

*}*

*y=0;*

*for(k=0;k<i;k++)*

*{*

*if(!(strcmp(r[k].token,yytext)))*

*{*

*y = 1;*

*break;*

*}*

*7*

*}*

*if(y==1)*

*rep=1;*

*if(yylineno!=qline && qline!=0)*

*{*

*if(qcnt%2!=0)*

*printf("\n-------------ERROR ! INCOMPLETE STRING at Line %d------------------\n",qline);*

*qcnt=0;*

*}*

*if(b\_c > b\_o)*

*printf("ERROR ! UNMATCHED ')'\n");*

*if(c\_c>c\_o)*

*printf("ERROR ! UNMATCHED '}'\n");*

*if(d\_c>d\_o)*

*printf("ERROR ! UNMATCHED ']'\n");*

*if(rep==0 && ntoken!=48 && ntoken!=49 && ntoken!=50 && mlc==0 && qcnt%2==0)*

*{*

*r[i].sno=i+1;*

*strcpy(r[i].token,yytext);*

*r[i].addr = x;*

*}*

*if(((ntoken>=1 && ntoken<=11) || ntoken==56) && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=0;*

*strcpy(r[i].attr,"KEYWORD");*

*}*

*printf("%s\t\tKeyword----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken==12 && mlc==0)*

*8*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=1;*

*strcpy(r[i].attr,"ID");*

*}*

*printf("%s\t\tIdentifier----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken==13 && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=2;*

*strcpy(r[i].attr,"CONSTANT");*

*}*

*printf("%s\t\tConstant----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken>=14 && ntoken<=32 && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=3;*

*strcpy(r[i].attr,"OPERATOR");*

*}*

*printf("%s\t\tOperator----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken>=33 && ntoken<=41 && mlc==0)*

*{*

*if(ntoken==41)*

*{*

*qcnt++;*

*qline=yylineno;*

*9*

*}*

*if(ntoken==34)*

*b\_o++;*

*if(ntoken==35)*

*b\_c++;*

*if(ntoken==36)*

*c\_o++;*

*if(ntoken==37)*

*c\_c++;*

*if(ntoken==38)*

*d\_o++;*

*if(ntoken==39)*

*d\_c++;*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=4;*

*strcpy(r[i].attr,"SPECIAL CHAR");*

*}*

*printf("%s\t\tSpecial Character----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken==42 && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=5;*

*strcpy(r[i].attr,"HEADER");*

*}*

*printf("%s\t\tHeader----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken>=43 && ntoken<=44 && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*10*

*{*

*ptoken=6;*

*strcpy(r[i].attr,"FORMAT SPECIFIER");*

*}*

*printf("%s\t\tFormat Specifier----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken>=45 && ntoken<=46 && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=12;*

*strcpy(r[i].attr,"VARIABLE LOCATION");*

*}*

*printf("%s\t\tVariable Location----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken==47 && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=7;*

*strcpy(r[i].attr,"ARRAY");*

*}*

*printf("%s\t\tArray----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken==48 && mlc==0)*

*{*

*ptoken=15;*

*printf("%s\t\tSingle Line Comment----------Line %d\n",yytext,yylineno);*

*slcline=yylineno;*

*}*

*else if(ntoken==49)*

*{*

*11*

*mlc=1;*

*printf("%s\t\tMulti Line Comment Start----------Line %d\n",yytext,yylineno);*

*mlcline = yylineno;*

*}*

*else if(ntoken==50 && mlc==1)*

*{*

*mlc=0;*

*printf("%s\t\tMulti Line Comment End----------Line %d\n",yytext,yylineno);*

*mlcline=0;*

*}*

*else if(ntoken==50 && mlc==0)*

*printf("\n---------------ERROR! UNMATCHED NESTED END COMMENT-------------\n");*

*else if(ntoken==51 && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=8;*

*strcpy(r[i].attr,"Preprocessor Directive");*

*}*

*printf("%s\t\tPreprocessor Directive----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken>=52 && ntoken<=53 && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=9;*

*strcpy(r[i].attr,"Pre Defined Function");*

*}*

*printf("%s\t\tPre Defined Function----------Line %d\n",yytext,yylineno);*

*}*

*12*

*else if(ntoken==54 && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=10;*

*strcpy(r[i].attr,"User Defined Function");*

*}*

*printf("%s\t\tUser Defined Function----------Line %d\n",yytext,yylineno);*

*}*

*else if(ntoken==55 && mlc==0)*

*{*

*if(rep==0 && qcnt%2==0)*

*{*

*ptoken=11;*

*strcpy(r[i].attr,"Main Function");*

*}*

*printf("%s\t\tMain Function----------Line %d\n",yytext,yylineno);*

*}*

*if(rep==0 && ntoken!=48 && ntoken!=49 && ntoken!=50 && mlc==0 && qcnt%2==0)*

*{*

*h[ptoken]++;*

*r[i].attrno = h[ptoken];*

*i++;*

*x=x+4;*

*}*

*ntoken=yylex();*

*}*

*if(mlc==1)*

*printf("\n--------------------ERROR! UNMATCHED COMMENT starting at Line %d------------\n",mlcline);*

*if(b\_c!=b\_o)*

*printf("\n--------------------ERROR ! UNMATCHED '('---------------------\n");*

*13*

*if(c\_c!=c\_o)*

*printf("\n-------------------------ERROR ! UNMATCHED '{'----------------\n");*

*if(d\_c!=d\_o)*

*printf("\n--------------------ERROR ! UNMATCHED '['----------------------\n");*

*printf("\n------------Symbol Table---------------------\n\nSNo\tToken\t\tAttribute\t\tMemory Location\n\n");*

*for(j=0;j<i;j++)*

*printf("%d\t%s\t\t< %s , %d >\t\t%d\n",r[j].sno,r[j].token,r[j].attr,r[j].attrno,r[j].addr);*

*return 0;*

*}*

*int yywrap(void)*

*{*

*return 1;*

*}*

**Input C program (binary\_search.c file)**

*#include<stdio.h>*

*int main(){*

*int a[10],i=0,n,m,c=0,l,u,mid;*

*printf("Enter size of array");*

*scanf("%d",&n);*

*printf("Enter elements in ascending order:");*

*while(i<n){*

*scanf("%d",&a[i]);*

*i++;*

*}*

*printf("Enter key");*

*14*

*scanf("%d",&m);*

*l=0,u=n-1;*

*while(l<=u){*

*mid=(l+u)/2;*

*if(m==a[mid]){*

*c=1;*

*break;*

*}*

*else if(m<a[mid])*

*u=mid-1;*

*else*

*l=mid+1;*

*}*

*if(c==0)*

*printf("Number not found");*

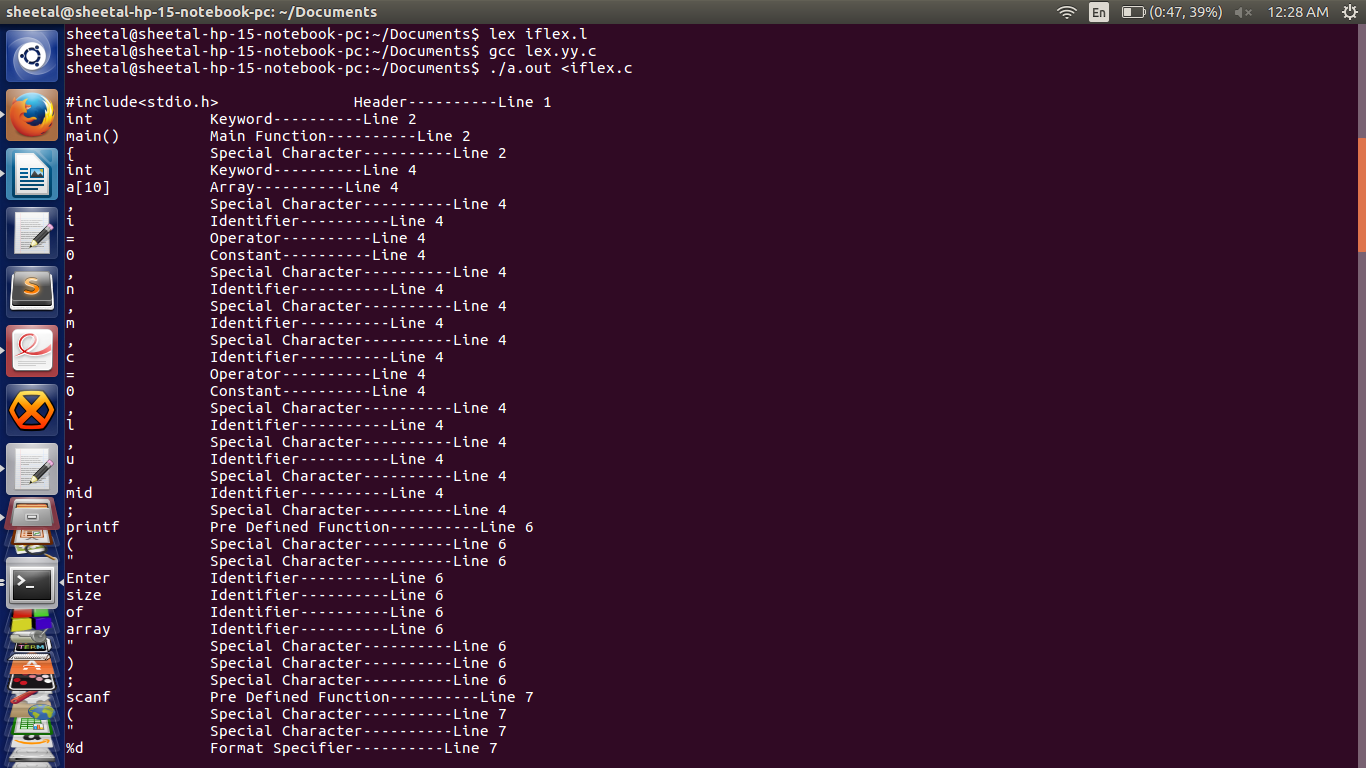
*else*

*printf("Number found");*

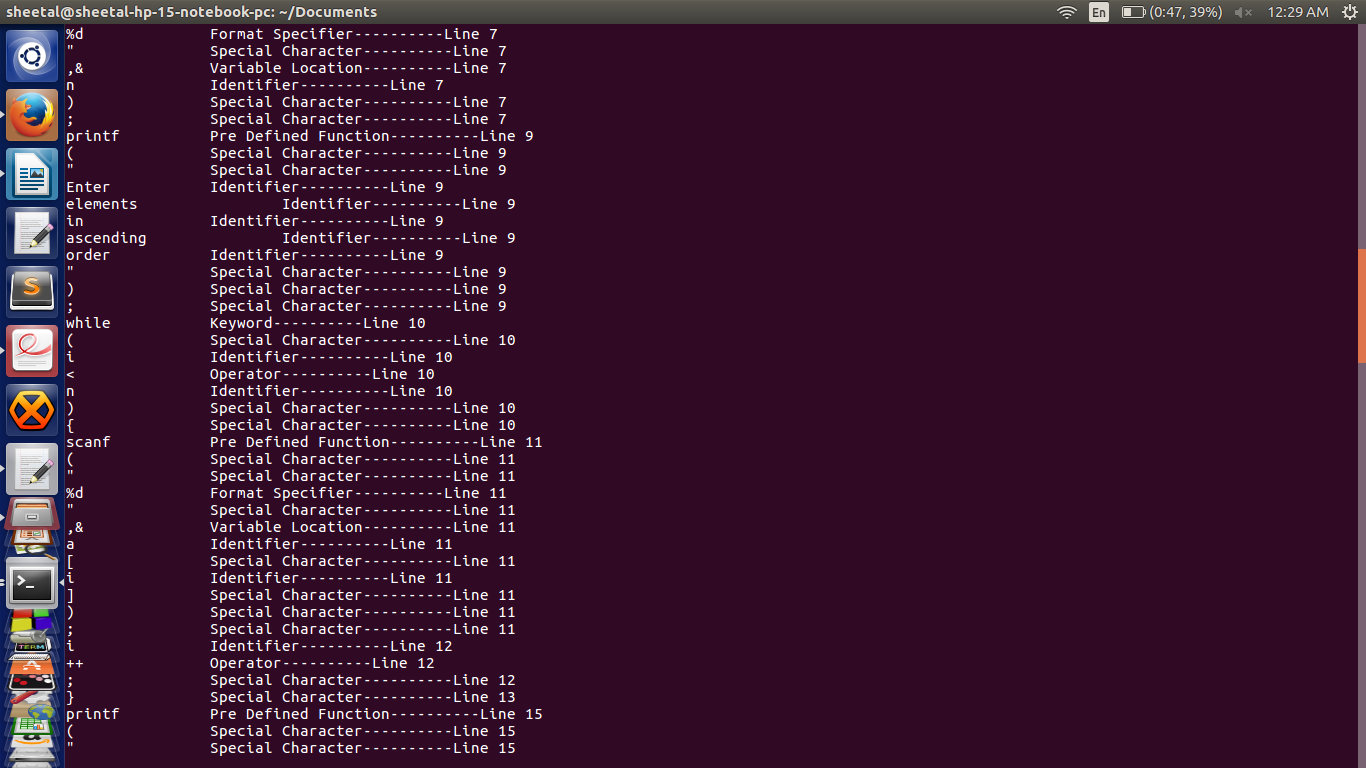
*return 0;*

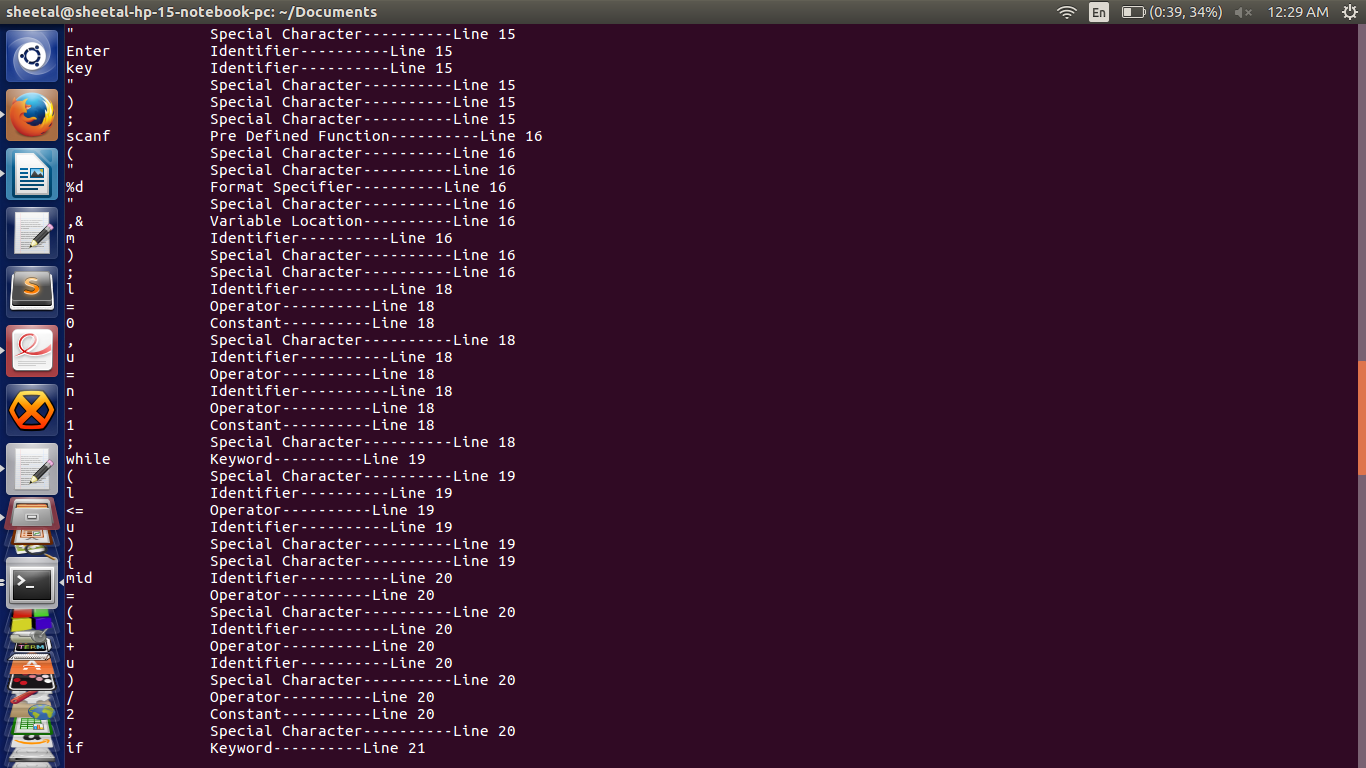
*}*

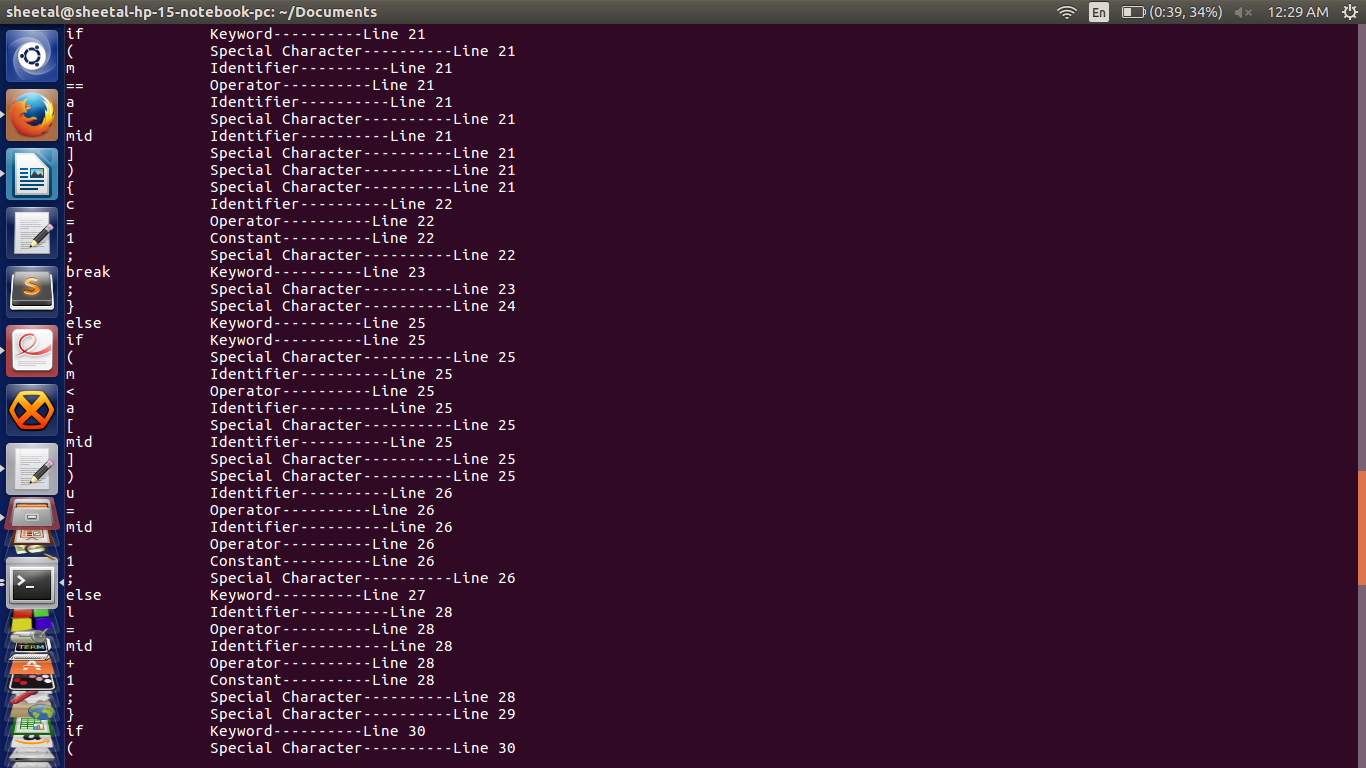
**Output :**

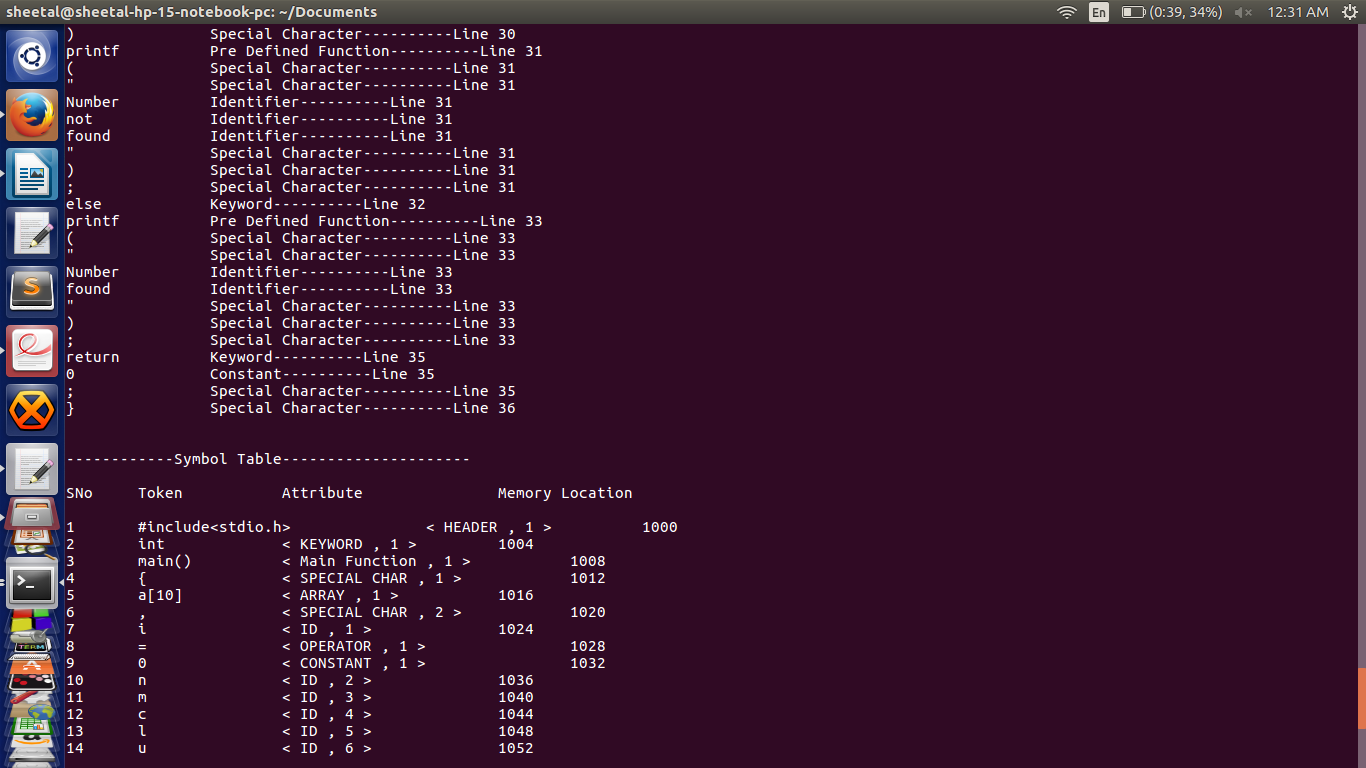


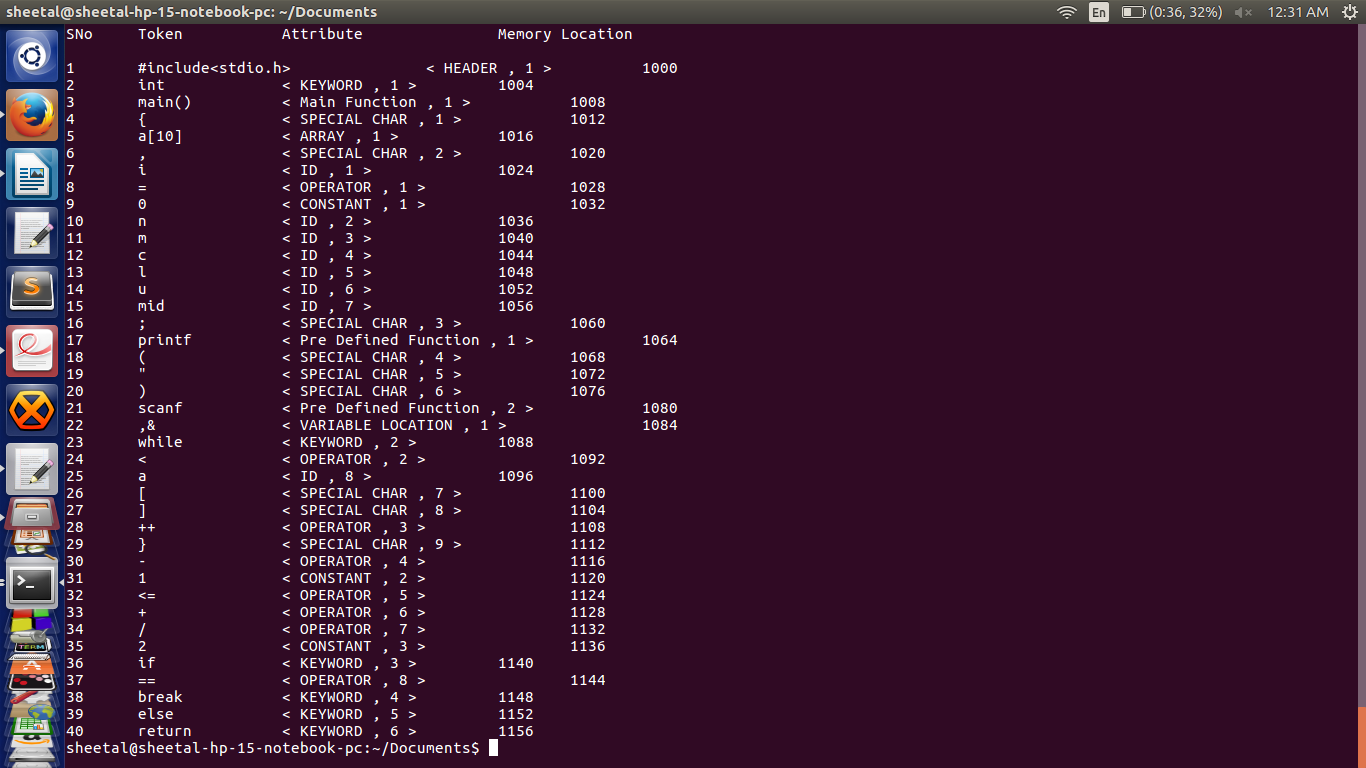
15

****

****

 16



17

**Explanation :**

**Files :**

1. **Parser.l** : Lex file which generates the stream of tokens and symbol table.

2. **Binary\_search.c** : The input C program

The flex script recognises the following classes of tokens from the input:

* Pre-processor instructions

Statements processed : *#include<stdio.h>, #define var1 var2*

Token generated : Header / Preprocessor Directive

* Single-line comments

Statements processed : *//...........*

Token generated : Single Line Comment

* Multi-line comments

Statements processed : */\*...........\*/, /\*.../\*...\*/*

Token generated : Multi Line Comment

* Errors for unmatched comments

Statements processed : */\*..........*

Token generated : Error with line number

* Errors for nested comments

Statements processed : */\*....../\*....\*/....\*/*

Token generated : Error with line number

* Parentheses (all types)

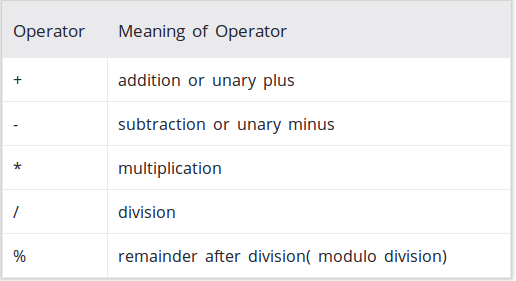
Statements processed : *(..), {..}, [..] (without errors)*

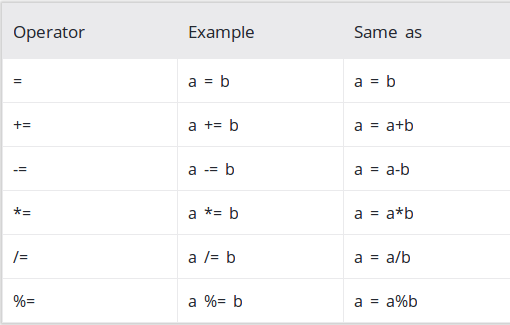
*(..)..), {..}..}, [..]..], (..., {..., [... (with errors)*

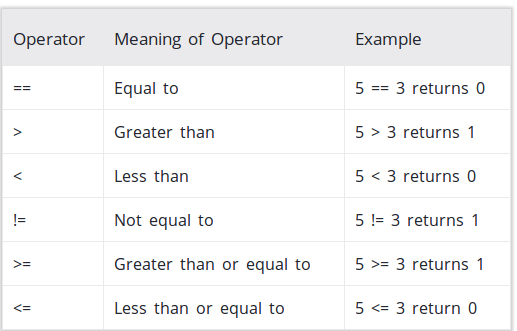
Tokens generated : Parenthesis (without error) / Error with line number (with error)

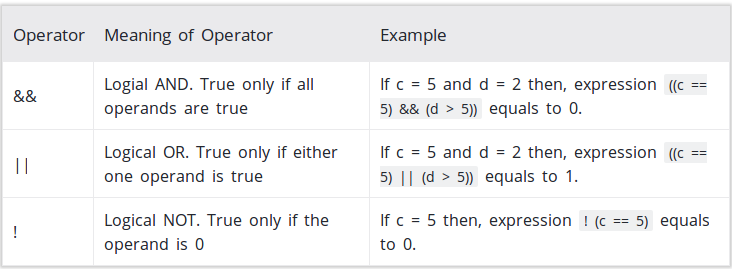
18

* Operators









19

* Literals

Statements processed : *int, float*

Tokens generated : Keyword

* Errors for incomplete strings

Statements processed : *char a[]= ”abcd*

Tokens generated : Error Incomplete string and line number

* Keywords

Statements processed : *if, else, void, while, do, int, float, break, return* and so on.

Tokens generated : Keyword

* Identifiers

Statements processed : *a, abc, a\_b, a12b4*

Tokens generated : Identifier

**Test Cases:**

**Without Errors:**

**1. Code :**

*#include<stdio.h>*

*void main()*

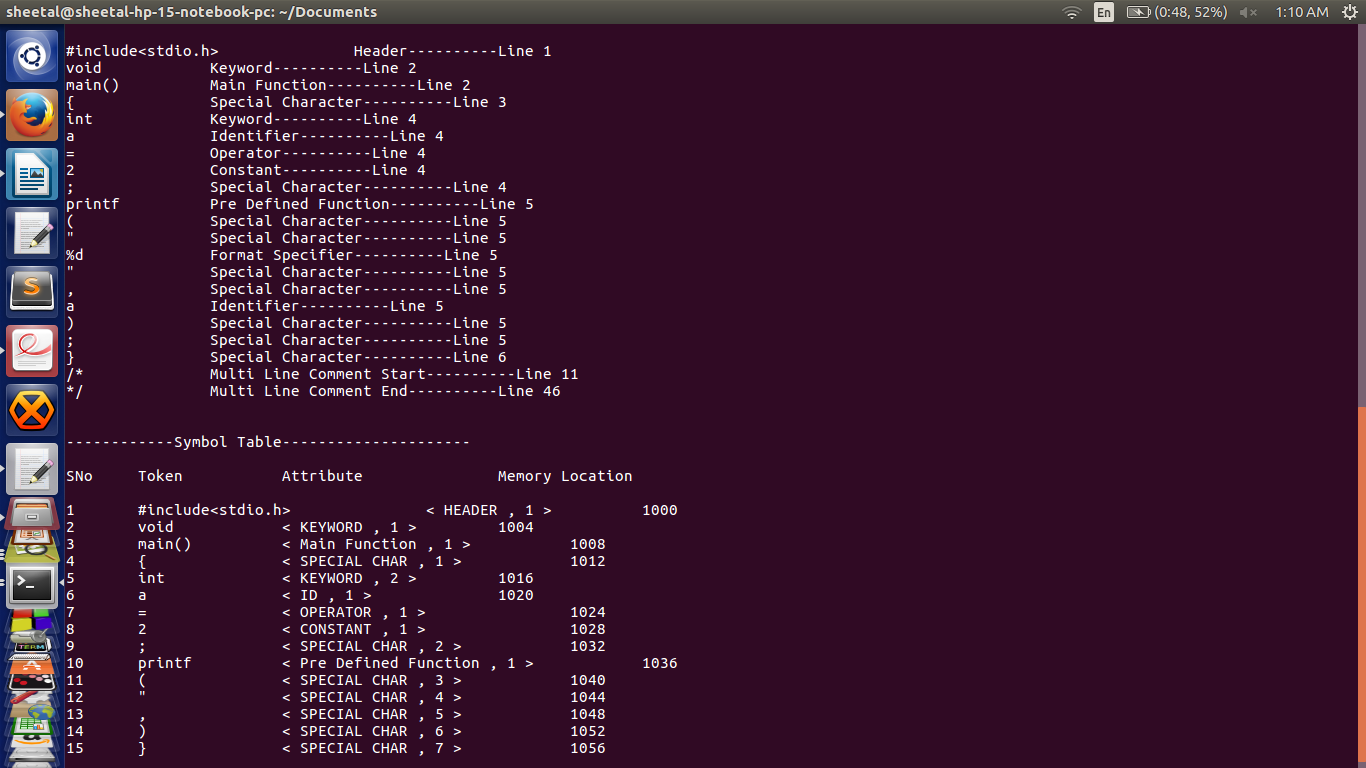
*{*

*int a=2;*

*printf("%d",a);*

*}*

**Output :**



20

**2. Code (with single & multi line comments)**

*#include<stdio.h>*

*void main()*

*{*

*int a=2;*

*//printf("%d",a);*

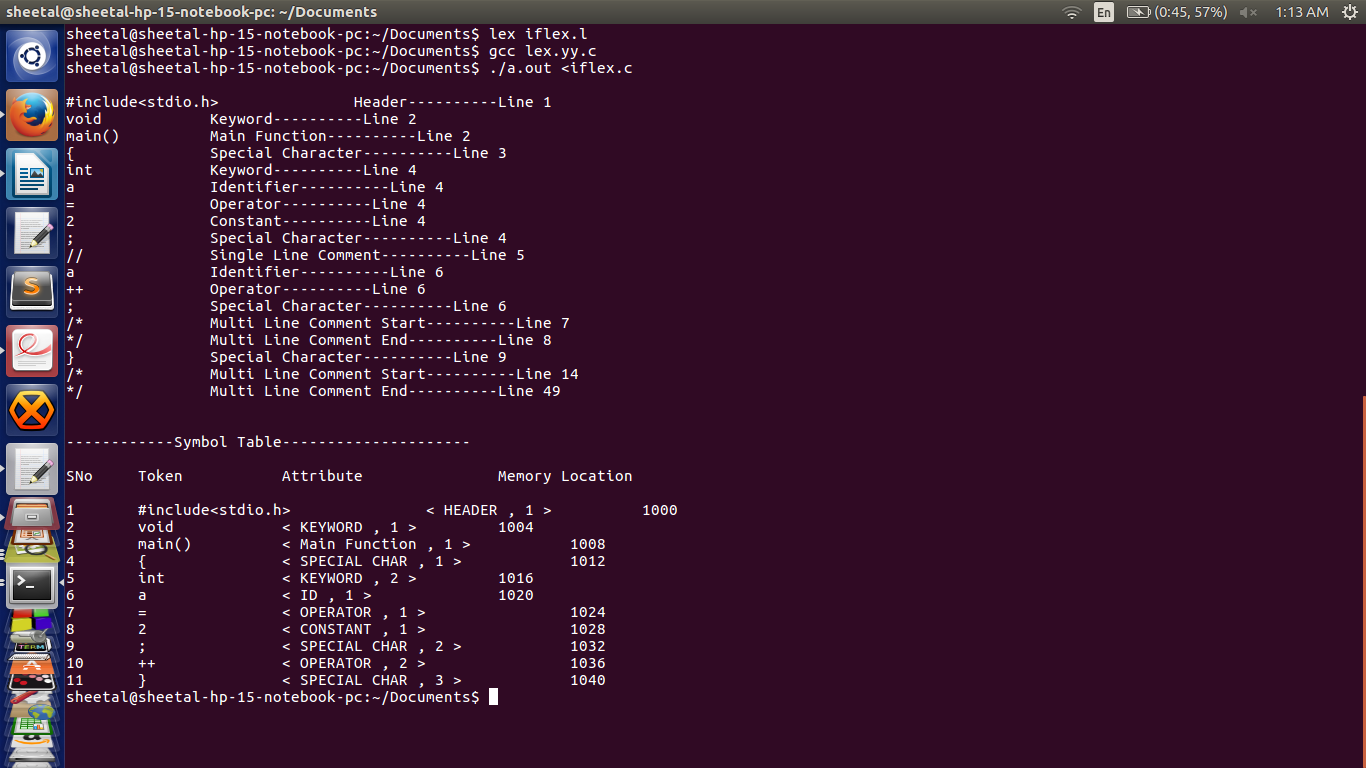
*a++;*

*/\* b=2;*

*c=4; \*/*

*}*

**Output :**

****

21

|  |  |  |
| --- | --- | --- |
| Statement | Output | Status |
| Int a =2; | Int : Keyword  a : Identifier  = : Assignment  2 : Constant  ; : Semicolon | PASS |
| // printf(“%d”,a); | // : Single Line Comment | PASS |
| /\* b=2; c=4; \*/ | /\* : Multi Line Comment start  \*/ : Multi Line comment end | PASS |

**With Errors :**

**1. Code (with nested multiline comment errors)**

*#include<stdio.h>*

*void main()*

*{*

*int a=2;*

*/\*printf("%d",a);*

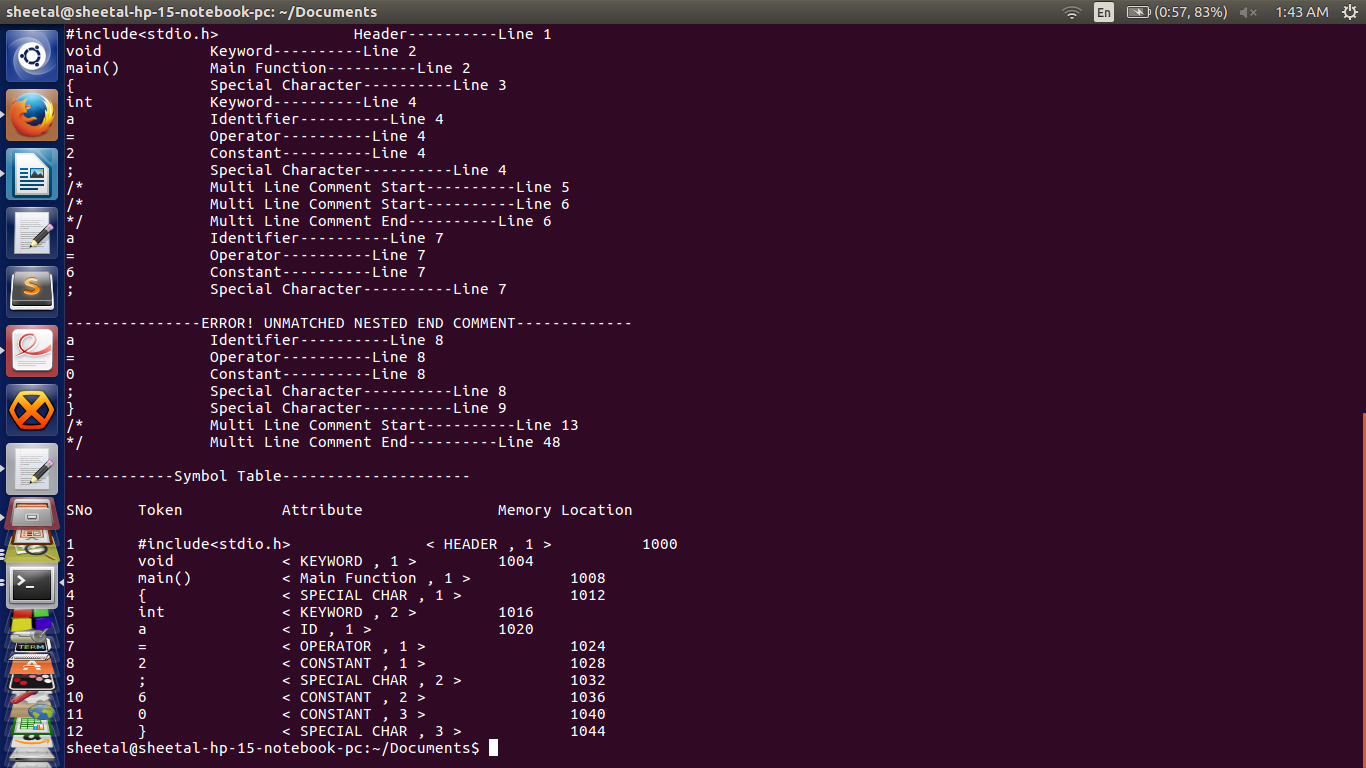
*/\*a++;\*/*

*a = 6;\*/*

*a = 0;*

*}*

**Output:**

 22

**2. Code (with unmatched multi line comment errors)**

*#include<stdio.h>*

*void main()*

*{*

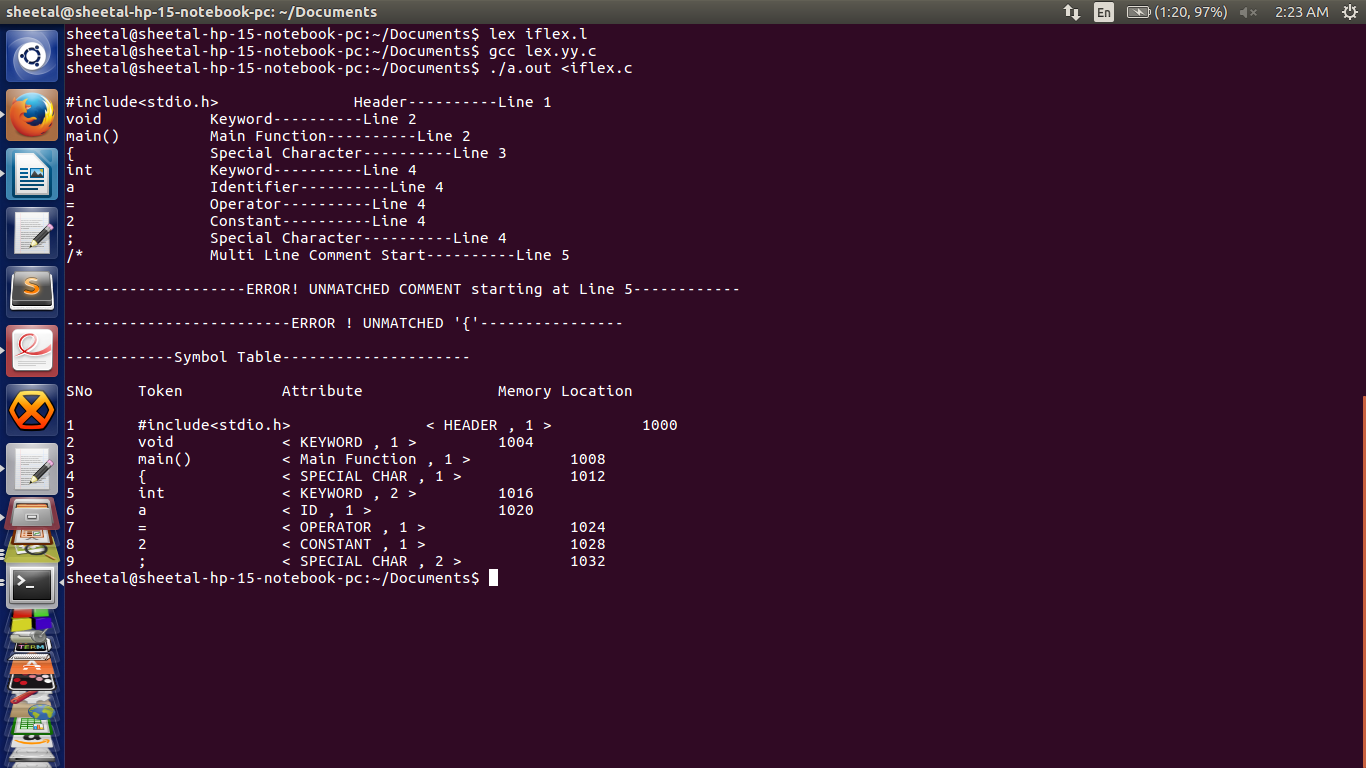
*int a=2;*

*/\*printf("%d",a);*

*a++;*

*}*

**Output :**

 23

**3.Code (with unmatched quotes and unmatched paranthesis errors)**

*#include<stdio.h>*

*void main()*

*{*

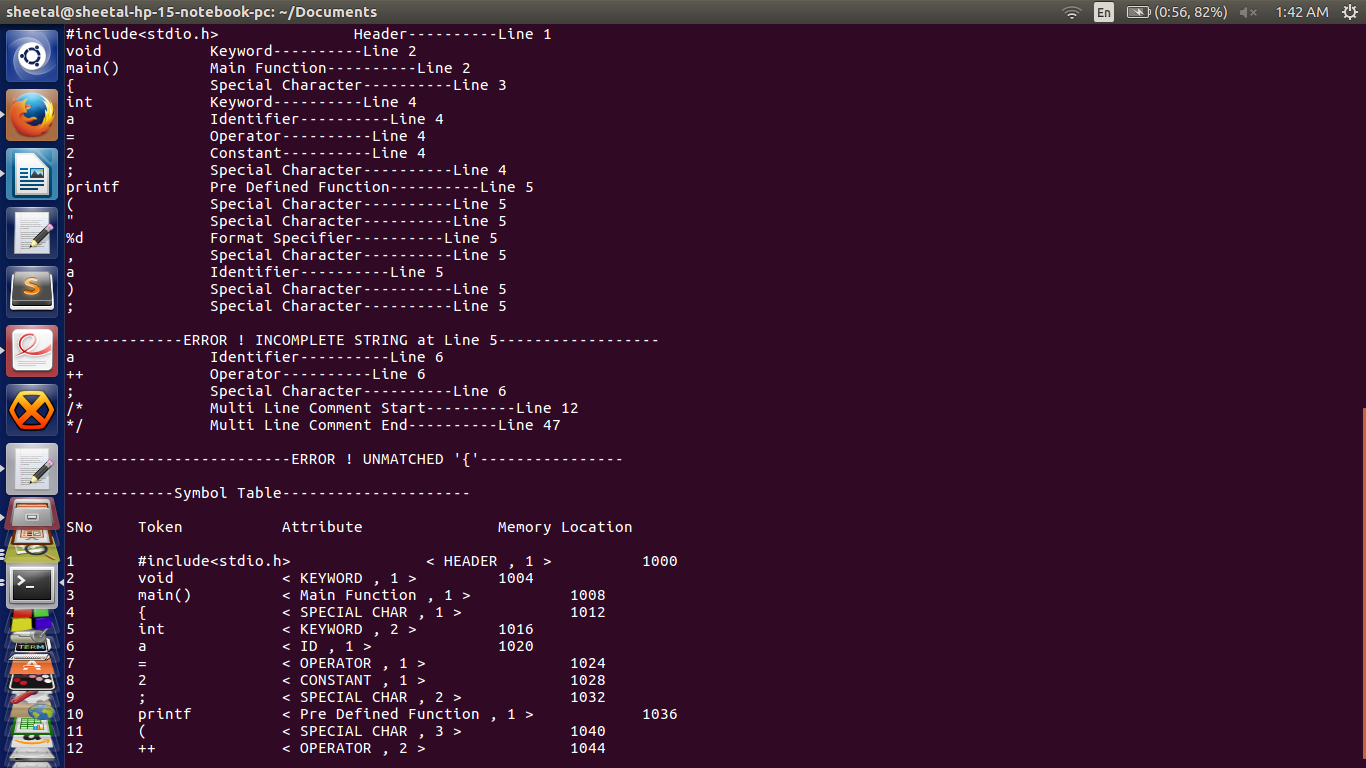
*int a=2;*

*printf("%d,a);*

*a++;*

**Output:**

24



|  |  |  |
| --- | --- | --- |
| Statement | Output | Status |
| /\*printf("%d",a);  /\*a++;\*/  a = 6;\*/ | ERROR ! Unmatched nested end comment | PASS |
| /\*printf("%d",a); | ERROR ! Unmatched comment starting at line 5 | PASS |
| printf("%d,a); | ERROR ! Incomplete string at line 5 | PASS |
| {  int a; | ERROR ! Unmatched ‘{‘ | PASS |

**Implementation**

The Regular Expressions for most of the features of C are fairly straightforward. However, a few features require a significant amount of thought, such as:

* **The Regex for Identifiers:** The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.

{alpha}({alpha}|{digit}|{und})\*

25

Where,

alpha [A-Za-z]

digit [0-9]

und [\_]

space [ ]

* **Multiline comments are supported:** This has been supported by checking the occurence of ‘/\*’ and ‘\*/’ in the code. The statements between them has been excluded. Errors for unmatched and nested comments have also been displayed.
* **Literals:** Different regular expressions have been implemented in the code to support all kinds of literals, i.e integers, floats, etc.

*Float :* ({digit}+)\.({digit}+)

* **User Defined Functions :**

{alpha}({alpha}|{digit}|{und})\*\(({alpha}|{digit}|{und}|{space})\*\)

* **Arrays:**

{alpha}({alpha}|{digit}|{und})\*\[{digit}\*\]

Where,

alpha [A-Za-z]

digit [0-9]

und [\_]

space [ ]

* **Error Handling for Incomplete String:** Open and close quote missing, both kind of errors have been handled in the rules written in the script.
* **Error Handling for Nested Comments:** This use-case has been handled by checking for occurrence of multiple successive ‘/\*’ or ‘\*/’ in the C code, and by omitting the text in between them.

At the end of the token recognition, the lexer prints a list of all the tokens present in the program. We use the following technique to implement this:

* We have assigned unique integers to all different kinds of tokens present in the C code.
* Based on these integers, we have displayed the type of the token.
* For storing these tokens and their attributes in the symbol table, we have defined a structure.

struct row

{

int sno;

char token[100];

char attr[100];

int attrno;

int addr;

}r[100000];

* As and when successive tokens are encountered, their respective values are stored in the structure and then later displayed.
* We also have functionalities for checking and accordingly omitting duplicate entries in the symbol table.

26

* In the end, each token is printed along with its type and line number.
* Errors like unmatched multi line comment, nested multi line comments, incomplete strings and unmatched parenthesis are also displayed along with their line numbers.
* The symbol table is displayed, having columns Serial Number, Token, attribute, attribute number and memory address.

**Results**

1. Token ---- Token Type ---- Line Number

2. Symbol Table :

Serial Number ---- Token ---<Attribute, Attribute Number>----Memory Location

**Future work:**

The flex script presented in this report takes care of all the rules of C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of C language and making it more efficient.

Features to be added :

1. Nested if else statement

2. Nested while loop

3. Structures

4. enum

**References**

1. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

2. Lex and Yacc By John R. Levine, Tony Mason, Doug Brown

3.<http://www.slideshare.net/Tech_MX/symbol-table-design-compiler-construction>

4. https://en.wikipedia.org/wiki/Symbol\_table

5.http://www.isi.edu/~pedro/Teaching/CSCI565-Spring11/Practice/SDT-Sample.pdf

27