# Easy Parser Project

*Raveena Mehta*

*Dept. of Electrical and Computer Engineering*

*Stevens Institute of technology*

*Hoboken, USA*

*rmehta19@stevens.edu*

*Manav Parekh*

*Dept. of Electrical and Computer Engineering*

*Stevens Institute of technology*

*Hoboken, USA*

*mparekh3@stevens.edu*

*Tanay Parekh*

*Dept. of Electrical and Computer Engineering*

*Stevens Institute of technology*

*Hoboken, USA*

*tparekh1@stevens.edu*

**Abstract—** There are many existing tools for parsing such as lex and yacc. But using these tools, makes the process of parsing very complicated. In our project, we focus on the problem of lexical analysis which includes identifying a token and categorizing them. We tried to create our own tool used for lexical analysis in parsing. We implement our system by removing comments from our source code then, converting each word from our source code to tokens. Once, we got the tokens, we categorized them into different categories such as integers, floats, keywords and special characters. We also checked the unmatched parenthesis error from our source code and generate error if any. We concluded that converting the whole source code into a string for implementation of our project does not give results as well as increased the complexity of our program.

***Keywords— lex, yacc, parsing, token***

## PROBLEM INTRODUCTION

Parsing is one of the main phases of a compiler. A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens, interactive commands, or program instructions and breaks them up into parts that can be used by other components in programming.

The process of parsing has 3 stages: Lexical analysis, syntactical analysis- Checks whether the generated tokens form a meaningful expression. This makes use of a context-free grammar that defines algorithmic procedures for components. These work to form an expression and define the particular order in which tokens must be placed and semantical parsing- The final parsing stage in which the meaning and implications of the validated expression are determined and necessary actions are taken.

A parser's main purpose is to determine if input data may be derived from the start symbol of the grammar. If yes, then in what ways can this input data be derived? This is achieved as follows:

* **Top-Down Parsing:** Involves searching a parse tree to find the left-most derivations of an input stream by using a top-down expansion. Parsing begins with the start symbol which is transformed into the input symbol until all symbols are translated and a parse tree for an input string is constructed. Examples include LL parsers and recursive-descent parsers. Top-down parsing is also called predictive parsing or recursive parsing.
* **Bottom-Up Parsing:** Involves rewriting the input back to the start symbol. It acts in reverse by tracing out the rightmost derivation of a string until the parse tree is constructed up to the start symbol This type of parsing is also known as shift-reduce parsing. One example is an LR parser.

Parsers are widely used in the following technologies:

* Java and other programming languages.
* HTML and XML.
* Interactive data language and object definition language.
* Database languages, such as SQL.
* Modeling languages, such as virtual reality modeling language.
* Scripting languages.
* Protocols, such as HTTP and Internet remote function calls.

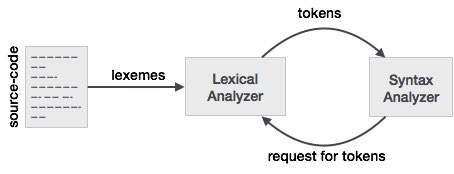
Top of Form

Techopedia Terms:    [#](https://www.techopedia.com/it-terms/1)    [A](https://www.techopedia.com/it-terms/a)    [B](https://www.techopedia.com/it-terms/b)    [C](https://www.techopedia.com/it-terms/c)    [D](https://www.techopedia.com/it-terms/d)    [E](https://www.techopedia.com/it-terms/e)    [F](https://www.techopedia.com/it-terms/f)    [G](https://www.techopedia.com/it-terms/g)    [H](https://www.techopedia.com/it-terms/h)    [I](https://www.techopedia.com/it-terms/i)    [J](https://www.techopedia.com/it-terms/j)    [K](https://www.techopedia.com/it-terms/k)    [L](https://www.techopedia.com/it-terms/l)    [M](https://www.techopedia.com/it-terms/m)    [N](https://www.techopedia.com/it-terms/n)    [O](https://www.techopedia.com/it-terms/o)    [P](https://www.techopedia.com/it-terms/p)    [Q](https://www.techopedia.com/it-terms/q)    [R](https://www.techopedia.com/it-terms/r)    [S](https://www.techopedia.com/it-terms/s)    [T](https://www.techopedia.com/it-terms/t)    [U](https://www.techopedia.com/it-terms/u)    [V](https://www.techopedia.com/it-terms/v)    [W](https://www.techopedia.com/it-terms/w)    [X](https://www.techopedia.com/it-terms/x)    [Y](https://www.techopedia.com/it-terms/y)    [Z](https://www.techopedia.com/it-terms/z)

Copyright © 2020 Techopedia Inc. - [Terms of Use](https://www.techopedia.com/about/termsofuse) - [Privacy Policy](https://www.techopedia.com/about/privacypolicy)

In our project we are focusing on lexical analysis.

A lexical analyzer is used to produce tokens from a stream of input string characters, which are broken into small components to form meaningful expressions. A token is the smallest unit in a programming language that possesses some meaning. Thus, in this process, the source code is converted into a series of tokens. This series of tokens is called token stream.



The output of the lexical analysis, performed by lexical analyzer is then given to syntax analyzer for further parsing process.

In order to understand parsing and lexical analysis in more detail, we also studied some articles related to parsing. We learnt about different types of parsing. Tolerant parsing is a kind of parsing in which we apply the ability to capture points of interest inside the program, while all the code that does not contain such points can be skipped with no or minimal analysis performed [2]. Recursive Descent parsers use GLL parsing techniques [4]. GLL is a fully general recursive descent-like parsing technique which follows grammar rules which supports even left recursive grammars. Another paper introduced an interpreter for recursive transition networks(RTN) grammar which students from AI field use and are acquainted with [5]. This article introduces a new concept of parsing using parser combinators [3]. It discusses about a higher-order function that takes one or more parsers as input and produces a new parser as its output.

Thus, after understanding more and getting a clear idea of what parsing means, we approached towards implementing our project. Our project, thus intends to implement lexical analysis. Our aim is to get a token stream from an input file. We will categorize these tokens and identify them as well. In our project, we will also check syntax errors from a file, especially parenthesis. We will check whether there are unmatched parenthesis or not.

.

## METHOD/ IMPLEMENTATION

Our objective in this project, is to implement lexical analysis part of parser and check syntax errors from our input file and generate error if present. Though, In syntactical analysis, we try to check syntax errors using token stream, we wanted to first, be able to check syntax errors from a normal input file. Hence, we included this part in our project of generating syntax errors from the input file, in order to take a small step towards syntactical analysis as well.

Focusing on lexical analysis, our project was divided into these sections :

* To develop easy parsing tool which read source file.
* Remove comments and spaces and separate tokens from it.
* Categorize tokens using regex search function into keyword, Integer constant, Float constant categories.
* Verify the tokens and lastly, check syntax of input file and generate error accordingly.
* Check for unmatched parenthesis syntax error using stack balance method.

We implemented our project using 2 approaches. We had an initial approach, which did not give us results. Thus, we implemented the second approach, to get our project running successfully. We will first discuss about our 1st approach and then talk about our final implementation of the project.

**Initial approach:**

The difference between our initial and final approaches are that initially, we tried to convert our source code into a single string variable. This string was then used, to implement our remaining project, and to detect and identify tokens. But, we encountered trouble in separating tokens from each other and verifying them. We could detect tokens but tokens in the next line of the file were getting concatenated with the ones from the previous line. Thus, were not able to verify them too. This is the reason why we resorted to a different approach altogether.

We will discuss the methods which were used for our initial approach. We implemented the parser.cpp file as the main file for our project.

**Parser.cpp:**

We started our implementation in parser.cpp file where we have RXEng class; Compiler class and main function.

In RXEng class, we define regex method using regex\_search function which uses pattern string to find pattern from input string and return bool value according to search result.

In regex\_search (s, m, e) function, we need three parameters:

String s: It is an input of the function where pattern will be found.

smatch m: It is a regex object where we store our found pattern.

regex e: It is a regex object which define pattern which need to be find.

We, then defined five functions:

bool CheckKeyWord (uint32\_t start, uint32\_t end) {}

bool CheckintCont (uint32\_t start, uint32\_t end) {}

bool CheckfloatCont (uint32\_t start, uint32\_t end) {}

bool CheckSpecialChar (uint32\_t start, uint32\_t end) {}

bool CheckWhiteSpace (uint32\_t start, uint32\_t end) {}

All these methods take starting and ending position as an input to subtract the string for regex\_search function. Output of these methods return bool value which give “true” if pattern will be present in file otherwise “false”.

CheckKeyWord function reads the dictionary text file which has all c++ keywords and then, searches the input string in that dictionary and give bool value in return. We also create start and end integers to track token starting and ending position which, we want to use in our regex method. Initial start value is defined 0 and end value defined 1. We also defined RXEng object r.

We create **for loop** to read string variable from starting to end by incremental of one iterator. In that we first put “if” condition which is for stat position of token. The loop continues until first character is found. We first tried with r.CheckWhiteSpace(start,end) method but as we got errors with it, we used simple space, line and tab character in “if” condition. The “if” condition moves starting and ending point till it cannot find any character. After finding first character, it locks the start position and checks end position with another “ if” condition and continues the “for” loop till it doesn’t find next space. Once the end position finds space, it will break “ for “ loop, thus, giving start and end position of first token. Then we passed that token through all our regex method to find their category.

After categorizing first token, we wanted to verify next token syntax using another similar for loop where we ended up with failure.

Pseudocode of our idea in this development is:

**for loop (read string from start to end)**

**if (find first token)**

**categorize using regex\_methods (start, end)**

**if (first token is in this category)**

**for (read from first token end to main string end)**

**if (next token found)**

**verify next token using if/else condition**

**if(true): continue.**

**else: throw error with position; break.**

**end**

**end**

**basicremovespace.txt:**

In this program, we tried a method to remove comments and blank space from source which we want to apply in our main.cpp file. This method could help us to remove unwanted comments which are not helpful for compiler. We also want to remove blank space to avoid unwanted memories access.

It has two function:

string removeComments (string prgm) {}

void removeSpaces(char \*str){}

removeComments function has two bool flags s\_cmt and m\_cmt which track comment starting and ending condition and using these flags we found comment section that we can subtract from source code before start implementation of lexical analysis.

removeSpaces function is counting all characters except space and the remove space from string.

**Syntaxerror1.txt:**

This code is for syntax errors like unmatched parentheses, brackets and braces. It has three function:

void incomment()

void inqote(int c)

void search(int c);

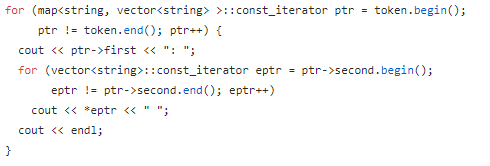
void incomment() function is used to find comment starting from “/\*” to “\*/” using getchar().

void inquote(int c) is read “\\” comments.

voidsearch() is for check parentheses, brackets and braces by tracking their opening and closing counts.

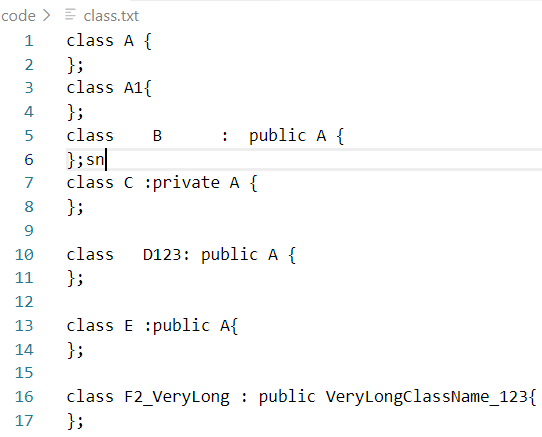
**The final approach:**

We have used main.cpp file for later approach. In main function, first we read input source file and then, instead of converting the whole input file into a string, we create a single string variable named **b** for every word that we encounter in the file, thus detecting tokens. We used RXEng object r in here as well. We use token.begin() and token.end() functions for tracking the tokens. Thus,Token categorizing method is successfully implemented in main.cpp file using same RXEng class methods. Input source file read word by word using while loop and those words are passed to all RXEng regex\_search function, to identify tokens and categorize them by storing them in map<string,vector<string>>. We use keyword.txt file for our token dictionary, from where we identify our tokens from the input file.



**Class\_parsing.cpp:**

In this code we implement parsing for classes. It has class identify function and also identify Inheritance classes. Using regex function “regex\_search(s,m,e)”.



**removeComments.cpp:**

We used only removeComments function from basicremovesspace.txt, as we did not need to remove spaces in this approach.

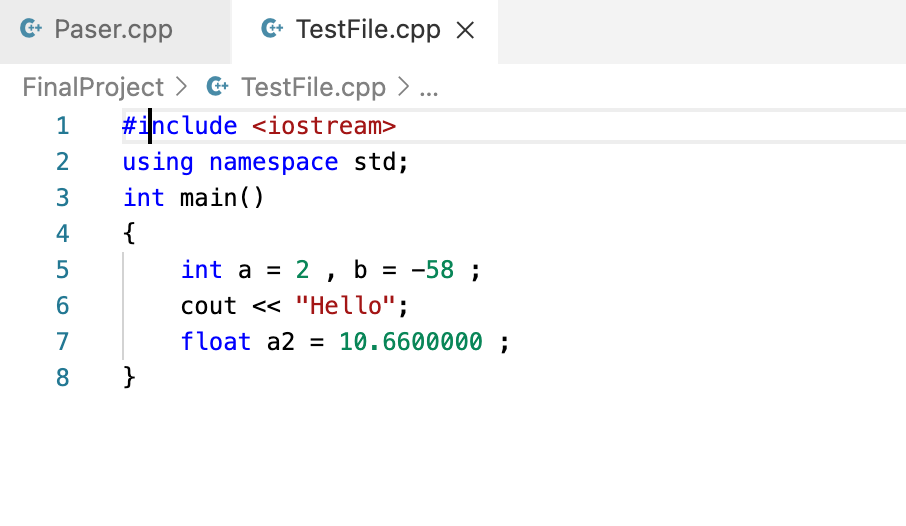




**syntaxerror.cpp:**

It is the same file as syntaxerror1.txt file, but we have removed the incomment() function , as we don’t need it.

In order to implement our whole project, we need to merge main.cpp file with removeComment.cpp and syntaxerror.cpp file to fully implement our project. We have not included the functions in these two files yet, as they were based on our initial approach. We need to find a way to merge these files with our main.cpp file, for the whole project to work properly. We use a TestFile.cpp to test our project and works as an input file.

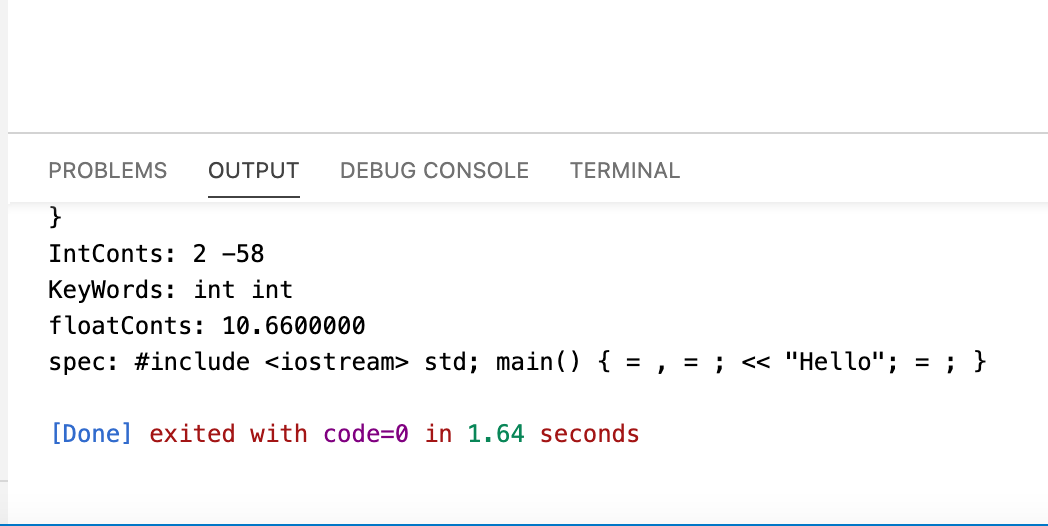


Github info: We have main.cpp, removeComments.cpp and syntaxerror.cpp files in “code” folder which represents the actual and final implementation of the project.

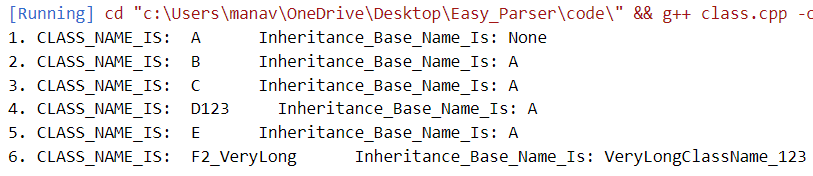
To show our initial approach work, We decided to create “Sample\_Study\_Code” folder where we have kept all our previous work as well as trial codes. Trial codes include: different approaches for different methods such as different input source files, reading source file method, class parsing, token position tracking, syntax error hierarchy loop, parenthesis method, comment and blank space removing method, etc.

## RESULTS

We did test our main.cpp file on our TestFile.cpp, to check the output of our project. We could successfully, categorize the tokens and identify them. We could also parse classes. We definitely reduced the complexities of our project by using the second approach.

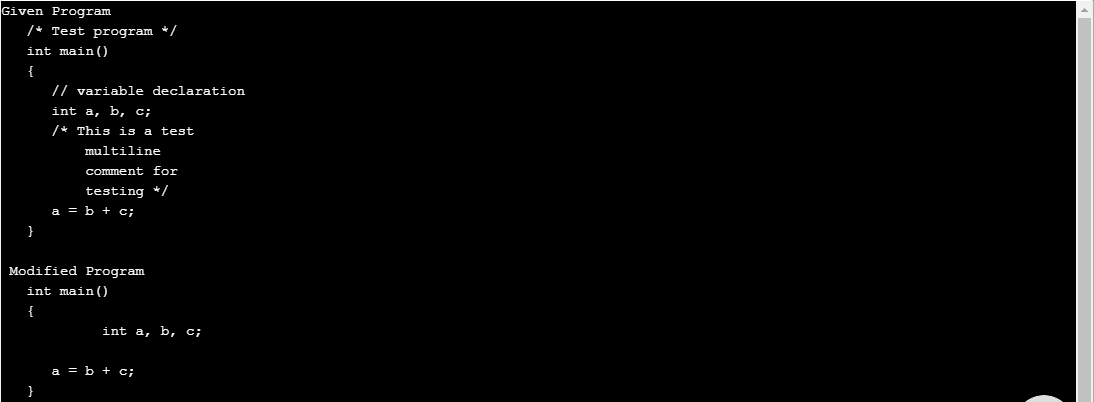
Output of **main.cpp**: 

Output of **Class\_parsing.cpp**:



We implemented removeComments.cpp file separately to get result as well. It successfully removes comments from a given program.

Output of **removeComments.cpp**:



## CONCLUSION

To summarize, we implemented our project through trial and error basis. We implemented our project using 2 approaches, first, by converting the whole source code into a string. Later, we converted each word into a string from our input file to track tokens and used token.begin() and token.end() functions for token categorization. We used regex\_match function for matching and identification purpose of tokens from our keyword.txt file. acting as a token dictionary.

We need to merge our removecomment.cpp and syntaxerror.cpp file with main.cpp file , for full implementation of our project. All the files work individually. Hence, our future work, is to merge them. We also can fully implement checking syntactical error from a file, as we focused on parenthesis, single and double quotes error only. We need to analyze our code further in terms of complexities.

##### REFERENCES

1. Matthew MacDonald, “Get to know VB .NET’s XML objects for easy parsing”, June 2003
2. A.V.Goloveshkin and S.S Mikhalkovich, “Tolerant parsing with a special kind of <<Any>> symbol: the algorithm and practical application”, 2016
3. Jan Kurs, Jan Vrany, Mohammmad Ghafari, Mircea Lungu and Oscar Nierstrasz, “Efficient parsing with parser combinators”, Dec. 2017
4. Elizabeth Scott and Adrian Johnstone, “GLL Parsing”, 2010
5. Robert M. Harlan and David M. Patrone, “ An Easy-to-Understand RTN Intrepreter”, Sept. 1993
6. <https://www.geeksforgeeks.org/introduction-of-lexical-analysis>
7. <https://www.youtube.com/watch?v=VGgIZl5WjH0>
8. <http://www.swig.org/article_cpp.html>
9. [Compiler Design - Lexical Analysis - Tutorialspoint](https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm#:~:text=Lexical%20analysis%20is%20the%20first%20phase%20of%20a%20compiler.&text=If%20the%20lexical%20analyzer%20finds,syntax%20analyzer%20when%20it%20demands.)
10. <https://www.techopedia.com/definition/3854/parser#:~:text=A%20parser%20is%20a%20compiler,by%20other%20components%20in%20programming>.
11. <https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Style_Conventions/Comments#:~:text=Single%2Dline%20comments%20>(informally%2C,%2F\*%20and%20end%20with%20\*%2F%20.
12. <https://www.guru99.com/compiler-design-lexical-analysis.html>