UNIVERSIDAD COMPLUTENSE DE MADRID
Facultad de Informática

Sistemas Informáticos **09 / 10**

XLOP 2.0

Modularizado de Gramáticas

XML Language-Oriented Processing

Autores

Fco. Javier Dones Piera
José Andrés Arteaga García
Juan Manuel Digón Vara

Director

Antonio Sarasa Cabezuelo

Proyecto de Sistemas Informáticos

2009 / 2010

XLOP 2.0

Modularizado de Gramáticas

(XML Language-Oriented Processing)

Autores:

Fco. Javier Dones Piera

José Andrés Arteaga García

Juan Manuel Digón Vara

Director:

Antonio Sarasa Cabezuelo

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

Nos gustaría agradecer toda la ayuda y el trabajo prestado durante este año principalmente a Antonio Sarasa, director del proyecto, por su total disponibilidad e implicación en el proyecto. A Brian por ser en ocasiones un particular beta tester y sobre todo a nuestros familiares y amigos.

-Gracias-

Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código, la documentación y/o el prototipo desarrollado.

Fco. Javier Dones Piera José Andrés Arteaga García Juan Manuel Digón Vara

Resumen

En este trabajo de Sistemas Informáticos se ha perfeccionado la herramienta XLOP, el cual es un entorno para el procesamiento de documentos XML mediante gramáticas de atributos. Esta ampliación se ha centrado en varios aspectos, primeramente se ha añadido una nueva funcionalidad, de manera que admita como entrada una gramática separada en módulos independientes, permitiendo un diseño de aplicaciones más claro y modular. Posteriormente, hemos sustituido y añadido una serie de algoritmos, que han mejorado la potencia y la eficiencia en la construcción de aplicaciones de XLOP. De estos algoritmos el más importante ha sido el algoritmo de marcado, el cual procesa la gramática e indica en que lugares se pueden añadir ciertos elementos llamados marcadores. Estos marcadores son nuevos no terminales, definidos mediante producciones vacías, los cuales, permiten albergar atributos heredados y otras instrucciones que permiten adelantar el cálculo de atributos semánticos.

Con el objetivo de mostrar el potencial de la nueva versión, se ha construido un juego basado en el popular juego de mesa Trivial. La aplicación se denomina XTrivial y permite generar jugos de trivial a través de su descripción como documento XML.

Palabras clave: XML, Gramáticas de atributos, Procesador de Lenguaje, Modularizado de gramáticas, Autómata LALR, Algoritmo de marcado, Trivial.

Abstract

In this work we have made some improvements in the tool called XLOP, which is an environment for processing XML documents through attribute grammars. Firs of all the improvements, we have added a new functionality to the tool that allows the input of an attribute grammar separated in several independent modules. Now the design of applications with XLOP is clearer and more modular. Later, we have substituted an added some algorithms that have improved the power and efficiency of those applications developed by XLOP. The most important of these algorithms was the so called markup algorithm. This one process the grammar and point several places where is possible to assign a mark. Marks are new non terminal elements defined by empty productions which allow the storage of inherited attributes and other instructions that optimize the semantic attributes calc.

With the purpose of showing the potential of this XLOP new version we have developed an application that represents a game based on the popular board game Trivial. This application is called XTrivial and allows the generation of several Trivial games through a description of the game provided by a XML document.

Keywords: XML, Attribute grammars, Language processors, Grammar modularization, LALR automaton, Markup algorithm, Trivial.

Índice

1. Prólogo 18

1.1 Introducción 18

1.2 Objetivos del trabajo 19

1.3 Estructura del documento 21

2. El entorno XLOP 22

2.1 Introducción y nuevos aspectos 22

2.2 Construcción de una aplicación 22

2.2.1 Documento XML y DTD 23

2.2.2 Gramática XLOP 25

2.2.3 Clase semántica 27

2.2.4 Ejecución 27

3. El algoritmo de marcado en XLOP 32

3.1 Introducción 32

3.2 Implementación del marcado en XLOP 33

3.2.1 Grafo R_λ 33

3.2.2 Fase 1: Construcción del autómata LALR(1) 35

3.2.2.1 Obtención de la estructura del grafo 35

3.2.2.2 Obtención de los símbolos de preanálisis 38

3.2.3 Fase 2: Detección de posiciones prohibidas 39

3.2.3.1 Eliminación de caminos cíclicos y entre nodos del mismo nivel 40

3.2.3.2 Eliminación de posiciones precedidas por un terminal 42

3.2.4 Fase 3: Agrupación 44

3.2.5 Fase 4: Eliminación de anomalías de fusión 46

3.2.6 Fase 5: Asignación de marcadores 46

3.3 Ejemplo ilustrativo 48

3.3.1 Fase 1 48

3.3.2 Fase 2 49

3.3.3 Fase 3 51

3.3.4 Fase 4 51

3.3.5 Fase 5 51

4. Modularización de gramáticas en XLOP 53
4.1. Introducción 53
4.2. Modularizado de gramáticas 53
4.2.1. Concepto 53
4.2.2. Ejemplo 54
4.3. Implementación de la modularización en XLOP 55
4.3.1. Estructura del sistema del espacio de nombres 55
4.3.1.1. Instrucción namespace 55
4.3.1.1.1. Ejemplo namespace 56
4.3.1.2. Instrucción qualify 56
4.3.1.2.1. Ejemplo qualify 58
4.3.1.3. Cualificación directa 59
4.3.2. Adaptación de la interfaz al modularizado de gramáticas 60
4.4. Ejemplo ilustrativo 61
5. Ejemplo de aplicación de la nueva versión de XLOP 67
5.1 Introducción 67
5.2 Algoritmo de no alcanzables 67
5.3 Algoritmo de no productivos 68
5.4 Algoritmo de primeros 69
6. Ejemplo de aplicación de la nueva versión de XLOP: XTrivial 71
6.1. Trivial, el juego original 71
6.2. Trivial en el entorno XLOP 71
6.3. Documento XML y la DTD 72
6.4. Gramática XLOP para XTrivial 77
6.4.1. Estilo de la clase semántica 77
6.4.2. Estructura de la gramática 78
6.4.2.1. Gramática principal 78
6.4.2.2. Gramática Features 79
6.4.2.3. Gramática TextQuestion 80
6.4.2.4. Gramática OptionQuestion 82
6.4.2.5. Gramática OptionImageQuestions 83

6.4.3. La clase semántica 85
6.4.3.1. Atributos 85
6.4.3.2. Creación del juego 86
6.4.3.3. Inicio del Juego 86
6.4.3.4. Nueva característica 86
6.4.3.5. Tratamiento de preguntas 87
6.4.3.6. Pregunta QuestionText 87
6.4.3.7. Pregunta QuestionOptions 88
6.4.3.8. Pregunta QuestionOptionImage 90
6.5. Implementación de XTrivial 91
6.6. Generación de la aplicación con XLOP 92
6.7. Una partida con XTrivial 94
7. Conclusiones y trabajo futuro 98
7.1. Conclusiones 98
7.2. Trabajo futuro 99

Capítulo 1

Prólogo

1.1 Introducción

XML (eXtensible Markup Language) es una especificación estandarizada, propuesta por el World Wide Web Consortium (W3C) para el marcado de documentos. Define una sintaxis genérica para el marcado de información a través de etiquetas. Es utilizado en gran cantidad de aplicaciones para la representación e intercambio de información, quedando ésta estructurada de una determinada manera.

XML es un *lenguaje de metamarcado*[Harold et al. 2001], entendiéndolo de manera que no existen un conjunto fijo de etiquetas y elementos al servicio del desarrollador. Así pues, permite al desarrollador crear los elementos que necesiten según sea preciso. Esto quiere decir que puede ser adaptado y extendido según la necesidad de la tarea acometida.

A pesar de esta flexibilidad que permite XML, es más estricto en otros aspectos. Las especificaciones XML definen una gramática para documentos XML que indica donde pueden alojadas las etiquetas, qué nombres de elementos son válidos, cómo deben estar ligados los atributos a los elementos, etc. Aquellos documentos que satisfagan esta gramática se dice que están *bien formados*, por el contrario que no las cumplan no están permitidos y cualquier procesador rechazara como entrada un archivo que no este bien formado.

El marcado en un documento XML, describe la estructura del documento y permite ver qué elementos están asociados con otros, además de poder describir la semántica del mismo. El marcado permitido, en una determinada aplicación XML puede estar documentado en un esquema (*schema*). Así cada instancia de un documento puede ser comparada con el esquema y si coincide se dice que es un archivo *válido*. Hay diferentes lenguajes para definir esquemas, y cada uno de ellos con diferentes niveles de expresividad. El más soportado y el único definido por la especificación XML en si misma es DTD (Document Type Definition). Un DTD lista el marcado legal de un documento y especifica donde y como debe ser incluido. Hay que indicar que los DTD son opcionales dentro de los documentos XML.

A pesar de todo, hay que recordar que XML es solamente un lenguaje de marcado, es decir, no indica ninguna manera de procesar el documento para alguna aplicación. Por eso el marcado XML se suele decir que es *descriptivo*, está orientado a plasmar la estructura lógica de los documentos, pero no las posibles formas de procesar dichos documentos [Coombs et al. 1987].

En este aspecto, el del procesamiento de los documentos XML, es donde se enmarca el propósito de XLOP. La aplicación XLOP (*XML Language-Oriented Processing*) es un entorno que utiliza gramáticas de atributos para describir el procesamiento de ficheros XML con un determinado vocabulario.

Las gramáticas de atributos permiten tanto describir la estructura sintáctica de los documentos XML, como las acciones a realizar para su procesamiento. Así pues, se puede entender la implementación de un procesador XML como el procesamiento de un determinado lenguaje de marcado especifico para un determinado tipo de documento. De esta manera, tomando como entrada una gramática de atributos, es posible generar automáticamente traductores para determinados documentos XML, entendiendo estos traductores como *procesadores de lenguaje* y el proceso de desarrollo como la construcción y mantenimiento de estos procesadores.

1.2 Objetivos del trabajo

Continuando con el trabajo iniciado en el pasado que dio origen a la herramienta XLOP, nuestra principal línea de trabajo este año ha sido la ampliación de las funcionalidades de XLOP y la mejora interna del mismo dando lugar a una versión mas refinada y eficiente. De entre estas mejoras destacan dos principalmente:

- La primera de ellas consta de una ampliación de la funcionalidad de la herramienta permitiendo la posibilidad de recibir como entrada una gramática que haya sido dividida en fragmentos más pequeños. Si entendemos cada fragmento de gramática como un subproceso encargado de realizar una determinada tarea dentro del procesamiento de un documento XML, entonces esta nueva funcionalidad permite la especificación y el diseño de aplicaciones de una manera modular, en la que cada modulo pueda realizar tareas independientes.

- La segunda ampliación ha consistido en la mejora e integración de ciertos algoritmos. El más importante y complejo de ellos, fue la adaptación del algoritmo de marcado existente, puesto que no funcionaba correctamente para ciertas gramáticas. Esta mejora ha sido fruto del trabajo de investigación desarrollado dentro del Departamento de Sistemas y Computación de la Facultad de Informática de la Universidad Complutense de Madrid.

  Además del algoritmo de marcado también han sido integrados unos algoritmos de limpieza de gramáticas. Estos algoritmos detectan las reglas no generadoras y no alcanzables de una determinada gramática, eliminándolas de ésta.

Adicionalmente y para mostrar la potencia de la herramienta XLOP se ha desarrollado un ejemplo complejo de aplicación. Para ello se ha desarrollado una aplicación llamada XTrivial que genera juegos basados en trivial a partir de documentos XML. Para resaltar la utilidad del modulado de gramáticas se han introducido 4 módulos dentro de esta aplicación que son:

1. Un módulo llamado **Features,** que procesa la parte del documento en la que se establecen mensajes de error en determinados puntos del programa.
2. Un módulo llamado **TextQuestions**, encargado de procesar aquellas preguntas cuyas respuestas solo pueden ser respondidas mediante texto plano.
3. Un módulo llamado **OptionQuestions**, el cual procesa aquellas preguntas que consisten en elegir una de entre varias opciones para responderla.
4. Un módulo llamado **ImageQuestions,** en el que se procesan aquellas preguntas en las que están involucradas la representación de imágenes.

Cabe mencionar el hecho de que para la realización de este proyecto se han desarrollado conocimientos adquiridos durante los estudios de Ingeniería Informática. Especialmente se han adquirido conocimientos sobre tecnologías XML así como la aplicación de conocimientos para la utilización de herramientas utilizadas para la construcción de procesadores de lenguajes.

1.3 Estructura del documento

La presente memoria se organiza de la siguiente manera:

- En el Capítulo 2 se realiza una breve presentación del entorno XLOP en la que explicaremos los nuevos detalles de la interfaz gráfica de la aplicación así como aspectos a nivel usuario de su utilización.

- En el Capítulo 3 se expondrá la nueva implementación del algoritmo de marcado de XLOP. Expondremos las necesidades que motivaron la implementación de este nuevo algoritmo, expondremos algunas estructuras necesarias para el correcto funcionamiento del mismo y analizaremos con detalle cada una de las fases que lo componen.

- En el Capítulo 4 se describe el proceso de modularización de las gramáticas en XLOP. En este capítulo expondremos los conceptos de espacio de nombres y todos los cambios necesarios para la lectura de gramáticas en varios ficheros y su posterior reconstrucción.

- En el Capítulo 5 se explicarán las motivaciones para la introducción de los algoritmos de limpieza, así como se estructura e implementación.

- En el Capítulo 6 detallaremos el desarrollo de la aplicación XTrivial. Detallaremos la construcción y todos los detalles de la especificación de la aplicación mediante el entorno XLOP.

- En el Capítulo 7 haremos un balance del proyecto y comentaremos unas posibles líneas de desarrollo que puede seguir XLOP en el futuro.

Capítulo 2

El entorno XLOP

2.1 Introducción y nuevos aspectos

En este capitulo vamos a exponer el nuevo entorno XLOP y a través de un sencillo ejemplo, señalaremos los aspectos mas importantes a la hora de construir una aplicación. De entre estos aspectos, haremos hincapié en el documento XML y su correspondiente DTD, la gramática XLOP y la clase semántica.

Las principales novedades en la interfaz gráfica de XLOP son dos, en primer lugar la aplicación debe ser capaz de leer varios módulos de gramáticas, permitiendo la admisión y el procesamiento de varios archivos. Así, se ha incluido un campo de texto en el cual se debe ir añadiendo todos los archivos, que representen la gramática modularizada. La segunda de estas novedades es la visualización gráfica del autómata LALR, que se podrá realizar una vez se haya procesado la gramática correctamente y se haya construido la aplicación.

2.2 Construcción de una aplicación

Para ilustrar la forma de uso del entorno XLOP construiremos, paso a paso, una sencilla aplicación que llamaremos “Messenger”. La finalidad de esta aplicación será procesar un documento XML que represente una conversación entre varios individuos, de manera que la salida de la aplicación sea un texto con la propia conversación mostrada como si de un servicio de mensajería instantánea se tratara.

2.2.1 Documento XML y DTD

A la hora de diseñar la aplicación, es importante tener bien definida tanto la estructura de los documentos XML que vamos a tratar, así como del procesamiento que vamos a realizar en estos documentos. Para ello se caracterizan estos archivos con una DTD. En nuestro ejemplo la DTD utilizada se muestra en la figura 2.2.1.

```
1  <?xml version='1.0' encoding='UTF-8'?>
2
3  <!ELEMENT Conver (Info)+>
4
5  <!ELEMENT Info (Men|Hora|De)*>
6
7  <!ELEMENT De (#PCDATA)>
8
9  <!ELEMENT Hora (#PCDATA)>
10
11 <!ELEMENT Men (#PCDATA)>
```

Figura 2.2.1. Definición de la DTD de la aplicación de ejemplo *Messenger*.

Aquellos documentos que siguen esta estructura, representarán conversaciones que estén compuestas por varios grupos de etiquetas *Info*. Estando estos definidos como si fueran una unidad de mensaje, donde aparecen el emisor, representado por la etiqueta *De*, la hora en la que se produjo el mensaje en la etiqueta *Hora* y el propio texto del mensaje representado por la etiqueta *Men*.

Nuestra aplicación efectuará un procesamiento secuencial, de principio a fin, del documento almacenando cada contenido de *Info* en una cadena de caracteres y mostrándola al final.

A pesar del carácter opcional que pueden tener los DTD, éstos no sólo tienen valor a la hora de definir la estructura de documentos XML, sino que también existen ciertos aspectos más técnicos que aconsejan su uso. Por ejemplo, para permitir la validación de los propios documentos XML y así se fuerza a trabajar con documentos, que sean totalmente correctos y útiles para la aplicación. También es importante mencionar que sin la especificación de una DTD, no es posible introducir espacios en blanco,

tabulaciones ni saltos de línea entre las etiquetas de un documento XML. Sin una DTD estos elementos no son ignorados a la hora del procesamiento, sino que son interpretados como #pcdata.

La figura 2.2.2 muestra una instancia XML valida para nuestra aplicación, que representa una conversación entre tres personas. Y la figura 2.2.3 representa la salida mostrada por la aplicación tras procesar el documento.

```
1  <!DOCTYPE Conver SYSTEM "Messenger.dtd">
2
3  <Conver>
4      <Info>
5          <De>Fran</De>
6          <Hora>16:00</Hora>
7          <Men>Ey chicos!!!</Men>
8      </Info>
9      <Info>
10         <De>Juanma</De>
11         <Hora>16:05</Hora>
12         <Men>A que hora es el partido?</Men>
13     </Info>
14     <Info>
15         <De>Josean</De>
16         <Hora>16:06</Hora>
17         <Men>Jugamos a las 8</Men>
18     </Info>
19     <Info>
20         <De>Juanma</De>
21         <Hora>16:08</Hora>
22         <Men>Venid preparados!</Men>
23     </Info>
24     <Info>
25         <De>Fran</De>
26         <Hora>16:10</Hora>
27         <Men>Vale</Men>
28     </Info>
29     <Info>
30         <De>Josean</De>
31         <Hora>16:15</Hora>
32         <Men>Nos vemos en la pista</Men>
33     </Info>
34 </Conver>
```

Figura 2.2.2. Ejemplo de documento XML para el ejemplo *Messenger*.

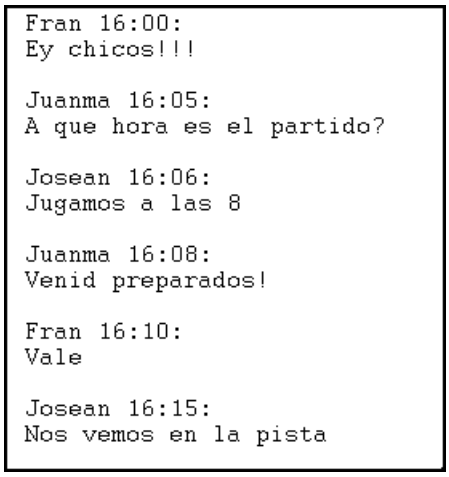

Figura 2.2.3. Ejemplo de ejecución
de la aplicación Messenger.

2.2.2 Gramática XLOP

El siguiente paso en la construcción de una aplicación, es la definición de la gramática de atributos que especifique el procesamiento. Esta gramática, estará almacenada en un documento con extensión .xlop que puede ser editable a través de cualquier editor de texto normal como podría ser el Bloc de Notas de Windows.

Debido a que en esta versión, se permite la posibilidad de que la gramática principal pueda estar separada en varios documentos, han de resolverse varios problemas. El primero de ellos, es como desde una determinada gramática, hacer referencia a un elemento sintáctico de otra gramática. Para ello, se han incluido el concepto de *espacio de nombres* que se detalla más en profundidad en el Capítulo 3.

El segundo problema se haya, en decidir cual es ahora el axioma de la gramática. Para ello el usuario, como diseñador de la aplicación que es, simplemente dejará seleccionado el nombre del archivo donde este el axioma y dentro de este archivo, el axioma será la primera regla de producción.

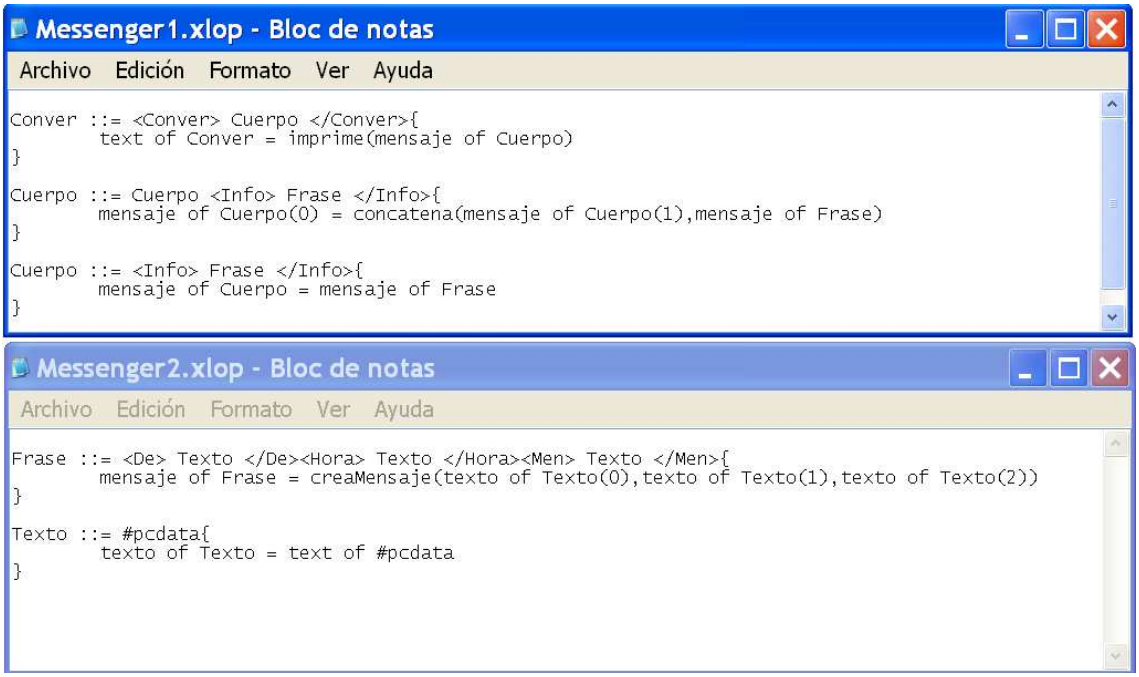

Figura 2.2.4. Gramática XLOP del ejemplo Messenger.

La gramática XLOP para nuestro ejemplo estará dividida en dos módulos, el primero se encargara de especificar el procesamiento del documento hasta encontrar una etiqueta *<Info>* y el segundo módulo se encargara de especificar el procesamiento de la información que se encuentre dentro de las etiquetas *<Info>*. El primer módulo, representado en la figura 2.2.4 por el archivo Messenger1.xlop, es el modulo que contendrá el axioma que, en este caso, será *Conver*. Como hemos mencionado antes, para que esto funcione al ejecutar XLOP debemos de dejar seleccionado el archivo Messenger1.xlop tal y como se muestra en la figura 2.2.6.

La única ecuación de la regla *Conver* permite una vez procesadas todas las reglas *Cuerpo* imprimir por pantalla a través del método *imprime (texto)* el texto almacenado de la conversación. Las reglas *Cuerpo* permiten procesar una o más mensajes de la conversación que esta encerrado entre las etiquetas *<Info>*.

La primera regla *Cuerpo*, a través del método *concatena (texto, texto)*, permiten concatenar la información contenida entre las etiquetas *<Info>* con los fragmentos de conversación ya almacenados. La segunda regla *Cuerpo* permite que se termine el procesamiento de este conjunto de etiquetas *<Info>* y se almacene una cadena de texto con su información en la variable *mensaje*.

La regla *Frase,* interpreta el contenido de las etiquetas *<De>*, *<Hora>* y *<Men>* y lo fusiona en una cadena de texto a través de la función *creaMensaje (texto, texto, texto)*. La regla *Texto* se ha introducido para representar los fragmentos de texto XML.

2.2.3 Clase semántica

La clase semántica, es una clase .java donde se implementan todas las funciones semánticas que han sido utilizadas en la gramática XLOP. Si fuera necesario importar una serie de clases pertenecientes a una librería que no pertenezca a la biblioteca estándar de Java, deberá de agregarse la librería en la sección habilitada para ello en la interfaz.

La clase semántica que representa nuestro ejemplo con las funciones semánticas *imprime (texto)*, *concatena (texto, texto)* y *creaMensaje (texto, texto, texto)* se muestra a continuación.

```
1   public class Messenger{
2       public Messenger(){
3       }
4       public String creaMensaje (String s1, String s2, String s3){
5           return s1+" "+s2+":\n"+s3+"\n";
6       }
7       public String concatena (String s1, String s2){
8           return s1+"\n"+s2;
9       }
10      public void imprime(String s1){
11          System.out.println(s1);
12      }
13  }
```

Figura 2.2.5. Clase semántica de la aplicación Messenger.

2.2.4 Ejecución

Una vez completados los pasos anteriores lanzaremos la aplicación XLOP y pasaremos a configurarla adecuadamente mediantes los siguientes pasos.

1. Primeramente, daremos un nombre a la aplicación a través del cuadro de texto *Nombre de la aplicación*. En nuestro caso el nombre será “Messenger”.

2. Después, ubicaremos los archivos necesarios que forman parte de la lógica especifica de la aplicación en la lista *Archivos de usuario.* Estos archivos principalmente son la clase semántica y la DTD de los archivos que procesa la aplicación generada en caso de depender de un DTD. También pueden introducirse archivos, que sea necesario copiarlos en la carpeta destino de la aplicación generada y que sean relevantes para el funcionamiento de la misma, tales como imágenes, archivos html, etc. En nuestro caso, a través del botón *Añadir archivo o carpeta,* añadimos la clase semántica “Messenger.java” y la DTD “Messenger.dtd".

3. A continuación en la lista *Librerías de usuario* se introducen las rutas de aquellas librerías necesarias para la ejecución de la aplicación. Por defecto XLOP añadirá la ruta de la librería *java-cup*, necesaria para la construcción de la aplicación. En nuestro ejemplo no es necesario añadir ninguna librería más.

4. En el cuadro de texto *Carpeta de salida* a través del botón *Explorar* elegiremos la ruta de la carpeta destino donde se almacenará la aplicación generada.

5. Adicionalmente se permite seleccionar varias opciones que completen el procesamiento. Marcando la casilla *Desactivar: distribución de atributos y ecuaciones,* estamos indicando que XLOP no realice ninguna optimización sobre la gramática especificada. Marcando la casilla *Generar trazas de depuración,* habilitamos el muestreo de mensajes informativos sobre la evaluación de las reglas de la gramática.

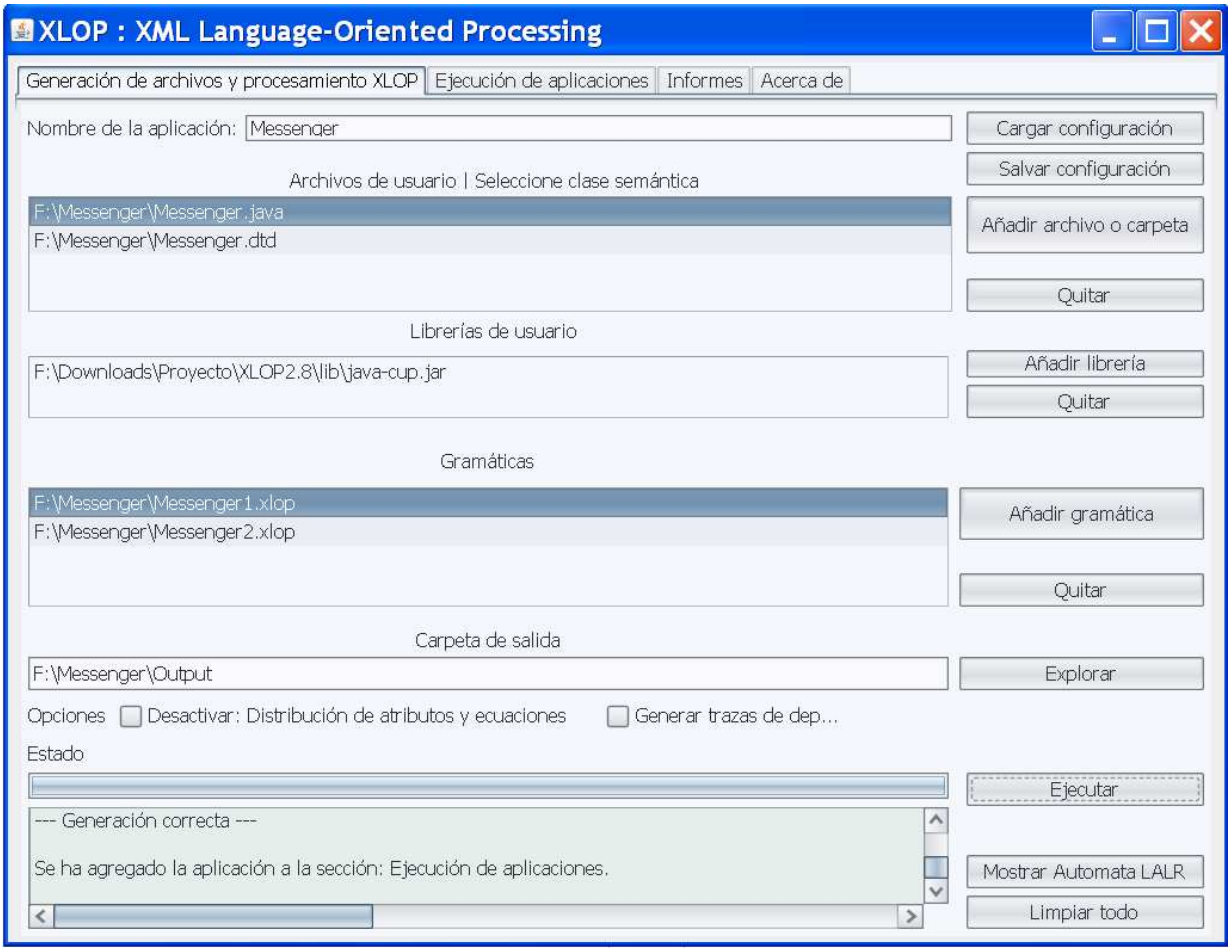

Figura 2.2.6. Interfaz de XLOP tras la generación de la aplicación Messenger.

Finalizados estos pasos, debemos asegurarnos de que dejemos seleccionada la clase semántica y aquella gramática que contendrá el axioma de la gramática final. Hecho esto, la ejecución comienza pulsando el botón *Ejecutar.* En el cuadro de texto *Estado* se nos informará del estado de la ejecución, en caso de ocurrir se mostrará información sobre algún tipo de error en la configuración de la aplicación en la propia gramática proporcionada. En nuestro ejemplo se debería llegar un estado como el que muestra la figura2.2.6 donde se nos indica que la aplicación se ha generado correctamente.

Si la generación de la aplicación ha sido satisfactoria, aparecerá un nuevo botón llamado *Autómata LALR* que lanzará la aplicación VCG Tool para mostrar gráficamente el autómata LALR generado. A través de esta aplicación y con ayuda de las flechas del teclado podemos navegar por el grafo y para cada estado ver que elementos LR (0) lo forman y sus símbolos de preanálisis. En la figura 2.2.7 se muestra un fragmento del autómata resultado del procesamiento de nuestro ejemplo.

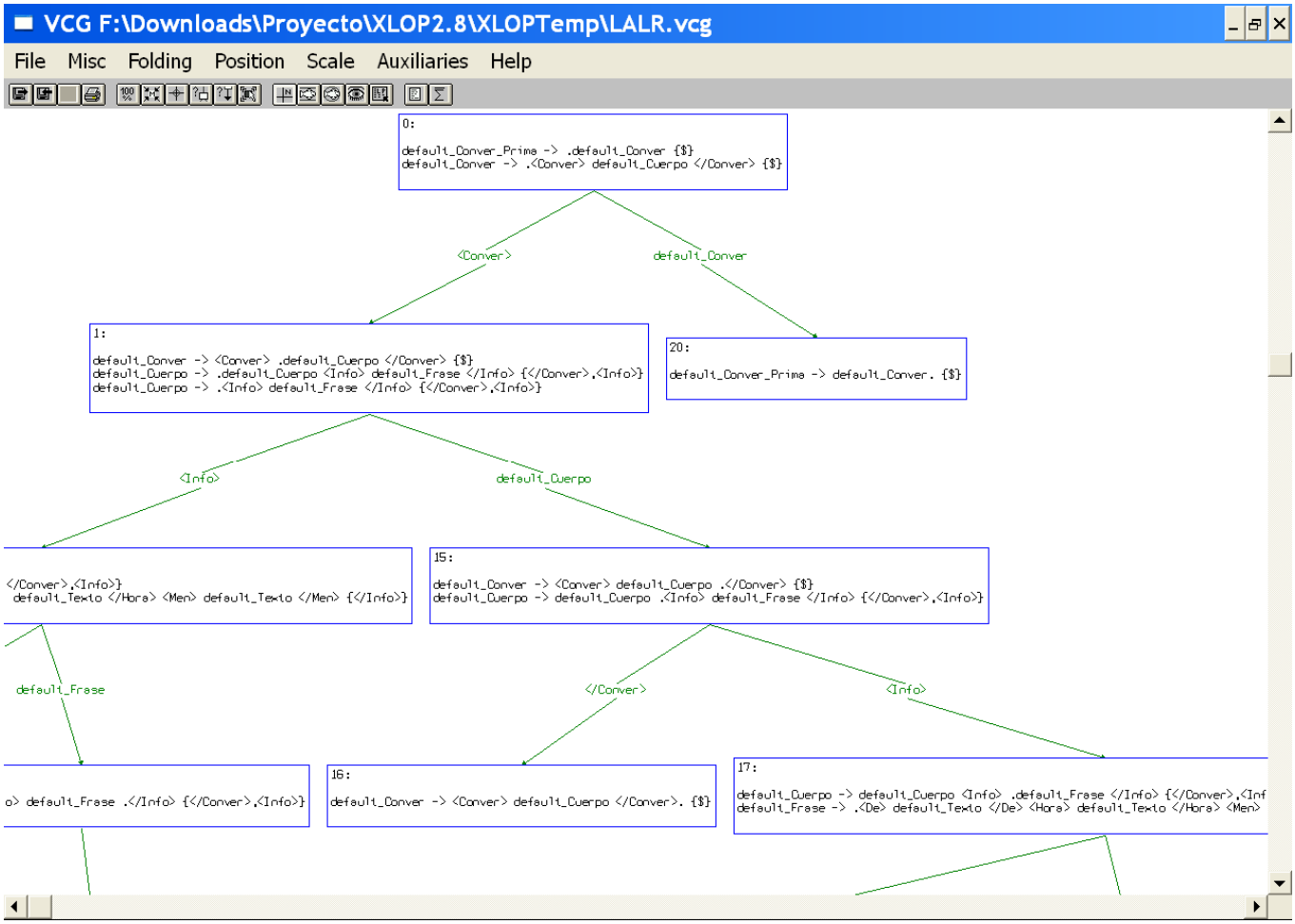

Figura 2.2.7. Fragmento del autómata LALR obtenido tras construir la aplicación Messenger.

Para ejecutar esta aplicación iremos a la pestaña *Ejecución de aplicaciones* y en la lista *Seleccione aplicación* deberá aparecer la que hemos configurado anteriormente. La dejamos seleccionada y a través del botón *Añadir XML a la lista* seleccionaremos el archivo XML que queramos procesar. En nuestro caso este documento se llama “Messenger.xml”. A continuación, mediante el botón ejecutar lanzaremos la ejecución de esta aplicación obteniendo la salida esperada.

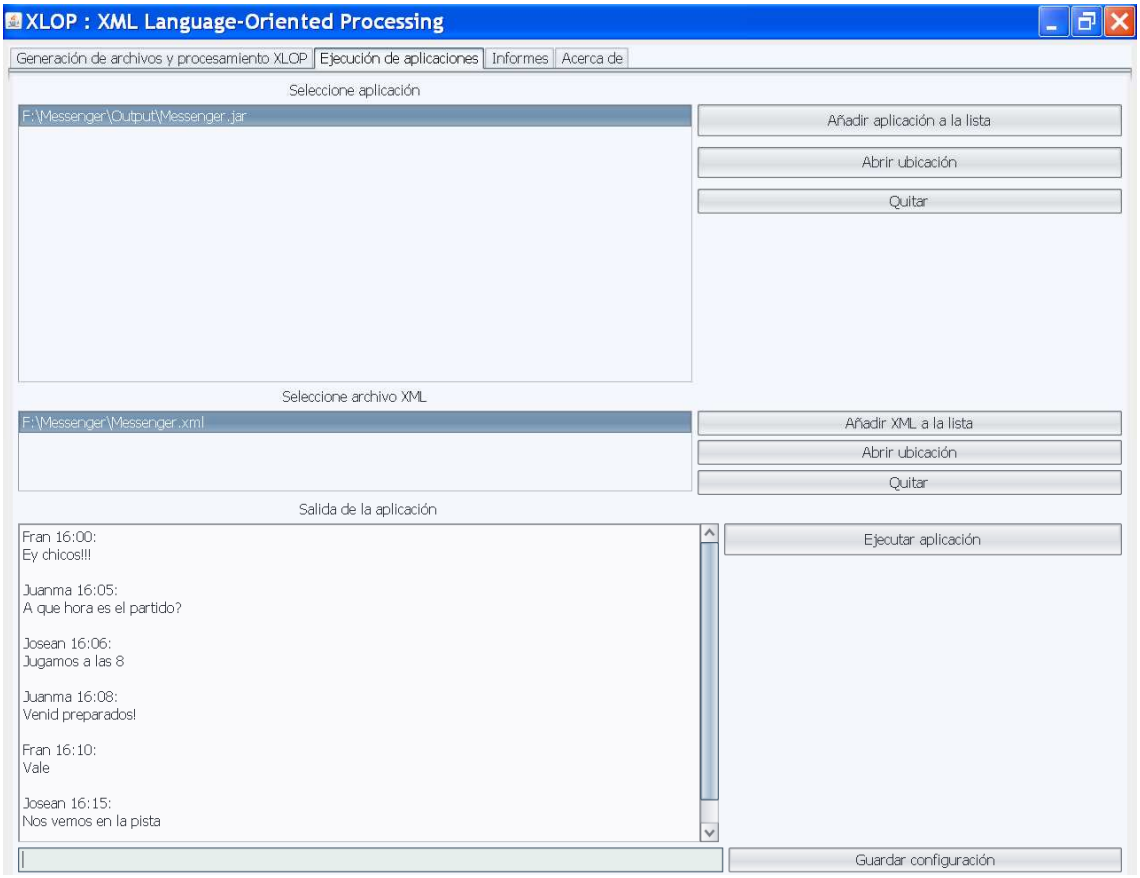

Figura 2.2.8. Salida de la aplicación Messenger.

Capítulo 3

El algoritmo de marcado en XLOP

3.1 Introducción

Tras la creación del modelo de objetos XLOP, se introduce una optimización del mismo insertando marcadores para la obtención del modelo XLOP marcado. Estos marcadores son unos nuevos no terminales definidos mediante producciones vacías que pueden albergar atributos heredados y otras instrucciones que permiten adelantar el cálculo de atributos semánticos.

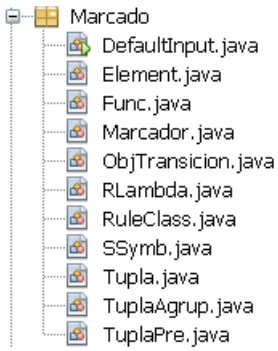

Imagen 3.1. Representa el nuevo paquete de clases resultado de la nueva implementación

La figura 1 muestras las clases necesarias en el proceso realizado por el modulo marcador. Con respecto a la anterior versión se ha de mencionar que se ha suprimido la estructura de grafo con la que se definía el autómata LALR, ahora la información sobre el autómata se almacena en la clase Marcador.java en una serie de tablas que representan los elementos LR (0) de cada estado, los elementos nucleares de cada estado y las transiciones entre estados.

Además de las nuevas clases también se ha integrado la herramienta VCG tool, desarrollada por la Saarland University, y cuyo objetivo es la representación gráfica de estructuras arbóreas o grafos.

3.2 Implementación del marcado en XLOP

El algoritmo esta íntegramente implementado en la clase Marcador.java y consta de cinco fases que explicaremos a continuación. Aunque previamente introduciremos un concepto que utilizaremos a lo largo todo el algoritmo. Se trata del grafo R_λ que nos ayudara a obtener elementos alcanzables mediante λ transición a partir de un elemento inicial.

3.2.1 Grafo R_λ

A lo largo de todas las fases del algoritmo de marcado resulta necesario definir una nueva estructura cuya misión sea la de proporcionarnos un conjunto de elementos alcanzables por λ transición desde un elemento concreto. Para esto consultaremos un grafo denominado R_λ que representa todos los posibles pares $< e_p, e_h>$ donde e_p y e_h son elementos LR (0) derivados de la gramática original, de forma que eh es derivable mediante λ transición a partir de e_p. Más formalmente definido como $R_\lambda \leftarrow \{< A \rightarrow \alpha.B\beta, B \rightarrow .\gamma > \mid A \rightarrow \alpha B\beta, B \rightarrow \gamma \in P\}$ donde P sea el conjunto de producciones de la gramática original.

Toda la información relativa a la implementación de esta nueva estructura se encuentra en la clase RLambda.java. Esta clase posee los atributos:

- El conjunto de transiciones del grafo, está implementado por una tabla hash especial HashTableDK<Element, Element> **result**. La clave de la tabla es un elemento LR (0) y como valor la lista de elementos LR (0) alcanzables por λ transición desde ese elemento.
- La tabla HashTableDK<Element, Element> **update**. Es una tabla igual a la tabla result tan solo que los elementos tienen actualizados sus elementos de preanálisis una vez calculados en la fase 1 del algoritmo de marcado.
- Un elemento **extensión**. Este elemento es el elemento LR (0) por cual se extiende la gramática original, cuyo único elemento del cuerpo consiste en el axioma original y la cabeza es el nuevo axioma de la gramática.

En la misma clase se definen una serie de métodos para la actualización del grafo, necesarios durante las fases del algoritmo, que permitan eliminar transiciones entre elementos (método deleteTransitions) o que permita actualizar los símbolos de preanálisis de los elementos del grafo (método update).

A modo de ejemplo para clarificar el concepto introduciremos una gramática y mostraremos el grafo R_λ resultante. Tengamos la siguiente gramática:

S → A a

S → B

A → C

A → D b

C → E

E → F

F → E c

F → B

D → B

B → a

En esta gramática el elemento extensión seria ***S' → S*** y el grafo R_λ quedaría así:

R_λ = {<S' → .S, S → .Aa>, < S' → .S, S → .B>, <S → .Aa, A → .C>, <S → .Aa, A → .Db>, < A → .C, C → .E>, < A → .Db, D → .B>, <S → .B,B →.a>, < D → .B, B → .a>, < C → .E, E → .F>, < E → .F, F → .Ec>, <F → .Ec, E→.F>, < E → .F , F → .B>, <F →.B, B → .a>}

3.2.2 Fase 1: Construcción del autómata LALR(1)

El objetivo de esta fase es obtener el autómata finito LALR (1) asociado a una gramática. Por motivos de eficiencia en los cálculos posteriores de las posiciones donde se pueden insertar marcadores se optó por sustituir la implementación del autómata como grafo. En su lugar toda la información relevante sobre cada estado del autómata se guarda en unas tablas que nos permiten un acceso más rápido a la información. Así las estructuras necesarias para describir el autómata son:

- Una tabla **Estados** donde se almacenan todos los conjuntos de elementos LR (0) asociados a un determinado estado.
- Una tabla **Nucleos** donde se almacenan todos los conjuntos de elementos LR (0) nucleares asociados a un determinado estado.
- Una lista **Transiciones** donde se almacenan tuplas del estilo <i, X, j> donde se indica que se transita del estado i al estado j una vez consumido el símbolo X.

La construcción se realiza en dos fases bien diferenciadas donde primero generaremos la estructura del autómata y posteriormente calcularemos los símbolos de preanálisis de cada elemento LR (0) asociado a los estados.

3.2.2.1 Obtención de la estructura del grafo

En esta fase se calculan los conjuntos de elementos LR (0) de cada estado (de ahora en adelante lo llamaremos **Estados$_i$** siendo i el identificador de estado), los conjuntos de elementos LR (0) nucleares de cada estado (de ahora en adelante lo llamaremos **Nucleos$_i$** siendo i el identificador de estado) y el conjunto de transiciones entre estados **Transiciones.** Para ello utilizaremos una pila Γ que contiene tuplas de la forma <**a**, **X**, **C**> donde **a** representa un identificador de estado del autómata, **X** es un símbolo de la gramática y **C** representa un conjunto de elementos LR (0) nucleares de un estado que existe o que aun no se ha creado**.** Así el conjunto **C** se obtiene transitando sobre el símbolo **X** los elementos LR (0) originales que pertenecían al estado identificado por **a**.

El proceso consiste en iterar sobre la pila Γ hasta que se quede vacía, de manera que en cada iteración se desapila una tupla y se comprueba si el conjunto de elementos nucleares de la tupla se corresponde con los elementos nucleares de un estado creado anteriormente. En caso de que exista un estado ***j*** con exactamente los mismo elementos nucleares que la tupla desapilada (sea <**n**, **X**, **K**>) se actualiza el conjunto **Transiciones** con una transición <**n**, **X**, **j**> del estado identificado por **n** de la tupla desapilada al estado **j** transitando con el símbolo **X** de la tupla desapilada.

En caso de que no exista ningún estado que contenga exactamente los mismos elementos nucleares de la tupla desapilada (sea <**n**, **X**, **K**>), entonces dichos elementos constituyen elementos nucleares de un nuevo estado que debe ser desarrollado. Sea i el identificador del nuevo estado, para desarrollarlo seguiremos los siguientes pasos:

1. Se crearan nuevos conjuntos $\textbf{Nucleos}_i$ y $\textbf{Estados}_i$ que contienen los elementos desapilados **K**. También se creara un nuevo conjunto **SimbolosTransicion** que almacene los símbolos que permitan transitar de este nuevo estado a otros. Utilizaremos una nueva pila Π donde se insertaran los elementos nucleares **K**

2. Se itera sobre la pila Π hasta que se quede vacía y se añaden al conjunto $\textbf{Estados}_i$ todos los elementos alcanzables por transición desde el elemento desapilado. Se añade al conjunto **SimbolosTransicion** el símbolo que hay a la derecha del punto de este elemento desapilado.

3. Para cada conjunto de elementos de $\textbf{Estados}_i$ que tienen un símbolo de transición común se crea una tupla con el identificador **i**, el símbolo de transición común y dicho conjunto de elementos insertándose en la pila Γ**.**

El algoritmo completo para esta etapa se muestra continuación, siendo necesario comentar que para que funcione correctamente es necesario definir el grafo $\mathbf{R_\lambda}$ e iniciar el grupo **Transiciones** a vacio, ya que inicialmente no hay ninguna transición definida. Además la pila de tuplas Γ se inicializa con el par <Ø, Ø,S' → .S> siendo S' → .S el elemento nuclear que se genera a través de la producción extendida de la gramática. Como este elemento no es generado por ningún estado se indica con los símbolos Ø que tanto el estado origen como el símbolo de transición están indefinidos.

```
R_λ ←{<A→α.Bβ, B→.γ> | A→αBβ, B→γ∈P}
Γ←[<⊥,⊥,{S'→.S}>];
i←0;
Transiciones←∅;
Mientras Γ≠[] hacer
  Sea Γ=[<n,X,K>|Γ']
     Γ←Γ';
     Si ¬∃ j∈[0,i): Nucleos_j=K entonces {
                                  Nucleos_i←K;
                                  Estados_i←K;
                                  Si (n≠⊥ ∧ x≠⊥) entonces Transiciones ← Transiciones ∪ {<n,X,j>};  FSi;
                                  SimbolosTransicion←∅;
                                  Π←[];
                                  Para cada e∈K hacer
                                        Π←[e|Π];
                                  FPara;
                                  Mientras Π≠[] hacer
                                     Sea Π=[ A →α.β|Π'];
                                     Π←Π';
                                     Preanalisis_{i,A →α.β}←∅;
                                     Si β=Xβ' entonces
                                           SimbolosTransicion ←SimbolosTransicion ∪ {X};
                                     FSi
                                     Para cada e∈ {e| < A →α.β,e>∈R_λ} hacer
                                        Si e∉Estados_i entonces
                                                    Estados_i←Estados_i ∪ {e};
                                                    Π←[e|Π];
                                        FSi
                                     FPara;
                                  FMientras;
                                  Para cada X∈SimbolosTransicion hacer
                                        K'←{<i,X,{A →αX.β}>| A →α.Xβ∈Estados_i};
                                        Γ←[K'|Γ];
                                  FPara;
                                  i←i+1; }
            En caso contrario {Transiciones ← Transiciones ∪ {<n,X,j>}; }
FMientras
```

3.2.2.2 Obtención de los símbolos de preanálisis

En el algoritmo de la fase anterior se puede comprobar que los símbolos de preanálisis de los elementos que forman parte de un estado (**Preanalisis**$_{i,A \rightarrow \alpha.\beta}$, donde i es el identificador de estado y A → α.β es el elemento) se inicializan a vacio. Para calcularlos utilizaremos una pila Γ de tuplas del estilo <a, γ> donde γ es un elemento LR (0) y a es el identificador de estado al que pertenece γ. El proceso consiste en iterar sobre la pila hasta que se quede vacía, de manera que en cada iteración se desapila un par y se comprueba si existen elementos LR (0) alcanzables desde el elemento desapilado, es decir, si en el elemento LR (0) el "." no esta situado en el extremo del cuerpo.

En caso de que existan elementos LR (0) alcanzables entonces se procede de la siguiente manera:

1. Se obtiene el conjunto de símbolos PRIMEROS de la parte aun no procesada del elemento LR (0). En caso de que en este conjunto de símbolos se encuentre λ se eliminara el símbolo λ y se concatenará los símbolos de preanálisis del elemento LR (0).

2. Se obtienen todos los elementos alcanzables por λ transición desde el elemento desapilado, para cada elemento se comprueba si los símbolos de preanálisis contiene alguno de los símbolos calculados en el punto anterior. Si no lo contiene se añade el conjunto de símbolos primeros al conjunto de símbolos de preanálisis del elemento LR (0) correspondiente y se apila el par formado por el identificador de estado actual y el elemento LR (0) cuyos símbolos de preanálisis han sido actualizados.

3. Se considera el elemento LR (0) al que se transita desde el elemento LR (0) desapilado, con ayuda de la lista **Transiciones** averiguaremos en que estado se encuentra el elemento destino**.** Se comprueba si el conjunto de símbolos de preanálisis del elemento destino contiene al conjunto de símbolos de preanálisis del elemento origen. En caso de no contenerlo, se añade este último conjunto al conjunto de símbolos de preanálisis del elemento destino. Además se introduce una nueva tupla en la pila Γ formado por el identificador del estado al que pertenece el elemento destino y el propio elemento destino.

El algoritmo completo para esta etapa se muestra a continuación, para que funcione correctamente es necesario inicializar el contenido del conjunto de símbolos de preanálisis de la producción extendida de la gramática con el símbolo $, así como la pila Γ con el par <0, S' → .S > siendo S' → .S el elemento nuclear que se genera a partir de la producción extendida de la gramática que se encuentra en el estado 0.

```
Preanalisis_{0, S'→.S} ← {$};
Γ←[<0,S'→.S>];
Mientras Γ≠[ ] hacer
  Sea Γ=[<i, A →α.β>|Γ'];
  Γ←Γ';
  Si β=Xβ' entonces {
   Θ←PRIMERO(β'Preanalisis_{i,A →α.Xβ'});
   Para cada e∈ {e| < A →α.Xβ',e>∈R_λ} hacer
     Si Θ⊄ Preanalisis_{i,e} entonces {
                                   Preanalisis_{i,e} ← Preanalisis_{i,e} ∪ Θ;
                                   Γ←[<i,e>|Γ];
                                 }
     FSi;
   FPara;
   Sea j: <i,X,j>∈Transiciones;
   Si Preanalisis_{i,A →α.Xβ'}⊄ Preanalisis_{j,A →αX.β'} entonces {
                          Preanalisis_{j,A →αX.β'} ←Preanalisis_{j,A →αX.β'} ∪ Preanalisis_{i,A →α.Xβ'};
                          Γ←[<j, A →αX.β'>|Γ];
                          }
   FSi;
  FSi;
FMientras;
```

3.2.3 Fase 2: Detección de posiciones prohibidas

El objetivo de esta fase es la detección de las posiciones prohibidas de la gramática. Se entiende por posición prohibida aquella posición dentro del cuerpo de una producción en el que no es posible alojar ningún marcador sin que se produzca un conflicto en la gramática resultante.

Esta etapa supone un cambio con respecto a la detección de posiciones prohibidas implementado en versiones anteriores de XLOP en la que se presentaba un algoritmo incompleto debido a que es un tema en continuo desarrollo. Así ahora podemos presentar un algoritmo más depurado en el que se pretende conseguir una implementación más completa y eficiente.

Para la implementación de esta fase se procederá sobre la estructura R_λ definida previamente, partiendo de ese grafo el algoritmo comenzara una serie de podas de nodos de manera que el grafo resultante cumpla que:

- No existan caminos cíclicos entre nodos
- Ningún par de nodos esta unido por ningún camino de aristas
- Si un nivel contiene nodos de un nivel que representan nodos de la forma A –> α.aβ, entonces todos los nodos del mismo nivel que contienen el símbolo a entre sus símbolos de preanálisis tienen podados sus hijos.

Como partimos de la información contenida en el grafo R_λ sabemos que en el grafo resultante cada nodo será un elemento LR (0) y que entre dos nodos existe una arista si existe un conjunto de λ transición que permitan llegar de un estado a otro. Así pues, los elementos LR (0) que estén representados en el grafo resultante serán los elementos de la gramática que no generen conflictos si alojan un marcador.

La obtención del grafo solución, que será a su vez un subconjunto de R_λ, se obtiene por iteraciones. En cada iteración se selecciona un conjunto de nodos candidatos se formará un nivel y en base a unos criterios que expondremos a continuación se efectuara dos podas sobre dichos nodos.

3.2.3.1 Eliminación de caminos cíclicos y entre nodos del mismo nivel

La idea de esta etapa consiste en ir seleccionando los nodos del grafo R_λ por niveles. Consideraremos dos conjuntos, el conjunto de nodos candidatos a formar un nivel $\mathbf{\Gamma}$ y el conjunto de nodos ya visitados **V**. Mediante un proceso iterativo seleccionaremos uno a uno los nodos de $\mathbf{\Gamma}$ comprobando si los nodos padres de estos ya han sido visitados. Si un nodo padre no hubiera sido visitado querría decir que hay al menos un camino alternativo que llega al nodo actual que no pasa por el nodo padre que hemos comprobado, lo que seria indicador de la existencia de un camino cíclico o un camino entre nodos de un mismo nivel. Para evitar esto eliminamos el nodo padre que no ha sido visitado evitando la posibilidad de estos caminos no deseados. Más formalmente, si se encuentra un nodo $\mathbf{\Gamma_i}$ de $\mathbf{\Gamma}$ para el que exista en el conjunto R_λ un par <e, $\mathbf{\Gamma_{i>}}$ de forma que el nodo **e** no ha sido visitado anteriormente entonces el nodo **e** debe ser eliminado.

El proceso de eliminación del nodo **e** consiste en eliminar todos los pares de R_λ en los que aparece **e** y sustituirlos por pares que relacionan los nodos que aparecen como orígenes en las aristas de la forma <**o**, **e**> con los nodos que aparecen como destinos de la forma <**e**, **d**>. Dando lugar a la inserción de las nuevas tuplas <**o**, **d**>. Los nodos eliminados se guardarían en el conjunto de nodos eliminados llamado **Prohibidos.** Si el nodo eliminado perteneciera al conjunto Γ entonces se actualizaría dicho conjunto eliminándolo y añadiendo los hijos de dicho nodo eliminado que se encuentren en el grafo R_λ. El proceso de eliminación se repetirá reiteradamente hasta que en el conjunto Γ no exista ningún nodo Γ_i para el que exista en el conjunto R_λ un par <e, $\Gamma_{i>}$ de forma que el nodo **e** no ha sido visitado anteriormente.

En la siguiente ilustración mostramos un ejemplo de un grafo que consta de 8 elementos en los que encontramos un camino entre nodos de un mismo nivel (nodos 3 y 4) y un ciclo (nodos 2 y5). En rojo están señalados los nodos que pasarían a ser eliminados.

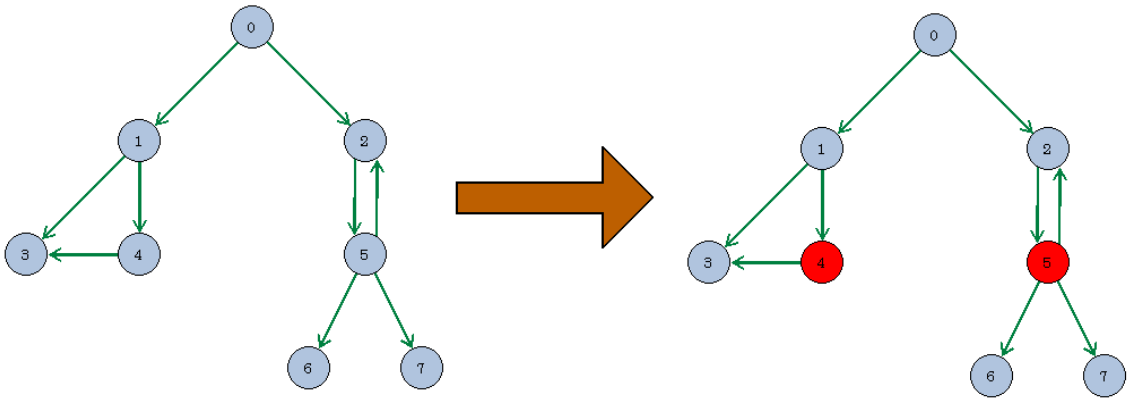

Figura 3.2.1. Proceso de detección de ciclos y caminos entre nodos de un mismo nivel.

Siguiendo el proceso de eliminación descrito anteriormente los nodos 6 y 7 pasarían a ser hijos del nodo 2. Así quedaría el grafo resultante:

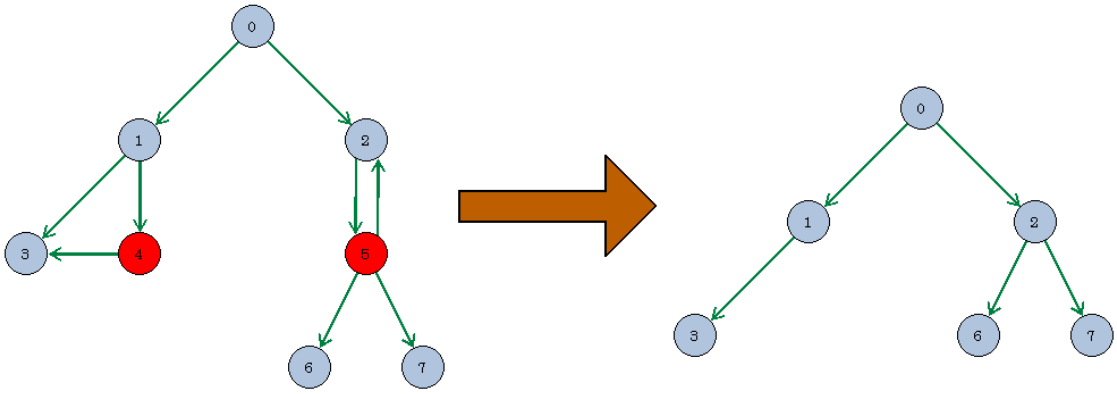

Figura 3.2.2. Proceso de prohibición de nodos

3.2.3.2 Eliminación de posiciones precedidas por un terminal

Esta poda hace referencia a los elementos LR (0) que son de la forma A → α.aβ. Para realizar esta poda se considera el conjunto de nodos a formar un nivel calculado en la etapa anterior, Γ, los estados del autómata LALR(representados por **Estados$_i$** siendo i el identificador de estado) y los conjuntos de símbolos de preanálisis de los elementos LR (0) de los estados del autómata, **Preanalisis$_i$**. Mediante un proceso iterativo, consideramos los nodos de Γ uno a uno de manera que para cada nodo Γ_i de Γ que represente un elemento LR (0) de la forma A → α.aβ, es decir que el punto preceda a un terminal. Se comprueba si entre los demás nodos de Γ existe alguno que aparezca junto al primero en alguno de los estados del autómata y que tenga entre sus símbolos de preanálisis el terminal a. Si se da el caso, se podan todos los nodos del conjunto $\mathbf{R_\lambda}$ que son hijos del nodo que tiene al terminal a entre sus símbolos de preanálisis. Así mismo los nodos que han sido podados de esta manera se almacenan en el conjunto de nodos **Prohibidos.**

El algoritmo completo se muestra a continuación, para que funcione correctamente es necesario que antes de comenzar se haya generado el conjunto $\mathbf{R_\lambda}$. Además también ha de inicializarse el conjunto de nodos candidatos a formar un nivel con todos los elementos LR (0) que aparecen en el conjunto $\mathbf{R_\lambda}$ y en cuyas tuplas aparecen como nodos origen pero nunca como nodos destino.

```
R_λ ←{<A→α.Bβ, B→.γ> | A→αBβ, B→γ∈P};
Γ←{e|∃ <e,e'>∈ R_λ ∧ ¬∃ <e'',e>∈ R_λ };
V←∅;
Prohibidos←∅;
Mientras  Γ≠[] hacer
      Repetir
          i←1;
          nΓ←|Γ|;
          Mientras i≤nΓ ∧ ∀ e: <e,Γ_i>∈ R_λ→ e∈V hacer
                    i←i+1;
          FMientras
          Para cada e∈{e|<e,Γ_i>∈ R_λ ∧ e∉V} hacer
                Prohibidos←Prohibidos∪{e};
                Si e∈Γ entonces Γ←(Γ\{e})∪{e'|<e,e'>∈Rλ ∧ e≠e'}; FSi
                R_λ ← (R_λ \{<e_0,e>|<e_0,e>∈ R_λ }\{<e,e_0>|<e,e_0>∈ R_λ })∪{<e_0,e_1>|<e_0,e>,<e,e_1>∈R_λ ∧ e_0≠e_1≠e};
          FPara
      Hasta i>nΓ;
      Para i←1 hasta |Γ| hacer
       Si Γ_i= A→α.aβ entonces
          Para j=1, j≠i hasta |Γ| hacer
                Sea Γ_j= B→γ.η
                Si ∃ s∈{0…NumEstados-1} : < A→α.aβ>∈Estados_s ∧ <B→γ.η>∈Estados_s ∧
                                                a∈PRIMERO(ηPreanalisis_s) entonces
                                Repetir
                                  H←{e|<B→γ.η,e>∈ R_λ};
                                  R_λ ← (R_λ \{<e_0,e>|<e_0,e>∈ R_λ∧e∈H }\{<e,e_0>|<e,e_0>∈ R_λ ∧e∈H })
                                  ∪{<e_0,e_1>|<e_0,e>,<e,e_1>∈R_λ ∧ e_0≠e_1≠e};
                                   Prohibidos←Prohibidos ∪ H;
                                 Hasta que H= ∅
                FSi
          FPara
       FSi
      FPara
      V←V∪Γ;
      Γ←{e|<e,e'>∈R_λ ∧ e∈Γ};
FMientras
```

3.2.4 Fase 3: Agrupación

En esta fase se busca crear grupos de elementos LR (0) atendiendo a los estados del autómata LALR obtenido en la fase 1 y a los niveles definidos en el grafo $\mathbf{R_\lambda}$ de la fase anterior. Procesaremos los estados del autómata uno a uno y comenzando por los elementos nucleares del estado descenderemos por los niveles que establezca el conjunto $\mathbf{R_\lambda}$ usando como criterio de agrupación que un elemento pertenece a un grupos si posee símbolos de preanálisis comunes con al menos otro elemento del grupo al que pertenece. Como resultado obtendremos un conjunto de grupos de elementos por cada estado del autómata, al que llamaremos $\mathbf{Grupos_j}$, siendo j el identificador del estado.

Para obtener este conjunto de grupos se itera sobre cada uno de los estados del autómata de manera que para un estado **j** concreto inicializaremos $\mathbf{Grupos_j}$ a vacio y el conjunto **F** de elementos candidatos a ser agrupados, se inicializa con el valor de los elementos nucleares del estado **j**, $\mathbf{Nucleos_j}$.

Posteriormente se itera sobre cada nivel de elementos LR (0) del estado **j** de forma que en cada iteración se comienza inicializando un conjunto **NGrupos** que almacena los grupos de elementos LR (0) asociados con el estado **j** en cuestión. La estructura de este nuevo conjunto consiste en almacenar pares de la forma <**Ω**, **g**> donde g es un grupo de elementos LR (0) y **Ω** es un conjunto de símbolos obtenidos mediante la función **PRIMERO** aplicada sobre los elementos de **g** de la forma que se explica mas adelante. Esta función PRIMERO se aplica sobre elementos LR (0) y calcula los símbolos primeros del símbolo situado a la derecha del "." y en caso de derivar λ se concatena los símbolos de preanálisis de dicho elemento.

N se itera sobre cada elemento LR (0) que se encuentra en **F** inicializando el conjunto **Ω** con el conjunto de símbolos **PRIMERO** de los símbolos a la derecha del "." y los símbolos de preanálisis del elemento LR (0) del conjunto **F** considerado en esa iteración. A continuación inicializamos a vacio un nuevo conjunto **Ω''** de símbolos de preanálisis que almacenará todos los símbolos obtenidos mediante la función **PRIMERO** aplicada sobre cada uno de los elementos LR (0) que forman un grupo e inicializar un conjunto **g** con el elemento LR (0) considerado en esa iteración. Este conjunto representará al grupo de elementos de **F** que son agrupables junto con dicho elemento.

Se itera sobre todos los pares del conjunto **NGrupos**, de manera que para cada par < **Ω'**, **g'**> se compara si el conjunto **Ω**' tienen algún elemento en común con el conjunto **Ω** obtenido previamente. En caso afirmativo se fusionan los grupos de elementos LR (0) **g** y **g'** y se añade el conjunto de símbolos **Ω'** al conjunto de símbolos **Ω''**, y se elimina el par < **Ω'**, **g'**> del conjunto **NGrupos**.

Finalmente se actualiza el conjunto **NGrupos** con el par < **Ω** U **Ω'', g**>. Una vez finalizada todas las iteraciones se actualiza el conjunto **Grupos**$_j$, con el conjunto de grupos obtenidos en el nivel actual **NGrupos.**

El algoritmo completo se muestra a continuación.

```
Para j←0 hasta NumEstados-1 hacer
        Grupos_j←∅;
        F←Nucleos_j,
                Mientras F≠∅ hacer
                  Ngrupos←∅;
                  Para cada A→αβ∈F hacer
                        Ω←PRIMERO(βPreanálisis_{j,A→αβ});
                        Ω''←∅;
                        g←{ A→αβ};
                        Para cada <Ω',g'>∈ Ngrupos hacer
                            Si Ω ∧ Ω'≠∅ entonces
                                        g←g∪g';
                                        Ω''←Ω''∪Ω';
                                    Ngrupos← Ngrupos\{<Ω',g'>};
                          FSi
                       FPara
                          Ngrupos← Ngrupos∪{<Ω∪Ω'',g'>};
                  FPara
                  Grupos_j← Grupos_j ∪ {g | <Ω,g>∈ Ngrupos};
                  F←{e'|e∈F ∧ <e,e'>∈ R_λ};
        FMientras;
FPara;
```

3.2.5 Fase 4: Eliminación de anomalías de fusión

Esta fase sirve como complemento de la fase anterior en la que eliminaremos los grupos que aparecen más de una vez en los conjuntos de grupos de elementos LR (0) obtenido por cada estado del autómata LALR.

El algoritmo completo se muestra a continuación.

```
Para j←0 hasta NumEstados-1 hacer
  Para k←j+1 hasta NumEstados-1 hacer
        C← Grupos_j ∧ Grupos_k;
        Grupos_j← Grupos_j\C;
        Grupos_k← Grupos_k\C;
  FPara;
FPara;
```

3.2.6 Fase 5: Asignación de marcadores

Una vez completadas todas las etapas anteriores del algoritmo estamos en condiciones de asignar los nuevos símbolos no terminales, denominados marcadores, representados por los elementos LR (0) que aparecen en los grupos de elementos LR (0) por estado generado entre las fases 3 y 4. Previamente a la asignación de marcadores tenemos que tener en cuenta que todos los elementos LR (0) que pertenecen al mismo grupo se les debe asignar el mismo marcador debido a que comparten información de preanálisis. Por otro lado, para cada estado del autómata LALR hay un conjunto de grupos de elementos se puede dar el caso de que un mismo elemento LR (0) este presente en mas de un grupo de diferentes estados. Estos razonamientos nos llevan a la conclusión de que para que la asignación de marcadores sea correcta y tenga sentido es necesario fusionar los grupos en los que aparece un elemento común en un único grupo.

La solución de esta fase será una tabla **Marcadores** que para cada elemento LR (0) devuelve el marcador que le ha sido asignado. Para la obtención de esta tabla se usa un conjunto Γ que almacenará los grupos de elementos LR (0) a cuyas posiciones se les debe asignar un marcador, inicialmente contendrá el conjunto vacio.

Suponiendo que el conjunto de grupos de elementos LR (0) por estado sea **Grupos$_j$** siendo j el identificador de estado. Se itera por cada conjunto de grupos por estado, de forma que para cada grupo **g** del conjunto **Grupos$_j$** se comprueba si existe algún otro grupo **g'** en Γ con el que comparta elementos LR (0). En caso afirmativo el grupo **g'** se elimina de Γ y se fusiona con el grupo **g**. Despues de haber comprobado esta condición sobre **g** para todos los grupos de Γ, entonces el grupo resultante se añade a Γ.

Una vez realizado esto todos se finalizaría la fusión de grupos y tan solo quedaría la asignación de marcadores para lo cual se recorre cada grupo de Γ, asignando a todos los elementos LR (0) de un grupo dado el mismo marcador.

```
Γ←∅;
Para j←0 hasta NumEstados-1 hacer
    Para cada g∈ Grupos_j hacer
        ng←∅;
        Para cada g'∈ Γ hacer
            Si g'∧g≠∅ entonces
                ng←ng∪g';
                Γ←Γ\{g'};
            FSi;
        FPara;
        Γ←Γ∪{ng∪g};
    FPara;
FPara;
Para j←0 hasta |Γ| hacer
  Para cada e∈Γ_j hacer
      Marcador[e]←M_j;
  FPara;
FPara;
```

3.3 Ejemplo ilustrativo

Vamos considerar una gramática, que hemos utilizado como uno de nuestros casos de prueba durante la realización del proyecto, para comentar los resultados obtenidos en cada fase del algoritmo. En este caso se trata de a misma gramática que hemos empleado en el capítulo explicativo del grafo $\mathbf{R_\lambda}$.

3.3.1 Fase 1

Como hemos comentado en la primera fase se construye la estructura del autómata LALR pero para ello primero hay que calcular el grafo $\mathbf{R_\lambda}$ como hemos calculado en el capitulo sobre este grafo.

S → A a

S → B

A → C

A → D b

C → E

E → F

F → E c

F → B

D → B

B → a

En esta gramática el elemento extensión seria ***S' → S*** y el grafo R_λ quedaría así:

R_λ = {<S' → .S, S → .Aa>, < S' → .S, S → .B>, <S → .Aa, A → .C>, <S → .Aa, A → .Db>, < A → .C, C → .E>, < A → .Db, D → .B>, <S → .B,B →.a>, < D → .B, B → .a>, < C → .E, E → .F>, < E → .F, F → .Ec>, <F → .Ec, E→.F>, < E → .F , F → .B>, <F →.B, B → .a>}

Aplicando el algoritmo de la fase uno y tras varias iteraciones se consigue el siguiente grafo que representa el autómata LALR, en cada estado se representan los elementos LR (0) asociados con ese estado así como sus elementos de preanálisis. Cada arista entre estados representa la transición entre dos estados e incluye el símbolo por el cual se transita.

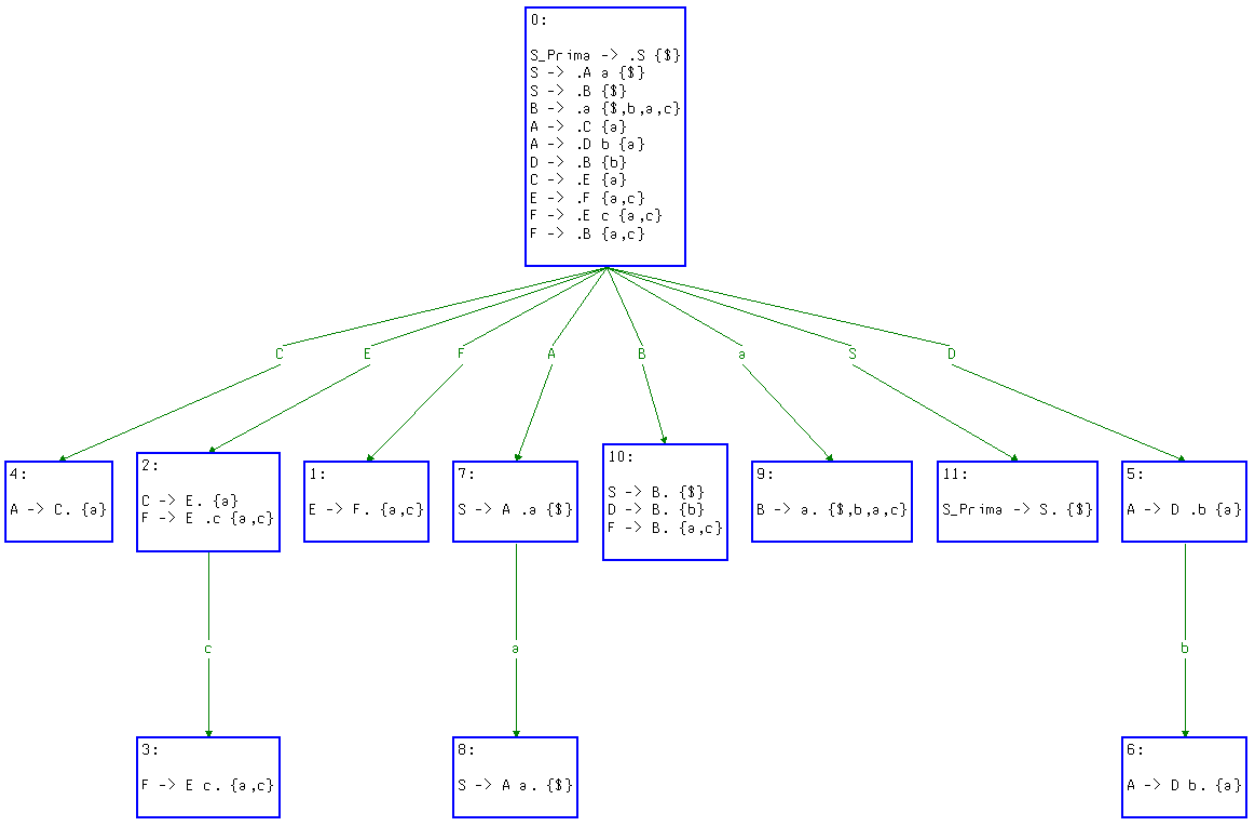

Imagen 3.3.1. Autómata LALR resultado de la gramática estudiada.

3.3.2 Fase 2

Posteriormente comenzamos la fase dos, la fase de prohibición constaba de un bucle principal cuya condición de salida era que la pila Γ estuviera vacía. Dentro de este bucle existían otros dos que detectaban los caminos cíclicos y otro que eliminaba las posiciones precedidas de un terminal. A continuación comentamos los resultados de estas iteraciones.

Inicialmente nuestra pila Γ contendrá la primera posición derivada del elemento extensión de la gramática. Es decir, Γ = {S'→.S} y por supuesto el conjunto de nodos visitados y nodos prohibidos es vacio, esto es, **V** = {Ø} y **Prohibidos** = {Ø}.

Así pues iteramos sobre la pila Γ hasta que quede vacía:

Primera iteración del bucle principal

- **Primer bucle**

  Sin efectos.

- **Segundo bucle**

  Sin efectos.

Γ = {S →.Aa, S →.B} V = {S'→.S}

Segunda iteración del bucle principal

- **Primer bucle**

  Sin efectos.

- **Segundo bucle**

  Sin efectos.

Γ = {A →.C, A →.Db, B →.a} V = {S'→.S, S →.Aa, S →.B}

Tercera iteración del bucle principal

- **Primer bucle**

  Prohibidos= {D →.B, F →.B, A →.Db, E →.F, C →.E, F →.Ec, A →.C}

  Puesto que se han prohibido estos nodos han de ser eliminados del conjunto R_λ de la manera explicada quedando así:

  R_λ = {<S' → .S, S → .Aa>, < S' → .S, S → .B>, <S → .Aa, B →.a>, <S → .B,B →.a}

- **Segundo bucle**

  Sin efectos.

Γ = { } V = {S'→.S, S →.Aa, S →.B, B →.a}

Como la pila Γ ha quedado vacía el bucle termina y por tanto la etapa.

3.3.3 Fase 3

A continuación se realiza la tercera fase del algoritmo donde representaremos para cada estado los conjuntos de grupos de elementos LR (0) obtenidos para cada uno de ellos.

Estado	Grupos
0	{[S' → .S], [S → .B, S → .A a], [B → .a]}
1	{[E → F.]}
2	{[C → E.], [F → E .c]}
3	{[F → E c.]}
4	{[A → C.]}
5	{[A → D .b]}
6	{[A → D b.]}
7	{[S → A .a]}
8	{[S → A a.]}
9	{[B → a.]}
10	{[S → B.], [D → B.], [F → B.]}
11	{[S' → S. {$}]}

3.3.4 Fase 4

Con estos resultados podemos observar que no hay ningún grupo de elementos LR (0) que se repita en dos o más estados por lo que la fase cuatro que resuelve las anomalías de fusión no tendrá efecto sobre estos datos. Así esta será la información que se le pase a la fase cinco para el cálculo de marcadores.

3.3.5 Fase 5

Finalmente en la última fase se recorren los grupos en busca de grupos comunes para asociarlos bajo un mismo marcador pero al no encontrar grupos con elementos comunes no modifica los grupos actuales.

Así los marcadores obtenidos por la fase cinco son:

Elemento	Marcador
S’ → .S	M0
S → .B, S → .A a	M1
B → .a	M2
E → F.	M3
C → E.	M4
F → E .c	M5
F → E c.	M6
A → C.	M7
A → D .b	M8
A → D b.	M9
S → A .a	M10
S → A a.	M11
B → a.	M12
S → B.	M13
D → B.	M14
F → B.	M15
S’ → S.	M16

Y la gramática marcada tendría una estructura así:

S’ -> M0 S M16

S -> M1 A M10 a M11

S -> M1 B M13

A -> C M7

A -> D M8 b M9

C -> E M4

E -> F M3

F -> E M5 c M6

F -> B M15

D -> B M14

B -> M2 a M12

Capítulo 4

Modularización de gramáticas en XLOP

4.1. Introducción

Tras la creación de la primera versión de XLOP, se piensa en la posibilidad de ampliar su potencia. La idea es que no sólo sea capaz de admitir una sola gramática, como hasta ahora, sino la posibilidad de introducir una gramática en varios módulos.

Esta idea permite además pensar en XLOP como una aplicación capaz de admitir una gramática principal y poder cargar módulos auxiliares, sin necesidad de cargar todos ellos, solamente aquellos que sean de utilidad para el usuario en ese momento determinado. Entendemos como gramática principal aquella en la que está el axioma de la gramática resultante de la unificación de todas las gramáticas. Todo eso se muestra en el ejemplo que hay al final del capítulo.

4.2. Modularizado de gramáticas

4.2.1. Concepto

Para conseguir el modularizado de gramáticas se amplía la interfaz de XLOP, con el fin de poder introducir varios módulos de la gramática, y se modifica el modelo de objetos.

Esto se consigue poniendo nombres cualificados a los no terminales, siendo “default” el nombre cualificado por defecto. Un nombre cualificado es simplemente un “prefijo” delante del no terminal. El apartado 4.2.2 contiene un ejemplo que ayuda a comprender este nuevo concepto.

Hay tres maneras de modificar los nombres cualificados de los no terminales:

- Mediante la instrucción *namespace*
- Mediante la instrucción *qualify*
- Cualificación directa

La idea es poder hacer referencia a no terminales que estén en otros módulos de la gramática. De esta manera se está haciendo referencia a un no terminal que está en otro módulo y que todavía no ha sido analizado.

El procedimiento que usa XLOP a la hora de analizar dichos módulos gramaticales es analizar cada módulo por separado, analizando su espacio de nombres e instrucciones internas. Una vez realizado el análisis del primer módulo, se añade a lo que, al final, será la gramática definitiva. Una vez concluido el análisis e incorporación a la gramática final, se continúa con el siguiente módulo repitiendo las mismas operaciones. Y así sucesivamente con los módulos restantes hasta que el último esté analizado e incluido en la gramática definitiva. En el apartado 4.3 se muestra un ejemplo ilustrativo de todo el procedimiento.

4.2.2. Ejemplo

Este es un fragmento de código de una gramática trivial. A ojos del usuario estaría escrita así:

persona ::= <persona> nombre edad <persona>
nombre ::= <nombre> #pcdata </nombre>
edad ::= <edad> #pcdata </edad>

Sin embargo en la implementación de XLOP aplicando el espacio de nombres quedaría de la siguiente manera:

default.persona ::= <persona> default.nombre default.edad <persona>
default.nombre ::= <nombre> #pcdata </nombre>
default.edad ::= <edad> #pcdata </edad>

Como podemos observar a los no terminales “persona”, “nombre” y “edad” se les añade el prefijo “default”, ya que, como se ha dicho en el apartado anterior, es el nombre cualificado por defecto.

4.3. Implementación de la modularización en XLOP

4.3.1. Estructura del sistema del espacio de nombres

Una vez explicado el concepto de nombres cualificados, se continúa con la estructura que tiene el espacio de nombres y como se ha implementado a nivel interno.

4.3.1.1. Instrucción namespace

Esta es la instrucción más genérica dentro de la estructura del sistema de espacio de nombres. Siempre se declara al principio de la gramática y solamente puede existir una instrucción *namespace* por módulo. La sintaxis es la siguiente:

namespace <nombre del espacio de nombres>;

Si no apareciera la instrucción namespace se tomará el espacio de nombres por defecto, en este caso es “*default”*.

A través de este sistema, en los módulos de gramática es posible emplear nombres locales o bien nombres cualificados. El nombre completo del no terminal vendrá dado por su nombre cualificado, seguido por un punto y por último el nombre local del terminal. El apartado 4.3.1.1.1. contiene un breve ejemplo acerca de la instrucción *namespace*.

4.3.1.1.1. Ejemplo namespace

Haciendo referencia al ejemplo del apartado 4.2.2. se va a aplicar la instrucción *namespace* a fin de observar el efecto de dicha instrucción sobre un fragmento de gramática. Sea pues dicho fragmento con la instrucción *namespace* correspondiente:

namespace es.ucm.fdi;
persona ::= <persona> nombre edad <persona>
nombre ::= <nombre> #pcdata </nombre>
edad ::= <edad> #pcdata </edad>

En este ejemplo los nombres locales son “persona”, “nombre” y “edad”. El nombre del espacio de nombres, el de la instrucción *namespace*, es “es.ucm.fdi”. Por lo tanto este fragmento de gramática es equivalente a este:

es.ucm.fdi.persona ::= <persona> es.ucm.fdi. nombre es.ucm.fdi.edad <persona>
es.ucm.fdi.nombre ::= <nombre> #pcdata </nombre>
es.ucm.fdi.edad ::= <edad> #pcdata </edad>

Se observa como todo no terminal sigue la estructura explicada en el apartado 4.3.1.1. Nombre cualificado, seguido de punto, seguido de nombre local del no terminal.

4.3.1.2. Instrucción qualify

Además de la instrucción *namespace* también es posible cambiar el nombre cualificado de un no terminal a través de la instrucción *qualify*. Así como la instrucción *namespace* modifica el espacio de nombres para todo el archivo de la gramática, la instrucción *qualify* solamente modifica el nombre cualificado para un no terminal determinado.

A diferencia de la instrucción *namespace*, la instrucción *qualify* puede, bien no ser utilizada, o bien aparecer una o más veces. La declaración de la instrucción *qualify* irá después de la instrucción *namespace*, siempre que haya instrucción *namespace*. Si no hubiera instrucción *namespace* y se utilizara la instrucción *qualify*, ésta última sería la primera en aparecer en el archivo.

La sintaxis de la instrucción *qualify* es la siguiente:

qualify <nombre local> ***as*** <nombre cualificado>;

El efecto que tiene esta instrucción es muy sencillo. Simplemente cambia el nombre local por el nombre cualificado.

Existe la restricción de que no puede haber dos instrucciones qualify contradictorios sobre el mismo no terminal o nombre local. Tampoco pueden existir dos instrucciones qualify idénticas sobre el mismo no terminal.

Este fragmento muestra la primera restricción. Aquí se ve como el no terminal “edad” intenta ser cualificado como “es.ucm.fdi.edad” y al mismo tiempo también intenta ser cualificado como “edu.mit.edad”.

```
qualify edad as es.ucm.fdi.edad;
qualify edad as edu.mit.edad;
```

Este segundo fragmento muestra la segunda restricción. Es trivial ver que el mismo no terminal cualificado dos veces idénticas no tiene sentido.

```
qualify edad as es.ucm.fdi.edad;
qualify edad as es.ucm.fdi.edad;
```

Aunque la instrucción *namespace* modifica todo el espacio de nombres del archivo donde está la gramática, la instrucción *qualify* tiene más prioridad. Así, cuando se encuentra el no *terminal* que se quiere cualificar mediante la instrucción *qualify*, en vez de utilizar el nombre cualificado que contiene la instrucción *namespace*, se utiliza el nombre cualificado que indique la instrucción *qualify*.

El apartado 4.3.1.2.1. contiene un ejemplo con un fragmento de gramática en el cual se muestra el efecto que tendría una gramática con una instrucción *namespace* y también una instrucción *qualify*.

4.3.1.2.1. Ejemplo qualify

Para este ejemplo seguimos utilizando el ejemplo del apartado 4.2.2. al cual se le ha introducido una instrucción namespace y una instrucción qualify para poder ver el efecto que tienen ambas cuando aparecen en un mismo archivo.

namespace es.ucm.fdi;
qualify edad as edu.mit.edad;
persona ::= <persona> nombre edad <persona>
nombre ::= <nombre> #pcdata </nombre>
edad ::= <edad> #pcdata </edad>

En este caso el espacio de nombres de la gramática es “es.ucm.fdi”, es decir, todo no terminal será cualificado con este nombre. Después de la instrucción namespace aparece la instrucción qualify. Aquí se quiere renombrar el no terminal “edad” por “edu.mit.edad”. De esta manera el espacio de nombres cualificará todos los no terminales excepto el no terminal “edad”, ya que cuando se vaya a renombrar se utilizará el nombre cualificado que aparece en la instrucción qualify. Esta es la gramática equivalente:

es.ucm.fdi.persona ::= <persona> es.ucm.fdi.nombre edu.mit.edad <persona>
es.ucm.fdi.nombre ::= <nombre> #pcdata </nombre>
edu.mit.edad ::= <edad> #pcdata </edad>

4.3.1.3. Cualificación directa

Esta es la tercera forma de cualificar un no terminal y además es la que mayor prioridad tiene. Para cualificar de esta forma simplemente se escribe delante del no terminal que se desea cualificar el nombre cualificado que queremos darle seguido de un punto y del nombre local que tenga el no terminal. Esta es la sintaxis:

… <nombre cualificado>.<nombre local> …

He aquí un breve ejemplo de cualificación directa. Se tiene la gramática del apartado 4.2.2.:

persona ::= <persona> nombre edad <persona>
nombre ::= <nombre> #pcdata </nombre>
edad ::= <edad> #pcdata </edad>

Se quiere cualificar de forma directa el no terminal "nombre" como "es.ucm.fdi". Al ser la forma que más prioridad tiene no importara si existe instrucción *namespace* o si existe instrucción *qualify* sobre el no terminal "nombre", ya que prevalecerá la cualificación directa. Simplemente hay que sustituir de manera manual el no terminal "nombre" por "es.ucm.fdi.nombre" y quedará cualificado. Así quedaría dicha gramática con el no terminal "nombre" cualificado tal y como se ha explicado:

persona ::= <persona> es.ucm.fdi.nombre edad <persona>
es.ucm.fdi.nombre ::= <nombre> #pcdata </nombre>
edad ::= <edad> #pcdata </edad>

4.3.2. Adaptación de la interfaz al modularizado de gramáticas

Para poder llevar acabo el modularizado de gramáticas es necesario una adaptación de la interfaz que permita seleccionar varias gramáticas al mismo tiempo.

Para conseguir esto se crea un nuevo campo dentro de la interfaz donde se permite al usuario seleccionar los fragmentos de gramáticas que compondrán la gramática final. Además el usuario debe dejar seleccionado el fragmento de gramática donde se encuentra el axioma de la gramática final. En la Figura 4.3.1. se puede observar recuadrado con rojo el nuevo campo introducido y como se han cargado 3 gramáticas, siendo la gramática que contiene el axioma “Messenger11.xlop”.

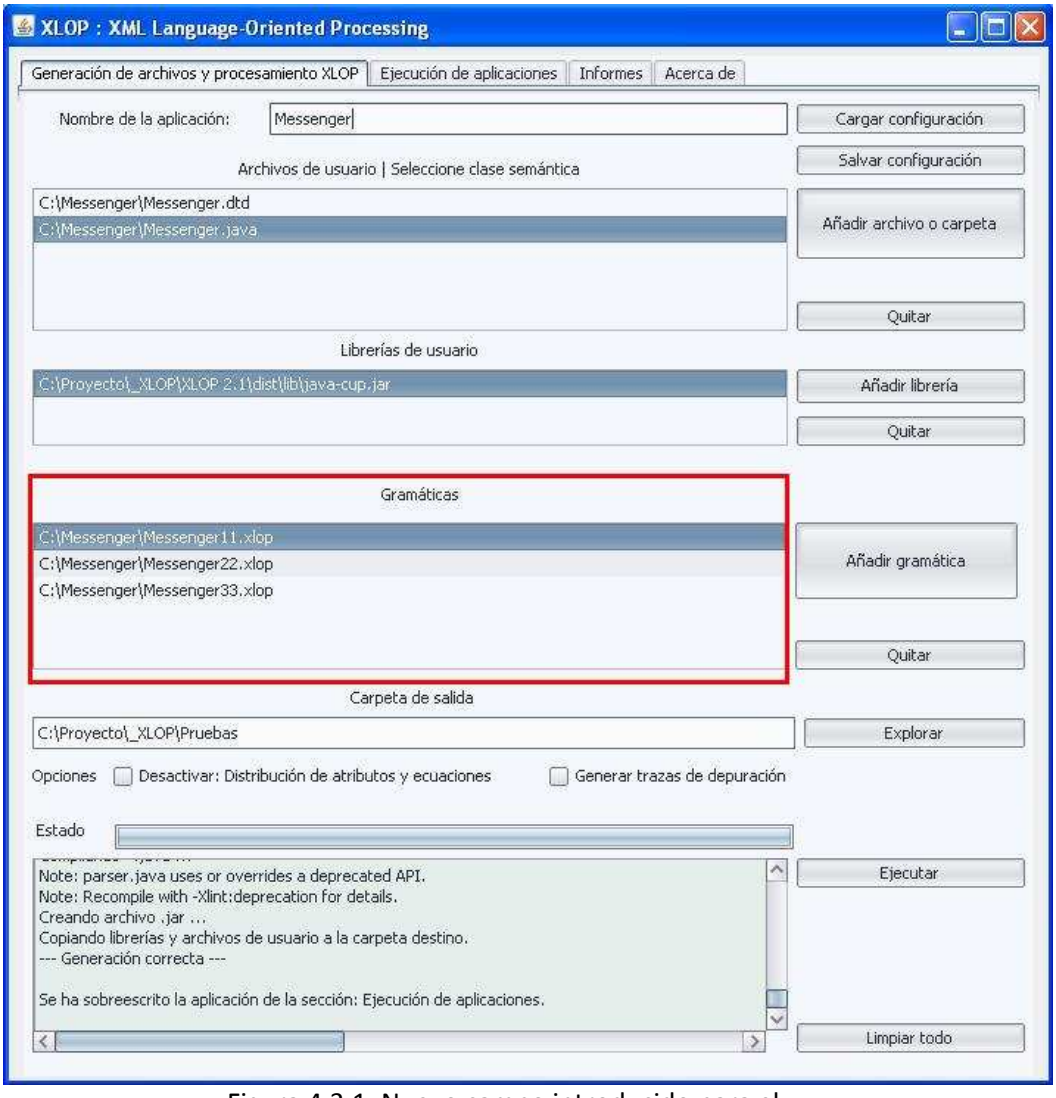

Figura 4.3.1. Nuevo campo introducido para el modularizado de gramáticas.

4.4. Ejemplo ilustrativo

Para poder ver la potencia que tiene esta nueva funcionalidad en la aplicación de XLOP, vamos a mostrar un ejemplo ilustrativo donde se vea el efecto de la instrucción “namespace”, la instrucción “qualify” y la cualificación directa. Además de poder ver dichas instrucciones, también se puede observar lo útil que puede ser tener una gramática modulada.

En el ejemplo vamos a usar tres módulos:

- **“Supermercado.xlop”**

```
namespace supermercado;
qualify Departamentos as departamento.Departamentos;

Supermercado ::= <supermercado> <departamentos> Departamentos </departamentos>
                 <empleados> empleado.Empleados </empleados> </supermercado> {
    depTotal of Supermercado = imprime(resultado of Departamentos)
    empTotal of Supermercado = imprime(resultado of empleado.Empleados)
}
```

- **“Departamentos.xlop”**

```
namespace departamento;

Departamentos ::= Departamentos Departamento{
    resultado of Departamentos(0) = concatenaDep(resultado of Departamentos(1),
                                                 resultado of Departamento)
}
```

Departamentos ::= Departamento{

resultado of Departamentos = resultado of Departamento

}

Departamento ::= <nombre> #pcdata </nombre>{

resultado of Departamento = text of #pcdata

}

- **“Empleados.xlop”**

namespace empleado;

Empleados ::= Empleados Empleado{

resultado of Empleados(0)= concatenaEmp(resultado of Empleados(1),resultado of Empleado)

}

Empleados ::= Empleado{

resultado of Empleados = resultado of Empleado

}

Empleado ::= <empleado> <dni> DNI </dni> <nombre> Nombre </nombre> </empleado>{

resultado of Empleado = concatenaDNI(dni of DNI, nombre of Nombre)

}

DNI ::= #pcdata{

dni of DNI = text of #pcdata

}

Nombre ::= #pcdata{

nombre of Nombre = text of #pcdata

}

Esta gramática lo que hace es simplemente imprimir el listado de departamentos y empleados que hay en el supermercado. Se puede ver la utilidad que tiene una gramática en módulos.

En primer lugar es muy importante observar que con un mínimo cambio en la gramática principal, "Supermercados.xlop", se podría añadir o eliminar algún no terminal al axioma "Supermercado" y cargar o dejar de cargar módulos en la aplicación XLOP. Es decir, si se quisiera cargar un nuevo módulo que calculara los productos que hay en el supermercado, simplemente habría que añadir el no terminal "productos" en el axioma y crear un módulo que contuviera la gramática "Productos". O si por el contrario se quisiera dejar de mostrar los departamentos que contiene el supermercado solamente habría que eliminar las apariciones del no terminal "Departamentos" y no cargar el archivo "Departamentos.xlop".

En segundo lugar se podría tener dos archivos distintos con el no terminal "departamentos". Con ello se podría hacer referencia a un archivo o a otro dependiendo de las necesidades del usuario.

Se va a analizar el ejemplo para que se vea el efecto de las instrucciones usadas en ella. Se toma como gramática principal la que contiene el axioma. En este caso "Supermercado.xlop" es la que contiene el axioma. En ella se observa que se ha utilizado la instrucción "*namespace*" para cambiar el espacio de nombres a "***supermercado***". También se ha utilizado una instrucción "*qualify*" para cualificar el no terminal "Departamentos" a "***departamento.Departamentos***". Además también se ha utilizado la cualificación directa con el no terminal "Empleados", que ahora pasa a tener el nombre cualificado "***empleado.Empleados***". De esta manera todos los no terminales distintos de "Departamentos" y "Empleados" tendrán como nombre cualificado el de la instrucción "*namespace*", en este caso "**supermercado**".

En el archivo "Departamentos.xlop" se utiliza la instrucción "namespace" siendo ahora el espacio de nombres de dicho archivo "***departamento***". Esto se hace para, que cuando carguemos las gramáticas en XLOP, el no terminal "Departamentos", del archivo "Supermercados.xlop" y "Departamentos.xlop", tengan el mismo nombre cualificado y así sean el mismo no terminal. Así de igual manera en el archivo "Empleados.xlop" se vuelve a cambiar el espacio de nombres por el de "***empleado***".

La figura 4.4 muestra la que sería la configuración de XLOP para esta gramática.

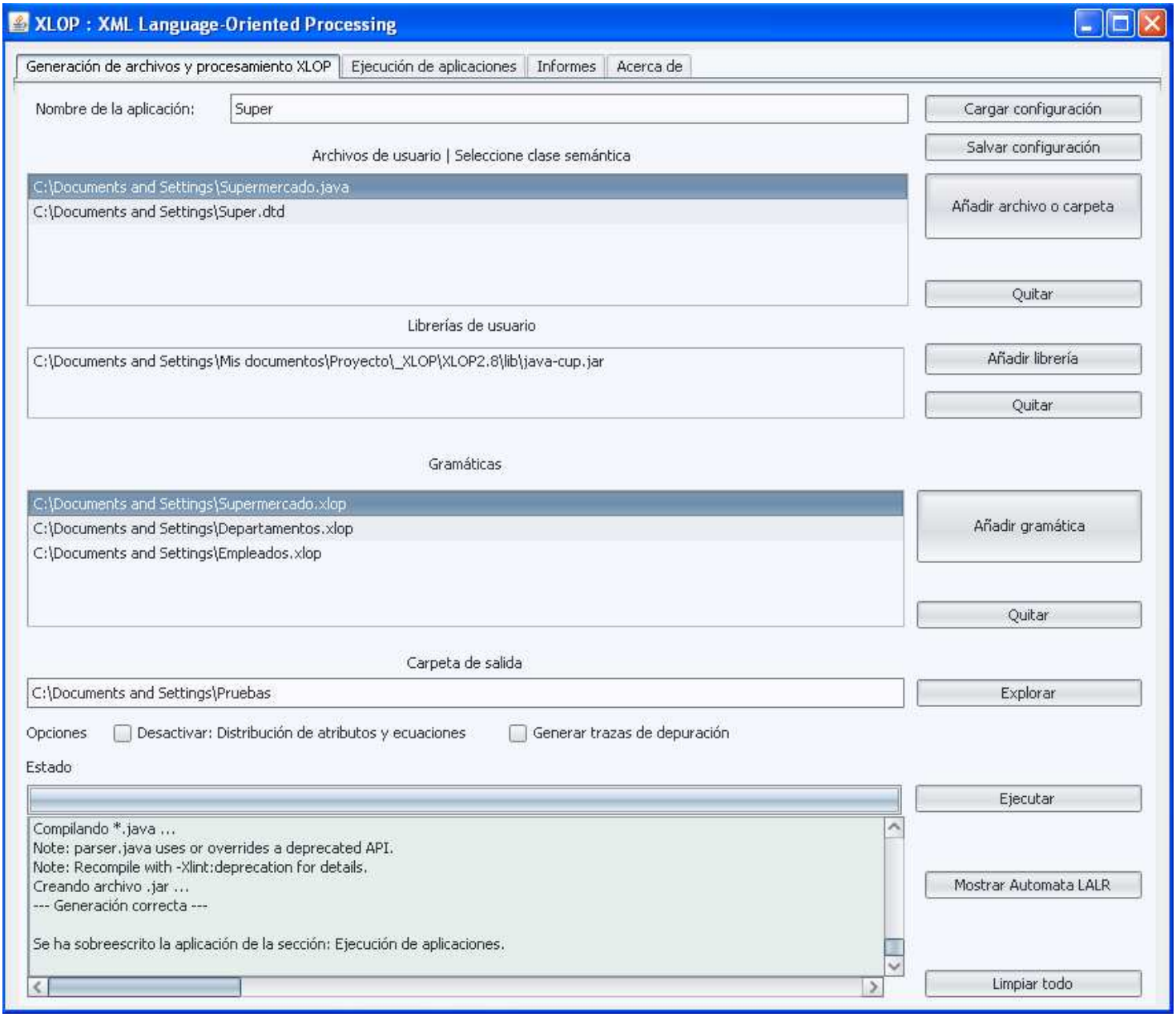

Figura 4.4 Configuración de XLOP para la gramática "Supermercado"

Como podemos observar en la figura 4.4 se han cargado las gramáticas "Supermercado.xlop", "Departamentos.xlop" y "Empleados.xlop". Además se ha dejado marcada la gramática "Supermercado.xlop" por contener el axioma de la gramática final.

Para acabar este ejemplo, y con objeto de ver el efecto de las instrucciones antes mencionadas de forma más clara, esta es la gramática final una vez analizados todos los módulos de la gramática:

```
supermercado.Supermercado = <supermercado> <departamentos>
                            departamento.Departamentos </departamentos> <empleados>
                            empleado.Empleados </empleados> </supermercado> {
  depTotal of supermercado.Supermercado = imprime(resultado of
                                          departamento.Departamentos)
  empTotal of supermercado.Supermercado = imprime(resultado of empleado.Empleados)
}

departamento.Departamentos = departamento.Departamentos departamento.Departamento {
  resultado of departamento.Departamentos = concatenaDep(
                                    resultado of departamento.Departamentos(1),
                                      resultado of departamento.Departamento)
}

departamento.Departamentos = departamento.Departamento {
  resultado of departamento.Departamentos = resultado of departamento.Departamento
}

departamento.Departamento = <nombre> #pcdata </nombre> {
  resultado of departamento.Departamento = text of #pcdata
}
empleado.Empleados = empleado.Empleados empleado.Empleado {
  resultado of empleado.Empleados = concatenaEmp(
                                  resultado of empleado.Empleados(1),
                                  resultado of empleado.Empleado)
}
```

```
empleado.Empleados = empleado.Empleado {
   resultado of empleado.Empleados = resultado of empleado.Empleado
}

empleado.Empleado = <empleado> <dni> empleado.DNI </dni>
                              <nombre> empleado.Nombre </nombre> </empleado> {
   resultado of empleado.Empleado = concatenaDNI(
                                                dni of empleado.DNI,nombre of empleado.Nombre)
}

empleado.DNI = #pcdata {
   dni of empleado.DNI = text of #pcdata
}

empleado.Nombre = #pcdata {
   nombre of empleado.Nombre = text of #pcdata
}
```

Capítulo 5

Ejemplo de aplicación de la nueva versión de XLOP

5.1 Introducción

Para conseguir que la ejecución de XLOP sea mucho más eficiente, se crean unos algoritmos de limpieza y optimización. Estos algoritmos eliminan reglas que no se pueden alcanzar y reglas que no son productivas. Esto también viene motivado porque ahora XLOP permite modularizar gramáticas, lo que puede producir reglas no alcanzables o no productivas.

Además de estos algoritmos también se cambia el algoritmo de primeros que tenía la versión anterior de XLOP. Este cambio es motivado porque este nuevo algoritmo es mucho más eficiente que el anterior.

5.2 Algoritmo de no alcanzables

El algoritmo de no alcanzables es un sencillo algoritmo de punto fijo. Consiste en construir una lista de no terminables "alcanzables" desde el axioma. Por lo tanto el algoritmo comienza creando la lista de "alcanzables" y añadiendo el axioma de la gramática. Se comienza analizando la regla del axioma y se añaden a la lista de "alcanzables" todos los no terminales del cuerpo de la regla del axioma. Es decir, si la regla es:

Axioma -> α

Se añadirán todos los no terminales de α a la lista de "alcanzables".

Después de esto se analiza la regla cuya cabeza es el siguiente no terminal que hay en la lista de "alcanzables". Y se repite el mismo proceso que se ha llevado acabo arriba hasta que la lista de "alcanzables" no cambie. He aquí el pseudocódigo de este algoritmo:

Alcanzables := {Axioma}
Repetir
Para cada producción A -> α, tal que A esté en Alcanzables
Añadir cada no terminal en α a Alcanzables
hasta que Alcanzables no cambie

5.3 Algoritmo de no productivos

El algoritmo de no productivos consiste en localizar las reglas que no sean productivas, es decir, aquellas reglas que no producen reglas nuevas. El algoritmo es muy parecido al algoritmo de no alcanzables explicado en el apartado anterior. Consiste en crear una lista de "productivos" donde se van a ir guardando los no terminales que sean productivos.

El algoritmo consiste en ir analizando las reglas, comprobar que los no terminales de su cuerpo están en la lista de "productivos" y, si es así, añadir la cabeza de la regla a la lista de "productivos". El algoritmo acaba cuando ya no se modifica la lista de "productivos" Este es el pseudocódigo de dicho algoritmo:

Productivos := {}
Repetir
Para cada producción A -> α, tal que todos los no terminales de α estén en Productivos
Añadir A a Productivos
hasta que Productivos no cambie

5.4 Algoritmo de primeros

El calculo de los símbolos primeros se calcula a través de los métodos privados *computeFirst(X)* y *primeros ()* de la clase *Func.* La información se almacena en una tabla llamada **primeros,** en la que para cada símbolo de la gramática se le asocia un conjunto de símbolos primeros. De esta forma, el símbolo primero asociado a un terminal será el propio terminal. De las misma manera los símbolos primeros asociados a un no terminal α serán todos aquellos símbolos por los que puede comenzar una forma sentencial derivable a partir de α.

El método *computeFirst(X),* donde *X* es una cadena que alberga símbolos terminales y no terminales, devuelve el conjunto de símbolos que componen los primeros de *X*. Así, si *X* es una cadena de longitud uno, *computeFirst(X)* devolverá el contenido de **primeros(X)**. El pseudocódigo de este método se muestra a continuación:

```
Función computeFirst(X: cadena) devuelve array[String]
   Var
      i,k:entero; result: array[String]
comienzo
     k= longitud(X)
     si k=0 entonces result= [λ]
     sino
         Result=primeros(X[1]) – { λ}
         i=1
         mientras i<k ^ λ ϵ primeros(x[i]) hacer
           i=i+1
           Result=Result U primeros(X[i]) – { λ}
         fin while
         si i=k ^ λ ϵ primeros(x[k]) entonces
            Result=Result U { λ}
         fin si
     fin si
     retorna Result
fin Funcion
```

El método *primeros ()* se encarga de inicializar la tabla **primeros** y guía la ejecución del algoritmo de primeros. Este algoritmo opera iterativamente, primero sobre producciones con un único símbolo en el cuerpo de la producción y luego sobre todo el conjunto de producciones de la gramática. En el siguiente pseudocódigo se ilustra el funcionamiento descrito donde **noTerminales** es la lista con todos los símbolos no terminales de la gramática, **terminales** es la lista de todos los símbolos terminales de la gramática, **derivaLambda** es una tabla que indica si desde un determinado símbolo es posible derivar lambda.

```
Procedimiento primeros ()
   Var
      A,a: cadena
comienzo
   para A en noTerminales hacer
       si derivaLambda[A] entonces
            primeros[A]={λ}
       sino
            primeros[A ]=Ø
       fin si
   fin para
   para a en terminales hacer
        primeros[a]={a}
        para A en noTerminales hacer
             si existe una producción A → a… entonces
                 primeros[A] = primeros[A]  U  {a}
             fin si
         fin para
    fin para
    repetir
       para p en producciones hacer
         primeros[cabeza(p)] = primeros[cabeza(p)] U computeFirst(cuerpo(p))
       fin para
   hasta que no haya cambios
fin procedimiento
```

Capitulo 6

Ejemplo de aplicación de la nueva versión de XLOP: XTrivial

6.1. Trivial, el juego original

Trivial Pursuit es un juego de mesa donde el avance está determinado por la habilidad del jugador para contestar preguntas sobre conocimientos generales. Scott Abbott, un editor deportivo del diario Canadian Press y Chris Haney, fotógrafo de la revista Montreal Gazette, desarrollaron la idea en diciembre de 1979; su juego fue lanzado al mercado dos años después [Wikipedia,2010].

En Estados Unidos, el juego se hizo popular en 1984, año en que se vendieron alrededor de 20 millones de ejemplares. Hasta 2009, se han vendido cerca de 100 millones de ejemplares, en 33 países y 19 idiomas [Agencia, 2010].

6.2. Trivial en el entorno XLOP

Partiendo de las posibilidades que brinda la nueva versión de XLOP, se ha desarrollado una versión del juego de mesa Trivial que aprovecha la versatilidad que ofrece el modulado de gramáticas y el sistema de espacios de nombres.

El objetivo era realizar una adaptación del juego de mesa Trivial en la cual distintas gramáticas representan distintos tipos de preguntas o mensajes de error. Cada trozo de gramática que representa un estilo de pregunta diferente es tratado de una manera concreta por la aplicación. Para este ejemplo hemos propuesto tres tipos distintos de preguntas que serán explicados con detalle en el presente capítulo.

Por otra parte, el contenido textual de las preguntas y los mensajes de error se encuentran en el archivos XML que se usa para confeccionar la aplicación, lo cual convierte el programa *XTrivial* en una herramienta con una gran adaptabilidad ya que ofrece la posibilidad de cambiar los tipos y contenidos de las preguntas que forman parte del juego en todo momento.

6.3. Documento XML y la DTD

La información necesaria para instanciar la aplicación *XTrivial* se obtiene mediante un archivo XML con el formato adecuado. Esto es así gracias a que *XTrivial* tiene un lenguaje de marcado basado en XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE trivial SYSTEM "trivialxml.dtd">
<trivial num="6">
    <Features>
        <Feature name="PrevioError">Ha de elegir un número de jugadores</Feature>
        <Feature name="NombresError">Error al introducir el nombre</Feature>
    </Features>
    <questions>
        <question tema="Literatura">
            <statement>
                    <text>¿Quién escribió "Quién te ha visto y quién te vé"?</text>
            </statement>
            <answers>
                    <text>miguel hernandez</text>
            </answers>
        </question>
        <questionOptions tema="Ciencia">
            <statement>
                    <text>¿Cuál es el nombre científico del apio?</text>
            </statement>
            <answers>
                    <option estado="correcta">Apium graveolens</option>
                    <option estado="falsa">Apium sativa</option>
                    <option estado="falsa">Apium emens</option>
                    <option estado="falsa">Mens sano</option>
            </answers>
        </questionOptions>
        <questionImageOptions tema="Geografia">
            <statement>
                    <text>¿Cuál es la bandera de Finlandia?</text>
            </statement>
            <answers>
                    <optionImage estado="correcta">imgs/finlandia.jpg</optionImage>
                    <optionImage estado="falsa">imgs/Francia.jpg</optionImage>
                    <optionImage estado="falsa">imgs/republicacheca.jpg</optionImage>
                    <optionImage estado="falsa">imgs/RepublicaDominicana.jpg</optionImage>
            </answers>
        </questionImageOptions>
        </questions>
</trivial>
```

Figura 6.3.1 Fragmento archivo XML de XTrivial

La figura 6.3.1 muestra un fragmento de archivo XML usado para la ejecución de *XTrivial*. En las primeras líneas se aprecia el prólogo, en el cual se establece la versión de XML que se está usando. En este caso es la versión 1.0 con el conjunto de caracteres ISO 8859-1 que es una norma ISO que define la codificación del alfabeto latino, incluyendo los diacríticos (como letras acentuadas, ñ, ç).

En la segunda línea se establece el tipo de documento, es decir el DTD utilizado. Para este ejemplo se utiliza el DTD "trivialXML.dtd".

Bajo el prólogo se localiza la etiqueta <trivial>, en la cual se encuentra anidada, con una estructura determinada, la información que compone las preguntas de XTrivial y los mensajes de error que se muestran al usuario.

En primer lugar, en el primer nivel de anidamiento se observa la etiqueta <Features> que establece la sección donde se encuentran los mensajes de error que se muestran al usuario. Cada mensaje se establece en cada etiqueta <Feature> que se encuentra en un nivel de anidamiento superior a <Features>. El atributo name de la etiqueta de apertura <Feature> indica el nombre del mensaje de error y el mensaje en sí se encuentra entre las etiquetas de apertura y cierre </Feature>.

En el mismo nivel de anidamiento que <Features> se encuentra la etiqueta <questions>. Esta etiqueta instaura el comienzo de la declaración de las preguntas de XTrivial. Para esta versión de XTrivial se ha desarrollado tres tipos distintos de preguntas. Cada tipo viene identificado por una etiqueta distinta. Aunque se traten de tipos de preguntas diferentes, tienen algunos elementos en común, como el atributo tema que va asociado a la etiqueta de apertura e indica el tema de la pregunta y el elemento <statemen> que representa el enunciado de la pregunta. La diferencia se encuentra en el campo <answers>. Para la pregunta tipo <question> el campo <answers> contiene la etiqueta <text>, que establece la respuesta para la pregunta. Para este tipo de pregunta es posible incluir más de una respuesta y todas ellas serían aceptadas como validas por XTrivial durante el juego.

```
<question tema="Deporte">
        <statement>
            <text>¿Qué deportista fue el abanderado español en los Juegos Olímpicos de Barcelona?</text>
    </statement>
    <answers>
            <text>principe felipe</text>
            <text>felipe de borbon</text>
            <text>principe felipe de borbon</text>
    </answers>
</question>
```

Figura 6.3.2 Ejemplo de respuesta múltiple en pregunta <question>

En la figura 6.3.2 se puede ver un ejemplo de pregunta del tipo <question> con múltiples respuestas. Esta característica es útil ya que ofrece una solución a preguntas que pueden plantear dudas acerca de cuál es la mejor forma de responderlas.

Otro tipo de pregunta es <questionOption>. En este tipo de pregunta la etiqueta <answer> esta compuesta por cuatro etiquetas de tipo <option>. Cada etiqueta <option> lleva asociado un atributo estado que indica la respuesta correcta.

```
<questionOptions tema="Ciencia">
        <statement>
            <text>¿Cuál es el nombre científico del apio?</text>
    </statement>
    <answers>
            <option estado="correcta">Apium graveolens</option>
            <option estado="falsa">Apium sativa</option>
            <option estado="falsa">Apium emens</option>
            <option estado="falsa">Mens sano</option>
    </answers>
</questionOptions>
```

Figura 7.3.3 Ejemplo de pregunta <questionOptions>

Por último, se tiene el tipo de pregunta <questionImageOptions>. En este tipo de preguntas la etiqueta <answer> contiene cuatro etiquetas de tipo <option> como en el tipo anterior, y al igual que en <questionOption> cada una de las respuestas tiene un atributo estado que indica cual es la verdadera. La diferencia se encuentra en que en este tipo de pregunta el texto es la ruta para una imagen que XTrivial carga para mostrarle al jugador.

```
<questionImageOptions tema="Geografia">
    <statement>
        <text>¿Cuál es la bandera de Finlandia?</text>
    </statement>
    <answers>
        <optionImage estado="correcta">imgs/finlandia.jpg</optionImage>
        <optionImage estado="falsa">imgs/Francia.jpg</optionImage>
        <optionImage estado="falsa">imgs/republicacheca.jpg</optionImage>
        <optionImage estado="falsa">imgs/RepublicaDominicana.jpg</optionImage>
    </answers>
</questionImageOptions>
```

Figura 6.3.4 Ejemplo de pregunta <questionImageOptions>

Por último en la imagen 6.3.5 se muestra el DTD que define la estructura y las restricciones de los archivos XML usados por XTrivial.

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
TODO define vocabulary identification
PUBLIC ID: -//vendor//vocabulary//EN
SYSTEM ID: http://server/path/Trivialxml.dtd
-->
<!--- Put your DTDDoc comment here. -->
<!ELEMENT trivial (questions|Features)*>
<!ATTLIST trivial
num CDATA #IMPLIED
>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT Features (Feature)*>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT Feature (#PCDATA)>
<!ATTLIST Feature
name CDATA #IMPLIED
>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT questions (questionImageOptions|questionOptions|question)*>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT question (answers|statement)*>
<!ATTLIST question
tema CDATA #IMPLIED
>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT statement (text)*>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT text (#PCDATA)>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT answers (optionImage|option|text)*>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT questionOptions (answers|statement)*>
<!ATTLIST questionOptions
tema CDATA #IMPLIED
>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT option (#PCDATA)>
<!ATTLIST option
estado CDATA #IMPLIED
>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT questionImageOptions (answers|statement)*>
<!ATTLIST questionImageOptions
tema CDATA #IMPLIED
>
<!--- Put your DTDDoc comment here. -->
<!ELEMENT optionImage (#PCDATA)>
<!ATTLIST optionImage
estado CDATA #IMPLIED
>
```

Figura 6.3.5 DTD estructura archivo XML de XTrivial

6.4. Gramática XLOP para XTrivial

En el presente apartado, se va a explicar con detalle, la gramática desarrollada para procesar el archivo XML, que contiene la información necesaria para instanciar XTrivial. Así mismo, también se va a hablar de la clase semántica que acompaña a la gramática anteriormente comentada.

6.4.1. Estilo de la clase semántica

La clase semántica desarrollada para *XTrivial*, ha sido pensada para que mantenga información necesaria para el juego. Esta información es almacenada en forma de atributos, que son gestionados de forma adecuada mientras se procesa el archivo XML que representa el juego.

Estos atributos son:

- **trivial**. Instancia del tipo Juego.
- **features**. Tabla que almacena elementos del tipo Feature.
- **preguntas**. Tabla que almacena elementos del tipo Pregunta.
- **contador**. Atributo de tipo entero que sirve para enumerar las preguntas que se procesan.

Además de estos atributos y las funciones semánticas oportunas para manejarlos, en la especificación se hace uso de la función semántica after(p0,p1) la cual devuelve siempre el valor del segundo argumento. Debido a que la evaluación de expresiones en XLOP es puramente aplicativa, esta función permite introducir dependencias entre los procesamientos que generan cambio en los atributos de la clase semántica.

6.4.2. Estructura de la gramática

La gramática de *XTrivial* está compuesta por trozos o módulos de gramática individuales. Cada trozo tiene un espacio de nombres propio que diferencia sus terminales y no terminales de los pertenecientes a otros módulos. *XTrivial* cuenta con cinco módulos distintos. En primer lugar se tiene el modulo "trivial.xlop", el cual se trata del módulo principal de la gramática. En el se instancia *XTrivial* y se hace referencia a los otros cuatro módulos. En segundo lugar está el módulo "features.xlop". En este modulo se encuentran las ecuaciones necesarias para procesar los mensajes de error que se tienen en el archivo XML.

Los trozos de gramática restantes se tratan de los módulos que definen la estructura de los distintos tipos de preguntas que usa XTrivial. Estos módulos son "TextQuestions.xlop", "OptionQuestions.xlop" y "OptionImageQuestions.xlop".

6.4.2.1. Gramática principal

La gramática principal se encuentra en el archivo "trivial.xlop". Como se aprecia en la figura 6.4.1 en la primera línea se declara el espacio de nombres del módulo como trivial, y se cualifica los no terminales que hacen referencia a los trozos de gramática en los que se especifica los distintos tipos de preguntas. Nótese que la cualificación del no terminal Features se hace de forma concreta.

En la primera ecuación se puede observar el uso de la función after(..) anteriormente mencionada a la hora de procesar los elementos de tipo Feature y la creación del juego. En este caso se fuerza a que primero se procesen todos los elementos de tipo Feature y posteriormente se instancie el juego mediante la función semántica newTrivial(num of <trivial>) en la cual se pasa como parámetro el número de temas del que constará el juego. De la misma forma en esa misma ecuación se procesa el conjunto de elementos del tipo Question antes de que se ejecute el juego con la función semántica run() mediante el uso de after(..).

En el resto de ecuaciones de este modulo se desciende por la gramática procesando e instanciado mediante la función correspondiente los elementos de clase Question.

```
namespace trivial;
qualify OptionsDescription as OptionQuestions.OptionsDescription;
qualify QuestionDescription as TextQuestions.QuestionDescription;
qualify QuestionImageDescription as OptionImageQuestions.QuestionImageDescription;

Trivial ::= <trivial> <Features> FeaturesTrivial.Features </Features> <questions> Questions </questions> </trivial>{
       trivialCreatedh of Questions = after(featuresStored of FeaturesTrivial.Features, newTrivial(num of <trivial>))
       trivialExecuted of Trivial = after(questionsCreated of Questions,run())
}

Questions ::=  Questions Question{
       trivialCreatedh of Questions(1) = trivialCreatedh of Questions(0)
       trivialCreatedh of Question = trivialCreatedh of Questions(0)
       questionsCreated of Questions(0) = after(questionsCreated of Questions(1), questionCreated of Question)
}

Questions ::= Question{
       trivialCreatedh of Question = trivialCreatedh of Questions
       questionsCreated of Questions = questionCreated of Question

}

Question ::= QuestionDescription {
       trivialCreatedh of QuestionDescription = trivialCreatedh of Question
       questionCreated of Question = newQuestionText(tema of QuestionDescription,
                                                     enunciado of QuestionDescription,
                                                     respuesta of QuestionDescription)
}

Question ::=  OptionsDescription {
       trivialCreatedh of OptionsDescription = trivialCreatedh of Question
       questionCreated of Question = newQuestionOptions(tema of OptionsDescription,
                                                        enunciado of OptionsDescription,
                                                        respuesta of OptionsDescription)
}

Question ::=  QuestionImageDescription {
       trivialCreatedh of QuestionImageDescription = trivialCreatedh of Question
       questionCreated of Question = newQuestionOptionsImage(tema of QuestionImageDescription,
                                                             enunciado of QuestionImageDescription,
                                                             respuesta of QuestionImageDescription )
}
```

Figura 6.4.1 Gramática "trivial.xlop"

6.4.2.2. Gramática Features

La gramática que procesa los elementos de tipo Features se encuentra en el archivo "Features.xlop". En esta gramática se hace uso de la función after(..) para indicar el orden en el que ha de ser almacenada la información de tipo Feature en el atributo features de la clase semántica. También se ha de notar el uso de la función semántica addFeature(..) mediante la cual se introduce una nueva característica a la clase semántica. En la figura 6.5.2.2 se puede observar este módulo de la gramática.

```
namespace FeaturesTrivial;
Features ::= Features Feature {
       featuresStored of Features(0) = after(featuresStored of Features(1),
                                             featureStored of Feature)
}

Features ::= Feature {
       featuresStored of Features = featureStored of Feature
}

Feature ::= <Feature> #pcdata </Feature> {
       featureStored of Feature = addFeature(name of <Feature>,
                                             text of #pcdata)
}
```

Figura 6.4.2 Gramática "Features.xlop"

6.4.2.3. Gramática TextQuestion

Esta gramática se localiza en el archivo "TextQuestion.xlop". Este trozo de la gramática procesa las preguntas de tipo QuestionDescripcion que se caracterizan por tener un enunciado y una o más respuestas validas.

Como se aprecia en la imagen 6.4.3, en la primera ecuación se almacenan los elementos tema, enunciado y respuesta en los atributos de QuestionDescripcion, que más tarde serán usados en la gramática principal, para guardar la pregunta de forma adecuada mediante la función semántica newQuestionText(..) en el atributo preguntas de la clase semántica.

Hay que destacar dos puntos en la gramática. Razonar como se procesa este módulo y como se consigue almacenar la pregunta como tal en el atributo preguntas de la clase semántica.

En primer lugar observar la penúltima ecuación donde se llama a la función semántica newAnswerText(..). Esta función introduce el elemento que se le pasa por variable, que no es otra cosa que una respuesta para la pregunta que se está procesando, en una lista, que es lo que devuelve la función semántica.

Esa lista queda almacenada como el atributo elem of Answers y posteriormente es usada por la función semántica combineAnswersText(..) en la tercera ecuación de la gramática. Con esta función se obtiene un resultado similar al que proporciona la función after(..) anteriormente explicada, ya que la dependencia entre procesamientos garantiza que se procesará en primer lugar el primer argumento de la función. En este caso el primer argumento se trata de una lista de elementos que representan respuestas para la pregunta, mientras que el segundo elemento es otra respuesta válida. Como resultado, la función devuelve una lista con todas las respuestas, que queda almacenada en el atributo elem del No Terminal Answers(0) de la ecuación. Cuando haya finalizado el procesamiento de todos los elementos de la pregunta ese elemento queda almacenado en el atributo respuesta de QuestionDescripcion como se aprecia en la primera ecuación de la gramática.

Por otra parte el atributo enunciado del No Terminal QuestionDescription en la primera ecuación se obtiene del elemento elem del No Terminal Statement, mientras que el elemento tema se obtiene a partir del atributo tema de la etiqueta <question>.

```
namespace TextQuestions;

QuestionDescription ::= <question> <statement>Statement</statement><answers> Answers </answers> </question>{
	trivialCreatedh of Statement = trivialCreatedh of QuestionDescription
	trivialCreatedh of Answers = trivialCreatedh of QuestionDescription
	tema of QuestionDescription = tema of <question>
	enunciado of QuestionDescription = elem of Statement
	respuesta of QuestionDescription = elem of Answers
}

Statement ::= <text> #pcdata </text> {
	elem of Statement = after(trivialCreatedh of Statement,text of #pcdata)
}

Answers ::= Answers Answer{
	trivialCreatedh of Answers(1) = trivialCreatedh of   Answers(0)
	trivialCreatedh of Answer = trivialCreatedh of   Answers(0)
	elem of Answers(0) = combineAnswersText(elem of Answers(1), elem of Answer)
}

Answers ::= Answer{
	trivialCreatedh of Answer= trivialCreatedh of   Answers
	elem of Answers = newAnswerText(elem of Answer)
}

Answer ::= <text> #pcdata </text> {
	elem of Answer= after(trivialCreatedh of Answer, text of #pcdata)
}
```

Figura 6.4.3 Gramática TextQuestions

6.4.2.4. Gramática OptionQuestion

El módulo que procesa esta fracción de gramática se encuentra en el archivo “OptionQuestions.xlop”. Las ecuaciones contenidas en este trozo de la gramática se encargan de procesar las preguntas de tipo OptionQuestion, que se caracterizan por tener un enunciado y cuatro respuestas de las cuales sólo una es válida. Como se aprecia en la imagen 6.4.4 en la primera ecuación se almacenan los elementos tema, enunciado y respuesta en los atributos de OptionDescription, que más tarde serán usados en la gramática principal para crear la pregunta de tipo QuestionText.

El procesamiento que hace este módulo de la gramática es similar al que hace la gramática TextQuestion. Hay un punto en la gramática en el cual se llama a la función semántica newAnswerOption(..) a la cual se le pasa dos argumentos. El primero es una respuesta a la pregunta y el segundo un atributo que indica si la respuesta que lo acompaña es válida o no. Como resultado, la función semántica devuelve una tabla, en la cual la clave es la respuesta a la pregunta y el valor asociado es un atributo booleano, que indica si dicha respuesta es correcta. Esta tabla es almacenada como atributo elem de OptionAnswers.

Posteriormente, este atributo es usado por la función semántica combineAnswersOptions(..) que se localiza en la tercera ecuación de la gramática. Al igual que la función combineAnswersText(..) del módulo TextQuestions, esta función aprovecha la dependencia de procesamientos, para que se obtenga en primer lugar el atributo elem del No Terminal OptionAnswers(1), el cual se trata de una tabla compuesta por las respuestas procesadas hasta el momento, con la estructura comentada en el párrafo anterior. De este modo a la función combineAnswersOptions(..) se le pasa como parámetros una tabla con las respuestas procesadas hasta el momento, una respuesta y un atributo que indica si la respuesta que lo acompaña como parámetro es cierta o no. Como resultado la función devuelve la tabla pasada como primer argumento con la respuesta introducida en segundo termino. Esta tabla se almacena como el atributo elem de OptionAnswers(0) y se va pasando como elemento sintetizado entre los No Terminales OptionAnswer. Finalmente esa tabla se guarda en el atributo respuesta de OptionDescription y en el módulo principal es usado para instanciar una pregunta de tipo QuestionOptions.

Finalmente los atributos enunciado y tema se obtienen de forma similar a como se adquieren en la gramática TextQuestion.

```
namespace OptionQuestions;

OptionsDescription ::= <questionOptions> <statement>Statement</statement>
                       <answers> OptionAnswers </answers> </questionOptions>{
      trivialCreatedh of Statement = trivialCreatedh of OptionsDescription
      trivialCreatedh of OptionAnswers= trivialCreatedh of OptionsDescription
      tema of OptionsDescription = tema of <questionOptions>
      enunciado of OptionsDescription = elem of Statement
      respuesta of OptionsDescription = elem of OptionAnswers
}

Statement ::= <text> #pcdata </text> {
      elem of Statement = after(trivialCreatedh of Statement,text of #pcdata)
}

OptionAnswers ::= OptionAnswers Option{
      trivialCreatedh of OptionAnswers(1) = trivialCreatedh of OptionAnswers(0)
      trivialCreatedh of Option = trivialCreatedh of OptionAnswers(0)
      elem of OptionAnswers(0) = combineAnswersOptions(elem of OptionAnswers(1),
                                                       elem of Option,estado of Option)
}

OptionAnswers ::= Option{
      trivialCreatedh of Option= trivialCreatedh of OptionAnswers
      elem of OptionAnswers = newAnswerOptions(elem of Option,estado of Option)
}

Option ::= <option> #pcdata </option> {
      elem of Option= after(trivialCreatedh of Option, text of #pcdata)
      estado of Option = estado of <option>
}
```

Figura 6.4.4. Gramática OptionsQustions

6.4.2.5. Gramática OptionImageQuestions

Esta fracción de la gramática se encuentra en el fichero "OptionImageQuestions.xlop". Con esta gramática, se procesa preguntas que tienen un enunciado y cuatro respuestas al igual que el módulo OptionQuestion, con la salvedad de que en este tipo de preguntas las respuestas son rutas de otros archivos. Aun así, no son otra cosa que cadenas de texto que llevan asociado un atributo que indica cual es la respuesta verdadera, lo cual hace que el procesamiento de este tipo de preguntas sea muy similar al de las preguntas OptionImage.

Al igual que en el modulo OptionImage se almacena las respuestas en una tabla junto con un booleano que indica si es la respuesta correcta mediante la función semántica newAnswerOptionsImage(..). Las respuestas se combinan mediante la función combineAnswersOptionImage(..) que se localiza en la tercera ecuación de la gramática, y la tabla que esta función devuelve se pasa como atributo sintetizado elem al No Terminal OptionImageAnswer(0). Finalmente este atributo sintetizado se almacena como respuesta de QuestionImageDescription. El resto de atributos necesarios para instanciar la pregunta y que también se almacenan en el No Terminal QuestionImageDescription se obtienen de la misma forma que en los otros trozos de gramática que se encargan de procesar preguntas.

```
namespace OptionImageQuestions;

QuestionImageDescription ::= <questionImageOptions> <statement>Statement</statement>
                             <answers> OptionImageAnswers </answers> </questionImageOptions>{
      trivialCreatedh of Statement = trivialCreatedh of QuestionImageDescription
      trivialCreatedh of OptionImageAnswers= trivialCreatedh of QuestionImageDescription
      tema of QuestionImageDescription = tema of <questionImageOptions>
      enunciado of QuestionImageDescription = elem of Statement
      respuesta of QuestionImageDescription = elem of OptionImageAnswers
}

Statement ::= <text> #pcdata </text> {
      elem of Statement = after(trivialCreatedh of Statement,text of #pcdata)
}

OptionImageAnswers ::= OptionImageAnswers OptionImage{
      trivialCreatedh of OptionImageAnswers(1) = trivialCreatedh of OptionImageAnswers(0)
      trivialCreatedh of OptionImage = trivialCreatedh of OptionImageAnswers(0)
      elem of OptionImageAnswers(0) = combineAnswersOptionsImage(elem of OptionImageAnswers(1),
                                                                 elem of OptionImage,
                                                                 estado of OptionImage)
}

OptionImageAnswers ::= OptionImage{
      trivialCreatedh of OptionImage= trivialCreatedh of OptionImageAnswers
      elem of OptionImageAnswers = newAnswerOptionsImage(elem of OptionImage,estado of OptionImage)
}

OptionImage ::= <optionImage> #pcdata </optionImage> {
      elem of OptionImage= after(trivialCreatedh of OptionImage, text of #pcdata)
      estado of OptionImage = estado of <optionImage>
}
```

uestionImageQuestions

6.4.3. La clase semántica

En el presente apartado se va a hablar de la clase semántica que, junto con las gramáticas ya comentadas y XLOP, permite que se instancie la aplicación XTrivial a partir de un fichero XML con la información adecuada.

Como ya se ha comentado en el apartado 6.4.1, además de las funciones que son llamadas desde los trozos de gramáticas que procesan el archivo XML, la clase semántica contiene ciertos atributos que mantienen información mientras se procesa el archivo XML.

6.4.3.1. Atributos

En la figura 6.4.6 se puede apreciar dichos atributos. Se puede observar una tabla features, donde se almacena los mensajes de error que se cargan mediante el fichero XML en XTrivial. A medida que se van procesando estas características se introducen en esta tabla que más tarde se pasa al juego cuando esté es instanciado. También se puede ver una tabla preguntas en la cual la clave es el tema de la pregunta y el valor asociado la propia pregunta. Además se tiene un atributo de tipo entero bajo el nombre de contador, en el cual se lleva la cuenta del número de preguntas que se han procesado. Sirve para asignarle un ID único a cada pregunta. Por último se tiene el atributo trivial que es de clase Juego. Este atributo representa a XTrivial.

```
public class trivialSemanticClass {

   private Hashtable<String,String> features;
   private HashtableDK<String,Pregunta> preguntas;
   private Juego trivial;
   private int contador;
```

Figura 6.4.6 Atributos de la clase semántica

6.4.3.2. Creación del juego

El juego se instancia mediante la función newTrivial(..). Esta función es llamada por el módulo principal de la gramática "trivial.xlop". Esta función se puede ver en la figura 6.4.7. Como parámetro se le pasa el String numpreg que indica el número total de temas que aparecen en el fichero XML. Cuando se instancia la variable trivial, se le pasa por parámetro el número de temas y la tabla que contiene las características.

```
public boolean newTrivial(String numPreg){
    trivial = new Juego(Integer.valueOf(numPreg).intValue(),features);
    return true;
}
```

Figura 6.4.7 Creación del juego

6.4.3.3. Inicio del Juego

La función semántica run() es llamada desde el módulo "trivial.xlop" cuando se ha terminado de procesar el fichero XML. Es la encargada de introducir las preguntas en el juego e iniciarlo.

```
public void run(){
    trivial.setPreguntas(preguntas);
    trivial.setVisible(true);
}
```

Figura 6.4.8 Función semántica run

6.4.3.4. Nueva característica

Cada vez que se encuentra una nueva característica en el fichero XML esta es procesada por el módulo "Features.xlop". Desde este módulo se llama a la función addFeature(..).

```
public boolean addFeature(String name, String text) {
   features.put(name,text);
   return true;
}
```

Figura 6.4.9 inclusión de nueva característica

6.4.3.5. Tratamiento de preguntas

Las funciones que permiten instanciar los elementos de tipo Question son newQuestionText(..), newQuestionOptions(..) y newQuestionOptionsImage(..). Estas funciones son llamadas desde el módulo "trivial.xlop" de la gramática en función de la etiqueta del fichero XML que define la pregunta.

```
<question tema="Deporte">

<questionOptions tema="Ciencia">

<questionImageOptions tema="Historia">
```

Figura 6.4.10. Ejemplo XML de tipo de pregunta

6.4.3.6. Pregunta QuestionText

Este tipo de pregunta se caracteriza por tener un tema, un enunciado, una o más respuestas válidas, y un ID único para cada pregunta. La función newQuestionText(..) es llamada desde el módulo "trivial.xlop" cuando se detecta en el archivo XML una etiqueta del tipo <question> y la pregunta instanciada se añade al atributo de la clase semántica preguntas, en el cual las preguntas se asocian en función de su tema. La lista de respuestas para esta clase de pregunta es procesada en el módulo "TextQuestions.xlop". Desde ese módulo se llama a la función newAnswerText(..), la cual devuelve una lista con una única respuesta. Esta lista es pasada como elemento sintetizado entre las respuestas de una misma pregunta, y para agregar una nueva respuesta a las ya procesadas se llama a la función combineAnswerText(..). A esta función se le pasa la lista de respuestas que se tiene hasta el momento y la respuesta que se quiere añadir. Como resultado devuelve otra lista.

```
public boolean newQuestionText(String tema, String enunciado, ArrayList<String> respuesta){
    Pregunta preg=new PreguntaText(tema,enunciado,respuesta,contador);
    preguntas.put(tema,preg);
    contador++;
    return true;
}

public ArrayList<String> combineAnswersText(ArrayList<String> respuestas, String resp){
    respuestas.add(resp);
    return respuestas;
}

public ArrayList<String> newAnswerText(String resp){
    ArrayList<String> respuestas=new ArrayList<String>();
    respuestas.add(resp);
    return respuestas;
}
```

Figura 6.4.11 Funciones procesamiento QuestionText

Por ejemplo, para esta pregunta, se descendería por el árbol generado por la gramática hasta la tercera respuesta “Príncipe Felipe de Borbón”. Con esta respuesta se crearía la lista de respuestas que se va pasando de una respuesta a otra. A continuación se introduciría en la lista la respuesta “Felipe de Borbón” y por último “Príncipe Felipe”.

```
<question tema="Deporte">
    <statement>
        <text>¿Qué deportista fue el abanderado español en los Juegos Olímpicos de Barcelona?</text>
    </statement>
    <answers>
        <text>Príncipe Felipe</text>
        <text>Felipe de Borbón</text>
        <text>Principe Felipe de Borbón</text>
    </answers>
</question>
```

Figura 6.4.12 Ejemplo de pregunta QuestionText

6.4.3.7. Pregunta QuestionOptions

Al igual que las preguntas de tipo QuestionText, esta clase de preguntas está caracterizada por un tema, un enunciado, una serie de respuestas y un ID único que identifica a cada pregunta. Esta clase de pregunta es procesada por el módulo que se encuentra en el fichero “OptionQuestions.xlop”. La diferencia radica en que en este tipo de preguntas se le ofrece al jugador cuatro respuestas diferentes por pregunta, de las cuales sólo una es válida. Las respuestas, se almacenan en una tabla en la cual la clave es la propia respuesta y su valor asociado es un booleano que indica si es verdadera o falsa. La forma de instanciar la pregunta es similar QuestionText, y la manera en la que la tabla con las respuestas se pasa de una respuesta a otra también.

```
public boolean newQuestionOptions(String tema, String enunciado, HashtableDK<String,Boolean> respuesta){
    Pregunta preg=new PreguntaOptions(tema,enunciado,respuesta,contador);
    preguntas.put(tema,preg);
    contador++;
    return true;
}

public HashtableDK<String,Boolean> combineAnswersOptions(HashtableDK<String,Boolean> respuestas, String resp,String estado){
     if(estado.equals("correcta"))
         respuestas.put(resp, true);
     else
         respuestas.put(resp, false);
    return respuestas;
}

public HashtableDK<String,Boolean> newAnswerOptions(String resp,String estado){
     HashtableDK<String,Boolean> respuestas=new HashtableDK<String,Boolean>();
     if(estado.equals("correcta"))
         respuestas.put(resp, true);
     else
         respuestas.put(resp, false);
     return respuestas;
}
```

Figura 6.4.13 Funciones procesamiento QuestionOptions

En la siguiente figura se puede ver un ejemplo de pregunta QuestionOption. Como en el tipo QuestionText el árbol generado por la gramática hace que en primer lugar se introduzca en la tabla de respuestas la solución “1812” la cual es falsa. Siguiendo la secuencia que genera el árbol en última instancia se introduce la solución “1603” la cual es verdadera.

```
<questionOptions tema="Historia">
    <statement>
        <text>¿En qué año fue descubierta la Antartida?</text>
    </statement>
    <answers>
        <option estado="correcta">1603</option>
        <option estado="falsa">1738</option>
        <option estado="falsa">1406</option>
        <option estado="falsa">1812</option>
    </answers>
</questionOptions>
```

Figura 6.4.14 Ejemplo de pregunta QuestionOptions

6.4.3.8. Pregunta QuestionOptionImage

Este tipo de preguntas se procesa mediante la gramática que se encuentra en el fichero "OptionImageQuestions.xlop". Es idéntico en estructura y método de procesamiento a las preguntas de tipo QuestionOptions. La diferencia entre esta clase de preguntas y las de QuestionOptions es que la información del fichero XML que va asociada a las repuestas no son respuestas en sí, si no nombres de archivos de tipo imagen que se le muestra al usuario.

```
public boolean newQuestionOptionsImage(String tema, String enunciado, HashtableDK<String,Boolean> respuesta){
    Pregunta preg=new PreguntaImagen(tema,enunciado,respuesta,contador);
    preguntas.put(tema,preg);
    contador++;
    return true;
}

public HashtableDK<String,Boolean> combineAnswersOptionsImage(HashtableDK<String,Boolean> respuestasI, String resp,String estado)
     if(estado.equals("correcta"))
         respuestasI.put(resp, true);
     else
         respuestasI.put(resp, false);
    return respuestasI;
}

public HashtableDK<String,Boolean> newAnswerOptionsImage(String resp,String estado){
     HashtableDK<String,Boolean> respuestasI=new HashtableDK<String,Boolean>();
     if(estado.equals("correcta"))
         respuestasI.put(resp, true);
     else
         respuestasI.put(resp, false);
     return respuestasI;
}
```

Figura 6.4.15 Funciones procesamiento QuestionOptionImage

En la figura se puede ver la estructura de la pregunta. Al igual que las preguntas QuestionOptions tiene cuatro respuestas, de las cuales sólo una es verdadera. El atributo estado de cada etiqueta <optionImage> es el que indica la veracidad de cada respuesta.

```
<questionImageOptions tema="Historia">
    <statement>
        <text>¿Cuál de éstos retratos pertenece a Felipe IV?</text>
    </statement>
    <answers>
        <optionImage estado="correcta">imgs/FelipeIV.jpg</optionImage>
        <optionImage estado="falsa">imgs/CarlosIII.jpg</optionImage>
        <optionImage estado="falsa">imgs/Fernandocatolico.jpg</optionImage>
        <optionImage estado="falsa">imgs/FernandoVII.jpg</optionImage>
    </answers>
</questionImageOptions>
```

Figura 6.4.16 Ejemplo de pregunta QuestionOptionImage

6.5. Implementación de XTrivial

En la figura 6.5.1 se puede apreciar un diagrama con las principales clases de XTrivial. Entre los atributos de la clase Juego, destacan la variable ventana de tipo GUI, preguntas de tipo HashtableDK, donde la clave es el tema de la pregunta y se puede mantener más de un valor asociado por cada clave. El valor asociado es la propia pregunta. Por último jugadores, que es un ArrayList de tipo Jugador.

La clase Pregunta es una clase abstracta de la cual heredan las clases PreguntaOptions, PreguntaText y PreguntaImagen.

La clase GUI tiene ocho variables de tipo JPanel que son referenciadas durante el juego desde el ArrayList listaPanel. Cada una de estas variables de tipo JPanel tiene un fondo distinto, lo cual le da mayor atractivo visual a la aplicación. Esto se consigue instanciado los paneles como objetos de tipo JPanelConFondo. Esta clase hereda de la clase JPanel e incluye un atributo privado imagen que es el fondo que muestra el panel. La GUI cambia de un panel a otro mediante el método activaPanel(..).

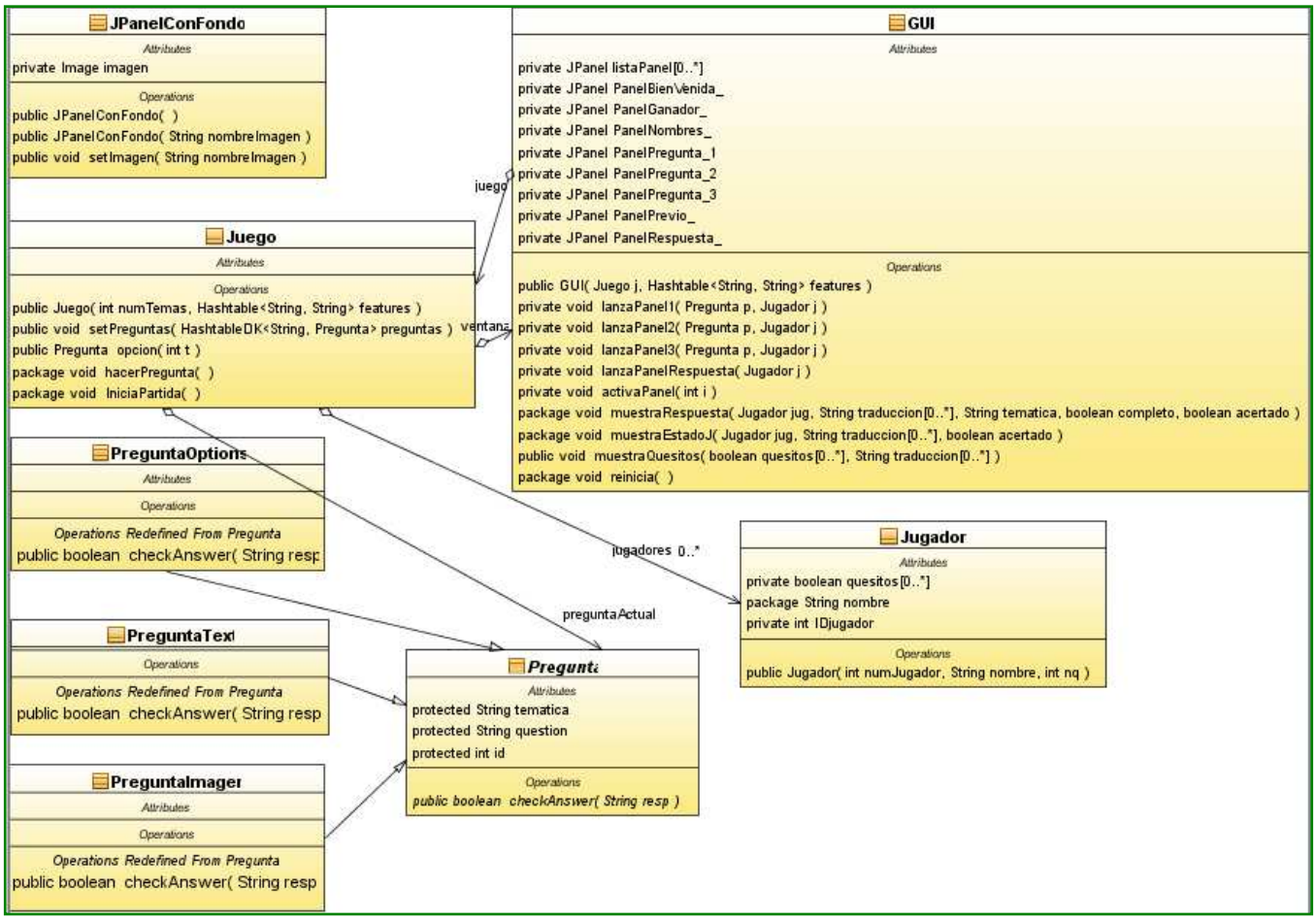

Figura 6.5.1 Esquema de XTrivial

Toda instancia de XTrivial, comienza con la creación de un objeto de tipo Juego. Este objeto almacena las preguntas y los jugadores que intervienen durante el juego. Las preguntas se inician mediante la función setPreguntas(..). Una vez que el juego tiene las preguntas se muestra la ventana inicial mediante la función setVisible(). Desde la GUI se pide el número de jugadores y sus nombres. Esta información es transferida al juego. A partir de este punto el juego estable el turno de cada jugador en función del orden con que el usuario haya introducido quien va a jugar a XTrivial. Para cada jugador, se elije una pregunta de forma aleatoria de entre las disponibles mediante la función hacerPregunta(). Ninguna pregunta es repetida si aún quedan preguntas sin hacer.

La clase GUI dispone de tres paneles distintos para formular una pregunta. Cada panel está preparado para un tipo distinto de pregunta. La respuesta del jugador es enviada al juego, el cual en función de si el jugador ha acertado o no, mostrará un mensaje de error o de acierto. En caso de acertar la pregunta, se marca en el atributo quesitos, en el tema correspondiente a la cuestión contestada.

El cambio de turno de un jugador a otro se realiza cuando se responde de forma errónea a una pregunta. Esta acción se realiza en la clase juego mediante la función cambiaTurno().

El fin de partida se alcanza cuando un jugador ha respondido al menos una pregunta de cada tema. Cuando esto sucede se le indica a la ventana que muestre el panel correspondiente. En este punto del juego se puede elegir terminar la partida o volver a jugar. En este último caso, la ventana informa al juego, el cual mediante la función iniciaPartida() vuelve a poner el juego en el estado inicial.

6.6. Generación de la aplicación con XLOP

En la imagen 6.6.1 se muestra la pestaña Generación de archivos y procesamiento XLOP, con los elementos necesarios para la generación de XTrivial. Como archivos de usuario hay que añadir la clase semántica, el archivo DTD que define la estructura del fichero XML y la carpeta donde se encuentren las imágenes necesarias para ciertas preguntas. En librerías de usuario hay que añadir el archivo Trivial.jar que contiene la lógica específica de los componentes que constituyen el marco de aplicación XTrivial. Por último en gramáticas se añade todos los módulos que componen la gramática de XTrivial. En archivos de usuario se selecciona la clase semántica y en las gramáticas el módulo “features.xlop” que se trata del módulo principal.

Tras presionar el botón ejecutar y generar la aplicación se cambia a la pestaña Ejecución de aplicaciones. En esta pestaña se selecciona el archivo XML que se desea usar y se presiona el botón ejecutar aplicación. Se puede ver una imagen de esta ventana en la figura 6.6.2.

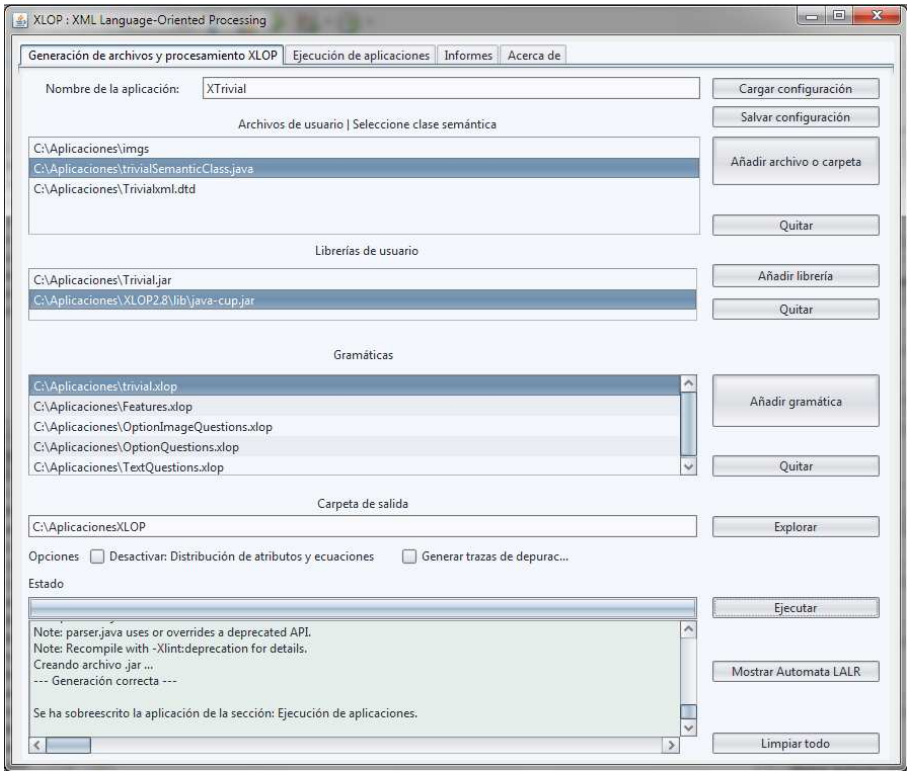

Figura 6.6.1 Generación de XTrivial con XLOP

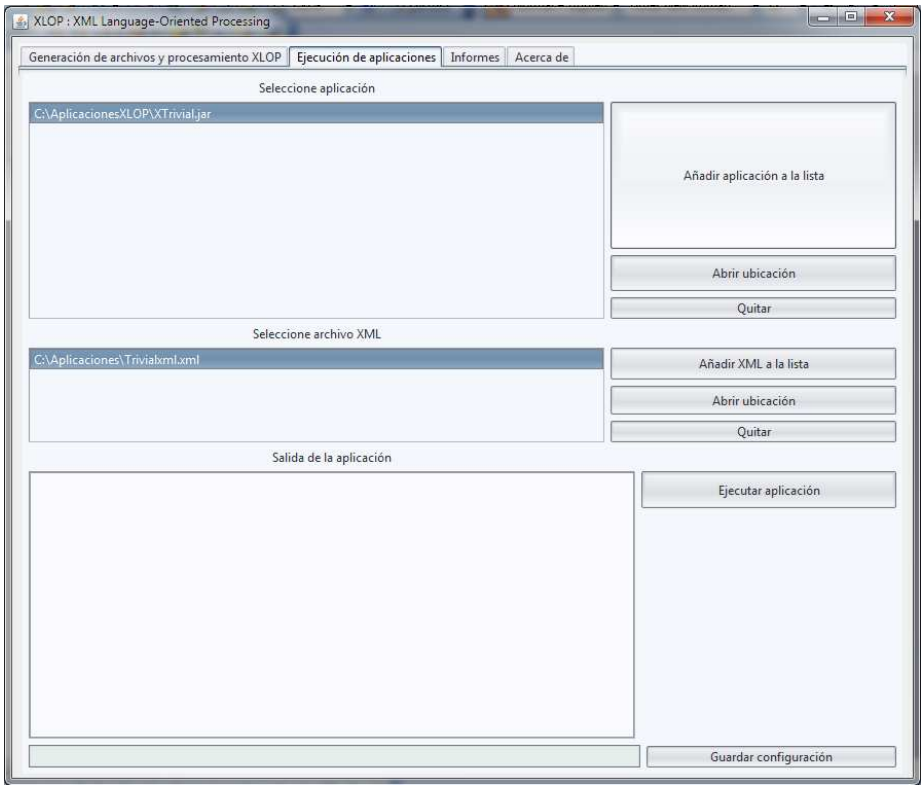

Figura 6.6.2. Ejecución de XTrivial con XLOP

6.7. Una partida con XTrivial

Al iniciar la aplicación XTrivial aparece la ventana inicial con el nombre de la aplicación. Haciendo click con el ratón en cualquier parte de la ventana se activa el siguiente panel donde se pide el número de jugadores.

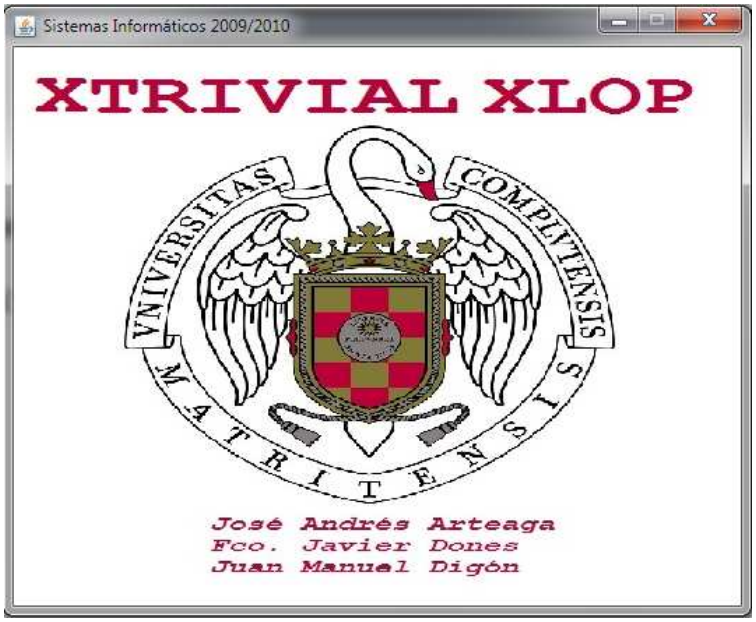

Figura 6.7.1 Venta inicial XTrivial

Tras seleccionar el número de jugadores se avanza al siguiente panel mediante el botón continuar, donde se pide el nombre de los jugadores.

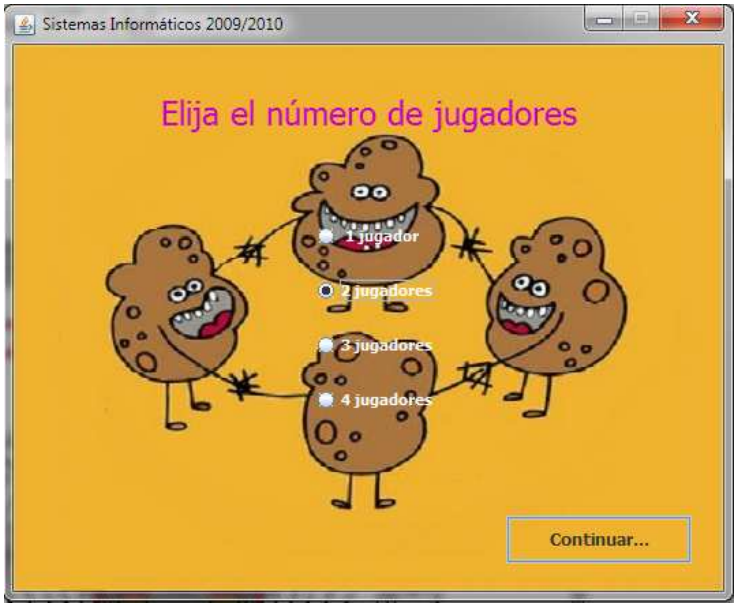

Figura 6.7.2. Selección de jugadores

Una vez introducidos los nombres, sólo hay que hacer click en el dado para comenzar a jugar.

Figura 6.7.3 Nombre de los jugadores

Existen tres tipos distintos de paneles, que se muestran en función del tipo de pregunta. Pregunta con una respuesta que ha de introducir el jugador mediante teclado, pregunta en la cual el jugador elige la respuesta entre cuatro opciones escritas, o pregunta en la que el jugador escoge la respuesta entre cuatro imágenes distintas.

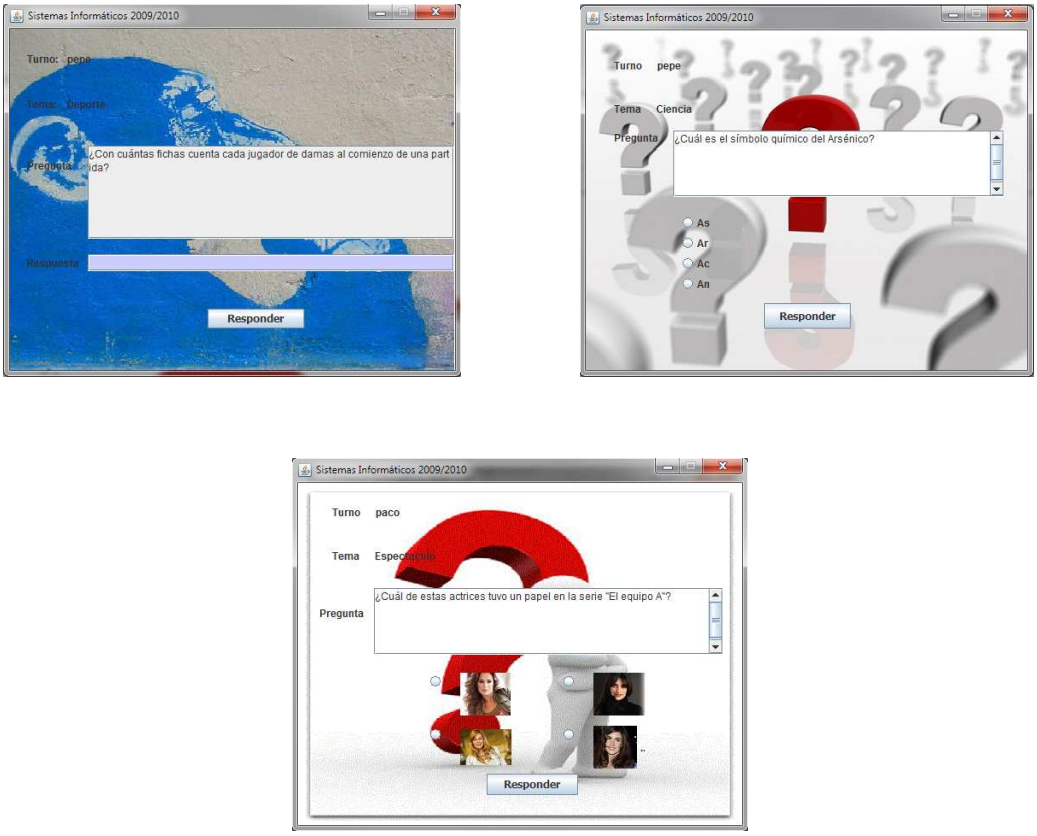

Figura 6.7.4. Los distintos paneles para cada tipo de pregunta

Cuando el jugador responde una pregunta se le muestra una ventana donde se le informa de si ha acertado o fallado la respuesta, los temas en los que a acertado y los que le faltan para terminar el juego.

Figura 6.7.5. Paneles para el acierto o el fallo de la pregunta.

Cuando un jugador consigue contestar de manera correcta una pregunta de cada tema se muestra una ventana con el nombre del ganador. Así mismo se da la opción de volver a jugar o terminar la partida.

Figura 6.7.6. Panel final para indicar el ganador.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

En este trabajo se han desarrollado unas mejoras y ampliaciones relativas a la herramienta XLOP que han dado lugar a una aplicación más potente y con un código más refinado y revisado que en la anterior versión. Cabe mencionar que el enfrentarnos a un código heredado de un proyecto de tal magnitud como XLOP ha supuesto un reto para la comprensión global del proyecto y para la integración de las nuevas funcionalidades. De hecho parte del proyecto ha consistido en la refactorización del código anterior como por ejemplo la modificación de las estructuras que se usaban para la definición del autómata LALR, adoptando una estructura mas sencilla o la revisión del algoritmo para el calculo de los elementos PRIMEROS de una gramática por otro mas eficiente.

Este trabajo nos ha permitido profundizar en el conocimiento de XML así como de la creación de aplicaciones que integren este tipo de documentos, lo que sin duda supondrá un valioso conocimiento de cara a la nuestra futura vida laboral.

También debemos mencionar que la fase del desarrollo del algoritmo de marcado nos ha permitido refrescar y ampliar nuestros conocimientos sobre la asignatura de Procesadores de Lenguaje con un tema del que no existe mucha documentación al respecto. De hecho el algoritmo implementado en esta versión ha sido íntegramente diseñado dentro de esta universidad.

Para comprobar la potencia de esta nueva versión de XLOP se ha desarrollado una aplicación no trivial llamada XTrivial la cual está basada en el popular juego de mesa Trivial. En ella se muestra una aplicación que consta de varios módulos totalmente independientes cada uno completando la aplicación XTrivial con una nueva manera de procesar un tipo de pregunta.

Finalmente estamos satisfechos por haber conseguido una versión funcional de XLOP cumpliendo todos los objetivos propuestos al principio del proyecto así como del trabajo en equipo realizado por todo el grupo.

7.2. Trabajo futuro

El trabajo iniciado años atrás no acaba aquí ni mucho menos acaba en esta versión, todavía se puede mejorar la aplicación y obtener una aplicación mas completa. Como muestra del trabajo que se puede desarrollar en los próximos años cabria destacar:

- Ampliación de los lenguajes usados para describir la implementación de los procesadores generados. Actualmente XLOP genera una implementación del procesador en CUP, un sistema de generación de traductores para Java que soporta gramáticas LALR (1).

- Ampliación de los lenguajes de programación con los que es posible integrarse ya que en la actualidad solo es posible la integración mediante código Java.

- Permitir una evolución del entorno XLOP para el desarrollo de un entorno de depuración gráfico que facilite al usuario la detección de errores en la especificación de entrada.

Referencias

1. Agencias (2010): Muere uno de los creadores del Trivial Pursuit. El País [Internet]2 de junio. Disponible en: http://www.elpais.com/articulo/gente/Muere/creadores/Trivial/Pursuit/elpepugen/20100602elpepuage_2/Tes [Acceso el 3 de agosto de2010].
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. Compilers: principles, techniques and tools (second edition). Addison-Wesley. 2007
3. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. 2008 (disponible en http://www.w3.org/TR/REC-xml/)
4. Coombs, J. H. Renear, A. H., DeRose, S. J. Markup Systems and the Future of Scholarly Text Processing. *Communications of the ACM*, 30 (11), 933-947. 1987
5. Gálvez Rojas, Sergio. Morata Mata, Miguel Ángel. Traductores y compiladores con LEX/YACC, JFLEX/CUP Y JAVACC. Edición Electrónica. http://www.lcc.uma.es/~galvez/ftp/libros/Compiladores.pdf. 2005.
6. Harold, Elliotte Rusty. Means, W.Scott. XML in a nutshell. O'Reilly. 2004.
7. Kodaganallur, V., Incorporating Language Processing into Java Applications: A JavaCC Tutorial, *IEEE Software* 21(4), 70-77, 2004.
8. Lam, T.C., Ding, J.J., Liu, J.C. XML Document Parsing: Operational and Performance Characteristics. IEEE Computer 41(9), 30-37. 2008.
9. Purdom, P., Brown, C.A. Semantic Routines and LR(k) parsers. Acta Informatica 14, 299-315.1980.
10. Sarasa, A., Martínez-Avilés, A., Sierra, J.L., Fernández-Valmayor, A. A Generative Approach to the Construction of Application-Specific XML Processing Components. 35th Euromicro Software Engineering and Advanced Applications Conference. 2009.
11. Sarasa, A., Sierra, J.L. Fernández-Valmayor, A. Procesamiento de Documentos XML Dirigido por Lenguajes en Entornos de E-Learning. IEEE RITA, *en prensa*. 2009b
12. Sarasa, A.Temprado-Battad, Bryan. Sierra,J.L. Fernández-Valmayor, A. Developing Web Services with XLOP. Cambridge University Press. 2009.
13. Sarasa, A., Sierra, J.L. Fernández-Valmayor, A.Processing Learning Objects with Attribute Grammars. 9th IEEE International Conference on Advanced Learning Technologies. 2009c.

14. Sarasa, A., Temprado, B., Sierra, J.L. Fernández-Valmayor, A. XML Language-Oriented Processing with XLOP. 5th International Symposium on Web and Mobile Information Services.2009d.

15. Sarasa, A., Temprado-Battad, B., Martínez-Avilés, A., Sierra, J.L., Fernández-Valmayor, A. Building an Enhanced Syntax-Directed Processing Environment for XML Documents by Combining StAX and CUP. Fourth International Workshop on Flexible Database and Information System Technology. DEXA'09. 2009e.

16. Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B. From Documents to Applications Using Markup Languages. IEEE Software 25(2), 68-76. 2008b.

17. Stanchfield, S., ANTXR: Easy XML Parsing based on The ANLR Parser Generator. Java Due.com, Hillcrest Comm. & FGM, Inc. javadude.com/tools/antxr/index.html. 2009.

18. Wikipedia (2010). Trivial Pursuit. Disponible en: http://es.wikipedia.org/wiki/Trivial_Pursuit [Acceso el 4 de agosto de 2010].