Lexical Analyser
(C language)

Team Members –

1. Bharadwaja M Chittapragada (211CS216)
2. Rishi Diwaker (211CS243)
3. Rohit Kumar (211CS244)

An abstract of the work:

A compiler is a vital software tool in the field of programming, transforming human-readable source code into machine-executable instructions. This indispensable component is integrated into most programming software packages, streamlining the development process. The compiler undertakes the task of translating high-level programming languages like C, C++, or Java into low-level counterparts, such as machine code or assembly code, tailored to specific processor architectures like Intel Pentium or PowerPC. This transformation enables computers to understand and execute the program efficiently.

The compilation process comprises several interconnected phases, each with its unique role and output format, forming a seamless pipeline:

1. Lexical Analysis: This phase parses the source code to identify tokens, such as keywords, identifiers, and operators.

2. Syntax Analysis: It checks the syntactic correctness of the code by analysing the arrangement of tokens and their adherence to language grammar.

3. Semantic Analysis: This phase focuses on the meaning of code constructs, ensuring they are semantically valid and meaningful.

4. Intermediate Code Generation: An intermediate representation of the code is generated, abstracting away from specific machine details.

5. Code Optimization: The compiler optimizes the code for efficiency, applying various techniques to improve performance.

6. Code Generation: The final machine code or assembly code is generated, tailored to the target processor.

In this project, we implement the lexical analysis phase of a C compiler, incorporating various functionalities:

1. Data Types: Recognizing data types like int and char and labels them as Keywords

2. Comments: Handling both single-line and multiline comments in the code.

3. Keywords: Identifying keywords intrinsic to the C language.

4. Identifiers: Detecting valid user-defined identifiers.

5. Looping Constructs: Supporting nested for and while loops.

6. Conditional Constructs: Recognizing if...else-if...else statements.

7. Operators: Handling arithmetic and logical operators like +, *, /, %, &, and |.

8. Delimiters: Identifying delimiters like ; and ,.

9. Structures: Detecting the structure construct in the code.

10. Functions: Recognizing function constructs along with their parameters.

11. Nested Conditionals: Supporting nested conditional statements.

12. Arrays: Handling 1-Dimensional arrays as per the language syntax.

This project represents an essential step in understanding and implementing the core functionality of a C compiler, providing a solid foundation for building more advanced compiler components.

An overview :

The provided code is an implementation of a lexical analyzer for the C programming language using the Flex (Fast Lexical Analyzer Generator) tool. The primary purpose of this code is to tokenize a C source code file and categorize each token into different categories, such as keywords, operators, identifiers, constants, and more. Additionally, it populates two symbol tables: one for symbols like identifiers and operators and another for constants like integers, floating-point numbers, strings, and character constants.

The code is structured as follows:

1. Header Section (%{...%}): This section includes header files and defines a hash function. It also defines structures for the symbol table and constant table, along with functions for looking up and inserting entries into these tables. The hash table size is set to 1009, which is a prime number.

2. Main Section: This section contains the main Flex definitions. It consists of regular expressions and corresponding actions for tokenizing and categorizing different parts of the input code.

3. C Code: This section contains C code snippets that are executed when specific regular expressions are matched. These snippets include printing token information and inserting tokens into the symbol and constant tables. If an error is encountered, an error message is printed.

4. Main Function (int main()): The main function initializes the symbol and constant tables, sets up input from a file provided as a command-line argument, and invokes the lexical analyzer (yylex()). After analysis, it prints the contents of the symbol and constant tables.

5. yywrap() Function: A function called when Flex reaches the end of the input file.

What Works:

- The code successfully tokenizes C source code, categorizing tokens into various categories such as keywords, operators, identifiers, constants (including integers, floating-point numbers, character constants, and strings), and more.

- It populates two tables: the Symbol Table (for identifiers, operators, etc.) and the Constant Table (for constants).

- Error handling is included for incomplete strings, preprocessor directives, unmatched comments, and general lexing errors.

- In case of collisions in hash function, a linked list approach is used to store and look-up in the symbol and constant table.

What Doesn't Work:

- The code may have some issues or limitations:

  - The code assumes that the input source code is correct; it does not perform full syntactic or semantic analysis, which would typically be done by a compiler after lexical analysis.

  - The code does not identify an error in header files. It is assumed that the code is preprocessed.

  - The code does not identify hierarchical scope errors.

Overall, this code serves as a basic example of how to build a lexical analyser for a C-like language, but it may require further development and testing to handle more complex scenarios and provide more informative error messages.

Code for LexicalAnalyser.l

```
%{
/* Including the required header files */
#include <stdio.h>
#include <string.h>
#include <limits.h>


/* Defining Hash function */


// For now the basic hash function is defined which can be improved
int hash_function(char *tokenName)
```

```
%{ /* Including the required header files */ #include <stdio.h> #include <string.h> #include <limits.h>
/* Defining Hash function */ // For now the basic hash function is defined which can be improved int
hash_function(char *tokenName)

/* We first look_up at the symbol table to check whether it exists ,

if it does we do not insert again

If value is not found then, we insert using linear Probing first

and if the symbol Table is full then we use chaining to insert

*/

if(look_up_symbol_table(tokenName))

{

return;

}

else

{

int hashValue = hash_function(tokenName);

if(SymbolTable[hashValue].length == 0)

{

insertSymbol(SymbolTable, hashValue, tokenName, tokenType);

return;

}


for (int i = hashValue + 1 ; i!=hashValue ; i = (i+1)%1009)

{

if(SymbolTable[i].length == 0)

{

insertSymbol(SymbolTable, i, tokenName, tokenType);

return;

}

}


struct SymbolTableNode* SymbolTablePtr = (struct SymbolTableNode*)malloc(sizeof(struct
SymbolTableNode)) ;

*SymbolTablePtr = SymbolTable[hashValue];
```

```
while( SymbolTablePtr -> next != NULL)
{
SymbolTablePtr = SymbolTablePtr -> next;
}
struct SymbolTableNode* temp = (struct SymbolTableNode*)malloc(sizeof(struct SymbolTableNode))
;
strcpy(temp->tokenName,tokenName);
strcpy(temp->tokenType,tokenType);
temp->length = strlen(tokenName);
temp->line_number = yylineno;
temp->next = NULL;
SymbolTablePtr -> next = temp;
free(SymbolTablePtr);
}
}

void print_symbol_table()
{
printf("****************************************************************************\n");
printf("TokenName\tTokenType\t\t\tLength\tline_number\n");
printf("****************************************************************************");
for(int i = 0 ; i < 1009 ; i++)
{
if(SymbolTable[i].length != 0)
{
struct SymbolTableNode* SymbolTablePtr = (struct SymbolTableNode*)malloc(sizeof(struct
SymbolTableNode));
*SymbolTablePtr = SymbolTable[i];
while(SymbolTablePtr!=NULL)
{
printf("\n%s\t\t%s\t\t%d\t\t%d",SymbolTable[i].tokenName,
SymbolTable[i].tokenType,SymbolTable[i].length,SymbolTable[i].line_number);
SymbolTablePtr = SymbolTablePtr -> next;
```

```
}

}

}

printf("\n*************************************************************************\n"
);

}



/* Defining Constant Table and its related functions */


struct ConstantTableNode

{

char tokenName[100]; // 37 because that is the maximum lenght of the identifier that is valid in C

char tokenType[100];

int length;

int line_number;

// int scope;

struct ConstantTableNode* next;

};

/*

No particular reason to choose 1009 as the size of the SymbolTable,

just needed a big number or else whats the point of hashing if the array is Dynamic.

And also 1009 is prime

*/

struct ConstantTableNode ConstantTable[1009];


/* Defining functions to lookup and insert into the Symbol Table */

int look_up_constant_table(char* tokenName)

{

int hashValue = hash_function(tokenName);

if(ConstantTable[hashValue].length == 0)
```

```
{
return 0;
}
else if(strcmp(ConstantTable[hashValue].tokenName,tokenName)==0)
{
return 1;
}
else
{
// Checking Linear Probing Hash first
for(int i = hashValue + 1 ; i!=hashValue ; i = (i+1)%1009)
{
if(strcmp(ConstantTable[i].tokenName,tokenName)==0)
{
return 1;
}
}
/* If we reach the same index again and we have not found the
the tokenName we are searching for, this means we have done chaining , so we will
search horizontally */
struct ConstantTableNode* ConstantTablePtr = (struct ConstantTableNode*)malloc(sizeof(struct
ConstantTableNode)) ;
ConstantTablePtr = ConstantTable[hashValue].next;
while( ConstantTablePtr!= NULL)
{
if(strcmp(ConstantTablePtr->tokenName,tokenName)==0) return 1;
else ConstantTablePtr = ConstantTablePtr -> next;
}
}
return 0;// if we never found then return 0;
}
```

```
void insertConstant(struct ConstantTableNode ConstantTable[], int hashValue,

char *tokenName, char *tokenType)

{

strcpy(ConstantTable[hashValue].tokenName,tokenName);

strcpy(ConstantTable[hashValue].tokenType,tokenType);

ConstantTable[hashValue].length = strlen(tokenName);

ConstantTable[hashValue].line_number = yylineno;

ConstantTable[hashValue].next = NULL;

}



void insert_into_constant_table(char *tokenName, char *tokenType)

{

/* Checking for Lexical Error of constant limits */


if( (int)*tokenName > INT_MAX || (int)*tokenName < INT_MIN)

{

printf("******************************************************************\n");

printf("ERROR: Exceeded the max Value of the Integer in C language\n");

printf("******************************************************************\n");

return ;

}


/* We first look_up at the symbol table to check whether it exists ,

if it does we do not insert again

If value is not found then, we insert using linear Probing first

and if the symbol Table is full then we use chaining to insert

*/

if(look_up_constant_table(tokenName))

{

return;

}
```

```
else
{
int hashValue = hash_function(tokenName);
if(ConstantTable[hashValue].length == 0)
{
insertConstant(ConstantTable, hashValue, tokenName, tokenType);
return;
}

for (int i = hashValue + 1 ; i!=hashValue ; i = (i+1)%1009)
{
if(ConstantTable[i].length == 0)
{
insertConstant(ConstantTable, i, tokenName, tokenType);
return;
}
}

struct ConstantTableNode* ConstantTablePtr =(struct ConstantTableNode*)malloc(sizeof(struct ConstantTableNode)) ;
*ConstantTablePtr = ConstantTable[hashValue];
while( ConstantTablePtr -> next != NULL)
{
ConstantTablePtr = ConstantTablePtr -> next;
}
struct ConstantTableNode* temp = (struct ConstantTableNode*)malloc(sizeof(struct ConstantTableNode));
strcpy(temp->tokenName,tokenName);
strcpy(temp->tokenType,tokenType);
temp->length = strlen(tokenName);
temp->line_number = yylineno;
temp->next = NULL;
ConstantTablePtr -> next = temp;
```

```
free(ConstantTablePtr);
}
}

void print_constant_table()
{
printf("********************************************************************************\n");
printf("TokenName\tTokenType\t\t\tLength\tline_number\n");
printf("********************************************************************************");
for(int i = 0 ; i < 1009 ; i++)
{
if(ConstantTable[i].length != 0)
{
struct ConstantTableNode* ConstantTablePtr = (struct ConstantTableNode*)malloc(sizeof(struct ConstantTableNode)) ;
*ConstantTablePtr = ConstantTable[i];
while(ConstantTablePtr!=NULL)
{
printf("\n%s\t\t%s\t\t%d\t\t%d",ConstantTable[i].tokenName, ConstantTable[i].tokenType,ConstantTable[i].length,ConstantTable[i].line_number);
ConstantTablePtr = ConstantTablePtr -> next;
}
free(ConstantTablePtr);
}
}
}

%}

/* Definitions for Cleaner Code */

DEF "define"
INC "include"
```

```
operator
[[<][=]|[>][=]|[=][=]|[!][=]|[>]|[<]|[\|][\|]|[&][&]|[\!]|[=]|[\^]|[\+][=]|[\-][=]|[\*][=]|[\/][=]|[\%][=]|
[\+][\+]|[\-][\-]|[\+]|[\-]|[\*]|[\/]|[\%]|[&]|[\|]|[~]|[<][<]|[>][>]]




%%

\n {yylineno++;}

\/\/(.*) {printf("%s \t- same line comment\n", yytext);}

([#][" "]*({INC})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"] {printf("%s \t-Preprocessor
statement\n",yytext);} //Matches preprocessor directives

[ \n\t] ;

([#][" "]*({DEF})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"] {printf("%s \t-Definition\n",yytext);}
//Matches definition

\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/ {printf("%s \t- multiple line comment\n", yytext);}

, {printf("%s \t- comma separator\n", yytext);insert_into_symbol_table(yytext, "comma
separator\t");}

; {printf("%s \t- semicolon\n", yytext);insert_into_symbol_table(yytext, "semicolon\t");}

\} {printf("%s \t- closing curly brackets\n", yytext);insert_into_symbol_table(yytext, "closing curly
brackets");}

\] {printf("%s \t- closing square brackets\n", yytext);insert_into_symbol_table(yytext, "closing square
brackets");}

\( {printf("%s \t- opening brackets\n", yytext);insert_into_symbol_table(yytext, "opening brackets");}

\) {printf("%s \t- closing brackets\n", yytext);insert_into_symbol_table(yytext, "closing brackets");}

\. {printf("%s \t- dot\n", yytext);insert_into_symbol_table(yytext, "dot");}

\[ {printf("%s \t- opening square brackets\n", yytext);insert_into_symbol_table(yytext, "opening
square brackets");}

\: {printf("%s \t- colon\n", yytext);insert_into_symbol_table(yytext, "colon\t\t");}

\\ {printf("%s \t- forward slash\n", yytext);insert_into_symbol_table(yytext, "forward slash\t");}

\{ {printf("%s \t- opening curly brackets\n", yytext);insert_into_symbol_table(yytext, "opening curly
brackets");}

auto|break|default|printf|case|void|scanf|const|do|double|long|enum|float|sizeof|for|goto|cha
r|if|int|register|continue|return|short|else|typedef|static|unsigned|struct|switch|signed|union|
extern|while|volatile|main/[\(|" "|\{|;|:|"\n"|"\t"] {printf("%s \t- Keyword\n", yytext);
insert_into_symbol_table(yytext, "Keyword\t\t");}

\"[^\n]*\"/[;|,|\)] {printf("%s \t- String Constant\n", yytext);
insert_into_constant_table(yytext,"String Constant\t");}
```

```
\'[A-Z|a-z]\'/[;|,|\)|:] {printf("%s \t- Character Constant\n", yytext); insert_into_constant_table(yytext,"Character Constant");}

[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[ {printf("%s \t- Array Identifier\n", yytext); insert_into_symbol_table(yytext, "Identifier\t");}


{operator}/[a-z]|[0-9]|;|" "|[A-Z]|\(|\"|\'|\)|\n|\t {printf("%s \t- Operator\n", yytext); insert_into_symbol_table(yytext, "Operator\t");}


[1-9][0-9]*([eE][-+]?[0-9]+)?|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^] {printf("%s \t- Integer Constant\n", yytext); insert_into_constant_table(yytext, "Integer Constant");}

([0-9]*)\.([0-9]+)([eE][-+]?[0-9]+)?/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^] {printf("%s \t- Floating Constant\n", yytext); insert_into_constant_table(yytext, "Floating Constant");}

[A-Za-z_][A-Za-z_0-9]*/[" "|;|,|\(|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\.|\{|\^|\t] {printf("%s \t- Identifier\n", yytext); insert_into_symbol_table(yytext, "Identifier\t");}



(.?) {

if(yytext[0]=='"')

{

printf("************************************************************\n");

printf("ERROR: incomplete string at line no. %d\n",yylineno);

printf("************************************************************\n");

}

else if(yytext[0]=='#')

{

printf("************************************************************\n");

printf("ERROR: Pre-Processor directive at line no. %d\n",yylineno);

printf("************************************************************\n");


}

else if(yytext[0]=='/')

{

printf("************************************************************\n");

printf("ERROR: unmatched comment at line no. %d\n",yylineno);
```

```
printf("***********************************************************\n");
}
else
{
printf("***********************************************************\n");
printf("ERROR: at line no. %d\n",yylineno);
printf("***********************************************************\n");
}
printf("%s\n", yytext);
return 0;
}
%%

int main(int argc , char **argv){

printf("***********************************************************\n");

int i;
for (i=0;i<1009;i++){
SymbolTable[i].length=0;
ConstantTable[i].length=0;
}

yyin = fopen(argv[1],"r");
yylex();
printf("\n\nSymbol Table\n");
print_symbol_table();
printf("\n\nConstant Table\n");
print_constant_table();
printf("\n");
printf("***********************************************************************\n");
}
```

```
int yywrap(){
return 1;
}
```

List of Scanner Recognised Tokens:

The scanner recognises the following tokens:

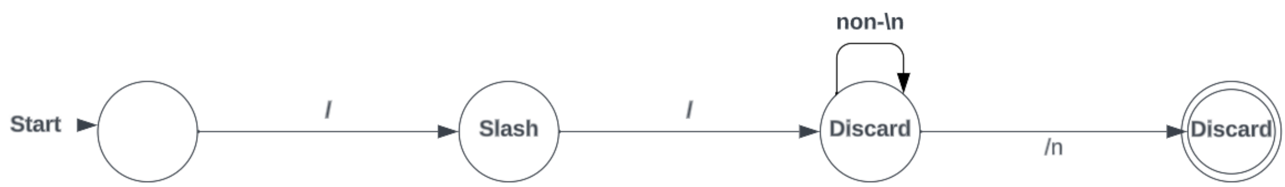
Token Name	Meaning
Preprocessor Statement	Represents preprocessor directives (e.g., `#include`) and prints the matched text as a preprocessor statement.
Definition	Represents macro definitions using `#define` and prints the matched text as a definition.
Multiple Line Comment	Represents multi-line comments enclosed in `/* */` and prints the matched text as a multiple-line comment.
Comma Separator	Represents a comma (`,`) and prints the matched text as a comma separator.
Semicolon	Represents a semicolon (`;`) and prints the matched text as a semicolon.
Closing Curly Brackets	Represents a closing curly bracket (`}`) and prints the matched text as a closing curly bracket.
Closing Square Brackets	Represents a closing square bracket (`]`) and prints the matched text as a closing square bracket.
Opening Brackets	Represents an opening parenthesis (`(`) and prints the matched text as an opening parenthesis.
Closing Brackets	Represents a closing parenthesis (`)`) and prints the matched text as a closing parenthesis.
Dot	Represents a dot (`.`) and prints the matched text as a dot.

Opening Square Brackets	Represents an opening square bracket (`[`) and prints the matched text as an opening square bracket.
Colon	Represents a colon (`:`) and prints the matched text as a colon.
Forward Slash (Escape Character)	Represents a forward slash (`\`) and prints the matched text as a forward slash.
Opening Curly Brackets	Represents an opening curly bracket (`{`) and prints the matched text as an opening curly bracket.
Keywords	Represents C keywords (e.g., `auto`, `break`, `if`, etc.) and inserts them into the symbol table.
String Constant	Represents string constants enclosed in double quotes (e.g., `"Hello, World!"`) and inserts them into the constant table.
Character Constant	Represents character constants enclosed in single quotes (e.g., `'A'`) and inserts them into the constant table.
Array Identifier	Represents array identifiers (variable names followed by `[`) and inserts them into the symbol table.
Operator	Represents operators in C code (e.g., `+`, `-`, `*`, `/`) and inserts them into the symbol table.
Integer Constant	Represents integer constants (e.g., `42`) and inserts them into the constant table.
Floating Constant	Represents floating-point constants (e.g., `3.14`) and inserts them into the constant table.
Identifier	Represents identifiers (variable names) and inserts them into the symbol table.

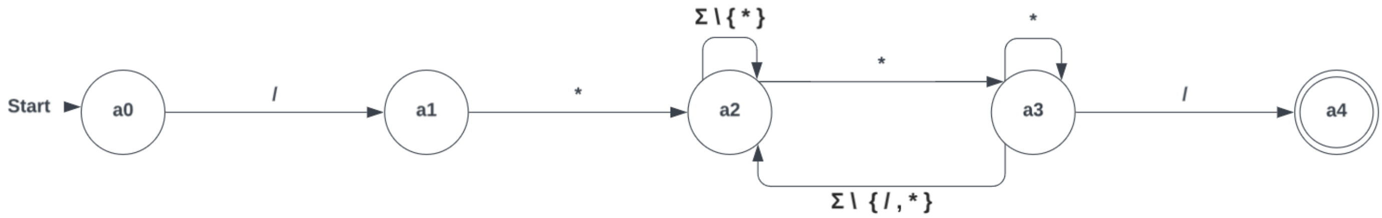
In the Flex program for lexical analysis of C code, a comprehensive list of C keywords is defined to be recognized and categorized as "Keyword" tokens. These keywords include commonly used language constructs such as "auto," "break," "default," "printf," "case," "void," "scanf," "const," "do," "double," "long," "enum," "float," "sizeof," "for," "goto," "char," "if," "int," "register," "continue," "return," "short," "else," "typedef," "static," "unsigned," "struct," "switch," "signed," "union," "extern," "while," "volatile," and "main." These keywords play essential roles in defining the structure and behavior of C programs, and their recognition is crucial for accurate lexical analysis of C code.

DFA for the scanner

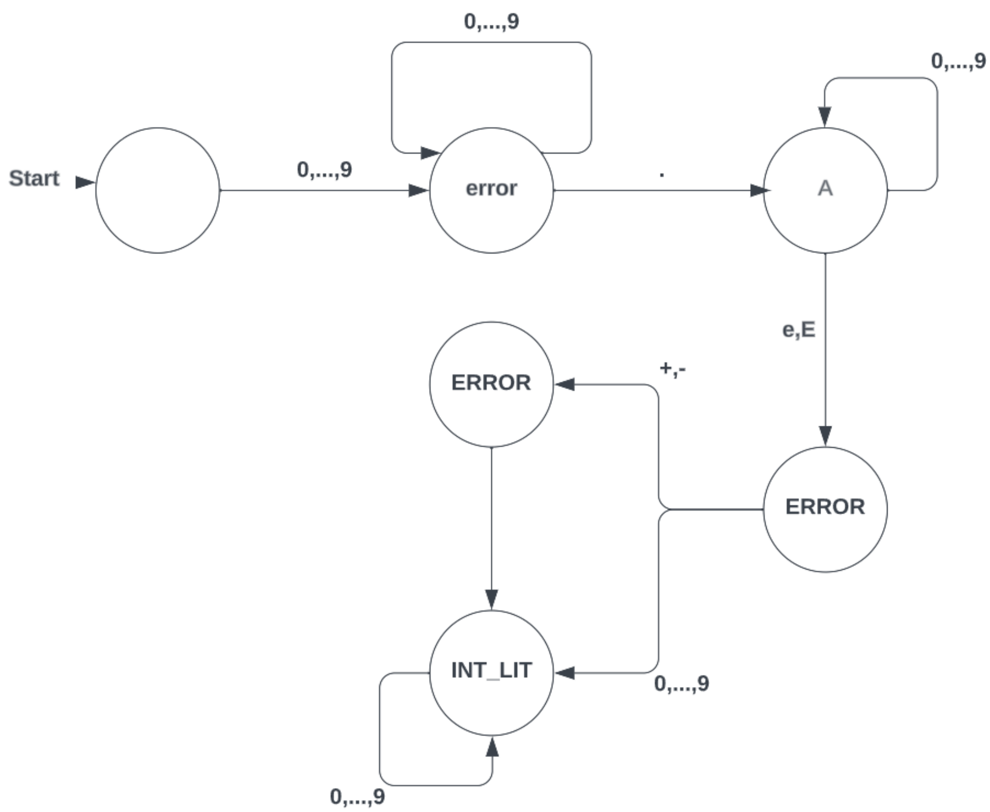
DFA for single line comments



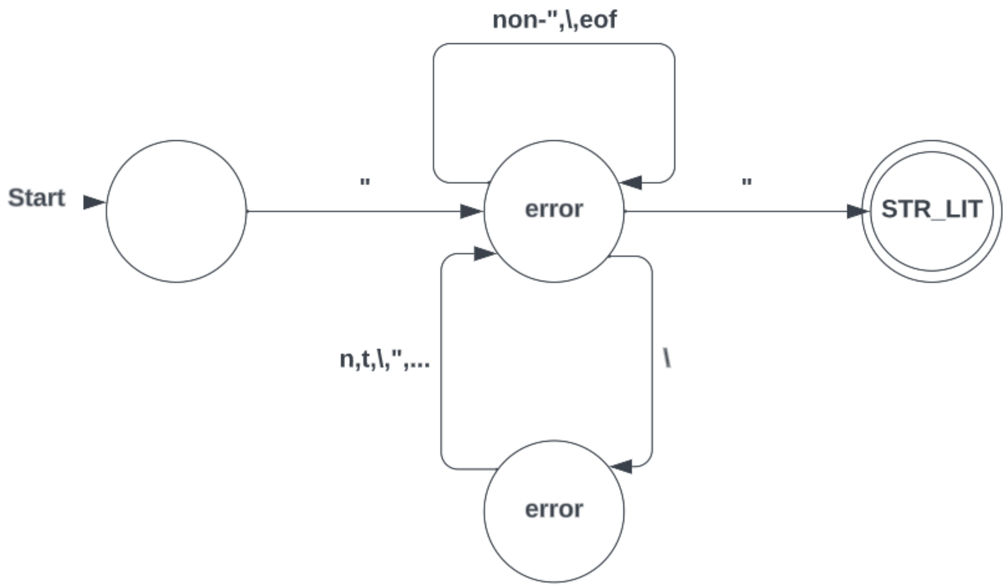
DFA for multiline comments



DFA for String Literals



DFA for Integer and Float



Discussion on assumptions made beyond what is in the basic language description :

In the provided code for the lexical analyser of the C programming language, there are several assumptions made beyond the basic language description. These assumptions are related to the way the lexer is designed and how it handles certain aspects of the C language:

1. Handling of Preprocessor Directives: The lexer assumes that preprocessor directives (e.g., `#include`, `#define`) are correctly formatted and will not check for syntax errors within these directives. It assumes that the preprocessing stage has already taken care of any issues. However, in a real compiler, these directives might be checked for proper syntax, and errors would be reported if they are not well-formed.

2. **Assumption of Valid Input**: The code assumes that the input source code is syntactically correct C code. It does not perform full syntactic or semantic analysis. For example, it does not check if variables are declared before use, if functions are called with the correct number and types of arguments, or if there are other semantic errors in the code. Real compilers perform these checks in later stages of compilation.

3. **Error Reporting Assumptions**: When an error is encountered, the code assumes that printing an error message with the line number is sufficient for debugging or diagnosis. In a production compiler, error messages would typically provide more detailed information, including the nature of the error, the location within the source code, and possibly suggestions for correction.

4. **Limited Testing and Error Handling**: The code assumes that it has been thoroughly tested for a wide range of C language constructs and potential input scenarios. It does not include extensive error handling for every possible edge case or invalid input.

5. **Specific Compiler Target**: The code does not consider different compiler targets or architectures. It assumes a single target architecture for the generated machine code.

In summary, the provided code makes several simplifying assumptions to focus on the core task of lexical analysis. While these assumptions are reasonable for educational or basic lexing purposes, a production-level compiler would need to handle a broader range of cases and perform more comprehensive error checking and reporting.

Report on test cases along with output :

We wrote a few test cases to demonstrate the symbol and constant table of the program.

test1.c

tests > test1.c > main()

```c
#include <stdio.h>

/*
 This code is error free, it's sole purpose is
 to demonstrate the symbol and constant table
 */
int main()
{
    printf("hello");
    int a = 1e9;
    int b = 2;
    int c = a + b;
    if ( c >= -1)
    {
        printf("%d",c);
    }
    return 0;
}
```

Output:

```
● (base) meherrushi@MeherRushi:~/Meher/projects/Compiler_Design/LexicalAnalyser$ ./a.out ../tests/test1.c
************************************************************
#include <stdio.h>      -Preprocessor statement
/*
 This code is error free, it's sole purpose is
 to demonstrate the symbol and constant table
 */      - multiple line comment
int      - Keyword
main     - Keyword
(        - opening brackets
)        - closing brackets
{        - opening curly brackets
printf   - Keyword
(        - opening brackets
"hello"          - String Constant
)        - closing brackets
;        - semicolon
int      - Keyword
a        - Identifier
=        - Operator
1e9      - Integer Constant
;        - semicolon
int      - Keyword
b        - Identifier
=        - Operator
2        - Integer Constant
;        - semicolon
int      - Keyword
c        - Identifier
=        - Operator
a        - Identifier
+        - Operator
b        - Identifier
;        - semicolon
if       - Keyword
(        - opening brackets
```

```
if       - Keyword
(        - opening brackets
c        - Identifier
>=       - Operator
-        - Operator
1        - Integer Constant
)        - closing brackets
{        - opening curly brackets
printf   - Keyword
(        - opening brackets
"%d"     - String Constant
,        - comma separator
c        - Identifier
)        - closing brackets
;        - semicolon
}        - closing curly brackets
return   - Keyword
0        - Integer Constant
;        - semicolon
}        - closing curly brackets


Symbol Table
************************************************************************
TokenName       TokenType                       Length  line_number
************************************************************************
a               Identifier                      1               7
b               Identifier                      1               8
c               Identifier                      1               9
{               opening curly brackets          1               5
}               closing curly brackets          1               13
main            Keyword                         4               4
return          Keyword                         6               14
printf          Keyword                         6               6
if              Keyword                         2               10
int             Keyword                         3               4
```

```
Symbol Table
************************************************************************
TokenName        TokenType                        Length  line_number
************************************************************************
a                Identifier                       1               7
b                Identifier                       1               8
c                Identifier                       1               9
{                opening curly brackets           1               5
}                closing curly brackets           1               13
main             Keyword                          4               4
return           Keyword                          6               14
printf           Keyword                          6               6
if               Keyword                          2               10
int              Keyword                          3               4
>=               Operator                         2               10
(                opening brackets                 1               4
)                closing brackets                 1               4
+                Operator                         1               9
,                comma separator                  1               12
-                Operator                         1               10
;                semicolon                        1               6
=                Operator                         1               7
************************************************************************


Constant Table
************************************************************************
TokenName        TokenType                        Length  line_number
************************************************************************
1e9              Integer Constant                 3               7
"hello"          String Constant                  7               6
"%d"             String Constant                  4               12
0                Integer Constant                 1               14
1                Integer Constant                 1               10
2                Integer Constant                 1               8
************************************************************************
```

test2.c

Dealing with errors that are spelling mistakes ie inserting random numbers within the number.

tests > C test2.c > main()

```c
#include<stdio.h>

void main() {
    int a = 1;
    int b = 2;
    int c = 1b2;
}
```

Here we can observe that 1b2 is an error

Output:

```
(base) meherrushi@MeherRushi:~/Meher/projects/Compiler_Design/LexicalAnalyser$ ./a.out ../tests/test2.c
************************************************************
#include<stdio.h>       -Preprocessor statement
void     - Keyword
main     - Keyword
(        - opening brackets
)        - closing brackets
{        - opening curly brackets
int      - Keyword
a        - Identifier
=        - Operator
1        - Integer Constant
;        - semicolon
int      - Keyword
b        - Identifier
=        - Operator
2        - Integer Constant
;        - semicolon
int      - Keyword
c        - Identifier
=        - Operator
************************************************************
ERROR: at line no. 6
************************************************************
1
```

The compiler detects the error at line 6.

```
Symbol Table
************************************************************************
TokenName       TokenType                       Length   line_number
************************************************************************
a               Identifier                      1               4
b               Identifier                      1               5
c               Identifier                      1               6
{               opening curly brackets          1               3
main            Keyword                         4               3
int             Keyword                         3               4
void            Keyword                         4               3
(               opening brackets                1               3
)               closing brackets                1               3
;               semicolon                       1               4
=               Operator                        1               4
************************************************************************


Constant Table
************************************************************************
TokenName       TokenType                       Length   line_number
************************************************************************
1               Integer Constant                1               4
2               Integer Constant                1               5
************************************************************************
```

test3.c

In this program we can see that an invalid string is present i.e the double quotes is not closed

```
tests > C test3.c > main()
#include<stdio.h>

int main() {

    //Valid string

    char[] valid = "Valid String";

    //Invalid String : Here the string is not closed

    char[] invalid = "Invalid String;

    return 0;
}
```

Output:

```
(base) meherrushi@MeherRushi:~/Meher/projects/Compiler_Design/LexicalAnalyser$ ./a.out ../tests/test3.c
************************************************************
#include<stdio.h>       -Preprocessor statement
int      - Keyword
main     - Keyword
(        - opening brackets
)        - closing brackets
{        - opening curly brackets
//Valid string  - same line comment
char     - Array Identifier
[        - opening square brackets
]        - closing square brackets
valid    - Identifier
=        - Operator
"Valid String"  - String Constant
;        - semicolon
//Invalid String : Here the string is not closed        - same line comment
char     - Array Identifier
[        - opening square brackets
]        - closing square brackets
invalid         - Identifier
=        - Operator
************************************************************
ERROR: incomplete string at line no. 11
************************************************************
"


Symbol Table
************************************************************************
TokenName       TokenType                       Length  line_number
************************************************************************
[               opening square brackets         1               7
]               closing square brackets         1               7
{               opening curly brackets          1               3
main            Keyword                         4               3
int             Keyword                         3               3
valid           Identifier                      5               7
invalid         Identifier                      7               11
```

```
Symbol Table
************************************************************************
TokenName       TokenType                       Length  line_number
************************************************************************
[               opening square brackets         1               7
]               closing square brackets         1               7
{               opening curly brackets          1               3
main            Keyword                         4               3
int             Keyword                         3               3
valid           Identifier                      5               7
invalid         Identifier                      7               11
char            Identifier                      4               7
(               opening brackets                1               3
)               closing brackets                1               3
;               semicolon                       1               7
=               Operator                        1               7
************************************************************************


Constant Table
************************************************************************
TokenName       TokenType                       Length  line_number
************************************************************************
"Valid String"          String Constant                 14              7
************************************************************************
```

test4.c

ests > test4.c > main()

```c
#include<stdio.h>

void main() {

    //The variable name length cannot exceed 37

    //Valid Variable name

    long valid = 56;

    /*Invlaid Valid name as the length of variable name exceeds 37*/

    int thisVariableNameWillExceed37LettersLimit = 56;

}
```

The variables in C have a maximum length of 37 characters, if the length of the variable exceeds that then we get an error

Output:

```
(base) meherrushi@MeherRushi:~/Meher/projects/Compiler_Design/LexicalAnalyser$ ./a.out ../tests/test4.c
***************************************************************
#include<stdio.h>       -Preprocessor statement
void     - Keyword
main     - Keyword
(        - opening brackets
)        - closing brackets
{        - opening curly brackets
//The variable name length cannot exceed 37     - same line comment
//Valid Variable name   - same line comment
long     - Keyword
valid    - Identifier
=        - Operator
56       - Integer Constant
;        - semicolon
/*Invlaid Valid name as the length of variable name exceeds 37*/        - multiple line comment
int      - Keyword
thisVariableNameWillExceed37LettersLimit        - Identifier
***************************************************************
ERROR: Exceeded the maxLength of the Valid Token in C language
***************************************************************
=        - Operator
56       - Integer Constant
;        - semicolon
}        - closing curly brackets


Symbol Table
***************************************************************************
TokenName        TokenType                        Length  line_number
***************************************************************************
{                opening curly brackets           1               3
}                closing curly brackets           1               15
main             Keyword                          4               3
int              Keyword                          3               13
void             Keyword                          4               3
long             Keyword                          4               9
valid            Identifier                       5               9
(                opening brackets                 1               3
```


```
(base) meherrushi@MeherRushi:~/Meher/projects/Compiler_Design/LexicalAnalyser$ ./a.out ../tests/test4.c
***************************************************************
#include<stdio.h>       -Preprocessor statement
void     - Keyword
main     - Keyword
(        - opening brackets
)        - closing brackets
{        - opening curly brackets
//The variable name length cannot exceed 37     - same line comment
//Valid Variable name   - same line comment
long     - Keyword
valid    - Identifier
=        - Operator
56       - Integer Constant
;        - semicolon
/*Invlaid Valid name as the length of variable name exceeds 37*/        - multiple line comment
int      - Keyword
thisVariableNameWillExceed37LettersLimit        - Identifier
***************************************************************
ERROR: Exceeded the maxLength of the Valid Token in C language
***************************************************************
=        - Operator
56       - Integer Constant
;        - semicolon
}        - closing curly brackets


Symbol Table
***************************************************************************
TokenName        TokenType                        Length  line_number
***************************************************************************
{                opening curly brackets           1               3
}                closing curly brackets           1               15
main             Keyword                          4               3
int              Keyword                          3               13
void             Keyword                          4               3
long             Keyword                          4               9
valid            Identifier                       5               9
(                opening brackets                 1               3
```

```
Symbol Table
************************************************************************
TokenName       TokenType                       Length  line_number
************************************************************************
{               opening curly brackets          1               3
}               closing curly brackets          1               15
main            Keyword                         4               3
int             Keyword                         3               13
void            Keyword                         4               3
long            Keyword                         4               9
valid           Identifier                      5               9
(               opening brackets                1               3
)               closing brackets                1               3
;               semicolon                       1               9
=               Operator                        1               9
************************************************************************


Constant Table
************************************************************************
TokenName       TokenType                       Length  line_number
************************************************************************
56              Integer Constant                2               9
************************************************************************
(base) meherruchi@MeherRuchi:~/Meher/projects/Compiler_Design/LexicalAnalyser$
```

test5.c

```c
tests > C test5.c
#include<stdio.h>

int main() {

    /*Closed
    Multiline
    Comment*/

    //here the multiline comment is not closed

    /*Not closed
    Multiline
    Comment

   return 0;
}
```

This program has an error of not completing the multiline comment. Our lexical analyser finds out and shows this error

Output:

```
(base) meherrushi@MeherRushi:~/Meher/projects/Compiler_Design/LexicalAnalyser$ ./a.out ../tests/test5.c
************************************************************
#include<stdio.h>       -Preprocessor statement
int     - Keyword
main    - Keyword
(       - opening brackets
)       - closing brackets
{       - opening curly brackets
/*Closed
    Multiline
    Comment*/   - multiple line comment
//here the multiline comment is not closed      - same line comment
************************************************************
ERROR: unmatched comment at line no. 9
************************************************************
/


Symbol Table
************************************************************************
TokenName       TokenType                       Length  line_number
************************************************************************
{               opening curly brackets          1               3
main            Keyword                         4               3
int             Keyword                         3               3
(               opening brackets                1               3
)               closing brackets                1               3
************************************************************************


Constant Table
************************************************************************
TokenName       TokenType                       Length  line_number
************************************************************************
************************************************************************
```

test6.c

```
tests > C test6.c > main()
#include<stdio.h>

void main() {

    //Correct format for variable name

    int num3 = 300;

    /*Incorrect Format for variable name
    Variable name cannot start with a number*/

    int 3num = 300;

    printf("%d", num3+5);
}
```

Output:

Here we see the incorrect naming of variable and identifier names should start with either a Letter or an underscore and not with a number.

```
● (base) meherrushi@MeherRushi:~/Meher/projects/Compiler_Design/LexicalAnalyser$ ./a.out ../tests/test6.c
 ************************************************************
 #include<stdio.h>        -Preprocessor statement
 void     - Keyword
 main     - Keyword
 (        - opening brackets
 )        - closing brackets
 {        - opening curly brackets
 //Correct format for variable name      - same line comment
 int      - Keyword
 num3     - Identifier
 =        - Operator
 300      - Integer Constant
 ;        - semicolon
 /*Incorrect Format for variable name
     Variable name cannot start with a number*/  - multiple line comment
 int      - Keyword
 ************************************************************
 ERROR: at line no. 11
 ************************************************************
 3


 Symbol Table
 ************************************************************************
 TokenName       TokenType                      Length  line_number
 ************************************************************************
 {               opening curly brackets         1               3
 num3            Identifier                     4               7
 main            Keyword                        4               3
 int             Keyword                        3               7
 void            Keyword                        4               3
```

```
3


Symbol Table
************************************************************************
TokenName       TokenType                      Length  line_number
************************************************************************
{               opening curly brackets         1               3
num3            Identifier                     4               7
main            Keyword                        4               3
int             Keyword                        3               7
void            Keyword                        4               3
(               opening brackets               1               3
)               closing brackets               1               3
;               semicolon                      1               7
=               Operator                       1               7
************************************************************************


Constant Table
************************************************************************
TokenName       TokenType                      Length  line_number
************************************************************************
300             Integer Constant               3               7
************************************************************************
(base) meherrushi@MeherRushi:~/Meher/projects/Compiler_Design/LexicalAnalyser$
```

test7.c

```
tests > C test7.c > ...
#include<stdio.h>

void main() {
    int a = 1;
    int b = 2;
    float e = 1.2.3;
}

```

Improper initialization of float. So the lexical analyser identifies it as an error because we cannot initialize properly.

```
(base) meherrushi@MeherRushi:~/Meher/projects/Compiler_Design/LexicalAnalyser$ ./a.out ../tests/test7.c
*************************************************************
#include<stdio.h>        -Preprocessor statement
void     - Keyword
main     - Keyword
(        - opening brackets
)        - closing brackets
{        - opening curly brackets
int      - Keyword
a        - Identifier
=        - Operator
1        - Integer Constant
;        - semicolon
int      - Keyword
b        - Identifier
=        - Operator
2        - Integer Constant
;        - semicolon
float    - Keyword
e        - Identifier
=        - Operator
*************************************************************
ERROR: at line no. 6
*************************************************************
```