UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Procesadores de Lenguajes

GRADO EN INGENIERÍA INFORMÁTICA

Mooma, un procesador de máquinas de Moore

Autor: Javier Córdoba Romero, Juan José Corroto Martin y Fernando Vallejo Banegas

Enero, 2019

Índice

1. Descripción del Problema **3**

2. Lenguaje Mooma **3**

2.1. Tabla de Tokens 6

2.2. Ejemplo de Máquina de Moore en Mooma 6

2.3. Compilando Mooma 7

3. Analizador usando ANTLR **7**

3.1. Gramática ANTLR 8

3.2. Análisis semántico 12

3.2.1. Funcionamiento general con ANTLR 12

3.2.2. Implementación del analizador semántico Mooma 13

3.3. Detección de errores 14

3.3.1. Errores en el análisis léxico y sintáctico 14

3.3.2. Errores en el análisis semántico 14

4. Analizador usando Flex+Cup **15**

4.1. Análisis Léxico 15

4.2. Analizador Sintáctico+Semántico 15

4.3. Uso de las herramientas flex y cup 18

5. Ejemplos de uso **19**

5.1. PredictionBuffer 19

5.2. IA de enemigos en un videojuego 23

1. Descripción del Problema

En este proyecto nos enfrentamos a un problema en el cual se nos pide crear un procesador de lenguaje para cierto Lenguaje de definición de máquinas Moore. Para ello vamos a necesitar la implementación de tres analizadores: léxico, sintáctico y semántico ,es decir, para este proyecto primero necesitamos crear un lenguaje para la definición de máquinas de Moore, y respecto a ese lenguaje va a trabajar nuestro "traductor", el cual será capaz de originar la misma máquina de Moore pero en un lenguaje distinto, lenguaje objeto (JAVA). Debido a que este proyecto abarca todo el proceso de la creación de un procesador de lenguaje podemos centrarnos en aprender, de forma práctica, el uso de los procesadores de lenguaje, ya que el dominio del problema (Máquinas de Moore) nos es conocido a todos.

Una máquina de Moore es un tipo de máquina secuencial en la que la respuesta solo depende del estado actual de la máquina y es independiente de la entrada. Una máquina de Moore es una estructura de la forma: $Mo = (Q, Ent, Sal, tran, res, q_0)$ Siendo:

- Q: conjunto de estados
- Ent: alfabeto de entrada
- Sal: alfabeto de salida
- $tran$: $QxEnt \rightarrow Q$, funcion de transicion
- res: $Q \rightarrow Sal$, funcion de respuesta
- q_0: estado inicial

Como podemos ver en el ejemplo [1] una Máquina de Moore es un autómata compuesto de estados (representados por letras mayúsculas) asociados a una salida (en este caso un número que se encuentra separado del nombre del estado por una barra "/") y transiciones típicas entre estados. En este ejemplo la salida cambia a 1 y permanece así cuando al menos la cadena: "0011" aparece en la entrada (estado I). En el resto de estados la salida es 0.

2. Lenguaje Mooma

Para crear estas Máquinas de Moore hemos hecho un lenguaje específico de Dominio al que hemos llamado "Mooma" [1].

Listado 1: EBNF de nuestro lenguaje de definición de Máquinas de Moore (Mooma).

```
1 PROGRAM ::= [ENVIRONMENT] {OUTPUT} {DEFINE} {AUTOMATON};
2 ENVIRONMENT ::= "environment" ":=" {LETRA} ";";
3 DEFINE ::= "define" IDENT "{" ENTRADA ";" SALIDA ";" "}";
```

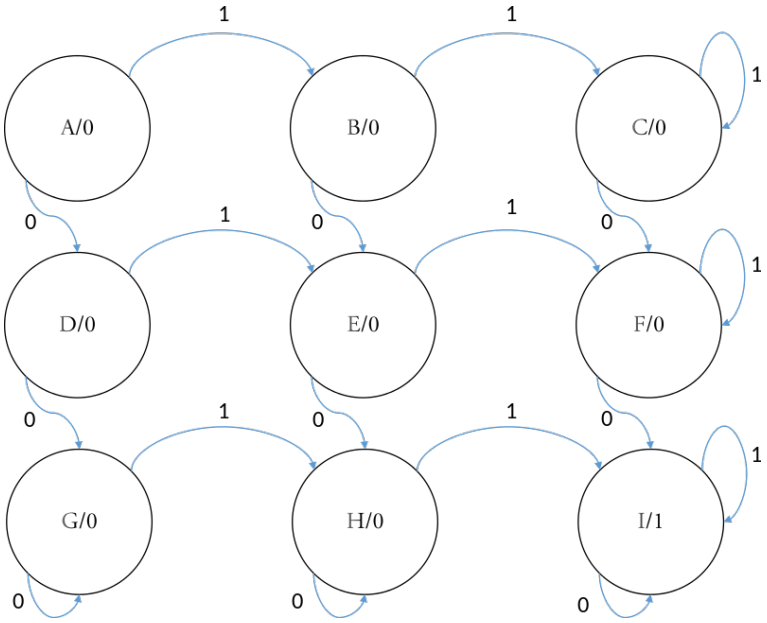

Figura 1: Ejemplo de Máquina de Moore.

```
AUTOMATON ::= "automaton" IDENT "(" IDENT ")" "{" STATES ";" INITIAL
   ";" TRANSITIONS "}";
ENTRADA ::= "in" ":=" {EVENTO","}[EVENTO];
SALIDA ::= "out" ":=" {IDENT","}[IDENT];
STATES ::= "states" ":=" [{IDENT "|" EVENTO,}] IDENT "|" EVENTO;
INITIAL ::= "initial" ":=" IDENT;
TRANSITIONS ::= "transitions" "{" {IDENT "|" {EVENTO,}EVENTO "->"
   IDENT ";"} "}";
IDENT ::= LETRA {LETRA|DIGITO};
LETRA ::= "A" | "B" | "C" | "D" | "E" | "F" | "G"
       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
       | "c" | "d" | "e" | "f" | "g" | "h" | "i"
       | "j" | "k" | "l" | "m" | "n" | "o" | "p"
       | "q" | "r" | "s" | "t" | "u" | "v" | "w"
       | "x" | "y" | "z" ;
DIGITO ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
   ;
EVENTO ::= (1|2|3|4|5|6|7|8|9){DIGITO};
OUTPUT ::= IDENT ":=" CODIGO ";";
CODIGO ::= "{:" {ASCII} ":}";
```

```
23 COMMENT ::= "/*" {ASCII} "*/";
```

Un dato importante para tener en cuenta es que nuestro lenguaje trabaja con eventos, es decir la creación del diccionario de correspondencia entre eventos y las entradas del problema ha de ser diseñado por el usuario para la comprensión del resultado.

Los programas Mooma se encuentran divididos en cuatro partes:

1. Declaración del entorno, aquí se incluirá una construcción que comenzará por ”environment” seguido del nombre o ruta del archivo que contiene el entorno (una clase Java que implementa una interfaz creada por nosotros, y que el usuario debe crear).

2. Código asociado a las salidas de los estados, que necesitará un identificador para poder ser usado posteriormente en la parte de declaración de entradas y salidas (marcado con *define*) y la declaración del autómata (marcado con *automaton*).

3. Declaración de entradas y salidas (eventos y código). Aquí se incluirá una construcción que empieza con la palabra “define”, seguido de un identificador, y se definirán un alfabeto de entrada (Recordamos que este alfabeto siempre consistirá en números que denotan eventos), y un alfabeto de salida. Este alfabeto de salida será una serie de identificadores definidos anteriormente, y denotan fragmentos de código en java que serán ejecutados como salida de un estado.

4. Declaración de un autómata concreto. Ésta será otra construcción que empezará con la palabra “automaton” y, entre paréntesis, el identificador de una construcción “define” (Lo cual denotará los alfabetos de entrada y salida del autómata). Dentro de la construcción se incluirán el conjunto de estados del autómata junto con el identificador de su salida asociada, el estado inicial y una lista con las transiciones entre estados del autómata.

Además, nuestro lenguaje tiene la opción de añadir comentarios empezando por “/*” y terminando por “*/”, pudiendo poner cualquier cosa entre esas dos cadenas.

El *Environment* no es obligatorio. En el *Environment* se puede implementar la función translate, encargada de traducir la entrada que el usuario quiere que tenga la máquina de Moore en eventos, que es lo que la máquina tendrá como lenguaje de entrada. En el apartado de ejemplos se podrá ver mejor el uso del *Environment*.

2.1. Tabla de Tokens

La tabla de tokens del lenguaje Mooma [1] contiene todos los tokens del lenguaje junto con un patrón que lo define y un lexema de ejemplo. En azul se pueden observar las palabras reservadas del lenguaje.

Token	**Patron**	**Lexema ejemplo**
environment	environment	environment
define	define	define
automaton	automaton	automaton
in	in	in
out	out	out
states	state	states
initial	initial	initial
transition	transitions	transitions
output	output	output
Llave_derecha	}	}
Llave_izquierda	{	{
Punto_y_coma	;	;
coma	,	,
asignacion	:=	:=
Par_derecho	)	)
Par_izquierdo	(	(
Flecha	->	->
Codigo_inicio	{:	{:
Codigo_final	:}	:}
Separator	|	|
Event	[1-9][0-9]*	37
Identificador	[A-Za-z][A-Za-z0-9]*	Foo5
Codigo	.*	System.out.println();

Cuadro 1: Tabla de tokens del lenguaje Mooma.

2.2. Ejemplo de Máquina de Moore en Mooma

Como ejemplo hemos definido una máquina bastante simple [2].

Primero podemos observar que se define el entorno, en este caso este entorno tendría un método o función llamado printear que mostraría un mensaje por la salida estándar.

A continuación podemos ver la zona de definición de código, donde se declaran las salidas posibles. Ahí también podemos ver el uso de los comentarios que se ignorarían completamente y no afectan a la ejecución.

Posteriormente tenemos la zona de definición de alfabetos donde en este caso definimos solo uno, que será el usado por nuestro autómata ejemplo. Este alfabeto solo tiene 2 eventos y como salida los dos codigos definidos anteriormente.

Como se puede apreciar tenemos un automata al que le pasamos el alfabeto anteriormente definido y tiene: 2 estados con sus salidas correspondientes, el estado q0 como inicial y 4 transiciones. Es decir cada vez que la máquina de Moore creada para el ejemplo alcance el estado q0 el código descrito en code1 será ejecutado, análogamente ocurre para el estado q1 con code2.

Por otra parte hay que destacar como son descritas las transiciones, como se puede observar son de la forma: $EstadoOrigen|EventoOcurrido \rightarrow EstadoDestino$.

2.3. Compilando Mooma

En nuestro caso, realizaremos el análisis de Mooma con dos herramientas completamente diferente:

- En el caso de ANTLR4 nos encontramos ante una implementación en Python (lenguaje interpretado), por lo que su diagrama de T es de la forma [3].
- En el caso de JFlex+Cup, nos encontramos con implementaciones en Java (lenguaje compilado e interpretado en máquina virtual), su diagrama de T es de la forma [4].

Ambos analizadores tienen el mismo objetivo, generar código java con la Máquina de Moore implementada, por lo que podemos apreciar como la parte derecha de ambos diagramas [3, 4] son idénticas, ya que generarán el mismo código Java que debe ser compilado e interpretado por su máquina virtual finalmente.

3. Analizador usando ANTLR

ANTLR es un procesador de lenguaje que toma como entrada una gramática, escrita en un archivo en el lenguaje propio de ANTLR con extensión *.g4*, y da como resultado un analizador léxico y un analizador sintáctico descendente implementado mediante funciones recursivas. Estos analizadores, para los cuales podemos escoger un lenguaje de programación de salida (En nuestro caso **Python3**), son fácilmente integrables en cualquier programa, y nos dejaría con la única responsabilidad de construir la parte de análisis semántico.

```
environment := Domain.Env;

code1 := {:
        printear("Hello");
        :};
code2 := /* HOLA \asd <> -.*/{:
        /* ADIOS \asd <> -.* */
        printear("Adios");
        :};


define alfabetos1{
    in := 1, 2;
    out := code1, code2;
}

automaton auto1 (alfabetos1){
    states := q0|code1, q1|code2;
    initial := q0;
    transitions{
        q0|1 -> q1;
        q0|2 -> q0;
        q1|1 -> q1;
        q1|2 -> q0;
    }
}
```

Figura 2: Ejemplo de Máquina de Moore usando Mooma.

3.1. Gramática ANTLR

Para construir un procesador de lenguajes con ANTLR lo primero que tenemos que hacer es escribir nuestra gramática en formato *.g4*. La gramática completa se puede ver en el archivo mooma.g4. A continuación vamos a discutir los aspectos más importantes:

La gramática ANTLR se divide en 2 tipos de reglas: Las reglas léxicas y las reglas sintácticas. Las reglas léxicas, aquellas reglas que empiezan por mayúscula, sirven para definir los tokens que el *lexer* va a analizar, y los que usaremos como terminales en la parte de las reglas sintácticas. Estas reglas sintácticas, escritas empezando por minúscula, son las que denotan la estructura del programa, y las que el *parser* utilizará para realizar su análisis sintáctico.

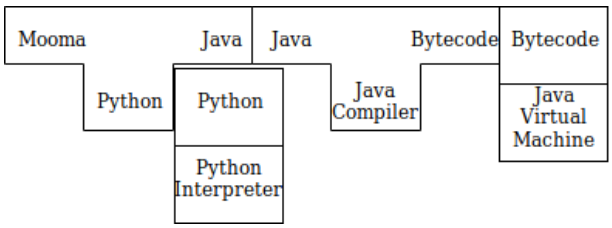

Figura 3: Diagrama de T para ANTLR4.

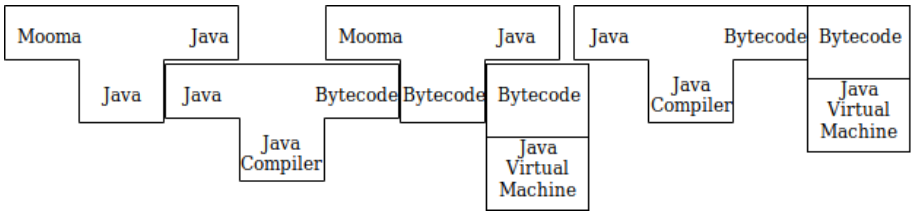

Figura 4: Diagrama de T para jflex y cup.

Es importante notar que, debido a la gramática que hemos utilizado, podemos reconocer las palabras reservadas tanto en letra minúscula como en mayúscula, , incluso alternando estas (e.g. “EnvIroNMent” se reconocería correctamente como el token “environment”) y no se puede definir el evento 0.

Aunque ANTLR es capaz de solucionar por sí solo las conflictos entre iniciales de la gramática, hemos decidido aplicar lo aprendido en clase y solucionarlo nosotros. Esto resulta en una gramática más compleja y difícil de manejar en la etapa semántica.

Una vez hemos terminado de escribir nuestra gramática, ejecutamos la herramienta de ANTLR de la siguiente manera (Se usa la estructura de directorios de nuestro repositorio):

```
In
  : I N
  ;

Out
  : O U T
  ;

Trans
  : T R A N S I T I O N S
  ;

Ini
  : I N I T I A L
  ;
```

Figura 5: Reglas léxicas para tokens de palabras reservadas.

```
/*
 * Generic tokens
 */

Ident
  : Letra (Letra | Digito)*
  ;

Evento
  : ('1' .. '9') Digito*
  ;

Codigo
  : '{:' (.)*? ':}'
  ;

Comentario
  : '/*' (.)*? '*/' -> skip
  ;
```

Figura 6: Reglas léxicas para algunos de los tokens utilizados.

Listado 2: Ejecución de ANTLR.

```
java -jar .\lib\antlr-4.7.1-complete.jar -Dlanguage=Python3 .\
   Lenguaje\mooma.g4
```

El argumento *-DLanguage=Python3* se usa para elegir el lenguaje objeto de ANTLR, en nuestro caso Python3. La ejecución del anterior comando nos genera multitud de archivos, entre los que se encuentran el *lexer* y el *parser* que nos interesan. Ahora solo necesitamos escribir un pequeño programa que sirva para utilizarlos, como el siguiente:

Listado 3: Programa Main para la ejecución de los analizadores léxico y sintáctico.

```
def main(args):
    file = FileStream(args.input_file)

    errorListener = moomaErrorListener()

    lexer = moomaLexer(file)
    stream = CommonTokenStream(lexer)
    parser = moomaParser(stream)
    parser.removeErrorListeners()
    parser.addErrorListener(errorListener)
    tree = parser.program()

```

```
/*
 * Parser rules
 */

program
  : environment l_output l_define l_automaton
  ;

environment
  : Env Asignacion Ident Punto_y_coma
  | ;

l_output
  : output l_output
  | ;

output
  : Ident Asignacion Codigo Punto_y_coma
  ;

l_define
  : define l_define
  | ;
```

Figura 7: Ejemplo de reglas sintácticas.

```
13      args.output = os.path.abspath(args.output)
14      if not os.path.isdir(args.output):
15          sys.stderr.write("Directory not found: {}\n".format(args.
    output))
16          sys.exit(2)
17
18      listener = MyMoomaListener(args.output)
19      walker = ParseTreeWalker()
20      walker.walk(listener, tree)
21
22      write_parsed_to_file(listener, args.package)
23
24      listener.file.close()
```

En este programa se pueden apreciar varias cosas: Es necesario crear un objeto *lexer* y un objeto *parser*. Esto hay que hacerlo en cualquier programa que utilice ANTLR. También se puede notar cómo usamos un listener de error personalizado, y un listener llamado **MyMoomaListener**, el cuál es el encargado de realizar el análisis semántico y del cuál hablaremos a continuación.

3.2. Análisis semántico

3.2.1. Funcionamiento general con ANTLR

ANTLR funciona realizando un análisis semántico con árbol de análisis sintáctico (Aunque también presenta la posibilidad de añadir acciones semánticas a la gramática descrita anteriormente, realizando el análisis semántico con un método sin recuerdo). El *parser* da como resultado un objeto de tipo *ParseTree*, el cuál recorremos con el mecanismo que ANTLR nos proporciona, llamado *ParseTreeWalker* (líneas 19 y 20 del programa anterior). Este método realiza un recorrido en profundidad del árbol sintáctico, y utiliza un *listener* para ejecutar la acción semántica que nosotros queramos en cada nodo del árbol.

El modo que tiene ANTLR de funcionar semánticamente es el siguiente: Para cada regla sintáctica definida habrá dos funciones, una de entrada y una de salida. Estas funciones se ejecutarán al visitar un nodo de esa regla sintáctica. Por ejemplo: tenemos una regla sintáctica llamada automaton y que denota la estructura de una construcción *Automaton*.

Listado 4: Regla sintáctica automaton.

```
1 automaton
2   : Auto Ident Par_izquierdo Ident Par_derecho Llave_izquierda states
    initial transitions Llave_derecha
3   ;
```

El *listener* que nosotros entremos tendrá acceso a las siguientes 2 funciones:

- enterAutomaton: se ejecuta cuando el recorrido del árbol llegue al nodo *automaton*.
- exitAutomaton: se ejecuta cuando todos los nodos hijos de *automaton* hayan sido evaluados.

Habrá tantas funciones de entrada y salida como reglas **sintácticas** tengamos. Las reglas léxicas no tienen acceso a estas funciones, siendo los tokens definidos **propiedades** de cada nodo. En nuestro ejemplo anterior, el nodo *automaton* tendrá las propiedades Auto, Ident, Par_izquierdo Par_derecho Llave_izquierda y Llave_derecha, siendo el valor de cada una el lexema del token al que refieren.

3.2.2. Implementación del analizador semántico Mooma

Para conseguir recoger toda la información necesaria para acabar escribiendo el archivo Java que queremos como resultado, hemos decidido realizar una aproximación orientada a objetos, teniendo las clases Lenguaje, Automata y Transición. Construiremos objetos cada vez que nos encontremos con una construcción en nuestro archivo de entrada, teniendo al final una lista de lenguajes y automatas, cada uno con la información que contienen en el archivo ".moo".

A continuación un ejemplo de función, en la cual creamos un objeto de la clase Lenguaje:

Listado 5: Función enterDefine.

```
def enterDefine(self, ctx:moomaParser.DefineContext):
        # Check if a language identifier has already been defined
        if str(ctx.Ident()) in self.languages:
            self.error = True
            self.errorLog.append("Identificador de lenguaje ya
   definido: {}".format(str(ctx.Ident())))

            # Create dummy language
            self.languages.append(Language("Dummy"))
            self.languageError = True
        else:
            self.languages.append(Language(str(ctx.Ident())))
```

Como se puede ver, la mayor parte del código está compuesta por la comprobación de posibles errores semánticos, de los cuales hablaremos más adelante.

El analizador acabará escribiendo en el archivo Machines.java los autómatas definidos en el archivo fuente, si no ha habido ningún error, junto con todos los archivos necesarios para la ejecución de los autómatas, dejando al usuario la responsabilidad de crear un programa que utilice los autómatas. En caso de error o errores, se escribirá por pantalla la lista de errores encontrados.

3.3. Detección de errores

3.3.1. Errores en el análisis léxico y sintáctico

El análisis léxico y semántico se realiza mediante los analizadores generados automáticamente por ANTLR, pero aún así podemos modificar cómo se trata el error.

Por defecto, ANTLR comprueba los errores del *lexer* y del *parser* mediante lo que llama como *ErrorListener*. Este listener tiene funciones que se llaman cuando ese error ocurre. La que más nos interesa es *SyntaxError*. Aunque su nombre indique que se llama cuando el *parser* encuentra un error, esta funcion también se llama en la fase del análisis léxico cuando se encuentra un token mal formado. La información que nos proporciona consiste en la línea y columna en la que el error se ha encontrado, así como un mensaje de error bastante descriptivo del error. El siguiente mensaje se puede replicar si se intenta procesar el archivo "test2.moo"de nuestro repositorio.

Listado 6: Mensaje de error para test2.moo.

```
line 19:0 no viable alternative at input '}'
}
^
```

Se ha optado por personalizar un poco el mensaje de error que da el *ErrorListener* por defecto, añadiendo la línea de error con la parte conflictiva, como se puede ver en el siguiente ejemplo (Sta es el nombre del token *states)*:

Listado 7: Mesaje de error para un token mal formado.

```
line 43:4 mismatched input 'stes' expecting Sta
    stes := estate1|code1, estate2|code2, estate3|code2;
    ^^^^
```

3.3.2. Errores en el análisis semántico

Además de recuperar toda la información semántica para generar los archivos Java necesarios, el analizador semántico también necesita comprobar que ciertos errores no han sido cometidos

que son imposibles de encontrar en el análisis léxico y sintáctico. Los errores que se comprueban en nuestro analizador semántico son:

- Identificadores de código de salida repetidos.
- Identificadores de lenguaje repetidos.
- Evento repetido en un lenguaje.
- Identificador de salida repetido en un lenguaje.
- Identificador de autómata repetido.
- El identificador de lenguaje asociado a un autómata no ha sido definido.
- Identificador de estado repetido en un mismo autómata.
- El identificador de salida asociado a un estado no existe en el lenguaje de salida del autómata.
- El estado inicial no es un estado del autómata.
- El estado de origen o de destino de una transición no es un estado del autómata.
- El evento de una transición no forma parte del lenguaje de entrada.

Si cualquiera de estos errores ocurre, el análisis no para. Al final del análisis se mostrarán por pantalla todos los errores semánticos que han ocurrido, y no se generará ningún archivo Java.

4. Analizador usando Flex+Cup

4.1. Análisis Léxico

Hemos realizado el análisis léxico usando JFlex, esta parte solo se encarga de reconocer cada uno de los tokens de [1] de forma que serán la entrada del análisis sintáctico que hemos realizado en Cup. El reconocimiento de estos tokens se ha realizado mediante expresiones regulares, de forma análoga a como se ha visto en clase.

De esta parte cabe destacar que las palabras reservadas son reconocidas tanto en letra minúscula como mayúscula (Igual que en el analizador construido con ANTLR). Además, en esta parte, al reconocer cada uno de los tokens ya se indica su tipo de dato (en Java), en nuestro caso todos los tokens se pasan al analizador sintáctico como tipo String.

4.2. Analizador Sintáctico+Semántico

En este caso, con cup realizamos un análisis sintáctico ascendente, la gramática usada es exactamente la misma que usamos en el analizador descendente de ANTLR4. Podríamos haber creado

una gramática distinta ya que en este caso la recursividad a izquierda no es un problema, pero por simplicidad hemos decidido mantenerla.

Cup realiza el análisis semántico al mismo tiempo que el sintáctico, de forma que a las producciones de la gramática hay que añadirles unas reglas semánticas para dar significado a dichas producciones. En nuestro caso (en el archivo moomaParser.cup) el funcionamiento es sencillo, tenemos unas estructuras globales (a nivel de clase) en las que las producciones irán almacenando información de la máquina, para, al reducir por la producción "program", esta información sea escrita en su respectivo archivo "Machines.java" con la sintaxis de java correcta. Además, al reducir la producción inicial también se escriben los archivos auxiliares de los que depende el correcto funcionamiento de la Máquina de Moore.

Algunas de las producciones que almacenan la información en dichas estructuras globales son las siguientes:

Listado 8: Producción transition.

```
1 l_transitions
2   ::= IDENT:i SEPARATOR l_evento:e FLECHA IDENT:c PUNTO_Y_COMA
    l_transitions {:
3     String [] s = new String [3];
4     s[0]=i;
5     s[1]=c;
6     s[2]=e;
7     transitions.add(s);:}
8   | ;
```

Se puede apreciar como se crea un Array de String, donde se guardan los identificadores (estado origen y estado destino) y el evento que debe ocurrir para realizar dicha transición, posteriormente este array se almacena en la lista global.

Listado 9: Producción states.

```
1 l_states
2   ::= IDENT:i SEPARATOR IDENT:c l_states_fact {:  String [] s = new
    String [2];
3     s[0]=i;
```

```
    s[1]=c;
    states.add(s);:}
  ;

```

De forma análoga ocurre en esta producción, el identificador del estado y su salida se almacenan en un array, que a su vez se almacena en la lista global.

Listado 10: Producción program.

```
program
  ::= environment:env l_output l_define l_automaton {:
  writeAuxiliar(env);
  try {
    writer = new PrintWriter("Machines.java", "UTF-8");
  }catch( Exception e){
    e.printStackTrace();
  }
  writer.println(String.format("package %s;\n\n",env));
  writer.println("public class Machines {\n");
  for(int i=0;i<automatons.size();i++){
    writer.println(String.format("\tpublic static IMooreMachine %s()
   {\n\t\tMooreMachine machine = new MooreMachine();\n\t\tmachine.
   setMachineName(\"%s\");",automatons.get(i),automatons.get(i)));
    for(int j=0;j<states.size();j++){
      String out = "";
      for(int k=0;k<outputs.size();k++){
        if(outputs.get(k)[0].equals(states.get(j)[1])){
          out=outputs.get(k)[1];
          out=out.replace("\n","\n\t\t");
          break;
        }
      }
      writer.println(String.format("\n\t\tState %s = new State(\"%s
   \");\n\t\t%s.setOutput(() -> {%s});\n\t\tmachine.addState(%s);",
   states.get(j)[0],states.get(j)[0],states.get(j)[0],out.substring
```

```
    (2,out.length()-2),states.get(j)[0]));
23       }
24       int tranCount = 0;
25       for(int j=0;j<transitions.size();j++){
26         tranCount+=1;
27         writer.println(String.format("\n\t\tTransition t%d = new
    Transition (%s,%s,\"%s\");\n\t\tmachine.addTransition(t%d);",
    tranCount,transitions.get(j)[0],transitions.get(j)[1],transitions.
    get(j)[2],tranCount));
28       }
29       writer.println(String.format("\n\t\tmachine.setInitialState(%s)
    ;",initial));
30       writer.println("\n\t\treturn machine;\n\t}");
31     }
32     writer.println("\n}");
33     writer.close();
34     :}
35     ;
```

La producción "program" es la más importante, ya que se puede observar como al reducir crea el archivo "Machines.java" que es el *core* de la Máquina de Moore, se puede apreciar como en cada String que escribimos en el fichero se formatea la información almacenada en las estructuras globales para que se cree el archivo correctamente.

4.3. Uso de las herramientas flex y cup

El uso de estas herramientas es muy sencillo, una vez escrito nuestro archivo .mooma para la creación de la Máquina de Moore deseada seguimos los siguientes pasos:

- Metemos el archivo mooma en el mismo directorio que los programas flex y cup.
- También es necesario introducir en el directorio el environment (el archivo .java que hereda de la interfaz que proporcionamos) que hayamos creado, y el directorio en cuestión debe tener el mismo nombre que el susodicho environment.
- Ejecutamos el comando para crear el analizador léxico:

```
1  jflex moomaLexer.flex
```

- Ejecutamos el comando para crear el analizador sintáctico y semántico:

```
java java_cup/Main moomaParser.cup
```

- Compilamos los analizadores:

```
javac analex.java parser.java Main.java sym.java
```

- Analizamos el archivo mooma:

```
java Main [Nombre].moo
```

- Compilamos todos los nuevos archivos generados:

```
javac *.java
```

- Ejecutamos nuestro programa principal para que empiece a funcionar nuestra Máquina de Moore.

5. Ejemplos de uso

El procesador de lenguaje que hemos construido genera una multitud de archivos Java para el modelado y simulación de máquinas genéricas de Moore, pero es necesario integrarlas en un programa que el usuario debe escribir. Además es necesario que implemente la interfaz *Environment* para que la simulación pueda ocurrir. Por eso, vamos a enseñar varios ejemplos de uso.

5.1. PredictionBuffer

En el dominio de los procesadores y el lenguaje máquina, muchas veces el procesador se va a encontrar con una instrucción condicional de salto. En estos casos, el procesador debe tomar el riesgo e intentar predecir si el salto se va a realizar o no. Esto es mejor que esperar a que la condición se resuelva, porque en caso de acertar, nos hemos ahorrado numerosos ciclos de reloj, aunque si se falla la predicción hemos perdido ciclos.

Una de las técnicas más báscias de predcción dinámica de salto es el uso del **buffer de predicción**. Se mantienen 2 bits con los resultados de los dos últimos saltos y su resultado. Esto se puede modelar mediante una máquina de Moore como se puede ver en la Figura 8.

En esta máquina tendríamos 4 estados, 2 posibles entradas (”Taken” y ”Not Taken”) y dos

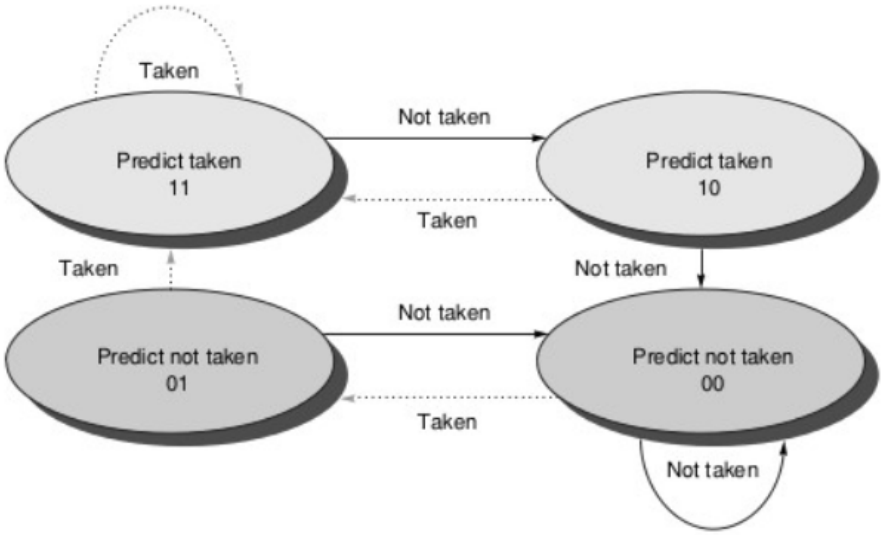

Figura 8: Autómata de Buffer de predicción.

posibles salidas (”Predicción de salto” y ”Predicción de no salto”). Para generar la máquina de Moore con nuestro procesador de lenguaje, el archivo ”.moo” quedaría de la siguiente manera:

Listado 11: PredictionBuffer.moo.

```
/*
Prediction Buffer para 2 bits
*/

environment := Env ;

code1 := {:
        Env.predictTaken()
        :};
code2 := {:
        Env.predictNotTaken()
        :};


define alfabeto1{
    in := 1, 2;
    out := code1, code2;
}

```

```
automaton automata1 (alfabeto1){
    states := q0|code2, q1|code2, q2|code1, q3|code1;
    initial := q2;
    transitions{
        q0|1 -> q0;
        q0|2 -> q1;
        q1|1 -> q0;
        q1|2 -> q2;
        q2|1 -> q3;
        q2|2 -> q2;
        q3|1 -> q0;
        q3|2 -> q2;
    }
}
```

El *Environment* para el ejemplo es el siguiente:

Listado 12: Environment para el autómata anterior.

```
public class Env implements IEnvironment {

        @Override
        public String translate(Object input) {
        if (input.toString().equals("Taken")){
            return "2";
        }
        else if (input.toString().equals("Not Taken")){
            return "1";
        }
        else{
            return "0";
        }
        }

    public static void predictTaken(){
        System.out.println("Predecimos salto");
```

```
18      }
19
20      public static void predictNotTaken(){
21          System.out.println("Predecimos no salto");
22      }
23
24  }
```

Aquí se puede ver el uso de la clase *Environment*. Mediante el método *translate* convertimos las entradas ”Taken” y ”Not Taken” en eventos. Además podemos implementar funciones que luego serán referenciadas en los códigos de salida del archivo ”PredictionBuffer.moo”. En este ejemplo solo muestran un mensaje por pantalla, pero el usuario puede hacer tantas funciones como quiera, y puede hacerlas tan complejas como quiera.

Una vez tenemos el *Environment* y el ”PredictionBuffer.moo”, solo tenemos que ejecutar el procesador de lenguaje, que nos generará varios archivos Java. Para que todo funcione, tenemos que escribir un programa que integre la máquina de Moore. Un programa básico que hemos escrito para este ejemplo es el siguiente:

Listado 13: Main para el autómata anterior.

```
1  import java.util.Scanner;
2  public class Main {
3      public static void main(String[] args){
4          Scanner sc = new Scanner(System.in);
5          MachineController mc = new MachineController(Machines.
       automata1());
6
7
8          System.out.print("Input: ");
9          String input = sc.nextLine();
10
11         while(!input.equals("")){
12             if (mc.addNewInput(input) != null){
13                 mc.getCurrentState().getOutput().accept(mc.
       getEnvironment());
```

```
14              }
15              else{
16                  System.out.println("Entrada no reconocida por el
    automata");
17              }
18              System.out.print("Input: ");
19              input = sc.nextLine();
20          }
21
22          sc.close();
23      }
24  }
```

Como se puede comprobar, solo es necesario crear un *MachineController*, pasándole como argumento la máquina de Moore que va a usar. El programa entra en un bucle para seguir tomando entradas para la máquina por la entrada estándar, y ejecuta el código de salida llamando al método *accept*, el cual necesita como argumento el entorno.

5.2. IA de enemigos en un videojuego

Los motores de juegos más populares, *Unity* y *Unreal Engine 4*, traen incorporado un módulo de IA que permite dotar a los enemigos del jugador de un comportamiento autónomo. Este comportamiento se modela a través de árboles de comportamiento. Un ejemplo de árbol de comportamiento se puede ver en la figura 9

Sin embargo, esta no es la única forma de modelar un comportamiento autónomo, estos comportamientos también se pueden modelar mediante una máquina de Moore.

En este ejemplo se ha modelado el comportamiento de un enemigo que ataca cuerpo a cuerpo y que prioriza el no ser atacado sobre el dañar a su enemigo (el jugador) 10.

La máquina, traducida al lenguaje *Mooma* está representada en el listado 15

Listado 14: GameIA.moo

```
1  /* IA de un enemigo en un videojuego */
2
```

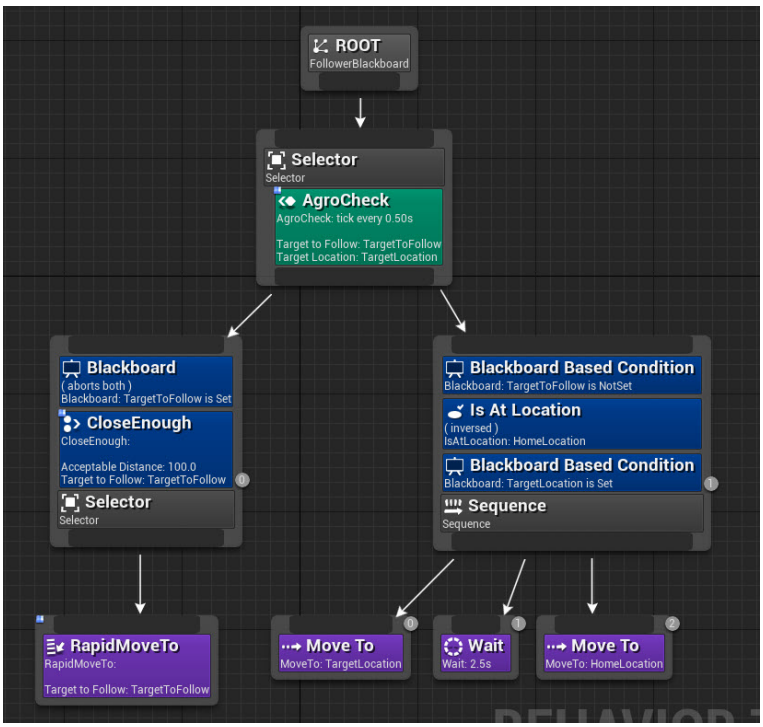

Figura 9: Ejemplo de Árbol de Comportamiento en *Unreal Engine 4*

```
environment := GameAIEnvironment;

code1 := {: env.notify("Huir") :};
code2 := {: env.notify("Parado") :};
code3 := {: env.notify("Andar") :};
code4 := {: env.notify("Atacar") :};


define EnemyAlphabet {
    in := 1, 2, 5, 10; /* Mirando, No Mirando, Lejos, Cerca Atacar */
    out := code1, code2, code3, code4;
}

automaton MeleeEnemy (EnemyAlphabet){
    states := Huir|code1, Parado|code2, Andar|code3, Atacar|code4;
    initial := Parado;
    transitions{
        Parado|1 -> Huir;
        Parado|2 -> Andar; Parado|5 -> Andar;
        Huir|5 -> Parado;
```

```
23          Huir|1 -> Huir;
24          Huir|2 -> Andar;
25          Andar|2 -> Andar; Andar|5 -> Andar;
26          Andar|1 -> Huir;
27          Andar|10 -> Atacar;
28          Atacar|1 -> Huir;
29          Atacar|10 -> Atacar;
30      }
31  }
```

Como podemos ver en la línea 12 del listado 15 se ha tenido que traducir las entradas del autómata por eventos numéricos.

En el archivo se menciona la clase *GameAIEnvironment*, el entorno de la máquina de Moore, representada en el listado **??**

Listado 15: GameIA.moo

```
1  public class GameAIEnvironment implements IEnvironment {
2      private HashMap<String, List<Runnable>> listeners;
3
4      public GameAIEnvironment() {
5          listeners = new HashMap<>();
6      }
7
8      public void notify(String state) {
9          if (listeners.containsKey(state)) {
10             listeners.get(state).forEach((Runnable action) -> action.
    run());
11         }
12     }
13
14     public void addListener(String state, Runnable action) {
15         if (!listeners.containsKey(state))
16             listeners.put(state, new LinkedList<>());
17
```

```
18          listeners.get(state).add(action);
19      }
20
21      public String translate(Object input) {
22          return input.toString();
23      }
24
25      public String translate(Enemy me, Character ch) {
26          double angle = WorldCanvas.angleBetweenPoints(ch.getX(), me.
        getX(), ch.getY(), me.getY()) * -1;
27          double distance = WorldCanvas.euclideanDistance(ch.getX(), me
        .getX(), ch.getY(), me.getY());
28
29          if (distance >= Character.VISION_LENGTH)
30              return "5";
31          else if ((ch.getAngle() - Character.VISION_CONE / 2) <= angle
         && (ch.getAngle() + Character.VISION_CONE / 2) >= angle)
32              return "1";
33          else if (distance <= Enemy.ATTACK_RANGE)
34              return "10";
35          else
36              return "2";
37      }
38  }
```

Como se puede ver, este entorno es mucho más complejo que el del búfer de prediccion (sección 5.1) ya que es necesario que el entorno interactúe con la interfaz. Para realizar esta interacción se ha optado por un diseño basado en el patrón *Observer* aprovechando las características funcionales añadidas en *Java 8*.

También es necesario que el entorno calcule distancias y ángulos para comprobar la condicion de *mirando*, estas operaciones se delegan a la clase *WorldCanvas* que es la encargada de la interfaz.

El código que ejecuta cada estado recibe una instancia de la clase *IEnvironment*, gracias a esto

es posible tener varias máquinas de Moore funcionando a la vez, ya que no se usan variables ni métodos estáticos.

Cada máquina de Moore tiene asociada una clase *Pawn* que es la que se encarga de manejar el personaje dentro del mundo y interactuar con la máquina.

El diagrama de clases del ejemplo se puede ver en la figura 11 y el ejemplo de ejecución del ejemplo se puede ver en la figura 12.

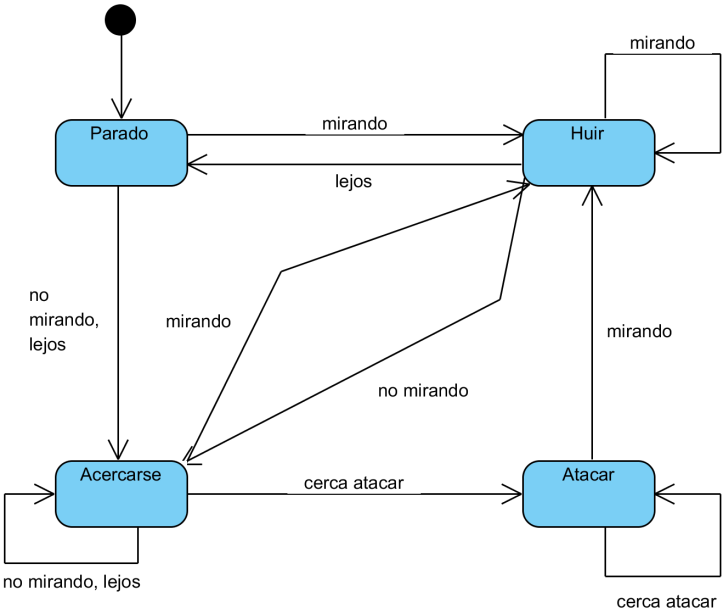

Figura 10: Máquina de estado del ejemplo GameIA

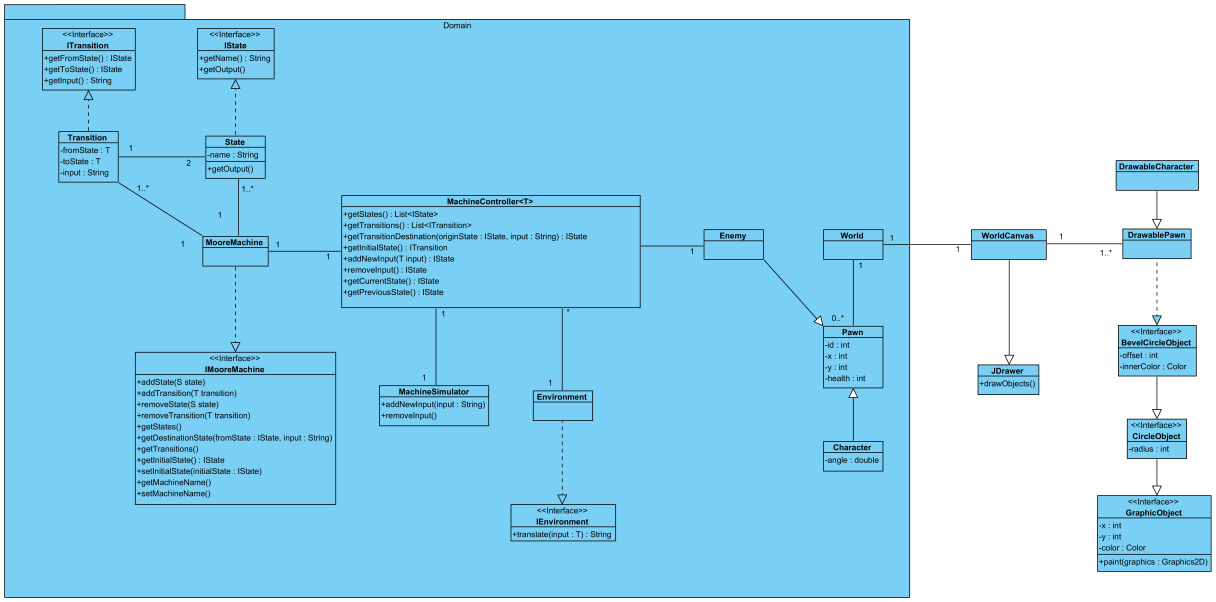

Figura 11: Diagrama de clases del ejemplo GameIA

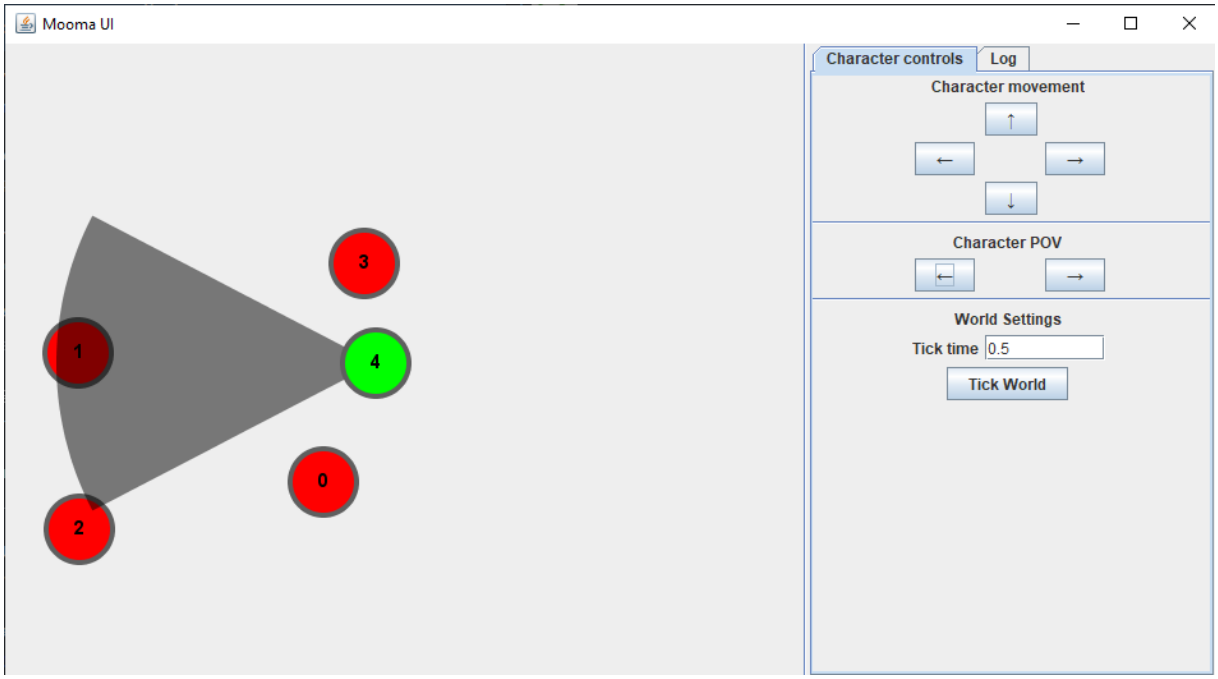

Figura 12: Ejemplo de ejecución del ejemplo GameIA