

# Lexical Analyzer for the C Language



National Institute of Technology Karnataka Surathkal

**Date:** 22nd January 2020

**Submitted To:** Dr. P. Santhi Thilagam

<b><u>Group Members:</u></b> 1 - Rahul Kumar	171CO133
2 - Pankaj Kumar Das	171CO128
3 - Rohit Kumar	171CO235

# Lexical Analyzer for the C Language

## **Abstract:**

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language). The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g. assembly language, object code, or machine code) to create an executable program.

## **Phases of Compiler**

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another.

The Phases are as below:

### **❑ Analysis**

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Intermediate Code Generation

### **❑ Synthesis**

1. Code Optimization
2. Code Generation

## **Objectives:**

This project aims to undertake a sequence of experiments to design and implement various phases of a compiler for the C programming language. Following constructs will be handled by the mini-compiler:

1. Data Types: int, char data types with all its sub-types. Syntax : int a=3;
2. Comments: Single line and multiline comments,

## Lexical Analyzer for the C Language

3. Keywords: char, else, for, if, int, long, return, short, signed, struct, unsigned, void, while, main
4. Identification of valid identifiers used in the language,
5. Looping Constructs: It will support nested for and while loops. Syntax: `int i; for(i=0;i<n;i++){ } int x; while(x<10){ ... x++}`
6. Conditional Constructs: if...else-if...else statements,
7. Operators: ADD(+), MULTIPLY(\*), DIVIDE(/), MODULO(%), AND(&), OR(|)
8. Delimiters: SEMICOLON(;), COMMA(,)
9. Structure construct of the language, Syntax: `struct pair{ int a; int b};`
10. Function construct of the language, Syntax: `int func(int x)`
11. Support of nested conditional statement,
12. Support for a 1-Dimensional array. Syntax : `char s[20];`

## Lexical Analyzer for the C Language

<b>Contents :</b>	<b>Page No.</b>
● <b>Introduction</b>	
1. Lexical Analyzer	5
2. Flex Script	6
3. C Program	7
● <b>Design of Programs</b>	
1. Code	7
2. Explanation	13
● <b>Test Cases</b>	
1. Without Errors	16
2. With Errors	24
● <b>Implementation</b>	30
● <b>Results / Future Work</b>	31
● <b>References</b>	31

### List of Figures:

1. **Figure 1:** Output for test cases about data types, keyword, identifier, nested for and while loop, conditional statement, single line comment, multiline comment.
2. **Figure 2:** Output for test case containing operators, structure, function and delimiters.
3. **Figure 3:** Output for test case containing function, print statement.
4. **Figure 4:** Output for test case containing nested conditional statement, array and print statement. Also there is an error in declaring integer variable.
5. **Figure 5:** Output for test case containing string constant and an incomplete string.
6. **Figure 6:** Output for test case containing an error in comments.

### Introduction

#### Lexical Analysis

The Lexical Analyzer is the first phase of the Analysis (front end) stage of a compiler. In layman's terms, the Lexical Analyzer (or Scanner) scans through the input source program character by character, and identifies 'Lexemes' and categorizes them into 'Tokens'. These 'tokens' are represented as a symbol table, and is given as input to the Parser (second phase of the front end of a compiler).

### Tokens

Tokens are essentially just a group of characters which have some meaning or relation when put together.

The Lexical Analyzer detects these tokens with the help of 'Regular Expressions'. While writing the Lexical Analyzer, we have to specify rules for each Token type using Regular Expression. These rules are used to check whether a certain group of characters fall under a given token category or not.

An example, in this case, would be an 'Identifier' token. We specify the rules for an identifier as follows: Any string of characters, that start with an \_ or an alphabet, followed by any number of \_'s, alphabets or numbers. The regular expression for Identifiers is  $\{S\}(\{S\}\{D\})^*$  where S is [a-zA-z\_] and D is [0-9] .

### Lexemes

Lexemes are instances of Tokens. An example would be ' long int ', which is a Lexeme of 'Keyword' Token.

### Symbol Table

A symbol table is generated in the Lexical Analyzer stage, which is basically a table with the columns 'Symbol', 'Type' and 'Token ID'. The symbol is the Lexime itself, the 'Type' is the token category and the 'Token ID' is a unique ID given to a token, which is used in the parser stage. There are no duplicate entries in a symbol table. Each

## Lexical Analyzer for the C Language

symbol is recorded only once, even if there are multiple instances.

A Lexical Analyzer is internally implemented based on the concept of FSM's (Finite State Machines). A DFA (Deterministic Finite State Automata) is internally built for each Token based on the Regular Expression provided. This is used to identify Lexemes and categorize them into Tokens.

### **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.

## Lexical Analyzer for the C Language

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.

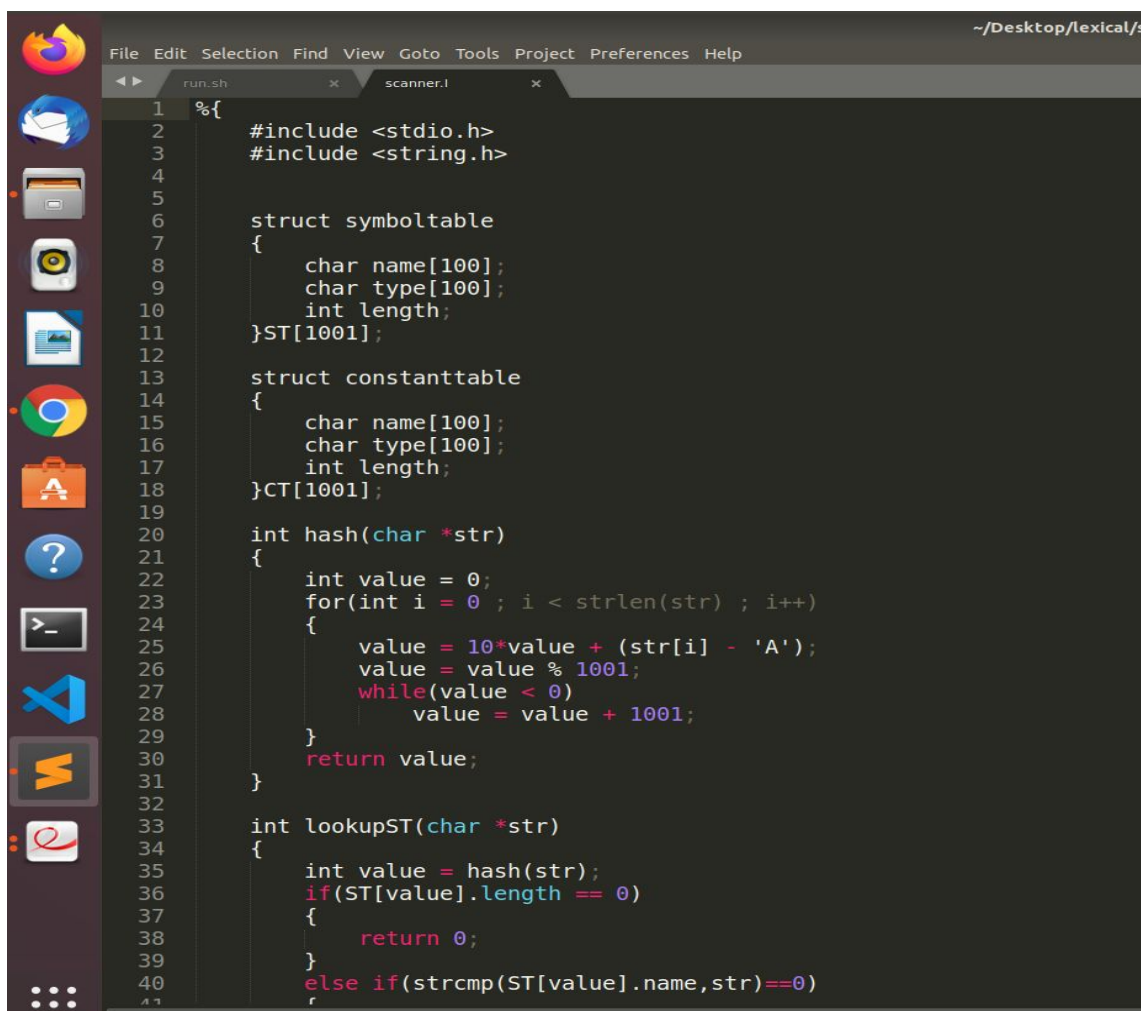
### C Program

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 into 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.

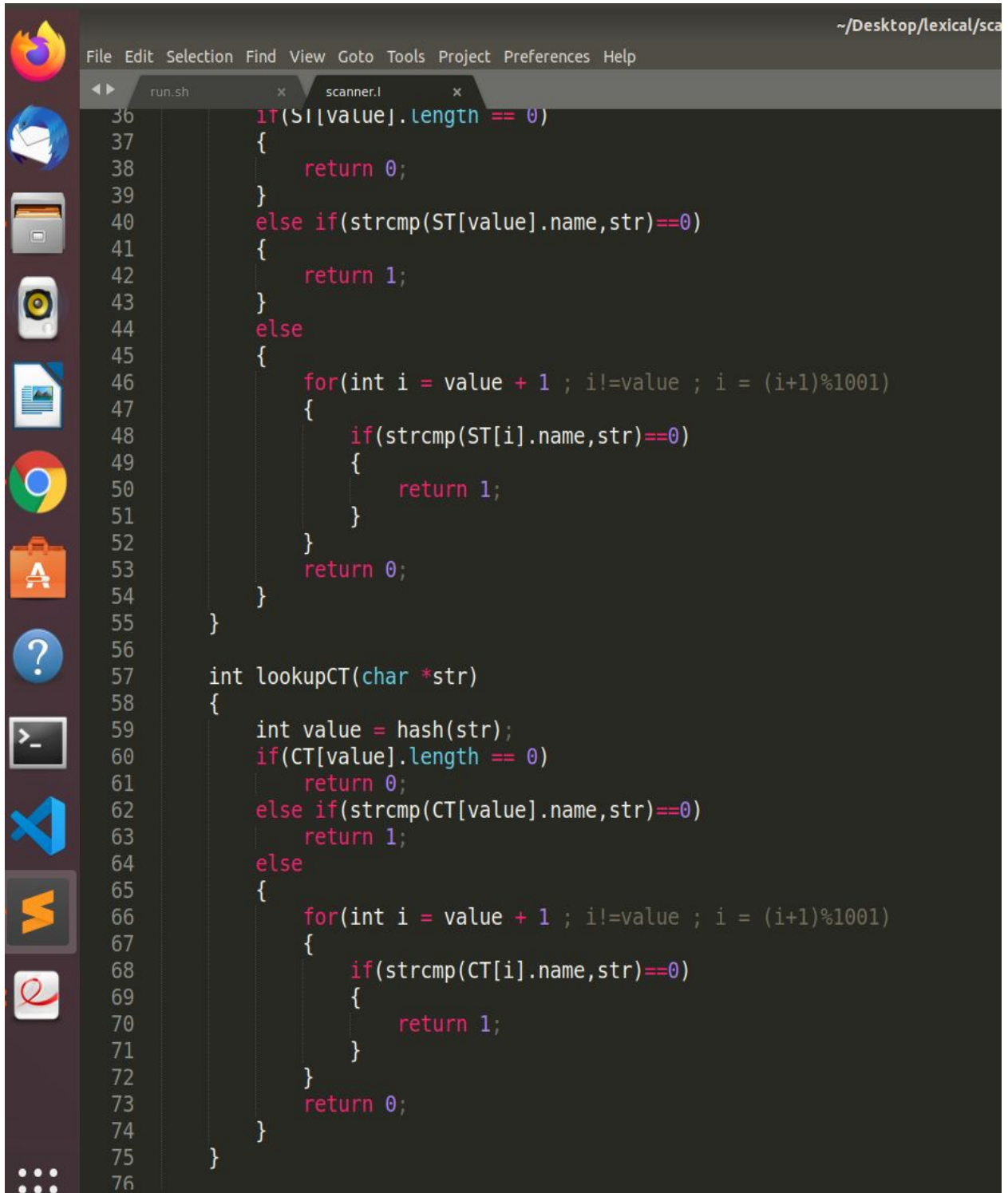
### Design of Programs

#### Code



```
1  %{
2      #include <stdio.h>
3      #include <string.h>
4
5
6      struct symboltable
7      {
8          char name[100];
9          char type[100];
10         int length;
11     }ST[1001];
12
13     struct constanttable
14     {
15         char name[100];
16         char type[100];
17         int length;
18     }CT[1001];
19
20     int hash(char *str)
21     {
22         int value = 0;
23         for(int i = 0 ; i < strlen(str) ; i++)
24         {
25             value = 10*value + (str[i] - 'A');
26             value = value % 1001;
27             while(value < 0)
28                 value = value + 1001;
29         }
30         return value;
31     }
32
33     int lookupST(char *str)
34     {
35         int value = hash(str);
36         if(ST[value].length == 0)
37         {
38             return 0;
39         }
40         else if(strcmp(ST[value].name,str)==0)
41         {
```

## Lexical Analyzer for the C Language

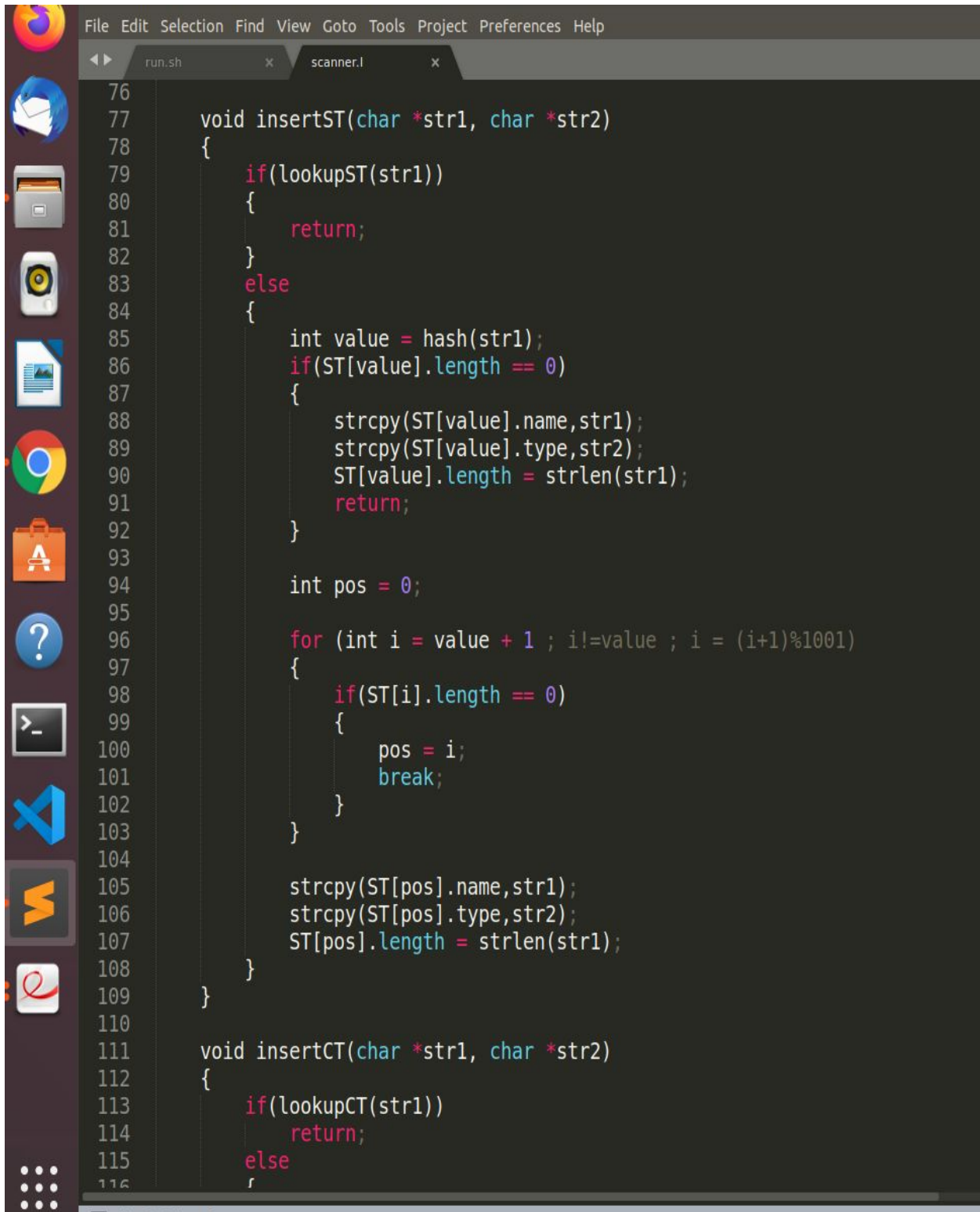


The screenshot shows a Linux desktop with a dark theme. On the left is a vertical dock with icons for Firefox, a mail client, a file manager, a CD/DVD drive, a document viewer, Google Chrome, a terminal, Visual Studio Code, a file manager, and a web browser. The main window is a code editor titled "scanner.l" with a menu bar (File, Edit, Selection, Find, View, Goto, Tools, Project, Preferences, Help) and a toolbar. The code is written in C and implements a lexical analyzer. It includes two functions: `lookupST` (lines 36-54) and `lookupCT` (lines 57-82). Both functions use a hash table to look up tokens. `lookupST` checks for string literals in a table `ST`, and `lookupCT` checks for character tokens in a table `CT`. Both functions use a linear search starting from `value + 1` and wrapping around at 1001. The code is as follows:

```
36     if(ST[value].length == 0)
37     {
38         return 0;
39     }
40     else if(strcmp(ST[value].name, str) == 0)
41     {
42         return 1;
43     }
44     else
45     {
46         for(int i = value + 1 ; i != value ; i = (i+1)%1001)
47         {
48             if(strcmp(ST[i].name, str) == 0)
49             {
50                 return 1;
51             }
52         }
53         return 0;
54     }
55 }
56
57 int lookupCT(char *str)
58 {
59     int value = hash(str);
60     if(CT[value].length == 0)
61         return 0;
62     else if(strcmp(CT[value].name, str) == 0)
63         return 1;
64     else
65     {
66         for(int i = value + 1 ; i != value ; i = (i+1)%1001)
67         {
68             if(strcmp(CT[i].name, str) == 0)
69             {
70                 return 1;
71             }
72         }
73         return 0;
74     }
75 }
76 }
```



## Lexical Analyzer for the C Language



```
76
77 void insertST(char *str1, char *str2)
78 {
79     if(lookupST(str1))
80     {
81         return;
82     }
83     else
84     {
85         int value = hash(str1);
86         if(ST[value].length == 0)
87         {
88             strcpy(ST[value].name, str1);
89             strcpy(ST[value].type, str2);
90             ST[value].length = strlen(str1);
91             return;
92         }
93
94         int pos = 0;
95
96         for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
97         {
98             if(ST[i].length == 0)
99             {
100                 pos = i;
101                 break;
102             }
103         }
104
105         strcpy(ST[pos].name, str1);
106         strcpy(ST[pos].type, str2);
107         ST[pos].length = strlen(str1);
108     }
109 }
110
111 void insertCT(char *str1, char *str2)
112 {
113     if(lookupCT(str1))
114         return;
115     else
116     {
```

## Lexical Analyzer for the C Language



```
114     return;
115 else
116 {
117     int value = hash(str1);
118     if (CT[value].length == 0)
119     {
120         strcpy(CT[value].name, str1);
121         strcpy(CT[value].type, str2);
122         CT[value].length = strlen(str1);
123         return;
124     }
125
126     int pos = 0;
127
128     for (int i = value + 1; i != value; i = (i + 1) % 1001)
129     {
130         if (CT[i].length == 0)
131         {
132             pos = i;
133             break;
134         }
135     }
136
137     strcpy(CT[pos].name, str1);
138     strcpy(CT[pos].type, str2);
139     CT[pos].length = strlen(str1);
140 }
141
142 void printST()
143 {
144     for (int i = 0; i < 1001; i++)
145     {
146         if (ST[i].length == 0)
147         {
148             continue;
149         }
150         printf(" %s\t|\t%s\n", ST[i].name, ST[i].type);
151     }
152 }
153
154
```

Line 1, Column 1

# Lexical Analyzer for the C Language

```

4 void printCT()
5 {
6     for(int i = 0 ; i < 1001 ; i++)
7     {
8         if(CT[i].length == 0)
9             continue;
10
11         printf(" %s\t|\t%s\n",CT[i].name, CT[i].type);
12     }
13 }
14
15 %}
16
17 DE "define"
18 IN "include"
19
20 operator [[<][=][>][=][=][=][!=][=][>][<][\|][\|][&][&][\!][=][\^][\+][=][\ -][=][\*][=]
21
22 %%
23
24 \n {yylineno++;}
25 ([#][ " "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)(>?))/["\n"|\| " "|\t"] {printf(" %s |
26 ([#][ " "]*({DE})[ " "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\| " "|\t"] {printf("%s | Macro
27 \\/(.*) {printf("\n%s | SINGLE LINE COMMENT \n\n", yytext);}
28 \\/^([*]|[\r\n]|(\*+([*]/|[\r\n])))\*+\/ {printf("\n%s | MULTI LINE COMMENT
29 [ \n\t] ;
30
31 ; {printf(" %s | SEMICOLON DELIMITER \n", yytext);}
32 , {printf(" %s | COMMA DELIMITER \n", yytext);}
33 \{ {printf(" %s | OPENING BRACES \n", yytext);}
34 \} {printf(" %s | CLOSING BRACES \n", yytext);}
35 \[ {printf(" %s | OPENING BRACKETS \n", yytext);}
36 \] {printf(" %s | CLOSING BRACKETS \n", yytext);}
37 \[ {printf(" %s | SQUARE OPENING BRACKETS \n", yytext);}
38 \] {printf(" %s | SQUARE CLOSING BRACKETS \n", yytext);}
39 \: {printf(" %s | COLON DELIMITER \n", yytext);}
40 \. {printf(" %s | FSLASH \n", yytext);}
41 \. {printf(" %s | DOT DELIMITER \n", yytext);}
42
43 auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|r
44 \"[\\n]*\\\"/[;|,|\\|) {printf(" %s | STRING CONSTANT \n", yytext); insertCT(yytext,"STRING CONS
45 \\[A-Za-z\\_\\|\\.\\|\\|\\.\\| {printf(" %s | Character CONSTANT \n", yytext); insertCT(yytext,"Cha

```



# Lexical Analyzer for the C Language

```

1 printf(" %s\n", yytext);
2 auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|
3 \("[^"]*"|"/[;|,|,|]) {printf(" %s | STRING CONSTANT \n", yytext); insertCT(yytext,"STRING
4 \'[A-Z|a-z|\'/[;|,|,|]) {printf(" %s | Character CONSTANT \n", yytext); insertCT(yytext,
5 [a-z|A-Z]([a-z|A-Z]|[0-9])* /\{ printf(" %s | ARRAY IDENTIFIER \n", yytext); insertST(yyt
6 {operator}/[a-z]|[0-9]|;|" "[A-Z]|\(|\)|\'|\)|\n|\t {printf(" %s | OPERATOR
7 [1-9][0-9]*|0/[;|,|," "[\]|<|>|=|!|\||&|\+|\-|\*|\/|\%|~|\||\}|:|\n|\t|\^] {printf(" %s |
8 ([0-9]*)\.([0-9]+)/[;|,|," "[\]|<|>|=|!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^] {printf(" %s | Flo
9 [A-Za-z_][A-Za-z_0-9]* /[" "];|,|\(|\)|<|>|=|!|\||&|\+|\-|\*|\/|\%|~|\n|\.\.\\{|\^|\t] {printf("
10
11 (.) {
12     if(yytext[0]=='#')
13     {
14         printf("Error in Pre-Processor directive at line no. %d\n",yylineno);
15     }
16     else if(yytext[0]=='/')
17     {
18         printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);
19     }
20     else if(yytext[0]=='"')
21     {
22         printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);
23     }
24     else
25     {
26         printf("ERROR at line no. %d\n",yylineno);
27     }
28     printf("%s\n", yytext);
29     return 0;
30 }
31
32 %%
33
34 int main(int argc , char **argv){
35
36     printf("
37     printf("
38     printf("
39     printf("=====
40     printf("Lexical Analyzer for the C Language\n");
41     printf("=====

```

## Lexical Analyzer for the C Language

```
%%  
  
int main(int argc , char **argv){  
  
    printf("=====\n");  
    printf("          Lexical Analyzer for the C Language\n");  
    printf("=====\\n\\n");  
  
    int i;  
    for (i=0;i<1001;i++){  
        ST[i].length=0;  
        CT[i].length=0;  
    }  
  
    yyin = fopen(argv[1],"r");  
    yylex();  
  
    printf("\\n\\n\\n=====\n");  
    printf("          SYMBOL TABLE\n");  
    printf("=====\\n\\n");  
    printST();  
  
    printf("\\n\\n\\n=====\n");  
    printf("          CONSTANT TABLE\n");  
    printf("=====\\n");  
    printCT();  
    printf("\\n");  
}  
  
int yywrap(){  
    return 1;  
}
```

### Explanation:

#### Definition Section:

In the definition section of the program, all necessary header files were included. Apart from that structure declaration for both the symbol table and constant table were made. In order to convert a string of the source program into a particular integer value a hash function was written that takes a string as input and converts it into a particular integer value. Standard table operations like look-up and insert were also written. Linear Probing hashing technique was used to implement the symbol table i.e. if there is a collision, then after the point of collision, the table is searched linearly in order to find an empty slot. Functions to print the symbol table and constant table was also written.

### **Rules section:**

In this section rules related to the specification of C language were written in the form of valid regular expressions. E.g. for a valid C identifier the regex written was `[A-Za-z_][A-Za-z_0-9]*` which means that a valid identifier need to start with an alphabet or underscore followed by 0 or more occurrence of alphabets, numbers or underscore. In order to resolve conflicts we used lookahead method of scanner by which a scanner decides whether a expression is valid token or not by looking at its adjacent character. E.g. in order to differentiate between comments and division operator lookahead characters of a valid operator were also given in the regular expression to resolve a conflict. If none of the patterns matched with the input, we said it is a lexical error as it does not match with any valid pattern of the source language. Each character/pattern along with its token class was also printed.

### **C code section:**

In this section both the tables (symbol and constant) were initialised to 0 and `yylex( )` function was called to run the program on the given input file. After that, both the symbol table and constant table were printed in order to show the result.

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 : Preprocessor Directive
- Errors in pre-processor instructions
  - Statements processed : `#include<stdio.h>, #include<stdio.?`
  - Token generated : Error with line number
- Single-line comments
  - Statements processed : `//.....`
  - Token generated : Single Line Comment
- Multi-line comments
  - Statements processed : `/*.....*/, /*.../*...*/`
  - Token generated : Multi Line Comment

## Lexical Analyzer for the C Language

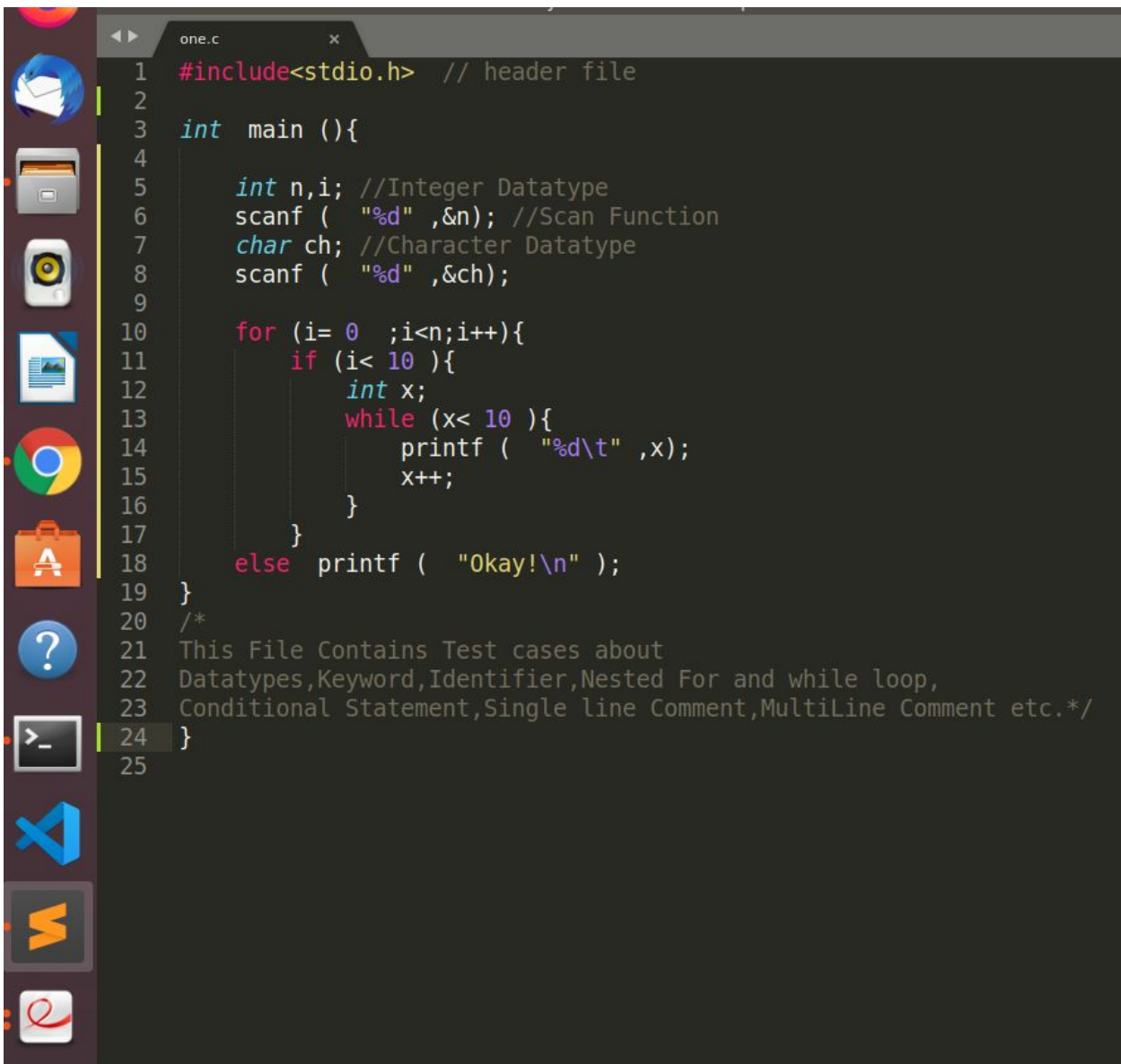
- Errors for unmatched comments
  - Statements processed : /\*.....
  - Token generated : Error with line number
- Errors for nested comments
  - Statements processed : /\*...../\*....\*/....\*/
  - Token generated : Error with line number
- Parentheses (all types)
  - Statements processed : (.), {..}, [..]
  - Token generated : Parenthesis
- Operators
- Literals (integer, float, string)
  - Statements processed : int, float, char
  - Tokens generated : Keywords
- Errors for unclean integers and floating point numbers
  - Statements processed : 123rf
  - Tokens generated : Error
- 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 and so on.
  - Tokens generated : Keyword
- Identifiers
  - Statements processed : a, abc, a\_b, a12b4
  - Tokens generated : Identifier
- Errors for any invalid character used that is not in C character set.
  - Keywords accounted for: auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto,

## Lexical Analyzer for the C Language

if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while, main.

### Test Cases:

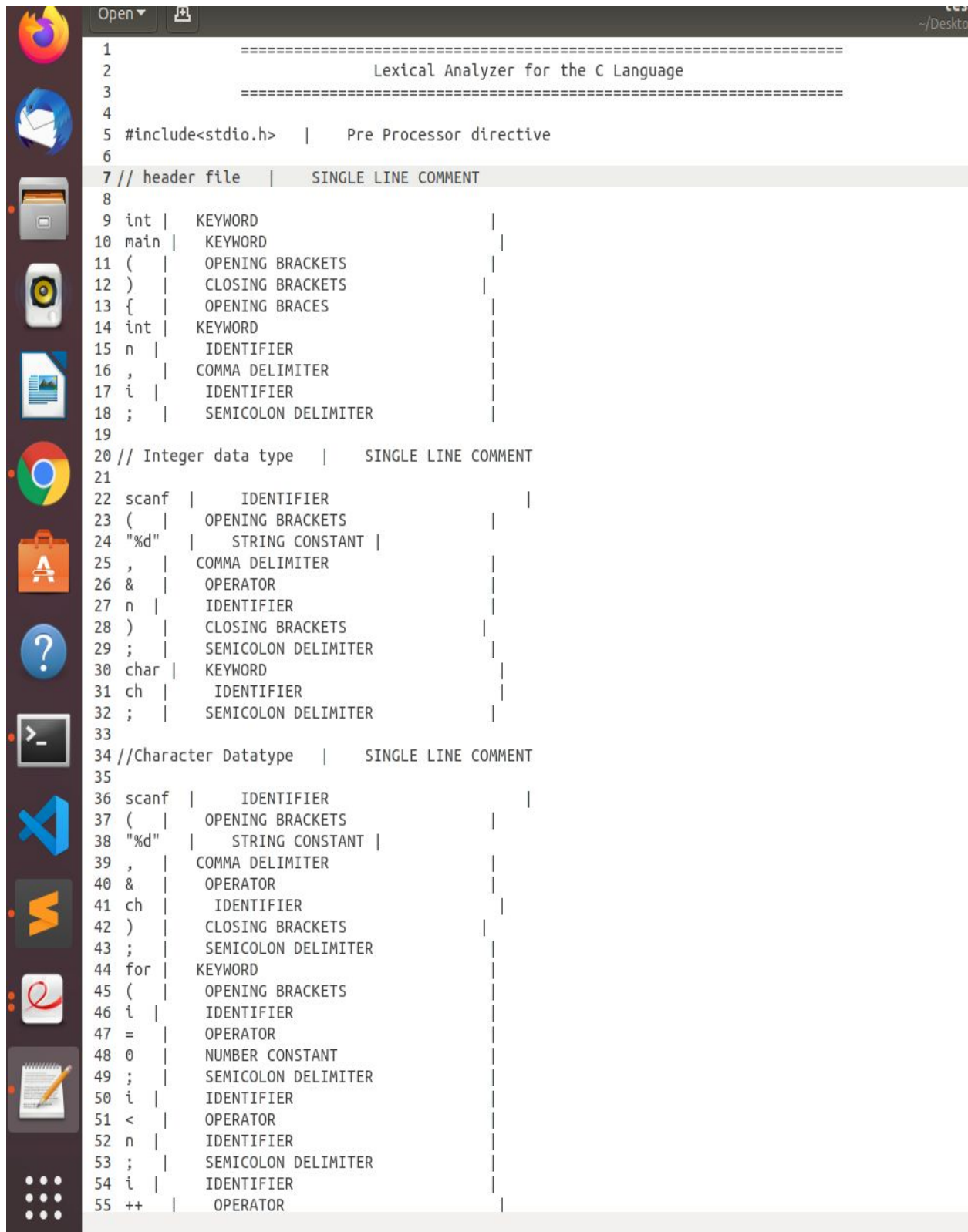
Test 1 Error free code -



```
1  #include<stdio.h> // header file
2
3  int main (){
4
5      int n,i; //Integer Datatype
6      scanf ( "%d" ,&n); //Scan Function
7      char ch; //Character Datatype
8      scanf ( "%d" ,&ch);
9
10     for (i= 0 ;i<n;i++){
11         if (i< 10 ){
12             int x;
13             while (x< 10 ){
14                 printf ( "%d\t" ,x);
15                 x++;
16             }
17         }
18     else printf ( "Okay!\n" );
19 }
20 /*
21 This File Contains Test cases about
22 Datatypes,Keyword,Identifier,Nested For and while loop,
23 Conditional Statement,Single line Comment,MultiLine Comment etc.*/
24 }
25
```

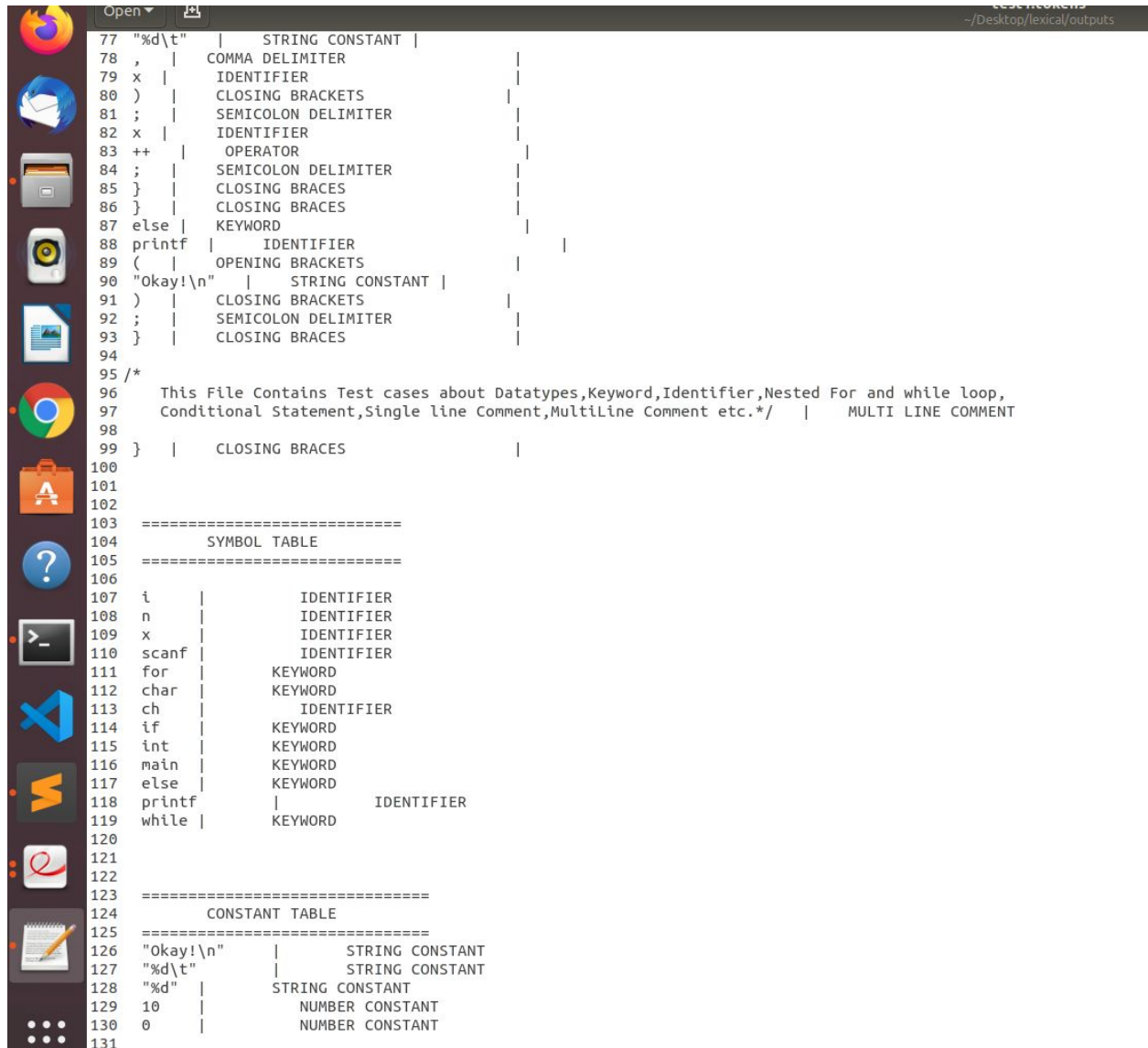


## Lexical Analyzer for the C Language



```
1  =====
2  Lexical Analyzer for the C Language
3  =====
4
5  #include<stdio.h> | Pre Processor directive
6
7  // header file | SINGLE LINE COMMENT
8
9  int | KEYWORD |
10 main | KEYWORD |
11 ( | OPENING BRACKETS |
12 ) | CLOSING BRACKETS |
13 { | OPENING BRACES |
14 int | KEYWORD |
15 n | IDENTIFIER |
16 , | COMMA DELIMITER |
17 i | IDENTIFIER |
18 ; | SEMICOLON DELIMITER |
19
20 // Integer data type | SINGLE LINE COMMENT
21
22 scanf | IDENTIFIER |
23 ( | OPENING BRACKETS |
24 "%d" | STRING CONSTANT |
25 , | COMMA DELIMITER |
26 & | OPERATOR |
27 n | IDENTIFIER |
28 ) | CLOSING BRACKETS |
29 ; | SEMICOLON DELIMITER |
30 char | KEYWORD |
31 ch | IDENTIFIER |
32 ; | SEMICOLON DELIMITER |
33
34 //Character Datatype | SINGLE LINE COMMENT
35
36 scanf | IDENTIFIER |
37 ( | OPENING BRACKETS |
38 "%d" | STRING CONSTANT |
39 , | COMMA DELIMITER |
40 & | OPERATOR |
41 ch | IDENTIFIER |
42 ) | CLOSING BRACKETS |
43 ; | SEMICOLON DELIMITER |
44 for | KEYWORD |
45 ( | OPENING BRACKETS |
46 i | IDENTIFIER |
47 = | OPERATOR |
48 0 | NUMBER CONSTANT |
49 ; | SEMICOLON DELIMITER |
50 i | IDENTIFIER |
51 < | OPERATOR |
52 n | IDENTIFIER |
53 ; | SEMICOLON DELIMITER |
54 i | IDENTIFIER |
55 ++ | OPERATOR |
```

## Lexical Analyzer for the C Language

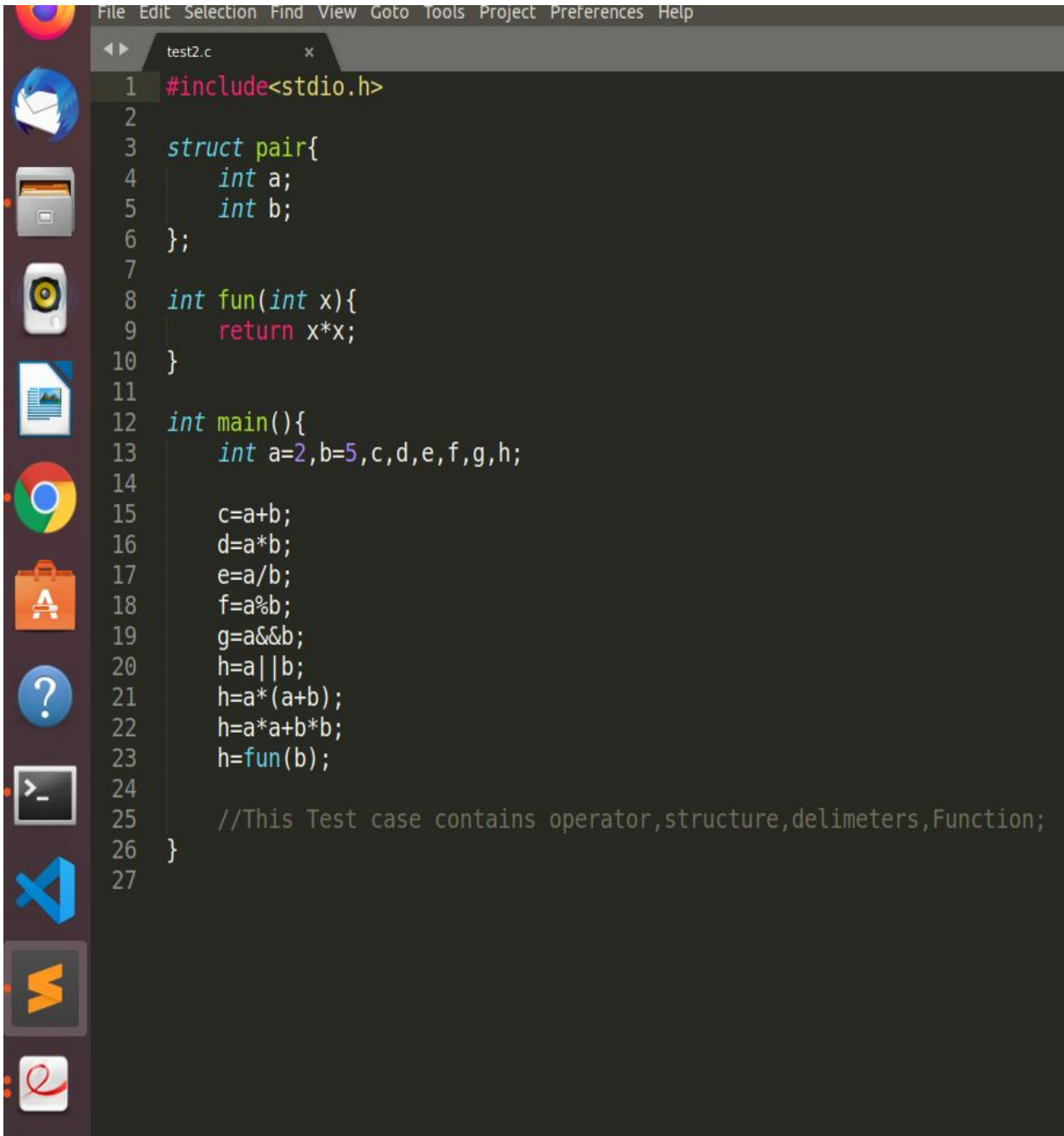


```
77 "%d\t" | STRING CONSTANT |
78 , | COMMA DELIMITER |
79 x | IDENTIFIER |
80 ) | CLOSING BRACKETS |
81 ; | SEMICOLON DELIMITER |
82 x | IDENTIFIER |
83 ++ | OPERATOR |
84 ; | SEMICOLON DELIMITER |
85 } | CLOSING BRACES |
86 } | CLOSING BRACES |
87 else | KEYWORD |
88 printf | IDENTIFIER |
89 ( | OPENING BRACKETS |
90 "Okay!\n" | STRING CONSTANT |
91 ) | CLOSING BRACKETS |
92 ; | SEMICOLON DELIMITER |
93 } | CLOSING BRACES |
94
95 /*
96 This File Contains Test cases about Datatypes,Keyword,Identifier,Nested For and while loop,
97 Conditional Statement,Single line Comment,MultiLine Comment etc.*/ | MULTI LINE COMMENT
98
99 } | CLOSING BRACES |
100
101
102
103 =====
104 SYMBOL TABLE
105 =====
106
107 i | IDENTIFIER
108 n | IDENTIFIER
109 x | IDENTIFIER
110 scanf | IDENTIFIER
111 for | KEYWORD
112 char | KEYWORD
113 ch | IDENTIFIER
114 if | KEYWORD
115 int | KEYWORD
116 main | KEYWORD
117 else | KEYWORD
118 printf | IDENTIFIER
119 while | KEYWORD
120
121
122
123 =====
124 CONSTANT TABLE
125 =====
126 "Okay!\n" | STRING CONSTANT
127 "%d\t" | STRING CONSTANT
128 "%d" | STRING CONSTANT
129 10 | NUMBER CONSTANT
130 0 | NUMBER CONSTANT
131
```

**Status: Pass**

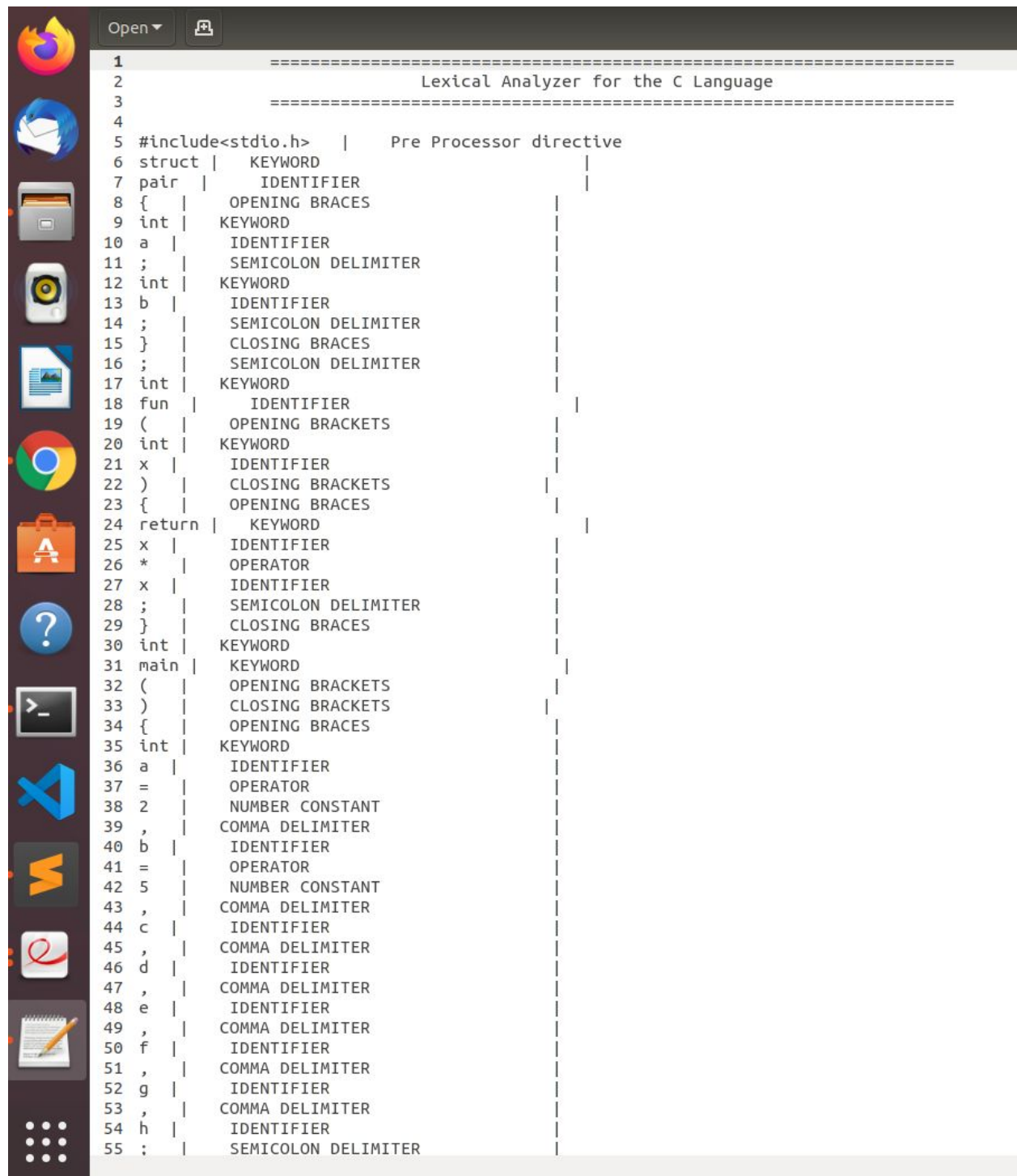
## Lexical Analyzer for the C Language

Test 2 Error free code -



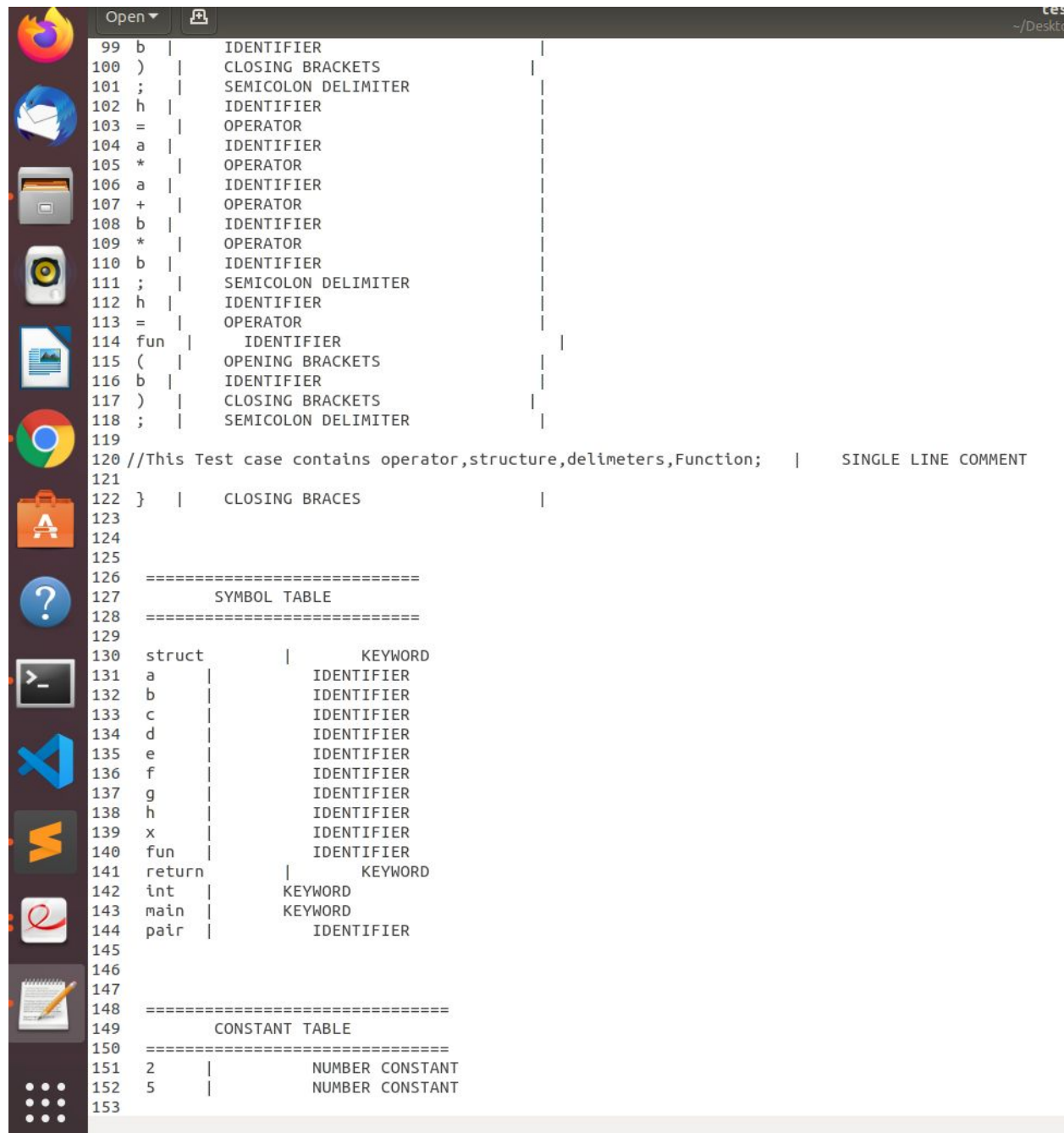
```
File Edit Selection Find View Goto Tools Project Preferences Help
test2.c x
1 #include<stdio.h>
2
3 struct pair{
4     int a;
5     int b;
6 };
7
8 int fun(int x){
9     return x*x;
10 }
11
12 int main(){
13     int a=2,b=5,c,d,e,f,g,h;
14
15     c=a+b;
16     d=a*b;
17     e=a/b;
18     f=a%b;
19     g=a&&b;
20     h=a||b;
21     h=a*(a+b);
22     h=a*a+b*b;
23     h=fun(b);
24
25     //This Test case contains operator,structure,delimeters,Function;
26 }
27
```

## Lexical Analyzer for the C Language



```
1 =====
2                               Lexical Analyzer for the C Language
3                               =====
4
5 #include<stdio.h> | Pre Processor directive
6 struct | KEYWORD |
7 pair | IDENTIFIER |
8 { | OPENING BRACES |
9 int | KEYWORD |
10 a | IDENTIFIER |
11 ; | SEMICOLON DELIMITER |
12 int | KEYWORD |
13 b | IDENTIFIER |
14 ; | SEMICOLON DELIMITER |
15 } | CLOSING BRACES |
16 ; | SEMICOLON DELIMITER |
17 int | KEYWORD |
18 fun | IDENTIFIER |
19 ( | OPENING BRACKETS |
20 int | KEYWORD |
21 x | IDENTIFIER |
22 ) | CLOSING BRACKETS |
23 { | OPENING BRACES |
24 return | KEYWORD |
25 x | IDENTIFIER |
26 * | OPERATOR |
27 x | IDENTIFIER |
28 ; | SEMICOLON DELIMITER |
29 } | CLOSING BRACES |
30 int | KEYWORD |
31 main | KEYWORD |
32 ( | OPENING BRACKETS |
33 ) | CLOSING BRACKETS |
34 { | OPENING BRACES |
35 int | KEYWORD |
36 a | IDENTIFIER |
37 = | OPERATOR |
38 2 | NUMBER CONSTANT |
39 , | COMMA DELIMITER |
40 b | IDENTIFIER |
41 = | OPERATOR |
42 5 | NUMBER CONSTANT |
43 , | COMMA DELIMITER |
44 c | IDENTIFIER |
45 , | COMMA DELIMITER |
46 d | IDENTIFIER |
47 , | COMMA DELIMITER |
48 e | IDENTIFIER |
49 , | COMMA DELIMITER |
50 f | IDENTIFIER |
51 , | COMMA DELIMITER |
52 g | IDENTIFIER |
53 , | COMMA DELIMITER |
54 h | IDENTIFIER |
55 ; | SEMICOLON DELIMITER
```

## Lexical Analyzer for the C Language

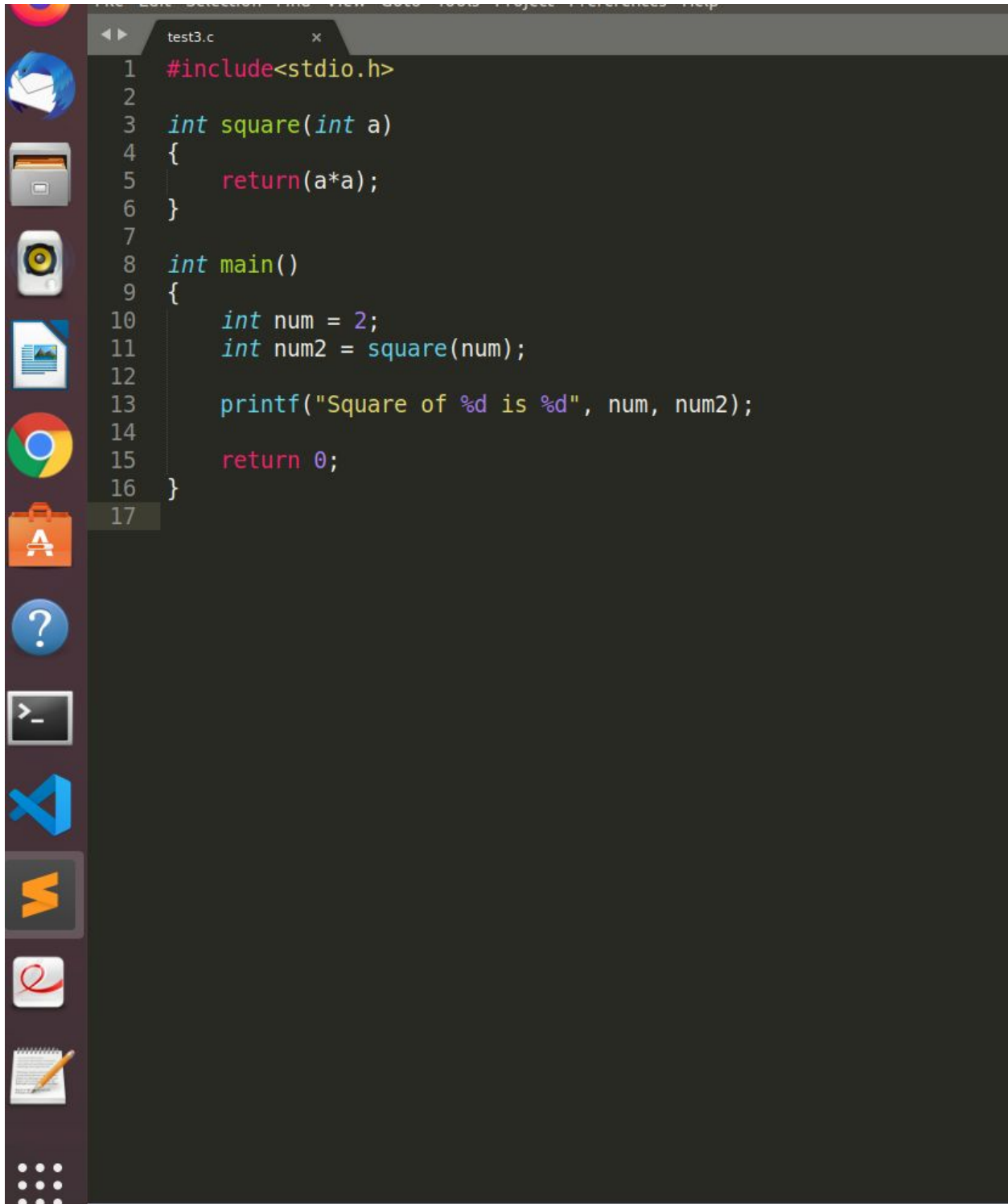


```
199 b | IDENTIFIER |
200 ) | CLOSING BRACKETS |
201 ; | SEMICOLON DELIMITER |
202 h | IDENTIFIER |
203 = | OPERATOR |
204 a | IDENTIFIER |
205 * | OPERATOR |
206 a | IDENTIFIER |
207 + | OPERATOR |
208 b | IDENTIFIER |
209 * | OPERATOR |
210 b | IDENTIFIER |
211 ; | SEMICOLON DELIMITER |
212 h | IDENTIFIER |
213 = | OPERATOR |
214 fun | IDENTIFIER |
215 ( | OPENING BRACKETS |
216 b | IDENTIFIER |
217 ) | CLOSING BRACKETS |
218 ; | SEMICOLON DELIMITER |
219
220 //This Test case contains operator,structure,delimiters,Function; | SINGLE LINE COMMENT
221
222 } | CLOSING BRACES |
223
224
225
226 =====
227 SYMBOL TABLE
228 =====
229
230 struct | KEYWORD
231 a | IDENTIFIER
232 b | IDENTIFIER
233 c | IDENTIFIER
234 d | IDENTIFIER
235 e | IDENTIFIER
236 f | IDENTIFIER
237 g | IDENTIFIER
238 h | IDENTIFIER
239 x | IDENTIFIER
240 fun | IDENTIFIER
241 return | KEYWORD
242 int | KEYWORD
243 main | KEYWORD
244 pair | IDENTIFIER
245
246
247
248 =====
249 CONSTANT TABLE
250 =====
251 2 | NUMBER CONSTANT
252 5 | NUMBER CONSTANT
253
```

**Status: Pass**

## Lexical Analyzer for the C Language

Test Case 3: Error free code -



The screenshot shows a code editor window titled 'test3.c' with a dark background. The code is written in C and is error-free. It includes the standard input/output header, defines a function to calculate the square of an integer, and uses it in the main function to calculate the square of 2 and print the result. The code is as follows:

```
1  #include<stdio.h>
2
3  int square(int a)
4  {
5      return(a*a);
6  }
7
8  int main()
9  {
10     int num = 2;
11     int num2 = square(num);
12
13     printf("Square of %d is %d", num, num2);
14
15     return 0;
16 }
17
```



## Lexical Analyzer for the C Language

```

1  =====
2  Lexical Analyzer for the C Language
3  =====
4
5  #include<stdio.h> | Pre Processor directive
6  int | KEYWORD
7  square | IDENTIFIER
8  ( | OPENING BRACKETS
9  int | KEYWORD
10 a | IDENTIFIER
11 ) | CLOSING BRACKETS
12 { | OPENING BRACES
13 return | KEYWORD
14 ( | OPENING BRACKETS
15 a | IDENTIFIER
16 * | OPERATOR
17 a | IDENTIFIER
18 ) | CLOSING BRACKETS
19 ; | SEMICOLON DELIMITER
20 } | CLOSING BRACES
21 int | KEYWORD
22 main | KEYWORD
23 ( | OPENING BRACKETS
24 ) | CLOSING BRACKETS
25 { | OPENING BRACES
26 int | KEYWORD
27 num | IDENTIFIER
28 = | OPERATOR
29 2 | NUMBER CONSTANT
30 ; | SEMICOLON DELIMITER
31 int | KEYWORD
32 num2 | IDENTIFIER
33 = | OPERATOR
34 square | IDENTIFIER
35 ( | OPENING BRACKETS
36 num | IDENTIFIER
37 ) | CLOSING BRACKETS
38 ; | SEMICOLON DELIMITER
39 printf | IDENTIFIER
40 ( | OPENING BRACKETS
41 "Square of %d is %d" | STRING CONSTANT
42 , | COMMA DELIMITER
43 num | IDENTIFIER
44 , | COMMA DELIMITER
45 num2 | IDENTIFIER
46 ) | CLOSING BRACKETS
47 ; | SEMICOLON DELIMITER
48 return | KEYWORD
49 0 | NUMBER CONSTANT
50 ; | SEMICOLON DELIMITER
51 } | CLOSING BRACES
52
53
54
55 =====

```

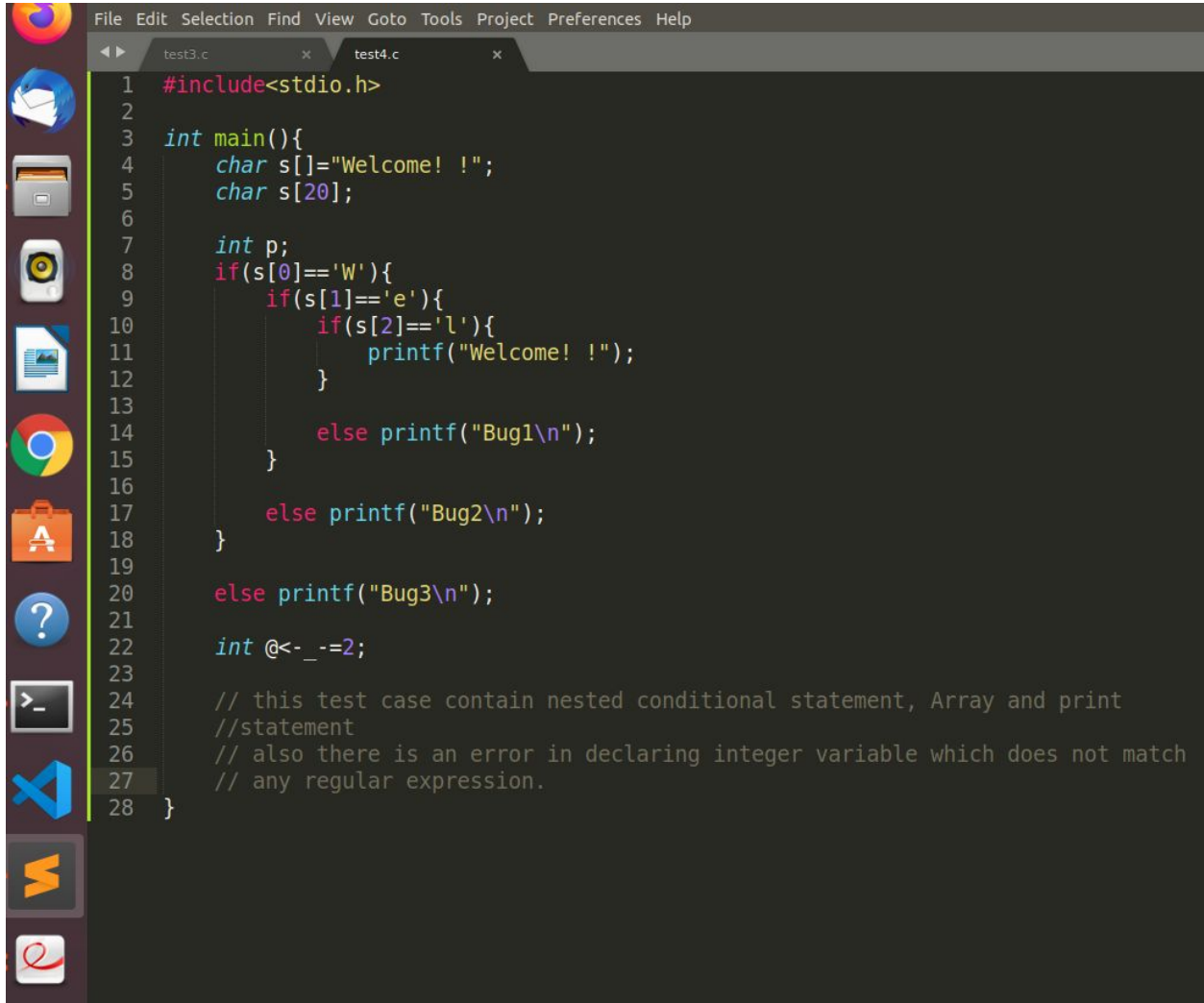
```

Open
22 main | KEYWORD
23 ( | OPENING BRACKETS
24 ) | CLOSING BRACKETS
25 { | OPENING BRACES
26 int | KEYWORD
27 num | IDENTIFIER
28 = | OPERATOR
29 2 | NUMBER CONSTANT
30 ; | SEMICOLON DELIMITER
31 int | KEYWORD
32 num2 | IDENTIFIER
33 = | OPERATOR
34 square | IDENTIFIER
35 ( | OPENING BRACKETS
36 num | IDENTIFIER
37 ) | CLOSING BRACKETS
38 ; | SEMICOLON DELIMITER
39 printf | IDENTIFIER
40 ( | OPENING BRACKETS
41 "Square of %d is %d" | STRING CONSTANT
42 , | COMMA DELIMITER
43 num | IDENTIFIER
44 , | COMMA DELIMITER
45 num2 | IDENTIFIER
46 ) | CLOSING BRACKETS
47 ; | SEMICOLON DELIMITER
48 return | KEYWORD
49 0 | NUMBER CONSTANT
50 ; | SEMICOLON DELIMITER
51 } | CLOSING BRACES
52
53
54
55 =====
56 SYMBOL TABLE
57 =====
58
59 a | IDENTIFIER
60 num | IDENTIFIER
61 square | IDENTIFIER
62 return | KEYWORD
63 int | KEYWORD
64 num2 | IDENTIFIER
65 main | KEYWORD
66 printf | IDENTIFIER
67
68
69
70 =====
71 CONSTANT TABLE
72 =====
73 "Square of %d is %d" | STRING CONSTANT
74 0 | NUMBER CONSTANT
75 2 | NUMBER CONSTANT

```

## Lexical Analyzer for the C Language

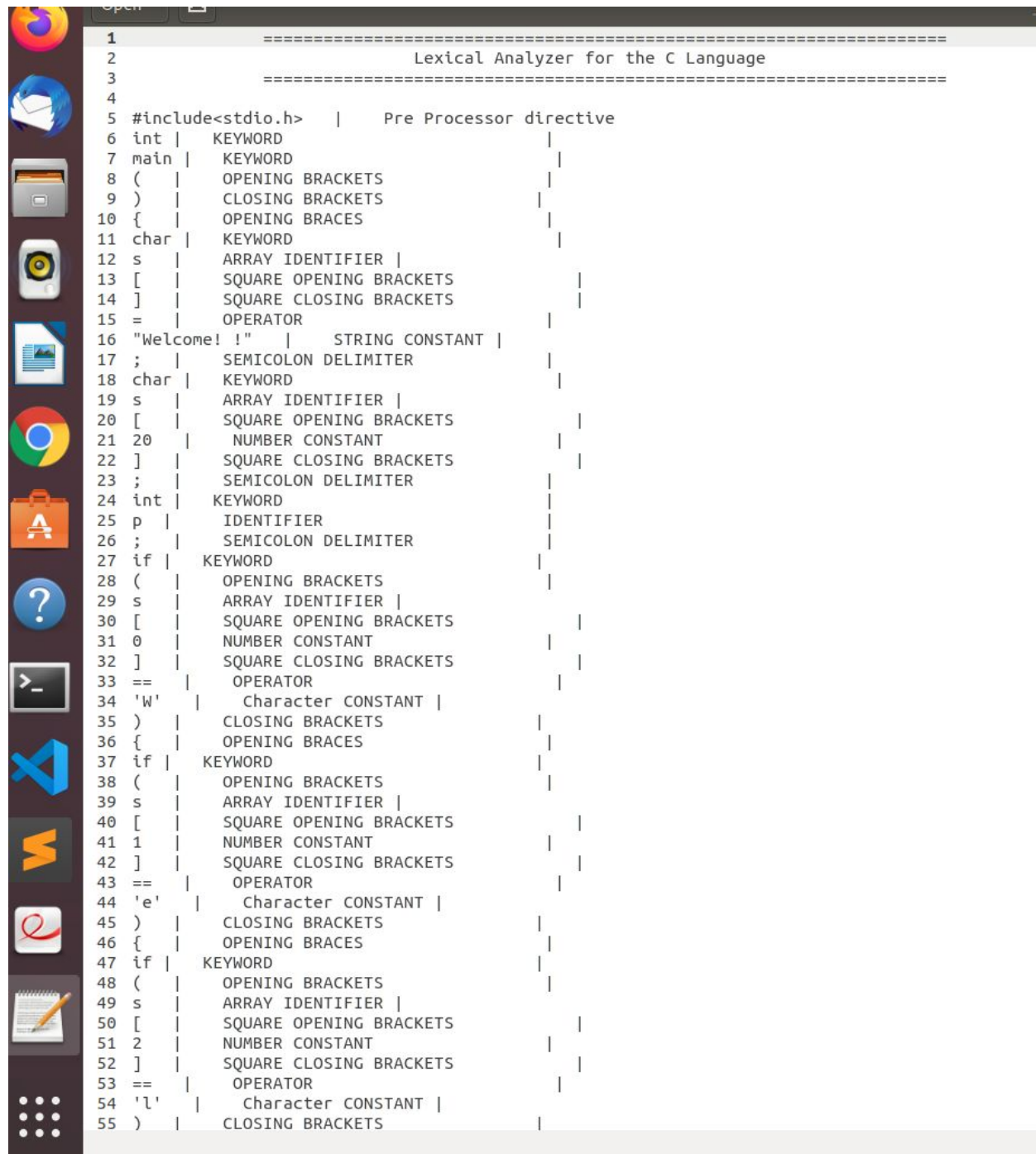
### Test Case 4: Error code -



```
1  #include<stdio.h>
2
3  int main(){
4      char s[]="Welcome! !";
5      char s[20];
6
7      int p;
8      if(s[0]=='W'){
9          if(s[1]=='e'){
10             if(s[2]=='l'){
11                 printf("Welcome! !");
12             }
13             else printf("Bug1\n");
14         }
15         else printf("Bug2\n");
16     }
17     else printf("Bug3\n");
18
19     int @<-_-=2;
20
21     // this test case contain nested conditional statement, Array and print
22     //statement
23     // also there is an error in declaring integer variable which does not match
24     // any regular expression.
25 }
26
27
28 }
```



## Lexical Analyzer for the C Language



The screenshot shows a Linux desktop with a sidebar of application icons including Firefox, LibreOffice, and various system utilities. The terminal window displays the following C code and its lexical analysis results:

```
1 =====
2                               Lexical Analyzer for the C Language
3                               =====
4
5 #include<stdio.h> | Pre Processor directive
6 int | KEYWORD |
7 main | KEYWORD |
8 ( | OPENING BRACKETS |
9 ) | CLOSING BRACKETS |
10 { | OPENING BRACES |
11 char | KEYWORD |
12 s | ARRAY IDENTIFIER |
13 [ | SQUARE OPENING BRACKETS |
14 ] | SQUARE CLOSING BRACKETS |
15 = | OPERATOR |
16 "Welcome! !" | STRING CONSTANT |
17 ; | SEMICOLON DELIMITER |
18 char | KEYWORD |
19 s | ARRAY IDENTIFIER |
20 [ | SQUARE OPENING BRACKETS |
21 20 | NUMBER CONSTANT |
22 ] | SQUARE CLOSING BRACKETS |
23 ; | SEMICOLON DELIMITER |
24 int | KEYWORD |
25 p | IDENTIFIER |
26 ; | SEMICOLON DELIMITER |
27 if | KEYWORD |
28 ( | OPENING BRACKETS |
29 s | ARRAY IDENTIFIER |
30 [ | SQUARE OPENING BRACKETS |
31 0 | NUMBER CONSTANT |
32 ] | SQUARE CLOSING BRACKETS |
33 == | OPERATOR |
34 'w' | Character CONSTANT |
35 ) | CLOSING BRACKETS |
36 { | OPENING BRACES |
37 if | KEYWORD |
38 ( | OPENING BRACKETS |
39 s | ARRAY IDENTIFIER |
40 [ | SQUARE OPENING BRACKETS |
41 1 | NUMBER CONSTANT |
42 ] | SQUARE CLOSING BRACKETS |
43 == | OPERATOR |
44 'e' | Character CONSTANT |
45 ) | CLOSING BRACKETS |
46 { | OPENING BRACES |
47 if | KEYWORD |
48 ( | OPENING BRACKETS |
49 s | ARRAY IDENTIFIER |
50 [ | SQUARE OPENING BRACKETS |
51 2 | NUMBER CONSTANT |
52 ] | SQUARE CLOSING BRACKETS |
53 == | OPERATOR |
54 'l' | Character CONSTANT |
55 ) | CLOSING BRACKETS |
```

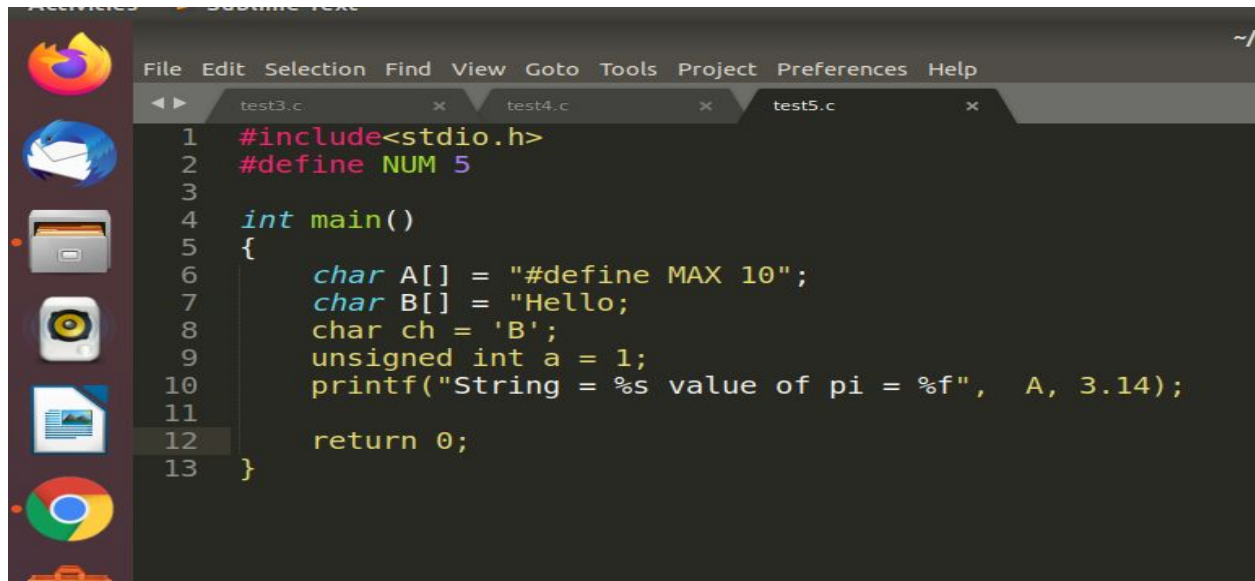
## Lexical Analyzer for the C Language

```
64 printf | IDENTIFIER |
65 ( | OPENING BRACKETS |
66 "Bug1\n" | STRING CONSTANT |
67 ) | CLOSING BRACKETS |
68 ; | SEMICOLON DELIMITER |
69 } | CLOSING BRACES |
70 else | KEYWORD |
71 printf | IDENTIFIER |
72 ( | OPENING BRACKETS |
73 "Bug2\n" | STRING CONSTANT |
74 ) | CLOSING BRACKETS |
75 ; | SEMICOLON DELIMITER |
76 } | CLOSING BRACES |
77 else | KEYWORD |
78 printf | IDENTIFIER |
79 ( | OPENING BRACKETS |
80 "Bug3\n" | STRING CONSTANT |
81 ) | CLOSING BRACKETS |
82 ; | SEMICOLON DELIMITER |
83 int | KEYWORD |
84 ERROR at line no. 43
85 @
86
87
88
89 =====
90 SYMBOL TABLE
91 =====
92
93 p | IDENTIFIER
94 s | IDENTIFIER
95 char | KEYWORD
96 if | KEYWORD
97 int | KEYWORD
98 main | KEYWORD
99 else | KEYWORD
100 printf | IDENTIFIER
101
102
103
104 =====
105 CONSTANT TABLE
106 =====
107 "Welcome! !" | STRING CONSTANT
108 "Bug3\n" | STRING CONSTANT
109 "Bug2\n" | STRING CONSTANT
110 "Bug1\n" | STRING CONSTANT
111 'w' | Character CONSTANT
112 'e' | Character CONSTANT
113 'l' | Character CONSTANT
114 20 | NUMBER CONSTANT
115 0 | NUMBER CONSTANT
116 1 | NUMBER CONSTANT
117 2 | NUMBER CONSTANT
118
```

Status: Pass

## Lexical Analyzer for the C Language

### Test Case 5: Error code



The screenshot shows a code editor with a dark theme. The menu bar includes File, Edit, Selection, Find, View, Goto, Tools, Project, Preferences, and Help. There are three tabs open: test3.c, test4.c, and test5.c. The active tab is test5.c, which contains the following C code:

```
1  #include<stdio.h>
2  #define NUM 5
3
4  int main()
5  {
6      char A[] = "#define MAX 10";
7      char B[] = "Hello;
8      char ch = 'B';
9      unsigned int a = 1;
10     printf("String = %s value of pi = %f", A, 3.14);
11
12     return 0;
13 }
```

```
=====
                          Lexical Analyzer for the C Language
=====

#include<stdio.h>      | Pre Processor directive
#define NUM 5         | Macro
int | KEYWORD
main | KEYWORD
( | OPENING BRACKETS
) | CLOSING BRACKETS
{ | OPENING BRACES
char | KEYWORD
A | ARRAY IDENTIFIER |
[ | SQUARE OPENING BRACKETS
] | SQUARE CLOSING BRACKETS
= | OPERATOR
"#define MAX 10" | STRING CONSTANT
; | SEMICOLON DELIMITER
char | KEYWORD
B | ARRAY IDENTIFIER |
[ | SQUARE OPENING BRACKETS
] | SQUARE CLOSING BRACKETS
= | OPERATOR
ERR_INCOMPLETE_STRING at line no. 13
"

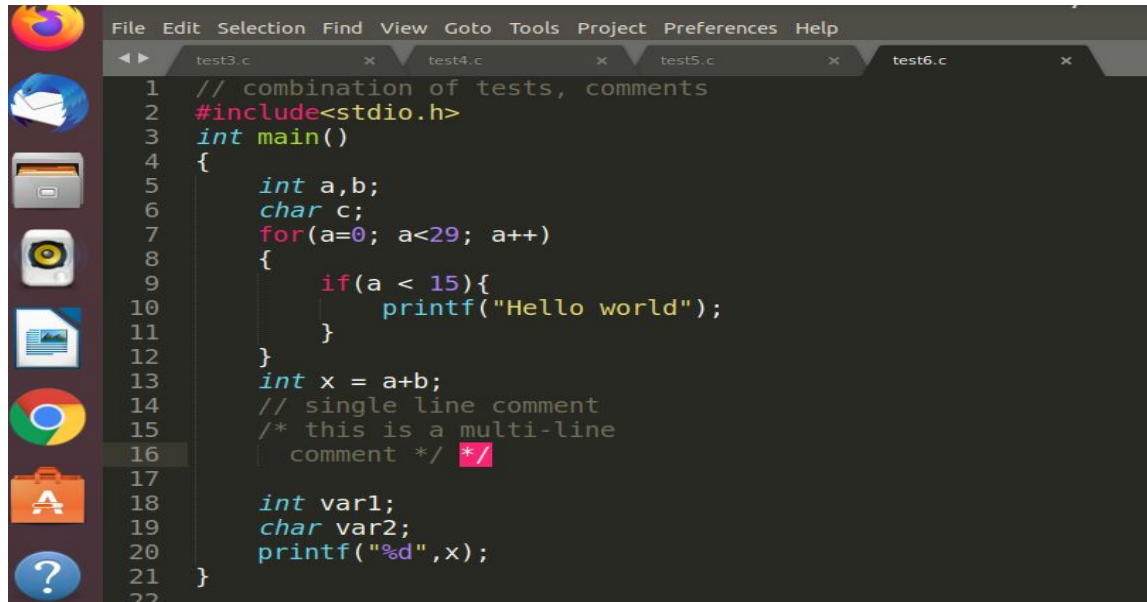
=====
SYMBOL TABLE
=====

A | IDENTIFIER
B | IDENTIFIER
char | KEYWORD
int | KEYWORD
main | KEYWORD

=====
CONSTANT TABLE
=====
"#define MAX 10" | STRING CONSTANT
```

## Lexical Analyzer for the C Language

### Test Case 6: Error code -



```
1 // combination of tests, comments
2 #include<stdio.h>
3 int main()
4 {
5     int a,b;
6     char c;
7     for(a=0; a<29; a++)
8     {
9         if(a < 15){
10             printf("Hello world");
11         }
12     }
13     int x = a+b;
14     // single line comment
15     /* this is a multi-line
16     comment */
17
18     int var1;
19     char var2;
20     printf("%d",x);
21 }
22
```



```
1 =====
2 Lexical Analyzer for the C Language
3 =====
4
5 // combination of tests, comments | SINGLE LINE COMMENT
6
7 #include<stdio.h> | Pre Processor directive
8 int | KEYWORD
9 main | KEYWORD
10 ( | OPENING BRACKETS
11 ) | CLOSING BRACKETS
12 { | OPENING BRACES
13 int | KEYWORD
14 a | IDENTIFIER
15 , | COMMA DELIMITER
16 b | IDENTIFIER
17 ; | SEMICOLON DELIMITER
18 char | KEYWORD
19 c | IDENTIFIER
20 ; | SEMICOLON DELIMITER
21 for | KEYWORD
22 ( | OPENING BRACKETS
23 a | IDENTIFIER
24 = | OPERATOR
25 0 | NUMBER CONSTANT
26 ; | SEMICOLON DELIMITER
27 a | IDENTIFIER
28 < | OPERATOR
29 29 | NUMBER CONSTANT
30 ; | SEMICOLON DELIMITER
31 a | IDENTIFIER
32 ++ | OPERATOR
33 ) | CLOSING BRACKETS
34 { | OPENING BRACES
35 if | KEYWORD
36 ( | OPENING BRACKETS
37 a | IDENTIFIER
38 < | OPERATOR
39 15 | NUMBER CONSTANT
40 ) | CLOSING BRACKETS
41 { | OPENING BRACES
42 printf | IDENTIFIER
43 ( | OPENING BRACKETS
44 "Hello world" | STRING CONSTANT
45 ) | CLOSING BRACKETS
46 ; | SEMICOLON DELIMITER
47 } | CLOSING BRACES
48 } | CLOSING BRACES
49 int | KEYWORD
50 x | IDENTIFIER
51 = | OPERATOR
52 a | IDENTIFIER
53 + | OPERATOR
54 b | IDENTIFIER
```

## Lexical Analyzer for the C Language

```
39 < | OPERATOR |
40 15 | NUMBER CONSTANT |
41 ) | CLOSING BRACKETS |
42 { | OPENING BRACES |
43 printf | IDENTIFIER |
44 ( | OPENING BRACKETS |
45 "Hello world" | STRING CONSTANT |
46 ) | CLOSING BRACKETS |
47 ; | SEMICOLON DELIMITER |
48 } | CLOSING BRACES |
49 } | CLOSING BRACES |
50 int | KEYWORD |
51 x | IDENTIFIER |
52 = | OPERATOR |
53 a | IDENTIFIER |
54 + | OPERATOR |
55 b | IDENTIFIER |
56 ; | SEMICOLON DELIMITER |
57
58 // single line comment | SINGLE LINE COMMENT
59
60
61 /* this is a multi-line
62    comment */ | MULTI LINE COMMENT |
63
64 ERROR at line no. 30
65 *
66
67
68
69 =====
70 SYMBOL TABLE
71 =====
72
73 a | IDENTIFIER
74 b | IDENTIFIER
75 c | IDENTIFIER
76 x | IDENTIFIER
77 for | KEYWORD
78 char | KEYWORD
79 if | KEYWORD
80 int | KEYWORD
81 main | KEYWORD
82 printf | IDENTIFIER
83
84
85
86 =====
87 CONSTANT TABLE
88 =====
89 "Hello world" | STRING CONSTANT
90 15 | NUMBER CONSTANT
91 29 | NUMBER CONSTANT
92 0 | NUMBER CONSTANT
93
```

**Status: 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.
- **Multiline comments should be supported:** To implement it a proper regular expression was written along with that lookahead character set for operators were thought so to resolve conflict with the division operator.
- **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 Unmatched Comments:** This has been handled by adding lookahead characters to operator regular expression. If there is an unmatched comment then it does not match with any of the patterns in the rule. Hence it goes to default state which in turn throws an error.
- **Error Handling for unclean integer constant:** This has been handled by adding appropriate lookahead characters for integer constant. E.g. `int a = 786rt`, is rejected as the integer constant should never follow an alphabet.

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 structures one for symbol table and other for constant table one corresponding to identifiers and other to constants.
- Four functions have been implemented `lookupST( )`, `lookupCT( )`, these functions return true if the identifier and constant respectively are already present in the table. `InsertST( )`, `InsertCT( )` help to insert identifier/constant in the appropriate table
- Whenever we encounter an identifier/constant, we call the `insertST()` or `insertCT()` function which in turns call `lookupST( )` or `lookupCT( )` and adds it to the corresponding structure.
- In the end, in `main( )` function, after `yylex` returns, we call `printST( )` and `printCT( )`, which in turn prints the list of identifier and constants in a proper format



## Lexical Analyzer for the C Language

### Results:

1. Token --- Token Class
2. Symbol Table:  
Token - Attribute
3. Constant Table  
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:

- Compilers Principles, Techniques and Tool by Alfred V.Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- <http://dinosaur.compilertools.net/lex/index.html>
- <http://www.csd.uwo.ca/~moreno/CS447/Lectures/Lexical.html/node11.html>