Francisco Rocha Juárez A01730560
Iker Guerrero Gonzalez A00830026

Introducción a lua:
No hay mejor manera de describir cual es el propósito de lua, que aquel que se da en la pagina oficial:

“Lua es un lenguaje de programación extensible diseñado para una programación procedimental general con utilidades para la descripción de datos. También ofrece un buen soporte para la programación orientada a objetos, programación funcional y programación orientada a datos. Se pretende que Lua sea usado como un lenguaje de script potente y ligero para cualquier programa que lo necesite. Lua está implementado como una biblioteca escrita en C limpio (esto es, en el subconjunto común de ANSI C y C++).

Siendo un lenguaje de extensión, Lua no tiene noción de programa principal (main): sólo funciona embebido en un cliente anfitrión, denominado programa contenedor o simplemente anfitrión (host). Éste puede invocar funciones para ejecutar un trozo de código Lua, puede escribir y leer variables de Lua y puede registrar funciones C para que sean llamadas por el código Lua. A través del uso de funciones C, Lua puede ser aumentado para abarcar un amplio rango de diferentes dominios, creando entonces lenguajes de programación personalizados que comparten el mismo marco sintáctico. La distribución de Lua incluye un programa anfitrión de muestra denominado lua, que usa la biblioteca de Lua para ofrecer un intérprete de Lua completo e independiente.”

Esto se puede encontrar en el manual de referencia para lua, en el siguiete enlace: https://www.lua.org/manual/5.1/es/manual.html

Categorías léxicas utilizadas:

1er color (palabras reservadas)

- and
- break
- do
- else
- elseif
- end
- false
- for
- function
- got
- if
- in
- local
- nil
- not
- or
- repeat

- return
- then
- true
- until
- while

2do color (nombres variables/funciones)

3er color (operadores)
- +
- -
- *
- /
- //
- %
- ^
- &
- |
- ~
- >>
- <<
- ==
- ~=
- <
- >
- <=
- >=
- ..
- #

4to color (comentarios)
- – (mas lo que le sigue en la linea)

5to color (delimitadores)
- (
-)
- {
- }
- [
-]

Tipos de variables (un color cada una)
- integer number
- real numbers
- string (entre “ ” o ‘ ‘)

Signos puntuacion:
- .
- ,
- :

Reflexión de la solución planteada:

Francisco Rocha Juárez: Considero que la solución que planteamos fue bastante eficiente, ya que en la función format que creamos para pasar al HTML, nos evitamos enlistar toda la lista de tokens, en su lugar lo logramos con una sola línea de código, que es la siguiente:

```
"<span class=#{token}>#{tchars}</span>"
```

Considero que el código cumple con su función como debería y se aplicaron las expresiones regulares correctamente.

Iker Guerrero González:
Me gusto mucho la manera en la que implementamos nuestra solución, ya que está requirió que pensáramos en cómo es que html funciona y cómo podemos usar eso a nuestra ventaja al momento de hacer el código. En especial considero que nuestra solución para detectar e imprimir saltos de línea fue bastante interesante.

Algoritmos implementados y sus complejidades

- Transformación de expresiones regulares a autómata finito no determinista (lineal al compilar)
- Transformación del autómata no determinista a uno determinista (exponencial al compilar)
- Lectura de los caracteres en el archivo a través del autómata no determinista para hacer las tuplas de tokens (linear)
- Transformación de la tupla de tokens en tuplas conteniendo número de línea y código html (lineal)
- Escritura en el documento html usando las tuplas para escribir los caracteres y saltos de línea (lineal)

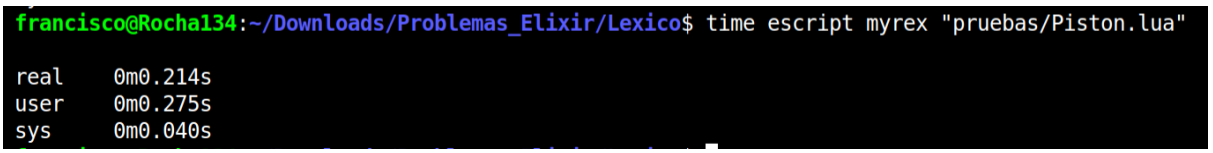
  En conclusión al momento de compilar si tendrá una complejidad exponencial de acuerdo al número de expresiones regulares, pero ya al momento de ejecutar tendrá una complejidad lineal.

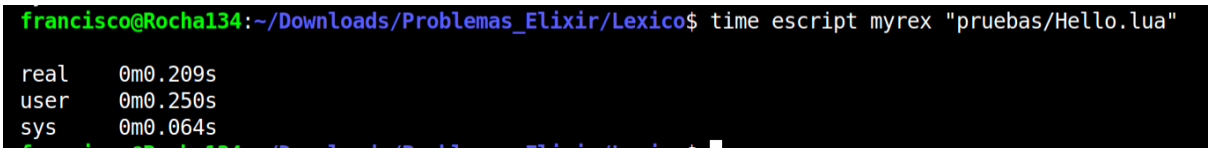
Tiempo de ejecución:

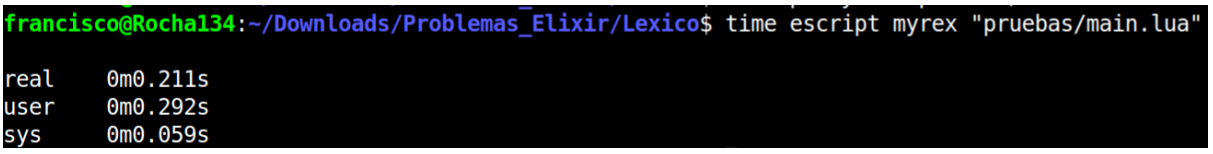
```
francisco@Rocha134:~/Downloads/Problemas_Elixir/Lexico$ time escript myrex "pruebas/Utility.lua"

real    0m0.221s
user    0m0.283s
sys     0m0.058s
```

```
francisco@Rocha134:~/Downloads/Problemas_Elixir/Lexico$ time escript myrex "pruebas/Switch.lua"

real    0m0.194s
user    0m0.276s
sys     0m0.035s
```

```
francisco@Rocha134:~/Downloads/Problemas_Elixir/Lexico$ time escript myrex "pruebas/push.lua"

real    0m0.225s
user    0m0.310s
sys     0m0.061s
```







¿Por qué se dejaron fuera las categorías léxicas avanzadas?

Como en todos los lenguajes de programación, en lua es difícil identificar las categorías léxicas avanzadas solo utilizando pattern matching en los caracteres escritos. Para poder hacer esto es necesario un analizador léxico contextual, el cual está fuera de la metas de este entregable.

Reflexión sobre las implicaciones éticas que el tipo de tecnología que desarrollaste pudiera tener en la sociedad:

Al programar hay que estar consciente que se adquiere un compromiso con todos los usuarios a los que va dirigida la aplicación. Uno como programador debe entregar un programa de calidad que vaya acorde a su ética profesional. Aun si un analizador léxico es en sí una herramienta muy útil para programadores haciendo un programa, también se puede prestar para malos usos, por ejemplo el robo de información. También es posible que el creador del analizador tenga alguna intención maliciosa que implemente en el analizador, como algún script que ejecute acciones no deseadas en el equipo del usuario, sin que este lo sepa. Si nosotros como programadores no somos éticos programando podemos causar grandes daños a otras personas e incluso a la sociedad en sí. Es por esto que es fundamental siempre adherirse al código de ética al momento de realizar cualquier programa.