



UNIVERSIDAD DE ANTIOQUIA

Facultad de Ingeniería
Departamento de Ingeniería de Sistemas

2508508 / Teoría de lenguajes y laboratorio

Semestre 2021 - I

Práctica #1 Alejandro Yarce Taborda y Andrés Guerra Montoya

Fecha de entrega: 09/08/2021

INFORME DE ANÁLISIS DEL ANALISIS LEXICO

Índice:

1. Introducción

2. Aspectos generales

3. Autómatas

3.1 Identify

3.2 Keyword

3.3 Operator

3.4 Números Constantes

3.5 Caracteres

3.6 Separadores

4. Expresiones regulares

5. Clases

1. Introducción

Un analizador léxico o analizador lexicográfico (en inglés scanner o tokenizer) es la primera fase de un compilador, consistente en un programa que recibe como entrada el código fuente de otro programa (secuencia de caracteres) y produce una salida compuesta de tokens (componentes léxicos) o símbolos. Estos tokens sirven para una posterior etapa del proceso de traducción, siendo la entrada para el analizador sintáctico (en inglés parser).

En este documento se explicarán los aspectos más relevantes de un analizador léxico y la manera en cómo se hizo uno de manera sencilla, que está basado en las estructuras muy básicas del lenguaje de programación Java; y se hablará de la construcción de los diferentes autómatas que se tomaron en cuenta.

2. Aspectos generales

En este trabajo se ha seleccionado el lenguaje de Java, que junto con javafx se hará el analizador léxico. Ésto solamente abarca una pequeña parte de lo que compone a un compilador, nuestro alcance se limitará a simplemente tokenizar las el código escrito y para ello se hizo un análisis con los autómatas.

3. Autómatas

Un autómata es un modelo matemático para una máquina de estado finito (FSM sus siglas en inglés). Una FSM es una máquina que, dada una entrada de símbolos, "salta" a través de una serie de estados de acuerdo a una función de transición (que puede ser expresada como una tabla). A continuación, los autómatas que se tomarán en cuenta y los tokens que se reconocerán.

3.1 Identify: en esta categoría corresponden cualquier nombre de variable, función o clase creado por el usuario. Las variables, las funciones y las clases deben tener mínimamente una letra, se inicia este nombre con una letra y puede estar seguido de más letras, números o puede tener un guion bajo y luego más letras o números, hasta que llegue un símbolo o un espacio.

	Número	Letra	_	Símbolo o espacio
Vacío	error	I1	error	
I1	I1	I1	I2	Acepte
I2	I1	I1	I2	

3.2 Keyword: en esta categoría están todas las palabras reservadas, como lo son boolean, if, while, for, etc.

[illegible]

INT								
	b	e	i	s	n	t	w	-
vacío	b1	e1	i1	s1			w1	
i1					i2			
i2						i3		
i3								Acepte

[illegible]

3.3. Operator: Básicamente los operadores aritméticos corresponden a las operaciones aritméticas como ejemplo +, -, *, /, %, ==, !=, <=, etc.

	+	-	*	/	%	=	!	Símbolo o espacio
Vacío	+	-	*	/	%	=	!	
+								Acepte
-								Acepte
*								Acepte
%								Acepte
=						==		Acepte
==								Acepte
!						!=		
!=								Acepte

3.4. Números Constantes: Los números enteros son cualquier número que corresponda al conjunto de los números naturales más sus opuestos incluyendo el número cero (0). En otras palabras, los números enteros son los números que empleamos para contar, incluyendo el cero (0), más todos los números opuestos, se escogieron números enteros.

	.	Número	Símbolo o espacio
Vacío	nu2	nu1	
nu1	nu2	nu1	Acepte
nu2		nu2	Acepte

3.5. Caracteres: Las cadenas de caracteres o de tipo String siempre deben empezar y terminar con comillas, de lo contrario no son charConst

	"	Letra	Número	Símbolo	Símbolo o espacio
Vacío	ch1				
ch1	ch2	ch1	ch1	ch1	
ch2					Acepte

3.6. Separadores: cualquier separador cómo (,), {, } y ;

Separator						
	()	{	}	;	Símbolo o espacio
Vacío	()	{	}	;	
()				Acepte
)						Acepte
{				}		Acepte
}						Acepte
;						Acepte

4. Expresiones regulares

Las expresiones regulares son patrones utilizados para encontrar una determinada combinación de caracteres dentro de una cadena de texto. Las expresiones regulares proporcionan una manera muy flexible de buscar o reconocer cadenas de texto. Consideramos utilizar expresiones regulares para facilitar a la hora de hacer el código, el único inconveniente encontrado, que no se resolvió, fue de que cuando varias cadenas estaban juntas, el analizador léxico no reconocía una por una, sino que simplemente las reconocía como una sola

Identificador = (?!(boolean|int|if|while|for|'|\"[0-9]*[a-zA-Z]+[0-9]*[a-zA-Z_]*(?!\"

Números constantes = (?!(a-z)+)[0-9]+(?!(a-z)+)

Operadores = [*|/|+|=|%|-|.]

Separadores = [{|}|;|(|)]

Caracteres = (')[a-z]+('\$

Palabras claves = (if|while|int|boolean|for)

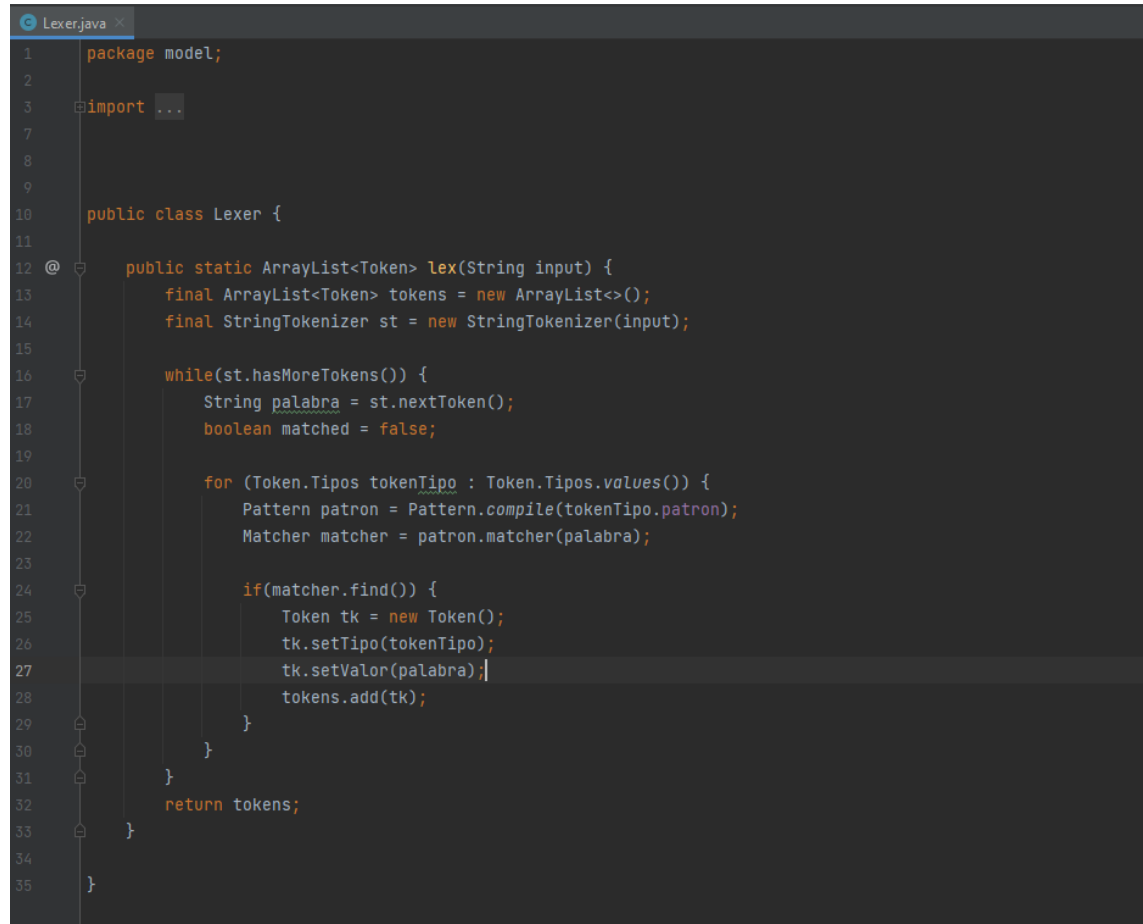
5. Clase Token

En esta clase se almacenan los tokens que se van a identificar, estos tokens fueron almacenados en maneras de expresiones regulares para que sea más fácil su codificación y no se vuelva algo extenso.

```
Token.java x
1 package model;
2
3 public class Token {
4
5
6     public Tipos getTipo() { return tipo; }
7
8
9     public void setTipo(Tipos tipo) { this.tipo = tipo; }
10
11
12
13     public String getValor() { return valor; }
14
15
16
17     public void setValor(String valor) { this.valor = valor; }
18
19
20
21
22     private Tipos tipo;
23     private String valor;
24
25     public enum Tipos {
26         IDENTIFY( s: "\\b(?:boolean|int|if|while|for|String|')[0-9]*[a-zA-Z_]+[0-9]*[a-zA-Z_]*(?!'\\b)",
27         NUMCONST ( s: "\\b(?:[a-z]+)[0-9]+(?:[a-z]+)\\b",
28         OPERATOR( s: "[*|/|+|=|%|-|"+",
29         SEPARATOR( s: "[{|}|;|(|)|]",
30         CHARCONST( s: "'[a-z]+'$",
31         KEYWORD ( s: "(if|while|int|boolean|for|String)");
32
33     public final String patron;
34
35     Tipos(String s) { this.patron = s; }
36
37 }
38
39
40 }
```

6. Clase Lexer

En la clase lexer es donde se guarda en un ArrayList el código a tokenizar y se evalúa.



```
1 package model;
2
3 import ...
4
5
6
7
8
9
10 public class Lexer {
11
12 @ public static ArrayList<Token> lex(String input) {
13     final ArrayList<Token> tokens = new ArrayList<>();
14     final StringTokenizer st = new StringTokenizer(input);
15
16     while(st.hasMoreTokens()) {
17         String palabra = st.nextToken();
18         boolean matched = false;
19
20         for (Token.Tipos tokenTipo : Token.Tipos.values()) {
21             Pattern patron = Pattern.compile(tokenTipo.patron);
22             Matcher matcher = patron.matcher(palabra);
23
24             if(matcher.find()) {
25                 Token tk = new Token();
26                 tk.setTipo(tokenTipo);
27                 tk.setValor(palabra);
28                 tokens.add(tk);
29             }
30         }
31     }
32     return tokens;
33 }
34
35 }
```