

Lexical Analyzer for the C Language



National Institute of Technology Karnataka Surathkal

Date: 17th January 2019

Submitted To: Dr. P. Santhi Thilagam

Group Members:

1. Gurupungav Narayanan	16C0114
2. Nihal Haneef	16C0128
3. Rishika Narayanan	16C0241

ABSTRACT

INTRODUCTION

A compiler is a program that can read a program in one language, the source language - and translate it to an equivalent program in another language, the target language. This project focuses on creating a “Mini C Compiler”, basically a compiler for a subset of the C language. The compiler construction has been divided into four phases called “Phases”.

PHASE 1

The first project phase requires us to construct a scanner or in other words a lexical analyzer which is the first part of a compiler. It takes the source program as an input and gives out a *stream of tokens as an output*.

The lexical analyzer maintains a data structure called the symbol table, but for this phase we will just be identifying the token and its type and displaying the necessary information. When the lexical analyzer discovers a lexeme constituting an identifier, it enters that lexeme into the symbol table. The scanner will also strip out *white spaces and comments* from the input and provide necessary *error messages* as and where it is necessary.

The scanner will take care of the following syntactical structures:

Data types - int (short and long) and char initialization and declaration.

```
int a = 5, b = 6;
```

Functions - Functions with one parameter of int or char data types. The return type being taken care of will also be int and char.

```
int perimeter (int a) {...}
```

Conditional - If else blocks and nested if else statements.

```
If(a==b) {...} else {...};
```

Array - Single dimensional array initialization.

```
int students [10];
```

Loops - While loop construct will be taken care of.

```
while(a==b) {...}
```

The scanner will analyze the following *tokens*:

Keywords - The lexer will identify int, long, short, long long, signed, unsigned, for, while, break, continue, if, else, return, for and const.

Identifiers - All identifiers following the C language semantics will be analyzed. The regex for an identifier is `[_a-zA-Z][_a-zA-Z0-9]*`

Strings - The lexer will identify strings in any C program. It can also handle double quotes that are escaped using a \ inside a string.

Constants - The lexer will identify integer, hexadecimal and octal constants.

Operators - The lexer will identify unary, arithmetic, relational, logical, assignment and conditional operators.

Special Symbols - The lexer will identify brackets, braces, parentheses, comma, semicolon, assignment operator and preprocessor directive.

TOOLS AND LANGUAGE

1. The language for which the compiler is being made for is a subset of the C language.
2. Lex/Flex will be used.
3. Yacc/Bison will be used.
4. Visual Studio Code will be the primary text editor
5. GitHub will be utilized for collaboration and version control.

Contents

Introduction	2
Phase 1	2
Tools and Language	3
Lexical Analyzer	6
Flex Script	6
C Program	7
Code	8
Explanation	9
Without Errors	10
With Errors	12
Other Case - 1	14
Other Case – 2	16
Overview	20
Results	21
Future Work	21
References	21

Table of Figures

Figure 1: Code for Scanner8

Figure 2: Code for Scanner9

Figure 3: Test Case Figure10

Figure 4: Output for Error free11

Figure 5: Test case with Error12

Figure 6: Out for Erroneous Test Case.....13

Figure 7: Test case.....14

Figure 8: Output Test case.....15

Figure 9: Test case.....16

Figure 10: Output 117

Figure 11: Output 218

Figure 12: Output 319

INTRODUCTION

LEXICAL ANALYZER

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

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

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

```
1  /* ----- */
2  /* Phase 1:  Creating the Scanner/Lexical analyser */
3  /* ----- */
4  /* Team Members:  Gurupungav Narayanan  16C0114 */
5  /*                Nihal Haneef         16C0128 */
6  /*                Rishika Narayanan     16C0241 */
7  /* ----- */
8
9
10 %{
11 #include <stdio.h>
12 #include "token.h"
13 int start;
14 %}
15
16 LETTER      [a-zA-Z]
17 DIGIT       [0-9]
18 WS          [ \t\r\f\v]+
19 IDENTIFIER_C (_{LETTER})(_{LETTER}|_{DIGIT})_)*
20 HEX         [0-9a-f]
21
22
23 /* States */
24 %x COMMENT
25 %x PREPROCESSOR
26
27 %%
28
29 \n                                {++yylineno;}
30
31 ^"#include"                      {BEGIN(PREPROCESSOR);}
32 <PREPROCESSOR> ""<[^<>\n]+>" | ""\["^"\n]+\n" {printf("%-40s - HEADER FILE - %d\n",yytext,HEADERFILE);BEGIN(INITIAL);}
33 <PREPROCESSOR>.                    {printf("ERROR IN %d: ILLEGAL HEADER FORMAT\n",yylineno);BEGIN(INITIAL);}
34
35 "int"                             {printf("%-40s - KEYWORD - %d\n",yytext,INT);}
36 "char"                           {printf("%-40s - KEYWORD - %d\n",yytext,CHAR);}
37 "float"                          {printf("%-40s - KEYWORD - %d\n",yytext,FLOAT);}
38 "signed"                         {printf("%-40s - KEYWORD - %d\n",yytext,SIGNED);}
39 "unsigned"                       {printf("%-40s - KEYWORD - %d\n",yytext,UNSIGNED);}
40 "short"                          {printf("%-40s - KEYWORD - %d\n",yytext,SHORT);}
41 "long"                           {printf("%-40s - KEYWORD - %d\n",yytext,LONG);}
42 "long long"                      {printf("%-40s - KEYWORD - %d\n",yytext,LLONG);}
43 "const"                          {printf("%-40s - KEYWORD - %d\n",yytext,CONST);}
44 "return"                         {printf("%-40s - KEYWORD - %d\n",yytext,RETURN);}
45 "if"                             {printf("%-40s - KEYWORD - %d\n",yytext,IF);}
46 "else"                           {printf("%-40s - KEYWORD - %d\n",yytext,ELSE);}
47 "for"                             {printf("%-40s - KEYWORD - %d\n",yytext,FOR);}
48 "while"                          {printf("%-40s - KEYWORD - %d\n",yytext,WHILE);}
49 "break"                          {printf("%-40s - KEYWORD - %d\n",yytext,BREAK);}
50 "continue"                       {printf("%-40s - KEYWORD - %d\n",yytext,CONTINUE);}
51 "void"                           {printf("%-40s - KEYWORD - %d\n",yytext,VOID);}
52 "enum"                           {printf("%-40s - KEYWORD - %d\n",yytext,ENUM);}
53
54 {IDENTIFIER_C}                   {
55 |                               {
56 |                               if (strlen(yytext)>32)
57 |                               printf("ERROR IN %d : INVALID TOKEN IDENTIFIER TOO LONG\n",yylineno);
58 |                               else
59 |                               printf("%-40s - IDENTIFIER - %d\n",yytext,IDENTIFIER);
60 |                               }
61
62 "/*".*                           {};
63
64 "/*"                             {start = yylineno; BEGIN(COMMENT);}
65 <COMMENT>.[\s]                   {};
66 <COMMENT>\n                      {yylineno++;}
67 <COMMENT>"*/"                    {BEGIN(INITIAL);}
68 <COMMENT>"/**"                   {printf("ERROR IN %d : INVALID TOKEN NESTED COMMENTS INVALID\n",yylineno);}
69 <COMMENT><<EOF>>                  {printf("ERROR IN %d: UNTERMINATED COMMENT\n",start);yyterminate();}
70
71 {\s}                             {};
72
73 [+&-]?[0][xX]{HEX}+              {printf("%-40s - CONSTANT - %d\n",yytext,HEXCONSTANT);}
74 [+&-]?[0][dD]{DIGIT}+\. {DIGIT}+ {printf("%-40s - CONSTANT - %d\n",yytext,FLOATCONSTANT);}
75 [+&-]?[0][dD]{DIGIT}+            {printf("%-40s - CONSTANT - %d\n",yytext,DECIMALCONSTANT);}
76
77 \["^"\n^"\"]*\n                  {printf("%-40s - ERROR: ILLEGAL TOKEN\n",yytext);}
78 \["^"\n]*\n                      {printf("%-40s - CONSTANT - %d\n",yytext,STR);}
79 \'({LETTER}|_{DIGIT})({LETTER}|_{DIGIT})+\' {printf("%-40s - ERROR: ILLEGAL TOKEN\n",yytext);}
80 \'\'                             {printf("%-40s - CONSTANT - %d\n",yytext,CHARCONSTANT);}
81
82 "["                               {printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,LEFTBRACKET);}
83 "]"                               {printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,RIGHTBRACKET);}
```

Figure 1: Code for Scanner


```

83 "(" {printf("%40s - SPECIAL SYMBOL - %d\n", yytext, LEFTBRACE);}
84 ")" {printf("%40s - SPECIAL SYMBOL - %d\n", yytext, RIGHTBRACE);}
85 "(" {printf("%40s - SPECIAL SYMBOL - %d\n", yytext, LEFTPARAMETHESES);}
86 ")" {printf("%40s - SPECIAL SYMBOL - %d\n", yytext, RIGHTPARAMETHESES);}
87 "," {printf("%40s - SPECIAL SYMBOL - %d\n", yytext, COMMA);}
88 ";" {printf("%40s - SPECIAL SYMBOL - %d\n", yytext, SEMICOLON);}
89
90 "++" {printf("%40s - OPERATOR - %d\n", yytext, INCREMENT);}
91 "--" {printf("%40s - OPERATOR - %d\n", yytext, DECREMENT);}
92 "==" {printf("%40s - OPERATOR - %d\n", yytext, EQUAL);}
93 "!=" {printf("%40s - OPERATOR - %d\n", yytext, NOTEQUAL);}
94 ">=" {printf("%40s - OPERATOR - %d\n", yytext, GREATEREQUAL);}
95 "<=" {printf("%40s - OPERATOR - %d\n", yytext, LESSEREQUAL);}
96 "||" {printf("%40s - OPERATOR - %d\n", yytext, LOGICALAND);}
97 "||" {printf("%40s - OPERATOR - %d\n", yytext, LOGICALOR);}
98 "+=" {printf("%40s - OPERATOR - %d\n", yytext, ADDASSIGN);}
99 "-=" {printf("%40s - OPERATOR - %d\n", yytext, SUBTRACTASSIGN);}
100 "*=" {printf("%40s - OPERATOR - %d\n", yytext, MULTASSIGN);}
101 "/=" {printf("%40s - OPERATOR - %d\n", yytext, DIVIDEASSIGN);}
102 "%=" {printf("%40s - OPERATOR - %d\n", yytext, MODASSIGN);}
103 "-" {printf("%40s - OPERATOR - %d\n", yytext, MINUS);}
104 "!" {printf("%40s - OPERATOR - %d\n", yytext, NOT);}
105 "+" {printf("%40s - OPERATOR - %d\n", yytext, PLUS);}
106 "*" {printf("%40s - OPERATOR - %d\n", yytext, MULT);}
107 "/" {printf("%40s - OPERATOR - %d\n", yytext, DIV);}
108 "%" {printf("%40s - OPERATOR - %d\n", yytext, MOD);}
109 ">" {printf("%40s - OPERATOR - %d\n", yytext, GREATER);}
110 "<" {printf("%40s - OPERATOR - %d\n", yytext, LESSER);}
111 "=" {printf("%40s - OPERATOR - %d\n", yytext, ASSIGNMENT);}
112
113 . {printf("ERROR IN %d : ILLEGAL CHARACTER: %s\n",yylineno,yytext);}
114
115 %%
116
117 int yywrap(){}
118 int main() {
119     yyin = fopen("test.c", "r");
120     yylex();
121     return 0;
122 }

```

Figure 2: Code for Scanner

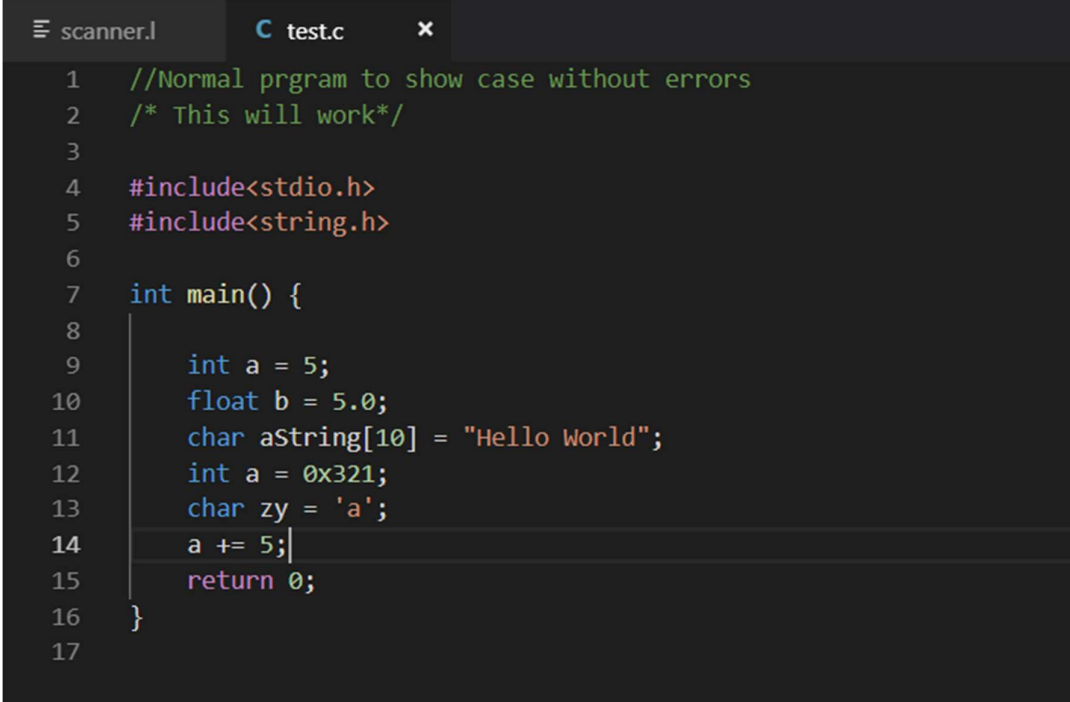
EXPLANATION

The flex script recognizes 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 incomplete strings
- Keywords
- Identifiers Keywords accounted for: int, char, float, signed, unsigned, short, long, long long, const, break, continue, for, while, if, else, void, enum and return.

TEST CASES

WITHOUT ERRORS



```
1 //Normal program to show case without errors
2 /* This will work*/
3
4 #include<stdio.h>
5 #include<string.h>
6
7 int main() {
8
9     int a = 5;
10    float b = 5.0;
11    char aString[10] = "Hello World";
12    int a = 0x321;
13    char zy = 'a';
14    a += 5;
15    return 0;
16 }
17
```

Figure 3: Test Case Figure

Overview

- The program is fed with the above C program which is error free.
- It contains both single line and multi-line comments.
- It contains int, float and char initializations.
- It contains the use of operators.
- It contains the use of various constants as well as special symbols.
- The output for the following is shown in the following page.

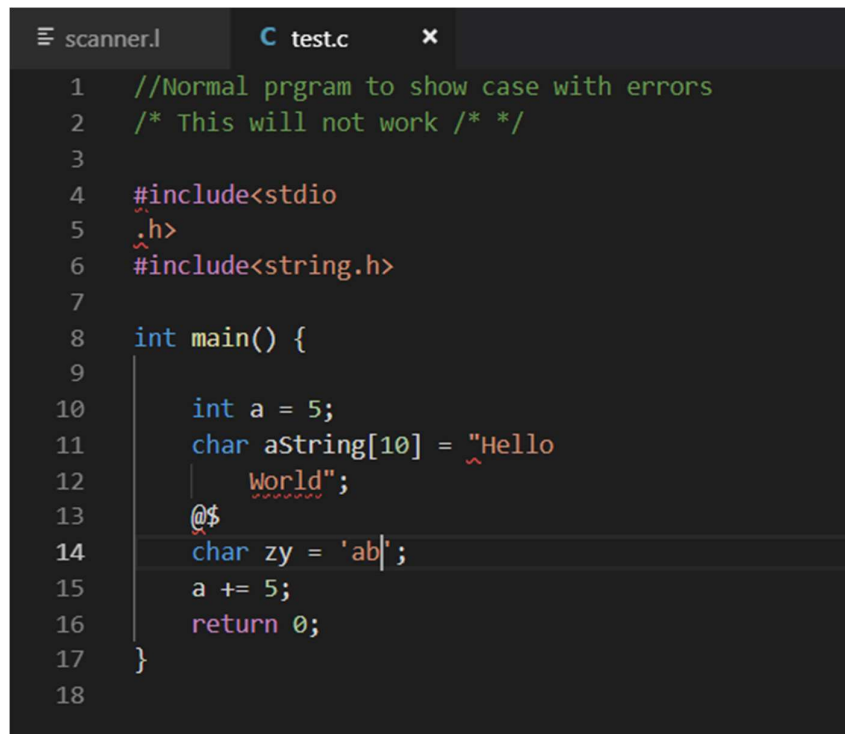
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
<stdio.h> - HEADER FILE - 305
<string.h> - HEADER FILE - 305
int - KEYWORD - 100
main - IDENTIFIER - 200
( - SPECIAL SYMBOL - 505
) - SPECIAL SYMBOL - 506
{ - SPECIAL SYMBOL - 503
int - KEYWORD - 100
a - IDENTIFIER - 200
= - OPERATOR - 414
5 - CONSTANT - 302
; - SPECIAL SYMBOL - 508
float - KEYWORD - 102
b - IDENTIFIER - 200
= - OPERATOR - 414
5.0 - CONSTANT - 301
; - SPECIAL SYMBOL - 508
char - KEYWORD - 101
aString - IDENTIFIER - 200
[ - SPECIAL SYMBOL - 501
10 - CONSTANT - 302
] - SPECIAL SYMBOL - 502
= - OPERATOR - 414
"Hello World" - CONSTANT - 304
; - SPECIAL SYMBOL - 508
int - KEYWORD - 100
a - IDENTIFIER - 200
= - OPERATOR - 414
0x321 - CONSTANT - 300
; - SPECIAL SYMBOL - 508
char - KEYWORD - 101
zy - IDENTIFIER - 200
= - OPERATOR - 414
'a' - CONSTANT - 303
; - SPECIAL SYMBOL - 508
a - IDENTIFIER - 200
+= - OPERATOR - 417
5 - CONSTANT - 302
; - SPECIAL SYMBOL - 508
return - KEYWORD - 109
0 - CONSTANT - 302
; - SPECIAL SYMBOL - 508
}
```

Figure 4: Output for Error free

Overview

- All tokens are identified.
- Comments are removed.
- Whitespaces have been removed.
- No error messages are shown.

WITH ERRORS



```
1 //Normal program to show case with errors
2 /* This will not work */
3
4 #include<stdio
5 .h>
6 #include<string.h>
7
8 int main() {
9
10     int a = 5;
11     char aString[10] = "Hello
12         World";
13     @$
14     char zy = 'ab|';
15     a += 5;
16     return 0;
17 }
18
```

Figure 5: Test case with Error

Overview

- The program is fed with the above C program which is erroneous.
- It contains nested multi line comments.
- It contains invalid header file calls.
- It contains invalid string declarations.
- It contains invalid character declarations.
- It contains illegal characters not in the language.
- The output for the following is shown in the following page.

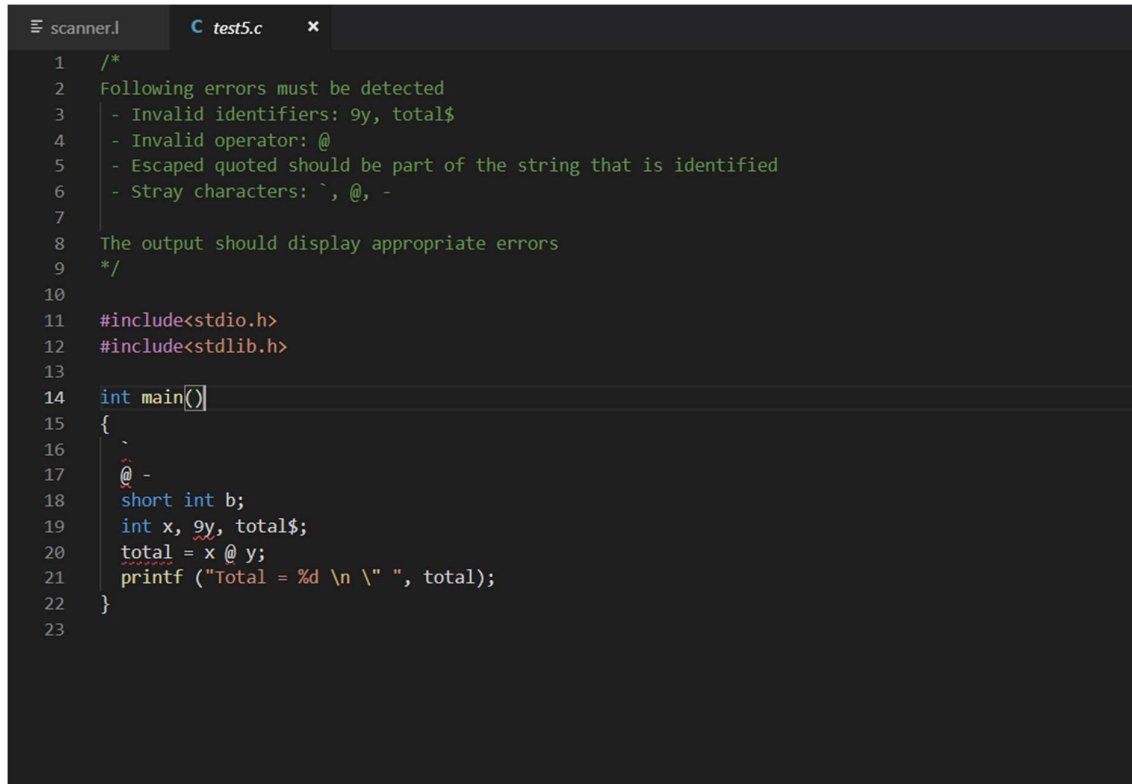
```
PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL
ERROR IN 2 : INVALID TOKEN NESTED COMMENTS INVALID
ERROR IN 4: ILLEGAL HEADER FORMAT
stdio - IDENTIFIER - 200
ERROR IN 5 : ILLEGAL CHARACTER: .
h - IDENTIFIER - 200
> - OPERATOR - 410
<string.h> - HEADER FILE - 305
int - KEYWORD - 100
main - IDENTIFIER - 200
( - SPECIAL SYMBOL - 505
) - SPECIAL SYMBOL - 506
{ - SPECIAL SYMBOL - 503
int - KEYWORD - 100
a - IDENTIFIER - 200
= - OPERATOR - 414
5 - CONSTANT - 302
; - SPECIAL SYMBOL - 508
char - KEYWORD - 101
aString - IDENTIFIER - 200
[ - SPECIAL SYMBOL - 501
10 - CONSTANT - 302
] - SPECIAL SYMBOL - 502
= - OPERATOR - 414
"Hello
- ERROR: ILLEGAL TOKEN
World - IDENTIFIER - 200
";
- ERROR: ILLEGAL TOKEN
ERROR IN 11 : ILLEGAL CHARACTER: @
ERROR IN 11 : ILLEGAL CHARACTER: $
char - KEYWORD - 101
zy - IDENTIFIER - 200
= - OPERATOR - 414
'ab' - ERROR: ILLEGAL TOKEN
; - SPECIAL SYMBOL - 508
a - IDENTIFIER - 200
+= - OPERATOR - 417
5 - CONSTANT - 302
; - SPECIAL SYMBOL - 508
return - KEYWORD - 109
0 - CONSTANT - 302
; - SPECIAL SYMBOL - 508
} - SPECIAL SYMBOL - 504
nihalh55@LAPTOP-9814AQOH:/mnt/c/Users/nihal/Desktop/6th Semester/Compiler Design/rscc/Phase 1/lexical-analyser$
```

Figure 6: Out for Erroneous Test Case

Overview

- Appropriate error messages have been shown for invalid header files.
- Appropriate error messages have been shown for invalid nested comments.
- Appropriate error messages have been shown for invalid character declarations.
- Appropriate error messages have been shown for invalid string declarations.
- Appropriate error messages have been shown for invalid tokens.

OTHER CASE - 1



```
1  /*
2  Following errors must be detected
3  - Invalid identifiers: 9y, total$
4  - Invalid operator: @
5  - Escaped quoted should be part of the string that is identified
6  - Stray characters: `, @, -
7
8  The output should display appropriate errors
9  */
10
11 #include<stdio.h>
12 #include<stdlib.h>
13
14 int main()
15 {
16     `
17     @ -
18     short int b;
19     int x, 9y, total$;
20     total = x @ y;
21     printf ("Total = %d \n \" ", total);
22 }
23
```

Figure 7: Test case

Overview

- The program is fed with the above C program which is erroneous.
- It contains invalid character declarations.
- It contains illegal characters not in the language.
- The output for the following is shown in the following page.

```

nihalh55@LAPTOP-9814AQOH:/mnt/c/Users/nihal/Desktop/6th Semester/Compiler Design/rsc/Phase 1/lexical-analyser$ ./a.out tests/test5.c
<stdio.h> - HEADER FILE - 305
<stdlib.h> - HEADER FILE - 305
int - KEYWORD - 100
main - IDENTIFIER - 200
( - SPECIAL SYMBOL - 505
) - SPECIAL SYMBOL - 506
{ - SPECIAL SYMBOL - 503
ERROR IN 16 : ILLEGAL CHARACTER: `
ERROR IN 17 : ILLEGAL CHARACTER: @
- - OPERATOR - 400
short - KEYWORD - 105
int - KEYWORD - 100
b - IDENTIFIER - 200
; - SPECIAL SYMBOL - 508
int - KEYWORD - 100
x - IDENTIFIER - 200
, - SPECIAL SYMBOL - 507
ERROR IN 19: INVALID IDENTIFIER FORMAT
, - SPECIAL SYMBOL - 507
total - IDENTIFIER - 200
ERROR IN 19 : ILLEGAL CHARACTER: $
; - SPECIAL SYMBOL - 508
total - IDENTIFIER - 200
= - OPERATOR - 414
x - IDENTIFIER - 200
ERROR IN 20 : ILLEGAL CHARACTER: @
y - IDENTIFIER - 200
; - SPECIAL SYMBOL - 508
printf - IDENTIFIER - 200
( - SPECIAL SYMBOL - 505
"Total = %d \n \" - CONSTANT - 304
, - SPECIAL SYMBOL - 507
total - IDENTIFIER - 200
) - SPECIAL SYMBOL - 506
; - SPECIAL SYMBOL - 508
} - SPECIAL SYMBOL - 504
nihalh55@LAPTOP-9814AQOH:/mnt/c/Users/nihal/Desktop/6th Semester/Compiler Design/rsc/Phase 1/lexical-analyser$ █

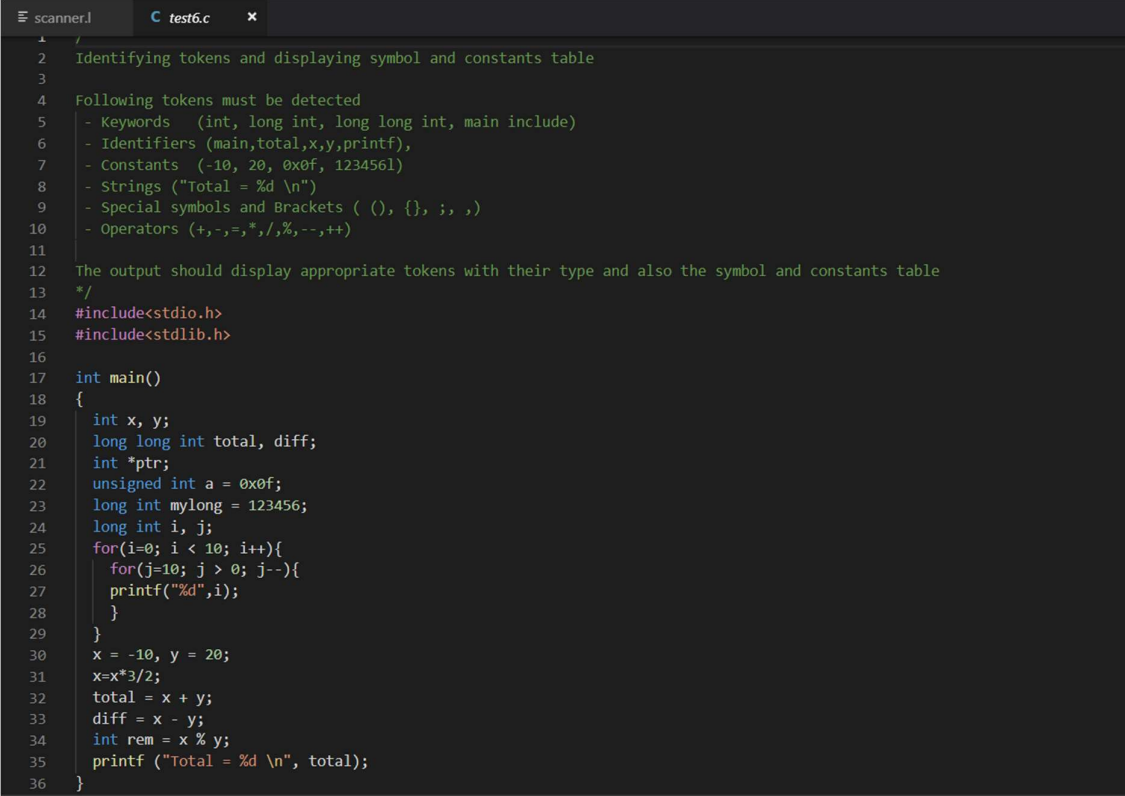
```

Figure 8: Output Test case

Overview

- Appropriate error messages have been shown for all cases.
- Appropriate token messages have shown.

OTHER CASE – 2



```
scanner.l  C test6.c x
1  /
2  Identifying tokens and displaying symbol and constants table
3
4  Following tokens must be detected
5  - Keywords (int, long int, long long int, main include)
6  - Identifiers (main,total,x,y,printf),
7  - Constants (-10, 20, 0x0f, 123456l)
8  - Strings ("Total = %d \n")
9  - Special symbols and Brackets ( (), {}, ;, ,)
10 - Operators (+, -, =, *, /, %, --, ++ )
11
12 The output should display appropriate tokens with their type and also the symbol and constants table
13 */
14 #include<stdio.h>
15 #include<stdlib.h>
16
17 int main()
18 {
19     int x, y;
20     long long int total, diff;
21     int *ptr;
22     unsigned int a = 0x0f;
23     long int mylong = 123456;
24     long int i, j;
25     for(i=0; i < 10; i++){
26         for(j=10; j > 0; j--){
27             printf("%d",i);
28         }
29     }
30     x = -10, y = 20;
31     x=x*3/2;
32     total = x + y;
33     diff = x - y;
34     int rem = x % y;
35     printf ("Total = %d \n", total);
36 }
```

Figure 9: Test case

Overview

- The program is fed with the above C program which is normal executable one.
- The output for the following is shown in the following page.


```
PROBLEMS 7 OUTPUT DEBUG CONSOLE TERMINAL
<stdlib.h>
int
main
(
)
{
int
x
,
y
;
long long
int
total
,
diff
;
int
*
ptr
;
unsigned
int
a
=
0x0f
;
long
int
mylong
=
123456
;
long
int
i
,
j
;
for
(
i
=
0
- HEADER FILE - 305
- KEYWORD - 100
- IDENTIFIER - 200
- SPECIAL SYMBOL - 505
- SPECIAL SYMBOL - 506
- SPECIAL SYMBOL - 503
- KEYWORD - 100
- IDENTIFIER - 200
- SPECIAL SYMBOL - 507
- IDENTIFIER - 200
- SPECIAL SYMBOL - 508
- KEYWORD - 107
- KEYWORD - 100
- IDENTIFIER - 200
- SPECIAL SYMBOL - 507
- IDENTIFIER - 200
- SPECIAL SYMBOL - 508
- KEYWORD - 100
- OPERATOR - 405
- IDENTIFIER - 200
- SPECIAL SYMBOL - 508
- KEYWORD - 104
- KEYWORD - 100
- IDENTIFIER - 200
- OPERATOR - 414
- CONSTANT - 300
- SPECIAL SYMBOL - 508
- KEYWORD - 106
- KEYWORD - 100
- IDENTIFIER - 200
- OPERATOR - 414
- CONSTANT - 302
- SPECIAL SYMBOL - 508
- KEYWORD - 106
- KEYWORD - 100
- IDENTIFIER - 200
- SPECIAL SYMBOL - 507
- IDENTIFIER - 200
- SPECIAL SYMBOL - 508
- KEYWORD - 112
- SPECIAL SYMBOL - 505
- IDENTIFIER - 200
- OPERATOR - 414
- CONSTANT - 302
```

Figure 10: Output 1

PROBLEMS	7	OUTPUT	DEBUG CONSOLE	TERMINAL
;				- SPECIAL SYMBOL - 508
i				- IDENTIFIER - 200
<				- OPERATOR - 411
10				- CONSTANT - 302
;				- SPECIAL SYMBOL - 508
i				- IDENTIFIER - 200
++				- OPERATOR - 401
)				- SPECIAL SYMBOL - 506
{				- SPECIAL SYMBOL - 503
for				- KEYWORD - 112
(- SPECIAL SYMBOL - 505
j				- IDENTIFIER - 200
=				- OPERATOR - 414
10				- CONSTANT - 302
;				- SPECIAL SYMBOL - 508
j				- IDENTIFIER - 200
>				- OPERATOR - 410
0				- CONSTANT - 302
;				- SPECIAL SYMBOL - 508
j				- IDENTIFIER - 200
--				- OPERATOR - 402
)				- SPECIAL SYMBOL - 506
{				- SPECIAL SYMBOL - 503
printf				- IDENTIFIER - 200
(- SPECIAL SYMBOL - 505
"%d"				- CONSTANT - 304
,				- SPECIAL SYMBOL - 507
i				- IDENTIFIER - 200
)				- SPECIAL SYMBOL - 506
;				- SPECIAL SYMBOL - 508
}				- SPECIAL SYMBOL - 504
}				- SPECIAL SYMBOL - 504
x				- IDENTIFIER - 200
=				- OPERATOR - 414
-10				- CONSTANT - 302
,				- SPECIAL SYMBOL - 507
y				- IDENTIFIER - 200
=				- OPERATOR - 414
20				- CONSTANT - 302
;				- SPECIAL SYMBOL - 508
x				- IDENTIFIER - 200
=				- OPERATOR - 414
x				- IDENTIFIER - 200
*				- OPERATOR - 405

Figure 11: Output 2

```
PROBLEMS 7 OUTPUT DEBUG CONSOLE TERMINAL
x          - IDENTIFIER - 200
=          - OPERATOR - 414
-10        - CONSTANT - 302
,          - SPECIAL SYMBOL - 507
y          - IDENTIFIER - 200
=          - OPERATOR - 414
20         - CONSTANT - 302
;          - SPECIAL SYMBOL - 508
x          - IDENTIFIER - 200
=          - OPERATOR - 414
x          - IDENTIFIER - 200
*          - OPERATOR - 405
3          - CONSTANT - 302
/          - OPERATOR - 406
2          - CONSTANT - 302
;          - SPECIAL SYMBOL - 508
total      - IDENTIFIER - 200
=          - OPERATOR - 414
x          - IDENTIFIER - 200
+          - OPERATOR - 404
y          - IDENTIFIER - 200
;          - SPECIAL SYMBOL - 508
diff       - IDENTIFIER - 200
=          - OPERATOR - 414
x          - IDENTIFIER - 200
-          - OPERATOR - 400
y          - IDENTIFIER - 200
;          - SPECIAL SYMBOL - 508
int        - KEYWORD - 100
rem        - IDENTIFIER - 200
=          - OPERATOR - 414
x          - IDENTIFIER - 200
%          - OPERATOR - 407
y          - IDENTIFIER - 200
;          - SPECIAL SYMBOL - 508
printf     - IDENTIFIER - 200
(          - SPECIAL SYMBOL - 505
"Total = %d \n" - CONSTANT - 304
,          - SPECIAL SYMBOL - 507
total      - IDENTIFIER - 200
)          - SPECIAL SYMBOL - 506
;          - SPECIAL SYMBOL - 508
}          - SPECIAL SYMBOL - 504
nihalh55@LAPTOP-9814AQOH:/mnt/c/Users/nihal/Desktop/6th Semester/Compiler Design/rscc/Phase 1/lexical-analyser$
```

Figure 12: Output 3

Overview

- Appropriate error messages have been shown for all cases.
- Appropriate token messages have shown.

IMPLEMENTATION

OVERVIEW

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***: 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 tokens present in the program. We use the following technique to implement this:

- ❖ A token.h header file was created which contains enumerated objects for each token.
- ❖ Once a token is matched with a particular regex, that is, it gets identified, then the token along with its unique ID obtained from the header file is displayed in the terminal.
- ❖ Error messages are displayed with reason along with the line number.

RESULTS, FUTURE WORK AND REFERENCES

RESULTS

A scanner has been created which is capable of taking a program written in C language as an input and give out a stream of tokens present in the program as the output. It removes whitespaces, comments and shows necessary error messages.

FUTURE WORK

We wish to first incorporate a symbol table functionality into this scanner as it will be required for the making of the parser in the second phase of the project.

Furthermore, the flex script presented in this report takes care of most of 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. Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986
2. <http://www.csd.uwo.ca/~moreno/CS447/Lectures/Lexical.html/node11.html>