EXREGAN

MANUAL TECNICO

Proyecto 1 del curso:

"Organización de lenguajes y compiladores 1"

Sección:

"C"

Catedrático:

Kevin Lajpop

Auxiliar:

Maynor Octavio Piló Tuy

Nombre del estudiante:

Josue Alejandro Perez Benito

Carne del estudiante:

201712602

Introducción

En el presente manual se indicará los pasos a seguir para la comprensión de las funcionalidades del programa EXREGAN; dichas funcionalidades constan del análisis de sistema de conjuntos-expresiones regulares-cadenas de prueba.

Dicho programa tiene como objetivo analizar e interpretar gráficamente la información a través del reconocimiento de patrones de tokens comenzando por el análisis de cada lexema de la cadena de entrada seguido de la revisión de patrones de tokens en el analizador sintáctico.

El programa se creo con el fin de facilitar la comprensión de las primeras fases del compilador.

Objetivos

Objetivo general

Desarrollar un programa capaz de implementar un analizador léxico y sintáctico para el análisis de cadenas de caracteres.

Objetivos específicos

- Implementar el método del árbol para el análisis de soluciones con el fin de poder desarrollar un autómata finito determinista.
- Implementar el análisis del método de Thompson para desarrollar autómatas finitos no deterministas.
- Trazar soluciones por medio del autómata finito determinista.

Requerimientos mínimos del sistema

Windows

Windows 10 (8u51 y superiores)

Windows 8.x (escritorio)

Windows 7 SP1

Windows Vista SP2

Windows Server 2008 R2 SP1 (64 bits)

Windows Server 2012 y 2012 R2 (64 bits)

RAM: 128 MB

Espacio en disco: 124 MB para JRE; 2 MB para Java Update

Procesador: Mínimo Pentium 2 a 266 MHz

Exploradores: Internet Explorer 9 y superior, Firefox

Mac OS X

Mac con Intel que ejecuta Mac OS X 10.8.3+, 10.9+

Privilegios de administrador para la instalación

Explorador de 64 bits

Se requiere un explorador de 64 bits (Safari, por ejemplo) para ejecutar Oracle Java en Mac.

Linux

Oracle Linux 5.5+1

Oracle Linux 6.x (32 bits), 6.x (64 bits)2

Oracle Linux 7.x (64 bits)2 (8u20 y superiores)

Red Hat Enterprise Linux 5.5+1 6.x (32 bits), 6.x (64 bits)2

Red Hat Enterprise Linux 7.x (64 bits)2 (8u20 y superiores)

Suse Linux Enterprise Server 10 SP2+, 11.x

Suse Linux Enterprise Server 12.x (64 bits)2 (8u31 y superiores)

Ubuntu Linux 12.04 LTS, 13.x

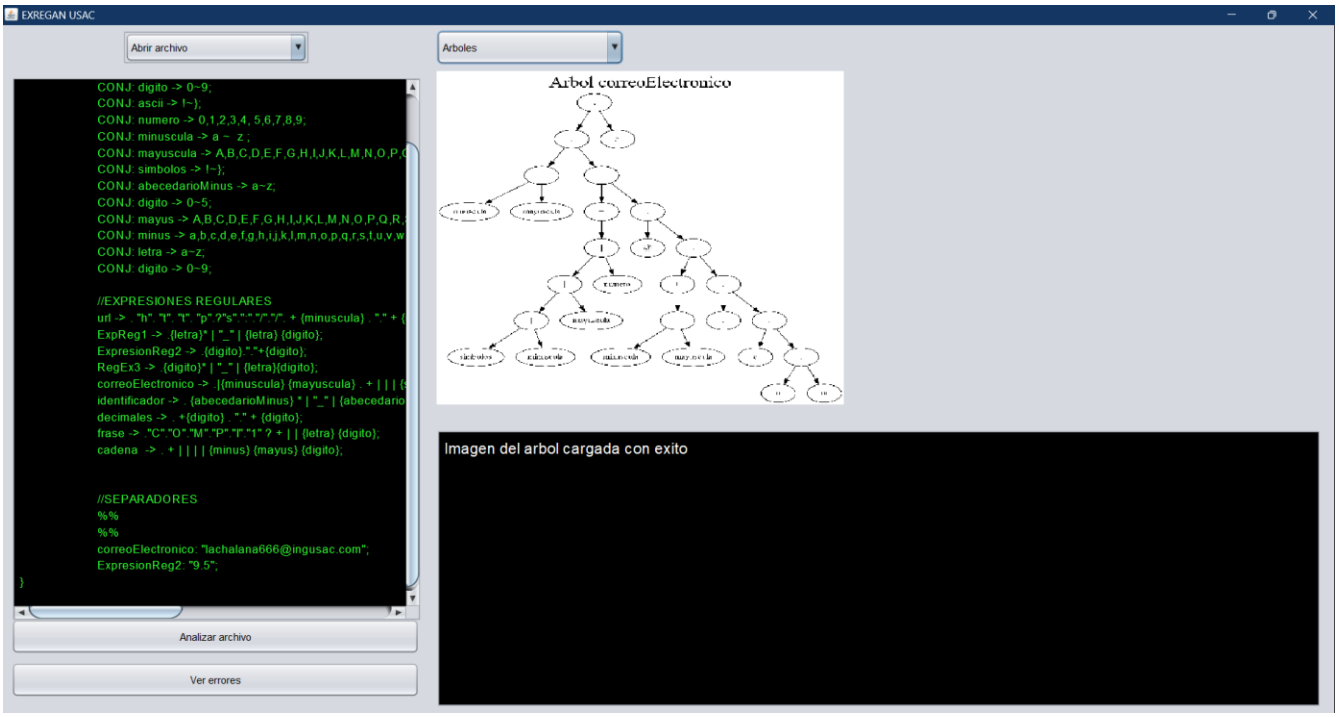
Ubuntu Linux 14.x (8u25 y superiores)

Ubuntu Linux 15.04 (8u45 y superiores)

Ubuntu Linux 15.10 (8u65 y superiores)

Exploradores: Firefox

Interfaz grafica



La presente interfaz consta con 2 **combo-box**:

- **La primera** para "*cargar, crear, actualizar o guardar*" el archivo;
- **La segunda** para "*seleccionar que reporte visualizar*".

Consta de 2 botones:

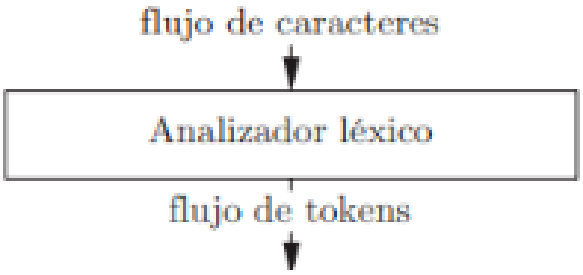
1. **Analizar archivo**: buscara los conjuntos, expresiones regulares y cadenas de entrada presentes en el archivo, además de los errores léxicos y sintácticos.
2. **Ver errores**: mostrara los errores léxicos y sintácticos encontrados en el archivo a evaluar.

Consta de 2 paneles:

1. Visualizador de archivos
2. Terminal

Analizador léxico

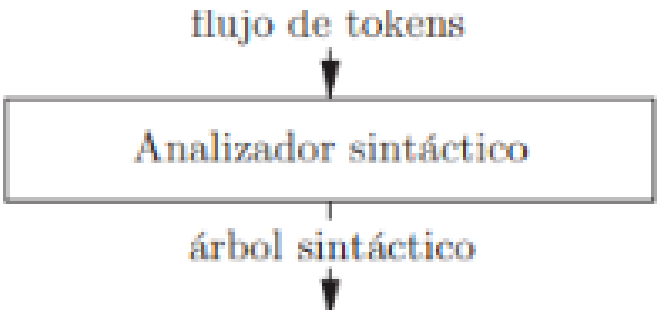
Dicho analizador se encargará de verificar que todas las entradas sean con caracteres validos para que el programa pueda entender sin dificultad las operaciones solicitadas.



Como se muestra en la imagen el analizador léxico retornara un flujo de tokens para que el analizador sintáctico pueda entrar en acción.

Analizador sintáctico

Dicho analizador se encargará de analizar que el flujo de tokens correspondiente cumpla con todas las leyes impuestas por la gramática definida.

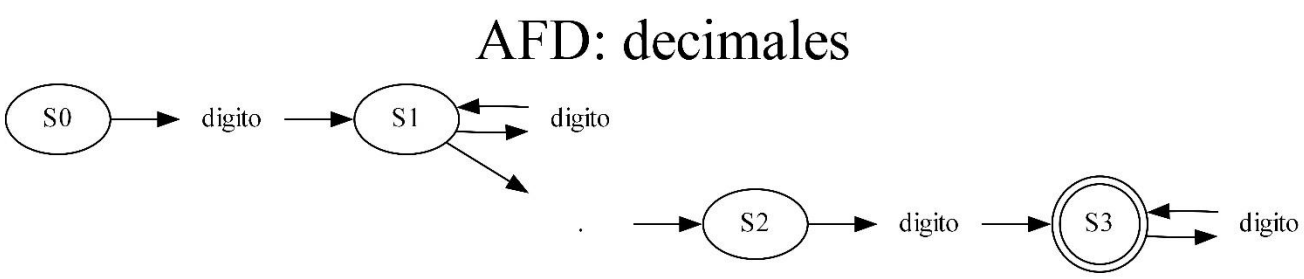


Como se muestra en la imagen retornará un árbol sintáctico que permitirá el cumplimiento de su siguiente fase.

Reportes

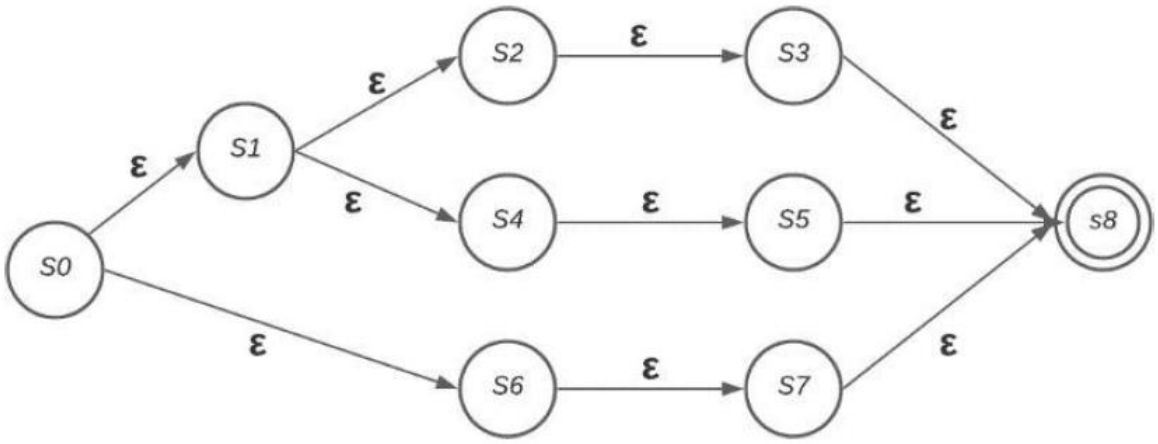
AFD

Los autómatas finitos deterministas no sufren de ambigüedad ni transiciones con épsilon.



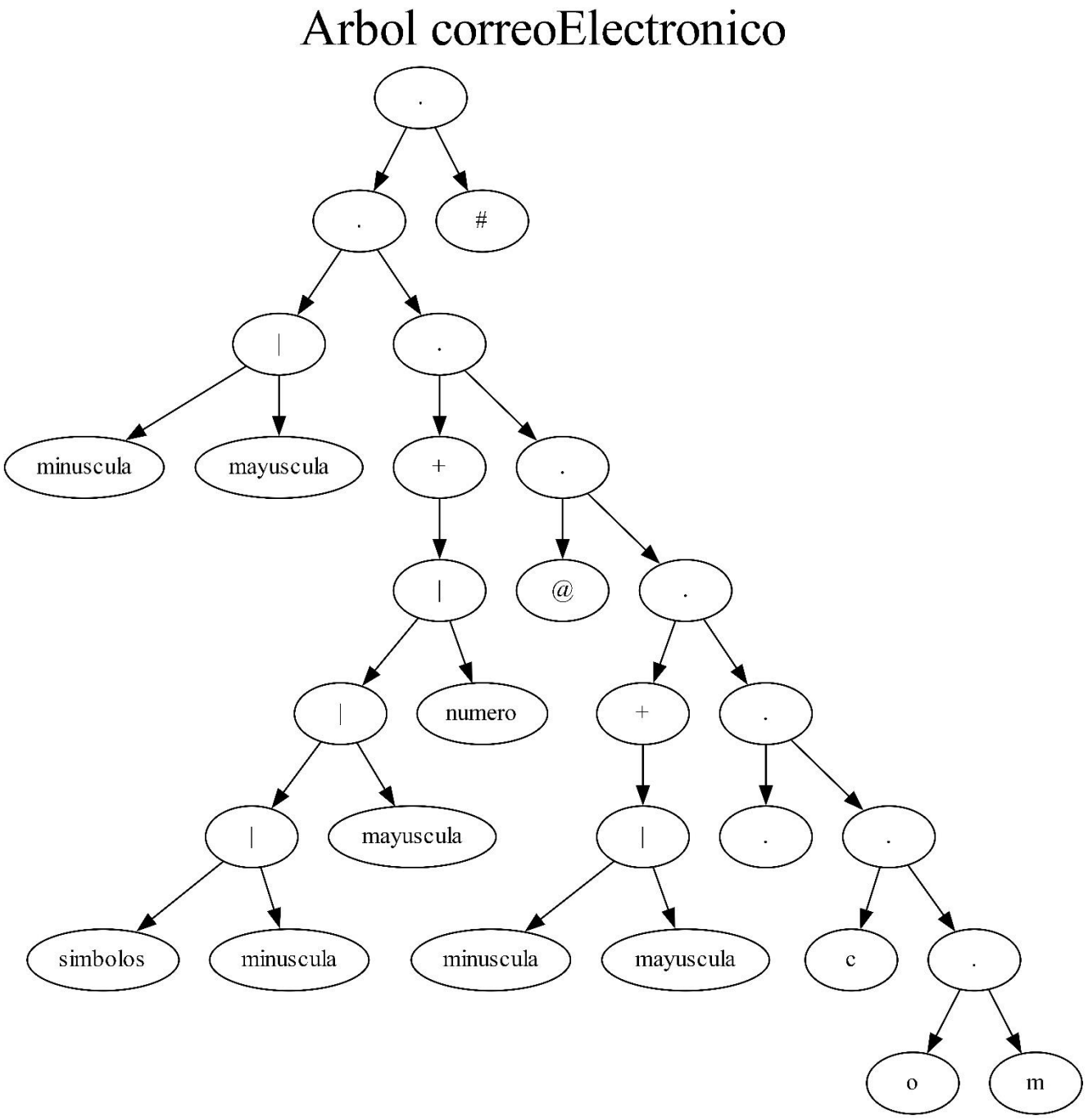
AFND

A diferencia de los AFD los AFND si presentan ambigüedad y transiciones con épsilon para el desarrollo de soluciones.



Método del árbol

El método del árbol es esencial para el desarrollo de la fase de análisis semántico, este provee una vista de las rutas para llegar a la solución de una cadena de entrada.



Errores

El presente programa mostrara los errores encontrados en 2 archivos HTML, uno para los errores léxicos y otro para los errores sintácticos.

LISTADO DE ERRORES LEXICOS ENCONTRADOS EN LA COMPILACIÓN

- Error léxico: / Linea: 18 Columna: 40
- Error léxico: / Linea: 18 Columna: 44

LISTADO DE ERRORES SINTACTICOS ENCONTRADOS EN LA COMPILACIÓN

- **Error sintáctico recuperable: . Linea: 18 Columna: 42**
- No se logro leer la expresion: frase
- No se logro leer la expresion: cadena

Salidas

EXREGAN al finalizar el análisis del archivo mostrara el resultado de las cadenas por medio de un archivo con extensión Json.

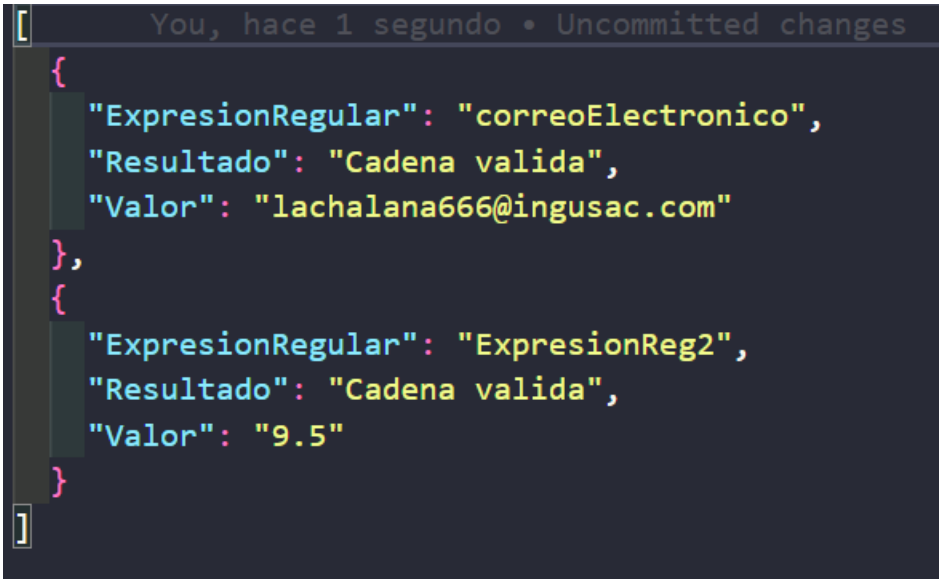


Tabla de siguientes

A partir del árbol sintáctico generado se obtiene la tabla de siguientes.

Hoja		Siguientes
a	1	2,3,4
a	2	2,3,4
b	3	2,3,4
b	4	5
#	5	--

Tabla de transiciones

A partir del árbol sintáctico y la tabla de siguientes logramos formar la tabla de transiciones.

Estado	Terminales	
	a	b
S0 {1}	S1	--
S1 {2,3,4}	S1	S2
S2 {2,3,4,5}	S1	S2

Sintaxis para el correcto funcionamiento

Definición de conjuntos

Para la definición de conjuntos deberás seguir la siguiente estructura:

CONJ: IDENTIFICADOR -> 1,2,3,4,5,6,7;

CONJ: IDENTIFICADOR -> A~Z;

CONJ: la palabra reservada CONJ será la pauta para indicar al programa que se está por declarar un conjunto, siempre debe ir seguido por ":" los dos puntos.

IDENTIFICADOR: después debemos ingresar un identificador único para llamar a nuestro conjunto.

->: EL presente símbolo representa la asignación de los elementos que tendrá nuestro conjunto.

1,2,3,4,5,6,7 ó **A~Z**: para definir los elementos que conformaran nuestro conjunto tenemos 2 opciones:

1. Definir nuestros elementos por medio de comas.
2. Definir un grupo especifico de elementos ayudándonos con el símbolo "~".

Siempre debemos terminar con el punto y coma "**;**".

Definición de expresiones regulares

Para establecer nuestras expresiones regulares usaremos la siguiente simbología:

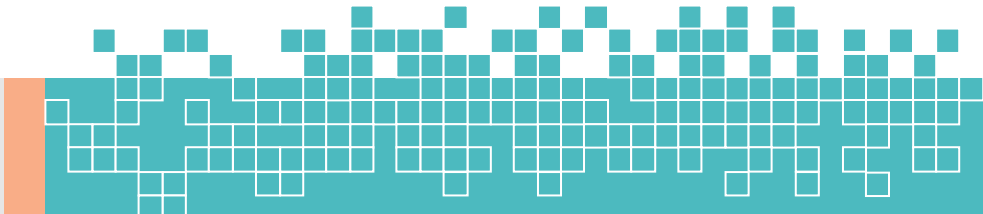
- . El punto representara la concatenación de elementos
- | el presente símbolo representara el OR o un elemento u otro.
- + el presente símbolo representara que el elemento puede venir 1 o más veces.
- * el presente símbolo representara que el elemento puede venir 0 o más veces.
- ? el presente símbolo representara que el elemento puede venir o no.
- “ ” las comillas adjuntaran cualquier elemento a nuestra expresión, dentro de ellas debe ir el elemento que se desea incluir.
- { } las llaves nos servirán para llamar a nuestros conjuntos, el identificador del conjunto debe ir dentro de ellas.

IDENTIFICADOR -> EXPRESION;

IDENTIFICADOR: Debemos ingresar un identificador único para llamar a nuestra expresión.

->: EL presente símbolo representa la asignación de los elementos que tendrá nuestro conjunto.

EXPRESION: la expresión deberá finalizarse con el punto y coma para ser valida “;”.



Definición de cadenas de entrada

IDENTIFICADOR: "CADENA";

IDENTIFICADOR: Se deberá llamar al identificador de la expresión regular a valuar, después deberá añadirse los dos puntos "**:**".

CADENA: Se deberá escribir la cadena a valuar dentro de comillas dobles "" y terminar con el punto y coma "**;**".

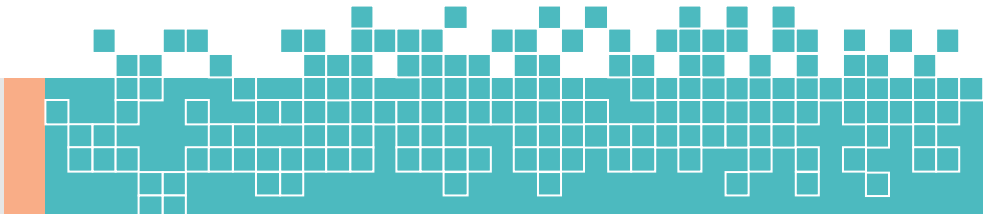
Analizador léxico propuesto

```
package Analizador;
import java_cup.runtime.Symbol;
import App.Brain;
%%

%{
    Brain br = new Brain();
%}

%class scanner
%cup
%public
%line
%column
%unicode
%ignorecase


/* Simbolos */
PUNTO = "."
DISY = "|"
MULT = "*"
PLUS = "+"
ITR = "?"
TILDE = "~"
COMA = ","
COM_SIMPLE = "\'"
COM_DOBLE = "\""
DOS_PUNTOS = ":"
PUNTO_COMA = ";"
LLAV_ABIERTA = "{"
LLAV_CERRADA = "}"

/* Palabras reservadas */
CONJUNTO = "CONJ"

/* Expresiones */
ENTERO = [0-9]+
/* DECIMAL = [0-9]+("."[ |0-9]+)? */
LETRA = [A-Za-zÑñ_ÁÉÍÓÚáéíóúÜü]
SPACE   = [ ,\t,\r]+
ENTER   = [\ \n]
```

```
LINEA = "\n"


/* ASCII */
ASCII33 = "!"
/* ASCII34 = COM_DOBLE */
ASCII35 = "#"
ASCII36 = "$"
ASCII37 = "%"
ASCII38 = "&"
/* ASCII39 = COM_SIMPLE */
ASCII40 = "("
ASCII41 = ")"
/* ASCII42 = MULT
ASCII43 = PLUS
ASCII44 = COMA */
ASCII45 = "-"
ASCII46 = PUNTO
ASCII47 = "/"
/* ASCII58 = DOS_PUNTOS
ASCII59 = PUNTO_COMA */
ASCII60 = "<"
ASCII61 = "="
ASCII62 = ">"
/* ASCII63 = ITR */
ASCII64 = "@"
ASCII91 = "["
ASCII92 = "\\"
ASCII93 = "]"
ASCII94 = "^"
ASCII95 = "_"
ASCII96 = "`"
/* ASCII123 = LLAV_ABIERTA
ASCII124 = DISY
ASCII125 = LLAV_CERRADA */
SIMBOLOS =
({COM_DOBLE}|{COM_SIMPLE}|{MULT}|{PLUS}|{COMA}|{DOS_PUNTOS}|{PUNTO_COMA}|{ITR}|{DISY}|
{LLAV_ABIERTA}|{LLAV_CERRADA})
ASCII =
({ASCII33}|{ASCII35}|{ASCII36}|{ASCII37}|{ASCII38}|{ASCII40}|{ASCII41}|{ASCII45}|{ASCI
I46}{ASCII47}|{ASCII60}|{ASCII61}|{ASCII62}|{ASCII64}|{ASCII91}|{ASCII92}|{ASCII93}|{A
SCII94}|{ASCII95}|{ASCII96})

/* Comentarios */
```

```
COMENTARIO_SIMPLE = "//"+ ({SPACE}|{LETRA}|{ENTERO}|{ASCII}|{SIMBOLOS})
({SPACE}|{LETRA}|{ENTERO}|{ASCII}|{SIMBOLOS})*
COMENTARIO_EXTENSO = "<!"({ENTER}|{SPACE}|{LETRA}|{ENTERO}|{ASCII}|{SIMBOLOS})
({ENTER}|{SPACE}|{LETRA}|{ENTERO}|{ASCII}|{SIMBOLOS})* "!>"

/* Operador de asignación */
OPERADOR = "->"

/* Separador */
SEPARADOR = "%%"

/* Identificador */
IDENTIFICADOR = ({LETRA}|{ENTERO}|{ASCII}) ({LETRA}|{ENTERO}|{ASCII})*


%%

<YYINITIAL> {CONJUNTO} { return new Symbol(sym.CONJUNTO, yyline, yycolumn, yytext());}

<YYINITIAL> {PUNTO} { return new Symbol(sym.PUNTO, yyline, yycolumn, yytext());}

<YYINITIAL> {DISY} { return new Symbol(sym.DISY, yyline, yycolumn, yytext());}

<YYINITIAL> {MULT} { return new Symbol(sym.MULT, yyline, yycolumn, yytext());}

<YYINITIAL> {PLUS} { return new Symbol(sym.PLUS, yyline, yycolumn, yytext());}

<YYINITIAL> {ITR} { return new Symbol(sym.ITR, yyline, yycolumn, yytext());}

<YYINITIAL> {TILDE} { return new Symbol(sym.TILDE, yyline, yycolumn, yytext());}

<YYINITIAL> {COMA} { return new Symbol(sym.COMA, yyline, yycolumn, yytext());}

<YYINITIAL> {COM_SIMPLE} { return new Symbol(sym.COM_SIMPLE, yyline, yycolumn,
yytext());}

<YYINITIAL> {COM_DOBLE} { return new Symbol(sym.COM_DOBLE, yyline, yycolumn,
yytext());}

<YYINITIAL> {DOS_PUNTOS} { return new Symbol(sym.DOS_PUNTOS, yyline, yycolumn,
yytext());}

<YYINITIAL> {PUNTO_COMA} { return new Symbol(sym.PUNTO_COMA, yyline, yycolumn,
yytext());}
```

```
<YYINITIAL> {LLAV_ABIERTA} { return new Symbol(sym.LLAV_ABIERTA, yyline, yycolumn,
yytext());}

<YYINITIAL> {LLAV_CERRADA} { return new Symbol(sym.LLAV_CERRADA, yyline, yycolumn,
yytext());}

<YYINITIAL> {ENTERO} { return new Symbol(sym.ENTERO, yyline, yycolumn, yytext());}

/* <YYINITIAL> {DECIMAL} { return new Symbol(sym.DECIMAL, yyline, yycolumn,
yytext());}
 */
<YYINITIAL> {LETRA} { return new Symbol(sym.LETRA, yyline, yycolumn, yytext());}

<YYINITIAL> {OPERADOR} { return new Symbol(sym.OPERADOR, yyline, yycolumn, yytext());}

<YYINITIAL> {ASCII} { return new Symbol(sym.ASCII, yyline, yycolumn, yytext());}

<YYINITIAL> {SEPARADOR} { return new Symbol(sym.SEPARADOR, yyline, yycolumn, yytext())
;}

<YYINITIAL> {IDENTIFICADOR} { return new Symbol(sym.IDENTIFICADOR, yyline, yycolumn,
yytext());}

<YYINITIAL> {SPACE}     { /* Ignorar */ }

<YYINITIAL> {ENTER}     { /* Ignorar */}

<YYINITIAL> {LINEA}     {/* Ignorar */}

<YYINITIAL> {COMENTARIO_SIMPLE} {/* Ignorar */}

<YYINITIAL> {COMENTARIO_EXTENSO} {/* Ignorar */}

<YYINITIAL> . {
        String errLex = "Error léxico : '"+yytext()+"' en la línea: "+(yyline+1)+" y
columna: "+(yycolumn+1);
        //System.out.println(errLex);
        br.ErroresL("<h4>Error léxico: "+yytext()+" Linea: "+(yyline+1)+" Columna:
"+(yycolumn+1)+"</h4>");

}
```



Analizador sintáctico propuesto

```
package Analizador;
import java_cup.runtime.*;
import Analizador.Conjunto;
import App.Brain;


//PARSER
parser code
{://codigo visible
    Brain br = new Brain();
    Conjunto conjs = new Conjunto();

    public void syntax_error(Symbol s){
        br.ErroresS("<h2>Error sintáctico recuperable: "+s.value+" Linea:
"+(s.left+1)+" Columna: "+(s.right+1)+"</h4>");
        //System.out.println("Error R de sintaxis: "+ s.value +" Linea "+(s.left+1)+"
columna "+(s.right+1) );
    }

    public void unrecovered_syntax_error(Symbol s) throws java.lang.Exception{
        br.ErroresS("<h2>Error sintáctico no recuperable: "+s.value+" Linea:
"+(s.left+1)+" Columna: "+(s.right+1)+"</h4>");
        //System.out.println("Error NR de sintaxis: "+ s.value +" Linea "+(s.left+1)+"
columna "+(s.right+1) );
    }

    public void addConj(String conjunto,String elemento){
        br.addConj(conjunto,elemento);
    }

    public void analizarER(String er, String erName){
        br.analizarER(er,erName);
    }

    public void analizarCadena(String name, String cadena){
        br.AnalizarEntrada(name,cadena);
    }
:}

terminal String CONJUNTO;
terminal String PUNTO;
terminal String DISY;
```

```
terminal String MULT;
terminal String PLUS;
terminal String ITR;
terminal String TILDE;
terminal String COMA;
terminal String COM_SIMPLE;
terminal String COM_DOBLE;
terminal String DOS_PUNTOS;
terminal String PUNTO_COMA;
terminal String LLAV_ABIERTA;
terminal String LLAV_CERRADA;
terminal String ENTERO;
terminal String ASCII;
/* terminal String DECIMAL; */
terminal String LETRA;
terminal String IDENTIFICADOR;
terminal String OPERADOR;
terminal String SEPARADOR;

non terminal ini;
non terminal instrucciones;
non terminal instruccion;
non terminal String conjunto;
non terminal String identificador;
non terminal String ascii;
non terminal String Symbol;
non terminal String expresion;
non terminal String exp;
non terminal String conjCall;
non terminal String separadores;
non terminal String entradas;
non terminal String lectura;

start with ini;

ini::= LLAV_ABIERTA instrucciones LLAV_CERRADA
;

instrucciones::= instruccion instrucciones
 | instruccion
;

instruccion ::=
```

```
    CONJUNTO DOS_PUNTOS identificador:a OPERADOR conjunto:b PUNTO_COMA {:
addConj(a,b); :}
    | expresion
    | separadores
    | error PUNTO_COMA
;

identificador ::=
    IDENTIFICADOR:a     {: RESULT=a; :}
    | LETRA:a           {: RESULT=a; :}
    | ENTERO:a          {: RESULT=a; :}
;

ascii ::=
    ASCII:a {: RESULT=a; :}
;

conjunto ::=
    ENTERO:a TILDE ENTERO:b                 {: RESULT=a+"~"+b; :}
    | LETRA:a TILDE LETRA:b                 {: RESULT=a+"~"+b; :}
    | LETRA:a COMA LETRA:b                  {: RESULT=a+","+b; :}
    | LETRA:a COMA LETRA:b conjunto:c       {: RESULT=a+","+b+c; :}
    | ENTERO:a COMA ENTERO:b                {: RESULT=a+","+b; :}
    | ENTERO:a COMA ENTERO:b conjunto:c      {: RESULT=a+","+b+c; :}
    | COMA ENTERO:a conjunto:b              {: RESULT=","+a+b; :}
    | COMA LETRA:a  conjunto:b              {: RESULT=","+a+b; :}
    | ascii:a TILDE ascii:b                 {: RESULT=a+"~"+b; :}
    | ascii:a TILDE Symbol:b                {: RESULT=a+"~"+b; :}
    | Symbol:a TILDE ascii:b                {: RESULT=a+"~"+b; :}
    | ascii:a COMA ascii:b                  {: RESULT=a+","+b; :}
    | ascii:a COMA ascii:b conjunto:c       {: RESULT=a+","+b+c; :}
    | Symbol:a COMA ascii:b                 {: RESULT=a+","+b; :}
    | Symbol:a COMA ascii:b conjunto:c       {: RESULT=a+","+b+c; :}
    | ascii:a COMA Symbol:b                 {: RESULT=a+","+b; :}
    | ascii:a COMA Symbol:b conjunto:c       {: RESULT=a+","+b+c; :}
    | ascii:a TILDE COM_DOBLE:b             {: RESULT=a+"~"+b; :}
    | COM_DOBLE:a TILDE ascii:b             {: RESULT=a+"~"+b; :}
    | ascii:a TILDE COM_SIMPLE:b            {: RESULT=a+"~"+b; :}
    | COM_SIMPLE:a TILDE ascii:b            {: RESULT=a+"~"+b; :}
    | ascii:a TILDE COMA:b                  {: RESULT=a+"~"+b; :}
    | COMA:a TILDE ascii:b                  {: RESULT=a+"~"+b; :}
    | ascii:a TILDE DOS_PUNTOS:b            {: RESULT=a+"~"+b; :}
    | DOS_PUNTOS:a TILDE ascii:b            {: RESULT=a+"~"+b; :}
    | ascii:a TILDE LLAV_ABIERTA:b          {: RESULT=a+"~"+b; :}
```

```
    | LLAV_ABIERTA:a TILDE ascii:b          {: RESULT=a+"~"+b; :}
    | ascii:a TILDE LLAV_CERRADA:b          {: RESULT=a+"~"+b; :}
    | LLAV_CERRADA:a TILDE ascii:b          {: RESULT=a+"~"+b; :}
    | COMA COM_DOBLE:a conjunto:b           {: RESULT=","+a+b; :}
    | COMA COM_SIMPLE:a conjunto:b          {: RESULT=","+a+b; :}
    | COMA DOS_PUNTOS:a conjunto:b          {: RESULT=","+a+b; :}
    | COMA PUNTO_COMA:a conjunto:b          {: RESULT=","+a+b; :}
    | COMA LLAV_ABIERTA:a conjunto:b        {: RESULT=","+a+b; :}
    | COMA LLAV_CERRADA:a conjunto:b        {: RESULT=","+a+b; :}
    | COMA ascii:a                          {: RESULT=","+a; :}
    | COMA Symbol:a                         {: RESULT=","+a; :}
    | COMA LETRA:a                          {: RESULT=","+a; :}
    | COMA ENTERO:a                         {: RESULT=","+a; :}
    | COMA COM_DOBLE:a                         {: RESULT=","+a; :}
    | COMA COM_SIMPLE:a                         {: RESULT=","+a; :}
    | COMA DOS_PUNTOS:a                         {: RESULT=","+a; :}
    | COMA PUNTO_COMA:a                         {: RESULT=","+a; :}
    | COMA LLAV_ABIERTA:a                               {: RESULT=","+a; :}
    | COMA LLAV_CERRADA:a                               {: RESULT=","+a; :}
    | ascii:a COMA COM_SIMPLE:b                         {: RESULT=a + ","+b; :}
    | ascii:a COMA COM_DOBLE:b                          {: RESULT=a + ","+b; :}
    | ascii:a COMA DOS_PUNTOS:b                         {: RESULT=a + ","+b; :}
    | ascii:a COMA PUNTO_COMA:b                         {: RESULT=a + ","+b; :}
    | ascii:a COMA LLAV_ABIERTA:b                       {: RESULT=a + ","+b; :}
    | ascii:a COMA LLAV_CERRADA:b                       {: RESULT=a + ","+b; :}
;

expresion ::=
    identificador:a OPERADOR exp:b PUNTO_COMA {: br.analizarER(b,a); :}
;


exp ::=
    Symbol:a conjCall:b                {: RESULT= a+b; :}
    | Symbol:a conjCall:b exp:c        {: RESULT=a+b+c; :}
    | Symbol:a exp:b                   {: RESULT=a+b; :}
;

conjCall ::=
    LLAV_ABIERTA identificador:a LLAV_CERRADA                  {: RESULT="{"+a+"}";
:}
    | LLAV_ABIERTA identificador:a LLAV_CERRADA conjCall:b      {: RESULT="{"+a+"}"+b;
:}
```

```
    | COM_DOBLE identificador:a COM_DOBLE                    {: RESULT="\""+a+"\"";
:}
    | COM_DOBLE identificador:a COM_DOBLE conjCall:b          {:
RESULT="\""+a+"\""+b; :}
    | COM_SIMPLE identificador:a COM_SIMPLE                    {: RESULT="\'"+a+"\'";
:}
    | COM_SIMPLE identificador:a COM_SIMPLE conjCall:b         {:
RESULT="\'"+a+"\'"+b; :}
    | COM_DOBLE ascii:a COM_DOBLE                              {: RESULT="\""+a+"\"";
:}
    | COM_DOBLE ascii:a COM_DOBLE conjCall:b                   {:
RESULT="\""+a+"\""+b; :}
    | COM_SIMPLE ascii:a COM_SIMPLE                            {: RESULT="\'"+a+"\'";
:}
    | COM_SIMPLE ascii:a COM_SIMPLE conjCall:b                 {:
RESULT="\'"+a+"\'"+b; :}
    | COM_DOBLE Symbol:a COM_DOBLE                             {: RESULT="\""+a+"\"";
:}
    | COM_DOBLE Symbol:a COM_DOBLE conjCall:b                  {:
RESULT="\""+a+"\""+b; :}
    | COM_SIMPLE Symbol:a COM_SIMPLE                           {: RESULT="\'"+a+"\'";
:}
    | COM_SIMPLE Symbol:a COM_SIMPLE conjCall:b                {:
RESULT="\'"+a+"\'"+b; :}
    | COM_DOBLE  COM_DOBLE:a COM_DOBLE                         {: RESULT="\""+a+"\"";
:}
    | COM_DOBLE  COM_DOBLE:a COM_DOBLE conjCall:b              {:
RESULT="\""+a+"\""+b; :}
    | COM_SIMPLE COM_DOBLE:a COM_SIMPLE                        {: RESULT="\'"+a+"\'";
:}
    | COM_SIMPLE COM_DOBLE:a COM_SIMPLE conjCall:b             {:
RESULT="\'"+a+"\'"+b; :}
    | COM_DOBLE  COM_SIMPLE:a COM_DOBLE                        {: RESULT="\""+a+"\"";
:}
    | COM_DOBLE  COM_SIMPLE:a COM_DOBLE conjCall:b             {:
RESULT="\""+a+"\""+b; :}
    | COM_SIMPLE COM_SIMPLE:a COM_SIMPLE                       {: RESULT="\'"+a+"\'";
:}
    | COM_SIMPLE COM_SIMPLE:a COM_SIMPLE conjCall:b            {:
RESULT="\'"+a+"\'"+b; :}
    | COM_DOBLE  COMA:a COM_DOBLE                              {: RESULT="\""+a+"\"";
:}
    | COM_DOBLE  COMA:a COM_DOBLE conjCall:b                   {:
RESULT="\""+a+"\""+b; :}
```

```
    | COM_SIMPLE COMA:a COM_SIMPLE                                {: RESULT="\'"+a+"\'";
:}
    | COM_SIMPLE COMA:a COM_SIMPLE conjCall:b                     {:
RESULT="\'"+a+"\'"+b; :}
    | COM_DOBLE  DOS_PUNTOS:a COM_DOBLE                          {: RESULT="\""+a+"\"";
:}
    | COM_DOBLE  DOS_PUNTOS:a COM_DOBLE conjCall:b               {:
RESULT="\""+a+"\""+b; :}
    | COM_SIMPLE DOS_PUNTOS:a COM_SIMPLE                         {: RESULT="\'"+a+"\'";
:}
    | COM_SIMPLE DOS_PUNTOS:a COM_SIMPLE conjCall:b              {:
RESULT="\'"+a+"\'"+b; :}
    | COM_DOBLE  PUNTO_COMA:a COM_DOBLE                          {: RESULT="\""+a+"\"";
:}
    | COM_DOBLE  PUNTO_COMA:a COM_DOBLE conjCall:b               {:
RESULT="\""+a+"\""+b; :}
    | COM_SIMPLE PUNTO_COMA:a COM_SIMPLE                         {: RESULT="\'"+a+"\'";
:}
    | COM_SIMPLE PUNTO_COMA:a COM_SIMPLE conjCall:b              {:
RESULT="\'"+a+"\'"+b; :}
    | COM_DOBLE  LLAV_ABIERTA:a COM_DOBLE                        {: RESULT="\""+a+"\"";
:}
    | COM_DOBLE  LLAV_ABIERTA:a COM_DOBLE conjCall:b             {:
RESULT="\""+a+"\""+b; :}
    | COM_SIMPLE LLAV_ABIERTA:a COM_SIMPLE                       {: RESULT="\'"+a+"\'";
:}
    | COM_SIMPLE LLAV_ABIERTA:a COM_SIMPLE conjCall:b            {:
RESULT="\'"+a+"\'"+b; :}
    | COM_DOBLE  LLAV_CERRADA:a COM_DOBLE                        {: RESULT="\""+a+"\"";
:}
    | COM_DOBLE  LLAV_CERRADA:a COM_DOBLE conjCall:b             {:
RESULT="\""+a+"\""+b; :}
    | COM_SIMPLE LLAV_CERRADA:a COM_SIMPLE                       {: RESULT="\'"+a+"\'";
:}
    | COM_SIMPLE LLAV_CERRADA:a COM_SIMPLE conjCall:b            {:
RESULT="\'"+a+"\'"+b; :}

;

Symbol ::=
    PUNTO:a                                          {: RESULT=a; :}
    | DISY:a                                         {: RESULT=a; :}
    | MULT:a                                         {: RESULT=a; :}
    | PLUS:a                                         {: RESULT=a; :}
```

```
    | ITR:a                                          {: RESULT=a; :}
;

lectura ::='
    identificador:a {: RESULT=a; :}
    | Symbol:a {: RESULT=a; :}
    | ascii:a {: RESULT=a; :}
    | identificador:a lectura:b {: RESULT=a+b; :}
    | Symbol:a lectura:b {: RESULT=a+b; :}
    | ascii:a lectura:b {: RESULT=a+b; :}
;

entradas ::=
    identificador:a DOS_PUNTOS COM_DOBLE lectura:b COM_DOBLE PUNTO_COMA {:
analizarCadena(a,b); :}
;

separadores ::=
    SEPARADOR
    | entradas
;
```