



DEPARTAMENTO DE SISTEMAS COMPUTACIONALES E INFORMÁTICA

ASUNTO: **Solicitud de Actividades**

Celaya, Gto., 2018-02-23

LENGUAJES Y AUTÓMATAS II

DOCENTE DESIGNADO: ISC. RICARDO GONZÁLEZ GONZÁLEZ

ACTIVIDAD 2 (VALOR 70 PUNTOS)

LEA CUIDADOSAMENTE, Y REALICE LAS SIGUIENTES ACTIVIDADES, CONSIDERANDO LOS CRITERIOS DE CALIDAD PROPUESTOS EN LOS DOCUMENTOS DE LA GUÍA TUTORIAL, Y LA RÚBRICA DE EVALUACIÓN.

1. TOMANDO EN CUENTA TODAS LAS INDICACIONES Y EXPLICACIONES EN CLASE Y UNA VEZ DEFINIDA LA GRAMÁTICA Y EL LENGUAJE PROTOTIPO A IMPLEMENTAR, COMO EQUIPO INVESTIGUEN, DISEÑEN E IMPLEMENTEN LA PRIMERA FASE O ETAPA DE UN ANALIZADOR LÉXICO.
2. LA ACTIVIDAD COMO EQUIPO DEBERÁ COMENZAR CON LA INVESTIGACIÓN Y FUNDAMENTACIÓN DE LOS SIGUIENTES TEMAS.
 - A. INVESTIGAR. ¿ QUÉ ES UN ANÁLISIS LÉXICO APLICADO A LA VALORACIÓN DE UN LENGUAJE ?
 - B. INVERTIGAR. ¿ EN QUÉ CONSISTE UN ANÁLISIS LÉXICO Y QUÉ LO CARACTERIZA?
 - C. IDENTIFICAR. ¿ QUÉ CASOS DE ESTUDIOS SON LOS IMPORTANTES A CONSIDERAR EN EL ANÁLISIS LÉXICO ?
 - D. INVESTIGAR Y PROPOSER. ¿ QUÉ PROCESOS Y PROBLEMAS ATIENDE UN ANÁLISIS LÉXICO ?
 - E. INVESTIGAR. ¿ CÓMO IMPLEMENTAR UN ANÁLISIS LÉXICO ?

NOTA : ESTOS TEMAS DEBEN SER EL RESULTADO DE UNA INVESTIGACIÓN Y ANÁLISIS COMO EQUIPO, Y NO UNA TRANSCRIPCIÓN DE TEMAS AISLADOS, PUES EN TODO MOMENTO LAS FUENTES DE CONSULTA DEBEN SER LA BASE DEL DESARROLLO DE ESTE PUNTO Y SUS APARTADOS.



Antonio García Cubas Pte. #600 esq. Av. Tecnológico Col. Alfredo V. Bonfil C.P. 38010
Celaya, Gto. AP 57, Comutador: (461)6117575. Correo electrónico: lince@itcelaya.edu.mx

www.itcelaya.edu.mx





3. COMO EQUIPO Y DERIVADO DEL ANÁLISIS ANTERIOR SE DEBEN PROPONER LOS ALGORITMOS Y ESTRUCTURAS DE DATOS NECESARIAS PARA LA IMPLEMENTACIÓN DE UN PROTOTIPO DE ANALIZADOR LÉXICO. ES DECIR, MANEJO Y OPERACIONES CON ARCHIVOS, TABLA DE SÍMBOLOS, DISEÑO DE AUTÓMATAS, PILA DE ERRORES, ETC.

CONCRETAMENTE EN ESTE PUNTO DEBERÁN COMO EQUIPO, REDACTAR Y DETALLAR TODOS LOS ELEMENTOS QUE SE USARÁN PARA LA IMPLEMENTACIÓN DEL ANALIZADOR LÉXICO.

4. PARA EL INCISO C DEL PUNTO 2 ANTERIOR, SE DEBERÁN CREAR PROGRAMAS DE MÍNIMO 25 LÍNEAS (SIN CONSIDERAR LOS COMENTARIOS) CADA UNO, EN LOS CUALES SE CODIFIQUE EN EL LENGUAJE PROTOTIPO, INSTRUCCIONES CON LÓGICA QUE EJEMPLIFIQUEN LOS ERRORES QUE EN CADA CASO DE ESTUDIO SE PROPONGAN.

TALES PROGRAMAS DEBEN ESTAR PERFECTAMENTE DOCUMENTADOS Y CORRELACIONADOS CON LOS CASOS DE ESTUDIO CORRESPONDIENTES.

5. CARACTERÍSTICAS QUE LA ACTIVIDAD 2 DEBE POSEER PARA CONSIDERARSE COMPLETA.

- A. EL FUNDAMENTO DE LA GRAMÁTICA A UTILIZAR, ASÍ COMO SU DEFINICIÓN FORMAL Y EL ALFABETO A UTILIZAR.
- B. LA CARECTERIZACIÓN DEL LENGUAJE PROTOTIPO, ES DECIR SU DESCRIPCIÓN MEDIANTE NOTACIÓN BNF. SE DEBEN INCLUIR ADEMÁS LOS SÍMBOLOS ESPECIALES COMO DELIMINADORES IMPLÍCITOS Y EXPLÍCITOS, OPERADORES LÓGICOS, RELACIONALES Y ARITMÉTICOS.
- C. CATEGORIZACIÓN E IDENTIFICACIÓN POR ID, DE CADA UNO LOS ELEMENTOS QUE EL LENGUAJE PROTOTIPO PROPONE.
- D. PLANTEAMIENTO Y FUNDAMENTACIÓN DE LOS CASOS DE USO DEL ANALIZADOR LÉXICO, ASI COMO LOS ERRORES EN QUE DERIVARÁ CADA UNO DE ELLOS.
- E. PREPARACIÓN DE LOS PROGRAMAS SUFICIENTES, ESCRITOS EN EL LENGUAJE PROTOTIPO, QUE APOYEN LA COMPROBACIÓN DE CADA CASO DE ESTUDIO. TALES PROGRAMAS DEBERÁN ESTAR COMPLETAMENTE DOCUMENTADOS.
- F. GENERACIÓN DE UN CATÁLOGO DE ERRORES, CORRELACIONADO A LOS CASOS DE USO PROPUESTOS.
- G. DISEÑO DE UNA SOLUCIÓN MODELADA EN OBJETOS, APOYADA EN DIAGRAMAS UML QUE DESCRIBAN LA ARQUITECTURA PROPUESTA PARA EL PROTOTIPO DEL ANALIZADOR LÉXICO.
- H. PROPUESTA Y MODELADO DE LAS PROPIEDADES Y COMPORTAMIENTOS DE UNA TABLA DE SÍMBOLOS, ASÍ COMO LA JUSTIFICACIÓN DE CADA UNA DE LAS COLUMNAS QUE LA INTEGREN.





- I. PROPUESTA Y MODELADO DE LAS PROPIEDADES Y COMPORTAMIENTOS DE UNA PILA DE ERRORES.
- J. MODELADO DE LOS PROCESOS DE APERTURA Y LECTURA DEL ARCHIVO DE CÓDIGO FUENTE, ASÍ COMO DEL PROCESO DE TOKENIZACIÓN (QUE NO DEBERÁ IMPLEMENTARSE A PARTIR DE FUNCIONALIDADES DE BIBLIOTECAS DE TERCEROS).
- K. DISEÑO, PRUEBAS, MODELADO E IMPLEMENTACIÓN DE LOS AUTOMÁTAS SUFICIENTES Y NECESARIOS PARA LA CATEGORIZACIÓN DE CADA UNO DE LOS TOKENS GENERADOS.
- L. MODELADO E IMPLEMENTACIÓN DE UN PROCESO DE :
- LECTURA DE CÓDIGO FUENTE => TOKENIZACIÓN => CATEGORIZACIÓN DE LOS TOKENS => CONTRUCCIÓN Y LLENADO DE LA TABLA DE SIMBOLOS => MANEJO Y DESPLIEGUE DE ERRORES => DESPLIEGUE DE LA TABLA DE SÍMBOLOS RESULTANTE.
- M. GENERACIÓN DE UN CALENDARIO DE ACTIVIDADES PLANIFICADAS VS. ACTIVIDADES REALIZADAS.
- N. GENERACIÓN DE UNA BITÁCORA DE INCIDENCIAS.
- * O. TODAS LAS EVIDENCIAS GENERADAS Y REUNIDAS DEBERÁN INTEGRARSE AL UN ARCHIVO PDF, NOMBRADO COMO SE INDICA MÁS ADELANTE.
ESTAS EVIDENCIAS PODRÁN ELABORARSE CON HERRAMIENTAS ELECTRÓNICAS, COMO PROCESADOR DE TEXTO, DE IMÁGENES, HOJAS DE CÁLCULO, ETC.
- P. EL NÚCLEO DE CADA ALGORITMO CODIFICADO TAMBIÉN DEBERÁ FORMAR PARTE DEL ARCHIVO DE EVIDENCIAS.

NOTAS GENERAL DE LA ACTIVIDAD:

- SE DEBE CONSIDERAR Y TOMAR EN CUENTA PARA EL CORRECTO CUMPLIMIENTO DE ESTA ACTIVIDAD, LO SOLICITADO EN LA GUÍA TUTORIAL, CONCRETAMENTE EN EL PUNTO 3 INCISO i (Trabajo en equipo).
- SE DEBE CONSIDERAR Y TOMAR EN CUENTA PARA EL CORRECTO CUMPLIMIENTO DE ESTA ACTIVIDAD LO SOLICITADO EN LA GUÍA TUTORIAL, CONCRETAMENTE EN EL PUNTO 6, (Evidencias tipo a).



**OBSERVACIONES:**

- ✓ LA REVISIÓN SERÁ EN DIVERSAS VERTIENTES. PUEDE SER AL MOMENTO DE SOLICITAR LA CARPETA DE EVIDENCIAS, O BIEN PUEDE SER AL SOLICITAR LA EXPOSICIÓN DE LA TAREA EN LA CLASE. TAMBIÉN PUEDE SER POR SOLICITUD EXPRESA DEL INTERESADO A PARTICIPAR EN CLASE EXPONIENDO BREVEMENTE SU ACTIVIDAD.
- ✓ AQUELLAS ACTIVIDADES EN FORMATO DIGITAL SE DEBERÁN TENER SIEMPRE, Y EN TODO MOMENTO A LA MANO EN UNA MEMORIA USB.
- ✓ ÉSTAS ACTIVIDADES PODRÁN SER SOLICITADAS EN LA CLASE, O BIEN PARA SU ENVÍO A UNA CUENTA DE CORREO.
- ✓ ESTAS ACTIVIDADES DEBEN ESTAR LISTAS E INTEGRADAS A LA CARPETA DE EVIDENCIAS (FÍSICAMENTE) A LA FECHA DE ENTREGA INDICADA AL FINAL DE ÉSTE DOCUMENTO.
- ✓ CADA HOJA QUE ENTREGUE DE SU ACTIVIDAD, DEBERÁ ESTAR FIRMADA AL MARGEN DERECHO.
- ✓ UNA VEZ ELABORADA SU ACTIVIDAD, RECUERDE DIGITALIZARLA Y NOMBRARLA EN BASE A LA SIGUIENTE NOMENCLATURA.
- ✓ SI SUS EVIDENCIAS ENVIADAS POR CORREO, NO CUMPLEN CON LA NOMENCLATURA SOLICITADA, NO SERÁN CONSIDERADAS COMO EVIDENCIAS PARA SU EVALUACIÓN.
- ✓ CON ESTA ACTIVIDAD, USTED DEBERÁ IR INTEGRANDO SUS CARPETAS FÍSICA Y ELECTRÓNICA DE EVIDENCIAS, Y AL FINAL DEL SEMESTRE EN UN DISCO COMPACTO HARÁ ENTREGA DE SU CARPETA ELECTRÓNICA DE EVIDENCIAS.
- ✓ PARA TENER DERECHO A LA REVISIÓN Y EVALUACIÓN DE SUS ACTIVIDADES, DEBE REGISTRAR SU ASISTENCIA A CLASE, EL DÍA SEÑALADO PARA LA ENTREGA DE LA MISMA.
- ✓ FALTAR A CLASE EL DÍA DE LA ENTREGA, ANULA LA REVISIÓN DE SUS EVIDENCIAS.
- ✓ POR ÚLTIMO, POR FAVOR GESTIONE APROPIADAMENTE SU TIEMPO, Y SEA PUNTUAL EN SU ENTREGA.
- ✓ AÚN PARA TRABAJOS EN EQUIPO APlican TODAS LAS MISMAS OBSERVACIONES ANTERIORES.



**LA NOMENCLATURA SOLICITADA ES :**

AAAA-MM-DD_MATERIA_DOCUMENTO_EQUIPO_NOCTROL_APELLIDOS_NOMBRE_SEM.PDF

(NOTA : * TODO EN MAYÚSCULA ***)****DONDE :**

AAAA : AÑO
 MM : MES
 DD : DIA
 MATERIA : SO, TSO, LAII, LI
 DOCUMENTO : A1-ACTIVIDAD 1, P1-PRACTICA 1, R1-REPORTE 1, T1-TAREA 1, PG1-PROGRAMA,
 ETC. (CAMBIANDO EL NÚMERO CONSECUITIVO POR EL QUE CORRESPONDA)
 EQUIPO : NÚMERO DEL EQUIPO QUE CORRESPONDA SEGÚN INDICACIÓN DEL PROFESOR.
 NOCTROL : SU NÚMERO DE CONTROL
 APELLIDOS : SUS APELLIDOS
 NOMBRE : SU NOMBRE
 SEM : EL PERIODO SEMESTRAL EN CURSO: ENE-JUN / AGO-DIC

EJEMPLO :

2018-03-12_LAII_A2_MEXICO_EQUIPO_99_9999999_PEREZ_PEREZ_JUAN_ENE-JUN18.PDF

FECHA DE ENTREGA:

**VÍA CORREO ELECTRÓNICO, EL LUNES 12 DE MARZO DEL 2018, CON HORA LÍMITE DE ENTREGA
HASTA LAS 14:00 HORAS (2 DE LA TARDE).**

**DESPUÉS DE ESTA HORA, LA ACTIVIDAD SERÁ CONSIDERADA COMO EXTEMPORÁNEA Y NO
CONTARÁ COMO EVIDENCIA PARA SU EVALUACIÓN.**

MUY IMPORTANTE:

**POR FAVOR ANEXE A SU ARCHIVO .PDF DE EVIDENCIAS, ESTA SOLICITUD DE ACTIVIDADES CON
TODAS LAS HOJAS FIRMADAS EN EL MARGEN DERECHO.**

CALENDARIO 2017-2018 - EVALUACIÓN 2

FEBRERO						
L	M	M	J	V	S	D
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28				

MARZO						
L	M	M	J	V	S	D
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

EVALUACIÓN FORMATIVA DE 1º OPORTUNIDAD (PARCIALES)



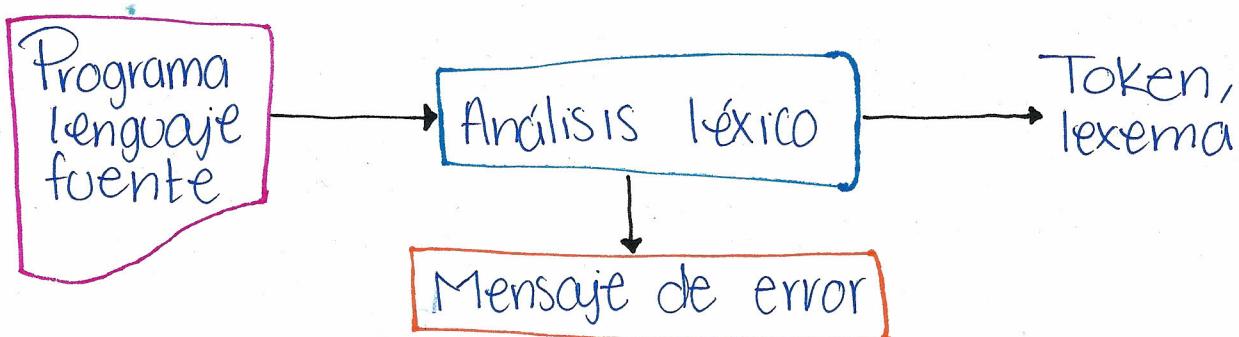
2. A.

¿Qué es un análisis léxico aplicado a la valoración de un lenguaje?

Léxico

El léxico de un lenguaje natural son todas las palabras y símbolos que lo componen. En un lenguaje de programación en léxico lo constituye todos los elementos individuales del lenguaje (token).

Análisis léxico aplicado a la valoración de un lenguaje.



Es la primera etapa de la compilación, que se encarga de leer el código fuente y separarlo en tokens, verifica que los tokens sean pertenecientes del lenguaje y la gramática, si es así el token es válido y será utilizado por la siguiente etapa de la compilación.

Referencias:

<https://www.api-developer.com/2014/01/conceptos-del-analisis-lexico.html>

<http://www.paginasprodigy.com/edserna/cursos/compiador/Notas/Notas2.pdf>.

2.B ¿EN QUÉ CONSISTE UN ANALIZADOR LÉXICO Y QUÉ LO CARACTERIZA?

ANALIZADOR LÉXICO

COMO YA SE MENCIONÓ ANTERIORMENTE, EL ANALIZADOR LÉXICO CONSISTE EN LEER LOS CARÁCTERES DE ENTRADA Y ELABORAR COMO SALIDA UNA SECUENCIA DE COMPONENTES LÉXICOS QUE UTILIZA EL ANALIZADOR SINTÁCTICO PARA HACER EL ANÁLISIS. LEYENDO EL PROGRAMA FUENTE, CARÁCTER POR CARÁCTER, Y CONSTRUIRE A PARTIR DE ÉSTE UNAS ENTIDADES PRIMARIAS LLAMADAS TOKENS. ES DECIR, EL ANALIZADOR LEXICOGRÁFICO TRANSFORMA EL PROGRAMA FUENTE EN TIRAS DE TOKENS, VALIDANDO QUE CADA TOKEN CUMPLA CON EL ALFABETO, GRAMÁTICA Y LENGUAJE ESTABLECIDO. PARA POSTERIORMENTE llenar la TABLA DE SÍMBOLOS.

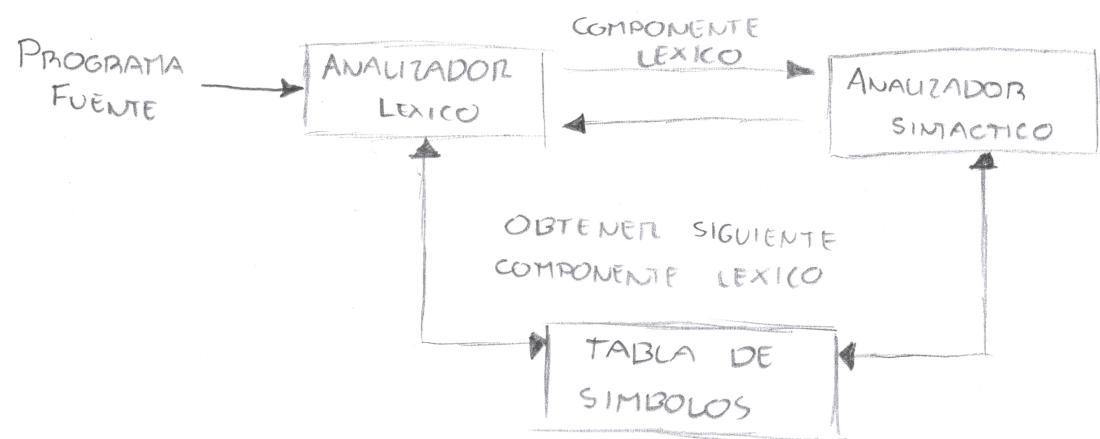
EN LA FASE DE ANÁLISIS, LOS TERMINOS COMPONENTES LÉXICOS (TOKENS), PATRÓN Y LEXEMA SE EMPLEAN CON SIGNIFICADOS ESPECÍFICOS. UN ANALIZADOR LÉXICO, INICIALMENTE LEE LOS LEXEMAS Y LE ASIGNA UN SIGNIFICADO PROPIO.

- **COMPONENTE LÉXICO:** Es la secuencia lógica y coherente de caracteres relativo a una categoría: IDENTIFICADOR, PALABRA RESERVADA, LITERALES (CADENA/NUMÉRICA), ENTRE OTROS.
- **PATRÓN:** Regla que genera la secuencia de caracteres que puede representar a un determinado componente léxico. (EXPRESIÓN REGULAR).
- **LEXEMA:** Es una CADENA DE CARÁCTERES QUE CONCUERDA CON UN PATRÓN QUE DESCRIBE COMPONENTE LÉXICO (VALOR DE CADENA).

LEXEMA	COMPONENTE LÉXICO	PATRÓN
DOUBLE	PALABRA RESERVADA	DOUBLE
=	OPERADOR	<,<=, =, =>, >
P1	IDENTIFICADOR	LETRA SEGUIDA DE LETRAS O NÚM
3.1416	CONSTANTE NUM.	LITERAL NUMÉRICA
"Hola Mundo"	CADENA	CADENAS ENTRE COMILLAS.

¿QUE LO CARACTERIZA?

EL ANALIZADOR LÉXICO, OPERA BAJO PETICIÓN DEL ANALIZADOR SINTÁCTICO DEVOLVIENDO UN COMPONENTE LÉXICO SEGÚN EL ANALIZADOR SINTÁCTICO LO VAYA NECESITANDO, ES DECIR QUE CUANDO EL ANALIZADOR LÉXICO RECIBE LA ORDEN DE OBTENER EL SIGUIENTE COMPONENTE LÉXICO, EL ANALIZADOR LÉXICO LEE LOS CARÁCTERES DE ENTRADA HASTA IDENTIFICAR EL SIGUIENTE COMPONENTE LÉXICO.



FUENTES BIBLIOGRÁFICAS

WWW.LCC.UMA.ES/~CALVEZ/FTP/TCI/TICTEMA2.PDF
ES.WIKIPEDIA.ORG/WIKI/ANALIZADOR-LEXICO

Identificadores repetidos

Supongamos que el código fuente se definen dos variables, y que estas variables tienen el mismo nombre.

Al momento de identificar la primera de las dos variables esta será agregada a la tabla de símbolos, en el momento es que sea identificada la segunda variable primero se hará una búsqueda en la tabla de símbolos y si el identificador ya está almacenado no será agregado nuevamente a la tabla.

Marcar este tipo de ocurrencias como errores será tarea de otras fases del compilador.

Identificadores inválidos

Si en el código fuente se define un identificador que no siga la expresión regular /[_\$a-Z][\$_a-Z0-9]*/ se tomará como un identificador inválido.

El analizador deberá marcar error cuando se encuentren estos casos. El número del error será el 110.

Identificación de palabras reservadas

Si en el código fuente se encuentran tokens que coincidan con alguna de las palabras reservadas del lenguaje, se tomarán como tal.

El analizador no generará error, ya que si no se identifica un palabra reservada el token puede ser considerado como un identificador.

Caracteres no definidos en el alfabeto

Si al momento de generar una palabra reservada, crear un identificador, una constante, etc. se utiliza un símbolo que no está definido en nuestro lenguaje esto generará un error.

El analizador generará un error si se identifica un carácter que no esté definido en el alfabeto el número del error será el 120.

Identificación de constantes

En el código fuente la generación de las constantes debes de ser de la manera en como están definidas en nuestra gramática para así no generar un error.

Si una constante no cumple con la gramática entonces el analizador generará un error, el número del error será 110.

```
#####
#      CASO DE ESTUDIO 1          #
#      Identificadores repetidos  #
#####

func fact ( int num ) : int {           //primera vez que aparece num
    if ( num == 1 ) {                  //segunda vez que aparece num
        return 1 ;
    } else {
        return num * fact ( num - 1 ) ; //tercera vez
    }
}

func fib ( int num ) : int {           //cuarta vez, y continúa apareciendo
    if ( num == 1 ) {
        return 1 ;
    } else if ( num == 2 ) {
        return 1 ;
    } else {
        return fib ( num - 1 ) + fib ( num - 2 ) ;
    }
}

int num = 0 ;
int fib ;
int fact ;
println "Dame un numero para hacer unos calculos:" ;
print ">" ;
read num ;
fact = fact ( num ) ;                //aquí, fact se repite otras dos veces
fib = fib ( num ) ;                 //y aquí se repite fib
println "El factorial de " + num + " es " + fact ;
println "El numero de fibonacci de " + num + " es " + fib ;
```

```
#####
#      CASOS DE ESTUDIO 2 y 4          #
#      IDENTIFICADORES INVÁLIDOS      #
#  CARACTERES NO INCLUIDOS EN EL ALFABETO  #
#####

string Geschäft = null ;           // Dentro de los caracteres del identificador
                                    // ä es inválido
string nombre = null ;
int edad = 0 ;
int opcion = 0 ;
double peso = 0 ;

while ( opcion != 1 ) {

    println "Bienvenido" ;
    println "Ingrese el nombre" ;
    read nombre ;
    println "Ingrese el nombre del negocio" ;
    read Geschäft ;
    println "Ingrese la edad" ;
    read edad ;
    println "Ingrese el peso" ;
    read peso ;

    print "¿Fueron correctos los datos ingresados?\n1.-Si\n2.-No" ;
    if ( opcion == 1 ) {
        print "Los siguientes datos fueron los siguientes :" ;
        print "Nombre : " + nombre ;
        print "Negocio : " + Geschäft ;
        print "Edad : " + edad ;
        print "Peso : " + peso ;
    }
}
println "спасиба" ;           // Gracias en ruso, son caracteres inválidos
```

```
#####
#      CASO DE ESTUDIO 3      #
# IDENTIFICACIÓN DE PALABRAS RESERVADAS #
#####

string1 g = 89 ;           // string1 sería considerado identificador
                           // y no palabra reservada
string num = 7.9 ;          // En este caso string es considerado palabra reservada

int letra_a = "Casa(45)"   //int es considerado como palabra reservada

if ( g = 0 ) {
    print "int g es igual a 0" ;           //Las palabras reservadas en el
                                             //programa son string, int,
                                             //for, println, print, read,
                                             //for, func, if
}

for ( g in 1 -> 1 ) {
    g = g-1
}
pritnln "nuevo valor de g" ;
read g ;

numeros ( g ) ;

func numeros1 ( g ) : intt {           //intt no es considerado palabra reservada
    println "Dame otro valor:" ;
    read num ;
    println "Dame mas valores:" ;
    read num_1 ;
    read $num2 ;
    sum= num + g ;
    print sum ;
}

numeros ( g int x ) ;

print "DANKE" ;
letra_a = letra_a + "lll" ;
```

```
#####
#      CASO DE ESTUDIO 5          #
#      IDENTIFICACIÓN DE CONSTANTES    #
#####

func sum ( int num1 , int num2 ) : int {
    int suma ;
    suma = num1 + num2 ;
    return suma ;
}
func res ( int num1 , int num2 ) : int {
    int resta ;
    resta = num1 - num2 ;
    return resta ;
}
func mult ( real num3 , real num4 ) : real {
    real multi ;
    multi = num3 * num2 ;
    return multi ;
}
func div ( real num3 , real num4 ) : real {
    real divi ;
    divi = num3 / num4 ;
    return divi ;
}
int num1 = 15 ;           //La constante 15 es válida como entero
int num2 = 02 x 15u ;     //La constante 02x15u no es válida como entero
real num3 = 15.02 ;       //La constante 15.02 es válida como decimal
real num4 = 16,42 ;       //La constante 16,42 no es válida como decimal
int suma = sum ( num1 , num2 ) ;
int resta = res ( num1 , num2 ) ;
real multi = mult ( num3 , num4 ) ;
real divi = div ( num3 , num4 ) ;
print ( suma , resta , multi , divi ) ;
```

② D Procesos y problemas del análisis léxico

El análisis léxico presenta una serie de procesos que deben ser atendidos, como lo son:

- La **tokenización**. Transformar el código fuente en unidades (**tokens**) significativos: identificadores, constantes, operadores, entre otros.
- La **remoción de comentarios**, además de **espacios**.
- El manejo de **mayúsculas y minúsculas**.
- El manejo de la **tabla de símbolos**. Introducir información acerca de los identificadores encontrados.

De igual manera, existen ciertos problemas y consideraciones que deben ser tomados en cuenta. Algunas de ellas son:

- Las **ambigüedades**. ¿Cómo especificar tokens o patrones? ¿Es "if" una palabra reservada o un identificador?
- ¿Cómo reconocer los tokens basándose en una especificación dada?

Nuestro lenguaje

Tomando en cuenta lo antes mencionado, se han tomado varias decisiones respecto a nuestro lenguaje y la implementación de su analizador léxico.

En breve, para el reconocimiento de los tokens se hará uso de autómatas y expresiones regulares. Deberá existir una prioridad durante el análisis para evitar ambigüedades; intentar reconocer palabras reservadas antes de identificadores, por ejemplo.

Detalles específicos de la implementación se discutirán más adelante.

Fuentes

cs.fsu.edu/~xyuan/cop5621/lect2_lexical.ppt (c. el 01/03/2018)

web.cs.udavis.edu/~pandey/Teaching/ECS142/Lects/lex.anal.pdf (c. el 01/03/2018)

Kjleach.eecs.umich.edu/c18/l3.pdf (c. el 01/03/2018)

2. E) Cómo implementar un análisis léxico?

Aspectos prácticos en la implementación de un analizador léxico

Principio de máxima longitud: Se da prioridad al componente léxico de máxima longitud.

Ejemplo: ende se interpreta como el identificador Ende y no como la palabra reservada end y la letra e.

Las palabras reservadas se reconocen como identificadores y se comprueba antes de decidir el tipo de token si se trata de una palabra reservada o un identificador.

Usar un tipo enumerado para los tipos de componentes léxicos. Usar un tipo enumerado para los estados del analizador.

Se mantienen dos apuntadores; Uno marca el inicio del lexema y el otro el carácter actual que se move.

Implementación

- Convertir el programa de entrada en una secuencia de componentes léxicos
- Se usa la de Finición léxica para diseñar un reconocedor del lenguaje. Esto es a través de expresiones regulares y autómatas. Despues implementarlo segun el diseño.
- Considerar las expresiones regulares de cada categoría léxica.
- Hacer un autómata para cada una de las expresiones regulares.
- Codificar el autómata como un programa

Bibliografías

- webdiis.unizar.es/~ezpeleta/lib/exe/fetch.php?media=missdetos2.analisislexico.pdf
- www.fdi.ucm.es/profesor/fpeinat/courses/compiling/tema1.6-Implementacion.pdf
- prezi.com/ypkeep2lelk/IMPLEMENTACION-de-un-analizador-lexical/

La gramática a utilizar es una gramática de tipo 3. Esta gramática fue elegida por la necesidad y conveniencia de traducir la gramática en expresiones regulares, las cuales facilitan el manejo y validación de las cadenas.

De la misma forma, con base en las expresiones regulares, se derivan los autómatas a utilizar.

Otra de las razones para utilizar una gramática tipo tres es para evitar cadenas vacías y asegurarse que siempre haya una producción.

Cabe destacar que el alfabeto a utilizar serán todos los caracteres imprimibles del código ASCII.

Ejemplo: definición formal de un entero con gramática tipo 3, según la gramática definida para nuestro lenguaje prototipo:

$$G = \langle V_n, V_t, P, \Sigma \rangle$$

$$V_n = \{\Sigma, \phi\}$$

$$V_t = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$$

$$\begin{array}{lll} P = \{ & \Sigma \rightarrow 1\phi & \phi \rightarrow 1\phi \\ & \Sigma \rightarrow 2\phi & \phi \rightarrow 2\phi \\ & \Sigma \rightarrow 3\phi & \phi \rightarrow 3\phi \\ & \Sigma \rightarrow 4\phi & \phi \rightarrow 4\phi \\ & \Sigma \rightarrow 5\phi & \phi \rightarrow 5\phi \\ & \Sigma \rightarrow 6\phi & \phi \rightarrow 6\phi \\ & \Sigma \rightarrow 7\phi & \phi \rightarrow 7\phi \\ & \Sigma \rightarrow 8\phi & \phi \rightarrow 8\phi \\ & \Sigma \rightarrow 9\phi & \phi \rightarrow 9\phi \\ & \Sigma \rightarrow 0\phi & \phi \rightarrow 0\phi \end{array} \quad \begin{array}{l} \phi \rightarrow 1 \\ \phi \rightarrow 2 \\ \phi \rightarrow 3 \\ \phi \rightarrow 4 \\ \phi \rightarrow 5 \\ \phi \rightarrow 6 \\ \phi \rightarrow 7 \\ \phi \rightarrow 8 \\ \phi \rightarrow 9 \\ \phi \rightarrow 0 \end{array} \quad \begin{array}{l} \phi \rightarrow 1 \\ \phi \rightarrow 2 \\ \phi \rightarrow 3 \\ \phi \rightarrow 4 \\ \phi \rightarrow 5 \\ \phi \rightarrow 6 \\ \phi \rightarrow 7 \\ \phi \rightarrow 8 \\ \phi \rightarrow 9 \\ \phi \rightarrow 0 \end{array} \quad \}$$

```
<bloque> ::= "{" <sentencias> "}"  
  
<sentencias> ::= <sentencia> <sentencia>*  
<sentencia> ::= <expresión> | <llamada método> | <ciclos> | <condicionales>  
| <return>  
  
<llamada a método> ::= <id> (<parámetros>) ;  
<método> ::= func <id> (<argumentos>) [: <tipo>] "{" <sentencias> "}"  
<parámetros> ::= <id> | <número> | <decimal> | <cadena> | <parámetros> ,  
<id> | <parámetros> , <número> | <parámetros> , <decimal> | <parámetros> ,  
<cadena>  
<argumentos> ::= <tipo> <id> | <argumentos>, <tipo> <id>  
<retorno> ::= return | return <id> | return <número> | return <decimal> |  
return <cadena> | return <booleano>  
<ciclos> ::= <ciclo for> | <ciclo while>  
<condicionales> ::= <if then else> | <if then>  
  
<if then else> ::= <if then> else <bloque> | <if then> else <if then else>  
<if then> ::= if (<expresión relacional>) <bloque>  
  
<ciclo for> ::= for (<id> in <rango>) <bloque>  
<ciclo while> ::= while (<expresión relacional>) <bloque>  
<rango> ::= <número> -> <número> | <número> -> <id> | <id> -> <número> |  
<id> -> <id>  
  
<lectura> ::= read <id> ;  
<impresion> ::= (print|println) <id>; | (print) <cadena> ;  
  
<expresión> ::= <asignación> | <expresión relacional>  
  
<asignación> ::= int <id> = <numero> ; | real <id> = <decimal>; | string  
<id> = <cadena> ; | bool <id> = <booleano> ; | bool <id> = <expresión  
relacional> ;  
<expresión relacional> ::= (<id>|<número>|<decimal>) <operador relacional>  
(<id>|<número>|<decimal>) | <cadena> <operador relacional> <cadena>  
<operador relacional> ::= < | > | <= | >= | == | !=
```

```
<operador asignador> ::= = | += | -= | *= | /= | **
```

```
<operador> ::= + | - | * | / | ** | ++ | --
```

```
<id> ::= (<letra>|_)$)(<letra>|<dígito>|_)$)*
```

```
<decimal> ::= <número>."<número>
```

```
<número> ::= <dígito><dígito>*
```

```
<cadena> ::= \"<símbolo>*\\"
```

```
<booleano> ::= true | false
```

```
<letra> ::= <letra minúscula> | <letra mayúscula>
```

```
<letra minúscula> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n  
| ñ | o | p | q | r | s | t | u | v | w | x | y | z
```

```
<letra mayúscula> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N  
| Ñ | O | P | Q | R | S | T | U | V | W | X | Y | Z
```

```
<dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<delimitadores> ::= ; | " " | \t
```

```
<comentario> ::= //<símbolo>*\n | #<símbolo>*\n
```

```
<tipo> ::= int | real | bool | string
```

```
<reservadas> ::= if | else | while | for | func | return | true | false |  
null | int | real | bool | string | print | println | read
```

```
<símbolos> ::= <caracteres ascii imprimibles>
```

Palabras reservadas

IDs del 1 al 16

TOKEN	LEXEMA/REGEX	ID
T_IF	if	1
T_ELSE	else	2
T_WHILE	while	3
T_FOR	for	4
T_FUNC	func	5
T_RETURN	return	6
T_TRUE	true	7
T_FALSE	false	8
T_NULL	null	9
T_INT	int	10
T_REAL	real	11
T_BOOL	bool	12
T_STRING	string	13
T_PRINT	print	14
T_PRINTF	printf	15
T_READ	read	16

Identificadores y constantes

IDs del 17 al 22

TOKEN	LEXEMA/REGEX	ID
T_VAR	[_\$a-Z][_\$a-Z0-9]*	17
T_FUN	[_\$a-Z][_\$a-Z0-9]*\(\)	18
T_INT_CONST	[0-9][0-9]*	19
T_REAL_CONST	[0-9][0-9]*\.[0-9][0-9]*	20
T_BOOL_CONST	true false	21
T_STR_CONST	\“[\x20-\x7E]*\”	22

Símbolos y operadores

IDs del 23 al 46

TOKEN	LEXEMA/REGEX	ID
T_COLON	:	23
T_SEMICOLON	;	24
T_LPAREN	(25
T_RPAREN)	26
T_LBRACE	{	27
T_RBRACE	}	28
T_LBRACKET	[29
T_RBRACKET]	30
T_LTE	<=	31
T_GTE	>=	32

T_NE	!=	33
T_LT	<	34
T_GT	>	35
T_EQ	==	36
T_ASSIGN	=	37
T_DEC	--	38
T_INC	++	39
T_NOT	!	40
T_PLUS	+	41
T_MINUS	-	42
T_STAR	*	43
T_DIV	/	44
T_RANGE	->	45
T_COMMA	,	46

Errores del 1 al 99 “reservados para el sistema”

Todo error relacionado a procesos del sistema, como errores de permisos, de lectura de archivos, y otros.

- **Error 10.** Error de archivo inexistente.
 - Este error se presenta cuando un archivo al que se hace referencia no existe.
- **Error 11.** Error de lectura de archivo.
 - Se presenta cuando ocurre un error de entrada/salida al leer cualquier archivo.

Errores del 100 al 199 “léxicos”

Todo error relacionado a la etapa del análisis léxico entra en esta categoría. Son pocos, pero hay espacio para expansión futura.

- **Error 110.** Error de token inesperado
 - Se presenta cuando el analizador no puede reconocer el token después de compararlo con todos los casos posibles.
- **Error 120.** Error uso de símbolos no definidos en el alfabeto.
 - El error se presenta si en cualquier parte del código fuente es detectado un símbolo que no pertenece al lenguaje.

Errores del 200 al 299 “sintácticos”

Se definirán en la siguiente etapa.

Errores del 300 al 399 “semánticos”

Se definirán en la última etapa.

Tabla de símbolos

La tabla de símbolos será global y no habrá diferencia de contexto. Los símbolos deberán ser únicos, de modo que no se repita el nombre. En la tabla de símbolos, se añadirá a cada registro el nombre o lexema, el identificador del token al que pertenece y el tipo de dato que sea.

Se hará uso de una tabla hash para guardar los registros. La llave será de tipo `String` usando el nombre del identificador, de modo que los identificadores no se repitan.

Las operaciones que se realizarán en la tabla de símbolos son: inserción de un registro y búsqueda en la tabla.

Registro de la tabla de símbolos

CAMPO	DESCRIPCIÓN
<code>String NOMBRE</code>	Es el nombre o lexema que será almacenado en la tabla de símbolos.
<code>int TOKEN_ID</code>	Es el identificador del token que ha sido reconocido, de acuerdo al catálogo.
<code>int TIPO</code>	El tipo de dato del lexema si es identificador; o si es una función lo que regresa. Se usará en las siguientes etapas.

Pila de errores

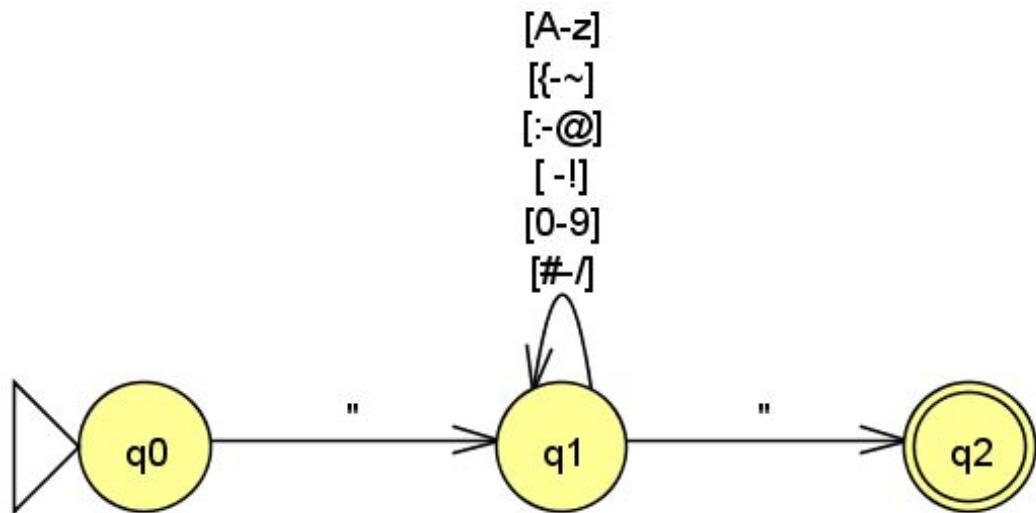
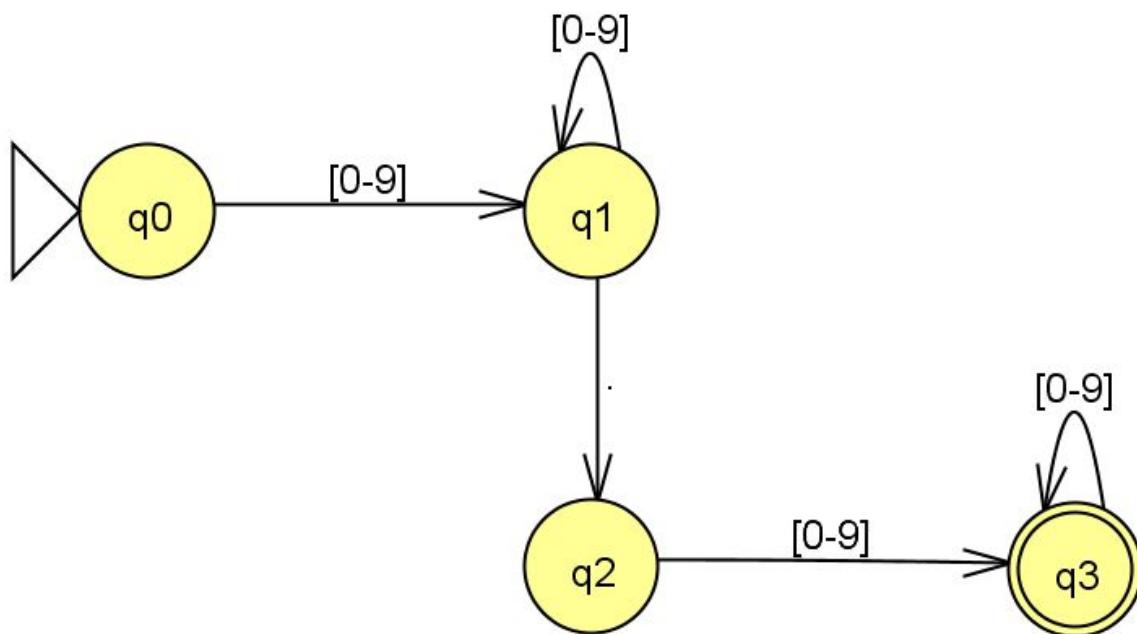
La pila de errores contará con 3 campos que serán agrupados en un TDA. En la pila se agregarán los errores encontrados dentro del código fuente, especificando en el la línea, el identificador del error y el lexema.

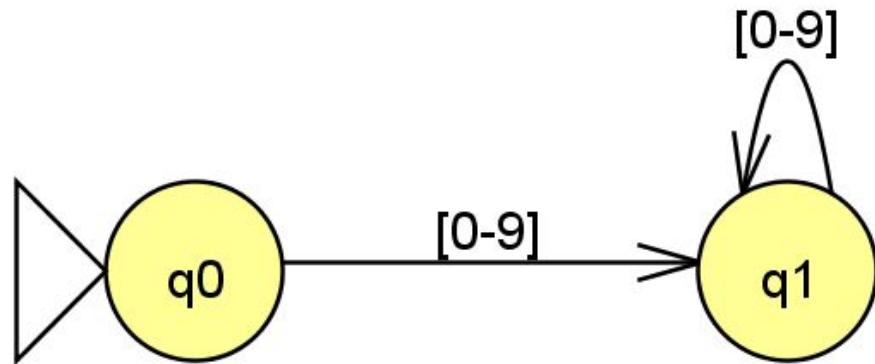
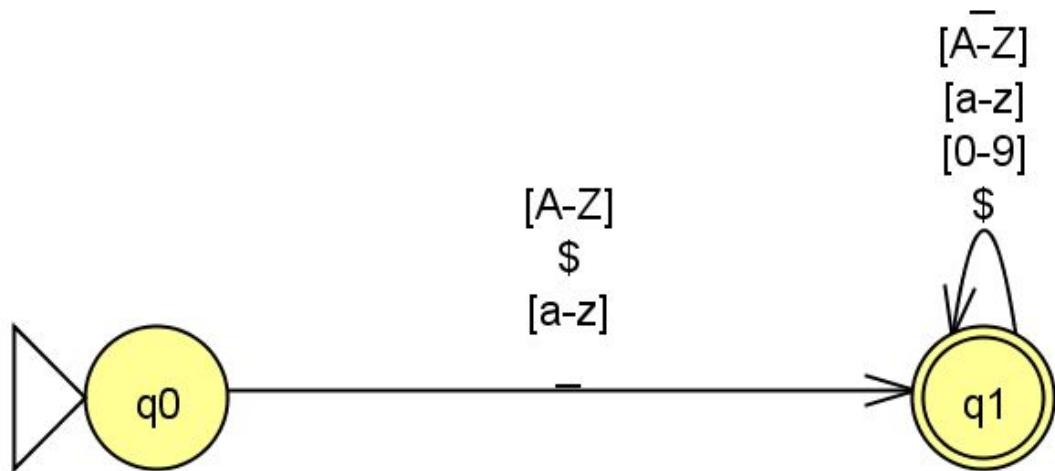
Como se puede inferir por el nombre, la estructura de datos a usar será una pila, que contendrá los registros.

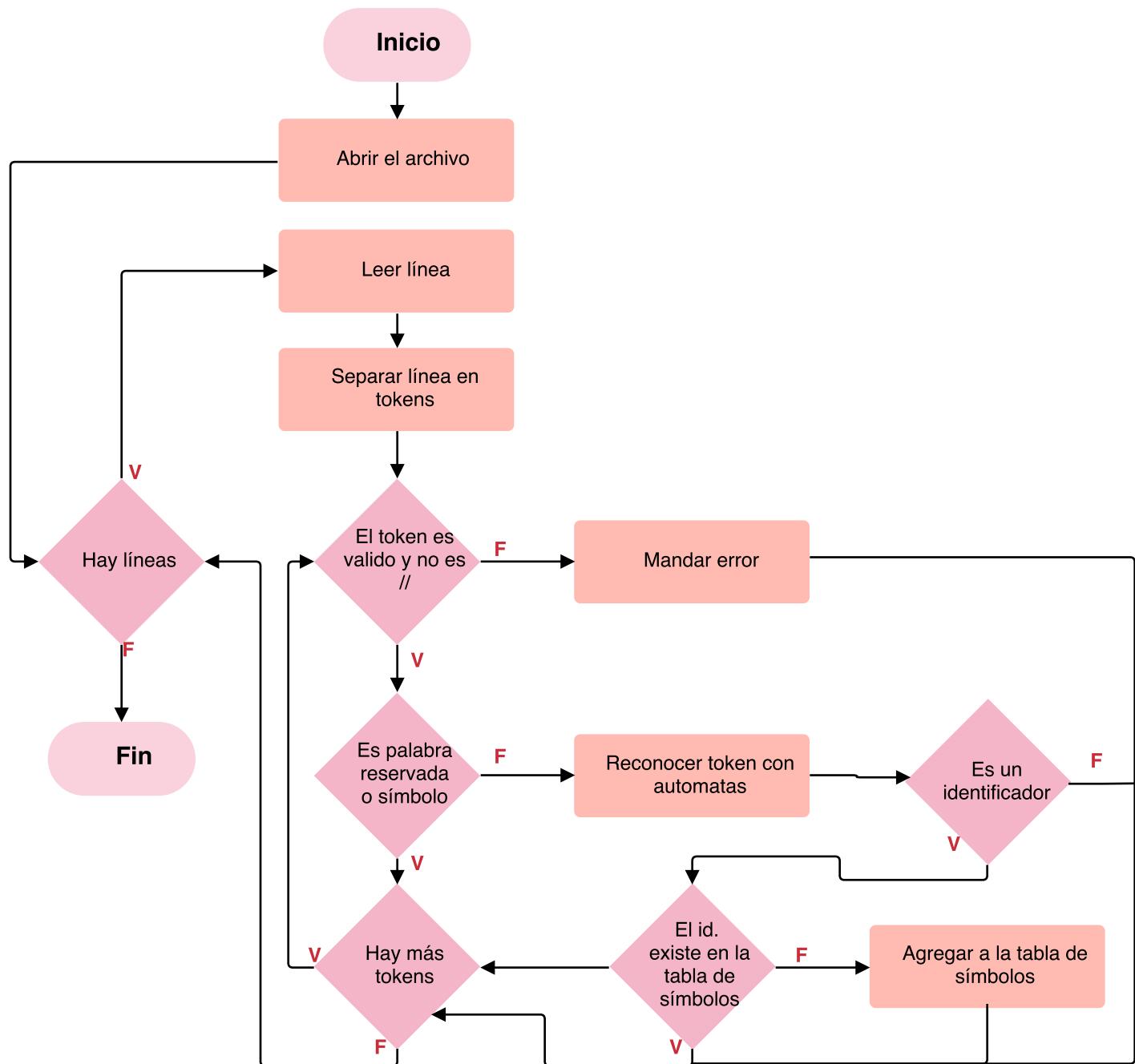
Las operaciones que se realizarán en la pila de errores serán meter y sacar los TDAs que representarán a los errores que se irán presentando.

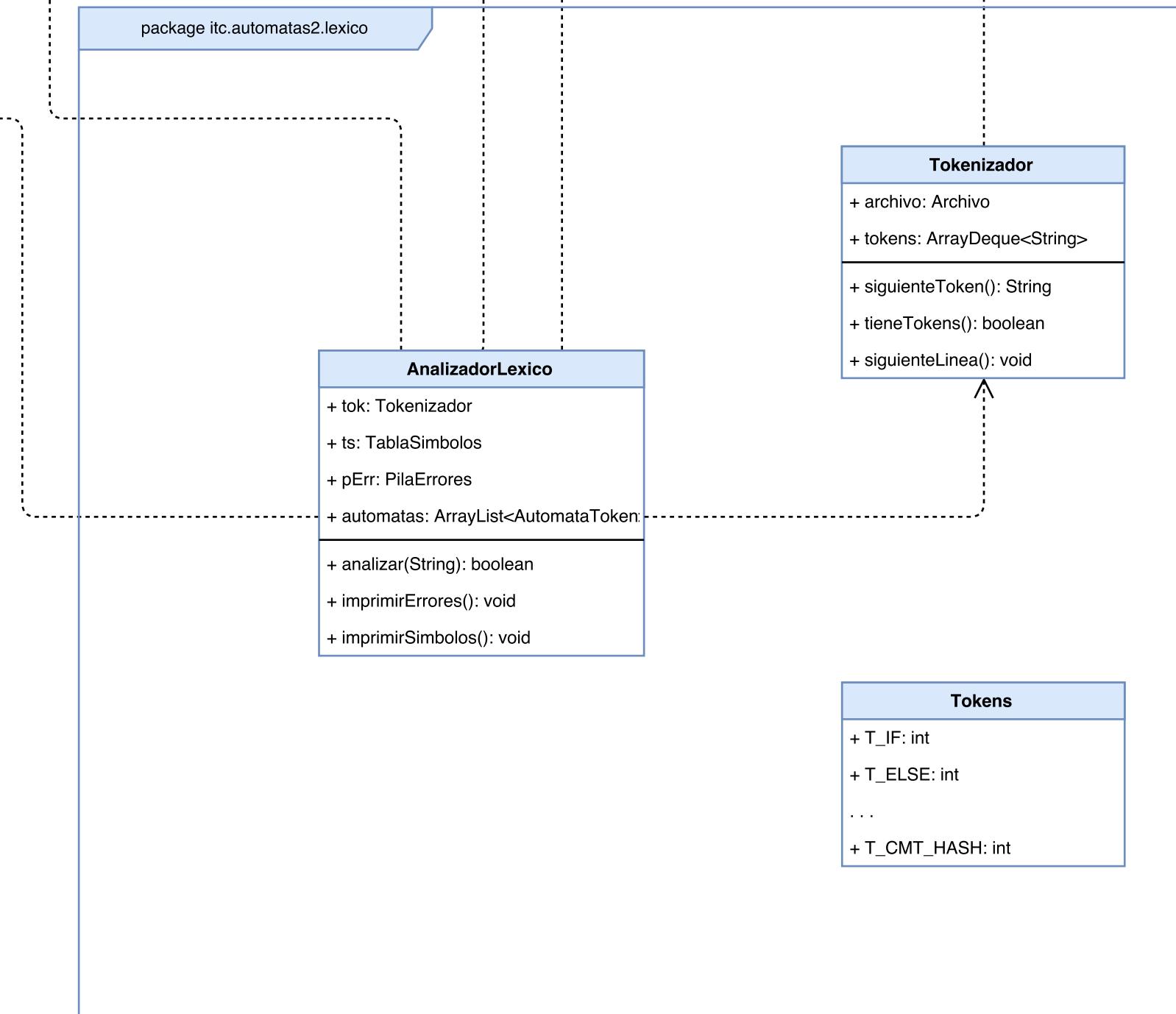
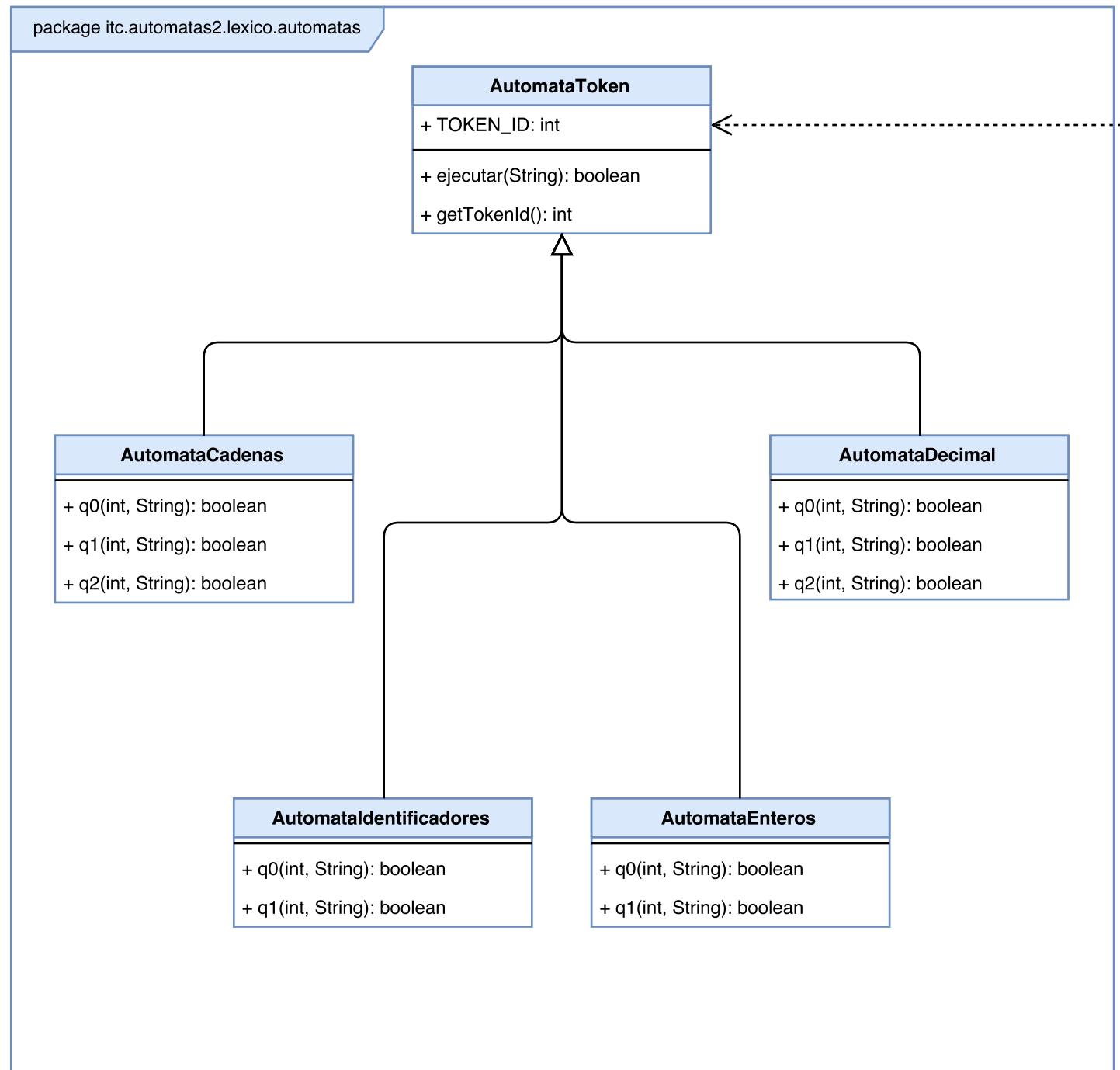
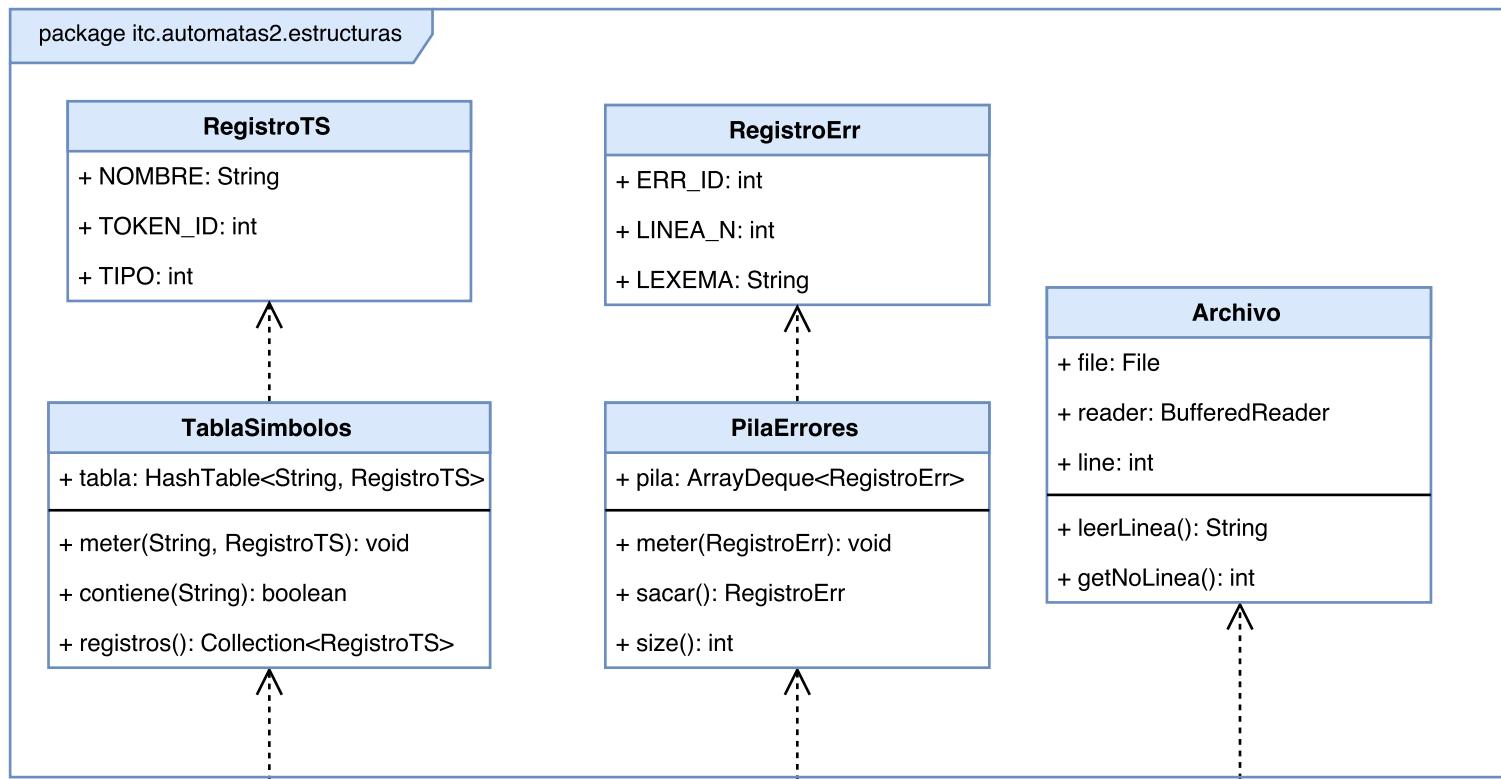
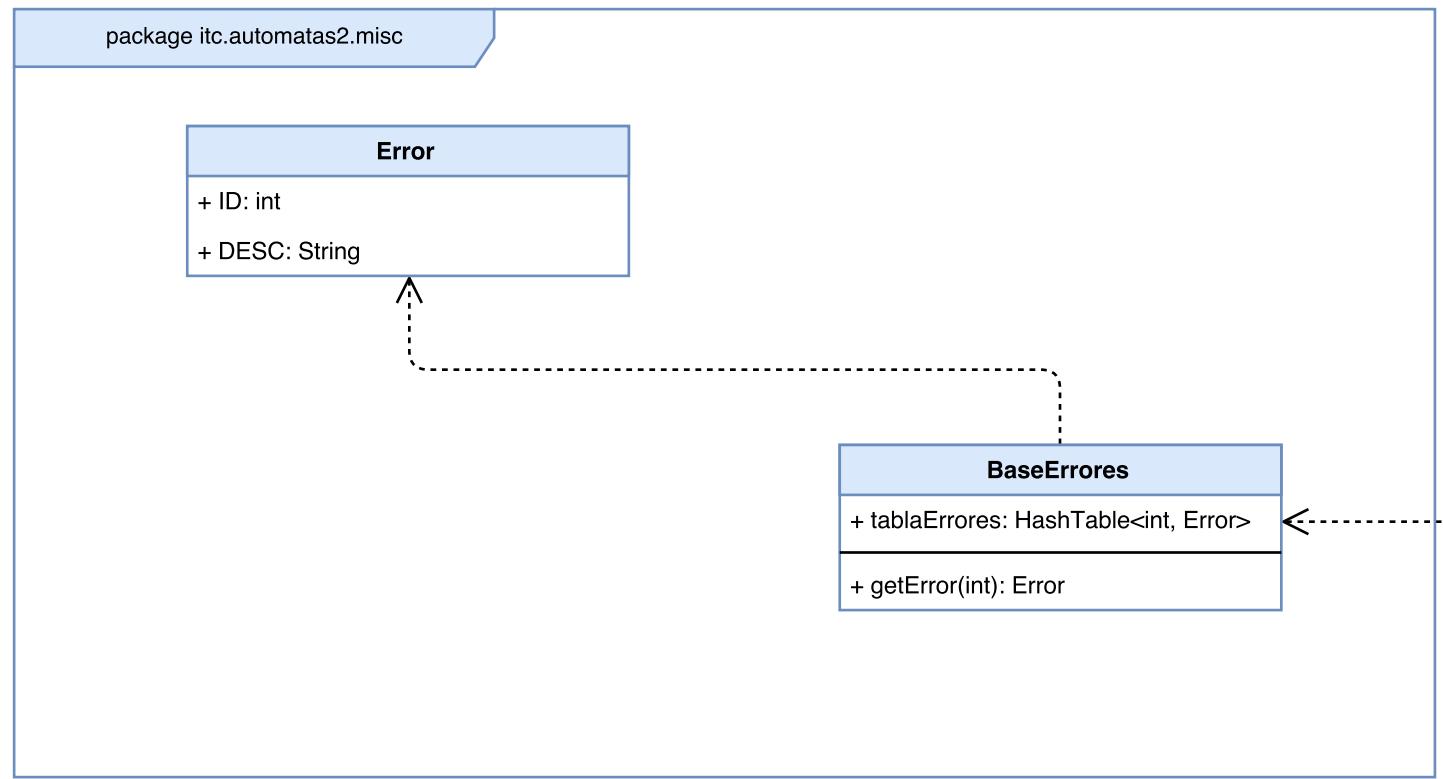
Registro de la pila de errores

CAMPO	DESCRIPCIÓN
int ERR_ID	A cada error le corresponde un identificador que describe el tipo de error.
int LINEA_N	Es el número de la línea en donde es identificado el error.
String LEXEMA	El lexema que produjo el error.

Autómata para el reconocimiento de una cadena**Autómata para el reconocimiento de números decimales**

Autómata para el reconocimiento de números enteros**Autómata para el reconocimiento de identificadores**





EQUIPO 1		ACTIVIDAD 2														LENGUAJES Y AUTÓMATAS II					
ACTIVIDADES / FASES		TIEMPOS POR ACTIVIDAD														TOTAL POR ACTIVIDAD		OBSERVACIONES			
		23/02	24/02	25/02	26/02	27/02	28/02	01/03	02/03	03/03	04/03	05/03	06/03	07/03	08/03	09/03	10/03	11/03	12/03	PLANEADO	REAL
Actividades iniciales																					
Lectura inicial de la solicitud de actividad		80															80	60			
		60																			
Investigación del marco teórico y discusión acerca de los temas		360		360	360												1,080	720			
		240		240	240																
Análisis																					
Planteamiento del problema						60											60	60			
						60															
Escritos del marco teórico						240											240	480	Contratiempos externos.		
						360	120														
Correcciones a la propuesta del lenguaje								40									40	40			
								40													
Revisión de la gramática								60									60	90	Se presentaron dificultades menores.		
								90													
Diseño																					
Definición de casos de estudio							480										480	480			
							480														
Modelado de estructuras de datos							240										240	360			
							360														
Categorización de tokens								120									120	80			
								80													
Categorización de errores								120									120	120			
								120													
Diseño de autómatas								120									120	180	Ver bitácora de incidencias.		
								180													
Diagramas de flujo									240								240	360			
									360												
Diagramas de clases									480								480	450			
									450												
Construcción																					
Implementación de componentes									480	480							960	1,680	Durante la integración se encontraron errores en el código, así que se le dedicó tiempo extra.		
									480	240											
Integración de componentes										480	480						960	960	Debido a la extensión en la implementación, se trabajó otro día no asignado.		
										240	240	480									
Actividades finales																					
Recopilación de la documentación (sólo responsable del equipo)																	30	30			
																	30				
Presentación de la actividad (sólo responsable del equipo)																	10	10			
																	10				
														TOTALES		minutos		5320	6160		
														horas/hombre		22.17		25.67			

1. Por cruce de horarios, en ocasiones no se pudo trabajar presencialmente, así que se decidió trabajar remotamente.

28 de febrero del 2018

2. Se encontraron dificultades al definir el tipo de gramática a utilizar en nuestro lenguaje.
3. Frecuentemente, se confunde el concepto de Analizador Léxico, con las otras etapas de compilación.

01 de marzo del 2018

4. Durante la definición de nuestro alfabeto, hubo incertidumbre en aceptar todos los caracteres del código ASCII en el alfabeto, ya que se pensaba que podría complicar el trabajo en cuanto a la formalización de los autómatas.

05 de marzo del 2018

5. Al momento de modelar los autómatas en JFLAP, se presentó un pequeño contratiempo en cuanto a simbolizar que una transición nos representará todos los símbolos del código ASCII.

08 de marzo del 2018

6. Se decidió unir el error de identificador inválido y el de constante mal formada en un solo error llamado token inesperado, debido a complicaciones en la validación.

10 de marzo del 2018

7. Se detectaron errores menores en la integración del código
8. Se cambió la tokenización de un split de cadenas a valorización carácter por carácter, debido a dificultades al validar cadenas y aspectos no considerados.

Anexo

Código generado
durante esta etapa

File - Main.java

```
1 package itc.automatas2;
2
3 import itc.automatas2.lexico.AnalizadorLexico;
4
5 public class Main {
6     public static void main(String[] args) {
7         if (args.length == 0) {
8             System.err.println("ERROR: Proporcione la ruta del archivo a analizar.");
9             System.exit(1);
10    }
11    AnalizadorLexico a = new AnalizadorLexico();
12    System.out.printf("ANÁLISIS DEL PROGRAMA \"%s\"\n", args[0]);
13    if (a.analizar(args[0]))
14        System.out.println("El analizador declaró el código como válido");
15    else
16        System.out.println("El analizador declaró el código como inválido");
17    a.imprimirErrores();
18    a.imprimirSimbolos();
19}
20}
21
```

File - Error.java

```
1 package itc.automatas2.misc;
2
3
4 /**
5  * TDA para contener la descripción de un error.
6 */
7 public class Error {
8     public final int ID;
9     public final String DESC;
10    public String REASON;
11
12
13    Error(int ID, String DESC) {
14        this.ID = ID;
15        this.DESC = DESC;
16        this.REASON = "";
17    }
18
19 /**
20  * Establece la razón del error. Útil para imprimirlo en pantalla.
21  *
22  * @param reason La razón del error.
23  * @return El mismo error, por conveniencia.
24  */
25    public Error setReason(String reason) {
26        this.REASON = reason;
27        return this;
28    }
29
30 /**
31  * @return Una representación del error de la forma "ID: DESC REASON"
32  */
33    @Override
34    public String toString() {
35        return String.format("ERROR %d: %s %s", ID, DESC, REASON);
36    }
37 }
38 }
```

File - BaseErrores.java

```
1 package itc.automatas2.misc;
2
3 import java.util.Hashtable;
4
5 /**
6  * Catálogo de errores internos. Esta clase contiene la definición de cada error posible.
7 */
8 public class BaseErrores {
9     private static Hashtable<Integer, Error> tablaErrores = new Hashtable<>();
10
11    static {
12        tablaErrores.put(10, new Error(10, "El archivo al que se hace referencia no existe."));
13        tablaErrores.put(11, new Error(11, "Ocurrió un error de E/S al leer el código fuente."));
14        tablaErrores.put(110, new Error(110, "Token inválido."));
15        tablaErrores.put(120, new Error(120, "Uso de símbolos no definidos en el alfabeto."));
16    }
17
18    /**
19     * Obtiene un objeto {alink Error Error} de acuerdo al ID proporcionado.
20     *
21     * @param ID El identificador del error.
22     * @return Un objeto {alink Error Error} si existe, <code>null</code> si no.
23     */
24    public static Error getError(int ID) {
25        return tablaErrores.get(ID);
26    }
27 }
```

File - Tokens.java

```
1 package itc.automatas2.lexico;
2
3 /**
4  * Catálogo de tokens e IDs.
5 */
6 public class Tokens {
7
8     //Palabras reservadas
9     public static final int T_IF = 1;
10    public static final int T_ELSE = 2;
11    public static final int T WHILE = 3;
12    public static final int T_FOR = 4;
13    public static final int T_FUNC = 5;
14    public static final int T_RETURN = 6;
15    public static final int T_TRUE = 7;
16    public static final int T_FALSE = 8;
17    public static final int T_NULL = 9;
18    public static final int T_INT = 10;
19    public static final int T_REAL = 11;
20    public static final int T_BOOL = 12;
21    public static final int T_STRING = 13;
22    public static final int T_PRINT = 14;
23    public static final int T_PRINTLN = 15;
24    public static final int T_READ = 16;
25
26     //Identificadores y constantes
27    public static final int T_VAR = 17;
28    public static final int T_FUN = 18;
29    public static final int T_INT_CONST = 19;
30    public static final int T_REAL_CONST = 20;
31    public static final int T_BOOL_CONST = 21;
32    public static final int T_STR_CONST = 22;
33
34     //Símbolos y operadores
35    public static final int T_COLON = 23;
36    public static final int T_SEMICOLON = 24;
37    public static final int T_LPAREN = 25;
38    public static final int T_RPAREN = 26;
39    public static final int T_LBRACE = 27;
40    public static final int T_RBRACE = 28;
41    public static final int T_LBRACKET = 29;
42    public static final int T_RBRACKET = 30;
43    public static final int T_LTE = 31;
44    public static final int T_GTE = 32;
45    public static final int T_NE = 33;
46    public static final int T_LT = 34;
```

File - Tokens.java

```
47     public static final int T_GT = 35;
48     public static final int T_EQ = 36;
49     public static final int T_ASSIGN = 37;
50     public static final int T_DEC = 38;
51     public static final int T_INC = 39;
52     public static final int T_NOT = 40;
53     public static final int T_PLUS = 41;
54     public static final int T_MINUS = 42;
55     public static final int T_STAR = 43;
56     public static final int T_DIV = 44;
57     public static final int T_RANGE = 45;
58     public static final int T_COMMA = 46;
59
60
61 }
```

File - Tokenizador.java

```
1 package itc.automatas2.lexico;
2
3 import itc.automatas2.estructuras.Archivo;
4
5 import java.io.FileNotFoundException;
6 import java.io.IOException;
7 import java.util.ArrayDeque;
8
9 /**
10  * Clase que genera los tokens que seran utilizados por el analizador.
11 */
12 public class Tokenizador {
13     public Archivo archivo;
14     private ArrayDeque<String> tokens;
15
16     /**
17      * Constructor de la clase
18      *
19      * @param ruta
20      * @throws FileNotFoundException
21      */
22     public Tokenizador(String ruta) throws FileNotFoundException {
23         archivo = new Archivo(ruta);
24         tokens = new ArrayDeque<>();
25     }
26
27     /**
28      * Obtiene un token de la cola que la misma clase mantiene.
29      *
30      * @return Un token de la cola, <code>null</code> si se alcanzó el final del archivo.
31      * @throws IOException Si ocurre un error de lectura del archivo.
32      */
33     public String siguienteToken() throws IOException {
34         if (tieneTokens()) {
35             return tokens.remove();
36         }
37
38         return null;
39     }
40
41     /**
42      * Se salta a la siguiente linea, ignorando los tokens que existan en la cola.
43      */
44     public void siguienteLinea() {
45         tokens.clear();
46     }
```

File - Tokenizador.java

```
47
48 /**
49 * El metodo utiliza la linea que la clase Archivo le envia, separa la linea en tokens usando el espacio como delimitador.
50 *
51 * @return Regresa VERDADERO si aun hay tokens y regresa FALSO si ya no hay tokens con los que trabajar.
52 * @throws IOException Si ocurre un error en la lectura o el archivo ya se cerró.
53 */
54 private boolean tieneTokens() throws IOException {
55     if (tokens.isEmpty()) {//Si tokens esta vacia se llenara con tokens
56         String linea = archivo.leerLinea();
57         while (linea != null && linea.trim().length() == 0) {
58             linea = archivo.leerLinea();
59         }
60
61         if (linea == null)
62             return false;
63
64         //Bandera para saber si se está leyendo una cadena
65         boolean string = false;
66         StringBuilder sb = new StringBuilder();
67         char[] str = linea.toCharArray();
68         for (int i = 0; i < str.length; i++) {
69             if (i == str.length - 1) { //El último caracter
70                 //Si no se está leyendo un string y no es delimitador, o si se está leyendo un string
71                 if ((!string && !Character.toString(str[i]).matches("[ \t]")) || string)
72                     sb.append(str[i]);
73                 if (sb.length() > 0)
74                     tokens.add(sb.toString());
75             } else if (!string) { //No se está leyendo un string
76                 switch (str[i]) {
77                     //Delimitadores
78                     case ' ':
79                     case '\t':
80                         if (sb.length() > 0) {
81                             tokens.add(sb.toString());
82                             sb.setLength(0);
83                         }
84                         break;
85                     case ';':
86                         if (sb.length() > 0) {
87                             tokens.add(sb.toString());
88                             sb.setLength(0);
89                         }
90                         tokens.add(Character.toString(str[i]));
91                         break;
92                 //Cadena
```

File - Tokenizador.java

```
93             case '':
94                 string = true;
95                 sb.append(str[i]);
96                 break;
97             //Cualquier símbolo
98             default:
99                 sb.append(str[i]);
100                break;
101            }
102        } else { //Se está leyendo una cadena
103            sb.append(str[i]);
104            if (str[i] == '"') { //Si la cadena termina, se completa el token
105                string = false;
106                tokens.add(sb.toString());
107                sb.setLength(0);
108            }
109        }
110    }
111 }
112 return true;
113 }
114 }
```

File - AnalizadorLexico.java

```
1 package itc.automatas2.lexico;
2
3 import itc.automatas2.estructuras.PilaErrores;
4 import itc.automatas2.estructuras.RegistroErr;
5 import itc.automatas2.estructuras.RegistroTS;
6 import itc.automatas2.estructuras.TablaSimbolos;
7 import itc.automatas2.lexico.automatas.*;
8 import itc.automatas2.misc.BaseErrores;
9 import itc.automatas2.misc.Error;
10
11 import java.io.FileNotFoundException;
12 import java.io.IOException;
13 import java.util.ArrayList;
14
15 public class AnalizadorLexico {
16     private Tokenizador tok;
17     private TablaSimbolos tS;
18     private PilaErrores pErr;
19     private ArrayList<AutomataToken> automatas;
20
21     /**
22      * Constructor de la clase
23      */
24     public AnalizadorLexico() {
25         automatas = new ArrayList<>();
26         automatas.add(new AutomataCadena());
27         automatas.add(new AutomataDecimal());
28         automatas.add(new AutomataNumero());
29         automatas.add(new AutomataIdentificador());
30         tS = new TablaSimbolos();
31         pErr = new PilaErrores();
32     }
33
34
35     /**
36      * Analiza un archivo de código fuente.
37      *
38      * @param ruta La ruta del archivo.
39      * @return <code>true</code> si se aceptó el código, <code>false</code> si fue rechazado.
40      */
41     public boolean analizar(String ruta) {
42         String token;
43         boolean error = false;
44         try {
45             tok = new Tokenizador(ruta);
46             while ((token = tok.siguienteToken()) != null) {
```

File - AnalizadorLexico.java

```
47         // Bandera para saber si el token ya fue reconocido
48         boolean found = false;
49
50         // Si el token empieza como comentario se ignora por completo el resto de la linea
51         if (token.startsWith("//") || token.startsWith("#")) {
52             tok.siguienteLinea();
53             continue;
54         }
55
56         if (!token.matches("[\u0020-\u007E]+")) { // Se verifica que el token contenga simbolos del alfabeto (ascii)
57             pErr.meter(new RegistroErr(120, tok.archivo.getNoLinea(), token)); // Si no, se crea un nuevo error
58             error = true;
59             continue;
60         }
61
62         // Se reconocen palabras reservadas y simbolos primero
63         switch (token) {
64             case "if":
65             case "else":
66             case "while":
67             case "for":
68             case "func":
69             case "return":
70             case "true":
71             case "false":
72             case "null":
73             case "int":
74             case "real":
75             case "bool":
76             case "string":
77             case "print":
78             case "println":
79             case "read":
80             case ":":
81             case ";":
82             case "(":
83             case ")":
84             case "{":
85             case "}":
86             case "[":
87             case "]":
88             case "<=":
89             case ">=":
90             case "!=":
91             case "<":
92             case ">":
```

File - AnalizadorLexico.java

```
93             case "==":
94             case "=":
95             case "--":
96             case "++":
97             case "!=":
98             case "+":
99             case "-":
100            case "*":
101            case "/":
102            case "->":
103            case ",":
104                found = true;
105                break;
106            }
107
108        // Se prosigue con la evaluacion en los autómatas
109        if (!found) {
110            for (AutomataToken a : automatas) {
111                if (!found && a.ejecutar(token)) {
112                    switch (a.getId()) {
113                        case Tokens.T_VAR:
114                            if (!tS.contiene(token)) {
115                                tS.meter(token, new RegistroTS(token, Tokens.T_VAR, 0));
116                            }
117                            found = true;
118                            break;
119                        case Tokens.T_STR_CONST:
120                        case Tokens.T_INT_CONST:
121                        case Tokens.T_REAL_CONST:
122                            found = true;
123                            break;
124                    }
125                }
126            }
127        }
128        // Si el token no fue reconocido, se marca error.
129        if (!found) {
130            pErr.meter(new RegistroErr(110, tok.archivo.getNoLinea(), token));
131            error = true;
132        }
133    }
134 } catch (FileNotFoundException e) {
135     pErr.meter(new RegistroErr(10, 0, ruta));
136     error = true;
137 } catch (IOException e) {
138     pErr.meter(new RegistroErr(11, 0, ruta));
```

File - AnalizadorLexico.java

```
139         error = true;
140     }
141     return !error;
142 }
143
144 /**
145 * Imprime la pila de errores en la salida estándar.
146 */
147 public void imprimirErrores() {
148     if (pErr.size() > 0) {
149         System.err.println("Se encontraron errores durante el análisis léxico:");
150         while (pErr.size() > 0) {
151             RegistroErr reg = pErr.sacar();
152             Error err = BaseErrores.getError(reg.ERR_ID);
153             switch (reg.ERR_ID) {
154                 //Genéricos
155                 case 10:
156                     err.setReason(String.format("(RUTA: %s)", reg.LEXEMA));
157                     break;
158                 case 11:
159                     err.setReason(String.format("(RUTA: %s)", reg.LEXEMA));
160                     break;
161                 //Léxicos
162                 case 110:
163                     err.setReason(String.format("No se pudo identificar el token \"%s\" en la línea %d", reg.LEXEMA, reg.
LINEA_N));
164                     break;
165                 case 120:
166                     err.setReason(String.format("El token '%s' en la línea %d contiene símbolos no reconocibles (fuera del
código ASCII). ", reg.LEXEMA, reg.LINEA_N));
167                     break;
168                 }
169                 System.err.println(err);
170             }
171         } else {
172             System.out.println("No se encontraron errores durante el análisis léxico");
173         }
174     }
175
176 /**
177 * Imprime la tabla de símbolos en la salida estándar.
178 */
179 public void imprimirSimbolos() {
180     if (tS.size() > 0) {
181         System.out.println("-----");
182         System.out.println("| TABLA DE SÍMBOLOS |");
```

File - AnalizadorLexico.java

```
183         System.out.println("-----");
184         System.out.printf(" | %-30s | %8s | %12s |\\n", "NOMBRE O LEXEMA", "TOKEN ID", "TIPO DE DATO");
185         System.out.println("-----");
186         for (RegistroTS reg : tS.registros()) {
187             System.out.printf(" | %-30s | %8d | %12d |\\n", reg.NOMBRE, reg.TOKEN_ID, reg.TIPO);
188         }
189         System.out.println("-----");
190     } else {
191         System.out.println("No se encontraron símbolos en el código");
192     }
193 }
194 }
```

File - AutomataToken.java

```
1 package itc.automatas2.lexico.automatas;
2
3 /**
4  * Clase prototipo para un autómata, que será extendida en cada autómata necesario.
5 */
6 public abstract class AutomataToken {
7     private final int TOKEN_ID;
8
9     AutomataToken(int TOKEN_ID) {
10         this.TOKEN_ID = TOKEN_ID;
11     }
12
13    /**
14     * Ejecuta el autómata de acuerdo al token dado.
15     *
16     * @param token Un <code>String</code> que contiene el token.
17     * @return El estado de aceptación <code>(true|false)</code>.
18     */
19    public abstract boolean ejecutar(String token);
20
21    /**
22     * Obtiene el ID del token que el autómata reconoce.
23     *
24     * @return El ID del token definido en {alink itc.automatas2.lexico.Tokens Tokens}.
25     */
26    public int getTokenId() {
27        return TOKEN_ID;
28    }
29}
30}
```

File - AutomataCadena.java

```
1 package itc.automatas2.lexico.automatas;
2
3 import itc.automatas2.lexico.Tokens;
4
5 /**
6  * Autómata que reconoce un identificador (variable).
7 */
8 public class AutomataCadena extends AutomataToken {
9
10    public AutomataCadena() {
11        super(Tokens.T_STR_CONST);
12    }
13
14    public boolean ejecutar(String token) {
15        return q0(0, token.toCharArray());
16    }
17
18    private boolean q0(int cont, char[] car) {
19        if (cont == car.length) {
20            return false;
21        } else {
22            if (Character.toString(car[cont]).equals("\\")) {
23                return this.q1(cont + 1, car);
24            }
25        }
26        return false;
27    }
28
29    private boolean q1(int cont, char[] car) {
30        if (cont == car.length) {
31            return false;
32        }
33        if (Character.toString(car[cont]).matches("[A-Za-z0-9{~-:@#-/ -!]")) {
34            return this.q1(cont + 1, car);
35        }
36        if (Character.toString(car[cont]).equals("\\")) {
37            return this.q2(cont + 1, car);
38        }
39        return false;
40    }
41
42    private boolean q2(int cont, char[] car) {
43        return cont == car.length;
44    }
45}
```

File - AutomataNumero.java

```
1 package itc.automatas2.lexico.automatas;
2
3 import itc.automatas2.lexico.Tokens;
4
5 /**
6  * Autómata que reconoce una constante numérica (entero).
7 */
8 public class AutomataNumero extends AutomataToken {
9
10    public AutomataNumero() {
11        super(Tokens.T_INT_CONST);
12    }
13
14    public boolean ejecutar(String token) {
15        return q0(0, token.toCharArray());
16    }
17
18    private boolean q0(int cont, char[] car) {
19        if (cont == car.length) {
20            return false;
21        } else {
22            if (Character.toString(car[cont]).matches("[0-9]")) {
23                return q1(cont + 1, car);
24            } else {
25                return false;
26            }
27        }
28    }
29
30    }
31
32    private boolean q1(int cont, char[] car) {
33        if (cont == car.length) {
34            return true;
35        }
36        if (Character.toString(car[cont]).matches("[0-9]")) {
37            return q1(cont + 1, car);
38        } else {
39            return false;
40        }
41    }
42
43
44 }
```

File - AutomataDecimal.java

```
1 package itc.automatas2.lexico.automatas;
2
3 import itc.automatas2.lexico.Tokens;
4
5 /**
6  * Autómata que reconoce una constante decimal.
7 */
8 public class AutomataDecimal extends AutomataToken {
9
10    public AutomataDecimal() {
11        super(Tokens.T_REAL_CONST);
12    }
13
14    public boolean ejecutar(String token) {
15        return q0(0, token.toCharArray());
16    }
17
18    private boolean q0(int cont, char[] car) {
19        if (cont == car.length) {
20            return false;
21        }
22        if (Character.toString(car[cont]).matches("[0-9]")) {
23            return this.q1(cont + 1, car);
24        }
25        return false;
26    }
27
28    private boolean q1(int cont, char[] car) {
29        if (cont == car.length) {
30            return false;
31        }
32        if (Character.toString(car[cont]).matches("[0-9]")) {
33            return this.q1(cont + 1, car);
34        }
35        if (Character.toString(car[cont]).equals(".")) {
36            return this.q2(cont + 1, car);
37        }
38        return false;
39    }
40
41    private boolean q2(int cont, char[] car) {
42        if (cont == car.length) {
43            return false;
44        }
45        if (Character.toString(car[cont]).matches("[0-9]")) {
46            return this.q3(cont + 1, car);
```

File - AutomataDecimal.java

```
47     }
48     return false;
49 }
50 }
51
52 private boolean q3(int cont, char[] car) {
53     if (cont == car.length) {
54         return true;
55     }
56     if (Character.toString(car[cont]).matches("[0-9]")) {
57         return this.q3(cont + 1, car);
58     }
59     return false;
60 }
61 }
62 }
```

File - AutomatalIdentificador.java

```
1 package itc.automatas2.lexico.automatas;
2
3 import itc.automatas2.lexico.Tokens;
4
5 /**
6  * Autómata que reconoce un identificador (variable).
7 */
8 public class AutomataIdentificador extends AutomataToken {
9
10    public AutomataIdentificador() {
11        super(Tokens.T_VAR);
12    }
13
14    public boolean ejecutar(String token) {
15        return q0(0, token.toCharArray());
16    }
17
18    private boolean q0(int cont, char[] car) {
19        if (cont == car.length) {
20            return false;
21        } else {
22            if (Character.toString(car[cont]).matches("[A-Za-z_]")) {
23                return this.q1(cont + 1, car);
24            } else {
25                return false;
26            }
27        }
28    }
29
30    private boolean q1(int cont, char[] car) {
31        if (cont == car.length) {
32            return true;
33        }
34        if (Character.toString(car[cont]).matches("[A-Za-z0-9_]")) {
35            return this.q1(cont + 1, car);
36        }
37        return false;
38    }
39}
40
41}
42}
```

File - Archivo.java

```
1 package itc.automatas2.estructuras;
2
3 import java.io.BufferedReader;
4 import java.io.File;
5 import java.io.FileNotFoundException;
6 import java.io.FileReader;
7 import java.io.IOException;
8
9 /**
10  * Clase para lectura de un archivo de texto. Se puede leer linea por linea o por completo.
11 */
12 public class Archivo {
13     private final String RUTA;
14     private final File file;
15     private BufferedReader reader;
16     private int line;
17
18     /**
19      * Crea un objeto de la clase.
20      *
21      * @param ruta La ruta del archivo.
22      * @throws FileNotFoundException Si el archivo no existe.
23      */
24     public Archivo(String ruta) throws FileNotFoundException {
25         this.RUTA = ruta;
26         this.file = new File(ruta);
27         //Si el archivo no existe tira excepción, que otra clase se encargue
28         if (!file.exists())
29             throw new FileNotFoundException();
30         this.reader = new BufferedReader(new FileReader(file));
31         this.line = 0;
32     }
33
34     /**
35      * Lee la siguiente linea del archivo.
36      *
37      * @return La siguiente linea, <code>null</code> si se llegó al final del archivo. Una vez que se
38      * llega al final del archivo, el mismo se cierra.
39      * @throws IOException Si ocurre un error en la lectura o el archivo ya se cerró.
40      */
41     public String leerLinea() throws IOException {
42         String tmp = reader.readLine();
43         line++;
44         if (tmp == null)
45             reader.close();
46         return tmp;
47     }
48 }
```

File - Archivo.java

```
47     }
48
49     /**
50      * Obtiene el número de línea actual.
51      *
52      * @return El número de línea actual.
53      */
54     public int getNoLinea() {
55         return line;
56     }
57 }
58
```

File - RegistroTS.java

```
1 package itc.automatas2.estructuras;
2
3 /**
4  * Registro de la tabla de símbolos, que contiene el nombre, el ID del token, y el tipo.
5 */
6 public class RegistroTS {
7     public final String NOMBRE;
8     public final int TOKEN_ID;
9     public final int TIPO;
10
11    /**
12     * Constructor de la clase
13     *
14     * @param NOMBRE
15     * @param TOKEN_ID
16     * @param TIPO
17     */
18    public RegistroTS(String NOMBRE, int TOKEN_ID, int TIPO) {
19        this.NOMBRE = NOMBRE;
20        this.TOKEN_ID = TOKEN_ID;
21        this.TIPO = TIPO;
22    }
23}
24
```

File - PilaErrores.java

```
1 package itc.automatas2.estructuras;
2
3 import java.util.ArrayDeque;
4
5 /**
6 *
7 */
8 public class PilaErrores {
9     private ArrayDeque<RegistroErr> pila = new ArrayDeque<>();
10
11    /**
12     * Inserta un nuevo error a la pila
13     *
14     * @param error Un objeto del tipo {alink RegistroErr RegistroErr}
15     */
16    public void meter(RegistroErr error) {
17        pila.push(error);
18    }
19
20    /**
21     * Remueve el último objeto de la pila
22     *
23     * @return Un objeto del tipo {alink RegistroErr RegistroErr}
24     */
25    public RegistroErr sacar() {
26        return pila.pop();
27    }
28
29    /**
30     * Obtiene el tamaño de la pila
31     *
32     * @return Un entero que representa el tamaño de la pila
33     */
34    public int size() {
35        return pila.size();
36    }
37 }
```

File - RegistroErr.java

```
1 package itc.automatas2.estructuras;
2
3 /**
4  * Registro de la pila de errores, que contiene el ID del error, el número de línea, y el lexema que lo causó.
5 */
6 public class RegistroErr {
7     public final int ERR_ID;
8     public final int LINEA_N;
9     public final String LEXEMA;
10
11    /**
12     * Constructor de la clase
13     * @param ERR_ID
14     * @param LINEA_N
15     * @param LEXEMA
16     */
17    public RegistroErr(int ERR_ID, int LINEA_N, String LEXEMA) {
18        this.ERR_ID = ERR_ID;
19        this.LINEA_N = LINEA_N;
20        this.LEXEMA = LEXEMA;
21    }
22}
23
```

File - TablaSimbolos.java

```
1 package itc.automatas2.estructuras;
2
3 import java.util.ArrayDeque;
4 import java.util.Hashtable;
5
6 /**
7  * Clase que representa una tabla de simbolos.
8  */
9 public class TablaSimbolos {
10     private Hashtable<String, RegistroTS> tabla;
11
12     /**
13      * Constructor de la clase
14      */
15     public TablaSimbolos() {
16         tabla = new Hashtable<>();
17     }
18
19     /**
20      * Metodo para meter un nuevo registro al HashTable
21      *
22      * @param llave La llave del nuevo registro.
23      * @param registro Un objeto del tipo {alink RegistroTS RegistroTS}.
24      */
25     public void meter(String llave, RegistroTS registro) {
26         tabla.put(llave, registro);
27     }
28
29     /**
30      * Metodo para verificar la existencia de un registro en el HashTable
31      *
32      * @param llave La llave del registro.
33      * @return <code>true</code> si el registro existe, <code>false</code> si no.
34      */
35     public boolean contiene(String llave) {
36         return tabla.containsKey(llave);
37     }
38
39     /**
40      * Metodo para obtener los valores de los registro en el HashTable
41      *
42      * @return Un objeto del tipo {alink RegistroTS RegistroTS}.
43      */
44     public ArrayDeque<RegistroTS> registros() {
45         return new ArrayDeque<>(tabla.values());
46     }
}
```

File - TablaSimbolos.java

```
47
48     /**
49      * Obtiene el tamaño de la tabla
50      * @return Un entero que representa el tamaño de la tabla
51      */
52     public int size() {
53         return tabla.size();
54     }
55 }
56 }
```