

Lexical Analyzer for the C Language



National Institute of Technology Karnataka

Date: 22nd January 2020

Submitted To: Dr. P. Santhi Thilagam

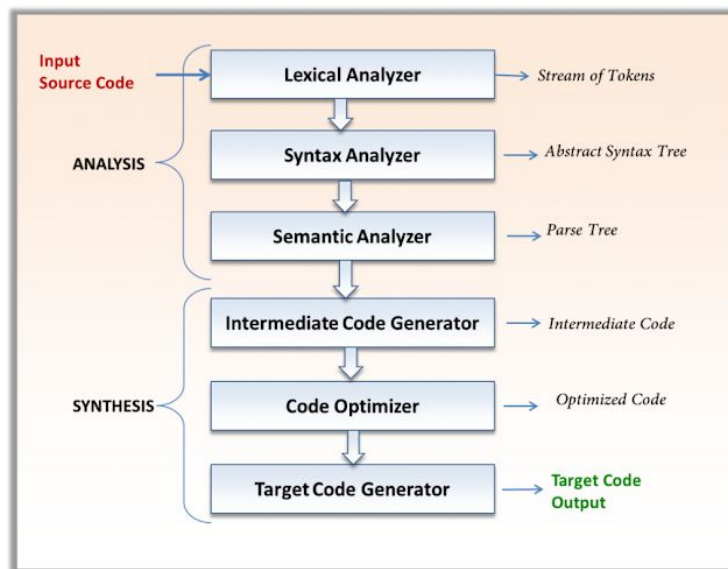
Group Members:

1. Nihal Haneef	16CO128
2. Akash Nair	171CO107
3. Meghna Savit	171CO222

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 I

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. 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. 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 syntactic 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, commas, semicolons, assignment operator and preprocessor directive.

TOOLS & 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.....	4
Lexical Analyzer.....	7
Flex Script	7
C Program	8
Symbol Table and Constant Table	8
Code	9
Explanation	12
Without Errors	13
With Errors	15
Other Case – 1	17
Other Case – 2	19
Other Case – 3	20
Implementation: Overview	24
Results	25
Future Work	25
References	25

TABLE OF FIGURES

Figure 1: Code for Scanner	9
Figure 2: Code for Scanner	10
Figure 3: Code for Scanner	10
Figure 4: Code for token.h	11
Figure 5: Code for No Errors	13
Figure 6: Output for No Errors	14
Figure 7: Code for Errors	15
Figure 8: Output for Errors	16
Figure 9: Code for Other Test Case -1	17
Figure 10: Output for Other Test Case -1	18
Figure 11: Code for Other Test Case -2	19
Figure 12: Output for Other Test Case -2	19
Figure 13: Code for Other Test Case -3	20
Figure 14: Output for Other Test Case -3.1	21
Figure 15: Output for Other Test Case -3.2	22
Figure 16: Output for Other Test Case -3.3	23

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.

SYMBOL TABLE & CONSTANT TABLE

This section contains the definition of the symbol table and the constants table and also defines functions for inserting into the table and displaying its contents within the `table.h` file, & contains enumerated constants for keywords, operators, special symbols, constants and identifiers in `tokens.h`.

The symbol table contains the list of identifiers captured by the scanner and the constant table contains all the constants captured by the scanner along with the type of the constant. A simple structure array was used to create a linear table so as to store this information.

DESIGN OF PROGRAMS

CODE

```

scanner.l
1  %
2
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <limits.h>
6  #include "token.h"
7
8  int start = 0;
9  struct symbol
10 {
11     char token[100];
12     char type[100];
13 }symbolTable[100000], constantTable[100000];
14
15 int i=0;
16 int c=0;
17
18 int symbolLookup(char* tokenName)
19 {
20     int k=0,flag=0;
21     for(k=0;k<=i;k++)
22     {
23         if(strcmp(symbolTable[k].token,tokenName)==0)
24         {
25             flag=1;
26             break;
27         }
28     }
29     return flag;
30 }
31
32 void symbolInsert(struct symbol table[], int index, char* tokenName, char* tokenType)
33 {
34     int flag;
35     flag=symbolLookup(tokenName);
36     if(flag==0)
37     {
38         strcpy(table[index].token, tokenName);
39         strcpy(table[index].type, tokenType);
40         i++;
41     }
42 }
43
44 void constantInsert(struct symbol table[], int index, char* constant, char* constantType)
45 {
46     strcpy(table[index].token, constant);
47     strcpy(table[index].type, constantType);
48     c++;
49 }
50
51 void display()
52 {
53     int k=0;
54     printf("\nSYMBOL TABLE \n\n");
55     printf("%-10s \t\t\t %s\n","Symbol","Type");
56     for(k=0;k<=i;k++)
57         printf("%-10s\t\t\t%-40s\n",symbolTable[k].token,symbolTable[k].type);
58     printf("\nCONSTANT TABLE \n\n");
59     printf("%-10s \t\t\t %s\n","Constant","Type");
60     for(k=0;k<=c;k++)
61         printf("%-10s\t\t\t%-40s\n",constantTable[k].token,constantTable[k].type);
62 }
63
64
65 %
66

```

Figure 1: Code for Scanner

```

17 LETTER [a-zA-Z]
18 DIGIT [0-9]
19 WS [ \t\n\r\v\f]+
20 IDENTIFIER_C ({LETTER}|{DIGIT})_*
21 HEX [0-9a-f]
22
23 %x COMMENT
24 %x PREPROCESSOR
25
26 %%
27 \n
28
29 #include
30 <PREPROCESSOR>({WS})?<">({LETTER})**>.>h>
31 <PREPROCESSOR>({WS})?<">({LETTER})**>.>h>
32 <PREPROCESSOR>
33
34 "int"
35 "char"
36 "float"
37 "void"
38 "long"
39 "long long"
40 "short"
41 "signed"
42 "unsigned"
43 "for"
44 "while"
45 "break"
46 "continue"
47 "if"
48 "else"
49 "return"
50 "const"
51 "enum"
52 "switch"
53
54 (DIGIT)+({LETTER})_*
55 IDENTIFIER_C
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Figure 2: Code for Scanner

```

132 '\{[^\n\}^\'' (printf("%-40s - CONSTANT - %d\n",yytext,SIZE); constantInsert(constantTable,c,yytext,"SIZE");)
133 '\{[^\n\}^\'' (printf("ERROR AT %d: UNTERMINATED CHAR\n", yylineno);)
134 '\{[^\n\}^\'' (printf("%-40s - CONSTANT - %d\n",yytext,CHARCONSTANT); constantInsert(constantTable,c,yytext,"CHAR");)
135 '[' (printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,LEFTBRACKET);)
136 ']' (printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,RIGHTBRACKET);)
137 '{' (printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,LEFTBRACE);)
138 '}' (printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,RIGHTBRACE);)
139 '(' (printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,LEFTPARENTHESES);)
140 ')' (printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,RIGHTPARENTHESES);)
141 ':' (printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,SEMICOLON);)
142 ',' (printf("%-40s - SPECIAL SYMBOL - %d\n",yytext,COMMA);)
143
144 "++" (printf("%-40s - OPERATOR - %d\n",yytext,INCREMENT);)
145 "--" (printf("%-40s - OPERATOR - %d\n",yytext,DECREMENT);)
146 "==" (printf("%-40s - OPERATOR - %d\n",yytext,EQUAL);)
147 "!=" (printf("%-40s - OPERATOR - %d\n",yytext,NOTEQUAL);)
148 ">=" (printf("%-40s - OPERATOR - %d\n",yytext,GREATEREQUAL);)
149 "<=" (printf("%-40s - OPERATOR - %d\n",yytext,LESSEREQUAL);)
150 "&&" (printf("%-40s - OPERATOR - %d\n",yytext,LOGICALAND);)
151 "||" (printf("%-40s - OPERATOR - %d\n",yytext,LOGICALOR);)
152 "+" (printf("%-40s - OPERATOR - %d\n",yytext,ADDASSIGN);)
153 "-" (printf("%-40s - OPERATOR - %d\n",yytext,SUBTRACTASSIGN);)
154 "*" (printf("%-40s - OPERATOR - %d\n",yytext,MULTASSIGN);)
155 "/" (printf("%-40s - OPERATOR - %d\n",yytext,DIVIDEASSIGN);)
156 "%" (printf("%-40s - OPERATOR - %d\n",yytext,MODASSIGN);)
157
158 "-" (printf("%-40s - OPERATOR - %d\n",yytext,MINUS);)
159 "+" (printf("%-40s - OPERATOR - %d\n",yytext,NOT);)
160 "+" (printf("%-40s - OPERATOR - %d\n",yytext,PLUS);)
161 "*" (printf("%-40s - OPERATOR - %d\n",yytext,MULT);)
162 "/" (printf("%-40s - OPERATOR - %d\n",yytext,DIV);)
163 "%" (printf("%-40s - OPERATOR - %d\n",yytext,MOD);)
164 ">" (printf("%-40s - OPERATOR - %d\n",yytext,GREATER);)
165 "<" (printf("%-40s - OPERATOR - %d\n",yytext,LESSER);)
166 "=" (printf("%-40s - OPERATOR - %d\n",yytext,ASSIGNMENT);)
167
168 "." (printf("ERROR AT %d: ILLEGAL CHARACTER\n", yylineno);)
169
170 "%"
171
172
173
174
175 int yywrap(){}
176 int main(){
177     int newtoken;
178     yyin = fopen("test6.c","r");
179     yylex();
180     display();
181     return 0;
182 }
183

```

Figure 3: Code for Scanner

```

1  #ifndef TOKEN_H
2  #define TOKEN_H
3
4  enum keywords
5  {
6      INT=100,
7      CHAR,
8      FLOAT,
9      SIGNED,
10     UNSIGNED,
11     SHORT,
12     LONG,
13     LONGLONG,
14     CONST,
15     RETURN,
16     IF,
17     ELSE,
18     FOR,
19     WHILE,
20     BREAK,
21     CONTINUE,
22     VOID,
23     ENUM,
24     SWITCH
25 };
26
27 enum IDENTIFIER
28 {
29     IDENTIFIER=200
30 };
31
32 enum constants
33 {
34     HEXCONSTANT=300,
35     FLOATCONSTANT,
36     DECIMALCONSTANT,
37     CHARCONSTANT,
38     STR,
39     HEADERFILE,
40     MACRO
41 };
42
43 enum operators
44 {
45     MINUS=400,
46     INCREMENT,
47     DECREMENT,
48     NOT,
49     PLUS,
50     MULT,
51     DIV,
52     MOD,
53     EQUAL,
54     NOTEQUAL,
55     GREATER,
56     LESSER,
57     GREATEREQUAL,
58     LESSEREQUAL,
59     ASSIGNMENT,
60     LOGICALAND,
61     LOGICALOR,
62     ADDASSIGN,
63     SUBTRACTASSIGN,
64     MULTASSIGN,
65     DIVIDEASSIGN,
66     MODASSIGN
67 };
68
69 enum special_symbols
70 {
71     DELIMITER=500,
72     LEFTBRACKET,
73     RIGHTBRACKET,
74     LEFTBRACE,
75     RIGHTBRACE,
76     LEFTPARANTHESES,
77     RIGHTPARANTHESES,
78     COMMA,
79     SEMICOLON
80 };
81
82 #endif

```

Figure 4: Code for token.h

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

```
C test.c
1  // Normal Program to show case program with no errors
2  /* This will work */
3
4  #include<stdio.h>
5  #include<string.h>
6
7  int main() {
8      int a = 5;
9      float b = 5.0;
10     char aString[10] = "Hello World";
11     int a = 0x321;
12     char zy = 'a';
13     a+=5;
14     return 0;
15 }
```

Figure 5: Code for No Errors

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.

```

<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
} - SPECIAL SYMBOL - 504

SYMBOL TABLE

Symbol      Type
main        IDENTIFIER
a            IDENTIFIER
b            IDENTIFIER
aString     IDENTIFIER
zy          IDENTIFIER

CONSTANT TABLE

Constant    Type
5            INTEGER CONSTANT
5.0          FLOAT CONSTANT
10           INTEGER CONSTANT
"Hello World" STRING
0x321        HEX CONSTANT
'a'          CHAR
5            INTEGER CONSTANT
0            INTEGER CONSTANT

```

Figure 6: Output for No Errors

Overview:

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

WITH ERRORS

```
C test1.c
1  /*
2  | - An All in one test case with all kinds of errors.
3  */
4
5  /*This should not*/ work */
6  #include<stdio
7  .h>
8  #include<string.h>
9
10 int main() {
11
12     int a = 5;
13     char aString[10] = "Hello
14         World";
15     @$
16     char zy = 'ab';
17     a += 5;
18     return 0;
19 }
```

Figure 7: Code for Errors

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.


```

work - IDENTIFIER - 200
ERROR AT 5: NESTED COMMENT
ERROR AT 6: ILLEGAL HEADER FORMAT
stdio - IDENTIFIER - 200
ERROR AT 7: 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
ERROR AT 13: UNTERMINATED STRING
world - IDENTIFIER - 200
ERROR AT 13: UNTERMINATED STRING
ERROR AT 13: ILLEGAL CHARACTER
ERROR AT 13: ILLEGAL CHARACTER
char - KEYWORD - 101
zy - IDENTIFIER - 200
= - OPERATOR - 414
ERROR AT 14: UNTERMINATED CHAR
; - 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

SYMBOL TABLE

Symbol      Type
work        IDENTIFIER
stdio        IDENTIFIER
h            IDENTIFIER
main         IDENTIFIER
a            IDENTIFIER
aString      IDENTIFIER
world        IDENTIFIER
zy           IDENTIFIER

CONSTANT TABLE

Constant    Type
5            INTEGER CONSTANT
10           INTEGER CONSTANT
5            INTEGER CONSTANT
0            INTEGER CONSTANT

```

Figure 8: Output for Errors

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  | - Test for single line comments
3  | - Test for multi-line comments
4  | - Test for single line nested comments
5  | - Test for multiline nested comments
6  |
7  | The output in lex should remove all the comments including this one
8  | */
9  |
10 | #include<stdio.h>
11 |
12 | void main(){s
13 |     // Single line comment
14 |
15 |     /* Multi-line comment
16 |        Like this */
17 |
18 |     /* here */ int a; /* "int a" should be untouched */
19 |
20 |     // This nested comment // This comment should be removed should be removed
21 |
22 |
23 |
24 |     /* To make things /* nested multi-line comment */ interesting */
25 |
26 |     return 0;
27 | }
28 |
```

Figure 9: Code for Other Test Case -1

Overview:

- The program is fed with the above C program which is erroneous.
- It contains single and multiline comments.
- The output for the following is shown in the following page.

```

<stdio.h>                                - HEADER FILE - 305
void                                       - KEYWORD - 116
main                                      - IDENTIFIER - 200
(                                         - SPECIAL SYMBOL - 505
)                                         - SPECIAL SYMBOL - 506
{                                         - SPECIAL SYMBOL - 503
s                                         - IDENTIFIER - 200
int                                       - KEYWORD - 100
a                                         - IDENTIFIER - 200
;                                         - SPECIAL SYMBOL - 508
interesting                              - IDENTIFIER - 200
ERROR AT 24: NESTED COMMENT
return                                   - KEYWORD - 109
0                                         - CONSTANT - 302
;                                         - SPECIAL SYMBOL - 508
}                                         - SPECIAL SYMBOL - 504

SYMBOL TABLE

Symbol                                Type
main                                IDENTIFIER
s                                    IDENTIFIER
a                                    IDENTIFIER
interesting                          IDENTIFIER

CONSTANT TABLE

Constant                              Type
0                                      INTEGER CONSTANT

```

Figure 10: Output for Other Test Case -1

Overview:

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

OTHER CASE - 2

```

/*
- Test for multi-line comment that doesn't end till EOF
The output in lex should print as error message when the comment does not terminate
It should remove the comments that terminate
s*/

#include<stdio.h>

void main(){

    // This is fine
    /* This as well
    like we know */

    /* This is not fine since
    this comment has to end somewhere

    return 0;
}

```

Figure 11: Code for Other Test Case -2

Overview:

- The program is fed with the above C program which does not end till EOF.
- The output for the following is shown in the following page.

```

<stdio.h>                                - HEADER FILE - 305
void                                       - KEYWORD - 116
main                                      - IDENTIFIER - 200
(                                         - SPECIAL SYMBOL - 505
)                                         - SPECIAL SYMBOL - 506
{                                         - SPECIAL SYMBOL - 503
ERROR AT 16: UNTERMINATED COMMENT

SYMBOL TABLE

Symbol                                     Type
main                                     IDENTIFIER

CONSTANT TABLE

Constant                                 Type

```

Figure 12: Output for Other Test Case -2

Overview:

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

OTHER CASE - 3

```

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 (+, -, =, *, /, %, --, ++)
```

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     float lk = 0.123456;
23     unsigned int a = 0x0f;
24     long int mylong = 123456;
25     long int i, j;
26     for(i=0; i < 10; i++){
27         for(j=10; j > 0; j--){
28             printf("%d",i);
29         }
30     }
31     x = -10, y = 20;
32     x=x*3/2;
33     total = x + y;
34     diff = x - y;
35     int rem = x % y;
36     printf ("Total=%d \n", total);
37 }
```

Figure 13: Code for Other Test Case -3

Overview:

- The program is fed with the above C program which does not end till EOF.
- The output for the following is shown in the following page.

<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
int	- KEYWORD - 100
x	- IDENTIFIER - 200
,	- SPECIAL SYMBOL - 507
y	- IDENTIFIER - 200
;	- SPECIAL SYMBOL - 508
long long	- KEYWORD - 107
int	- KEYWORD - 100
total	- IDENTIFIER - 200
,	- SPECIAL SYMBOL - 507
diff	- IDENTIFIER - 200
;	- SPECIAL SYMBOL - 508
int	- KEYWORD - 100
*	- OPERATOR - 405
ptr	- IDENTIFIER - 200
;	- SPECIAL SYMBOL - 508
float	- KEYWORD - 102
lk	- IDENTIFIER - 200
=	- OPERATOR - 414
0.123456	- CONSTANT - 301
;	- SPECIAL SYMBOL - 508
unsigned	- KEYWORD - 104
int	- KEYWORD - 100
a	- IDENTIFIER - 200
=	- OPERATOR - 414
0x0f	- CONSTANT - 300
;	- SPECIAL SYMBOL - 508
long	- KEYWORD - 106
int	- KEYWORD - 100
mylong	- IDENTIFIER - 200
=	- OPERATOR - 414
123456	- CONSTANT - 302
;	- SPECIAL SYMBOL - 508
long	- KEYWORD - 106
int	- KEYWORD - 100
i	- IDENTIFIER - 200
,	- SPECIAL SYMBOL - 507
j	- IDENTIFIER - 200
;	- SPECIAL SYMBOL - 508
for	- KEYWORD - 112
(- SPECIAL SYMBOL - 505
i	- IDENTIFIER - 200
=	- OPERATOR - 414
0	- CONSTANT - 302
;	- SPECIAL SYMBOL - 508
i	- IDENTIFIER - 200
<	- OPERATOR - 411
10	- CONSTANT - 302
;	- SPECIAL SYMBOL - 508

Figure 14: Output for Other Test Case –3.1

```

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
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
return - 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

```

Figure 15: Output for Other Test Case –3.2

SYMBOL TABLE	
Symbol	Type
main	IDENTIFIER
x	IDENTIFIER
y	IDENTIFIER
total	IDENTIFIER
diff	IDENTIFIER
ptr	IDENTIFIER
lk	IDENTIFIER
a	IDENTIFIER
mylong	IDENTIFIER
i	IDENTIFIER
j	IDENTIFIER
printf	IDENTIFIER
rem	IDENTIFIER
CONSTANT TABLE	
Constant	Type
0.123456	FLOAT CONSTANT
0x8f	HEX CONSTANT
123456	INTEGER CONSTANT
0	INTEGER CONSTANT
10	INTEGER CONSTANT
10	INTEGER CONSTANT
0	INTEGER CONSTANT
"%d"	STRING
-10	INTEGER CONSTANT
20	INTEGER CONSTANT
3	INTEGER CONSTANT
2	INTEGER CONSTANT
"Total=%d \n"	STRING

Figure 16: Output for Other Test Case -3.3

Overview:

- Appropriate error messages have been shown for all cases.
- Appropriate token messages have shown.
- Symbols and Constants are shown respectively.

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 kinds 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 & 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

Furthermore, the flex script presented in this report takes care of most of the rules of the 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 the 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>