RAMAIAH UNIVERSITY OF APPLIED SCIENCES

# ASSIGNMENT - 1

| | |
|---|---|
| **Course Code** | 19CSC305A |
| **Course Name** | Compilers |
| **Programme** | B. Tech |
| **Department** | CSE |
| **Faculty** | FET |


| | |
|---|---|
| **Name of the Student** | K Srikanth |
| **Reg. No** | 17ETCS002124 |
| **Semester/Year** | 5th Semester/ 3rd Year |
| **Course Leader/s** | Mr. Hari Krishna S. M. |

RAMAIAH UNIVERSITY OF APPLIED SCIENCES

| Declaration Sheet | | | | |
|---|---|---|---|---|
| Student Name | K Srikanth | | | |
| Reg. No | 17ETCS002124 | | | |
| Programme | B. Tech | Semester/Year | 5th / 3rd | |
| Course Code | 19CSC305A | | | |
| Course Title | Compilers | | | |
| Course Date | 14/09/2020 | to | 16/02/2021 | |
| Course Leader | Mr. Hari Krishna S. M. | | | |

**Declaration**

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

| | | | |
|---|---|---|---|
| Signature of the Student | | Date | |
| Submission date stamp (by Examination & Assessment Section) | | | |

| Signature of the Course Leader and date | Signature of the Reviewer and date |
|---|---|
| | |

RAMAIAH UNIVERSITY OF APPLIED SCIENCES

| Faculty of Engineering and Technology | | | |
|---|---|---|---|
| Ramaiah University of Applied Sciences | | | |
| Department | Computer Science and Engineering | Programme | B. Tech in Computer Science and Engineering |
| Semester/Batch | 05th /2018 | | |
| Course Code | 19CSC305A | Course Title | Compilers |
| Course Leader | Mr. Hari Krishna S. M. & Ms. Suvidha | | |

| Assignment | | | | | |
|---|---|---|---|---|---|
| **Register No.** | K Srikanth | **Name of the Student** | 17ETCS002124 | | |
| **Sections** | | **Marking Scheme** | **Marks** | | |
| | | | **Max Marks** | **First Examiner Marks** | **Moderator** |
| **Part A 1** | | | | | |
| | **A 1.1** | Implementation in *Lex* | 06 | | |
| | **A 1.2** | Results and Comments | 04 | | |
| | | **Part-A 1 Max Marks** | **10** | | |
| | **A 2.1** | Implementation in *Lex* | 10 | | |
| | **A 2.2** | Results and Comments | 05 | | |
| | | **Part-A 2 Max Marks** | **15** | | |
| | | **Total Assignment Marks** | **25** | | |

| Course Marks Tabulation | | | | |
|---|---|---|---|---|
| **Component- CET B Assignment** | **First Examiner** | **Remarks** | **Second Examiner** | **Remarks** |
| A.1 | | | | |
| A.2 | | | | |
| **Marks (out of 25)** | | | | |

**Signature of First Examiner** **Signature of Moderator**

RAMAIAH UNIVERSITY OF APPLIED SCIENCES

**Please note:**

1. Documental evidence for all the components/parts of the assessment such as the reports, photographs, laboratory exam / tool tests are required to be attached to the assignment report in a proper order.
2. The First Examiner is required to mark the comments in RED ink and the Second Examiner's comments should be in GREEN ink.
3. The marks for all the questions of the assignment have to be written only in the **Component – CET B: Assignment** table.
4. If the variation between the marks awarded by the first examiner and the second examiner lies within +/- 3 marks, then the marks allotted by the first examiner is considered to be final. If the variation is more than +/- 3 marks then both the examiners should resolve the issue in consultation with the Chairman BoE.

## Assignment

**Instructions to students:**

1. The assignment consists of **1** questions: Part A – **2** Question.
2. Maximum marks is **25**.
3. The assignment has to be neatly word processed as per the prescribed format.
4. The maximum number of pages should be restricted to **25**.
5. The printed assignment must be submitted to the course leader.
6. **Submission Date: 28th November 2020**
7. **Submission after the due date is not permitted.**
8. **IMPORTANT**: It is essential that all the sources used in preparation of the assignment must be suitably referenced in the text.
9. Marks will be awarded only to the sections and subsections clearly indicated as per the problem statement/exercise/question

**Preamble:**

The aim of this course is to train the students in the design and implementation of compilers and various components of a compiler, including a scanner, parser, and code generator. The students are exposed to GNU compiler, construction tools and their application. Students are trained to design and implement a compiler for a simple language.

Name: K Srikanth Registration Number: 17ETCS002124

**Part A**

**Introduction**

Our aim is to build a Lexical analyzer or scanner that matches strings in the input using Lex, based on the patterns (regular expressions), and converts the strings to tokens.

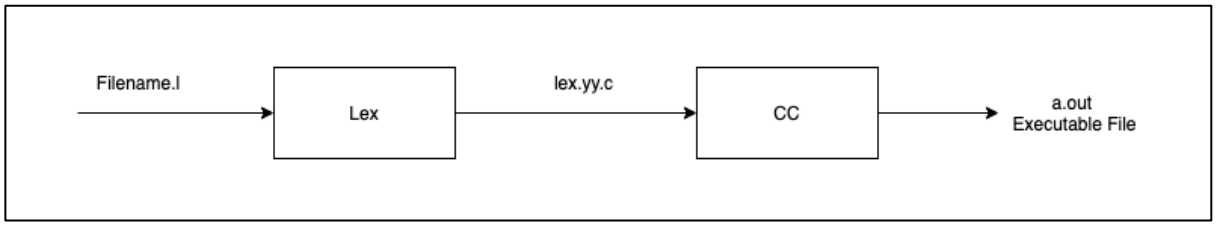


Figure 1 Flowchat of how lex file executes

From image 1 the process of building a Lexical analyzer is create a file with ".l" Expectation
And compile using flex with this command

```
Flex Filename.l
```

**After running the about command without errors you will get a lex.yy.c file and compile the produced c file using this command**

```
Gcc lex.yy.c
```

**After running the above command, you will get a executable file a.out you can run it using this command**

```
./a.out
```

**To write regular expression in our lex file here are important operators used commonly in regular expressions,**

| Character | Matches |
|---|---|
| . | Any character except newline |
| \n | Newline |
| \t | Tab-space |
| + | One or more copies of preceding expression |
| * | Zero or more copies of preceding expression |
| ? | Zero or one copy of preceding expression |
| ^ | Starts with |
| a\|b | a or b |
| "ab" | String literal, Exact match for ab |
| [a-z] | Character class, Any character between a and z |
| (ab) | Grouping, |
| {id} | Substitute |

**Substitutes for some regular expressions,**

| {id} | Substitute regular expression | |
|---|---|---|
| alphabet | [a-zA-Z] | Any one alphabet in **a** to **z** or **A** to **Z** |
| digit | [0-9] | Any one digit between 0 and 9 |
| underscore | _ | A single underscore |
| whitespace | [ \t\r\f\v]+ | One or more whitespaces |

**Let's define all the reserved keywords used in our language first, which will be directly passed to the parser without any change. They are given in the following table. The strings can be directly matched.**

| **Pattern** | **Token** |
|---|---|
| int | INT DATATYPE |
| float | FLOAT DATATYPE |
| char | CHAR DATATYPE |
| main | MAIN |
| printf | PRINTF KEYWORD |
| scanf | SCANF KEYWORD |
| switch | SWITCH STATEMENT |
| return | RETURN STATEMENT |
| case | CASE STATEMENT |
| default | DEFAULT STATEMENT |

**Different types of input and output formats**

| **Pattern** | **Token** | **EXPLAINATION** |
|---|---|---|
| \"%d\" | INT_FORMAT | %d enclosed by two double quotes. |
| \"%f\" | FLOAT_FORMAT | %f enclosed by two double quotes. |
| \"%s\" | STRING_FORMAT | %s enclosed by two double quotes. |
| \".*\" | STRING_LITERAL | Any expression between two double quotes. |

Now let's define the patterns for different arithmetic, logical and comparison operator

| pattern | Token | Explaination |
|---|---|---|
| \+<br>\-<br>\/<br>*<br>\^<br>\% | PLUS<br>MINUS<br>DIV<br>MULT<br>POW<br>MOD | PLUS operator, the symbol usually used as repeater in regular expressions, hence preceded by escape '\' ,other **arithmetic** operators could be dealt in the same way to avoid operator misunderstanding. |
| "--"<br>"++" | DECREMENT<br>INCREMENT | Unary operators will be just matched a string itself. |
| "<"<br>">"<br>">="<br>"<="<br>"=="<br>"!=" | LT<br>GT<br>GT_EQ<br>LT_EQ<br>EQUAL<br>NOT_EQUAL | Comparison operators will be matched as a string literal. |

**A1.1**

In Lex we have a facility called start conditions or states, these types are useful when we have to match patterns depending upon that particular condition. i.e., It acts like a flag. We will explain that using an example in our lex file. BEGIN is a keyword which lets us switch between states, the state where no conditions are active is called INITIAL. BEGIN activates a STATE

**Deterministic Finite Automata**

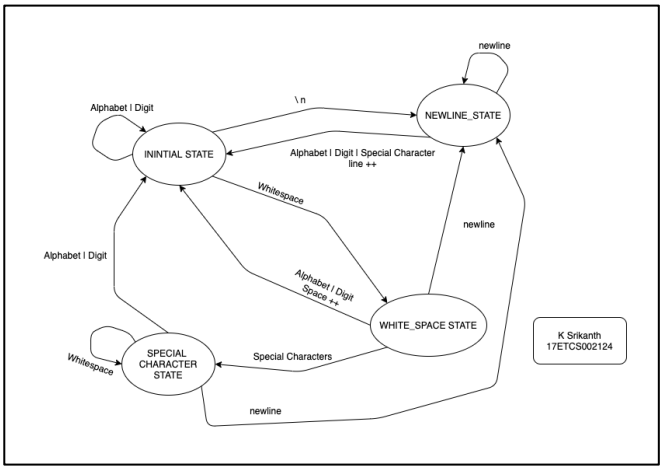


Figure 2 Deterministic Finite Automata for the given problem statement

**SYNTAX FOR A LEX PROGRAM**

```
%{
//1.declaration section to declare headers and user defined function.
%}
%%
//2. regular expressions with their actions
%%
//3. main()
```

Name: K Srikanth Registration Number: 17ETCS002124

**LEX PROGRAM**

```
%{
    int yylex();
    void yyerror(char *s);
    int line=0;
    int spaces=0;
%}
alphabet [a-zA-Z]
number [0-9]
newline \n
whitespace [ \t]
special_character [\!\?\.\[\]\(\)\,]
%x newline_state
%x spaces_state
%x special_state
%%
<<EOF>> {printf("\nThe Number of Spaces are %d.\n",spaces);printf("The Number of lines
are %d.\n",line);exit(0);}
{alphabet}|{number} {printf("%s",yytext);}
{newline} {BEGIN newline_state;}
{whitespace} {BEGIN spaces_state;}
{special_character} {BEGIN special_state;printf("%s",yytext);}

<newline_state>{alphabet}|{number} {BEGIN INITIAL;line++;printf("\n%s",yytext);}
<newline_state>{newline} {BEGIN newline_state;}
<newline_state>{special_character} {BEGIN special_state;printf("\n%s",yytext);}
<newline_state>{whitespace} ;

<spaces_state>{alphabet}|{number} {BEGIN INITIAL;spaces++;printf(" %s",yytext);}
<spaces_state>{newline} {BEGIN newline_state;}
<spaces_state>{special_character} {BEGIN special_state;printf("%s",yytext);}
<spaces_state>{whitespace} {BEGIN spaces_state;}

<special_state>{alphabet}|{number} {BEGIN INITIAL;printf("%s",yytext);}
<special_state>{newline} {BEGIN newline_state;}
<special_state>{special_character} {BEGIN special_state;printf("%s",yytext);}
<special_state>{whitespace} {BEGIN special_state;}
%%
int yywrap(){ return 1;}
void yyerror (char *s) {fprintf (stderr, "%s at line %d\n", s, yylineno);}
int main()
{
 yyin = fopen("Input.txt", "r");
 if(yyin==NULL) printf("\nError\n");
 else{
 printf("\Started Lexing\n"); printf("17ETCS002124 K Srikanth\n");yylex();} //start
lexing
fclose(yyin);
return 0;
```

Name: K Srikanth Registration Number: 17ETCS002124

**Declaration of all the headers and user defined variables**

```
int yylex();
    void
yyerror(char *s);
    int line=0;
    int spaces=0;
```

**Declaration of States and regular expressions**

```
alphabet [a-zA-Z]
number [0-9]
newline \n
whitespace [ \t]
special_character [\!\?\.\[\]\(\)\,]
%x newline_state
%x spaces_state
%x special_state
```

**Here we have three states**

**1. Newline_state:** Logic for Newline state whenever a new line occurs go to this state
**2. Space_state:** Logic for Space_state whenever a new line occurs go to this state
**3. Special_state:** Logic for Special_state whenever a new line occurs go to this state

```
<<EOF>> {printf("\nThe Number of Spaces are
%d.\n",spaces);printf("The Number of lines are
%d.\n",line);exit(0);}
{alphabet}|{number} {printf("%s",yytext);}
{newline} {BEGIN newline_state;}
{whitespace} {BEGIN spaces_state;}
{special_character} {BEGIN
special_state;printf("%s",yytext);}
```

- <<EOF>> aka end of the file means that after reading all the input from a text file print the given statements
- {alphabet}|{number} whenever you occur this alphabet or a number print it using yytext
- Initializing the states
    - {newline} when a new line occurs begin newline state
    - {whitespace} when a whitespace occurs begin spaces_state
    - {special_character} when a special character occurs begin special_state and print the string using yytext

**New Line State**

```
<newline_state>{alphabet}|{number} {BEGIN
INITIAL;line++;printf("\n%s",yytext);}
<newline_state>{newline} {BEGIN newline_state;}
<newline_state>{special_character} {BEGIN
special_state;printf("\n%s",yytext);}
<newline_state>{whitespace} ;
```

- **Whenever this state encounters a** {**alphabet}|{number}** begin the initial state and increment the line and print the string using yytext
- **Whenever this state encounters a {newline}** begin newline_state
- **Whenever this state encounters a {special_character}** begin special_state and print the string using yytext
- **Whenever this state encounters a {whitespace}** ignore it

**Space State**

```
<spaces_state>{alphabet}|{number} {BEGIN
INITIAL;spaces++;printf(" %s",yytext);}
<spaces_state>{newline} {BEGIN newline_state;}
<spaces_state>{special_character} {BEGIN
special_state;printf("%s",yytext);}
<spaces_state>{whitespace} {BEGIN spaces_state;}
```

- **Whenever this state encounters a** {**alphabet}|{number}** begin the initial state and increment the spaces and print the string using yytext
- **Whenever this state encounters a {newline}** begin newline_state
- **Whenever this state encounters a {special_character}** begin special_state and print the string using yytext
- **Whenever this state encounters a {whitespace}** begin spaces_state

**Special State**

```
<special_state>{alphabet}|{number} {BEGIN
INITIAL;printf("%s",yytext);}
<special_state>{newline} {BEGIN newline_state;}
<special_state>{special_character} {BEGIN
special_state;printf("%s",yytext);}
<special_state>{whitespace} {BEGIN special_state;}
```

- **Whenever this state encounters a** {**alphabet}|{number}** begin the initial state and increment the spaces and print the string using yytext
- **Whenever this state encounters a {newline}** begin newline_state
- **Whenever this state encounters a {special_character}** begin special_state and print the string using yytext
- **Whenever this state encounters a {whitespace}** begin special_state

```
int yywrap(){ return 1;}
void yyerror (char *s) {fprintf (stderr, "%s at line %d\n",
s, yylineno);}
```

Now we define a function yywrap () which notifies lex when it reaches end of file and Define a function yyerror(*s) to show the error and the line it occurred on.

**Main function**

```
yyin = fopen("Input.txt", "r");
 if(yyin==NULL) printf("\nError\n");
 else{
 printf("\Started Lexing\n"); printf("17ETCS002124 K
Srikanth\n");yylex();}
fclose(yyin);
```

The function yylex() which reads the input stream and do corresponding actions that could be printing statements or returning tokens.
The yyin input stream pointer points to an input file which is to be scanned, but in default points to standard input buffer. (stdin i.e. user input)

**A1.2**
**Results**
**Text File**

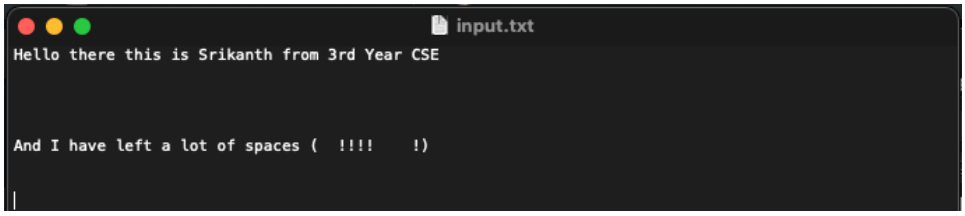


Figure 3 Input text file for Lex compilation question 1

**Link for my code**
Lex File
Input text file

**To Compile Lex File**

```
Last login: Sat Dec  5 08:15:36 on ttys001
❯ cd desktop
❯ touch input.txt
❯ flex q_1.l
❯ pwd
/Users/srikanth/desktop
❯ gcc lex.yy.c
q_1.l:45:10: warning: unknown escape sequence '\S' [-Wunknown-escape-sequence]
 printf("\Started Lexing\n"); printf("17ETCS002124 K Srikanth\n");yylex(...
         ^~
1 warning generated.
❯ ./a.out
Started Lexing
17ETCS002124 K Srikanth
Hello there this is Srikanth from 3rd Year CSE
And I have left a lot of spaces(!!!!!)
The Number of Spaces are 15.
The Number of lines are 1.
```

Figure 4 Lex Output for the given input text file (Figure 3)

**A2.1**

**Declaration of all the headers and user defined variables**

```
int yylex();
    void yyerror(char *s);
```

**Declaration of States and regular expressions**

```
alphabet [a-zA-Z]
number [0-9]
newline \n
whitespace [ \t]
special_character [\!\?\.\[\]\(\)\,]
%x PREPROCESSING
%x MULTILINECOMMENT
%x SINGLELINECOMMENT
```

**Here we have three states**

**1. PREPROCESSING:** Logic for preprocessing state whenever a preprocessor statement occurs go to this state

**2. MULTILINECOMMENT:** Logic for multilinecomment whenever a multiline comment occurs go to this state

**3. SINGLELINECOMMENT:** Logic for singlelinecomment whenever a single line comment occurs go to this state

**PREPROCESSING State**

```
<<EOF>> {exit(0);}
^"#include" {BEGIN PREPROCESSING; printf("%10s
PREPROCESSING\n",yytext); }
<PREPROCESSING>{whitespace} ;
<PREPROCESSING>"<"[^<>\n]*">" {BEGIN INITIAL;}
<PREPROCESSING>\"[^<>\n]*\" {BEGIN INITIAL;}
<PREPROCESSING>"\n" {yylineno++; BEGIN INITIAL;}
<PREPROCESSING>. {yyerror("Mistake in Header");}
```

- Here our condition or flag is PREPROCESSING, this flag is initially off or so-called INITIAL state, when it encounters #include at the begin of the lexeme, BEGIN activates the flag and conditional execution of regular expression begins. We print Header preprocessing on activation.
- Second expression says, skip whitespaces if it encounters any, in the PREPROCESSING state.
- Third expression says, go back to INITIAL state when it encounters anything (except <, >, newline), if that thing is enclosed between “<” and “>”.
- Fourth expression says, go back to INITIAL state when it encounters anything (except <, >, newline), if that thing is enclosed within a pair of double quotes.
- Fifth expression says, go back to INITIAL state when it notices a newline in PREPROCESSING state.
- If nothing above matches the input stream in this state, throw an error as it is considered as an invalid expression.

**MULTILINECOMMENT State**

```
"/*" {BEGIN MULTILINECOMMENT; printf("%10s
MULTILINECOMMENT\n",yytext); }
<MULTILINECOMMENT>.|{whitespace} ;
<MULTILINECOMMENT>\n {yylineno++;}
<MULTILINECOMMENT>"*/" {BEGIN INITIAL;}
<MULTILINECOMMENT>"/*" {yyerror("Comment format invalid");}
```

- Activate the condition multilinecomment when /* is found
- When a whitespace is encountered skip it.
- When a new line is encountered increment the line number.
- When */ go back to INITIAL state.
- Throw an error when invalid expression /* is found, as it causes ambiguity in closing comment.

**SINGLELINECOMMENT State**

```
"//" {BEGIN SINGLELINECOMMENT; printf("%10s
SINGLELINECOMMENT\n",yytext); }
<SINGLELINECOMMENT>\n {yylineno++; BEGIN INITIAL;}
<SINGLELINECOMMENT>. ;
```

- Activate the condition singlelinecomment when // is found
- Go back to INITIAL state on finding newline character.
- Skip everything else during the active condition.

**For all the Expressions and identifiers**

A **valid identifier** should always start with an alphabet. A **reserved identifier** is the one that starts with an underscore. But an identifier can never start with a digit and cannot contain any special character other than underscore.
{underscore} is a substitute for _
{alphabet} represents any upper or lower case alphabet
{digit} represents any digit between 0 and 9
An identifier should start with {underscore} or {alphabet} → ({underscore}|{alphabet}) Can be followed by any number of {underscore} or {alphabet} or {digit}
→ ({underscore}|{alphabet}|{digit})*
**IDENTIFIER: ({underscore}|{alphabet}) ({underscore}|{alphabet}|{digit})***
**Expression for an Integer value:** print INT VALUE
An integer value contains 1 or more **digits** → **{digit}+**
**Expression for a Float value:** print FLOAT VALUE
An float value contains 1 or more digits before and after a decimal point → **{digit}+[\.]{digit}+**
**Expression for a char literal:** print CHAR_LITERAL
A single character enclosed in a pair of single or pair of double quotes
→ **(\"{alphabet}\"|'{alphabet}')**
**Expression for separators: print SEPARATOR**
**{ } [ ] ( ) ; ,** are the most commonly seen separators in the C programs. → **[\{\}\[\]\(\)\;\,]**

```
int yywrap(){ return 1;}
void yyerror (char *s) {fprintf (stderr, "%s at line %d\n",
s, yylineno);}
```

Now we define a function yywrap () which notifies lex when it reaches end of file and Define a function yyerror(*s) to show the error and the line it occurred on.

**Main function**

```
yyin = fopen("Input.c", "r");
 if(yyin==NULL) printf("\nError\n");
 else{
 printf("\Started Tokenizing\n"); printf("17ETCS002124 K
Srikanth\n");yylex();}
fclose(yyin);
```

The function yylex() which reads the input stream and do corresponding actions that could be printing statements or returning tokens.
The yyin input stream pointer points to an input file which is to be scanned, but in default points to standard input buffer. (stdin i.e., user input)

**A2.2**
**Results**

**Link for my code**
Lex File
Input C file

**C File**

```c
#include <stdio.h>
int main()
{
    int num1,num2;
    float result;
    char ch;    //to store operator choice
    printf("Enter first number: ");
    scanf("%d",&num1);
    printf("Enter second number: ");
    scanf("%d",&num2);
    printf("Choose operation to perform (+,-,*,/,%): ");
    scanf(" %c",&ch);
    result=0;
    switch(ch)
    {
        case '+':
            result=num1+num2;
            break;
        case '-':
            result=num1-num2;
            break;
        case '*':
            result=num1*num2;
            break;
        case '/':
            result=(float)num1/(float)num2;
            break;
        case '%':
            result=num1%num2;
            break;
        default:
            printf("Invalid operation.\n");
    }

    printf("Result: %d %c %d = %f\n",num1,ch,num2,result);
    return 0;
}

```

*Figure 5 C file input to be tokenized using LEX*

**To Compile Lex File**

```
❯ flex q_2.l
❯ gcc lex.yy.c
q_2.l:84:13: warning: unknown escape sequence '\S' [-Wunknown-escape-sequence]
    printf("\Started Tokenizing\n"); printf("17ETCS002124 K Srikanth\n");yylex();}
            ^~
1 warning generated.
❯ ./a.out
Started Tokenizing
17ETCS002124 K Srikanth
  #include PREPROCESSING
       int INT DATATYPE
      main MAIN
         ( SEPARATOR
         ) SEPARATOR
         { SEPARATOR
        int INT DATATYPE
          n CHAR_LITERAL
          u CHAR_LITERAL
          m CHAR_LITERAL
          1 INT VALUE
          , SEPARATOR
          n CHAR_LITERAL
          u CHAR_LITERAL
          m CHAR_LITERAL
          2 INT VALUE
          ; SEMICOLON
      float FLOAT DATATYPE
          r CHAR_LITERAL
          e CHAR_LITERAL
          s CHAR_LITERAL
          u CHAR_LITERAL
          l CHAR_LITERAL
          t CHAR_LITERAL
          ; SEMICOLON
       char CHAR DATATYPE
          c CHAR_LITERAL
          h CHAR_LITERAL
          ; SEMICOLON
         // SINGLELINECOMMENT
     printf PRINTF KEYWORD
          ( SEPARATOR
          " CHAR_LITERAL
          E CHAR_LITERAL
          n CHAR_LITERAL
          t CHAR_LITERAL
          e CHAR_LITERAL
          r CHAR_LITERAL
          f CHAR_LITERAL
          i CHAR_LITERAL
          r CHAR_LITERAL
          s CHAR_LITERAL
          t CHAR_LITERAL
          n CHAR_LITERAL
          u CHAR_LITERAL
          m CHAR_LITERAL
          b CHAR_LITERAL
          e CHAR_LITERAL
          r CHAR_LITERAL
          : CHAR_LITERAL
          " CHAR_LITERAL
          ) SEPARATOR
          ; SEMICOLON
```

*Figure 6 Lex Token generation for input C (Figure 5)*

```
      scanf SCANF KEYWORD
          ( SEPARATOR
          " CHAR_LITERAL
          % MOD ARITHMETIC OPERATOR
          d CHAR_LITERAL
          " CHAR_LITERAL
          , SEPARATOR
          & CHAR_LITERAL
          n CHAR_LITERAL
          u CHAR_LITERAL
          m CHAR_LITERAL
          1 INT VALUE
          ) SEPARATOR
          ; SEMICOLON
     printf PRINTF KEYWORD
          ( SEPARATOR
          " CHAR_LITERAL
          E CHAR_LITERAL
          n CHAR_LITERAL
          t CHAR_LITERAL
          e CHAR_LITERAL
          r CHAR_LITERAL
          s CHAR_LITERAL
          e CHAR_LITERAL
          c CHAR_LITERAL
          o CHAR_LITERAL
          n CHAR_LITERAL
          d CHAR_LITERAL
          n CHAR_LITERAL
          u CHAR_LITERAL
          m CHAR_LITERAL
          b CHAR_LITERAL
          e CHAR_LITERAL
          r CHAR_LITERAL
          : CHAR_LITERAL
          " CHAR_LITERAL
          ) SEPARATOR
          ; SEMICOLON
      scanf SCANF KEYWORD
          ( SEPARATOR
          " CHAR_LITERAL
          % MOD ARITHMETIC OPERATOR
          d CHAR_LITERAL
          " CHAR_LITERAL
          , SEPARATOR
          & CHAR_LITERAL
          n CHAR_LITERAL
          u CHAR_LITERAL
          m CHAR_LITERAL
          2 INT VALUE
          ) SEPARATOR
          ; SEMICOLON
```

*Figure 7 Lex Token generation for input C (Figure 5) continued*

```
      ; SEMICOLON
switch SWITCH STATEMENT
      ( SEPARATOR
      c CHAR_LITERAL
      h CHAR_LITERAL
      ) SEPARATOR
      { SEPARATOR
 case CASE STATEMENT
      ' CHAR_LITERAL
      + PLUS
      ' CHAR_LITERAL
      : CHAR_LITERAL
      r CHAR_LITERAL
      e CHAR_LITERAL
      s CHAR_LITERAL
      u CHAR_LITERAL
      l CHAR_LITERAL
      t CHAR_LITERAL
      = CHAR_LITERAL
      n CHAR_LITERAL
      u CHAR_LITERAL
      m CHAR_LITERAL
      1 INT VALUE
      + PLUS
      n CHAR_LITERAL
      u CHAR_LITERAL
      m CHAR_LITERAL
      2 INT VALUE
      ; SEMICOLON
break BREAK STATEMENT
      ; SEMICOLON
 case CASE STATEMENT
      ' CHAR_LITERAL
      - MINUS
      ' CHAR_LITERAL
      : CHAR_LITERAL
      r CHAR_LITERAL
      e CHAR_LITERAL
      s CHAR_LITERAL
      u CHAR_LITERAL
      l CHAR_LITERAL
      t CHAR_LITERAL
      = CHAR_LITERAL
      n CHAR_LITERAL
      u CHAR_LITERAL
      m CHAR_LITERAL
      1 INT VALUE
      - MINUS
      n CHAR_LITERAL
      u CHAR_LITERAL
      m CHAR_LITERAL
      2 INT VALUE
      ; SEMICOLON
break BREAK STATEMENT
      ; SEMICOLON
 case CASE STATEMENT
      ' CHAR_LITERAL
      * MULT
      ' CHAR_LITERAL
      : CHAR_LITERAL
      r CHAR_LITERAL
      e CHAR_LITERAL
```

*Figure 8 Lex Token generation for input C (Figure 5) continued*

```
      } SEPARATOR
printf PRINTF KEYWORD
      ( SEPARATOR
      " CHAR_LITERAL
      R CHAR_LITERAL
      e CHAR_LITERAL
      s CHAR_LITERAL
      u CHAR_LITERAL
      l CHAR_LITERAL
      t CHAR_LITERAL
      : CHAR_LITERAL
      % MOD ARITHMETIC OPERATOR
      d CHAR_LITERAL
      % MOD ARITHMETIC OPERATOR
      c CHAR_LITERAL
      % MOD ARITHMETIC OPERATOR
      d CHAR_LITERAL
      = CHAR_LITERAL
      % MOD ARITHMETIC OPERATOR
      f CHAR_LITERAL
      \ CHAR_LITERAL
      n CHAR_LITERAL
      " CHAR_LITERAL
      , SEPARATOR
      n CHAR_LITERAL
      u CHAR_LITERAL
      m CHAR_LITERAL
      1 INT VALUE
      , SEPARATOR
      c CHAR_LITERAL
      h CHAR_LITERAL
      , SEPARATOR
      n CHAR_LITERAL
      u CHAR_LITERAL
      m CHAR_LITERAL
      2 INT VALUE
      , SEPARATOR
      r CHAR_LITERAL
      e CHAR_LITERAL
      s CHAR_LITERAL
      u CHAR_LITERAL
      l CHAR_LITERAL
      t CHAR_LITERAL
      ) SEPARATOR
      ; SEMICOLON
return RETURN TYPE
      0 INT VALUE
      ; SEMICOLON
      } SEPARATOR
```

*Figure 9 Lex Token generation for input C (Figure 5) continued*