

## Actividad Integradora 3.4 Resaltador de sintaxis

22 de Abril de 2022

ITESM Puebla

Estudiantes:

Angel Rubén Vázquez Rivera

A01735407

Jorge Angel Delgado Morales

A01551955



### Índice

<b>Lenguaje seleccionado</b>	<b>3</b>
<b>Archivo lexer.xml</b>	<b>3</b>
<b>Definiciones empleadas</b>	<b>3</b>
Key-words	3
Números	3
Puntuadores	3
Alfabeto	3
Llaves	3
Comillas	3
Inicialización (#include)	3
Tipos de datos	3
<b>Reglas definidas</b>	<b>4</b>
Charlist y strings	4
Comentario simple	4
Comentario múltiple	4
Enteros positivos y negativos	4
Números flotantes	4
Números hexadecimales	4
Números binarios	4
Números exponenciales	4
Include	5
Complemento Include	5
Tipo de datos	5
Letras	5
Palabras clave	5
Palabras	5

Salto de línea	5
Tabulación	5
Espacio blanco	6
Llaves	6
Puntuadores	6
Comillas	6
<b>Estilos designados por token</b>	<b>6</b>
<b>Funcionamiento - código</b>	<b>6</b>
<b>Funcionamiento - ejecución</b>	<b>7</b>
<b>Tiempos de ejecución</b>	<b>8</b>
<b>Complejidad</b>	<b>8</b>
<b>Reflexiones</b>	<b>8</b>
Jorge Angel Delgado Morales A01551955	8
Angel Rubén Vázquez Rivera A01735407	8



## Reglas definidas

### Charlist y strings

`\["^"]*\\"\'["^"]*\'` : {token, {string, TokenLine, TokenChars}}.

### Comentario simple

`(\\V)(.)(\\)` : {token, {comment, TokenLine, TokenChars}}.

`(\\V)(.)(["^"])` : {token, {comment, TokenLine, TokenChars}}.

### Comentario múltiple

`(\\*)(.*n?)*(\\V)` : {token, {comentm, TokenLine, TokenChars}}.

### Enteros positivos y negativos

`\\{No}+|\\-{No}+\\. {No}+|{No}+` : {token, {integer, TokenLine, TokenChars}}.

### Números flotantes

`{No}+\\. {No}+` : {token, {float, TokenLine, TokenChars}}.

### Números hexadecimales

`{No}+[x|X]{No}+` : {token, {hex, TokenLine, TokenChars}}.

### Números binarios

`(([0bB][01]+)|([0bB]([01]+_[01]+)+))n?` : {token, {binary, TokenLine, TokenChars}}.

### Números exponenciales

%Exponencial  
`{No}+[e|E]?{No}+` : {token, {exp, TokenLine, TokenChars}}.

%Exponencial  
`{No}+[E|e]?\\{No}+` : {token, {exp, TokenLine, TokenChars}}.

%Exponencial  
`{No}+\\. {No}+[E|e]?{No}+` : {token, {exp, TokenLine, TokenChars}}.

%Exponencial

`{No}+\.{No}+[E|e]?-\{No}+` : {token, {exp, TokenLine, TokenChars}}.

### **Include**

`\#include` : {token, {include, TokenLine, TokenChars}}.

### **Complemento Include**

`\<{Alpha}+\.{Alpha}\>` : {token, {header, TokenLine, TokenChars}}.

### **Tipo de datos**

`int|float|char|string|void|long` : {token, {dataType, TokenLine, TokenChars}}.

### **Letras**

`{Alpha}` : {token, {letters, TokenLine, TokenChars}}.

### **Palabras clave**

`auto|break|case|const|continue|default|do|double|else|enum|extern|for|goto|if|inline|long|register|restrict|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|volatile|while|_Alignas|_Alignof|_Atomic|_Bool|_Complex|_Generic|_Imaginary|_Noreturn|_Static_assert|_Thread_local|_Packed`  
: {token, {keyword, TokenLine, TokenChars}}.

### **Palabras**

`{Alpha}+` : {token, {word, TokenLine, TokenChars}}.

### **Salto de línea**

`\n` : {token, {newLine, TokenLine, TokenChars}}.

### **Tabulación**

`\t` : {token, {tabulacion, TokenLine, TokenChars}}.

## Espacio blanco

\s : {token, {space, TokenLine, TokenChars}}.

## Llaves

\\(|\\)|\\{|\\}|\\[|\\]| : {token, {keys, TokenLine, TokenChars}}.

## Puntuadores







\\%:|%:|\\%>|\\%|\\<%|\\:|\\<:|\\##|\\#|\\,|\\|=|\\^|=|\\&|=|\\>>|=|\\<<|=|\\-=|\\+=|\\%=|\\|=|\\\*=|\\...|\\:|\\=|\\;|\\:|\\?|\\||\\|\\&&|\\|\\^|\\!|=|\\=|=|\\>|=|\\<|=|\\>|\\<|\\>>|\\<<|\\%|\\|\\!|\\~|\\-|\\+|\\\*|\\&|\\-|\\++|\\->|\\.|\\| : {token, {punctuators, TokenLine, TokenChars}}.

## Comillas

\" : {token, {comillaDoble, TokenLine, TokenChars}}.

' : {token, {comillaSimple, TokenLine, TokenChars}}.

## Estilos designados por token

Elemento	Color
Tipo de dato	
Keywords, include	
Integer, float, hexadecimal, binary	
String	
Puntuadores y headers	
Comentarios	

## Funcionamiento - código

1. Se transforma el contenido del archivo a analizar en un string con la función `File.read`, para posteriormente transformarlo a una lista de caracteres con la función `to_charlist`.

2. Se pasa la lista de caracteres que se obtuvo al lexer.
3. Mediante pattern matching, se obtienen los tokens, y se ignora el :ok y número de líneas (primer y tercer elemento de la tupla que regresa el lexer)
4. Se escribe en el archivo el inicio del html como el head
5. Se itera sobre los tokens
  - a. Se analiza cada token
    - i. Si es un salto de línea, se agrega un tag de terminación de un 'li', un salto de línea y un tag de apertura de otro 'li'
    - ii. Si es un comentario de líneas múltiples, se separan todos los elementos con la función String.split, usando como delimitador el salto de línea (\n), y cada línea resultante se agrega en un li (list element)
    - iii. Si es cualquier otro token, se agrega un span cuya clase es la misma que el ID de dicho token y su contenido el charset de este
6. Se agrega el cierre del html

## Funcionamiento - ejecución

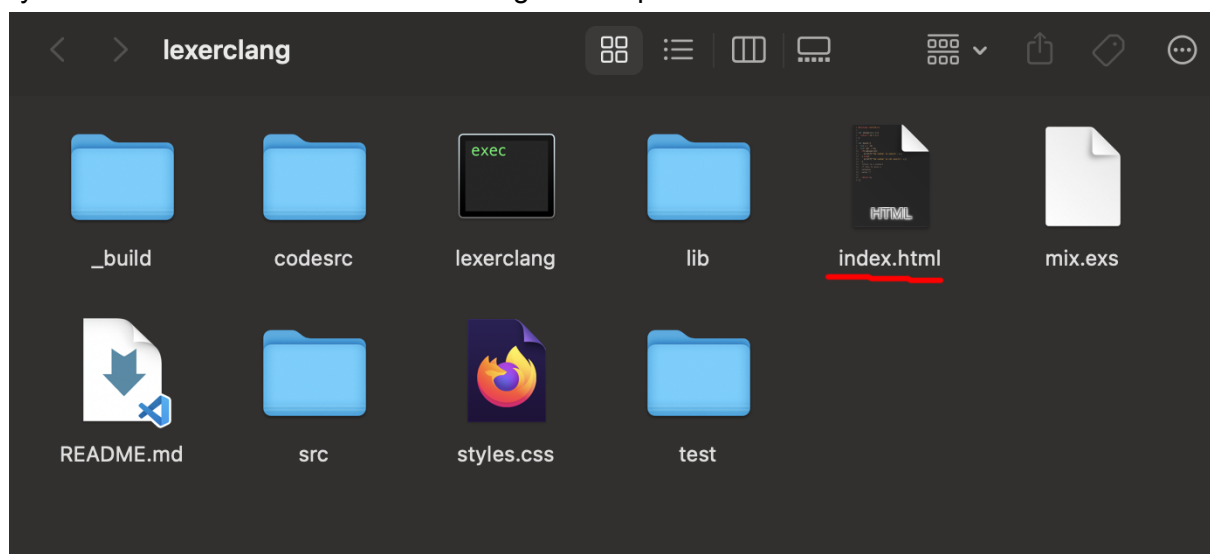
Convertir el archivo a un ejecutable

```
> mix escript.build
Compiling 1 file (.ex)
Generated escript lexerclang with MIX_ENV=dev
```

Correr el ejecutable, este debe de recibir un string con el path donde se encuentra el código de C a ejecutar (path relativo)

```
> escript lexerclang "codesrc/ex1.c"
```

Si el código corrió sin errores, se generará en la carpeta del proyecto un archivo llamado "index.html", el cual contiene la página web para visualizar el código con marcado de syntax. Abrir dicho archivo con su navegador de preferencia.



## Tiempos de ejecución

```
> time escript lexerclang "codesrc/ex1.c"
escript lexerclang "codesrc/ex1.c"  0.13s user 0.11s system 159% cpu 0.153 total
> time escript lexerclang "codesrc/ex1.c"
escript lexerclang "codesrc/ex1.c"  0.13s user 0.08s system 148% cpu 0.141 total
> time escript lexerclang "codesrc/ex1.c"
escript lexerclang "codesrc/ex1.c"  0.13s user 0.10s system 153% cpu 0.148 total
> time escript lexerclang "codesrc/ex1.c"
escript lexerclang "codesrc/ex1.c"  0.14s user 0.10s system 146% cpu 0.159 total
> time escript lexerclang "codesrc/ex1.c"
escript lexerclang "codesrc/ex1.c"  0.14s user 0.11s system 166% cpu 0.148 total
```

## Complejidad

La complejidad de este programa es lineal u  $O(n)$ ; donde  $n$  es el número de caracteres en el código a analizar. Ya que el lexer itera sobre todos los elementos del código fuente para transformarlos en tokens. Posteriormente itera sobre todos los tokens (no de forma simultánea, por lo que la complejidad sigue siendo lineal), y en el peor de los casos tendremos tantos tokens como caracteres. Por cada token se hacen operaciones de tiempo constante u  $O(1)$ , como lo son escribir en el archivo.

## Repositorio GitHub

<https://github.com/jorgedel4/LexerCLang>

## Reflexiones

### Jorge Angel Delgado Morales A01551955

Sin duda con esta actividad pude aprender como la gran importancia que tienen los lexers o analizadores lexicográficos y su aplicación a situaciones de la vida real, en este caso un remarcador de syntax para el lenguaje C. Ya que este nos permite 'tokenizar' una secuencia de caracteres dadas las reglas y definiciones que queramos. Esto es útil ya que el código que realiza todo este análisis es creado de forma automática, lo cual nos ahorra bastante esfuerzo.

Otra cosa que aprendí en general es a pensar de otra manera a la hora de programar, ya que estaba acostumbrado a usar otros lenguajes como C++ y Python, los cuales emplean otros estilos de programación algo diferentes al de Elixir. Cosas como la inmutabilidad y la falta de métodos iterativos en este último lenguaje me hicieron tener que encontrar otras formas de realizar los métodos que usualmente usaría.

Creo que esto también me permitió entender un poco más cómo es que funciona una de las herramientas que uso casi a diario debido a mi carrera, el editor de texto.



## **Angel Rubén Vázquez Rivera A01735407**

Con la presente actividad pude comprender a fondo el funcionamiento de los resaltadores sintácticos de múltiples lenguajes de programación de la actualidad, pues algunos lenguajes exigen que los análisis sintácticos sean más detallados y extensos, dado que en este caso se implementó frente a un lenguaje que si bien fué y es uno de los lenguajes más utilizados en la actualidad, las reglas sintácticas con las que cuenta no son las más minuciosas hablando generalmente, pero aún así representó un reto bastante interesante por resolver, ya que fué necesario implementar definiciones y reglas basadas en expresiones regulares, que requerían de un buen planteamiento y análisis previo a su implementación, pues cada regla debe cumplir con casos específicos del contenido de archivos de código fuente.

Algo muy interesante de esta actividad fué la necesidad que se nos fué expuesta al contar con restricciones como la inmutabilidad en elixir, pues recorrer cada elemento del código exigía nuevas maneras de implementación, además de que las funciones como Enum.each o Enum.reduce resultaban de gran ayuda al agregar tokens a cada uno de los elementos analizados.

Ahora bien, al pensar en los beneficios de este tipo de herramientas como los resaltadores sintácticos existentes en múltiples ambientes en la actualidad, puedo apreciar aún más su ayuda, pues de lo contrario detectar errores o identificar secciones del código contaría con una mayor complejidad y por supuesto que sería más aburrido codificar.