TECNOLÓGICO NACIONAL DE MÉXICO EN CELAYA

DEPARTAMENTO DE INGENIERÍA EN SISTEMAS COMPUTACIONALES

ACTIVIDAD 2

LENGUAJES Y AUTOMATAS II

DOCENTE DESIGNADO: ISC. RICARDO GONZÁLEZ GONZÁLEZ

EQUIPO 1. INTEGRANTES:

HERNANDEZ MARTINEZ ALINA ADRIANA

AVILA CARDENAS RICARDO

CID SOTO PABLO

RICO GONZALEZ EDUARDO

CELAYA, GTO A 19 MARZO 2020

A. INVESTIGAR. ¿QUÉ ES UN ANÁLISIS LÉXICO APLICADO A LA VALORACIÓN DE UN LENGUAJE FORMAL?

ANÁLISIS Y SINTESIS

Cualquier compilador debe realizar dos tareas principales: análisis del programa a compilar y síntesis de un programa en lenguaje máquina que, cuando se ejecute, realizara correctamente las actividades descritas en el programa fuente. Para el estudio de un compilador, es necesario dividir su trabajo en fases. Cada fase representa una transformación al código fuente para obtener el código objeto.

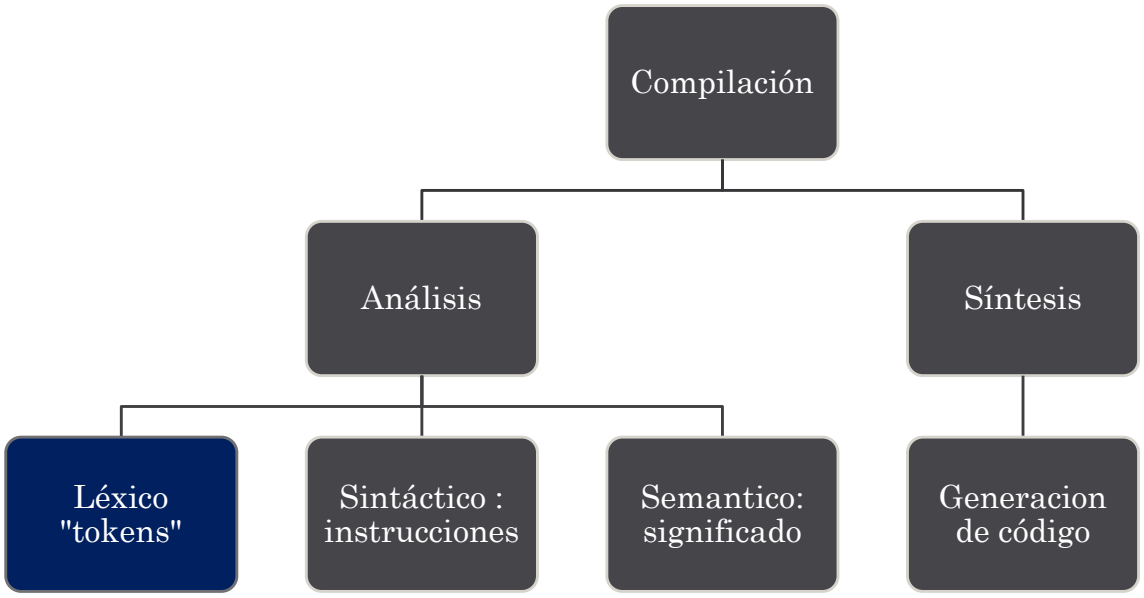


Figura 1. Procesos dentro de la compilación

La fase del análisis léxico convierte una cadena de caracteres que conforman el programa fuente en un grupo de palabras, que son secuencias de caracteres con significado propio.

Es decir, este módulo está destinado a leer caracteres del archivo de entrada (aquí se encuentran las cadenas a analizar) y se encarga de reconocer que las sub-cadenas correspondan a los símbolos del lenguaje y retornar los tokens correspondientes y sus atributos

A la secuencia de caracteres que representa un componente léxico se le llama lexema (o con su nombre en inglés token). En el caso de los identificadores creados por el programador no solo se genera un componente léxico, sino que se genera otro lexema en la tabla de símbolos.

Figura 2. Analizador léxico.

La especificación de un lenguaje de programación a menudo incluye un conjunto de reglas que definen el léxico. Estas reglas consisten comúnmente en **EXPRESIONES REGULARES** que indican el conjunto de posibles secuencias de caracteres que definen un token o lexema.

La palabra léxico significa diccionario y aplicado en el ambiente de lenguajes de programación se utiliza para denotar los símbolos del lenguaje .Cada lenguaje de programación define reglas que permiten componer un texto en un programa como una secuencia de símbolos . El conjunto de reglas se denominan gramática o mas usualmente la sintaxis del lenguaje , Sintaxis significa con orden , cada regla establece una clase definida de objetos o categorías sintácticas , como algunas partes típicas de un programa : *ACCIONES , DECLARACIONES, CONDICIONES, EXPRESIONES , ETC …*

Asociado a cada palabra (símbolo) y a cada frase (categoría sintáctica) debe existir un significado. Que se traduce en valores objetos (constantes y variables) de acuerdo a sus tipos; o en grupos de acciones o en nombres de objetos, o en la especificación sobre las acciones que pueden efectuarse sobre

estos objetos, todas las reglas que aportan esta información se denominan semántica del lenguaje

Si bien las reglas para construir frases y símbolos son finitas el conjunto de programas es infinito

Ejemplo:

Si ***existe*** {
Posicion = 60;
} ***sino*** {
Posicion = 1+2;
}
	1. ***Si*** palabra reservada
	2. ***existe*** expresión booleana
	3. {principio de la instrucción
	4. **Posicion** identificador
	5. = símbolo de asignación
	6. 60 constante
	7. ; delimitador
	8. } final de la instrucción
	9. ***sino*** palabra reservada
	10. {principio de la instrucción
	11. **Posición** identificador
	12. = símbolo de asignación
	13. 1 constante
	14. + símbolo aritmético
	15. 2 constante
	16. ; delimitador
	17. } final de la instrucción

Tabla 1. Ejemplo reconocimiento léxico

Para describir con rigurosidad los lenguajes de programación se emplea una notación formal que se denomina Metalenguaje. El formulismo más conocido, y que emplearemos en la descripción, es el Formalismo Extendido Backus-Nauer. (BNF, Backus Nauer Formalism). Básicamente consiste en describir una frase, (categoría sintáctica) o parte abstracta de un programa, mediante la secuencia de componentes, de menor categoría, que pueden reemplazar dicha frase. Las reglas deben especificar hasta llegar al reemplazo por los símbolos que componen el diccionario.

En este ambiente, los elementos léxicos del lenguaje se denominan símbolos terminales. Las componentes estructurales del lenguaje que serán reemplazadas, se denominan símbolos no terminales. La regla que establece el reemplazo de un símbolo no terminal por una secuencia de símbolos terminales y no terminales se denomina producción. Las reglas deben

permitir verificar, con facilidad, si una secuencia de símbolos es o no una sentencia correcta del lenguaje.

GRAMATICA TIPO 2 LIBRE DE CONTEXTO DEL LENGUAJE PROTOTIPO (BNF)

G = < Vn, Vt, P, S >

Vn = {S, programa, idetificador, declaraciones, procedimientos, principal, tipoDato, letra, digito, instrucciones, entrada, salida, parametros,}

Vt = 0 | 1 | 2 | … | 9, abac | … | z A | B | C | … | Z | + | - | * | / | & | | |, < | > | == | …

P = {

1. <programa>:: =**programa** <identificador> ´**{**´<declaraciones> * <procedimientos> * principal´ **()**´ ´**{**´<instrucciones>*´**}**´ ´**}'**
2. <identificador>::= '**$**'<letra>[(<letra> | <digito>)*]
3. <declaraciones>:: =<tipoDato> { <identificador> [´**,**´<identificador>] } **';'**
4. <tipoDato>:== **entero | real | cadena**
5. <letra> ::= a | b | c | … | x | y | z | A | B | C | … | X | Y | Z |
6. <digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
7. <procedimientos>::= **subrutina** <identificador> ´**()**´ ´**{**´[<instrucciones>*]´**}**´
8. <instrucciones>::={ <expresion> | <estructuraDecision> | <estructuraCiclo> | <entrada> | <salida> | <asignacion>}
9. <asignación> ::= <identificador> **'='** (<entrada> | <salida> | <expresión>)
10. <entrada> ::= **entrada(** <identificador> | <digito>| <concatenación> **);**

11. <salida>::=**salida(**<identificador> | <digito> | <concatenación> **);**

12. <concatenación>::= { (<identificador> | <digito>)**'+'**{<concatenación>} }

13. <expresion>::=(<identificador>| {<digito>}) [<OprAritmetico> <expresion>] | ´**(**´<expresión>´**)**´

14. <estructuraDecision>::= **si '('** <condición> ´**)**´´**{**´ <instrucciones> ´**}**´ [**sino** ´**{**´<instrucciones>´**}**´]

15. <condición>::=<expresión> [(<OpRelacional>|<OpLogico>)<condicion>]

16. <estructuraCiclo>::= **mientras**´**(**´ <condición> ´**)**´ ´**{**´ <instrucciones> ´**}**´

17. <OpAritmetico>::=+ | - | * | / |^

18. <OpRelacional>::=< | > | == |<= | >=

19. <OpLogicos>::= &&| != | |

20. <delimitadores> ::= ; |{}|()

21. <comentario>::= **'/*'** 0|1|2|…|9, abac|…|z A|B|C|…|Z | + | - | * | / | & | ||, < | > | == | … **'*/'**

S = <programa>

A. INVESTIGAR. ¿EN QUÉ CONSISE UN ANÁLISIS LÉXICO Y QUÉ LO CARACTERIZA?

Como se mencionaba en el punto anterior, este proceso es la primera fase de un compilador y consiste de un programa que recibe como entrada una secuencia de caracteres produciendo una salida compuesta de tokens. Cada token es una secuencia de caracteres que representa una unidad de información en el programa fuente, los token caen en diversas categorías.

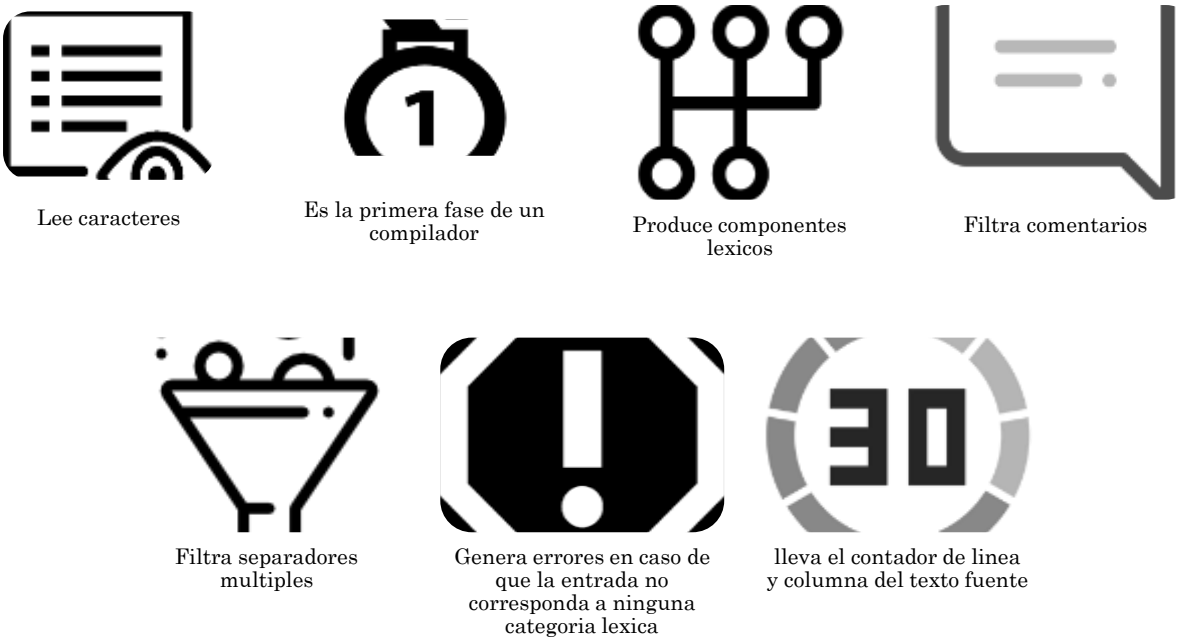
Palabras reservadas: Como if y while, las cuales son cadenas fijas de letras.

Identificadores: Son cadenas definidas por el usuario, compuestas por lo regular de letras y números, y que comienzan con una letra.

Símbolos especiales: Como los símbolos aritméticos + y *; además de algunos símbolos compuestos de múltiples caracteres, tales como > = y <>.

En cada caso un token representa cierto patrón de caracteres que el analizador Léxico reconoce, o ajusta desde el inicio de los caracteres de entrada restantes.

CARACTERISTICAS DEL ANALIS LÉXICO



El analizador Iéxico debe funcionar de manera tan eficiente como sea posible. Por lo tanto, también necesitamos poner mucha atención a los detalles prácticos de la estructura del analizador Léxico

El trabajo del analizador Iéxico es leer los caracteres del código fuente y formarlos en unidades lógicas para que lo aborden las partes siguientes del compilador (generalmente el analizador sintáctico). Las unidades lógicas que genera el analizador Iéxico se denominan tokens, y formar caracteres en tokens es muy parecido a formar palabras a partir de caracteres con una

oración en un lenguaje natural como el inglés o cualquier otro y decidir lo que cada palabra significa.

Los tokens como entidades lógicas se deben distinguir claramente de las cadenas de caracteres que representan. Por ejemplo, el token de la palabra reservada if se debe distinguir de la cadena de caracteres "if' que representa. Para hacer clara la distinción, la cadena de caracteres representada por un token se denomina en ocasiones su valor de cadena o su lexema. Algunos tokens tienen sólo un lexema: las palabras reservadas tienen esta propiedad.

Cualquier valor asociado a un token se denomina atributo del token, y el valor de cadena es un ejemplo de un atributo

Un analizador léxico necesita calcular al menos tantos atributos de un token como sean necesarios para permitir el procesamiento siguiente. Por ejemplo, se necesita calcular el valor de cadena de un token NUM, pero no es necesario calcular de inmediato su valor numérico, puesto que se puede calcular de su valor de cadena.

En ocasiones el mismo analizador Iéxico puede realizar las operaciones necesarias para registrar un atributo en el lugar apropiado, o puede simplemente pasar el atributo a una fase posterior del compilador.

Por ejemplo, un analizador Iéxico podría utilizar el valor de cadena de un identificador para introducirlo a la tabla de símbolos, o podría pasarlo para introducirlo en una etapa posterior.

Aunque la tarea del analizador Léxico es convertir todo el programa fuente en una secuencia de tokens, pocas veces el analizador hará todo esto de una vez. En realidad, el analizador Iéxico funcionará bajo el control del analizador sintáctico, devolviendo el siguiente token simple desde la entrada bajo demanda mediante una función

La función declarada de esta manera devolverá, cuando se le llame, el siguiente token desde la entrada, y además calculará atributos adicionales, como el valor de cadena del token. La cadena de caracteres de entrada por lo regular no tiene un parámetro para esta función, pero se conserva en un

buffer (Localidad de memoria intermedia) o se proporciona mediante las facilidades de entrada del sistema.

COMPONENTES LÉXICOS, PATRONES, LEXEMAS

Patrón: es una regla que genera la secuencia de caracteres que puede representar a un determinado componente léxico (una expresión regular).

Lexema: cadena de caracteres que concuerda con un patrón que describe un componente léxico. Un componente léxico puede tener uno o infinitos lexemas. Por ejemplo: palabras reservadas tienen un ´único lexema. Los números y los identificadores tienen infinitos lexemas.

Componente léxico: se suelen definir como un tipo enumerado. Se codifican como enteros. También se suele almacenar la cadena de caracteres que se acaba de reconocer (el lexema), que se usara posteriormente para el análisis semántico.

Compon. léxico	Lexema	Patrón
identificador	indice, a, temp	letra seguida de letras o dígitos
num_entero	1492, 1, 2	dígito seguido de más dígitos
if	if	letra i seguida de letra f
do	do	letra d seguida de o
op_div	/	carácter /
op_asig	=	carácter =

Tabla 2.Componentes Léxicos

Es importante conocer el lexema (para construir la tabla de símbolos). Los componentes léxicos se representan mediante una estructura registro con tipo de token y lexema.

RECONOCIMIENTO DE LEXEMAS Y TABLA DE SIMBOLOS

Para distinguir unas palabras (o lexemas) de otras se utilizan patrones de reconocimiento. En muchos casos los patrones se describen mediante expresiones regulares. Los identificadores pueden también sustituirse por referencias a la tabla de símbolos, para una utilización más eficiente. La tabla de símbolos asocia a cada identificador un número, así como una serie de atributos (tipo de datos, etc.).

Una tabla de símbolos también se la llama tabla de nombres o tabla de identificadores, es una estructura de datos que usa el proceso de traducción de un lenguaje de programación, por un compilador o un intérprete, donde cada símbolo en el código fuente de un programa está asociado con información tal como la ubicación, el tipo de datos y el ámbito de cada variable, constante o procedimiento.

Permanece sólo en tiempo de compilación, no de ejecución, excepto en aquellos casos en que se compila con opciones de depuración. La tabla almacena la información que en cada momento se necesita sobre las variables del programa, información tal como: ***Nombre, tipo, dirección de localización, tamaño, etc.*** La gestión de la tabla de símbolos es muy importante, ya que consume gran parte del tiempo de compilación.

De ahí que su eficiencia sea crítica. Aunque también sirve para guardar información referente a los tipos creados por el usuario, tipos enumerados y, en general, a cualquier identificador creado por el usuario, nos vamos a centrar principalmente en las variables de usuario

Almacenamiento del nombre. Se puede hacer con o sin límite. Si lo hacemos con límite, emplearemos una longitud fija para cada variable, lo cual aumenta la velocidad de creación, pero limita la longitud en unos casos, y desperdicia espacio en la mayoría.

El tipo también se almacena en la tabla, como veremos en un apartado dedicado a ello.

Dirección de memoria en que se guardará. Esta dirección es necesaria, porque las instrucciones que referencian a una variable deben saber dónde encontrar el valor de esa variable en tiempo de ejecución, también cuando se trata de variables globales. En lenguajes que no permiten recursividad, las direcciones se van asignando secuencialmente a medida que se hacen las declaraciones. En lenguajes con estructuras de bloques, la dirección se da con respecto al comienzo del bloque de datos de ese bloque, (función o procedimiento) en concreto.

También podemos guardar información de los números de línea en los que se ha usado un identificador, y de la línea en que se declaró.

IDENTIFICADOR	DIRECCIÓN	TIPO	DIMENSIÓN	OTROS ATRIBUTOS
compañía	STATIC+0	C	10	...
x3	STATIC+10	I	0	...
forma1	STATIC+12	B	0	...
b	STATIC+13	F	3	...

Tabla 3.Reconocimiento , Tabla de símbolos

La interfaz de la tabla de símbolos debe quedar clara desde el principio de manera que cualquier modificación en la implementación de la tabla de símbolos no tenga repercusión en las fases del compilador ya desarrolladas.

Las operaciones básicas que debe poseer son:

Crear ()	Insertar(símbolo)	Buscar (nombre)	Imprimir()
crea una tabla vacía.	añade a la tabla el símbolo dado	devuelve el símbolo cuyo nombre coincide	a efectos informativos, visualiza por la

		con el parámetro. Si el símbolo no existe devuelve null.	salida estándar la lista de variables almacenadas en la tabla de símbolos junto con sus valores asociados.

Tabla 4. Operaciones básicas

La tabla de símbolos propuesta será la siguiente:

ID	***TOKEN***	***LEXEMA***	***LINEA***	***VALOR***
1	29	$hola	1	0
2	24	{	2	-
3	29	$suma	3	0
4	15	=	4	-
6	28	6	5	-
8	13	+	6	-
7	28	5	7	-
5	25	}	8	-

Tabla 5. Tabla de símbolos del lenguaje prototipo

ID: Este es un id que se asigna a cada token que se ingresa en la tabla, este debe ser único para cada uno.

TOKEN: Este TOKEN que se asigna al lexema corresponde al número de dependiendo de su clasificación, es decir si es una palabra reservada todos los tokens que correspondan a una palabra reservada tendrán el mismo TOKEN. A continuación, se listan el TOKEN que corresponde a cada token según sea su clasificación.

LEXEMA: Esta columna corresponde al nombre de todos los tokens que se encuentran en la tabla, ya sea una palabra reservada, un identificador, delimitador etc.

LINEA: es el numero de la línea correspondiente al programa

TOKEN	LEXEMA
1	programa
2	principal
3	subrutina
4	entrada
5	salida
6	si
7	sino
8	mientras
20	+
21	-
22	*
23	/
24	^
25	=
30	<
31	>
32	==
33	>=
34	<=
40	&&

41	!=
42	|
50	entero
51	real
52	cadena
60	;
61	{
62	}
63	[
64	]
65	(
66	)
67	/*
68	*/
70	Identificador ($xy)

Tabla 6. Valores de los tokens

C. IDENTIFICAR. ¿QUÉ CASOS DE ESTUDIOS SON LOS IMPORTANTES A CONSIDERAR EN EL ANÁLISIS LÉXICO?

D. ¿QUÉ PROCESOS Y PROBLEMAS ATIENDE UN ANÁLISIS LÉXICO?

ERRORES LÉXICOS

Los errores léxicos se detectan cuando el analizador léxico intenta reconocer componentes léxicos y la cadena de caracteres de la entrada no encaja con ningún patrón. Son situaciones en las que usa un carácter invalido (@,\,",¿,...), que no pertenece al vocabulario del lenguaje de programación, al escribir mal un identificador, palabra reservada u operador.

Errores léxicos típicos son:

- **nombre ilegales de identificadores:** un nombre contiene caracteres inválidos.

- **números incorrectos:** un número contiene caracteres inválidos o no está formado correctamente, por ejemplo 3,14 en vez de 3.14 ´o 0.3.14.
- **errores de ortografía en palabras reservadas:** caracteres omitidos, adicionales o cambiados de sitio, por ejemplo la palabra hwile en vez de while.

Los errores léxicos se deben a descuidos del programador. En general, la recuperación de errores léxicos es sencilla y siempre se traduce en la generación de un error de sintaxis que será detectado más tarde por el analizador sintáctico cuando el analizador léxico devuelve un componente léxico que el analizador sintáctico no espera en esa posición. Los métodos de recuperación de errores léxicos se basan bien en saltarse caracteres en la entrada hasta que un patrón se ha podido reconocer; o bien usar otros métodos más sofisticados que incluyen la inserción, borrado, sustitución de un carácter en la entrada o intercambio de dos caracteres consecutivos.

TABLA DE ERRORES

Para un mayor control de errores en el análisis léxico se propone una tabla de errores en la cual se clasificarán dependiendo el análisis que se esté ejecutando.

A continuación, le listan los id de error junto con el mensaje de descripción que corresponde a cada uno.

Empezaran a partir del número 100.

CASOS	ORIGEN
100	Carácter no valido
200	Números mal formados
300	Errores en palabras reservadas
400	Nombres no validos de los identificadores.

CASOS DE ERRORES LEXÍCOS:

DESCRIPCIÓN: ***SE DECLARA UN IDENTIFICADOR SIN EL CARÁCTER $,*** este caso ocurre cuando el programador intenta declarar una variable que no contenga el carácter $ al inicio de este.

- **ERROR:** 100
- **MENSAJE:** "No puede definirse como un identificador, debe de iniciar con un signo de $"
- **EJEMPLO:**

Entero numero=12;

Cadena palabra;

DESCRIPCIÓN: ***EXISTEN ESPACIOS EN BLANCO EN LA NOMECLATURA DE LAS VARIABLES,*** este caso ocurre cuando el programador intenta declarar una variable y existen espacio en blanco entre ellas.

- **ERROR:** 101
- **MENSAJE:** "No deben de existir espacios en blanco entre los identificadores"
- **EJEMPLO:**

$es un numero=12;

Cadena $la palabra;

DESCRIPCIÓN: ***LAS PALABRAS RESERVADAS Y DECLARACIONES SE ESCRIBEN DE MANERA INCORRECTA O CONTIENEN SIMBOLOS EXTRAÑOS***

- **ERROR:** 102
- **MENSAJE:** "El token no se puede establecer como identificador ni palabra reservada, no pertenece al lenguaje"
- **EJEMPLO:**

S1 , S| , S?

S1N0 , S|N0 , SIN°

MIENTR4Z , $°>ídem

DESCRIPCIÓN: ***NO SE PUEDE ESCRIBIR UNA CONSTANTE NÚMERICA CON CARACTERES DIFERENTES A NÚMEROS,*** si el programador ingresa alguna constante numérica, pero le coloca letras o algún otro carácter no establecido

- **ERROR:** 103
- **MENSAJE:** "Las constantes numéricas no pueden mezclarse con letras o caracteres extraños"
- **EJEMPLO:**

Entero $numero = 123ABC;

123|.

123?)

123xc__23

DESCRIPCIÓN: ***NO PUEDE EXISTIR UN $ INTERMEDIARIO EN LOS IDENTIFICADORES***

- **ERROR:** 104
- **MENSAJE:** "No se puede establecer como identificador"
- **EJEMPLO:**

Entero $enter$o;

Real soyreal$;

Cadena $hola$c;

DESCRIPCIÓN: ***NUMEROS INVALIDOS,*** el error se presenta cuando el programador ingresa un nuero que contenga varios puntos o punto y coma.

- **ERROR:** 105
- **MENSAJE:** "Los números decimales no son válidos"
- **EJEMPLO:**

REAL $numero = 12.3.4,2 ;

DESCRIPCIÓN: ***ACENTOS EN LOS CARACTERES DEL TOKEN,*** si el programador ingresa alguna cadena que contenga algún acento en alguno de sus caracteres será incorrecto.

- **ERROR:** 106
- **MENSAJE:** "los caracteres con acentos no son válidos"
- **EJEMPLO:**

$programación , $camión , $estación , $jugaría

DESCRIPCIÓN: ***NO SE PUEDEN UTILIZAR OPERADORES EN LA NOMECLATURA DE LAS PALABRAS,*** si el programador ingresa alguna variable la cual contenga un operador aritmético, lógico u relacional en medio de esta

- **ERROR:** 107

- **MENSAJE:** "Token invalido no se pueden utilizar operadores en la nomenclatura de las variables"
- **EJEMPLO:**

$progr*amación, $ca>mión, $es&&tación , $jug>=aría

DESCRIPCIÓN: ***SE INTRODUCE UNA SERIE DE TOKENS VALIDOS SIN ESPACIO ENTRE ELLOS,*** si el programador ingresa alguna cadena la cual está formada por varios tokens validos en el lenguaje, pero estas no contienen un espacio entre ellas se genera el error

- **ERROR:** 108
- **MENSAJE:** "se deben de separar los tokens"
- **EJEMPLO:**

Entero$numero=32;

Cadena$unacadena=" hola soy cadena";

PROGRAMAS PRUEBA

Programa 1

```
//Iniciamos con el cuerpo base del programa
programa $prueba {
   //En la siguiente línea nos generara el error 100, ya que
   //un identificador no puede estar sin el signo $
   entero numero;
   real $x,$y;
   //En la siguiente línea genera el error 108 ya que se ingresan
```

```
    //tokens validos sin espacio entre ellos
    cadena$hola;

    principal () {
        $x = 1;
        //La siguiente línea genera el error 105, ya que
        //un numero no puede tener dos puntos
        $y = 2..43;

        si($x >= $y){
            salida($x);
        }
        sino {
            salida($y);
        }
    }
}
```

Programa 2

```
//Iniciamos con el cuerpo base del programa
programa $prueba {

    //La siguiente línea genera el error 107 ya que se ingresan
    //operadores en la nomenclatura de las palabras
    entero $pru*ba;
```

```
    real $x,$y,$res;

    //La siguiente línea genera el error 106 ya que se ingresan
    //acentos en el identificador
    cadena $númeroReal;

    principal () {
        $x = 7654764;

        //La siguiente línea genera el error 103 ya que se ingresan
        //caracteres no validos en un número
        $y = 5849FJDSAJFKAJF;

        si($x >= $y) {
            $res = $x * $y;
            salida($res);
        }
        sino{
            $res = 2 * $x;
            salida($res);
        }
    }
}
```

PROGRAMA 3

```
//Iniciamos con el cuerpo base del programa

programa $prueba {


   //La siguiente línea genera el error 101, ya que

   //no debe existir espacio en medio del identificador

   entero $prue ba;


   real $x,$y,$res;


   //La siguiente línea genera el error 104, ya que

   //se tiene el signo $ dos veces

   cadena $numero$Real;


   principal () {

      $x = entrada ();

      $y = entrada ();


      si ($x >= $y) {

         $res = $x * $y;

         salida($res);

      }

      //La siguiente línea genera el error 102, ya que

      //se escribe mal la palabra reservada sino
```

```
        sin0{
            $res = 2 * $x;
            salida($res);
        }
    }
}
```

E. ¿CÓMO IMPLEMENTAR UTILIZANDO UN ANÁLISIS LÉXICO?

DESCRIPCIÓN, MODELADO Y FASES DEL ANALIZADOR LEXICO

Generación y manipulación de archivo de texto.

- Abrir archivo de texto.
- Obtener el contenido del archivo de texto.
- Guardar y especificar la ruta del archivo de texto.
- Cerrar archivo de texto.

Separación y Manipulación de tokens

- Tomar código fuente.
- Separar en tokens en base en delimitadores, operadores, espacio en blanco.
- Recorrer carácter por carácter en cada token para validarlo.
- Si es válido agregarlo a la tabla de símbolos con su respectiva clasificación.

Creación y manipulación de la tabla de símbolos

- Generar la tabla de símbolos.
- Tomar tokens válidos.
- Agregar cada token valido a la tabla de símbolos.
- Mostar tabla de símbolos.

DIAGRAMAS UNIFIED MODELING LANGUAGE

Diagrama de casos de uso

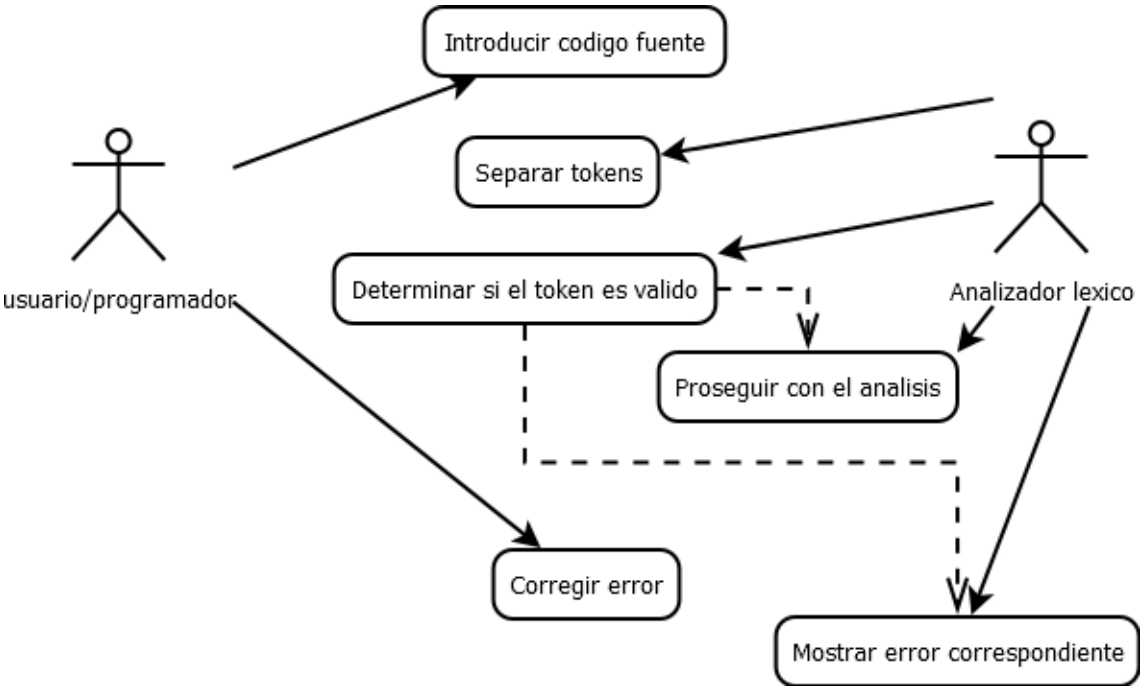
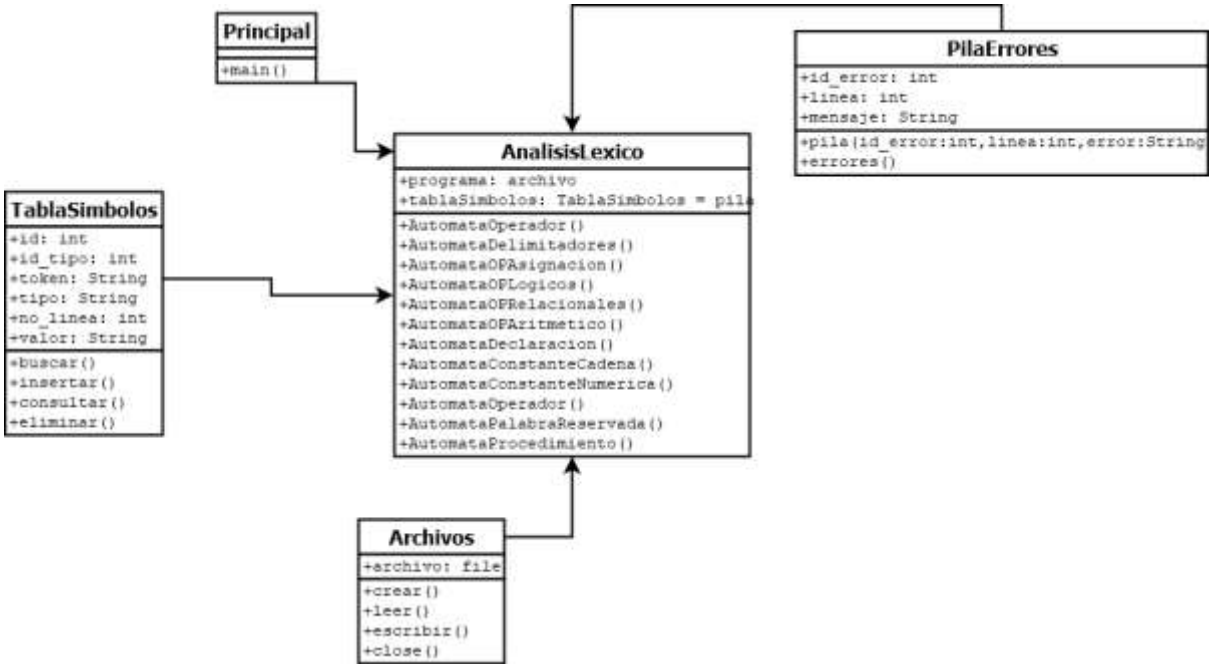
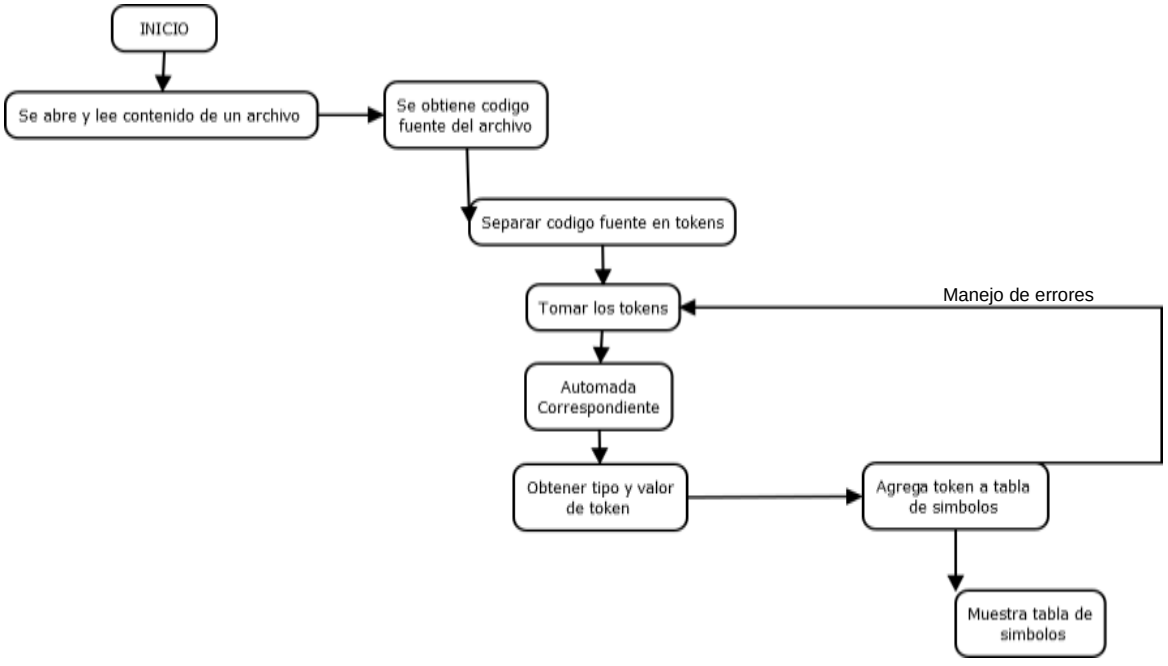


Diagrama de clases y objetos

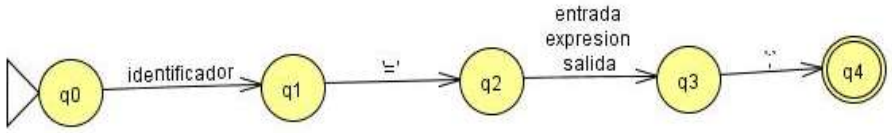


Pseudocodigo



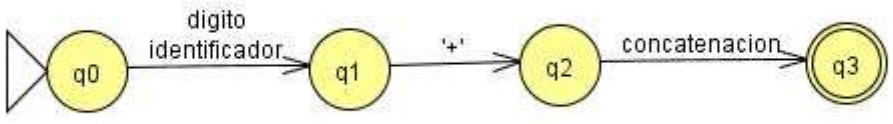
AUTOMATAS

Asignación



	identificador	=	entrada	expresión	salida	;
→Q0	Q1					
Q1		Q2				
Q2			Q3	Q3	Q3	
Q3						Q4
*Q4						

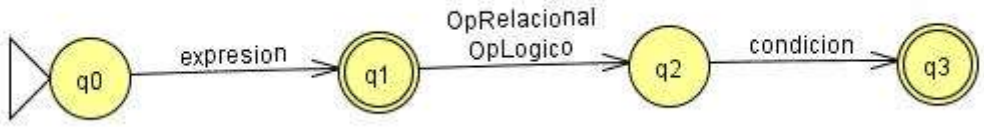
Concatenación



	digito	identificador	+	concatenación
→Q0	Q1	Q1		

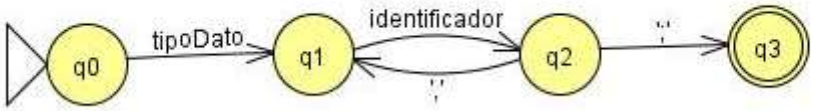
Q1			Q2	
Q2				Q3
*Q3				

Expresión



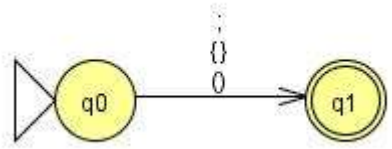
	expresión	Operacional	OpLogico	Condición
→Q0	Q1			
*Q1		Q2	Q2	
Q2				Q3
*Q3				

Tipo de dato



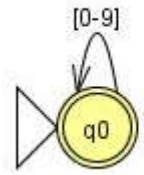
	tipoDato	identificador	;
→Q0	Q1		
Q1		Q2	
Q2			Q1,Q3
*Q3			

Delimitadores



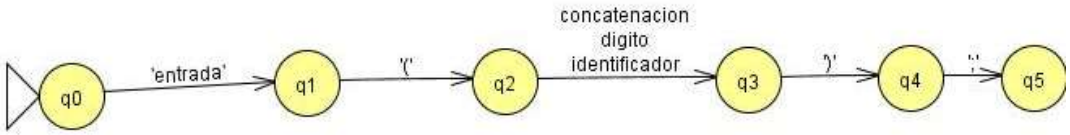
	{}	()
→Q0	Q1	Q1
*Q1		

Dígitos



	0-9
→*Q0	Q0

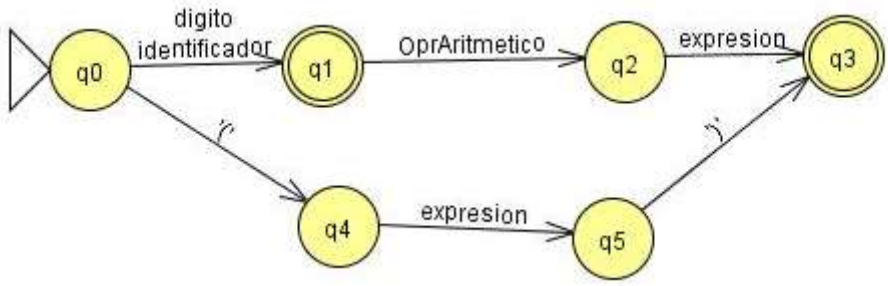
Entradas



	entrada	(	concatenacion	digito	identificador	)	;
→Q0	Q1						

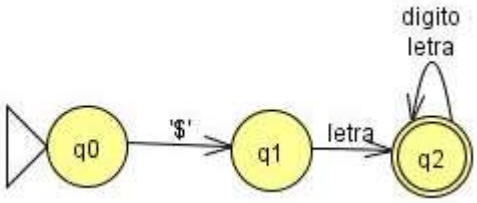
Q1		Q2					
Q2			Q3	Q3	Q3		
Q3						Q4	
Q4							Q5
*Q5							

Expresión



	digito	identificador	OprAritmetico	expresion	(	)
→Q0	Q1	Q1			Q4	
*Q1			Q2			
Q2				Q3		
*Q3						
Q4				Q5		
Q5						Q3

Identificador

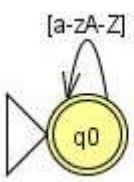


	$	letra	Digito
→Q0	Q1		
Q1		Q2	
*Q2		Q2	Q2

Instrucción

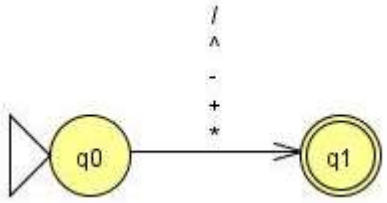
	salida	estructuraDecision	entrada	estructuraCiclo	asignacion	expresion
→Q0	Q1	Q1	Q1	Q1	Q1	Q1
*Q1						

Letras



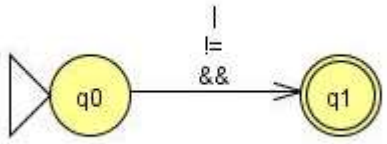
	a-zA-Z
→*Q0	Q0

Operadores aritméticos



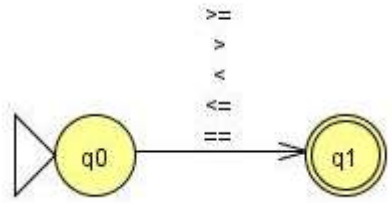
	/	^	-	+	*
*Q0	Q1	Q1	Q1	Q1	Q1
→Q1					

Operadores Lógicos



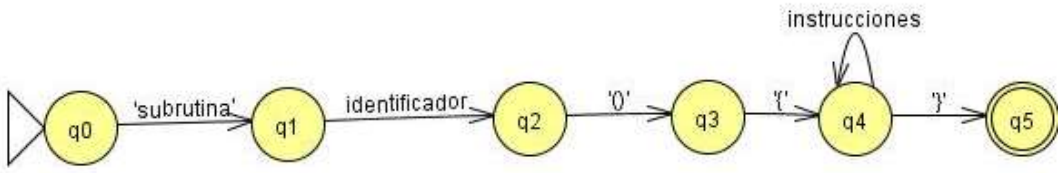
	|	¡=	&&
→Q0	Q1	Q1	Q1
*Q1			

Operadores relacionales



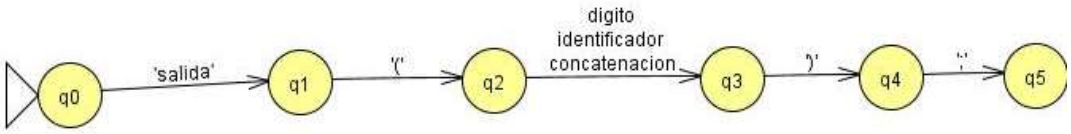
	>=	>	<	<=	==
→Q0	Q1	Q1	Q1	Q1	Q1
*Q1					

Subrutina



	subrutina	identificador	()	{	instruccion	}
→Q0	Q1					
Q1		Q2				
Q2			Q3			
Q3				Q4		
Q4					Q4	Q5
*Q5						

Salida



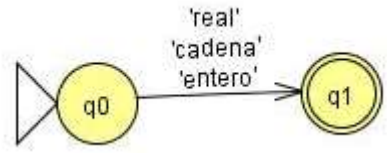
	salida	(	digito	identificador	concatenacion	)	;
→Q0	Q1						
Q1		Q2					
Q2			Q3	Q3	Q3		
Q3						Q4	
Q4							Q5
*Q5							

Estructura de decisión

	si	(	condicion	)	{	instruccion	}	sino
→Q0	Q1							
Q1		Q2						
Q2			Q2					
Q3				Q4				
Q4					Q5			
Q5						Q6		
Q6							Q7	
*Q7								Q8
Q8					Q9			
Q9						Q10		
Q10							Q11	

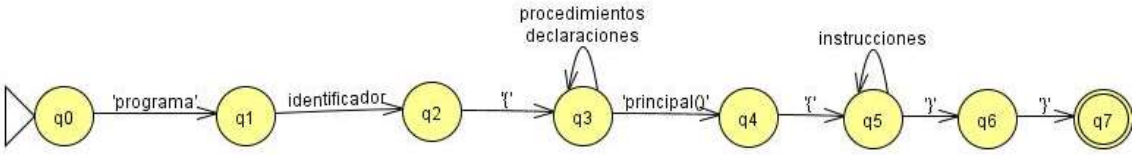
*Q11								

Tipo de dato



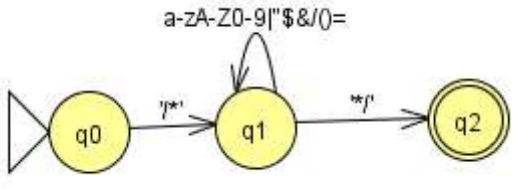
	real	cadena	entero
→Q0	Q1	Q1	Q1
*Q1			

Programa



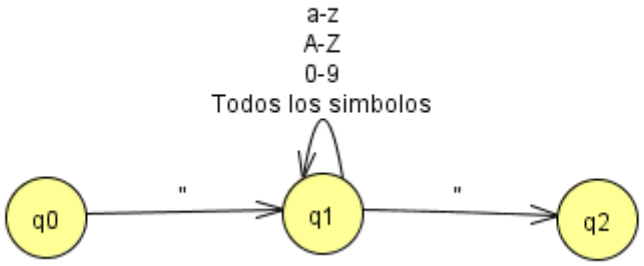
	programa	identificador	{	procedimientos	declaraciones	principal()	}	instrucciones
→Q0	Q1							
Q1		Q2						
Q2			Q3					
Q3				Q3	Q3	Q4		
Q4							Q5	
Q5							Q6	Q5
Q6							Q7	
*Q8								

Comentario



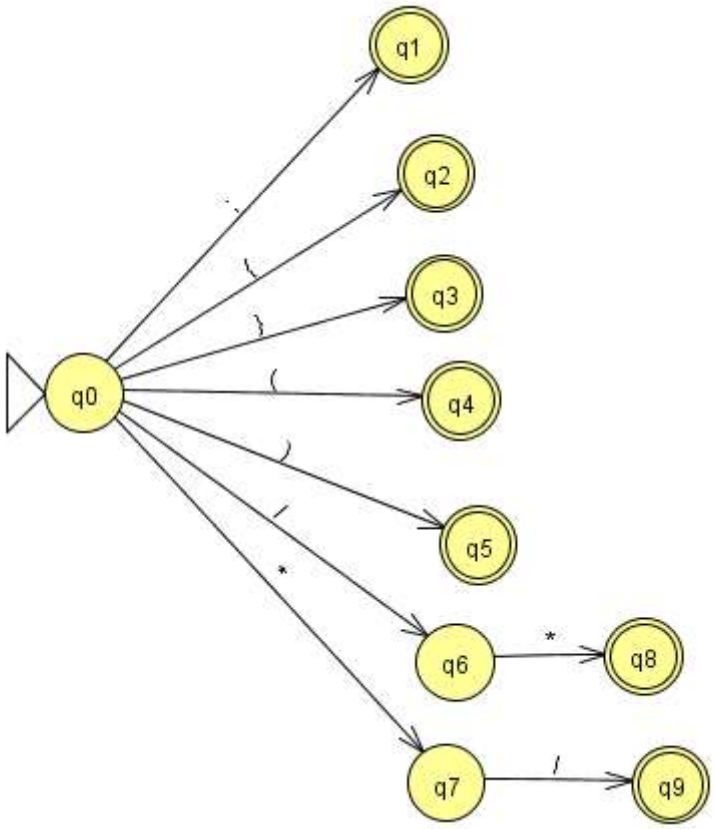
	/*	a-zA-Z0-9|"$&/()=	*/
→Q0	Q1		
Q1		Q1	Q2
*Q2			

Cadena



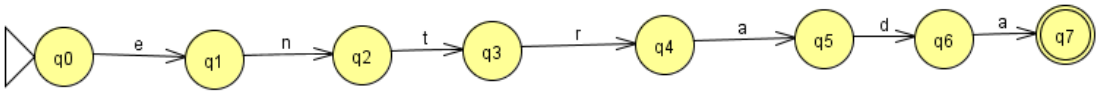
	“	a-zA-Z0-9|"$&/()=	“
→Q0	Q1		
Q1		Q1	Q2
*Q2			

Delimitadores



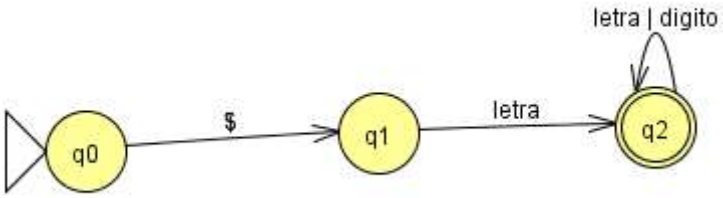
	;	{	}	(	)	/	*
→Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7
*Q1							
*Q2							
*Q3							
*Q4							
*Q5							
Q6							Q8
Q7						Q9	
*Q8							
*Q9							

Entrada



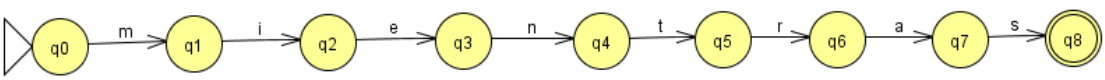
	e	n	t	r	a	d
→Q0	Q1					
Q1		Q2				
Q2			Q3			
Q3				Q4		
Q4					Q5	
Q5						Q6
Q6					Q7	
*Q7						

Id



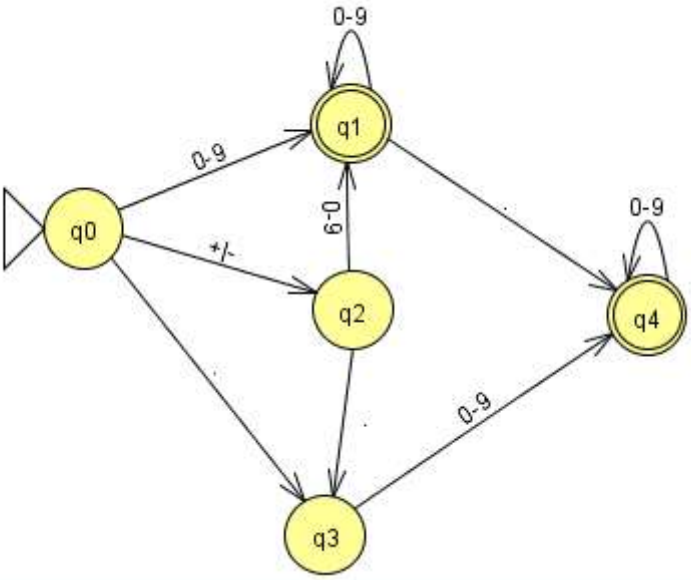
	$	letra	digito
→Q0	Q1		
Q1		Q2	
*Q2		Q2	Q2

Mientras



	m	i	e	n	t	r	a	s
→Q0	Q1							
Q1		Q2						
Q2			Q3					
Q3				Q4				
Q4					Q5			
Q5						Q6		
Q6							Q7	
Q7								Q8
*Q8								

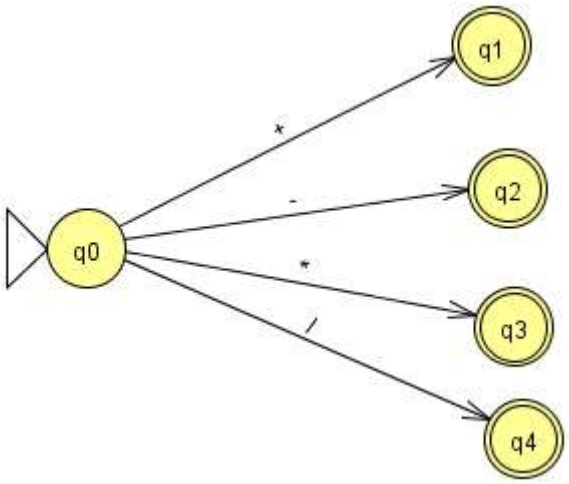
Números



	0-9	+	-	.
→Q0	Q1	Q2	Q2	Q3
*Q1	Q1			Q4
Q2	Q1			Q3

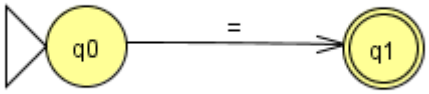
Q3	Q4			
*Q4	Q4			

Operaciones aritméticas



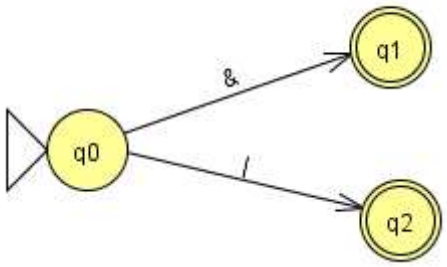
	+	-	*	/
→Q0	Q1	Q2	Q3	Q4
*Q1				
*Q2				
*Q3				
*Q4				

Operación asignación



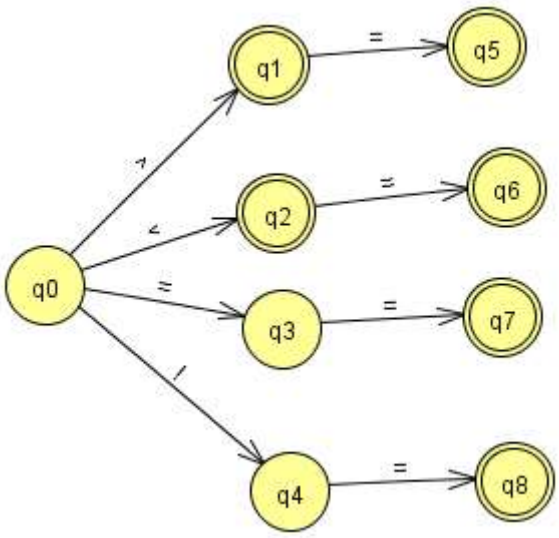
	=
→Q0	Q1
*Q1	

Operadores lógicos



	&	|
→Q0	Q1	Q2
*Q1		
*Q2		

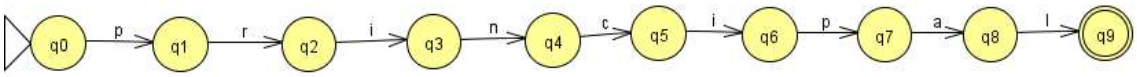
Operadores relacionales



	>	<	=	¡
→Q0	Q1	Q2	Q3	Q4
*Q1			Q5	
*Q2			Q6	
Q3			Q7	
Q4			Q8	
*Q5				

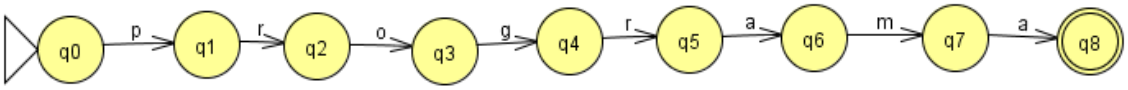
*Q6				
*Q7				
*Q8				

Principal



	p	r	i	n	c	a	l
→Q0	Q1						
Q1		Q2					
Q2			Q3				
Q3				Q4			
Q4					Q5		
Q5			Q6				
Q6	Q7						
Q7						Q8	
Q8							Q9
*Q9							

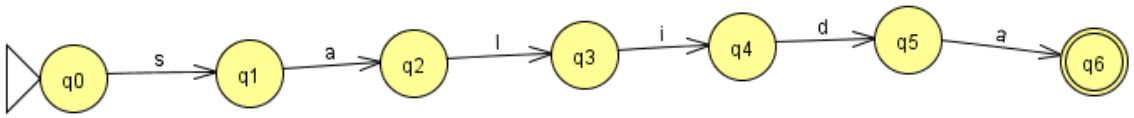
Programa



	p	r	o	g	a	m
→Q0	Q1					
Q1		Q2				
Q2			Q3			
Q3				Q4		

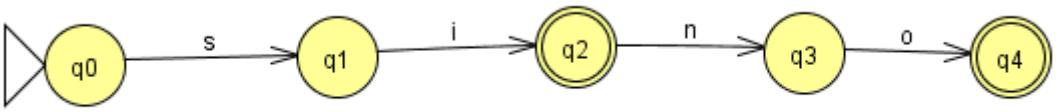
Q4		Q5				
Q5					Q6	
Q6						Q7
Q7					Q8	
Q8						

Salida



	s	a	l	i	d
→Q0	Q1				
Q1		Q2			
Q2			Q3		
Q3				Q4	
Q4					Q5
Q5		Q6			
*Q6					

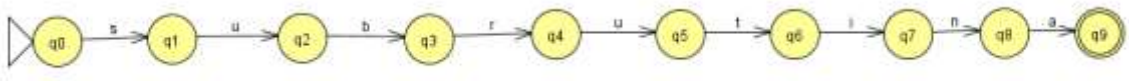
Si-sino



	s	i	n	o
→Q0	Q1			
Q1		Q2		
*Q2			Q3	
Q3				Q4

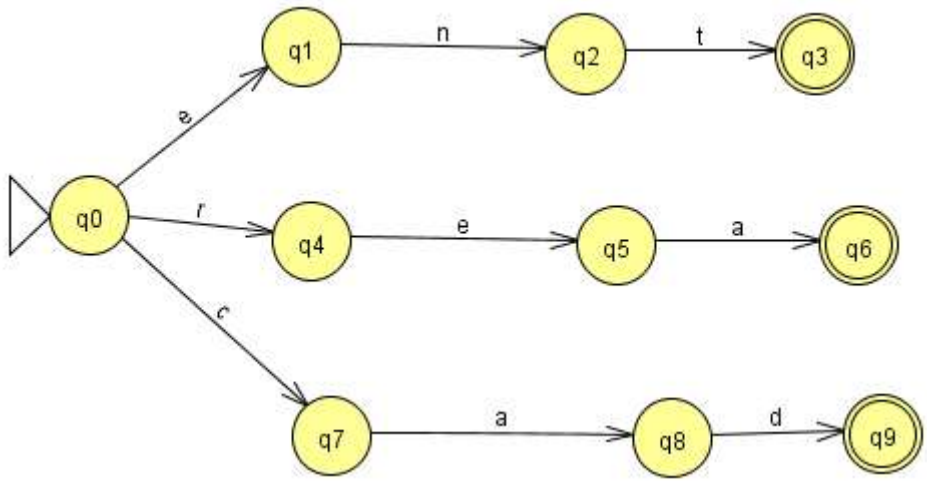
*Q4				

Subrutina



	s	u	b	r	t	i	n	a
→Q0	Q1							
Q1		Q2						
Q2			Q3					
Q3				Q4				
Q4		Q5						
Q5					Q6			
Q6						Q7		
Q7							Q8	
Q8								Q9
*Q9								

Tipo de dato



	e	n	t	r	a	c	d

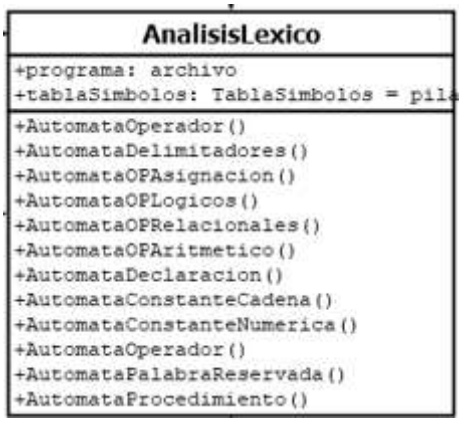
→Q0	Q1			Q4		Q7	
Q1		Q2					
Q2			Q3				
*Q3							
Q4	Q5						
Q5					Q6		
*Q6							
Q7					Q8		
Q8							Q9
*Q9							

Documentación

A continuación, se detallará como fue implementado el Analizador Léxico dentro de nuestro Pseudocompilador que estamos desarrollando, se mostrara el diagrama de clase correspondiente a la parte mostrada, junto con una muestra del código en el cual se muestran los métodos que contiene la clase, los métodos se encuentran sin desplegar para mostrar la estructura en un espacio más compacto.

Clase "AnalisisLexico":

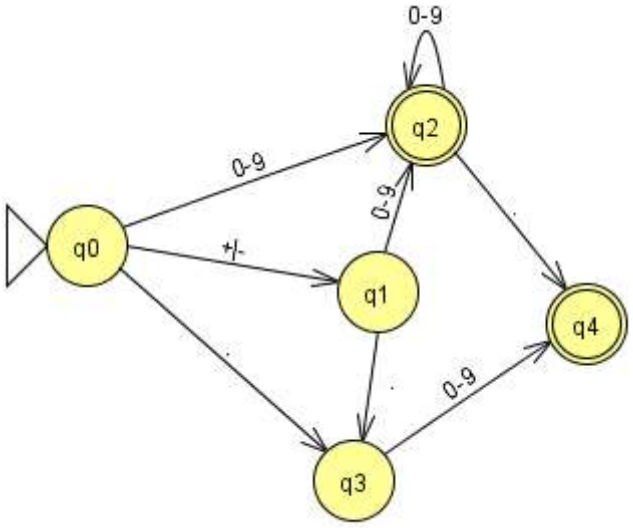
El método leerCodigo hace lo referente a abrir el archivo y leerlo a una cadena para poder ser analizada, se procede al método lexemas el cual divide el código fuente en lexemas a partir de los delimitadores explícitos e implícitos para por último dentro del método tokenizar, se llama a los autómatas correspondientes de decir si un lexema es correcto dentro del lenguaje y se puede generar un token y ser agregado a la tabla de símbolos o en su defecto agregar un error a la pila de errores.



```java
public class AnalisisLexico {
    TablaSimbolos tablaSimbolos = new TablaSimbolos();
    PilaErrores pilaErrores = new PilaErrores();

    //MÉTODO QUE SEPARA EL PROGRAMA EN LEXEMAS
    public void lexemas(){...}

    //MÉTODO QUE CONVIERTE LOS LEXEMAS EN SUS RESPECTIVOS TOKENS
    int tokenizar(String lexema, int linea){...}

    //MÉTODO QUE LEE EL CÓDIGO FUENTE DEL ARCHIVO
    String leerCodigo(String archivo){...}
}
```

Cada autómata se encuentra implementado en un objeto diferente para tener una mejor organización de los mismos, cuenta en la parte superior los valores que puede regresar el autómata, si regresa un error o el id de un token se deja de llamar a los autómatas siguientes y se agrega a la tabla de símbolos, en caso contrario, continua con el siguiente autómata y se procede a realizar el mismo procedimiento hasta terminar con todos los lexemas que conforman el código fuente.

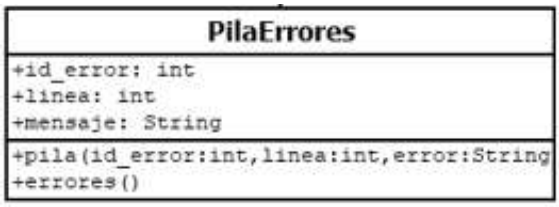
Las siguientes imágenes muestran la estructura del autómata, así como de la clase que lo conforma, esto se repite en cada autómata que se implementó, el siguiente ejemplo corresponde a los números.



```java
public class Numero {
    //55     Entero
    //56     Real
    //103    Error
    //105    Error
    //-1     No valido

    String entrada;
    int tama;
    int actual;

    public int comprobar(String entrada){...}

    int q0(){...}

    int q1(){...}

    int q2(){...}

    int q3(){...}

    int q4(){...}
}
```

Clase "PilaErrores"

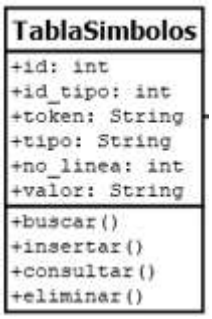
La pila de errores al mostrar lo contenido, muestra todo lo almacenado dentro de ella, imprimiendo en pantalla el id del error, la línea en la cual se genero y el correspondiente mensaje de error.




```java
public class PilaErrores {
    public Stack<Error> pila = new Stack<~>();

    //MÉTODO PARA INGRESAR ALGÚN ERROR ENCONTRADO EN LA PILA DE ERRORES
    public void ingresa(int id,int linea) { pila.push(new Error(id,linea)); }

    //MÉTODO PARA MOSTRAR LO CONTENIDO EN LA PILA DE ERRORES
    public void mostrar(){...}

    //MÉTODO PARA IMPRIMIR EL CORRESPONDIENTE MENSAJE DE ERROR
    public String listaErrores(Error e){...}
}
```

Clase "TablaSimbolos"

Al hacer uso de la tabla de simbolos podemos insertar nuevos registros dentro de ella, estos registros solo serán insertados si es un token valido, si es invalido paso a ser ingresado el error en la pila de errores, podemos consultar un registro en específico a través del método llamado buscar por medio de su identificador, así como podemos mostrar todos los registros a través de consultar y vaciar nuestra tabla de símbolos con eliminar.




```
public class TablaSimbolos {
    private ArrayList<Simbolo> tablaSimbolos = new ArrayList<>();

    //MÉTODO QUE MUESTRA LO CONTENIDO EN LA TABLA DE SIMBOLOS
    public void consultar(){...}

    //MÉTODO QUE PERMITE INGRESAR UN REGISTRO EN LA TABLA DE SIMBOLOS
    public void ingresar(Simbolo s){...}

    //MÉTODO QUE VACIA LA TABLA DE SIMBOLOS
    public void eliminar() { tablaSimbolos.clear(); }

    //MÉTODO QUE PERMITE BUSCAR UN REGISTRO DENTRO DE LA TABLA DE SIMBOLOS
    public Simbolo buscar(int id){...}
}
```

Para ejemplificar el comportamiento de nuestro analizador léxico veamos el siguiente código de ejemplo que incorpora la mayoría de los errores contemplados en nuestro diseño.

```
/*Iniciamos nuestro programa*/
programa $prueba {
    /*Marcara un error 100, ya que hace falta el signo $ en el identificador*/
    ent pru;
    /*Marcara un error 102, ya que el signo é no pertenece al lenguaje*/
    rea $x,$y,$rés;
    /*Marcara un error 103, ya que los números no pueden contener carácteres extraños o letras*/
    ent $nument = 32A;
    /*Marcara un error 104, ya que no se puede establecer un identificador con un $ en medio*/
    cad $numero$Real = "hola";

    principal () {
        /*Marcara un error 104, ya que no se pueden formar decimales con dos puntos*/
        $x=765476.4.2;
        si($x>=$y) {
            $res = $x * $y;
            salida($res);
        }
        sino{
            $res = 2 * $x;
            salida($res);
        }
    }
}
```

Al ejecutarlo nos genera una tabla de símbolos como la siguiente.

```
=============================================================================================
ID      TOKEN   LEXEMA          LINEA   VALOR
0       1       programa        2
1       70      $prueba         2
2       61      {               2
3       50      ent             4
4       60      ;               4

                                        ......

57      62      }               22
58      62      }               23
59      62      }               24
=============================================================================================
```

Como vemos se nos generaron 59 tokens, además de los siguientes errores como estaba comentado dentro del programa ejemplo.

```
================================================================================
ID      LINEA   MENSAJE
100     4       No puede definirse como un identificador, debe de iniciar con un signo de $
102     6       El token no se puede establecer como identificador ni palabra reservada, no pertenece al lenguaje
103     8       Las constantes numéricas no pueden mezclarse con letras o caracteres extraños
104     10      No se puede establecer como identificador
105     14      Los números decimales no son válidos
================================================================================
```

Al corregir los errores y ejecutar, vemos lo siguiente, los 5 errores que teníamos desaparecieron y los tokens se agregaron dentro de la tabla de símbolos, ahora tenemos 64 registros dentro de ella.

```
61      60      ;               21
62      62      }               22
63      62      }               23
64      62      }               24
================================================================================
```

Al analizar la tabla de símbolos antes de la ejecución vemos lo siguiente:

```
2       61      i               2
3       50      ent             4
4       60      ;               4
5       51      rea             6
```

Comparandolo con la ejecución ya corregida, podemos ver que nuestro token con lexema $pru, ya se encuentra dentro de nuestra tabla de simbolos a comparación de cuando nos generaba un error.

```
2       61      i               2
3       50      ent             4
4       70      $pru            4
5       60      ;               4
6       51      rea             6
```

BITACORA DE INCIDENCIAS

Fecha	Hora	Descripción de la incidencia	Solución
Martes 25 de Febrero del 2020	7:00 AM	falta establecer algunos puntos de la notación formal del lenguaje , el profesor realizo las observaciones pertinentes en clase al momento de exponer el trabajo	El equipo se dio a la tarea de reunirse para integrar mediante una notación formal (BNF) los elementos de su lenguaje estos ya habían sido establecidos en la actividad numero 1 . Pero faltaba ser mas explícitos en el caso de las expresiones así que se delegaron tareas a cada integrante
Jueves 27 de Febrero del 2020	8:00 AM	se solicito elaborar la tabla de símbolos del lenguaje prototipo y la pila de errores	Los miembros del equipo se reunieron para discutir de manera pertinente los elementos que contendrían dichas estructuras, cada uno aporto conocimiento mediante una lluvia de ideas y una vez que se estableció esto se delegaron las tareas para formalizar lo que se había comentado en la reunión de trabajo.
Martes 3 de Marzo del 2020	8:00 AM	falta elaborar los autómatas correspondientes a las expresiones regulares y el maestro de igual manera solicito implementar las matrices de estado correspondientes a cada autómata . se realizaron unas pequeñas observaciones en la tabla de símbolos respecto a los campos	se descargo el software JFLAP para elaborar en este nuestros autómatas ,cada miembro realizo la instalación ya que se repartieron de manera equitativa dichos autómatas , esto fue una tarea sencilla ya que como se tenia bien formalizada la gramática y las expresiones regulares simplemente se diseñaron los autómatas correspondientes y se evaluaron en el mismo software , las pequeñas observaciones respecto a la tabla de símbolos se realizaron al momento en la misma reunión que se tuvo entre los miembros del equipo después de la revisión del profesor.
Martes 10 de Marzo del 2020	8:00 AM	se solicitó la elaboración del modelado uml del analizador léxico , y plasmar la manera en la que interactúan los elementos de este análisis en nuestros diagramas previos a la programación , también diseñar algunos programas muestra y plasmar los casos de uso en cuanto a los errores léxicos	Para la elaboración del modelado se fijó una reunión y prácticamente lo que se realizo fue bajo una metodología SCRUM

Martes 17 de Marzo del 2020	12:00 AM	se emitió un comunicado por parte de la escuela en donde informaba a cerca de la suspensión de labores a partir del 20 de marzo debido a la pandemia por el covid-19 , justo después de que el profesor publicara la solicitud de la actividad 2	El equipo el día martes asistió a la institución para tratar temas relacionados con la actividad ante la situación y se tomaron cartas en el asunto , Se decidió seguir trabajando bajo el mismo enfoque de esta metodología ágil de desarrollo , establecer un tablero con sprints para la planificación de las actividades semanales , delegar tareas priorizarlas y delegar roles para el desarrollo del software.