# DESIGNING A SCANNER FOR C LANGUAGE

A Mini Project Report Submitted by


**Sahil Bhanastarkar**
(4NM17CS156)

**Rohit J**
(4NM17CS148)


UNDER THE GUIDANCE OF

**Mr.Ranjan Kumar H. S.**

Department of Computer Science and Engineering

in partial fulfilment of the requirements for the award of the Degree of Bachelor of Engineering in Computer Science & Engineering from Visvesvaraya Technological University, Belgaum



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

N.M.A.M. INSTITUTE OF TECHNOLOGY

(An Autonomous Institution under VTU, Belgaum) (AICTE approved, NBA Accredited, ISO 9001:2008 Certified)
NITTE -574 110, Udupi District, KARNATAKA.


**April 2020**

NITTE EDUCATION TRUST | N.M.A.M. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution affiliated to Visvesvaraya Technological University, Belagavi)
Nitte – 574 110, Karnataka, India

(ISO 9001:2015 Certified), Accredited with 'A' Grade by NAAC

: 08258 - 281039 – 281263, Fax: 08258 – 281265

# DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

# CERTIFICATE

"DESIGNING A SCANNER FOR C

LANGUAGE"

is a bonafide work carried out by

Sahil Bhanastarkar (4NM17CS156) Rohit J (4NM17CS148)

In partial fulfilment of the requirements for the award of Bachelor of Engineering Degree in Computer Science and Engineering prescribed by Visvesvaraya Technological University,Belgaum during the year 2018-2019.

It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report.

The Mini project report has been approved as it satisfies the academic requirements in respect of the project work prescribed for the Bachelor of Engineering Degree.

Signature of Guide Signature of HOD

# I. ACKNOWLEDGEMENT

We believe that our project will be complete only after we thank the people who have contributed to make this project successful.

First and foremost, our sincere thanks to our beloved principal, **Dr. Niranjan N. Chiplunkar** for giving us an opportunity to carry out our project work at our college and providing us with all the needed facilities.

We express our deep sense of gratitude and indebtedness to our guide**Mr. Ranjan Kumar H S**, Assistant Professor, Department of Computer Science and Engineering, for his inspiring guidance, constant encouragement, support and suggestions for improvement during the course of our project.

We sincerely thank **Dr. K.R. Udaya Kumar Reddy**, Head of Department of Computer Science and Engineering, Nitte Mahalinga Adyantaya Memorial Institute of Technology, Nitte.

We also thank all those who have supported us throughout the entire duration of our project.

Finally, we thank the staff members of the Department of Computer Science and Engineering and all our friends for their honest opinions and suggestions throughout the course of our project.

Student names and USN

**Sahil Bhanastarkar**
(4NM17CS156)

**Rohit J**
(4NM17CS148)

## ABSTRACT

A compiler is a computer program that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages or passes, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.

Syntax analysis or parsing is the second phase of a compiler. A lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata. A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. Parsing, syntax analysis or syntactic analysis is the process of analysing a string of symbols, either in natural language or in computer languages, conforming to the rules of a formal grammar. Syntactic analysis, or parsing, is needed to determine if the series of tokens given are appropriate in a language - that is, whether or not the sentence has the right shape/form.

However, not all syntactically valid sentences are meaningful, further semantic analysis has to be applied for this. For syntactic analysis, context-free grammars and the associated parsing techniques are powerful enough to be used - this overall process is called parsing. There are many techniques for parsing algorithms, and the two main classes of algorithm are top-down and bottom-up parsing. The top-down parsing uses leftmost derivation, and the bottom-up parsing uses rightmost derivation.

## TABLE OF CONTENTS

# INTRODUCTION

## THE LEXICAL ANALYSIS

In computer science, lexical analysis is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an identified "meaning"). A program that performs lexical analysis may be called a lexer, tokenizer, or scanner (though "scanner" is also used to refer to the first stage of a lexer). Such a lexer is generally combined with a parser, which together analyze the syntax of programming languages, web pages, and so forth.
The script written by us is a computer program called the “lex” program, is the one that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming Language.
A lexer takes an arbi-trary input stream and tokenizes it, i.e., divides it up into lexical tokens. This tokenized output can then be processed further, usually by yacc, or it
can be the "end product."

## Syntax Analysis

In computer science, syntax analysis is the process of checking that the code is syntactically correct. The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequences of symbols, keywords, identifiers etc. The parser needs to be able to handle the infinite number of possible valid programs that may be presented to it. The usual way to define the language is to specify a grammar. A grammar is a set of rules (or productions) that specifies the syntax of the language (i.e. what is a valid sentence in the language). There can be more than one grammar for a given language. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree. The syntax analyser for the C language by writing two scripts, one that acts as a lexical analyzer (lexer) and outputs a stream of tokens, and the other one that acts as a parser. The lexer is known as the lex program. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

Structure of Lex Program

The structure of the lex program consists of three sections:

{definition section}

```
%{

#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include "y.tab.h"

int cmnt_strt = 0;

%}
```

%%

{rules section}

The rules section contains the patterns and actions that specify the lexer. Here is our sample word count's rules section:

```
%%
{word}      { wordCount++; charCount += yyleng; }
{eol} { charCount++; lineCount++; }
.     charCount++;
```

%%

{C code section}

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section.

## Structure of Yacc Program

The parser written is known as the Yacc program. The structure of the Yacc file is similar to that of the lexer, consisting of three sections:

{declarations}

%%

{rules}

%%

{routines}

The declarations section of a yacc file may consist of the following:

● %token - identifies the token names that the yacc file accepts

```
%token HEADER_FILE

%token IDENTIFIER

 /* Constants */
%token DEC_CONSTANT HEX_CONSTANT
%token STRING

 /* Logical and Relational operators */
%token AND OR LSEQ GREQ EQ NOTEQ

 /* Short hand assignment operators */
%token MUL_EQ DIV_EQ MOD_EQ ADD_EQ SUB_EQ
%token INCREMENT DECREMENT

 /* Data types */
%token SHORT INT LONG LONG_LONG SIGNED UNSIGNED CONST

 /* Keywords */
%token IF FOR WHILE CONTINUE BREAK RETURN ELSE
```

● %start - identifies a nonterminal name for the start symbol

● %right - identifies tokens that are right-associative with other tokens

● %left - identifies tokens that are left-associative with other tokens

```
%start start

%left ','
%right '='
%left OR
%left AND
%left EQ NOTEQ
%left '<' '>' LSEQ GREQ
%left '+' '-'
%left '*' '/' '%'
%right '!'
```

● %nonassoc - identifies tokens that are not associative with other tokens

The rules section consists of the context free grammar used to generate the parse tree. A general rule has the following structure:

nonterminal

: sentential form

| sentential form

.....

| sentential form ;

Actions may be associated with rules and are executed when the associated sentential form is matched.

The routines section may include the C program that specifies the input file, action routines and other user defined functions.

## IMPLEMENTATION

Below is a list containing the different tokens that are identified by our Flex program. It also gives a detailed description of how the different tokens are identified and how errors are detected if any.

### Keywords

The keywords identified are: int, long, short, long long, signed, unsigned, for, break, continue, if, else, return.

```
  /* Keywords*/
"int"                   {return INT;}
"long"                  {return LONG;}
"long long"             {return LONG_LONG;}
"short"                 {return SHORT;}
"signed"                {return SIGNED;}
"unsigned"              {return UNSIGNED;}
"for"                   {return FOR;}
"while"                 {return WHILE;}
"break"                 {return BREAK;}
"continue"              {return CONTINUE;}
"if"                    {return IF;}
"else"                  {return ELSE;}
"return"                {return RETURN;}
```

### Identifiers

- Identifiers are identified and added to the symbol table. The rule followed is represented by the regular expression (_|{letter})({letter}|{digit}|_){0,31} .

```
identifier (_|{letter})({letter}|{digit}|_){0,31}
```

- The rule only identifies those lexemes as identifiers which either begin with a letter or an underscore and is followed by either a letter, digit or an underscore with a maximum length of 32.

- The first part of the regular expression (_|{letter}) ensures that the identifiers begin with an underscore or a letter and the second part ({letter}|{digit}|_){0,31}

matches a combination of letters, digits and underscore and ensures that the maximum length does not exceed 32. The definitions of {letter} and {digit} can be seen in the code at the end.

- Any identifier that begins with a digit is marked as a lexical error and the same is displayed on the output. The regex used for this is { digit}+({letter}|_)+

## Comments

Single and multi line comments are identified. Single line comments are identified by //.* regular expression. The multiline regex is identified as follows:

- We make use of an exclusive state called <CMNT> . When a /* pattern is found, we note down the line number and enter the <CMNT> exclusive state. When the */ is found, we go back to the INITIAL state, the default state in Flex, signifying the end of the comment.

- Then we define patterns that are to be matched only when the lexer is in the <CMNT> state. Since, it is an exclusive state, only the patterns that are defined for this state (the ones prepended with <CMNT> in the lex file are matched, rest of the patterns are inactive.

- We also identify nested comments. If we find another /* while still in the <CMNT> state, we print an error message saying that nested comments are invalid.
- If the comment does not terminate until EOF , an error message is displayed along with the line number where the comment begins. This is implemented by checking if the lexer matches <<EOF>> pattern while still in the <CMNT> state, which means that a */ has not been found until the end of file and therefore the comment has not been terminated.

```
"/*"                    {cmnt_strt = yylineno; BEGIN CMNT;}
<CMNT>.|{ws}            ;
<CMNT>\n                {yylineno++;}
<CMNT>"*/"              {BEGIN INITIAL;}
<CMNT>"/*"              {printf("Line %3d: Nested comments are not valid!\n",yylineno);}
<CMNT><<EOF>>           {printf("Line %3d: Unterminated comment\n", cmnt_strt); yyterminate();}
```

## Strings

The lexer can identify strings in any C program. It can also handle double quotes that are escaped using a \ inside a string. Further, error messages are displayed for unterminated strings. We use the following strategy

- We first match patterns that are within double quotes.
- But if the string is something like “This is \” a string”, it will only match “This is \” . So as soon as a match is found we first check if the last double quote is escaped using a

backslash.

- If the last quote is not escaped with a backslash we have found the string we are looking for and we add it to the constants table.

But in case the last double quote is escaped with a backslash we push the last double quote back for scanning. This can be achieved in lex using the command yyless(yyleng - 1).

- yyless(n) tells lex to "push back" all but the first n characters of the matched token. yyleng holds the length of the matched token.
- And hence yyless(yyleng -1) will push back the last character i.e the double quote back for scanning and lex will continue scanning from " is a string" .
- We use another built-in lex function called yymore() which tells lex to append the next matched token to the currently matched one.
- Now the lexer continues and matches "is a string" and since we had called yymore() earlier it appends it to the earlier token "This is \ giving us the entire string "This is \" a string" . Notice that since we had called yyless(yyleng - 1) the last double quote is left out from the first matched token giving us the entire string as required.
- The following lines of code accomplish the above described process.

```
\"[^\"\n]*\"   {

  if(yytext[yyleng-2]=='\\') /* check if it was an escaped quote */
  {
       yyless(yyleng-1);     /* push the quote back if it was escaped */
       yymore();            /* Append next token to this one */
  }
  else{
  insert( constant_table, yytext, STRING);
 }
```

- We use the regular expression \ "[^\"\n]*$ to check for strings that don't terminate. This regular expression checks for a sequence of a double quote followed by zero or more occurrences of characters excluding double quotes and new line and this sequence should not have a close quote. This is specified by the $ character which tests for the end of line. Thus, the regular expression checks for strings that do not terminate till the end of line and it prints an error message on the screen.

## Integer Constants

- The Flex program can identify two types of numeric constants: decimal and hexadecimal. The regular expressions for these are [+-]?{digit}+[lLuU]? and [+-]?0[xX]{hex}+[lLuU]? respectively.
- The sign is considered as optional and a constant without a sign is by default positive.

All hexadecimal constants should begin with 0x or 0X .

- The definition of {digit} is all the decimal digits 0-9 . The definition of {hex} consists of the hexadecimal digits 0-9 and characters a-f .
- Some constants which are assigned to long or unsigned variables may suffix l or L and u or U or a combination of these characters with the constant. All of these conditions are taken care of by the regular expression.

## Preprocessor Directives

The filenames that come after the #include are selectively identified through the exclusive state <PREPROC> since the regular expressions for treating the filenames must be kept different from other regexes.

Upon encountering a #include at the beginning of a line, the lexer switches to the state <PREPROC> where it can tokenize filenames of the form “stdio.h” or <stdio.h> . Filenames of any other format are considered as illegal and an error message regarding the same is printed.

```
^"#include"                    {BEGIN PREPROC;}
<PREPROC>"<"[^<>\n]+">"         {return HEADER_FILE;}
<PREPROC>{ws}                   {;}
<PREPROC>\"[^"\n]+\"           {return HEADER_FILE;}
<PREPROC>\n                    {yylineno++; BEGIN INITIAL;}
<PREPROC>.                     {printf("Line %3d: Illegal header file format \n",yylineno);}
"//".*                         ;
```

# RESULTS

## Test-cases & Screenshots

```c
//Testcase 1.c

#include<
int main(){

    int a = 10;
    intyuhi b = 10

    for(a = 2 a<3; a++)
    b = b + 1;

    printf (" This string is enclosed in double quotes ");
    printf (" This string is not enclosed in double quotes );



    return 0;
}
```

```
sahil@sahil-HP-Laptop-14s-cr1xxx:~/Desktop/CDProject$ ./parser testcases/test-case-1.c
Line   3: Illegal header file format
Line no: 7 Error message: syntax error Token: b
Line no: 9 Error message: syntax error Token: )
Line  13: Unterminated string " This string is not enclosed in double quotes );
Line no: 17 Error message: syntax error Token: return

Parsing complete
```

```c
//Testcase 2.c

#include<stdio.h>


int main()
{
        int array1[10];
        int array2[20;

        for(int a = 3; a<4; a++
        {
                a = array1[0];
        }

        func();
}

int func()
{
        return 20;
```

```
sahil@sahil-HP-Laptop-14s-cr1xxx:~/Desktop/CDProject$ ./parser testcases/test-case-2.c
Line no: 9 Error message: syntax error Token: ;
Line no: 11 Error message: syntax error Token: int
Line no: 12 Error message: syntax error Token: {
Line no: 16 Error message: syntax error Token: (

Parsing complete
```

```c
//Testcase 3.c

#include<stdio.h>
int main()
{
  int x, y;
  long long int total, diff;
  int *ptr;
  int a = 86;

  if (a > 90)
  {
          printf ("Grade is AA");
  }
  else if (a > 80)
  {
          printf ("Grade is AB");
  }
  else
  {
          printf ("Grade is BB");
  }

  x = -10, y = 20;
  x=x*3/2;
  total = x + y;
}

```

```
sahil@sahil-HP-Laptop-14s-cr1xxx:~/Desktop/CDProject$ ./parser testcases/test-case-4.c

Parsing complete
```

```c
//Testcase 4.c

#include<stdio.h>
int main()
{
   int c = 0,d,e,f;

   c = c + 1;  // c = 1
   d = c * 5;  // d = 5
   e = d / 4;  // e = 1
   f = d % 3;  // f = 2
   f = f / 0;
}
```

```
sahil@sahil-HP-Laptop-14s-cr1xxx:~/Desktop/CDProject$ ./parser testcases/test-case-5.c

Parsing complete
```

## CONCLUSION

The lexical analyzer and the syntax analyzer for a subset of C language, which include selection statements, compound statements, iteration statements, jumping statements, user defined functions and primary expressions are generated. It is important to note that conflicts (shift-reduce and reduce-reduce) may occur in case of syntax analyzer if proper care is not taken while specifying the context-free grammar for the language. We should always specify unambiguous grammar for the parser to work properly.

PROJECT CODE:

https://github.com/sahilb8/mini-c-parser