
Syntax Highlighter (C++, C, JS)

José Carlos Martínez Núñez



June 6, 2022

Syntax Highlighter (C++, C, JS)

This is a syntax-highlighter was made for the class Implementation of Computational Methods, using FLEX (fast lexical analyzer generator) and C++.

Author: [José Carlos Martínez Núñez](#) | [A01639664](#)

Table of Contents

- Syntax Highlighter (C++, C, JS)
 - Table of Contents
 - Lexical Categories
 - Color Schemes
 - * Dracula
 - * Rainbow
 - Installation
 - Usage/Examples
 - The implemented algorithms and their execution time
 - Screenshots
 - Time Complexity
 - * How does FLEX work?
 - * Other Algorithms
 - The SpeedUp by using multiple threads
 - * Using One Thread
 - * Using 8 Threads
 - * Calculating the SpeedUp
 - Ethical Implications that this type of technology could have on society
 - * Lexical Analysis
 - * Multiprocessing / Parallel Programming
 - Acknowledgments
 - References

Lexical Categories

The Lexical Categories supported by this program are:






- Preprocessor Keywords
- Reserved Words
- Types
- Operators
- Booleans
- Grouping Characters
- Multi Line Comments
- Single Line Comments
- Strings
- Function Names
- Class Identifiers
- Identifiers
- Package Names







These are all matched with their respective regular expressions in the **Lexer.l** file.

Color Schemes













The syntax-highlighter supports two color schemes for each lexical category:

Dracula

Token	Color
Preprocessor Keywords	 #ff79c6
Reserved Words	 #ff79c6
Types	 #ff79c6
Operators	 #ff79c6
Booleans	 #bd93f9
Grouping Characters	 #ffb86c
Multi Line Comments	 #6272a4

Token	Color
Single Line Comments	 #6272a4
Strings	 #f1fa8c
Function Names	 #50fa7b
Class Identifiers	 #8be9fd
Identifiers	 #f8f8f2
Package Names	 #f1fa8c

Rainbow

Token	Color
Preprocessor Keywords	 #00e9b0
Reserved Words	 #feb300
Types	 #306cc9
Operators	 #00ff00
Booleans	 #18b646
Grouping Characters	 #e8002e
Multi Line Comments	 #81800c
Single Line Comments	 #ffff00
Strings	 #00aeae
Function Names	 #65ecb1
Class Identifiers	 #e4f4df
Identifiers	 #ba00ff

Token	Color
Package Names	 #903a47

Installation

```
# Generate Lexical Analyzer
flex Lexer.l
# Compile Generated Analyzer with the main.cpp
g++ -pthread -std=c++17 Lexer.cpp main.cpp -o "syntax-highlighter"
```

Usage/Examples

```
./syntax-highlighter [FILE | DIRECTORY]
```

The output file will be saved in `output/[FILE | DIRECTORY].html`.

The implemented algorithms and their execution time

Once the program opens a file, the lexical analyzer will loop through each character, once a match is found the resulting string will be HTML encoded, once again looping through each matched character and replacing every HTML specific symbol (<, >, etc.) with their respective alternatives. Then depending on the selected color scheme it'll assign a color to the specific matched string and write it to the output file.

The output of the program is an html document with the lexical categories in their respective colors. The console output will be the execution time of the program measured in milliseconds.

Note: The execution time of the program will vary depending on your computer specs.

Screenshots

The following example files are located in the **examples** folder.

```
Tecnológico de Monterrey
José Carlos Martínez Núñez
A01639664

// Simple Express.js Server
const express = require("express");
const PORT = process.env.PORT || 3001;
const app = express();

app.get("/api", (req, res) => {
  res.json({ message: "Hello from server side!" });
});

app.listen(PORT, () => {
  console.log(`Server listening on ${PORT}`);
});
```

```
Tecnológico de Monterrey
José Carlos Martínez Núñez
A01639664

// Simple Express.js Server
const express = require("express");
const PORT = process.env.PORT || 3001;
const app = express();

app.get("/api", (req, res) => {
  res.json({ message: "Hello from server side!" });
});

app.listen(PORT, () => {
  console.log(`Server listening on ${PORT}`);
});
```

“example.js” Execution Time 2 milliseconds

```
Tecnológico de Monterrey
José Carlos Martínez Núñez
A01639664

// Leap Year Checker
#include <iostream>
using namespace std;

int main() {
  int year;
  cout << "Enter a year: ";
  cin >> year;

  // leap year if perfectly divisible by 400
  if (year % 400 == 0) {
    cout << year << " is a leap year." ;
  }
  // not a leap year if divisible by 100
  // but not divisible by 400
  else if (year % 100 == 0) {
    cout << year << " is not a leap year." ;
  }
  // leap year if not divisible by 100
  // but divisible by 4
  else if (year % 4 == 0) {
    cout << year << " is a leap year." ;
  }
  // all other years are not leap years
  else {
    cout << year << " is not a leap year." ;
  }

  return 0;
}
```

```
Tecnológico de Monterrey
José Carlos Martínez Núñez
A01639664

// Leap Year Checker
#include <iostream>
using namespace std;

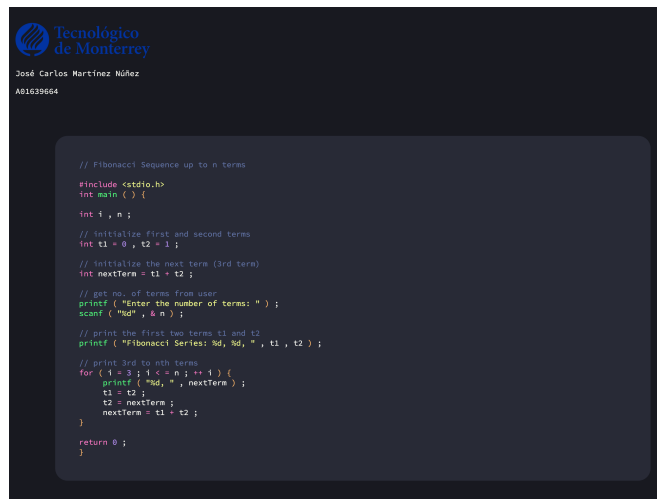
int main() {
  int year;
  cout << "Enter a year: ";
  cin >> year;

  // leap year if perfectly divisible by 400
  if (year % 400 == 0) {
    cout << year << " is a leap year." ;
  }
  // not a leap year if divisible by 100
  // but not divisible by 400
  else if (year % 100 == 0) {
    cout << year << " is not a leap year." ;
  }
  // leap year if not divisible by 100
  // but divisible by 4
  else if (year % 4 == 0) {
    cout << year << " is a leap year." ;
  }
  // all other years are not leap years
  else {
    cout << year << " is not a leap year." ;
  }

  // leap year if not divisible by 100
  // but divisible by 4
  else if (year % 4 == 0) {
    cout << year << " is a leap year." ;
  }
  // all other years are not leap years
  else {
    cout << year << " is not a leap year." ;
  }

  return 0;
}
```

“example.cpp” Execution Time 1 milliseconds



```

// Fibonacci Sequence up to n terms
#include <stdio.h>
int main ( ) {
    int i , n ;

    // initialize first and second terms
    int t1 = 0 , t2 = 1 ;

    // initialize the next term (3rd term)
    int nextTerm = t1 + t2 ;

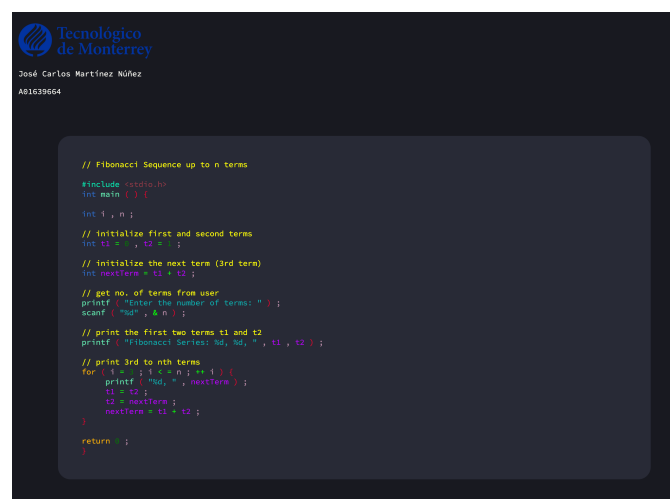
    // get no. of terms from user
    printf ( "Enter the number of terms: " ) ;
    scanf ( "%d" , &n ) ;

    // print the first two terms t1 and t2
    printf ( "Fibonacci Series: %d, %d, " , t1 , t2 ) ;

    // print 3rd to nth terms
    for ( i = 3 ; i <= n ; ++ i ) {
        printf ( "%d, " , nextTerm ) ;
        t1 = t2 ;
        t2 = nextTerm ;
        nextTerm = t1 + t2 ;
    }

    return 0 ;
}

```



```

// Fibonacci Sequence up to n terms
#include <stdio.h>
int main ( ) {
    int i , n ;

    // initialize first and second terms
    int t1 = 0 , t2 = 1 ;

    // initialize the next term (3rd term)
    int nextTerm = t1 + t2 ;

    // get no. of terms from user
    printf ( "Enter the number of terms: " ) ;
    scanf ( "%d" , &n ) ;

    // print the first two terms t1 and t2
    printf ( "Fibonacci Series: %d, %d, " , t1 , t2 ) ;

    // print 3rd to nth terms
    for ( i = 3 ; i <= n ; ++ i ) {
        printf ( "%d, " , nextTerm ) ;
        t1 = t2 ;
        t2 = nextTerm ;
        nextTerm = t1 + t2 ;
    }

    return 0 ;
}

```

“example.c” Execution Time 1 milliseconds

Time Complexity

How does FLEX work?

Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. It is a computer program that generates lexical analyzers (also known as “scanners” or “lexers”).

A Flex lexical analyzer usually has time complexity $O(n)$ in the length of the input. That is, it performs a constant number of operations for each input symbol.

Other Algorithms

Apart from the code generated by the lexical analyzer once a token is matched we run another $O(n)$ algorithm to remove any HTML special characters.

This means that the total time complexity of the whole program is $O(n^2)$.

Nevertheless, as seen in the last section the program runs fairly fast.

The SpeedUp by using multiple threads

For these tests a total of **898 files** were used (located at /examples folder) with a total of **320 different subfolders** divided among the three languages JavaScript, C and C++.

Using One Thread

```
File: "examples/c/C/sorting/shaker_sort.c"
Finished processing in 1336 microseconds.
File: "examples/c/C/sorting/pigeonhole_sort.c"
Finished processing in 1781 microseconds.
File: "examples/c/C/geometry/vectors_3d.c"
Finished processing in 10516 microseconds.
File: "examples/c/C/geometry/geometry_datatypes.h"
Finished processing in 1942 microseconds.
File: "examples/c/C/geometry/quaternions.c"
Finished processing in 14037 microseconds.
File: "examples/c/C/graphics/spirograph.c"
Finished processing in 48439 microseconds.
File: "examples/c/C/games/naval_battle.c"
Finished processing in 603927 microseconds.
File: "examples/c/C/games/tic_tac_toe.c"
Finished processing in 224974 microseconds.
File: "examples/c/C/greedy_approach/prim.c"
Finished processing in 16030 microseconds.
File: "examples/c/C/greedy_approach/dijkstra.c"
Finished processing in 2642 micr
oseconds.
File: "examples/c/selectionsort.c"
Finished processing in 2029 microseconds.
Done!
Finished in 9313 milliseconds.
```

Figure 1: Converting all files located in /examples using one thread

As seen in the image the total time taken to convert all files was **9313 milliseconds**.

Using 8 Threads

```
File: "examples/c/C/leetcode/src/242.c"
Finished processing in 851 microseconds.
File: "examples/c/C/leetcode/src/206.c"
Finished processing in 743 microseconds.
File: "examples/c/C/leetcode/src/1.c"
Finished processing in 888 microseconds.
File: "examples/c/C/leetcode/src/142.c"
Finished processing in 782 microseconds.
File: "examples/c/C/leetcode/src/35.c"
Finished processing in 1184 microseconds.
File: "examples/c/C/leetcode/src/26.c"
Finished processing in 767 microseconds.
File: "examples/c/C/leetcode/src/520.c"
Finished processing in 2475 microseconds.
File: "examples/c/C/leetcode/src/234.c"
Finished processing in 1139 microseconds.
File: "examples/c/C/leetcode/src/701.c"
Finished processing in 939 microseconds.
File: "examples/c/C/leetcode/src/215.c"
Finished processing in 1004 microseconds.
File: "examples/c/C/leetcode/src/66.c"
Finished processing in 2059 microseconds.
Done!
Finished in 4634 milliseconds.
```

Figure 2: Converting all files located in /examples using eight thread

As seen in the image the total time taken to convert all files was **4634 milliseconds**.

Calculating the SpeedUp

The SpeedUp of a program is calculated by the following formula:

$$S_p = \frac{T_1}{T_p}$$

Where:

- p is the number of processors (or cores)

- T_1 is the time taken to execute the single processor version of the program
- T_p is the time taken to execute the multi-processor version of the program using p processors
- Lastly, S_p is the SpeedUp obtained by using p processors.

Using the above formula we can calculate the **SpeedUp** using a total of **8 threads**.

$$S_p = \frac{9313\text{ms}}{4634\text{ms}} \approx 2$$

We can conclude that by using **8 threads** we practically **doubled** the speed of our program.

Ethical Implications that this type of technology could have on society

Lexical Analysis

This type of technology has many uses apart from creating syntax highlighters, technologies like Flex allow us to create lexical analyzers that can be used for all types of purposes from creating our own interpreters for other programming languages to automating lots of processes that require identification of tokens. One example of how these technologies can help society is analyzing laws to fix ambiguous wording or in the research field to analyze dead languages from our past so that understand more about other civilizations. Performing all these tasks with the help of computers will greatly improve the amount of time it'll take to do this manually. It is incredibly important that technologies like these are used for good and not personal gain.

Multiprocessing / Parallel Programming

This technology has incredible implications on society as it can basically speed up most actions that take a long time in all programming applications. Using the example stated above this type of technology can seriously speed up the process of analyzing languages, making it so that we can perform the advances stated above orders of magnitude faster, taking advantage of all the computer power at our disposal.

Acknowledgments

The following GitHub repos were used as example code for this program:

- [@mandliya/algorithms_and_data_structures](#)
- [@TheAlgorithms/C](#)
- [@trekhele/javascript-algorithms](#)

References

- Funchal, G. (2011, April 14). Most efficient way to escape XML/HTML in C++ string?. Stack Overflow. <https://stackoverflow.com/a/5665377>
- Levine, John R.; Mason, Tony; Brown, Doug (1992). *lex & yacc* (2nd ed.). O'Reilly. p. 279. ISBN 1-56592-000-7. A freely available version of lex is flex.