Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

Febrero-Junio 2023

TC 2037
Implementación de
Métodos Computacionales
Grupo 605

Actividad Integradora 3.4
Resaltador Léxico

Equipo 3
Integrantes:
A01652327 Diego Esparza Hurtado
A00834672 Marco Antonio Rodríguez Amezcua

Profesores:
Dr. Jesús Guillermo Falcón Cardona

Fecha:
7 de mayo de 2023

Analizador léxico:

Para esta actividad se utilizó el lenguaje de programación Python tanto para programar el analizador léxico como para ser utilizado como base para ser analizado por el analizador léxico.

Dentro de la investigación que se realizó, se empleó la documentación oficial de Python para definir las distintas categorías léxicas que existen en el lenguaje. Posteriormente, se definieron las expresiones regulares que identifican a los tokens de dichas categorías.

Luego, con apoyo de la librería 're' se programaron las expresiones regulares en Python para identificar los tokens en un archivo con terminación '.py'. Dentro de la implementación, lo que se realiza es obtener de cada línea del archivo los tokens con el método 'match()' que busca el patrón de la categoría por cada caracter de la línea.

El resultado obtenido es un archivo HTML que también cuenta con cualidades de CSS para mostrar el mismo archivo '.py' ingresado, pero con cada uno de los tokens coloreado con su respectivo color dependiendo de su categoría léxica.

El programa en Python tiene predeterminado el nombre de 'archivo.py' para el archivo que será analizado, pero esto se puede modificar en la línea 119. De igual manera, el archivo '.html' se genera con el nombre: 'Archivo_Salida.html', pero también puede ser modificado en la línea 131. Para correr el archivo se puede utilizar el comando: 'python (nombreArchivo).py', que en este caso sería: 'python Equipo3_Act3_4.py'.

Categorías Léxicas:

1. **Entero:**

$$Entero = decinteger \cup bininteger \cup octinteger \cup hexinteger$$

$$decinteger = (nonzerodigit("\_"digit^{+})^{*}) \cup (0^{+}(\_0)^{*})$$
$$bininteger = 0(\text{b} \cup \text{B})("\_"bindigit^{+})^{+}$$
$$octinteger = 0(\text{o} \cup \text{O})("\_"octdigit^{+})^{+}$$
$$hexinteger = 0(\text{x} \cup \text{X})("\_"hexdigit^{+})^{+}$$
$$nonzerodigit = 1 \ldots 9$$
$$digit = 0 \ldots 9$$
$$bindigit = 0 \cup 1$$
$$octdigit = 0 \ldots 7$$
$$hexdigit = (digit) \cup (a \ldots f) \cup (A \ldots F)$$

2. **Flotante:**

$$Flotante = pointfloat \cup exponentfloat$$

$$pointfloat = (digitpart\ "."\ digitpart) \cup (digitpart\ ".")$$
$$exponentfloat = (digitpart \cup pointfloat)exponent$$
$$digitpart = digit("\_"\ digit^{+})^{*}$$
$$digit = 0 \ldots 9$$
$$fraction = "."\ digitpart$$
$$exponent = \big((e \cup E)(-)digitpart\big) \cup \big((e \cup E)digitpart\big)$$

3. **Imaginario:**

$$Imaginario = (Flotante \cup digitpart)(j \cup J)$$

$$digitpart = digit("\_"\ digit^{+})^{*}$$
$$digit = 0 \ldots 9$$

4. **Delimitador:**

En este caso, la lectura de alguno de estos símbolos se marca como delimitador, es decir, la expresión regular cuenta con la unión de cada uno de estos símbolos. Únicamente el símbolo @ se encuentra tanto en la categoría delimitador como operador, por lo que se le da preferencia en la implementación al operador.

(	)	[	]	{	}	
,	:	.	;	@	=	−>
+=	−=	*=	/=	//=	% =	@ =
& =	| =	^ =	>>=	<<=	**=	

5. **Operador:**

En este caso, la lectura de alguno de estos símbolos se marca como operador, es decir, la expresión regular cuenta con la unión de cada uno de estos símbolos.

+	−	*	**	/	//	%
<<	>>	&	|	^	~	:=
<	>	<=	>=	==	! =	@

6. **Identificador:**

$$Identificador = (a \dots z \cup A \dots Z)(a \dots z \cup A \dots Z \cup 0 \dots 9 \cup "\_")^*$$

7. **Palabras Reservadas:**

Las siguientes palabras son de uso reservado y deben escribirse exactamente de la manera en la que aparecen en la tabla. No pueden ser empleadas como identificadores. La expresión regular es la unión de todas las palabras.

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

8. **Comentario:**

Cualquier elemento del lenguaje que se encuentre luego de un '#' se considerará comentario.

$$Comentario = \#\Sigma^*$$

$$\Sigma = \text{Caracteres aceptados unicode}$$

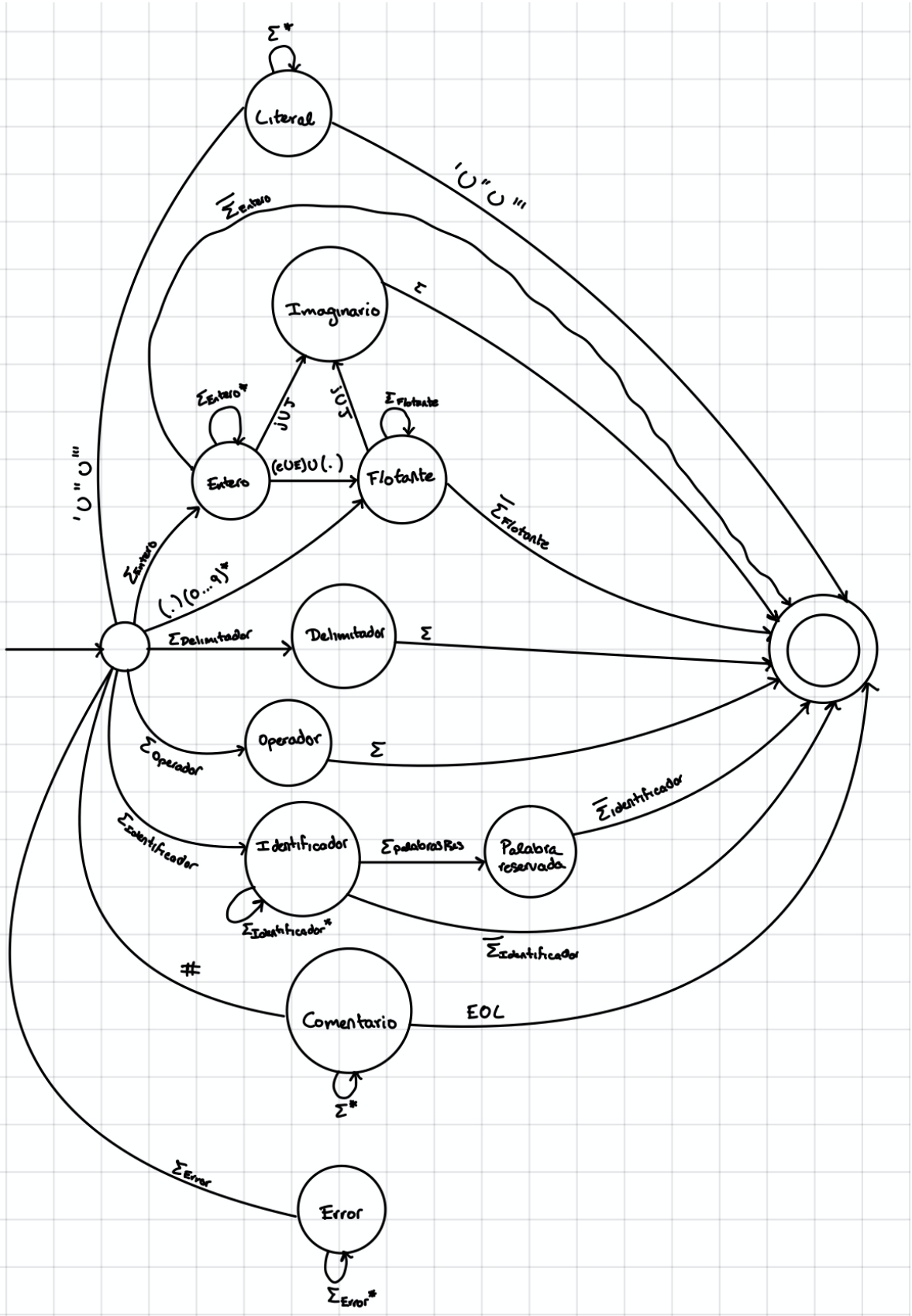
9. **Símbolo error:**

Los siguientes elementos del lenguaje no pueden ser empleados fuera de los literales o comentarios para evitar un error.

$$Error = \$ \cup ? \cup \text{`}$$

10. **Literal:**

$$Literal = ('\Sigma^*') \cup ("\Sigma^*") \cup ('''\Sigma^*''')$$


Σ*
Literal
'∪"∪'''
Σ̄Entero
Imaginario
τ
ΣEntero*
JUJ
JUJ
ΣFlotante*
'∪"∪'''
Entero
(eUE)U(.)
Flotante
Σ̄Flotante
ΣEntero
(.)(0...9)*
ΣDelimitador
Delimitador
Σ
ΣOperador
Operador
Σ
ΣIdentificador
Identificador
ΣpalabrasRes
Palabra reservada
Σ̄Identificador
ΣIdentificador*
Σ̄Identificador
#
Comentario
EOL
Σ*
ΣError
Error
ΣError*

Reflexión:

Para que un código pueda ser ejecutado, es necesario saber que está escrito de la manera adecuada para que una computadora pueda correrlo. En este caso, de aquí deriva la importancia de los analizadores de lenguaje, principalmente un analizador de sintaxis, que valida un orden válido de tokens, por lo que se requiere también de un analizador léxico. Los programas de Python son leídos por el analizador léxico, que a su vez manda los tokens obtenidos al analizador de sintaxis para verificar un correcto orden.

En esta actividad se realizó un analizador léxico que logra identificar los distintos tokens a través del uso de expresiones regulares para los caracteres del archivo que se recibe. Esto se realizó con apoyo de la librería 're' de Python junto con el método '.match()', que busca por cada carácter un patrón de coincidencia, por lo que únicamente revisa cada carácter una sola vez. Esto implica que su complejidad es lineal: O(n), ya que solamente revisa cada carácter una sola vez, es decir, realiza n comparaciones. Luego de obtener una coincidencia con una categoría, se añade el token al archivo HTML con su respectiva etiqueta para colorearse, por lo que la complejidad no se ve afectada.

Marco:

Considero que la función que tiene un analizador de sintaxis es primordial para el entendimiento de la gramática dentro de un lenguaje de programación ya que nos permite checar un token a la vez dentro de la sintaxis de una entrada de datos, de lo contrario los lenguajes de programación serían modelos abstractos y su comprensión no sería del todo lógica. Además, la capacidad expresiva que contienen representa la mayor parte de las construcciones y estructuras presentes en un compilador, en relación con su lectura y proceso de los tokens.

Diego:

En conclusión, dentro de esta actividad se realizó un analizador léxico con complejidad O(n) que permite identificar de manera eficiente con el uso de expresiones regulares los tokens que se encuentran dentro de un archivo de entrada '.py' y se obtiene un archivo '.html' con el contenido del archivo y los tokens coloreados dependiendo de su categoría léxica. Este es el primer paso para poder emplear estos tokens en un analizador de sintaxis y saber si la estructura de las expresiones de un archivo es adecuada para su interpretación y ejecución.

Referencias:

Python. (2023). *Análisis léxico.* Documentación de Python. Recuperado el 15 de abril de 2023. Sitio web: https://docs.python.org/es/3/reference/lexical_analysis.html

Python. (2023). *Re-Operaciones con expresiones regulares.* Documentación de Python. Recuperado el 15 de abril de 2023. Sitio web: https://docs.python.org/es/3/library/re.html