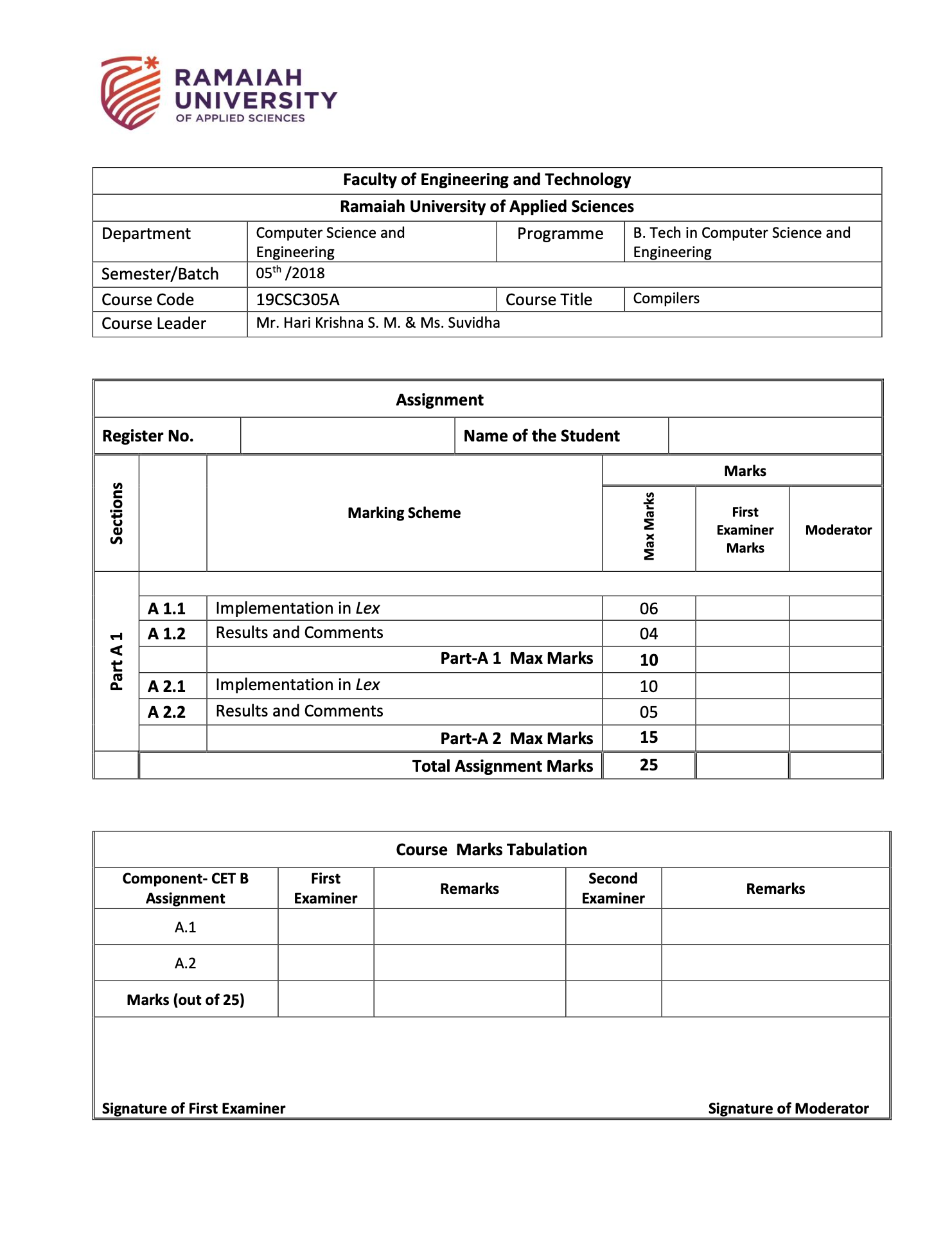


|  |  |
| --- | --- |
| **ASSIGNMENT - 1** | |
| **Course Code** | 19CSC305A |
| **Course Name** | Compilers |
| **Programme** | B. Tech |
| **Department** | CSE |
| **Faculty** | FET |

#### 

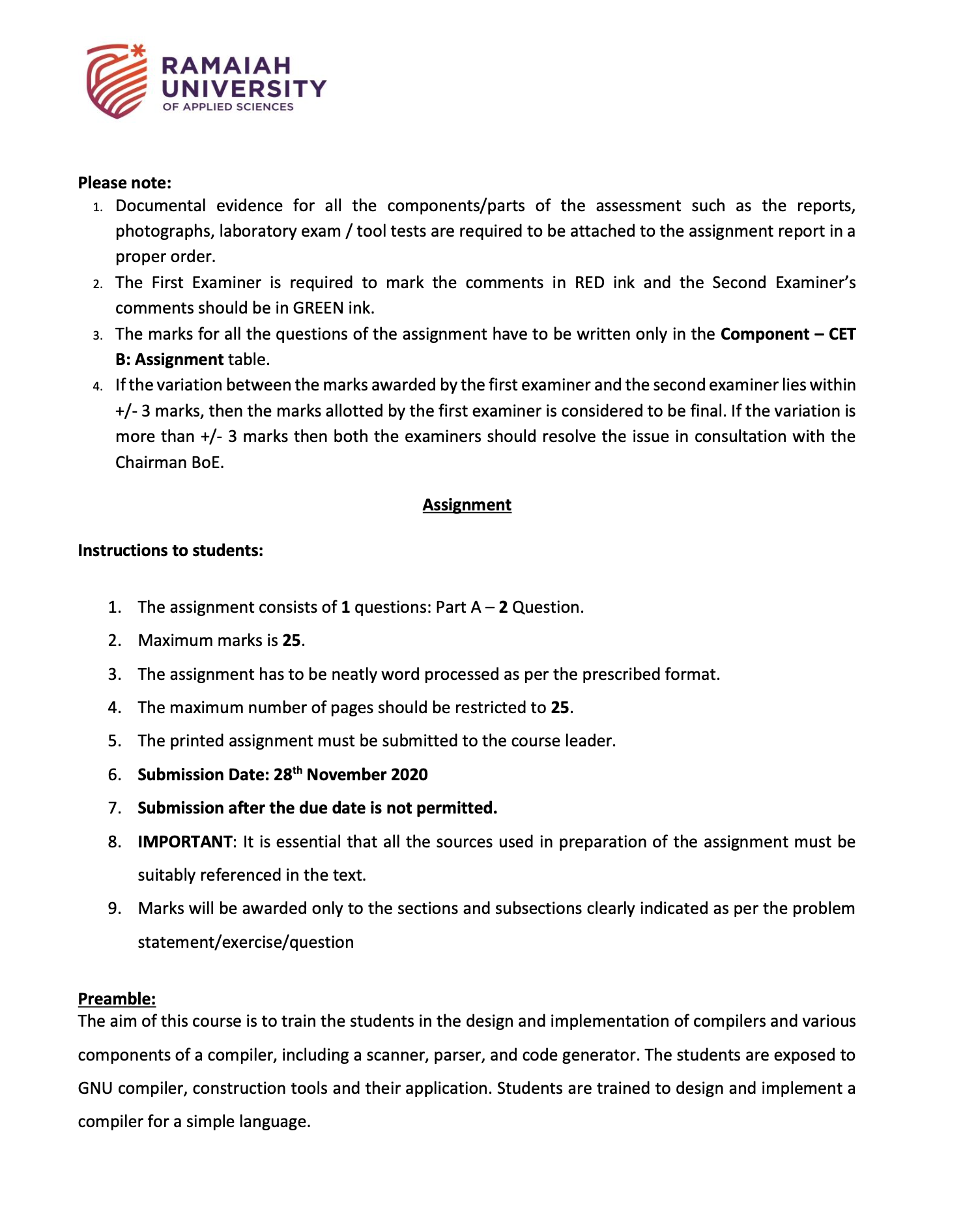
|  |  |
| --- | --- |
| **Name of the Student** | K Srikanth |
| **Reg. No** | 17ETCS002124 |
| **Semester/Year** | 5th Semester/ 3rd Year |
| **Course Leader/s** | Mr. Hari Krishna S. M. |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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 | | | | |
|  | | | |  | | | | |



17ETCS002124

K Srikanth



**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.

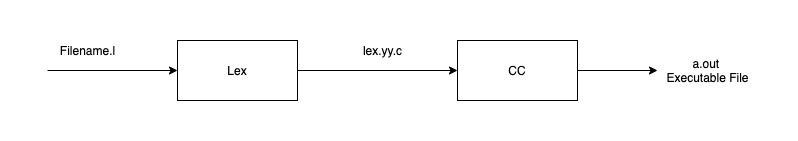
****

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*

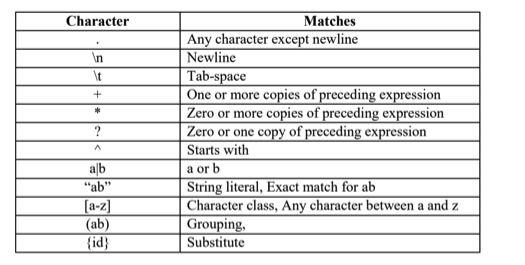
**After running the about command without errors you will get a lex.yy.c file and compile the produced c file using this command**

*Gcc lex.yy.c*

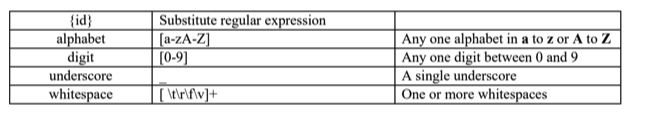
**After running the above command, you will get a executable file a.out you can run it using this command**

*./a.out*

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



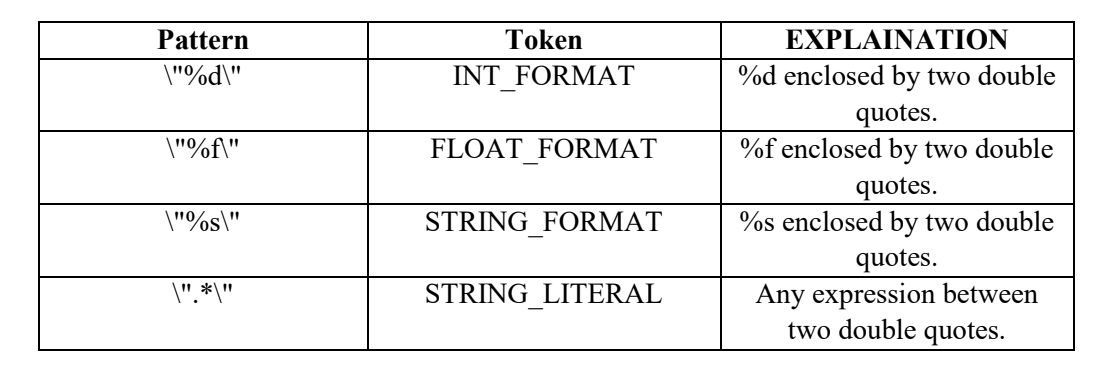
**Substitutes for some regular expressions,**



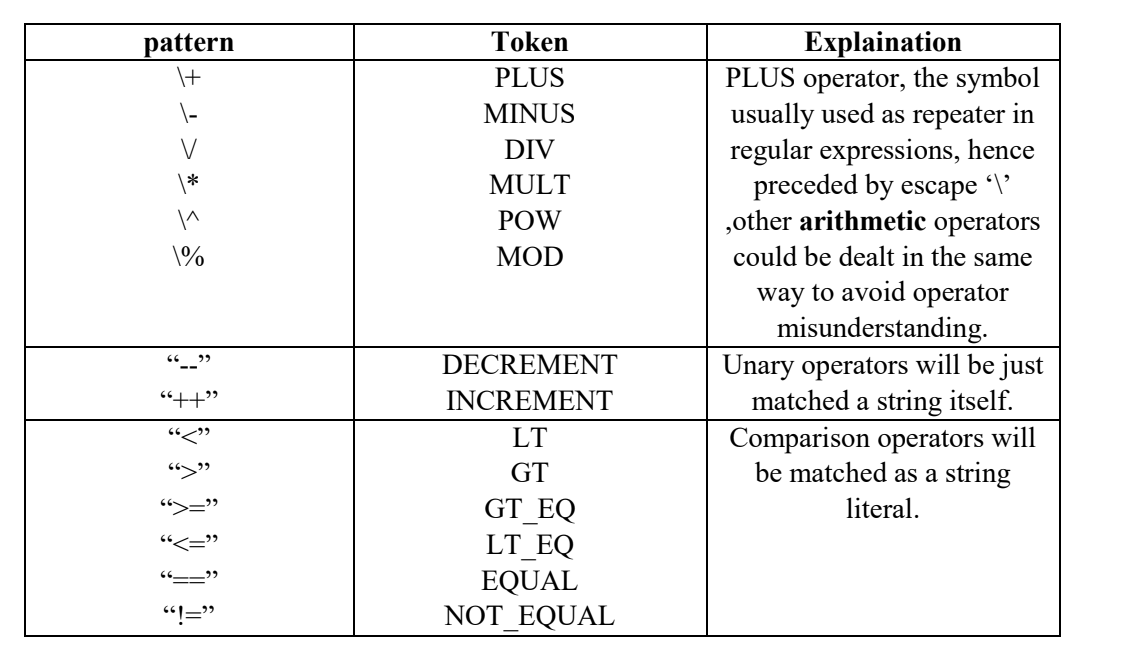
**Let’s define all the reserved keywords used in our language first, which will be directly passed to the parser without any change. They are given in the following table. The strings can be directly matched.**

|  |  |
| --- | --- |
| **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**



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



**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**

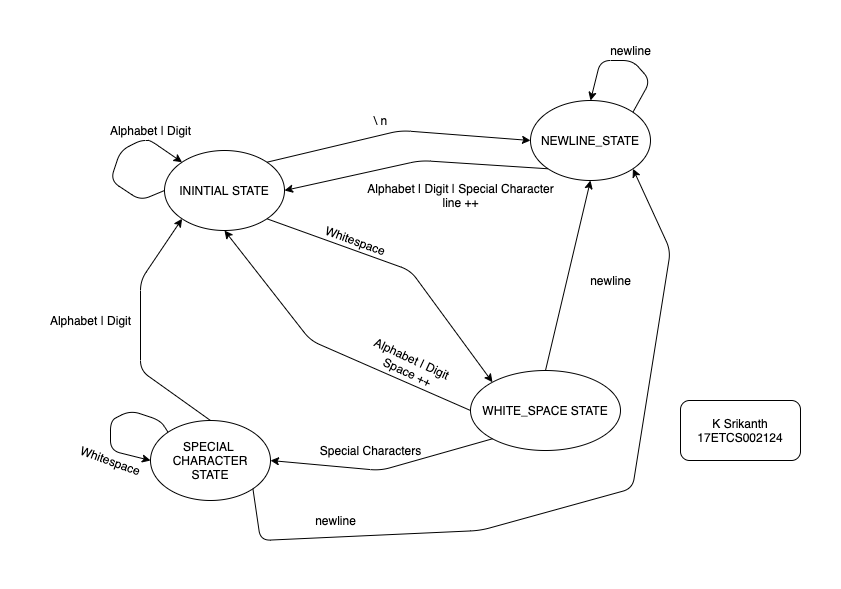
****

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()

**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;

}

**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**

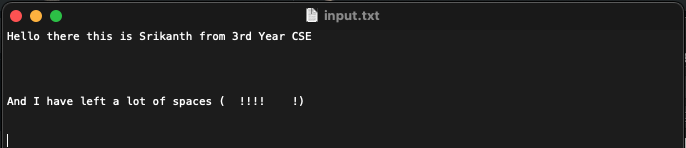
****

Figure 3 Input text file for Lex compilation question 1

**Link for my code**

[Lex File](https://github.com/Srikanth-Kandarp/Labs-Code-5th-Sem/blob/main/Compilers/Codes/assignment.l)

[Input text file](https://github.com/Srikanth-Kandarp/Labs-Code-5th-Sem/blob/main/Compilers/Codes/input.txt)

**To Compile Lex File**

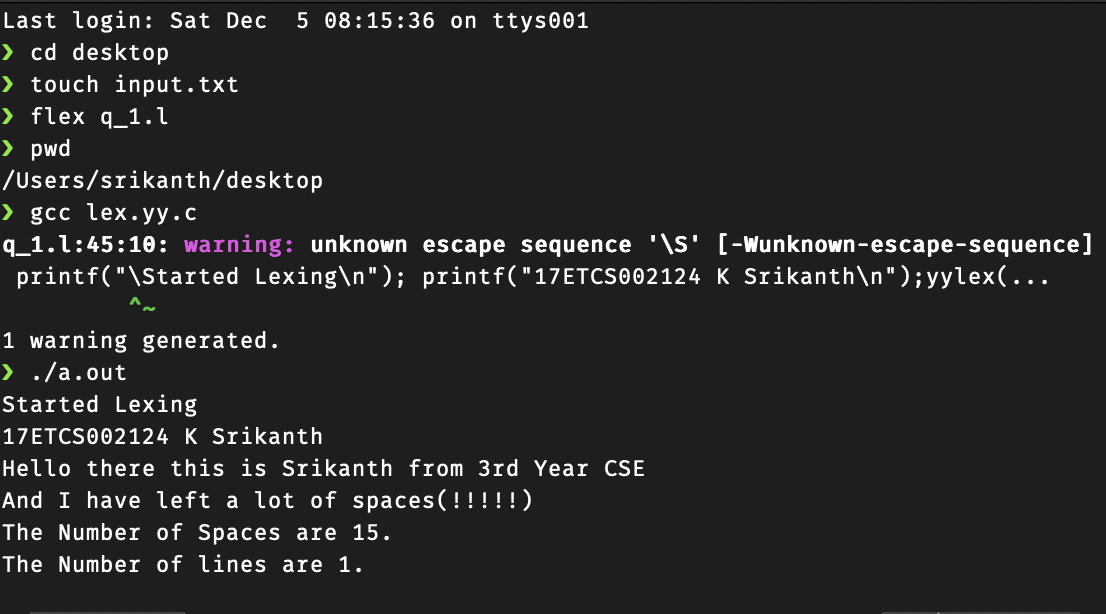
****

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**

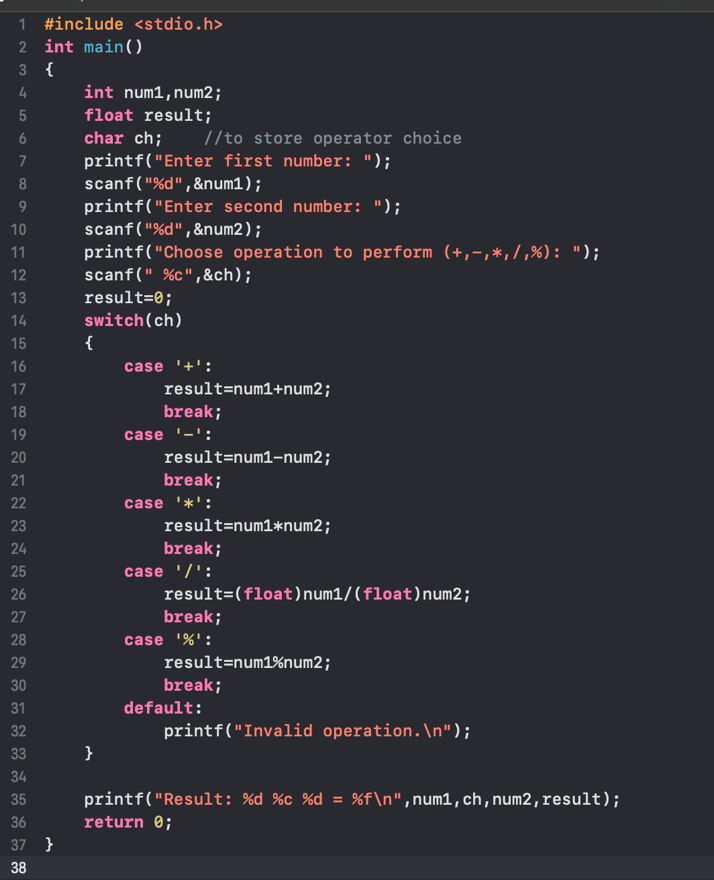
**Results**

**Link for my code**

[Lex File](https://github.com/Srikanth-Kandarp/Labs-Code-5th-Sem/blob/main/Compilers/Codes/assignmentq2.l)

[Input C file](https://github.com/Srikanth-Kandarp/Labs-Code-5th-Sem/blob/main/Compilers/Codes/input.c)

**C File**

****

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

**To Compile Lex File**

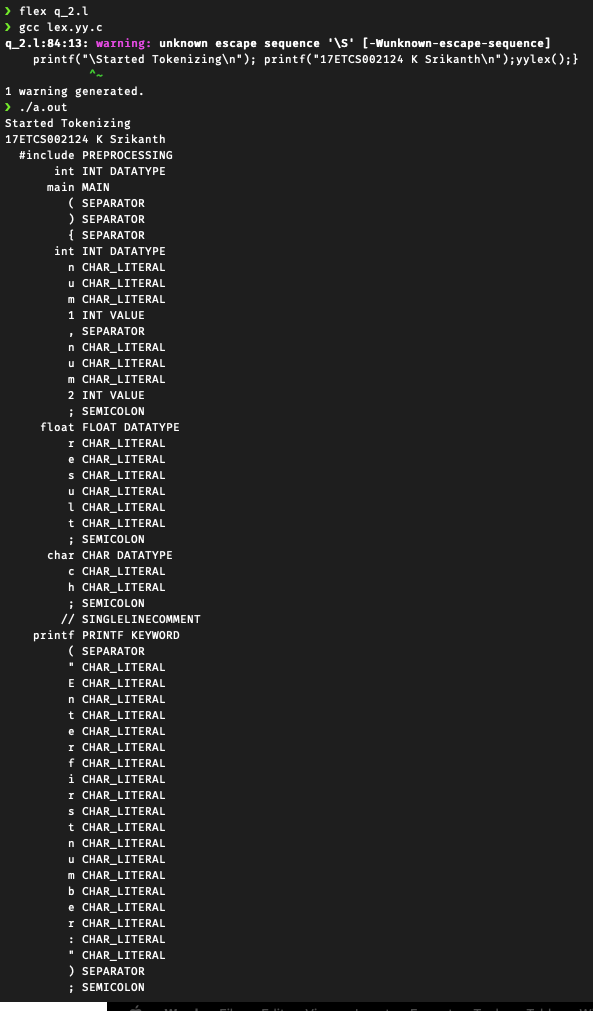
****

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

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

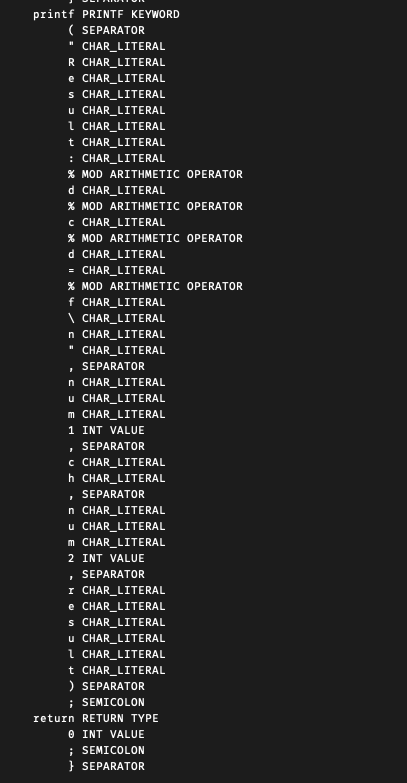
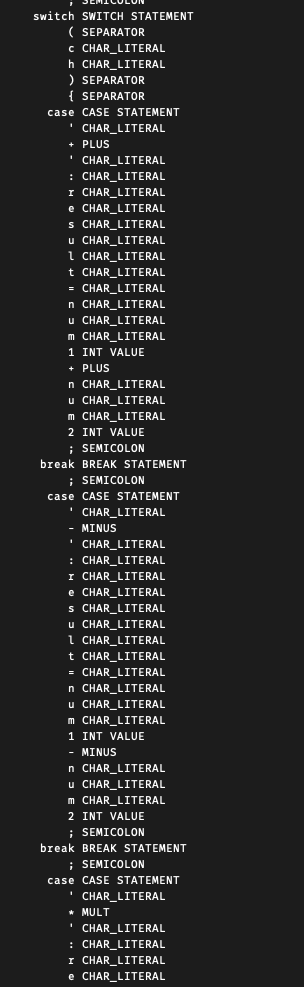
****

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

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