

```

4 # Prevent database truncation if the environment is production
5 abort("The Rails environment is running in production mode!") if Rails.env.production?
6 require 'spec_helper'
7 require 'rspec/rails'
8
9 require 'copybara/rspec'
10 require 'copybara/rails'
11
12 Copybara.java_driver = :webkit
13 Category.delete_all; Category.create
14 Shoulda::Matchers.configure do |config|
15   config.integrate do |with|
16     with.test_framework :rspec
17     with.library :rails
18   end
19 end
20
21 # Add additional requires below this line. Note: require is not needed.
22
23 # Requires supporting ruby files with support for ActiveSupport
24 # spec/support/ and its subdirectories. Files starting with "rspec" are auto-loaded.
25 # in .spec.rb will both be required and loaded.
26 # run twice. It is recommended that you do not use "require" in this file.
27 # end with .spec.rb. You can configure the application in this file.
28 # making no the named file in the "spec" directory.
29
30 No results found for 'mongoid'

```

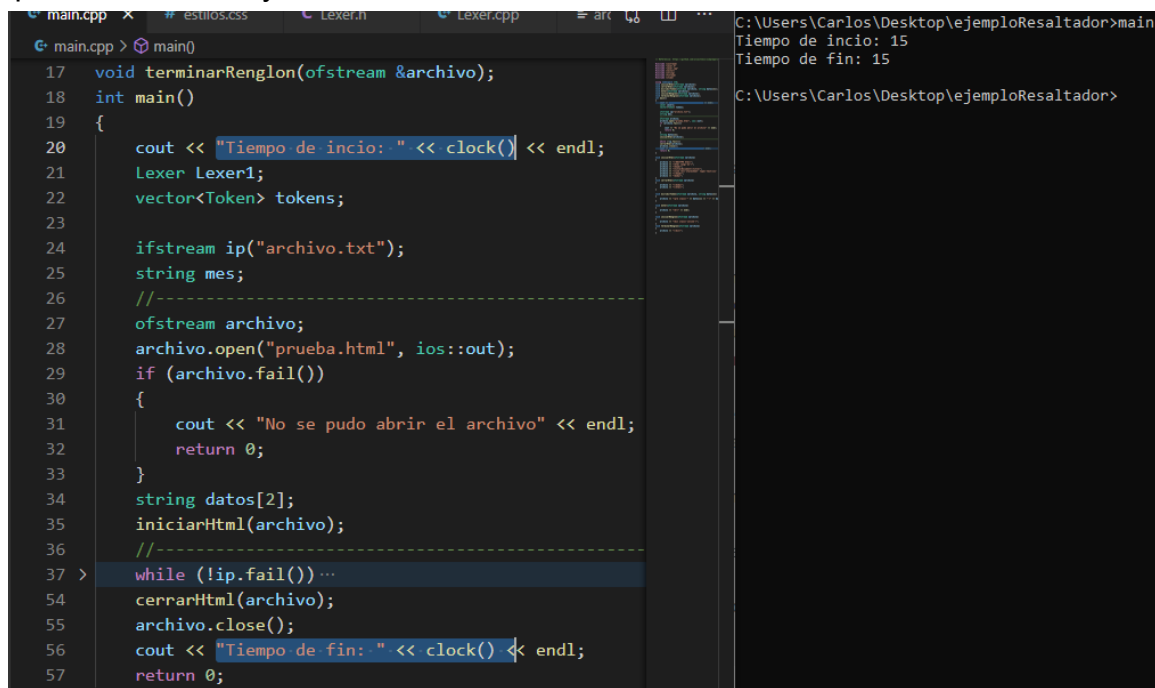
Carlos Estrada Ceballos - A01638214
Abigail Velasco García - A01638095
Natalia Velasco García - A01638047

Reflexión:

Para este problema trabajamos sobre estados para poder generar una solución, dependiendo de los estados actuales y de la próxima entrada es que podemos definir cada uno de los tokens que se generan en la entrada para poder asignarles un valor y mostrar una salida correcta dependiendo de su clasificación, esta solución la pensamos de acuerdo a un diagrama de un autómata finito el cual de igual manera dependiendo un estado actual y de la entrada siguiente podemos determinar el valor para cada token.

Los algoritmos implementados a nuestra consideración son la mejor manera de obtener el valor para cada token, guardamos en una variable temporal el estado actual y esperamos la siguiente entrada para poder determinar el siguiente estado, dependiendo de la entrada podemos determinar si un estado se termina y generamos uno nuevo para un diferente token, dichos tokens los vamos almacenando en un arreglo para que al terminar la lectura de todas las entradas podamos mostrar un resultado completo, las entradas las esperamos con un switch case para poder dar una respuesta dependiendo del valor de la entrada.

Para poder medir nuestros tiempos de ejecución utilizamos la biblioteca ctime, lo que hacemos es definir un tiempo inicial y uno final con el método clock(), al final de la ejecución del programa hacemos la diferencia de ambos tiempos para poder determinar la duración de la ejecución, hicimos la prueba con el código de ejemplo que se nos proporcionó y el resultado fue que nos regresa una diferencia de 0, esto nos quiere decir que el código que creamos está muy bien optimizado, sin embargo aun sabiendo el resultado de la diferencia nosotros sabemos que detrás del proceso si hay un tiempo de espera para generar el resultado solo que este es tan pequeño que incluso es muy difícil de medir.



```
main.cpp x # estilos.css C Lexer.h C Lexer.cpp = art C ...
G main.cpp > main()
17 void terminarRenglon(ofstream &archivo);
18 int main()
19 {
20     cout << "Tiempo de inicio: " << clock() << endl;
21     Lexer Lexer1;
22     vector<Token> tokens;
23
24     ifstream ip("archivo.txt");
25     string mes;
26     //-----
27     ofstream archivo;
28     archivo.open("prueba.html", ios::out);
29     if (archivo.fail())
30     {
31         cout << "No se pudo abrir el archivo" << endl;
32         return 0;
33     }
34     string datos[2];
35     iniciarHtml(archivo);
36     //-----
37 > while (!ip.fail())...
54     cerrarHtml(archivo);
55     archivo.close();
56     cout << "Tiempo de fin: " << clock() << endl;
57     return 0;

```

```
C:\Users\Carlos\Desktop\ejemploResaltador>main
Tiempo de inicio: 15
Tiempo de fin: 15

C:\Users\Carlos\Desktop\ejemploResaltador>
```

Complejidad de nuestro algoritmo

Como ya mencionamos la diferencia en tiempos de ejecución que nos genera es de 0, pero teniendo en cuenta que siempre existe un tiempo de ejecución constante por muy pequeño que sea para cada entrada podemos decir que la complejidad de nuestro algoritmo es orden lineal, ya que depende totalmente del número de entradas que se tenga, y estas entradas al tener un tiempo de ejecución constante nos da como resultado una complejidad lineal.