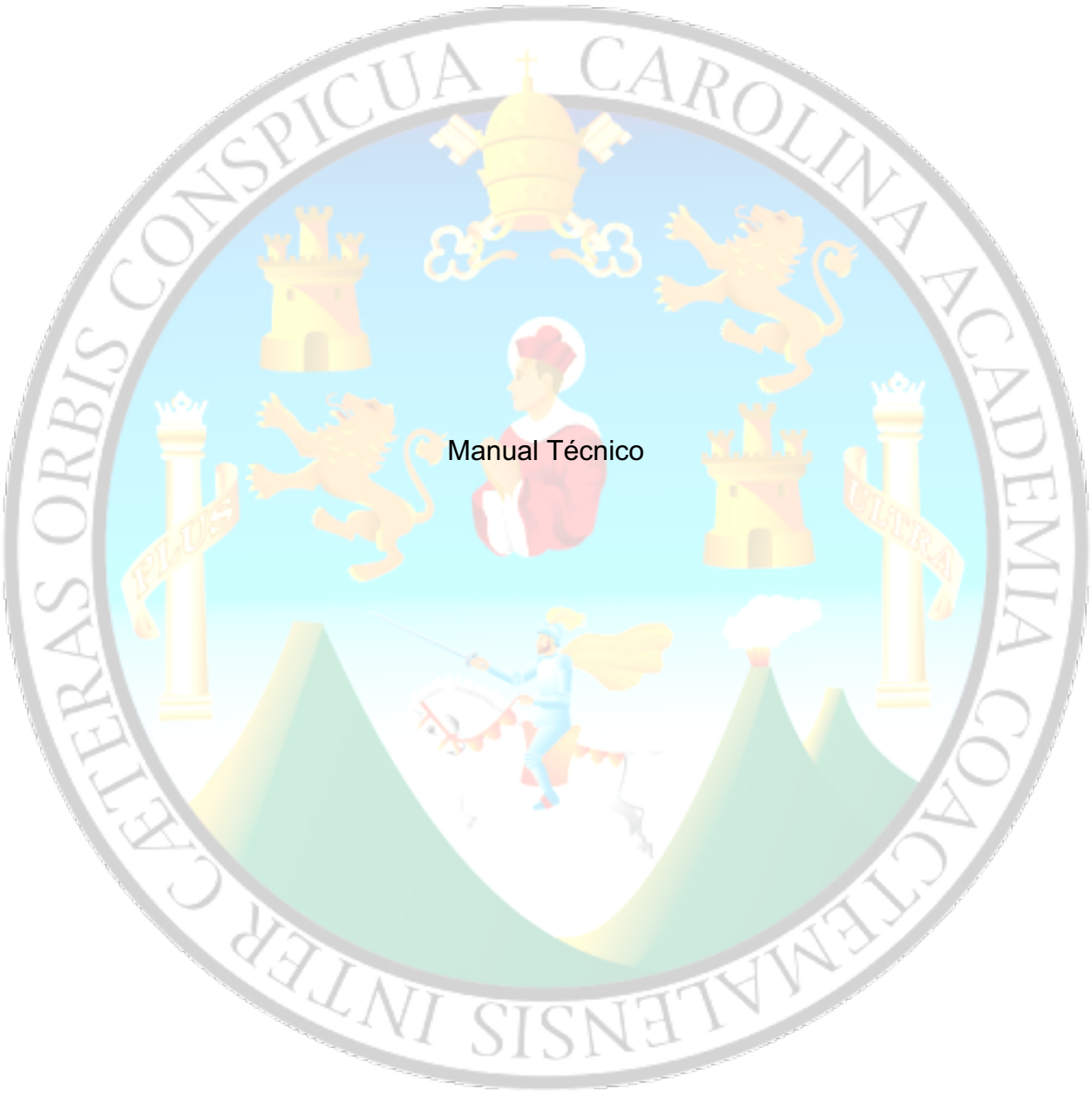
Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores 1
Sección A



Manual Técnico

Carlos Ojani NG Valladares
201801434

Para este proyecto se llevó a cabo un prototipo de detección de errores léxicos para los principales lenguajes orientados a la programación web. JavaScript, CSS y HTML.

El programa consiste en el análisis léxico de estos tres lenguajes, acorde a los archivos de entrada que se le proporciona al programa. Este proceso sirve para reconocer todos esos caracteres que no forman parte de cada uno de estos lenguajes. Por lo que se implementaron tres analizadores para los tres respectivos lenguajes.

Esto se llevo a cabo en el lenguaje de programación Python.

Inicialmente, la aplicación cuenta con una interfaz grafica, amigable para el usuario. Consta de una consola, en donde se mostrarán errores léxicos encontrados y la bitácora de estados del CSS. También de una caja de texto que contendrá el código a analizar y una caja de selección, para elegir cuál lenguaje analizar. Por último, el botón de analizar. A continuación, la interfaz del programa:

Es botón de Scan manda a llamar el scanner del lenguaje que esté seleccionado en la caja de selección. Entre las opciones están:

- JavaScript
- CSS
- HTML
- JS Parser

En términos generales, cada analizador se realizo de manera similar. Esto se logro enviando el texto dentro de la caja de texto como parámetro a un objeto scanner.

Luego, con el manejo de strings dentro de Python se separaron por líneas la entrada y asimismo, por caracteres cada línea. Esto, para llevar a cabo la lectura carácter por carácter e ir concatenando hasta ir “formando” cada palabra o símbolos permitidos en cada lenguaje.

Para ir concatenando los caracteres, el programa se fue guiando de estados. Es decir, dependiendo del estado se iba concatenando el conjunto de palabras, números o símbolos.

Para el manejo de palabras reservadas, en cada uno de los analizadores se crearon listas que contienen todas las palabras que soporta cada lenguaje. Por medio de un método que recibe como parámetro la palabra concatenada o actual, se compara con cada elemento de la lista y retorna el tipo de token y su valor, para posteriormente ser almacenado en una lista de tokens. En caso de no encontrar alguna coincidencia se retorna el token tipo ID. A continuación, se presenta el fragmento de código que realiza dicho trabajo:

```
self.reserved = [
    'function', 'return', 'while', 'for', 'document', 'getElementById', 'if', 'else', 'constructor', 'console', 'log',
    'break', 'true', 'false', 'value', 'new', 'Object', 'push', 'Array', 'appendChild', 'setAttribute',
    'innerHTML', 'innerText', 'element', 'createElement', 'JSON', 'Items', 'stringify', 'clear', 'fromHTML', 'forEach',
    'location', 'href', 'sessionStorage', 'getItem', 'null', 'this', 'Math', 'pow', 'class', 'save', 'ajax', 'parseInt',
    'instanceof', 'default', 'break', 'debugger', 'in', 'void','typeof','try','switch', 'throw', 'catch', 'finally'
]
```

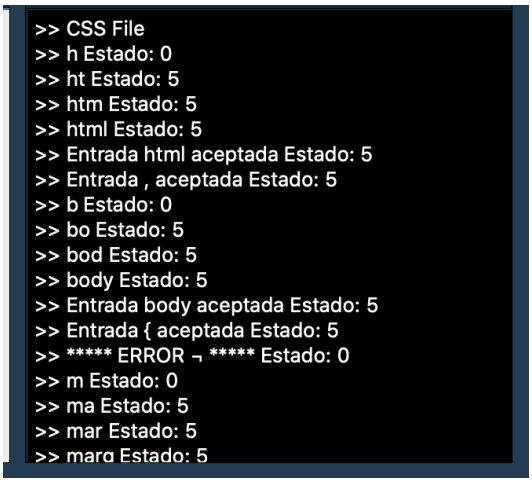
```
def reservedToken(self, word):
    for item in self.reserved:
        if word == item:
            return 'RESERVED_' + item.upper()
    return 'ID'
```

Reporte Bitácora CSS

Este reporte se realizó únicamente guiándose en qué estados iba pasando el analizador. Se generó una lista de transiciones que guarda objetos de tipo transición. Cada transición tiene un estado, valor actual, destino y estatus (Estado final o no).

```
class Transition:
    def __init__(self, value, destiny, state, status):
        self.__value = value
        self.__state = state
        self.__destiny = destiny
        self.__status = status
```

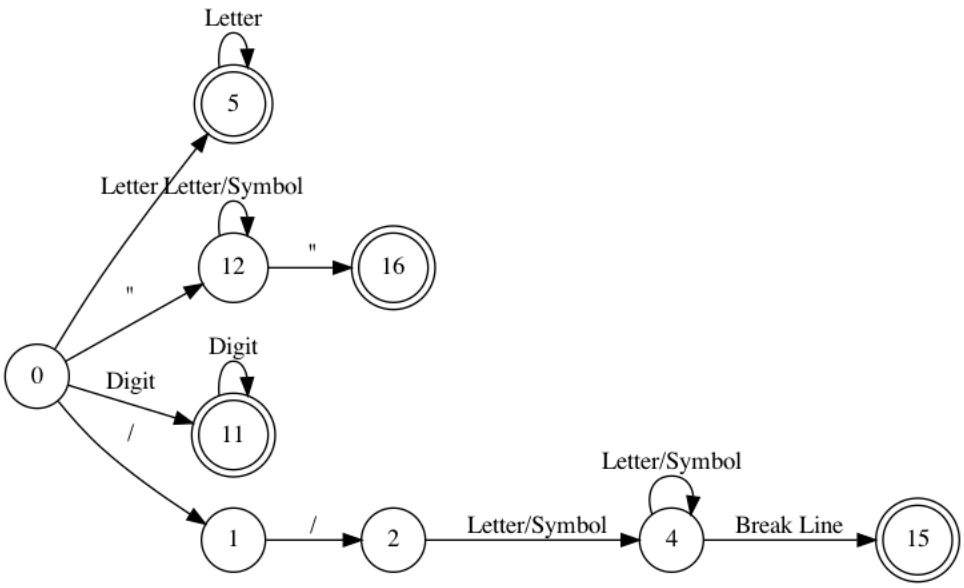
Luego, teniendo a lista de transiciones. Dentro de la consola de la interfaz se fue recorriendo esa lista y pintando en pantalla por cuál estado iba pasando cada entrada aceptada. En caso de encontrar un error, lo pinta. A continuación se presenta el resultado de este reporte:



Reporte Árbol Generado (JS)

De manera similar, se maneja mediante transiciones. Solo que en este reporte, la lista de transiciones es enviada como parámetro a un objeto que grafica los estados y a dónde se dirigen. En este caso, se utilizó la librería Graphviz, ya que permite graficar basándose en la programación orientada a objetos, facilitando la generación del archivo .dot. Dentro de este objeto de tipo Automata, se recorre la lista de transiciones recibida y se generan los nodos (Únicos, por cierto), luego los enlaces de nodos, guiándose de donde empieza y cuál será el nodo destino. Si la transición tiene estatus final, el nodo

destino de la transición se visualizará en un nodo de doble circulo. A continuación. un ejemplo:



Este reporte únicamente se generará cuando se analicen archivos JavaScript.

Reporte de Análisis Léxico

Para los tres lenguajes por igual, se genera un reporte en formato html para los tokens generados y los errores encontrados. Esto se maneja muy sencillo, almacenando los tokens y errores en listas distintas. Luego, recorriendo y a la vez escribiendo en cada lista el texto correspondiente a html. En este caso se almacena la información en tablas y se utilizó Bootstrap para darle estética a los reportes.

Tabla de Errores

Num.	Error
1	La entrada "^" en la fila 47 no pertenece al lenguaje.
2	La entrada "^" en la fila 78 no pertenece al lenguaje.
3	La entrada "`" en la fila 99 no pertenece al lenguaje.
4	La entrada "`" en la fila 99 no pertenece al lenguaje.
5	La entrada "–" en la fila 119 no pertenece al lenguaje.
6	La entrada "¡" en la fila 134 no pertenece al lenguaje.
7	La entrada "%" en la fila 141 no pertenece al lenguaje.
8	La entrada "#" en la fila 149 no pertenece al lenguaje.
9	La entrada "@" en la fila 166 no pertenece al lenguaje.
10	La entrada "$" en la fila 175 no pertenece al lenguaje.
11	La entrada "¿" en la fila 180 no pertenece al lenguaje.
12	La entrada "@" en la fila 192 no pertenece al lenguaje.

Tabla de Tokens

Token	Lexema	Columna	Fila
COMMENT	/********** * ==========* * ==========* * ==========* * ==========ARCHIVO DE PRUEBA DE JS==========* * ==========* * ==========PATHL -> /home/user/output/js/==========* * ==========PATHW -> c:\user\output\js\==========* * ==========* * ==========* * ==========* **********/	1	14
COMMENT	/********** * Dentro de un archivo de tipo javascript * * pueden encontrarse comentarios de tipo * * multilinea o de tipo unilinea, estos * * pueden aparecer en cualquier parte del * * archivo de entrada tomando en cuenta que, * * el primero es el que contiene el path del * * directorio al cual se enviara la salida * * ya analizada y limpiada. * **********/	1	27
RESERVED_FUNCTION	function	1	30
ID	session	2	30
LEFT_PARENT	(	3	30
RIGHT_PARENT	)	4	30
LEFT_BRACE	{	5	30
ID	var	1	32

Parser

Se realizó un analizador sintáctico simple, como introducción al tema. Consiste en analizar una expresión algebraica y validarla, básicamente.

Para realizarlo, se llevo a cabo un procedimiento llamado Para/Match que recibe un no terminal y lo compara con un símbolo de preanálisis. Este símbolo de preanálisis viene de la lista de tokens que obtenemos luego de realizar el análisis léxico de la expresión. Para eso, se utilizó la siguiente grámatica:

```
E -> T EP
EP -> + T EP
    | - T EP
    | EPSILON
T -> F TP
TP -> * F TP
    | / F TP
    | EPSILON
F ->  (E)
    | NUMERO
    | ID
```

Para realizar el análisis por medio del método de parea se realizan los siguientes pasos:

- Para cada no terminal del lado izquierdo de las producciones, se crea un método.
- Para cada no terminal del lado derecho, se hace una llamada al método correspondiente.
- Para cada terminal del lado derecho, se hace un match para validar si lo que viene es lo esperado.
- Se programa el método match, que compara la entrada (el terminal) con el token de preanálisis. De no existir coincidencia se genera un error sintáctico, de lo contrario, se sigue con el análisis, recorriendo el siguiente token de la lista.

A continuación, el método match utilizado en el parser:

```
def match(self, type):
    if type != self.preToken.get_type():
        new_error = Error(self.getError(type))
        self.error_list.append(new_error)
        print("expected", self.getError(type))
    if self.preToken.get_type() != 'LAST':
        self.numPreToken = self.numPreToken + 1
        self.preToken = self.data[self.numPreToken]
```