Lexical Analyzer for the C Language



National Institute of Technology Karnataka Surathkal

**Date: 21/01/2020**

**Submitted To:**

**Ms. Sushmita**

**Group Members:**

**Abstract:**

*Aim*

To design and implement a lexical analyser using lex for a subset of the C language.

*Features supported*

1. Variable data types - int, char along with its sub types - short, long, signed, unsigned.

2. Looping constructs - while loops along with nested while loops.

3. Identification and classification of tokens.

4. Identification of functions accepting a single parameter.

5. Maintenance of a symbol table and a constant table using hashing techniques.

6. Error detection for multi-line comments and nested comments that are not terminated before the end of the program.

7. Checking for strings that does not end before the end of a statement and displaying corresponding error message.

*Nature of output*

1. Error messages for the errors handled.

2. The token will be displayed along with the type:

- Keyword

- Identifier

- Literal

- Operator

- Punctuator

3. Symbol table

4. Constant table

**Contents**:

• Introduction Page No

o Lexical Analyzer

o Flex Script

o C Program

• Design of Programs

o Code

o Explanation

• Test Cases

o Without Errors

o With Errors

• Implementation

• Results / Future work

• References

**List of Figures and Tables:**

1. Table 1: Test Cases without errors

2. Table 2: Test cases with errors

3. Figure 1: Input for: For loop with valid and invalid strings

4. Figure 2: Output for: For loop with valid and invalid strings

5. Figure 3: Input for: Various forms of multi-line comment

6. Figure 4: Output for: Various forms of multi-line comment

7. Figure 5: Input for: Sample C program for binary search.

8. Figure 6: Output for: Sample C program for binary search.

**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.

The script also has an option to take standard input instead of taking input from a file.

**Design of Programs**

**Code:**

Lex Code : (scanner.l file)

|  |
| --- |
| %{ |
|  | int yylineno; |
|  |  |
|  | //Keywords |
|  | #define WHILE 1 |
|  | #define VOID 2 |
|  | #define RETURN 3 |
|  | #define MAINFUNC 4 |
|  | #define BREAK 5 |
|  | #define CONTINUE 7 |
|  | #define IF 8 |
|  | #define INT 10 |
|  | #define CHAR 11 |
|  | #define UNSIGNED 12 |
|  | #define SIGNED 13 |
|  | #define LONG 14 |
|  | #define SHORT 15 |
|  | #define ELSE 16 |
|  | #define FOR 17 |
|  | #define STRUCT 18 |
|  |  |
|  | //Identifier and Constant |
|  | #define ID 20 |
|  | #define CONST 21 |
|  |  |
|  | //Operators |
|  | //Comparators |
|  | #define LE 22 |
|  | // Less than equal to |
|  | #define GE 23 |
|  | // Greater than equal to |
|  | #define EQ 24 |
|  | // Check for equality |
|  | #define NE 25 |
|  | // Not equal to check |
|  | #define L 77 |
|  | // Less than |
|  | #define G 78 |
|  | // Greater than |
|  |  |
|  | //Logical |
|  | #define OR 26 |
|  | #define AND 27 |
|  | #define NOT 28 |
|  |  |
|  | //Assignment |
|  | #define ASS 29 |
|  | // = Simple assignment operator. |
|  | #define ADDASS 30 |
|  | // += Add AND assignment operator. |
|  | #define SUBASS 31 |
|  | // -= Subtract AND assignment operator. |
|  | #define MULASS 32 |
|  | // \*= Multiply AND assignment operator. |
|  | #define DIVASS 33 |
|  | // /= Divide AND assignment operator. |
|  | #define MODASS 34 |
|  | // %= Modulus AND assignment operator. |
|  |  |
|  | //Arithmetic |
|  | #define PLUS 35 |
|  | #define SUB 36 |
|  | #define MULT 37 |
|  | #define DIV 38 |
|  | #define MOD 39 |
|  | #define PP 40 |
|  | // ++ |
|  | #define MM 41 |
|  | // -- |
|  |  |
|  | //Bitwise Ops |
|  | #define BA 42 |
|  | // Bitwise and |
|  | #define BO 43 |
|  | // Bitwise or |
|  | #define BC 44 |
|  | // Bitwise complement |
|  | #define OC 45 |
|  | //one's complement |
|  | #define LS 46 |
|  | // left shift |
|  | #define RS 47 |
|  | //right shift |
|  |  |
|  | // Miscelaneous tokens |
|  | #define SEMICOLON 53 |
|  | #define BA1 54 |
|  | // '(' bracket |
|  | #define BA2 55 |
|  | // ')' bracket |
|  | #define BB1 56 |
|  | // '[' bracket |
|  | #define BB2 57 |
|  | // ']' bracket |
|  | #define BC1 58 |
|  | // '{' bracket |
|  | #define BC2 59 |
|  | //'}' bracket |
|  | #define COMMA 60 |
|  | // ',' |
|  | #define Q 61 |
|  | // Quote " |
|  | #define SQ 62 |
|  | // Single Quote ' |
|  | #define HEAD 63 |
|  | // Header file |
|  | #define ARR 64 |
|  | // Array |
|  | #define SLC 65 |
|  | // Single comment '/' |
|  | #define MLCO 66 |
|  | // Multiline Comment Open '/\*' |
|  | #define MLCC 67 |
|  | // Multilien Comment Close '\*/' |
|  | #define DEF 68 |
|  | // Macro |
|  | #define PRINTF 69 |
|  | #define SCANF 70 |
|  | #define FUNC 71 |
|  | #define STRING 72 |
|  | #define INTCONST 73 |
|  | #define FLOATCONST 74 |
|  | #define CHARCONST 75 |
|  | #define INVALIDSTRING 76 |
|  | #define DOT 80 |
|  | %} |
|  |  |
|  | alpha [A-Z||a-z] |
|  | digit [0-9] |
|  | und [\_] |
|  | space [ ] |
|  |  |
|  | %% |
|  | \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; |
|  | while return WHILE; |
|  | printf return PRINTF; |
|  | continue return CONTINUE; |
|  | scanf return SCANF; |
|  | int return INT; |
|  | char return CHAR; |
|  | signed return SIGNED; |
|  | unsigned return UNSIGNED; |
|  | long return LONG; |
|  | short return SHORT; |
|  | const return CONST; |
|  | else return ELSE; |
|  | for return FOR; |
|  | struct return STRUCT; |
|  | define return DEF; |
|  |  |
|  | #include<{alpha}{alpha}\*\.h> return HEAD; |
|  |  |
|  | #define{space}+{alpha}({alpha}|{digit}|{und})\*{space}+{digit}+ return DEF; |
|  | #define{space}+{alpha}({alpha}|{digit}|{und})\*{space}+({digit}+)\.({digit}+) return DEF; |
|  | #define{space}+{alpha}({alpha}|{digit}|{und})\*{space}+{alpha}({alpha}|{digit}|{und})\* return DEF; |
|  |  |
|  | ({alpha}|{und})({alpha}|{digit}|{und})\* return ID; |
|  | {alpha}({alpha}|{digit}|{und})\*\[{digit}\*\] return ARR; |
|  | {digit}+ return INTCONST; |
|  | ({digit}+)\.({digit}+) return FLOATCONST; |
|  |  |
|  | \"[^\n|^\"]\*[\n] return INVALIDSTRING; |
|  |  |
|  | {alpha}({alpha}|{digit}|{und})\*\({alpha}({alpha}|{digit}|{und}|{space})\*\) return FUNC; |
|  |  |
|  | \"[^\n]\*\" return STRING; |
|  | \'{alpha}\' return CHARCONST; |
|  |  |
|  | "<=" return LE; |
|  | ">=" return GE; |
|  | "==" return EQ; |
|  | "!=" return NE; |
|  | ">" return G; |
|  | "<" return L; |
|  |  |
|  | "[|][|]" return OR; |
|  | "&&" return AND; |
|  | "!" return NOT; |
|  |  |
|  | "=" return ASS; |
|  | "+=" return ADDASS; |
|  | "-=" return SUBASS; |
|  | "\*=" return MULASS; |
|  | "/=" return DIVASS; |
|  | "%=" return MODASS; |
|  |  |
|  | "+" return PLUS; |
|  | "-" return SUB; |
|  | "\*" return MULT; |
|  | "/" return DIV; |
|  | "%" return MOD; |
|  | "++" return PP; |
|  | "--" return MM; |
|  |  |
|  | "&" return BA; |
|  | "[|]" return BO; |
|  | "~" return OC; |
|  | "<<" return LS; |
|  | ">>" return RS; |
|  |  |
|  | "//" return SLC; |
|  | "/\*" return MLCO; |
|  | "\*/" return MLCC; |
|  |  |
|  | ";" return SEMICOLON; |
|  | "(" return BA1; |
|  | ")" return BA2; |
|  | "[" return BB1; |
|  | "]" return BB2; |
|  | "{" return BC1; |
|  | "}" return BC2; |
|  | "," return COMMA; |
|  | "\"" return Q; |
|  | "'" return SQ; |
|  | \t ; |
|  | "." return DOT; |
|  | %% |
|  |  |
|  | //Data Structure for the symbol and constant table |
|  | struct symbol |
|  | { |
|  | char token[100]; // Name of the token |
|  | char type[100]; // Token type: Identifier, string constant, floating point constant etc |
|  | }symbolTable[100000], constantTable[100000]; |
|  |  |
|  | int i=0; // Number of symbols in the symbol table |
|  | int c=0; // Number of constants in the constant table |
|  |  |
|  | //Insert function for symbol/constant table |
|  | void symbolInsert(struct symbol table[], int index, char\* tokenName, char\* tokenType) |
|  | { |
|  | strcpy(table[index].token, tokenName); |
|  | strcpy(table[index].type, tokenType); |
|  | } |
|  |  |
|  | int main(void) |
|  | { |
|  | int newToken, // The current token being processed |
|  | j,k, // Iterators |
|  | ba\_c=0,ba\_o=0,ba\_l, // Number of open and close paranthesis, last line where the open parantesis was used |
|  | bb\_o=0,bb\_c=0,bb\_l, // Number of open and close square braces, last line where the open sqaure brace was used |
|  | bc\_o=0,bc\_c=0,bc\_l, // Number of open and close curly braces, last line where the open curly brace was used |
|  | rep=0; // Flag to denote whether the current token is already in symbol table |
|  |  |
|  | //Taking the input program |
|  | yyin= fopen("test.c","r"); |
|  |  |
|  | //Reading a single token from the program |
|  | newToken = yylex(); |
|  | printf("\n"); |
|  |  |
|  | int mlc=0, // Flag to denote whether current token is part of a multiline comment |
|  | slcline=0, // Line number of the single line comment |
|  | mlcline; // Starting line number of multi line comment |
|  |  |
|  | while(newToken) |
|  | { |
|  | rep = 0; |
|  |  |
|  | if(yylineno==slcline) // If token belongs to a single line comment, ignore all the tokens |
|  | { |
|  | newToken=yylex(); |
|  | continue; |
|  | } |
|  |  |
|  | for(k=0;k<i;k++) // Checking whether token already exists in symbol table |
|  | { |
|  | if(!(strcmp(symbolTable[k].token,yytext))) |
|  | { |
|  | rep = 1; |
|  | break; |
|  | } |
|  | } |
|  |  |
|  | for(k=0;k<c;k++) // Checking whether token already exists in constant table |
|  | { |
|  | if(!(strcmp(constantTable[k].token,yytext))) |
|  | { |
|  | rep = 1; |
|  | break; |
|  | } |
|  | } |
|  |  |
|  | if(ba\_c > ba\_o) |
|  | printf("\n-------------ERROR : UNMATCHED ')' at Line %d------------------\n", yylineno); |
|  |  |
|  | if(bb\_c>bb\_o) |
|  | printf("\n-------------ERROR : UNMATCHED ']' at Line %d------------------\n", yylineno); |
|  |  |
|  | if(bc\_c>bc\_o) |
|  | printf("\n-------------ERROR : UNMATCHED '}' at Line %d------------------\n", yylineno); |
|  |  |
|  | if(rep==0 && newToken!=65 && newToken!=66 && newToken!=67 && mlc==0) |
|  | { |
|  | strcpy(symbolTable[i].token,yytext); |
|  | } |
|  |  |
|  | if(newToken ==1 && mlc==0) |
|  | { |
|  | printf("%s\t\tWhile Loop----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken ==4 && mlc==0) |
|  | { |
|  | printf("%s\t\tMain function----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken ==8 && mlc==0) |
|  | { |
|  | printf("%s\t\tIf statement----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken ==16 && mlc==0) |
|  | { |
|  | printf("%s\t\tElse statement----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken ==17 && mlc==0) |
|  | { |
|  | printf("%s\t\tFor Loop----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken ==18 && mlc==0) |
|  | { |
|  | printf("%s\t\tStruct definition/declaration----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(((newToken>=1 && newToken<=15)) && mlc==0) // Keywords |
|  | { |
|  | printf("%s\t\tKeyword----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken==20 && mlc==0) // Identifiers |
|  | { |
|  | if(rep == 0) |
|  | { symbolInsert(symbolTable, i, yytext, "ID"); |
|  | i++; |
|  | } |
|  | printf("%s\t\tIdentifier----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken==73 && mlc==0) |
|  | { |
|  | if(rep==0) |
|  | { |
|  | symbolInsert(constantTable, c, yytext, "int"); |
|  | c++; |
|  | } |
|  | printf("%s\t\tInteger Constant----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken==74 && mlc==0) |
|  | { |
|  | if(rep==0) |
|  | { |
|  | symbolInsert(constantTable, c, yytext, "float"); |
|  | c++; |
|  | } |
|  | printf("%s\t\tFloating Point Constant----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(((newToken>=22 && newToken<=25)||(newToken>=77 && newToken<=78)) && mlc==0) |
|  | { |
|  | printf("%s\t\tComparision Operator----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken>=26 && newToken<=28 && mlc==0) |
|  | { |
|  | printf("%s\t\tLogical Operator----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken>=29 && newToken<=34 && mlc==0) |
|  | { |
|  | printf("%s\t\tAssignment Operator----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken>=35 && newToken<=41 && mlc==0) |
|  | { |
|  | printf("%s\t\tArithmetic Operator----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken>=42 && newToken<=47 && mlc==0) |
|  | { |
|  | printf("%s\t\tBitwise Operator----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(((newToken>=53 && newToken<=62)||newToken==80) && mlc==0) |
|  | { |
|  | if(newToken==54) |
|  | { |
|  | ba\_o++; |
|  | ba\_l = yylineno; |
|  | } |
|  | if(newToken==55) |
|  | ba\_c++; |
|  | if(newToken==56) |
|  | { |
|  | bb\_o++; |
|  | bb\_l = yylineno; |
|  | } |
|  | if(newToken==57) |
|  | bb\_c++; |
|  | if(newToken==58) |
|  | { |
|  | bc\_o++; |
|  | bc\_l = yylineno; |
|  | } |
|  | if(newToken==59) |
|  | bc\_c++; |
|  | printf("%s\t\tSpecial Character----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken==63 && mlc==0) |
|  | { |
|  | printf("%s\t\tHeader----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken==64 && mlc==0) |
|  | { |
|  | char id[100] = ""; |
|  | for(int t = 0; ; t++) |
|  | { |
|  | if(yytext[t] == '[') |
|  | break; |
|  | id[t] = yytext[t]; |
|  | } |
|  |  |
|  | for(k=0;k<i;k++) // Checking whether token already exists in symbol table |
|  | { |
|  | if(!(strcmp(symbolTable[k].token,id))) |
|  | { |
|  | rep = 1; |
|  | break; |
|  | } |
|  | } |
|  |  |
|  | if(rep == 0) |
|  | { |
|  | symbolInsert(symbolTable, i, id, "ID"); |
|  | i++; |
|  | } |
|  | printf("%s\t\tArray Identfier----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken==65 && mlc==0) |
|  | { |
|  | printf("%s\t\tSingle Line Comment----------Line %d\n",yytext,yylineno); |
|  | slcline=yylineno; |
|  | } |
|  |  |
|  | else if(newToken==66) |
|  | { |
|  | mlc=1; |
|  | printf("%s\t\tMulti Line Comment Start----------Line %d\n",yytext,yylineno); |
|  | mlcline = yylineno; |
|  |  |
|  | } |
|  |  |
|  | else if(newToken==66 && mlc==1) |
|  | { |
|  | printf("%s\t\tNested multi Line Comment Start----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken==67 && mlc==1) |
|  | { |
|  | mlc=0; |
|  | printf("%s\t\tMulti Line Comment End----------Line %d\n",yytext,yylineno); |
|  | mlcline=0; |
|  | } |
|  |  |
|  | else if(newToken==67 && mlc==0) |
|  | printf("\n---------------ERROR : UNMATCHED NESTED END COMMENT-------------\n"); |
|  |  |
|  | else if(newToken==68 && mlc==0) |
|  | { |
|  | printf("%s\t\tPreprocessor Directive----------Line %d\n",yytext,yylineno); |
|  | newToken=yylex(); |
|  | continue; |
|  | } |
|  |  |
|  | else if(newToken>=69 && newToken<=70 && mlc==0) |
|  | { |
|  | printf("%s\t\tPre Defined Function----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken==71 && mlc==0) |
|  | { |
|  | char id[100] = ""; |
|  | for(int t = 0; ; t++) |
|  | { |
|  | if(yytext[t] == '(') |
|  | break; |
|  | id[t] = yytext[t]; |
|  | } |
|  |  |
|  | for(k=0;k<i;k++) // Checking whether token already exists in symbol table |
|  | { |
|  | if(!(strcmp(symbolTable[k].token,id))) |
|  | { |
|  | rep = 1; |
|  | break; |
|  | } |
|  | } |
|  |  |
|  | if(rep == 0) |
|  | { |
|  | symbolInsert(symbolTable, i, id, "ID"); |
|  | i++; |
|  | } |
|  |  |
|  | printf("%s\t\tUser Defined Function----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken==72 && mlc==0) |
|  | { |
|  | if(rep==0) |
|  | { |
|  | symbolInsert(constantTable, c, yytext, "string"); |
|  | c++; |
|  | } |
|  | printf("%s\t\tString literal----------Line %d\n",yytext, yylineno); |
|  | } |
|  |  |
|  | else if(newToken==75 && mlc==0) |
|  | { |
|  | if(rep==0) |
|  | { |
|  | symbolInsert(constantTable, c, yytext, "char"); |
|  | c++; |
|  | } |
|  | printf("%s\t\tCharacter Constant----------Line %d\n",yytext,yylineno); |
|  | } |
|  |  |
|  | else if(newToken==76 && mlc==0) |
|  | { |
|  | printf("\n--------------------ERROR : INCOMPLETE STRING starting at Line %d------------\n",yylineno); |
|  | } |
|  |  |
|  | newToken=yylex(); |
|  | } |
|  |  |
|  | if(mlc==1) |
|  | printf("\n--------------------ERROR : UNMATCHED COMMENT starting at Line %d------------\n",mlcline); |
|  |  |
|  | if(ba\_c<ba\_o) |
|  | printf("\n--------------------ERROR : UNMATCHED '(' at Line %d ------------------------\n",ba\_l); |
|  |  |
|  | if(bb\_c<bb\_o) |
|  | printf("\n--------------------ERROR : UNMATCHED '[' at Line %d ------------------------\n",bb\_l); |
|  |  |
|  | if(bc\_c<bc\_o) |
|  | printf("\n--------------------ERROR ! UNMATCHED '{' at Line %d ------------------------\n",bc\_l); |
|  |  |
|  | printf("\n------------Symbol Table---------------------\n\nSNo\tToken\t\tAttribute\n\n"); |
|  |  |
|  | for(j=0;j<i;j++) |
|  | printf("%d\t%s\t\t< %s >\t\t\n",j+1,symbolTable[j].token,symbolTable[j].type); |
|  |  |
|  | printf("\n------------Constant Table---------------------\n\nSNo\tToken\t\tAttribute\n\n"); |
|  |  |
|  | for(j=0;j<c;j++) |
|  | printf("%d\t%s\t\t< %s >\t\t\n",j+1,constantTable[j].token,constantTable[j].type); |
|  |  |
|  | return 0; |
|  | } |
|  |  |
|  | int yywrap(void) |
|  | { |
|  | return 1; |
|  | } |

**Explanation:**

***Explanation:***

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

• Pre-processor instructions

• Single-line comments

• Multi-line comments

• Errors for unmatched comments

• Errors for nested comments

• Parentheses (all types)

• Operators

• Literals (integer, float, string)

• Errors for unclean integers and floating point numbers

• Errors for incomplete strings

• Keywords

• Identifiers

*Keywords accounted for:* if, else, void, while, do, int, float, break, return and so on.

**Test Cases:**

**Without Errors:**

Serial No Test Case Expected Output Status

1 int a=25; KEYWORD: int IDENTIFIER: a OPERATOR: = INTEGER: 25

SEMICOLON

PASS

**With Errors:**

Serial No Test Case Expected Output Status

1 atmeg = “dfsfds IDENTIFIER: atmeg OPERATOR: = ERR\_INCOMPLETE\_STRING: “dfsfds

2 /\* dead meat ERR\_UNMATCHED\_COMMENT:

/\* dead meat

PASS

PASS

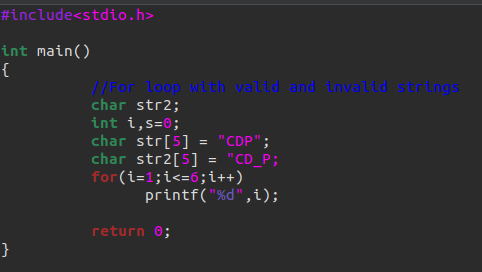


Figure 1: Input for: For loop with valid and invalid strings

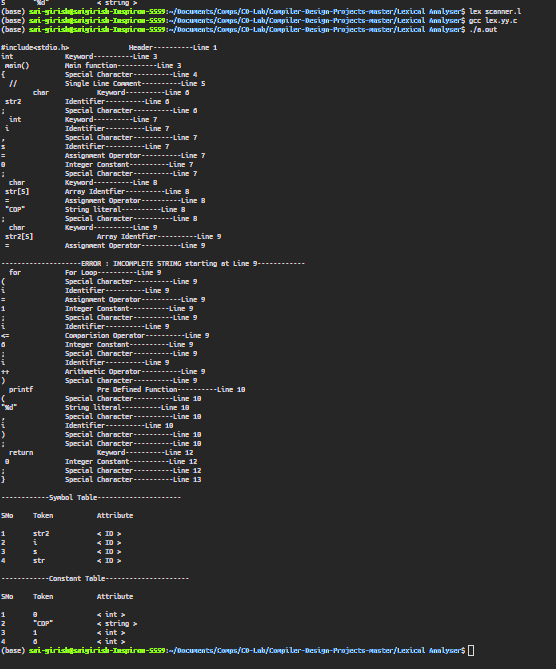


Figure 2: Output for: For loop with valid and invalid strings

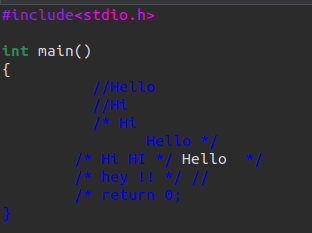


Figure 2: Input for: Various forms of multi-line comment

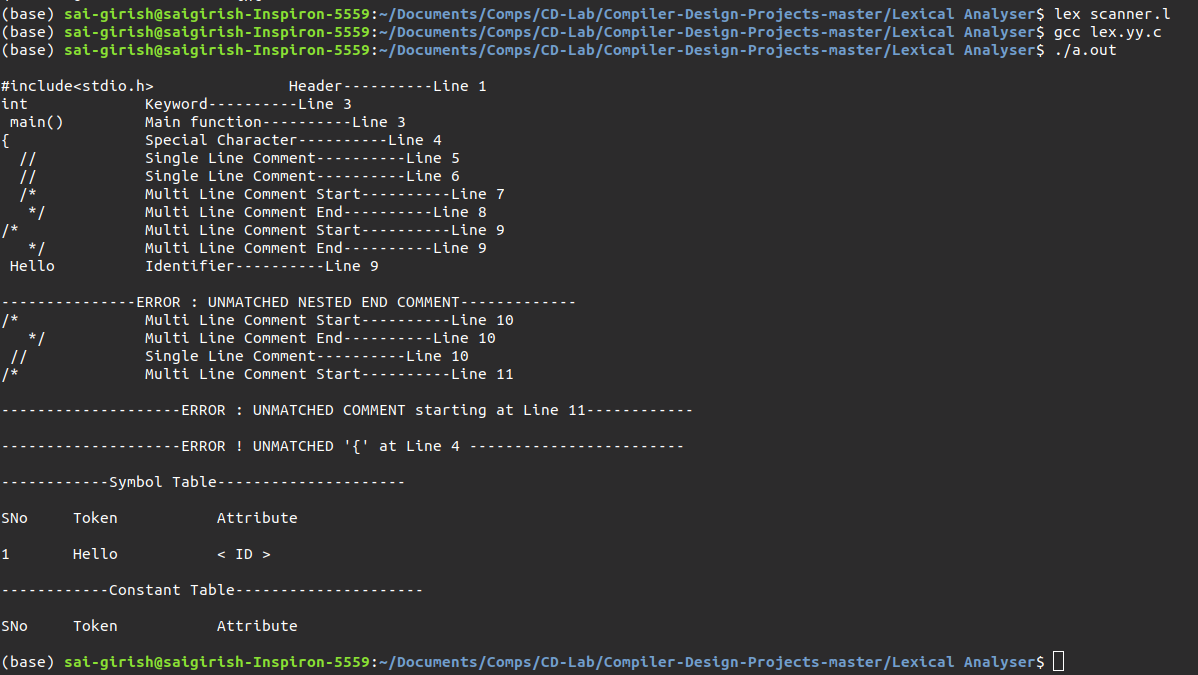


Figure 4: Output for: Various forms of multi-line comment

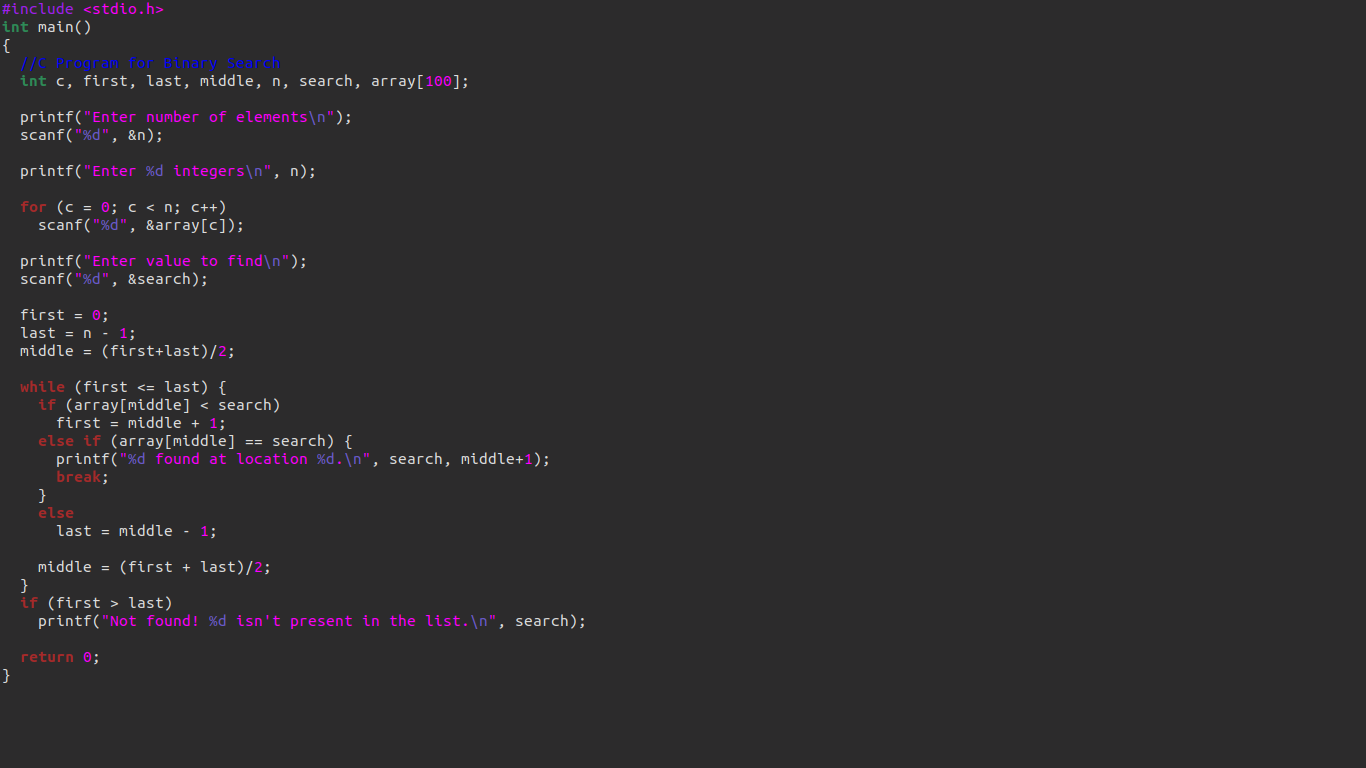
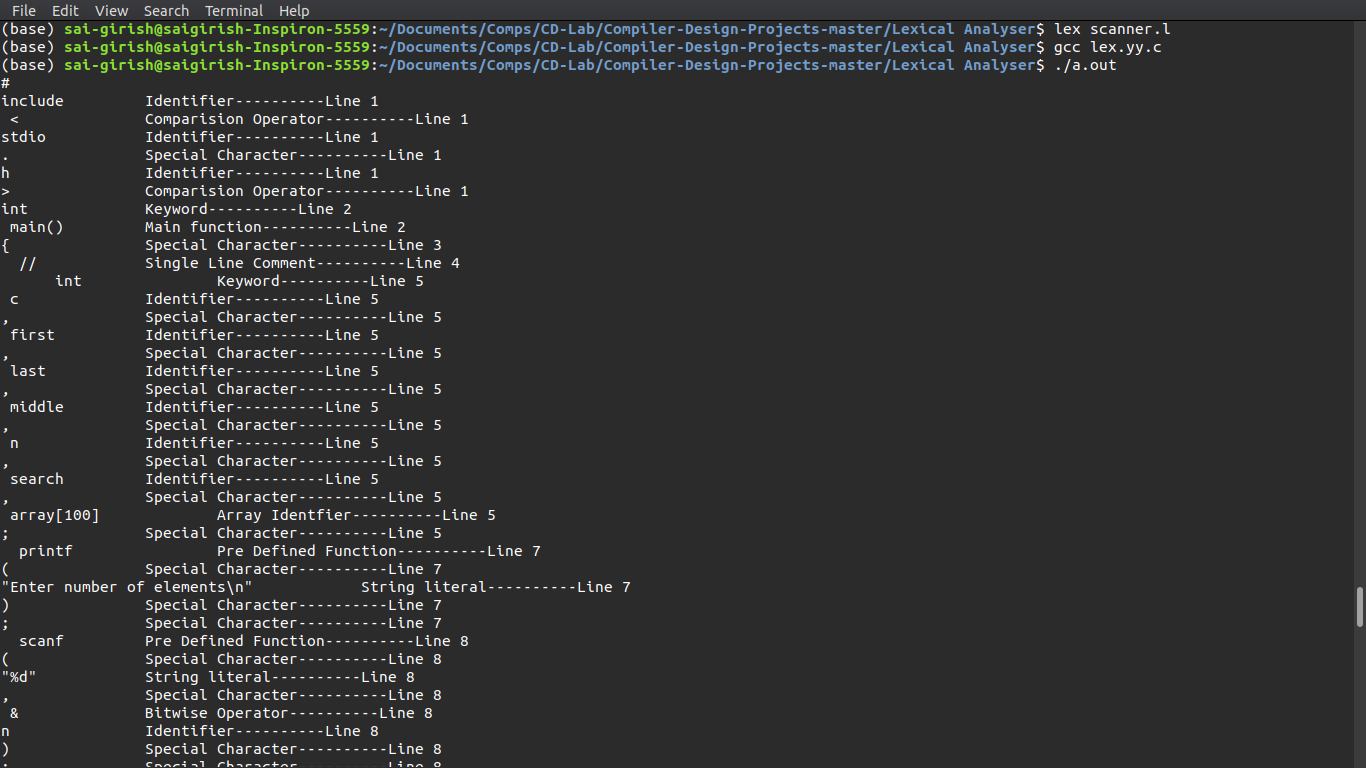
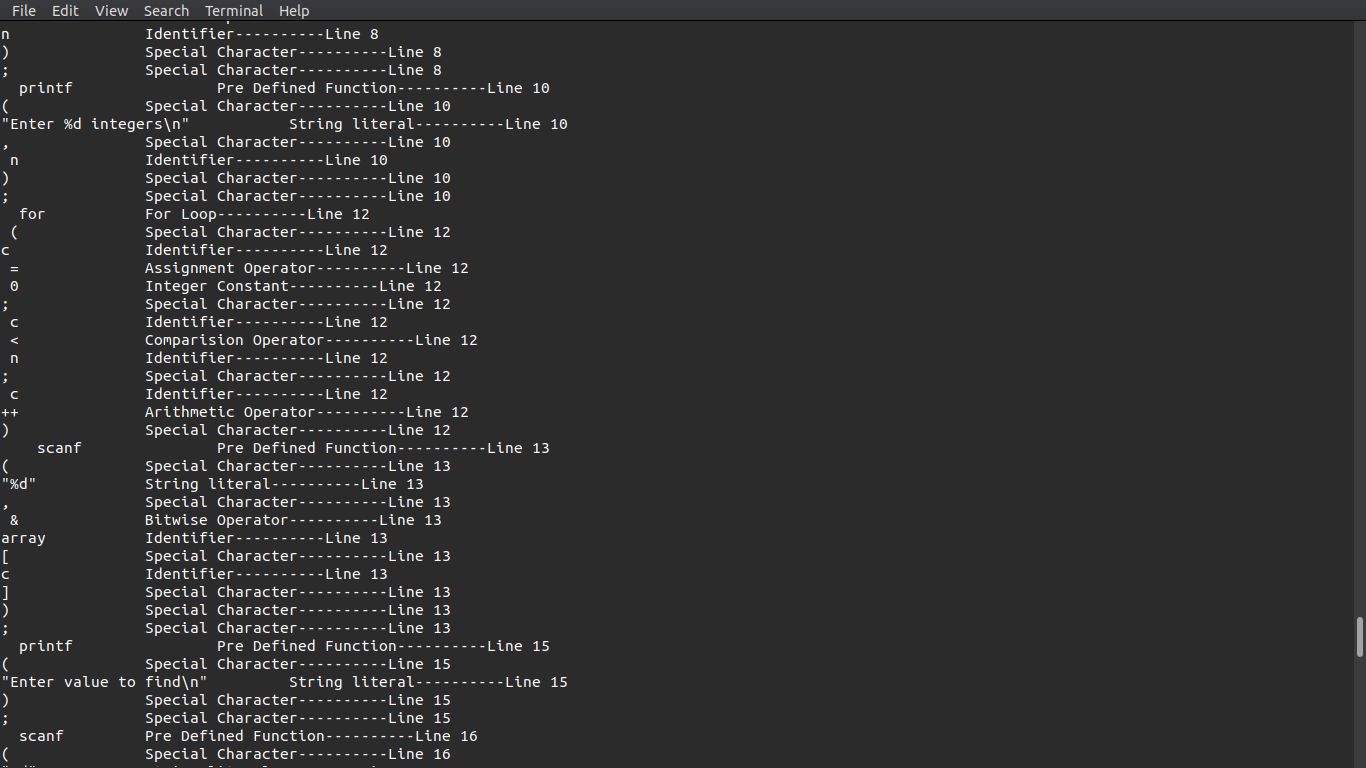
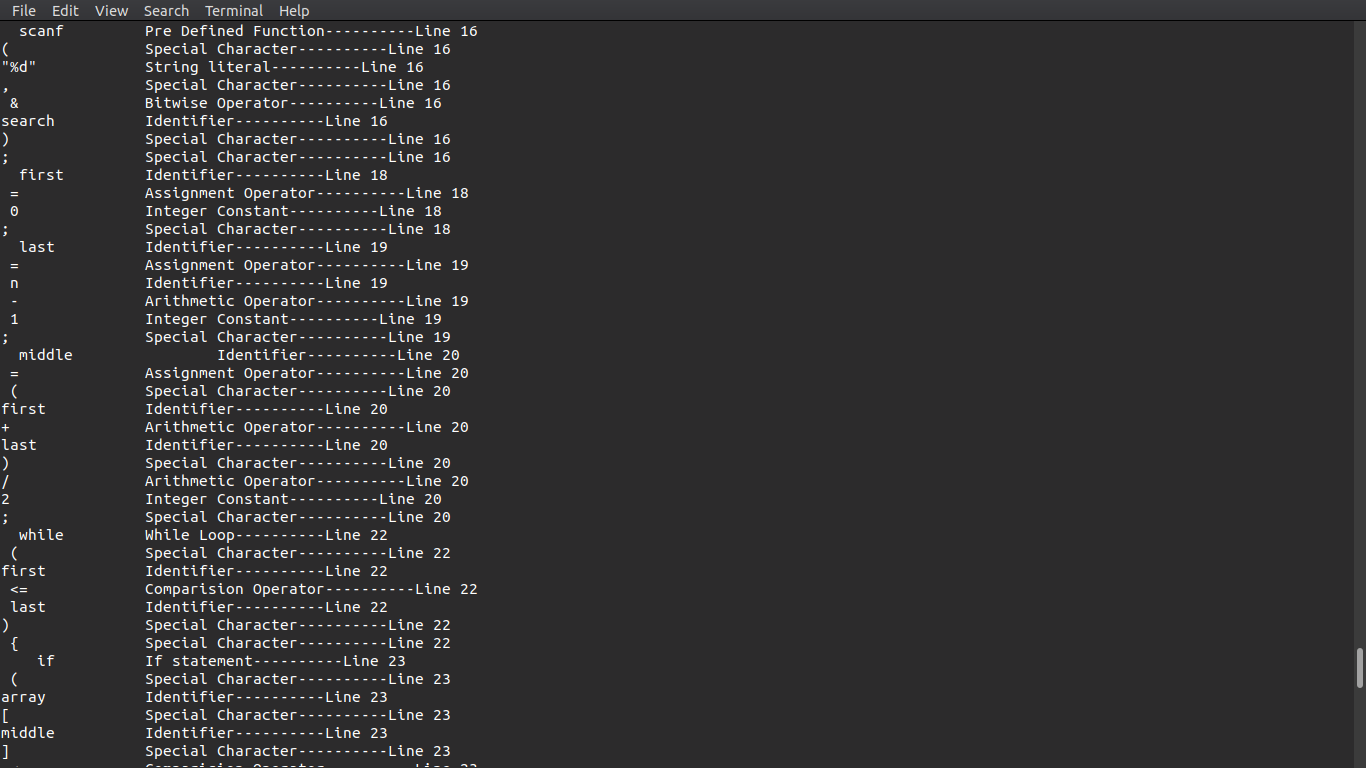
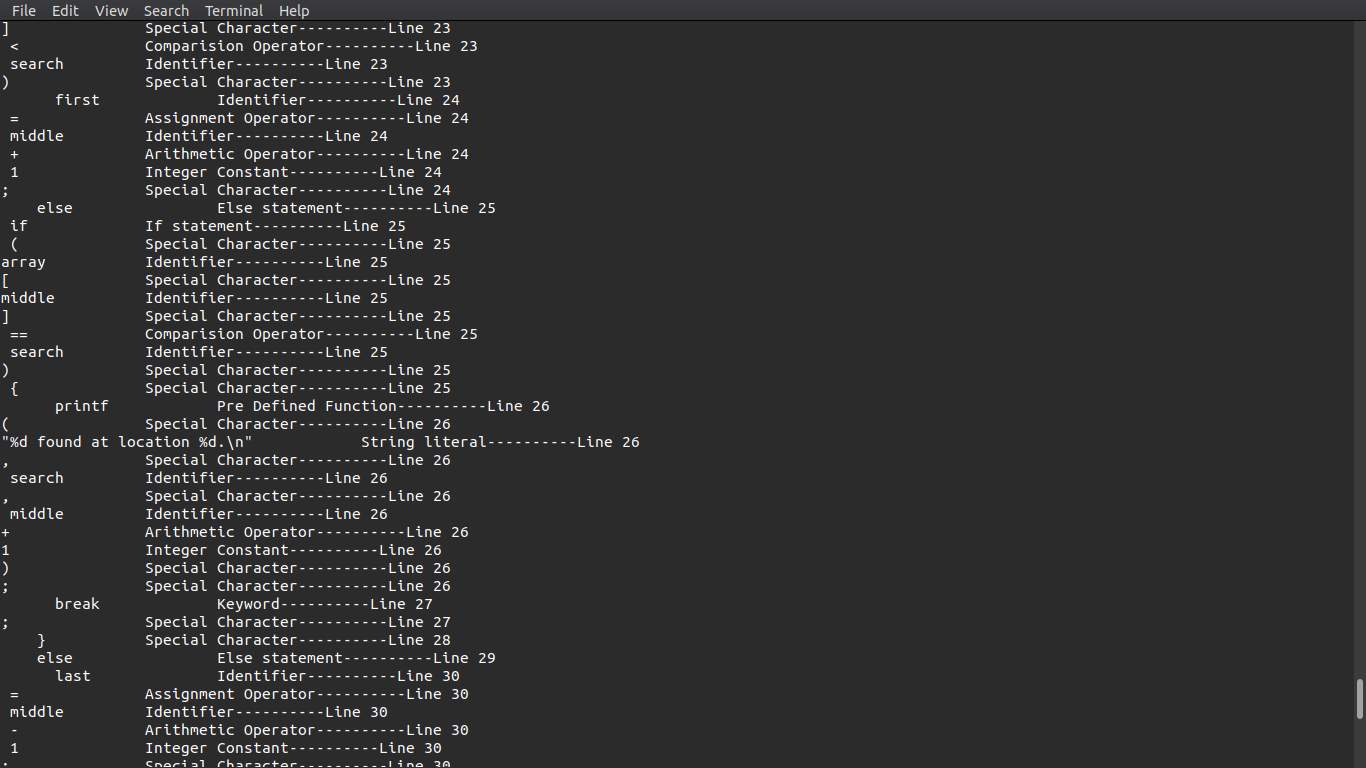


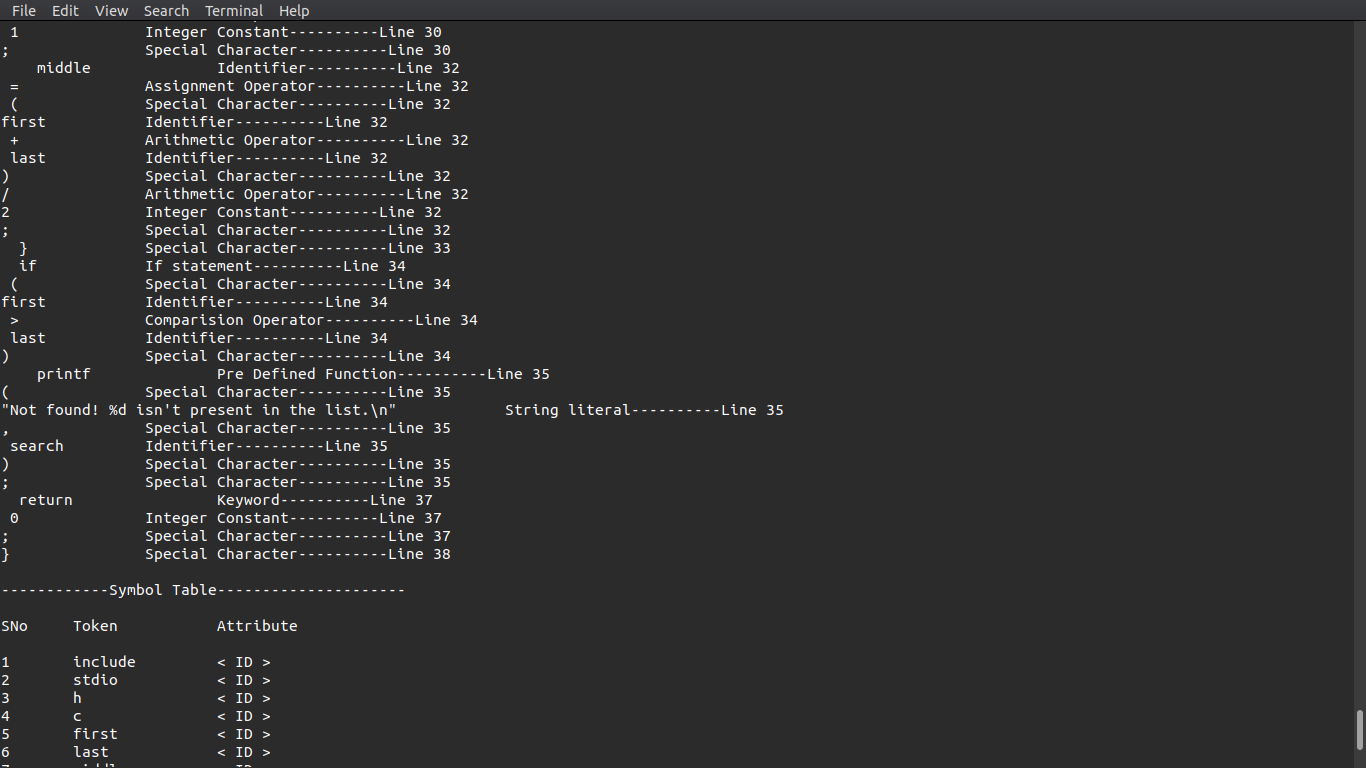
Figure 5: Input for: Sample C program for binary search.











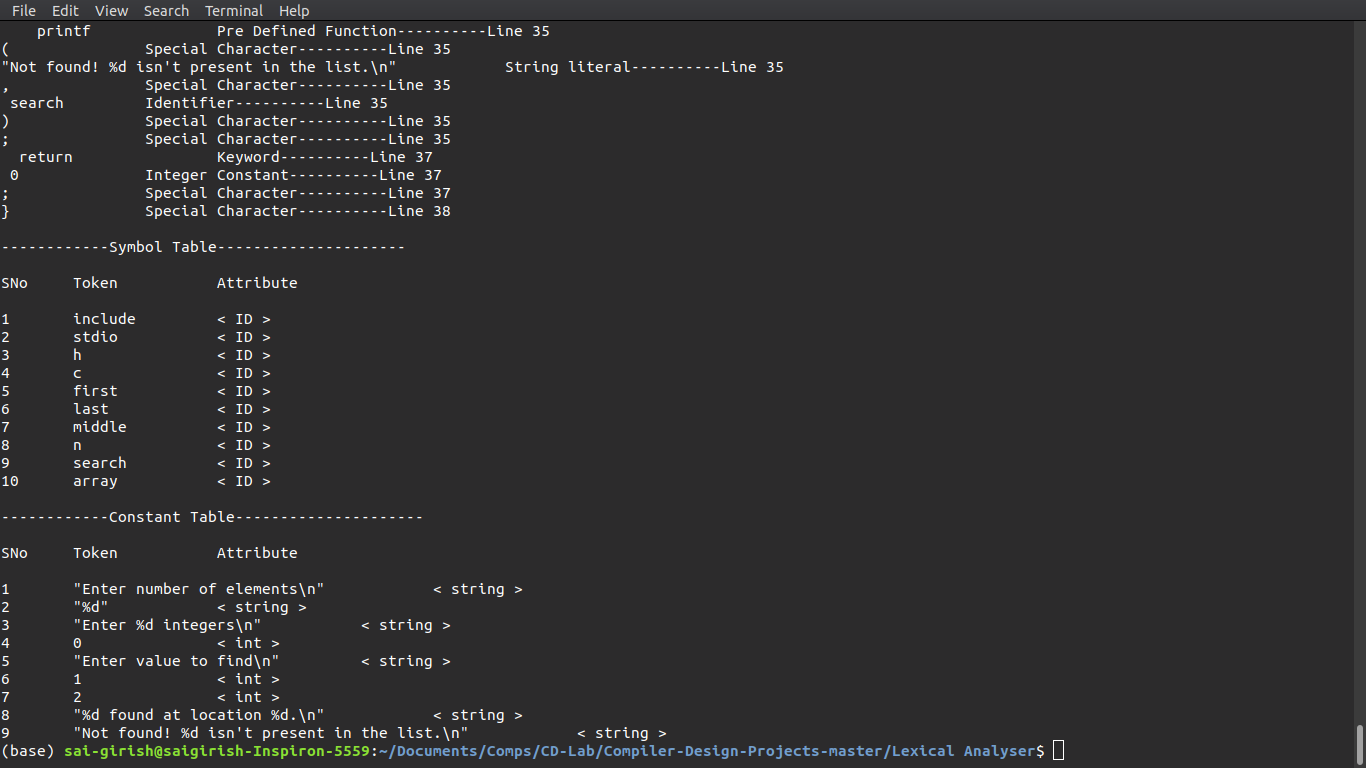


Figure 6: Output for: Sample C program for binary search.

**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.

- **Multiline comments should be supported:** This has been supported by using custom regular algorithm especially robust in cases where tricky characters like \* or / are used within the comments.

- **Literals:** Different regular expressions have been implemented in the code to support all kinds of literals, i.e integers, floats, strings, etc.

- **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 the custom defined regular expressions which help throw errors when comment opening or closing is missing.

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

- We maintain two linked lists of words, one corresponding to identifiers and other to constants.

- Two functions have been implemented, namely add\_to\_table() and check\_present() which is used for adding a new identifier/constant to the linked list and for checking if the identifier/constant is already present in the linked list, respectively.

- Whenever we encounter an identifier/constant, we call the add\_to\_table() function which in turns call check\_present() and adds it to the corresponding liked list.

- In the end, in main() function, after yylex returns, we call print\_symbol\_table(), which

in turn prints the list of identifier and constants in a proper format.

**Results :**

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

2. Symbol Table :

Serial Number ---- Token ---Attribute

3. Constant table

Serial Number ---- Token ----Attribute

**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.

**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