Implementación de un procesador de lenguajes para la generación de Autómatas de Moore

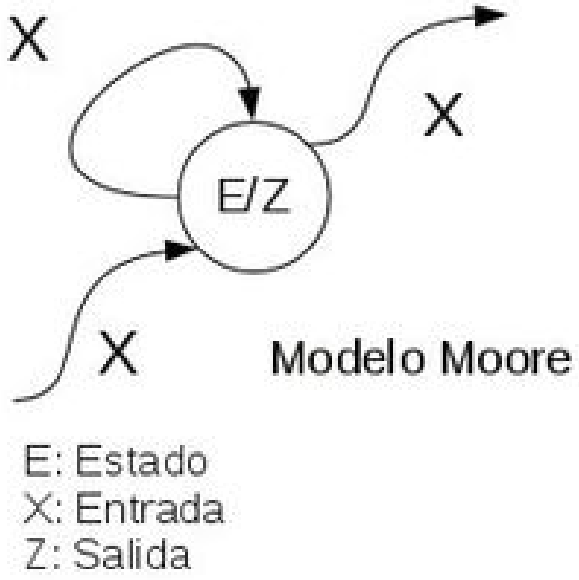
Julián García Sánchez Iván Illán Barraya
Alejandro Medina Jiménez Javier Monescillo Buitrón

20 de enero de 2019

Índice

1. ¿Qué es una máquina de Moore?

En Teoría de la computación un **autómata** o **máquina de Moore** es un autómata de estados finitos muy particular, la salida en un momento dado sólo depende de su estado en ese instante, mientras que la transición al siguiente estado dependerá del estado en el que se encuentre y de la entrada introducida.



Ejemplo de máquina de Moore simple.

Una máquina de Moore *Mmor* se define como una 6-tupla:

$$Mmor = (S, S_0, \Sigma, \Lambda, T, G)$$

donde definimos los siguientes elementos:

- S: es un conjunto finito de estados.
- S_0: es el estado inicial, y además es un elemento de S.
- Σ: un conjunto finito llamado alfabeto de entrada.
- Λ: un conjunto finito llamado alfabeto de salida.
- T : una función de transición $T : S \times \Sigma \rightarrow S$ que mapea un estado y una entrada al siguiente estado.
- G : una función de salida $G : S \rightarrow \Lambda$ que mapea cada estado al alfabeto de salida.

1.1. Descripción del problema

El problema que se presenta es la construcción de un procesador de lenguajes cuya entrada consista en un lenguaje de dominio específio en el cuál se declaran una o varias máquinas de Moore. Este lenguaje se ha definido como **Moor**. La salida del procesador de lenguajes consistirá en uno o varios ficheros generados en un lenguaje de alto nivel. Nosotros hemos elegido **Java**.

2. Solución propuesta

Para diseñar este procesador de lenguajes, utilizaremos los conocimientos de la materia *Procesadores de Lenguajes*durante las distintas etapas del proceso. El proceso consta de 4 etapas:

- Diseño del procesador de lenguajes
- Analizador Léxico
- Analizador Sintáctico
- Analizador Semántico
- Generación de archivos en lenguaje objeto

La primera etapa está más relacionada con lo relacionado al diseño del lenguaje fuente, procesador de lenguajes, o arquitectura. Mediante el uso de diagramas tipo T se explicará el funcionamiento del mismo. Las siguientes tres etapas forman parte de la fase de análisis; mientras que la última etapa forma parte de la fase de síntesis.

3. Diseño del procesador de lenguajes

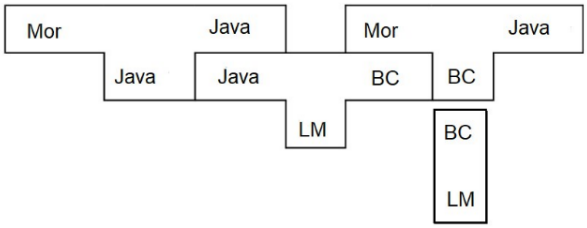
3.1. Diagramas tipo T

En primer lugar, se diseña la construcción del procesador de lenguajes empleando los diagramas **tipo T**. En este diagrama se explica el funcionamiento del procesador que se pretende implementar. Tomamos **Moor** como lenguaje fuente, y Java como lenguaje objeto; además el lenguaje que implementa es Java, es decir, nuestro compilador compila a Java, y esta escrito a Java.
Para utilizar ese compilador, utilizamos un compilador auxiliar, que está escrito en código máquina para dar lugar a bytecode, el típico archivo **.class** que generamos al compilar el archivo **.java**.
Por último tenemos que compilar el bytecode, con un compilador que acepta bytecode escrito en código máquina. Puede observar este diseño en la siguiente figura.

3.2. Lenguaje fuente

El lenguaje **Moor** que se ha definido consta de dos secciones principales:

- Sección de declaración de código
- Sección de declaración de autómatas



Diagramas tipo T.

La *zona de declaración de código* es la parte donde se asignará un comportamiento a un código ya predefinido o que se vaya a ejecutar. Por la forma en la que se ha definido el lenguaje, esta zona será como una zona de *imports* típicamente conocida en lenguajes de alto nivel.

La estructura de esta zona es: cN # *código* # donde c es el comportamiento, N es el número de comportamiento y los **tokens** # son los delimitadores del código.
La característica principal de esta zona es que solo se podrán realizar este tipo de declaraciones, ya que en la otra zona solo podremos declarar autómatas.

Ejemplo : c1 # print('hola') #

Respecto a la *zona de declaración de autómatas* se tendrá una serie de palabras que se utilizarán para la definición del autómata. No obstante en esta zona solo se podrán declarar autómatas y las funciones asociadas a los mismos, también, como en cualquier lenguaje de alto nivel, se podrán incluir los clásicos comentarios '//' o '/* Texto comentario '*/'.

Será posible la declaración de múltiples autómatas en el fichero de prueba, además, se indicará en una tabla todos los campos que son imprescindibles para la realización del mismo. Cualquier otro carácter introducido en el fichero de prueba será considerado como error.

3.3. Tabla de Token-Acción

Primero, presentamos la tabla Token-Acción. En esta tabla se describen tanto los tokens que componen nuestro lenguaje, como la función de cada uno de ellos.

Token	Acción
moore	Función para declarar un autómata seguido de un nombre que se escribirá entre llaves {} moore Ejemplo {}
estados	Indica la cantidad de estados totales que tendrá el autómata puede ser un único estado o varios separados por ',' estados q0,q1;
estado_in	Se selecciona un único estado inicial que tendrá que estar en los estados anteriores estado_in q0;
alf_in	La entrada o eventos del autómata que hará posible las transiciones entre estados, único o entre comas ',' alf_in a,b,c;
alf_out	La salida o comportamientos del autómata y tendrán que ser coincidentes con la zona de declaración de comportamientos, escrito un identificador único o entre comas ','
transicion	Será un tipo de función que se escribirá entre llaves {}, y permitirá que se transite mediante una entrada de un estado a otro: (<estado_origen>, entrada, <estado_destino>); se podrán indicar varias mediante el uso de comas ','
comportamiento	Será un tipo de función que se escribirá entre llaves {}, y permite asignar a un estado un comportamiento: (<estados>,<comportamiento>); se podrán indicar varios mediante el uso de comas ','

Notese, que para cerrar una sentencia es necesario de indicar al final de dicha sentencia el carácter ';', esto no es necesario para las funciones *moore*, *transicion* y *comportamiento*.

3.4. Tabla de tokens

En segundo lugar, creamos la tabla de tokens, dónde también mostramos lexemas de ejemplo y los patrones asociados a cada token.

Token	Lexema	Patrón
moore	moore	m·o·o·r·e
estados	estados	e·s·t·a·d·o·s
estado_in	estado_in	e·s·t·a·d·o·_·i·n
alf_in	alf_in	a·l·f·_i·n
alf_out	alf_out	a·l·f·_o·u·t
transicion	transicion	t·r·a·n·s·i·c·i·o·n
comportamiento	comportamiento	c·o·m·p·o·r·t·a·m·i·e·n·t·o
Paréntesis abierto	(	(
Paréntesis cerrado	)	)
Llave abierta	{	{
Llave cerrada	}	}
Punto y coma	;	;
Coma	,	,
Asterisco barra	*/	*·/
Barra asterisco	/*	/·*
ID	hola	[A-Za-z][A-Zaz0-9_]*
CMP	c1	c[1-9][0-9]*
CODIGO	#codigo aquí #	#· codigo aquí ·#

3.5. EBNF

Por último, en la siguiente figura puede ver una representación de las expresiones regulares que definen el léxico de nuestro lenguaje en EBNF; el metalenguaje que hemos estudiado durante la asignatura.

PROGRAMA	::=	DEC_COMP AUTOMATA {AUTOMATA}
DEC_COMP	::=	CMP CODIGO { CMP CODIGO }
CODIGO	::=	'#' ASCII '#'
AUTOMATA	::=	**moore** ID CUERPO_AUTOMATA
CUERPO_AUTOMATA	::=	'{' ESTADOS ESTADO_INI ALF_IN ALF_OUT TRANSICION COMPORTAMIENTOS '}'
ESTADOS	::=	**estados** { ID ',' } ID ';'
ESTADO_INI	::=	**estado_in** ID ';'
ALF_IN	::=	**alf_in** { EVENTOS ',' } EVENTOS ';'
ALF_OUT	::=	**alf_out** { CMP ',' } CMP ';'
TRANSICION	::=	**transicion** '{' TRANSICION_DEF { ',' TRANSICION_DEF } ';' '}'
TRANSICION_DEF	::=	'(' ID ',' ID ',' ID ')'
COMPORTAMIENTOS	::=	**comportamientos** '{' COMP_DEF { ',' COMP_DEF } ';' '}'
COMP_DEF	::=	'(' ID ',' ID ',' ID ')'
CMP	::=	'c'NUMEROS
NUMEROS	::=	0 | 1 | .. | 9
COMENTARIOS	::=	'/*' ASCII '*/'

4. Analizador Léxico

El analizador léxico tiene varias funciones:

- Reconocer los símbolos / tokens que componen el texto fuente.
- Eliminar comentarios del texto fuente.
- Eliminar espacios en blanco, saltos de línea, tabulaciones, etc.
- Informar de los errores léxicos detectados

Como salida del analizador léxico, se obtiene una representación de la cadena de entrada en forma de cadena de tokens, que será posteriormente utilizada en la fase de análisis sintáctico. Para construir el analizador sintáctico hemos utilizado dos herramientas: JFlex y ANTLR. Por un lado, JFlex es una herramienta software en la que se declaran los tokens que componen nuestro lenguaje fuente, así como las expresiones regulares asociadas a los mismos; y genera una serie de archivos que cumplen la función del analizador léxico. Por otro lado, ANTLR Jarillo.

4.1. Jflex

En esta sección se muestra el código en JFlex donde se construye el analizador léxico del lenguaje Moor.

Listing 1: Analizador Léxico en JFlex

```
//* -------------------------Seccion codigo-usuario ------------------------*/
package plmoore;
import java.util.*;
import java.io.*;
import java_cup.runtime.Symbol;

%%


/* -----------------Seccion de opciones y declaraciones -----------------*/

/*--OPCIONES --*/

%class AnalizadorAutomata
%unicode
%cup
%cupdebug
%ignorecase

/* Posibilitar acceso a la columna y fila actual de analisis*/

%line
%column


/*--DECLARACIONES --*/
```

```
%{

%}

%init{
   System.out.println("Iniciando Analizador Lexico ... ");

%init}


%eof{

   System.out.println("Fin del Analizador Lexico ... ");

%eof}

TERLINEA = \r|\n|\r\n
CARACTERIN = [^\r\n]
ESPACIOBLANCO = {TERLINEA} | [ \t\f]
COMENTARIO = {COMENTARIOTRADICIONAL} | {FINLINEACOMENT}
COMENTARIOTRADICIONAL = "/*" [^*] ~"*/" | "/*" "*"+ "/"
FINLINEACOMENT  = "//" {CARACTERIN}* {TERLINEA}?
ENTEROS = 0 | [1-9][0-9]*
CMP = c{ENTEROS}
ESTADOS = "estados"
EINICIAL = "estado_in"
ALF_OUT = "alf_out"
ALF_IN = "alf_in"
TRANS = "transicion"
COMPORT = "comportamiento"
ID = [:jletter:] [:jletterdigit:]*
MOORE = "moore"

%state CODIGO

%%

/* ------------------------Seccion de reglas y acciones ----------------------*/

<YYINITIAL> {

   {CMP} { return new Symbol(sym.CMP, yyline, yycolumn, new String(yytext())); }
   "#" {yybegin(CODIGO); return new Symbol(sym.ALM_OP, yyline, yycolumn, new
      String(yytext())); }
   {MOORE} { return new Symbol(sym.MOORE, yyline, yycolumn, new String(yytext()));}
   "{" { return new Symbol(sym.LLCORCH_OP, yyline, yycolumn, new String(yytext()));}
   "}" { return new Symbol(sym.LLCORCH_CL, yyline, yycolumn, new String(yytext()));}
   {ESTADOS} { return new Symbol(sym.ESTADOS, yyline, yycolumn, new
      String(yytext()));}
   {EINICIAL} { return new Symbol(sym.ESTADO_INI, yyline, yycolumn, new
      String(yytext()));}
   {ALF_IN} { return new Symbol(sym.ALF_IN, yyline, yycolumn, new
```

```
      String(yytext()));}
   {ALF_OUT} { return new Symbol(sym.ALF_OUT, yyline, yycolumn, new
      String(yytext()));}
   {TRANS}  {  return new Symbol(sym.TRANS, yyline, yycolumn, new String(yytext()));}
   {COMPORT} { return new Symbol(sym.COMPORTAMIENTO, yyline, yycolumn, new
      String(yytext()));}
   "," { return new Symbol(sym.COMA, yyline, yycolumn, new String(yytext()));}
   ";" { return new Symbol(sym.PUNTO_COMA, yyline, yycolumn, new String(yytext()));}
   "(" { return new Symbol(sym.LLPARENT_OP, yyline, yycolumn, new
      String(yytext()));}
   ")" { return new Symbol(sym.LLPARENT_CL, yyline, yycolumn, new
      String(yytext()));}
   {ID} {return new Symbol(sym.ID, yyline, yycolumn, new String(yytext()));}
   {COMENTARIO}                 {/*Se ignoran los comentarios */}
   {ESPACIOBLANCO}              {/*Se ignoran los espacios en blanco */}
}

<CODIGO>{
   . + /"#" { return new Symbol(sym.CODIGO, yyline, yycolumn, new
      String(yytext()));}
   "#" { yybegin(YYINITIAL); return new Symbol(sym.ALM_CL, yyline, yycolumn, new
      String(yytext()));}
}

[^]          { String errLex = "Error lexico : '"+yytext()+"' en la linea:
    "+(yyline+1)+" y columna: "+(yycolumn+1);
      System.out.println(errLex); }
```

4.2. ANTLR

Jarillo

5. Analizador sintáctico

El analizador sintáctico tiene varias funciones:

- Analizar la secuencia de tokens y verificar si son correctos sintácticamente.
- Obtener una representación interna del texto.
- Informar de los errores sintácticos detectados.

En resumen, dada la cadena de tokens ontenida como resultado de la fase de análisis léxico, se compruena que dicha secuencia está escrita correctamente; y se obtiene una representación interna de la misma, que servirá como entrada para el proceso de análisis semántico.

Como sabemos, existen dos formas de realizar el análisis sintáctico (ascendente y descendente). Nosotros hemos escogido construir un analizador sintáctico ascendente con CUP y ANTLR.

Por un lado, CUP es una herramienta que hemos utilizado durante las prácticas de la asignatura; y que permite declarar las producciones que constituyen las reglas sintácticas del lenguaje fuente; además de reglas semánticas asociadas a las mismas, que se tratarán en más detalle en la siguiente sección.

En la sección .Archivo Cup"puede ver el código asociado a esta fase de la construcción del procesador de lenguajes.

5.1. Control de errores en cup

Para controlar los errores sintácticos en CUP, se han utilizado producciones de error. Las producciones de error utilizan un símbolo terminal .error"; de tal manera que cuando se produce una reducción al mismo, se invoque a una rutina de error asociada a este símbolo. En esta rutina de error, se invoca al método report_error de la clase Parser.

En la sección .Archivo Cup"puede ver el código asociado a esta fase de la construcción del procesador de lenguajes.

5.2. Antlr

Jarillo.

6. Analizador semántico

El analizador semántico tiene varias funciones:

- Dar significado a las construcciones del lenguaje fuente.
- Generación de código.
- Acabar de completar el lenguaje fuente.

La tercera función del analizador semántico tiene que ver con aspectos como controlar que la cadena introducida esté formada por elementos del alfabeto de entrada, que los estados que se declaren en la máquina de Moore sean los únicos a los que se puede transitar, o que no se declare varias veces una transición; entre otros aspectos.

Medina - Esto de abajo lo puedes usar, o no. Lo que tú quieras

Para controlar este tipo de errores, se han introducido reglas semánticas en las producciones declaradas en el archivo CUP mencionado en la sección anterior. En estas reglas se controlan todos los aspectos mencionados con anterioridad, además de añadir funcionalidad al código generado. Es decir, este conjunto de reglas semánticas no solo tienen el propósito de completar el lenguaje fuente, sino de dar funcionalidad a la máquina (o máquinas) de Moore que se declaran en el mismo.

En la sección .Archivo Cup"puede ver el código asociado a esta fase de la construcción del procesador de lenguajes.

6.1. ANTLR

6.2. Generación de código

Javier

7. Archivo Cup

En este archivo se muestran tanto las producciones asociadas a la gramática y utilizadas para realizar el análisis sintáctico, como las reglas semánticas asociadas a cada producción; que permiten el análisis semántico.

Listing 2: Analizador Sintáctico y Semántico en CUP

```
/* ----------------------Seccion de declaraciones package e
    imports--------------------*/
package plmoore;

import java.io.*;
import java_cup.runtime.*;
import java.util.ArrayList;
import java.util.Hashtable;

/* ----------------------Seccion componentes de codigo de usuario
    --------------------*/
parser code {:


   public void report_error(String message, Object info) {

       /* Crea un StringBuffer llamado 'm' con el string 'Error' en el. */

       StringBuffer m = new StringBuffer("Error");

       /* Chequea si la informacion pasada al metodo es del mismo
          tipo que el tipo java_cup.runtime.Symbol. */

       if (info instanceof java_cup.runtime.Symbol) {

           /* Declara un objeto 's' del tipo java_cup.runtime.Symbol con la
              informacion que hay en el objeto info que esta siendo convertido
              como un objeto java_cup.runtime.Symbol. */
           java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);

           /* Chequea si el numero de linea en la entrada es mayor o
              igual que cero. */
           if (s.left >= 0) {
               /* Anade al final del mensaje de error StringBuffer
                  el numero de linea del error en la entrada. */
               m.append(" en la linea "+(s.left+1));
               /* Chequea si el numero de columna en la entrada es mayor
                  o igual que cero. */
               if (s.right >= 0)
                   /* Anade al final del mensaje de error StringBuffer
                      el numero de columna del error en la entrada. */
                   m.append(", columna "+(s.right+1));
           }
       }
```

```
        /* Anade al final del mensaje de error StringBuffer creado en
           este metodo el mensaje que fue pasado a este metodo. */
        m.append(" : "+message);

        /* Imprime los contenidos del StringBuffer 'm', que contiene
           el mensaje de error. */
        System.err.println(m);
    }
    public void report_fatal_error(String message, Object info) {
        report_error(message, info);
        System.exit(1);
    }

    public void syntax_error(Symbol s){
        System.out.println("Error recuperable de sintaxis: "+s.value+" Linea
            "+(s.left+1)+" columna "+(s.right+1) );
    }

    public void unrecovered_syntax_error(Symbol s) throws java.lang.Exception{
        System.out.println("Error no recuperable de sintaxis: "+s.value+" Linea
            "+(s.left+1)+" columna "+(s.right+1) );
    }


:}

action code {:

   MoldeAutomataMoore machine;
   int contador = 0;
        ArrayList<MoldeAutomataMoore> maquinas = new
            ArrayList<MoldeAutomataMoore>();;
   Hashtable<String, String> comp_codigo = new Hashtable<String, String>();
        String fallos_ejecucion = "";
        boolean continuar = true;
:}

/* ------------Declaracion de la lista de simbolos de la gramatica-----------*/
non terminal programa, dec_comp, dec_automata, cuerpo_automata, estados,
    dec_estados, estado_ini, alf_in, dec_alf_in, alf_out, dec_alf_out, transicion,
    dec_transicion, comportamientos, dec_comportamientos;
non terminal MoldeAutomataMoore automata;
non terminal String codigo, comp_def, transicion_def;


terminal String CMP, CODIGO, MOORE, ID, LLCORCH_OP, LLCORCH_CL, ESTADOS,
    ESTADO_INI, ALF_IN, ALF_OUT, TRANS, COMPORTAMIENTO, COMA, PUNTO_COMA,
    LLPARENT_OP, LLPARENT_CL, ALM_OP, ALM_CL;


/* --------------Declaracion de la gramatica -----------*/

programa ::= dec_comp dec_automata {:
```

```
        System.out.println("Analisis finalizado");
        if(continuar){
           System.out.println("Generando fichero...");
           GeneracionCodigo generar = new GeneracionCodigo(maquinas);

        }else{
           System.out.println("Hubo fallos durante la ejecucion");
           System.out.println(fallos_ejecucion);
        }

                                  :} ;

dec_comp ::= CMP:com codigo:code {: comp_codigo.put(com, code); :} dec_comp
      | CMP:com codigo:code {: comp_codigo.put(com, code); :} ;

dec_automata ::= automata:machin {:
                                   System.out.println("Maquina identificada!
                                      n_maquinas: "+ ++contador); maquinas.add(machin);
                                 :} dec_automata

      | automata:machin {:
                                   System.out.println("Maquina identificada!
                                      n_maquinas: "+ ++contador); maquinas.add(machin);
                                  :};

codigo ::= ALM_OP CODIGO:code ALM_CL {: RESULT=code; :}
        | error {:
           fallos_ejecucion += " Error declarando codigo de usuario\n";
           parser.report_error(fallos_ejecucion,null);
           continuar = false;
        :};


automata ::= MOORE ID:id {: machine = new MoldeAutomataMoore(id); :}
    cuerpo_automata {: RESULT = machine; :}
        | error {:
           fallos_ejecucion += " Error declarando maquina de Moore\n";
           parser.report_error(fallos_ejecucion,null);
           continuar = false;
        :};


cuerpo_automata ::= LLCORCH_OP estados estado_ini alf_in alf_out comportamientos
    transicion LLCORCH_CL
        | error {:
           fallos_ejecucion += " Error declarando maquina de Moore\n";
           parser.report_error(fallos_ejecucion,null);
           continuar = false;
        :};

estados ::= ESTADOS dec_estados PUNTO_COMA
        | error {:
```

```
          fallos_ejecucion += " Error declarando estados\n";
          parser.report_error(fallos_ejecucion,null);
          continuar = false;
        :};

dec_estados ::= ID:id {:
                        if(!machine.addEstado(id)){
fallos_ejecucion += "In line: "+((idleft)+1)+" El estado "+id+" ya esta declarado
    \n"; continuar = false;}

                    :} COMA dec_estados

         | ID:id {:
                        if(!machine.addEstado(id)){fallos_ejecucion += "In line:
                            "+((idleft)+1)+" El estado "+id+" ya esta declarado
                            \n"; continuar = false;}
                    :} ;



estado_ini ::= ESTADO_INI ID:id {: machine.setEstado_inicial(id); :} PUNTO_COMA
        | error {:
          fallos_ejecucion += " Error declarando estado inicial\n";
          parser.report_error(fallos_ejecucion,null);
          continuar = false;
        :};


alf_in ::= ALF_IN dec_alf_in PUNTO_COMA
        | error {:
          fallos_ejecucion += " Error declarando alfabeto de entrada\n";
          parser.report_error(fallos_ejecucion,null);
          continuar = false;
        :};

dec_alf_in ::= ID:id {: if(!machine.addEvento(id)){fallos_ejecucion += "In line:
    "+((idleft)+1)+" El evento "+id+" ya esta declarado \n"; continuar = false;}

                    :} COMA dec_alf_in

     | ID:id {: if(!machine.addEvento(id)){fallos_ejecucion += "In line:
         "+((idleft)+1)+" El evento "+id+" ya esta declarado \n"; continuar =
         false;}
                    :} ;

alf_out ::= ALF_OUT dec_alf_out PUNTO_COMA
        | error {:
          fallos_ejecucion += " Error declarando alfabeto de salida\n";
          parser.report_error(fallos_ejecucion,null);
          continuar = false;
        :};

dec_alf_out ::= CMP:id {: if(!machine.addComp(id)){fallos_ejecucion += "In line:
```

```
    "+((idleft)+1)+" El comportamiento "+id+" ya esta declarado \n"; continuar =
    false;}

                    :} COMA dec_alf_out

      | CMP:id {: if(!machine.addComp(id)){fallos_ejecucion += "In line:
          "+((idleft)+1)+" El comportamiento "+id+" ya esta declarado \n"; continuar
          = false;}
                      :} ;

comportamientos ::= COMPORTAMIENTO LLCORCH_OP dec_comportamientos PUNTO_COMA
    LLCORCH_CL
        | error {:
          fallos_ejecucion += " Error declarando alfabeto de comportamientos\n";
          parser.report_error(fallos_ejecucion,null);
          continuar = false;
        :};

dec_comportamientos ::= comp_def:tupla {: boolean error =
    machine.addComportamiento(tupla);

                if(!error){ fallos_ejecucion += "In line: "+((tuplaleft)+1)+" El
                    comportamiento "+tupla+" ya esta declarado \n"; continuar =
                    false; }

                            :} COMA dec_comportamientos

                | comp_def:tupla {: boolean error =
                    machine.addComportamiento(tupla);
                   if(!error){ fallos_ejecucion += "In line: "+((tuplaleft)+1)+"
                       El comportamiento "+tupla+" ya esta declarado \n";
                       continuar = false; }
                                :} ;

comp_def ::= LLPARENT_OP ID:id COMA CMP:comp LLPARENT_CL {:
    RESULT=id+"-"+comp+"-"+comp_codigo.get(comp); :}
        | error {:
          fallos_ejecucion += " Error declarando alfabeto comportamientos\n";
          parser.report_error(fallos_ejecucion,null);
          continuar = false;
        :};

transicion ::= TRANS LLCORCH_OP dec_transicion PUNTO_COMA LLCORCH_CL
        | error {:
          fallos_ejecucion += " Error declarando alfabeto de transiciones\n";
          parser.report_error(fallos_ejecucion,null);
          continuar = false;
        :};

dec_transicion ::= transicion_def:tupla {: if(!machine.addTransicion(tupla)){
                                        fallos_ejecucion += "In line:
                                            "+((tuplaleft)+1)+" El comportamiento
                                            "+tupla+" ya esta declarado \n";
```

```
                                        continuar = false; }
                                    :} COMA dec_transicion

    | transicion_def:tupla {: if(!machine.addTransicion(tupla)){
                                        fallos_ejecucion += "In line:
                                            "+((tuplaleft)+1)+" El comportamiento
                                            "+tupla+" ya esta declarado \n";
                                            continuar = false; }
                                    :} ;

transicion_def ::= LLPARENT_OP ID:estado_in COMA ID:evento COMA ID:estado_out
   LLPARENT_CL {: RESULT=estado_in+"-"+evento+"-"+estado_out; :} ;
```

8. Bibliografía

Referencias

[1] Autómatas de Moore https://es.wikipedia.org/wiki/M%C3%A1quina_de_Moore