UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Diseño de un entorno de desarrollo Java para gramáticas de atributos y de Christiansen

Ricardo Pérez Cortés
Tutor: Alfonso Ortega de la Puente

JUNIO 2018

Diseño de un entorno de desarrollo Java para gramáticas de atributos y de Christiansen

AUTOR: Ricardo Pérez Cortés
TUTOR: Alfonso Ortega de la Puente

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2018

Resumen (castellano)

Este trabajo fin de grado soluciona una necesidad de desarrollo del grupo de investigación en el que me integro. En el contexto de la tesis de Marina de la Cruz Echeandía, tutora de este trabajo, se presentaba Geema: una aplicación que implementa su propuesta para la programación automática evolutiva conocida como evolución con gramáticas de Christinansen. Estas gramáticas son una extensión adaptable de las de atributos. En aquella tesis no tuvo cabida la realización de un entorno de desarrollo para la especificación de las gramáticas que pudiera ser utilizado como módulo de entrada de Geema. El presente trabajo soluciona ese problema.

Los entornos de desarrollo de los lenguajes de programación modernos suelen añadir funcionalidad extra a la mera edición de los programas: resaltado de sintaxis, autocompletado con símbolos definidos, e incluso algunas herramientas que faciliten el correcto desarrollo como compilación en tiempo de edición, etc.

Las gramáticas de atributos son un modelo clásico con muchos más años de historia y de desarrollo que las de Christiansen. Uno de sus principales problemas de diseño es la aparición de dependencias entre atributos circulares. Su detección general es un bien conocido problema NP. Uno de los principales atractivos del entorno de desarrollo propuesto en este trabajo es la incorporación de test de corrección de las gramáticas. Nos centramos en los aplicables a gramáticas de atributos, en concreto a una versión simplificada y conservadora de test de circularidades y la detección de problemas de inicialización de atributos.

De esta manera el presente trabajo soluciona una necesidad de desarrollo de las herramientas del grupo de investigación (edición de gramáticas de Christiansen) y añade la funcionalidad relevante de la detección de problemas de diseño de la misma (considerada como gramática de atributos).

Abstract (English)

This Bachelor Thesis solves a need regarding developing tools of my research group. The tool named Geema was introduced associated with my advisor Marina de la Cruz Echeandía's PhD. Geema implements her proposal for evolutive automatic programming known as *Christiansen Grammar Evolution*. These grammars are an adaptable extension to attribute grammars. That PhD research excluded the design and implementation of a development environment for these grammars that could be used as input module for its graphic user interface. The current project solves this problem.

Modern integrated development environments (IDE) for programming languages usually add extra functionality to programming edition: syntax highlighting, automatic completion of defined symbols, and even some tools to ease the correctness of the development process as compilation at edition time, etc.

Attribute grammars are a classic model developed for much more time than Christiansen grammars. One of the main design problems is the circularity question among their attributes. General circularity tests are well known NP algorithms. One of the more appealing features of our IDE is to incorporate tests for checking grammar correctness. We are focused on those applicables to attribute grammars, specifically to a simplified version of circularity tests and on the detection of lack of initial values for attributes.

In this way the current project solves a development need of our research group's tools (Christiansen grammars edition) while offering the relevant feature of design correctness checking for attribute grammars.

Palabras clave (castellano)

Gramáticas de Christiansen, Entorno de desarrollo, Test de circularidad, Gramáticas de Atributos, Gramáticas Formales, Java, Compiladores

Keywords (inglés)

Christiansen Grammals, Development Environment, Circularity Test, Attribute Grammars, Formal Grammars, Java, Compilers

Agradecimientos

Primero que nada y por encima de todo a mi madre y a mi hermana, Monserrat Cortés Mompo y Aitana Pérez Cortés, sin su apoyo y amor incondicional probablemente me habría suicidado antes de empezar esta carrera (ni que decir tiene lo que les debo entonces solo durante esta carrera).

Luego a mí mismo, sin mí es evidente que este trabajo no se habría hecho.

A David Ramos, por ser mí amigo durante tantos años y no huir de mí a pesar de conocerme.

También me gustaría agradecerle a mi tutor de TFG, Alfonso Ortega, quien me ha soportado durante todas las crisis que me han ocurrido intentando hacer este TFG y a su amigo y profesor también de la universidad David Camacho por darme un lugar donde hacer el TFG.

Me gustaría agradecerle a Marly, mi psicóloga puesto que, aunque aún me queda camino por recorrer en mi tratamiento, jamás habría acabado este trabajo en el estado en el que estaba cuando empecé.

Por último, me gustaría hacer una mención especial a todos los youtubers que me han ayudado a procrastinar y cualquier persona que me haya evitado que me vuelva completamente loco.

Ah, y a Clive la roca.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

Desde el momento en que se empezó a cultivar las plantas y encerrar los animales para no tener que ir a buscaros, quedó claro que la humanidad busca conseguir los máximos resultados con el menor esfuerzo posible.

La informática no es excepción y desde la creación de los ordenadores quedó claro con la creación de compiladores para ayudar a dichos objetivos.

No obstante, quedó patente en el desastre que fue el desarrollo del primer compilador de FORTRAN que nos habíamos saltado algunos pasos, por lo que se desarrollaron métodos y herramientas para reducir la complejidad de creación de compiladores, entre ellas las gramáticas formales y las gramáticas de atributos.

A día de hoy la utilización de las gramáticas de atributos se ha ampliado y popularizado bastante desde el momento de su concepción y han ampliado su mira de usos y aplicaciones. No obstante, en muchas ocasiones dichas aplicaciones quedan limitadas por las propias restricciones de las gramáticas y complican la creación de software haciendo que numerosos problemas de restricción en la sintaxis, recaigan y dificulten la semántica (por ejemplo, las tablas de símbolos de los compiladores).

Las gramáticas de atributos fueron propuestas por Knuth en 1968 [1], pero su objetivo era la descripción la sintaxis de los lenguajes de programación cuya sintaxis se expresa como una gramática independiente del contexto.
En la actualidad se han explorado variantes adaptables de este modelo siendo uno de ellos las gramáticas de Christiansen (en las cuales se basa este TFG). Estas permiten la edición de las reglas de derivación que componen la gramática mientras está en uso. La potencia expresiva de ambos modelos es la misma (ambos modelos son computacionalmente completos), por lo que el interés de las gramáticas de Christiansen es de ofrecer un modelo más adaptable y cómodo para el diseñador. Desde el punto de vista teórico las gramáticas de atributos pueden considerarse como el subconjunto de las gramáticas de Cristiansen en las que las reglas no cambian durante su uso. Aun así, son poco conocidas, en parte a la poca cantidad de programas desarrollados para su existencia más allá del punto teórico.

El grupo de investigación en el que estoy trabajando en la E.P.S desarrolló en el pasado una plataforma para el uso de gramáticas de Christiansen para programación evolutiva que no incluía ningún módulo que facilitase el desarrollo de las gramáticas en sí [2]. Por ello este TFG se centra en la creación de un entorno de desarrollo, el cual se conectará con la plataforma y añadirá funcionalidad.

Este TFG además quiere crear un test de circularidad básico para evitar problemas a la hora de generar gramáticas mal diseñadas e informar a los desarrolladores de que es el caso para que puedan plantearse nuevas gramáticas sin problemas.

1.2 Objetivos

El objetivo de este TFG es la creación de un entorno de desarrollo integrado de gramáticas de Christiansen para interactuar con la que ya existe en la EPS para la programación evolutiva utilizándolas. El TFG tocará los siguientes puntos:

1. **Implementación y desarrollo de un editor**

   Se requiere crear un editor en Java (para facilitar la conexión con la aplicación de la EPS), el cual sea capaz de recoger las gramáticas de Christiansen, así como la sintaxis de una manera medianamente usable. Además, la aplicación debe tener algunas de las funcionalidades básicas de un editor.

2. **Diseño y creación de un módulo para comunicar ambas aplicaciones**

   Se requiere que la aplicación sea capaz de generar ficheros utilizables por la plataforma con la que se va a conectar. Para ello se deben generar ficheros de interconexión en un formato legible para la plataforma.
   También deben diseñarse cambios en la plataforma para que sea posible introducir dicha información mediante ficheros.

3. **Diseño teórico de un test implementable en la aplicación para la no circularidad de la propagación de atributos**

   El editor de gramáticas de Christiansen tiene a modo de "herramientas de ingeniería de software la depuración de ciertos errores típicos. Siendo uno de los más conocidos y difíciles de tratar el de la circularidad. El diseño teórico de dicho test de una manera eficiente algorítmicamente será un apartado de la memoria. Se espera que en un futuro se puedan implementar mecanismos de depuración basados en los métodos propuestos para detectar violaciones de la circularidad.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 1: Introducción**

  En este capítulo se introduce la motivación el alcance y los objetivos a los que se espera llegar con este TFG.

- **Capítulo 2: Estado del arte**

  En este capítulo se habla del estado en el que se encuentran actualmente los entornos de desarrollo, las gramáticas de atributos y los test para comprobar si hay dependencias circulares entre atributos.

- **Capítulo 3: Requisitos**

  En este capítulo se exponen los diferentes requisitos que la aplicación ha de suplir solución.

- **Capítulo 4: Diseño**

  En este capítulo se explica la división que se ha hecho en módulos de la aplicación y se explica el diseño de cada uno.

- **Capítulo 5: Pruebas**

  En este capítulo se comentan brevemente las pruebas realizadas sobre la aplicación para asegurar su correcto funcionamiento.

- **Capítulo 6: Conclusiones y trabajo futuro**

  En este capítulo se resume a las conclusiones que se ha llegado con el desarrollo de la aplicación y se exponen objetivos a los que se podría apuntar si se continua con el desarrollo del TFG.

2 Estado del arte

2.1 Estado del arte de entornos de desarrollo integrados

Un entorno de desarrollo integrado (IDE) es una aplicación informática cuyo objetivo es prestar servicios integrales para facilitar el desarrollo de software.
A día de hoy existen entornos de desarrollo integrado para una amplia gama de software.
Respecto a las herramientas para trabajar con gramáticas de atributos la mayoría de ellas están integradas en el lenguaje de programación del lenguaje utilizado para desarrollar los compiladores o intérpretes para los que se usen las gramáticas. Bison/flex [3] se presentan como herramientas para el lenguaje de programación C, JCup/Jflex [4]para Java y el caso de ANTLR [5] es el más parecido a un entorno de desarrollo completo, pero realmente no tiene un editor independiente ya que está integrado en el sistema mediante el que se genera el procesador del lenguaje. La funcionalidad que presenta respecto a la especificación de la gramática es en esencia la misma que la de cualquier IDE típico. Esas características se resumen a continuación.

La complejidad de un entorno de desarrollo integrado varía dependiendo de la funcionalidad que se quiera que dicho entorno provea, siendo el único requisito que sea lo suficientemente complejo como para permitir el desarrollo de parte del software que se quiera realizar con él. Encontramos así IDEs de complejidad variada desde la mínima absoluta (como el "Bloc de notas de Windows") hasta algunos orientados totalmente a hacer una aplicación y proporcionando multitud de algoritmos actualizados en tiempo real para hacer más sencillo el trabajo del desarrollador (como "Netbeans" de SunMicrosystems o Eclipse de IBM).

2.1.1 Funcionalidades y técnicas utilizadas en IDEs de desarrollo de software implementadas en el proyecto

Los algoritmos utilizados varían en función de un IDE a otro y dependiendo de la complejidad de éste, no obstante, anunciaremos aquí algunos de los que se utilizan en la aplicación o podrían aportar alta usabilidad al proyecto si se implementaran en un futuro:

-Guardado y cargado del proyecto: Es una funcionalidad tan utilizada que se asume por defecto que un IDE lo tiene. Consiste en el almacenamiento en el disco duro del proyecto para la utilización del mismo entre sesiones.

-Herramientas de edición de texto estándar: Igual que la anterior esta por defecto en la mayoría de los IDEs y muchos lenguajes aportan librerías para su fácil implementación.

Dichas herramientas incluyen el *copiado* (almacenamiento temporal de datos seleccionados), *pegado* (incluye automáticamente una copia de los datos previamente copiados en el proyecto), *cortado* (igual que copiado, pero elimina el texto copiado del documento), *deshacer* (deshace la última acción realizada por el usuario) y *rehacer* (rehace la última acción que haya deshecho el usuario).

-Resaltado de sintaxis y soporte de lenguajes: consiste en el resaltado de palabras clave, así como de expresiones y literales del lenguaje de programación objetivo del IDE. Algunos IDEs hacen pre-compilaciones incluso y detectan algunos valores introducidos previamente por el usuario (como variables, funciones, constantes definidas…) y los resaltan.
Durante el tiempo que hemos podido dedicar al estudio del estado del arte de este aspecto, hemos llegado a la conclusión de que las herramientas que ofrecen esta funcionalidad no permiten reutilizar su código como un paquete de utilidad externo más. Cualquier usuario de Netbeans, Eclipse y cualquier otro editor Java está familiarizado con esta funcionalidad. En este proyecto hemos implementado nuestra propia versión de esta funcionalidad inspirándonos en ellos.

-Visualización de la jerarquía de directorios del proyecto: algunos IDEs, permiten una visualización de los distintos archivos que componen el proyecto en jerarquía de árbol, así como una navegación fácil entre ellos, llegando al punto de cambiar de fichero en ocasiones dentro del código si enlaza a una función desarrollada en otro IDE.
-Generación de ficheros necesarios y plantillas: la mayoría de lenguajes de programación utiliza ficheros que ejecutan diversos comandos necesarios para la compilación (*makefile* por ejemplo), los cuales pueden ser generados automáticamente por el IDE correspondiente. Además, en la creación de varias partes de los proyectos se suele escribir partes necesarias pero inmutables, que pueden ser automatizadas para ahorrar trabajo al programador (como declaraciones de interfaces o clases).

-Refactorización: modificación del código sin alterar su funcionamiento. Se utiliza para crear de manera simple códigos limpios y ordenados, reestructurando la posición de ciertos elementos para que siga ciertos formatos o también para el cambio del nombre simultáneamente de elementos en todo el proyecto de manera que el usuario no tenga que ir buscando las utilizaciones por sí mismo y cambiarlas individualmente.

-Compilación y depuración: es común en muchos IDEs hacer de puente entre el usuario y el compilador/depurador de modo que simplifican las necesidades del usuario al no tener este que compilar y depurar fuera del IDE. Además, utilizan técnicas como resaltado o métodos más amigables con el usuario visualmente.

2.2 Estado del arte de gramáticas de Christiansen

2.2.1 Gramáticas formales en teoría de autómatas

Noam Chomsky formalizó y clasificó las gramáticas formales generativas por primera vez en la década de 1950. Posteriormente se empezarían a utilizar para el desarrollo y diseño de lenguajes de programación y sus correspondientes compiladores.
Una gramática formal [6] es una estructura algebraica utilizada para la producción de cadenas de texto o palabras. La gramática queda determinada por la 4-tupla:
G={NT, T, S, P}

Dónde:
-G es la gramática.
-NT es el conjunto de símbolos no terminales.
-T es el conjunto de símbolos terminales.
-S es el axioma o el símbolo inicial
-P es el conjunto de reglas de producción.

2.2.1.1 Símbolos

Un símbolo es la entidad básica en una gramática formal. Dentro de ellos se dividen en dos tipos: los terminales y los no terminales.

Los símbolos terminales representan biunívocamente un elemento del diccionario utilizado (pudiendo ser la palabra vacía λ, una palabra de longitud 0).

Los símbolos no terminales se utilizan a modo de variables en las reglas de producción. Un símbolo no terminal no tiene por sí mismo ningún significado. El símbolo inicial es siempre un símbolo no terminal.

2.2.1.2 Reglas de producción

Una regla de producción se divide en una parte izquierda y en una parte derecha, normalmente en la representación separadas por un asignador.
Tanto la parte izquierda como la derecha pueden consistir en una sucesión de símbolos terminales y no terminales, no obstante, nos ceñiremos a las **gramáticas independientes de contexto** en las cuales sólo se puede poner un símbolo no terminal en la parte derecha.
Las reglas de producción sirven para desarrollar las distintas cadenas de las gramáticas mediante un proceso llamado derivación.
La derivación cambia de la regla activa a una de las posibles en orden para poder encontrar la regla o reglas que se ajusten a las cadenas que estamos comprobando si se ajustan a nuestra gramática, empezando siempre por el axioma de la gramática.

2.2.2 Gramáticas de atributos [7]

Las gramáticas independientes de contexto sirven bastante bien a la hora de representar cualquier sintaxis que sean capaces de representar, pero está demostrado que no son computacionalmente completas, es más cada gramática independiente de contexto tiene una biyección con un autómata con pila.
Las gramáticas independientes del contexto no pueden gestionar lenguajes recursivamente enumerables puesto que son equivalentes a los autómatas a pila y no a las máquinas de Turing [7]
Por otro lado, los algoritmos para la gestión de las gramáticas no restringidas son más complejos que los necesarios para tratar las independientes del contexto.
Para abordar esta situación, se propusieron las gramáticas de atributos, que conservan un núcleo independiente del contexto y añaden aquello que les falta para ser computacionalmente completas: un sistema de atributos (semánticos) y las acciones para calcular sus valores

Las **acciones semánticas** son secciones de código añadidas a cada regla de producción que se ejecutan si esa regla de producción es utilizada.
Los **atributos** son variables dentro de la semántica asociados a un símbolo los cuales se pasan entre la ejecución de reglas.

Los atributos son pasados entre reglas mediante dos métodos: síntesis y herencia.
La *síntesis* ocurre cuando el atributo depende del no terminal situado en la parte izquierda de la regla de producción el cual copia el valor a la regla de producción que derivó dicho no terminal.
La *herencia* puede ocurrir de padres a hijos o entre hermanos.
Cuando ocurre de padres a hijos es lo opuesto a la síntesis, se ha derivado un no terminal y el valor de este ha sido copiado al atributo de la parte izquierda de la regla derivada.
Cuando ocurre entre hermanos, ocurre tras la síntesis de un no terminal el cual no era el último no terminal de la regla. Los valores sintetizados del no terminal anterior se heredan al siguiente no terminal a su "hermano".
Las acciones semánticas pueden ser seleccionadas para que ocurran en diferentes **instantes**. Dentro de una misma regla el instante indica cuántos símbolos de la parte derecha han sido derivados, correspondiendo el instante cero al momento anterior a la herencia del primer hermano y el último instante al momento anterior a la síntesis de la parte izquierda.

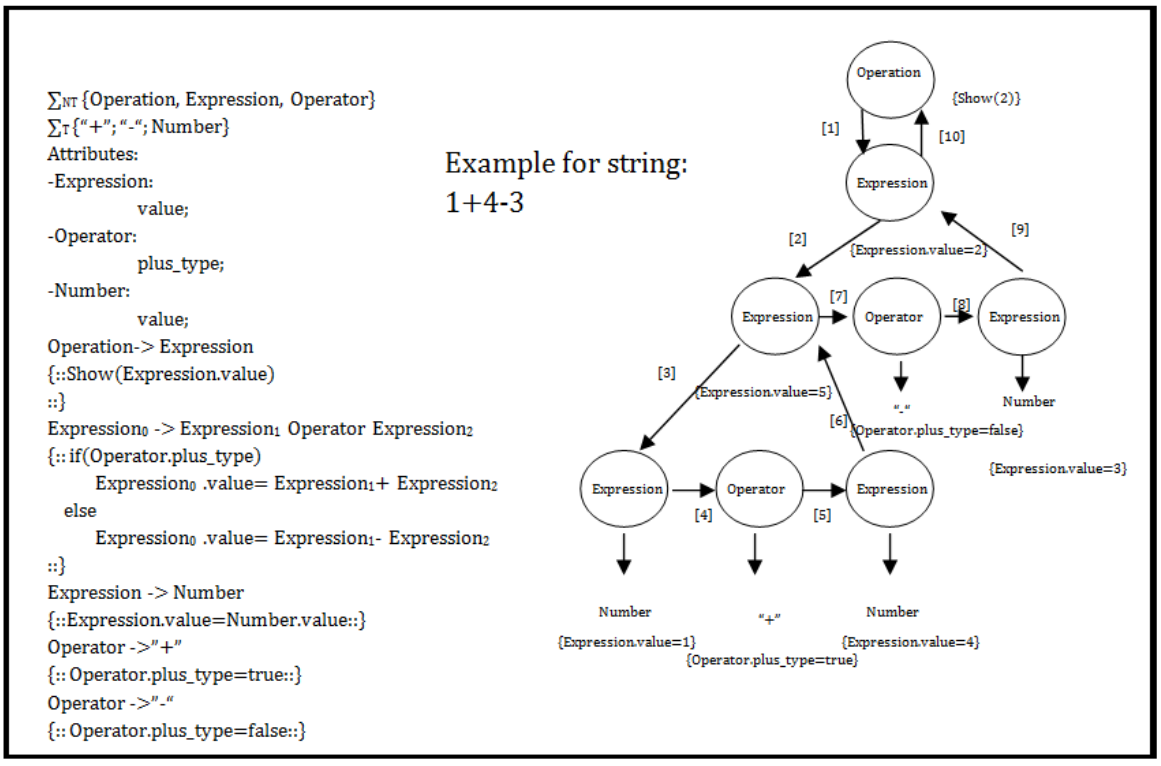

Figura 2-1: Ejemplo de una gramática de atributos

2.2.3 Gramáticas de Christiansen [8]

Las gramáticas de Christiansen pueden ser consideradas como una extensión de las gramáticas de atributos para transformarlas en gramáticas adaptativas (una gramática se considera adaptativa si se puede modificar mientras se usa) y conseguir de esa manera añadirles ventajas que tienen las gramáticas dependientes del contexto.

Es más, las gramáticas de atributos se pueden considerar como un subconjunto de las gramáticas de Christiansen en las cuales las reglas son inmutables y globales.

La gran ventaja que ofrecen las gramáticas de Christiansen con respecto a las gramáticas de atributos es que añaden la posibilidad de añadir y eliminar reglas en tiempo de ejecución. Para ello se modifica el proceso de derivación de manera que cuando se va a derivar un símbolo no terminal, las reglas aplicables se toman de uno de los atributos de la parte izquierda de la regla (un atributo heredado) en lugar de tomarlos del conjunto global de reglas que se utiliza en gramáticas de atributos.

En una gramática de Christiansen todo símbolo tiene como primer atributo por defecto uno heredado que contiene la gramática (de Christiansen) que se podrá utilizar para derivar el símbolo. Normalmente se utiliza otro atributo para facilitar la adaptación de esta gramática y producir la que se utilizará en otros momentos de derivación.

2.3 Estado del arte en el testeo para la no circularidad

Los problemas de circularidad y de bucles son una constante en muchas áreas de la informática. En el caso de las gramáticas de atributos, no es una excepción. La teorización de problemas en esta área data desde los inicios de los compiladores durante los años 60.

Un bucle (o ciclo) [9] ocurre en una gramática de atributos cuando el valor de atributo depende de otro durante su derivación de manera que el segundo atributo depende a su vez del primero.

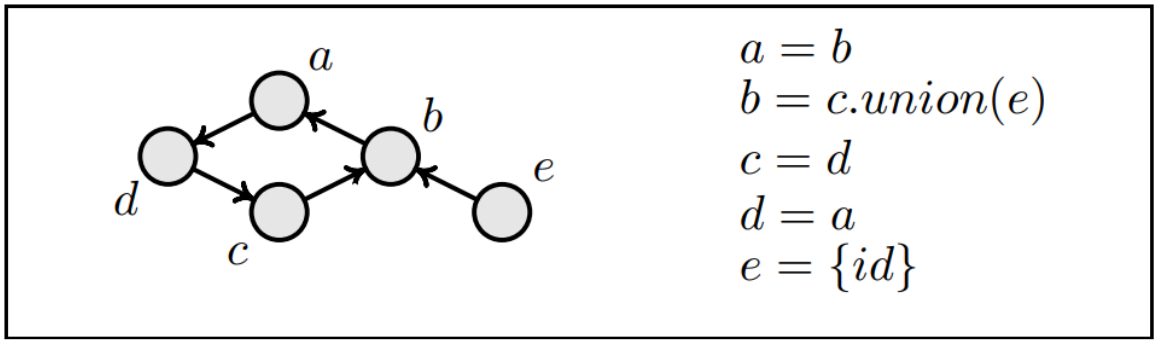

Figura 2-2: Ejemplo de un ciclo

Esto ocurre debido a un mal diseño del uso de los atributos, por ello se ha intentado desarrollar métodos para testear la circularidad de una gramática de atributos y así poder automatizar la detección de malos diseños de esta por parte del desarrollador.

Knuth fue el primero en desarrollar un algoritmo que resuelve dicho problema, no obstante, se demostró que el tiempo requerido para ejecutar dicho algoritmo era exponencial en el peor de los casos. Posteriormente Jazeri demostró que cualquier algoritmo que se desarrollara tendría un coste exponencial [10] y [11]. La demostración se basa en la construcción de una gramática de atributos con ciclos cuyo testeo sea exponencial independientemente del algoritmo (primero mediante grafos de dependencia, y posteriormente aclaró su demostración con el uso de máquinas de Turing).

2.3.1 Gramáticas no fuertemente circulares.

Para resolver el problema inicial en tiempo polinómico, se definieron dos tipos diferentes de gramáticas de atributos:

-*No circulares totalmente*: el problema inicial, el cual es imposible resolver de manera que el peor caso de su evaluación resulte mejor que el caso exponencial.

-*No circulares fuertemente* [12] y [13]: una gramática es fuertemente no circular si al combinar todas las dependencias de todos los posibles sub-árboles no hay ciclos (incluyendo combinaciones que no pueden aparecer normalmente).

Una gramática puede ser comprobada que es fuertemente no circular en tiempo polinómico [14].
Además, está definida de manera que una gramática fuertemente no circular es no circular totalmente (aunque el caso contrario no es cierto).

2.3.2 Conclusiones

Entre las aproximaciones que hemos podido encontrar en el tiempo dedicado al estudio del estado del arte de este apartado dentro del proyecto podríamos clasificar su aproximación de la siguiente manera

- Las que se basan en propiedades de las dependencias consideradas como un grafo [15] [16]
- Las que formalizan y tratan de forma explícita la diferente naturaleza de los accesos a los atributos para la lectura de sus valores o su cálculo (asignación o escritura) en función de la relación que tiene con el sistema de propagación. Tal es el caso de los algoritmos inicialmente propuestos por Knuth y otros siguientes que pueden encontrarse en referencias como [17]
- Las que analizan las propiedades de las dependencias traduciendolas a un dominio distinto como esquemas recursivos o máquinas de Turing. Estos enfoques suelen estar más preocupados por posibilitar un análisis de la complejidad o incluso por la propuesta de algoritmos capaces de calcular los atributos de las gramáticas. No es nuestra preocupación por lo que no añadiremos las referencias encontradas en este aspecto.

Como se explicará más adelante, nos hemos inspirado en el segundo enfoque por ser el más adaptado a las estructuras de datos necesarias por el resto de la aplicación para la gestión de las gramáticas.

3 Requisitos

3.1 Requisitos funcionales

A continuación, se muestra la lista de requisitos funcionales:
RF1-La aplicación ha de permitir al usuario introducir de manera diferenciada el código referente a la sintaxis de las gramáticas de Christiansen del código referente a la semántica.
RF2-La aplicación ha de tener asociada cada regla sintáctica a la acción semántica correspondiente.

RF3-Las acciones semánticas han de estar separadas dependiendo del instante de la regla correspondiente en el que se quiere que la acción se ejecute.
RF4-La aplicación ha de ofrecer funcionalidades básicas de edición (crear un nuevo proyecto, cargar un proyecto anterior, guardar el proyecto actual, copiado de texto, pegado de texto, etc.).
RF5-La aplicación ha de ser capaz de mostrar la información correspondiente a los símbolos de la sintaxis.
RF6-La aplicación ha de ser capaz de resaltar palabras reservadas, constantes y comentarios en el código Java introducido en el código referente a la semántica.
RF7-La aplicación debe generar ficheros legibles y por separado (tanto de la sintaxis mediante la gramática independiente del contexto como de las acciones semánticas de cada regla) por la plataforma a la que se va a conectar.
RF8-La aplicación ha de implementar ciertas comprobaciones de corrección de la gramática entendida como gramática de atributos: una versión reducida de test de circularidades en la propagación de atributos, así como la correcta inicialización de los atributos y la no interrupción de su proceso de propagación. Desde ahora y por abreviar nos referiremos a este conjunto de comprobaciones como la cuestión de *la circularidad*.

3.2 Requisitos no funcionales

A continuación, se muestra la lista de requisitos no funcionales:
RNF1-Java debe ser utilizado como el principal lenguaje de programación.
RNF2-El entorno de desarrollo para la creación de la aplicación ha de ser Netbeans.
RNF3-El resaltado debe de hacerse en menos de un segundo desde que el usuario escribe y no ha de bloquear la entrada de información.
RNF4-El detector de circularidad ha de funcionar en un tiempo polinómico.

3.3 Diagrama de casos de uso y matriz de trazabilidad

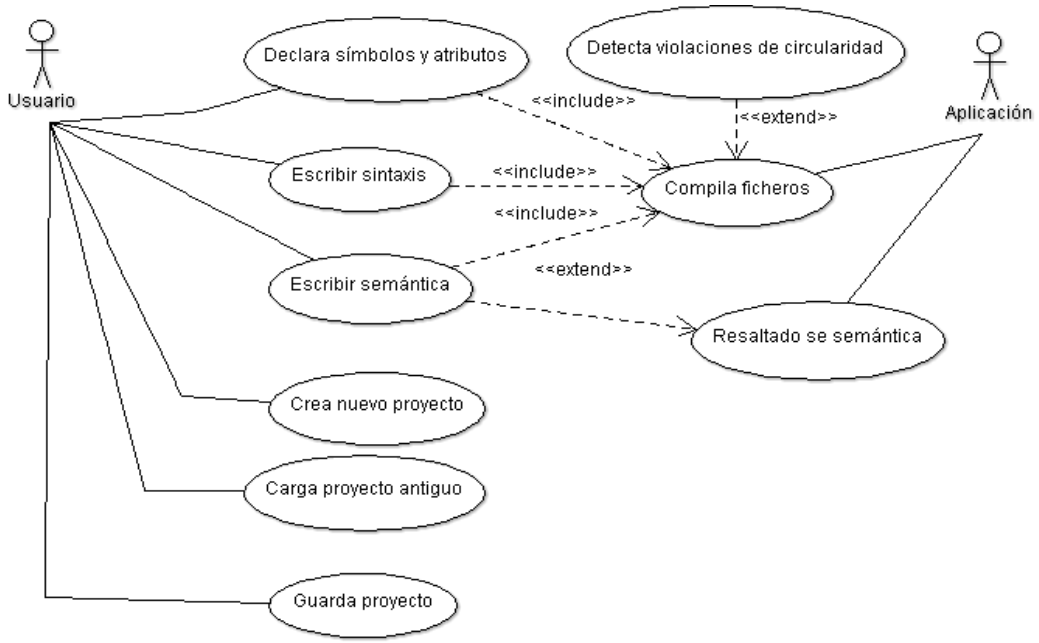

Figura 3-1: Diagrama de casos de uso

Id	Caso de uso	Actor	Descripción	Requisitos asociados
CU1	Declarar símbolos y atributos	Usuario	El usuario declara los símbolos y atributos que van a ser usados en el proyecto	RF1,RF5
CU2	Escribir sintaxis	Usuario	El usuario escribe la sintaxis que será usada en el proyecto	RF1,RF2
CU3	Escribir semántica	Usuario	El usuario escribe la semántica que será usada en el proyecto	RF1,RF2,RF3
CU4	Crear nuevo proyecto	Usuario	El usuario crea un nuevo proyecto en blanco	RF4
CU5	Cargar proyecto antiguo	Usuario	El usuario carga desde un proyecto ya existente en la memoria del ordenador	RF4
CU6	Guardar proyecto	Usuario	El usuario guarda el proyecto actual en la memoria del ordenador	RF4
CU7	Compilar ficheros	Aplicación	La aplicación compila los ficheros en un formato utilizable por la plataforma.	RF7
CU8	Detectar	Aplicación	La aplicación detecta violaciones de	RF8,RNF4

	violaciones de circularidad		circularidad mientras compila el proyecto en CU8 e informa al usuario.	
CU9	Resaltar semántica	Aplicación	La aplicación resalta palabras reservadas, constantes y comentarios en el texto introducido por el usuario en CU3	RF6,RNF3

Tabla 3-1: Matriz de trazabilidad

Sería muy adecuado que, en este punto, para comprender la lista de funciones de cada módulo que aparece en las siguientes secciones, se consulte el manual de usuario. Se supondrá en lo restante que se ha hecho así para evitar redundancias.

4 Diseño

En este capítulo se pretende dar una idea del diseño de la aplicación; de los módulos que tiene y de qué funcionalidades deben de estar recogidas en cada módulo. Además, se especifican las interconexiones inicialmente definidas de cada módulo entre ellos.

La aplicación ha sido desarrollada en diversos módulos, cada módulo centrado en una serie de funcionalidades relacionadas. Los módulos quedan definidos en la siguiente figura:

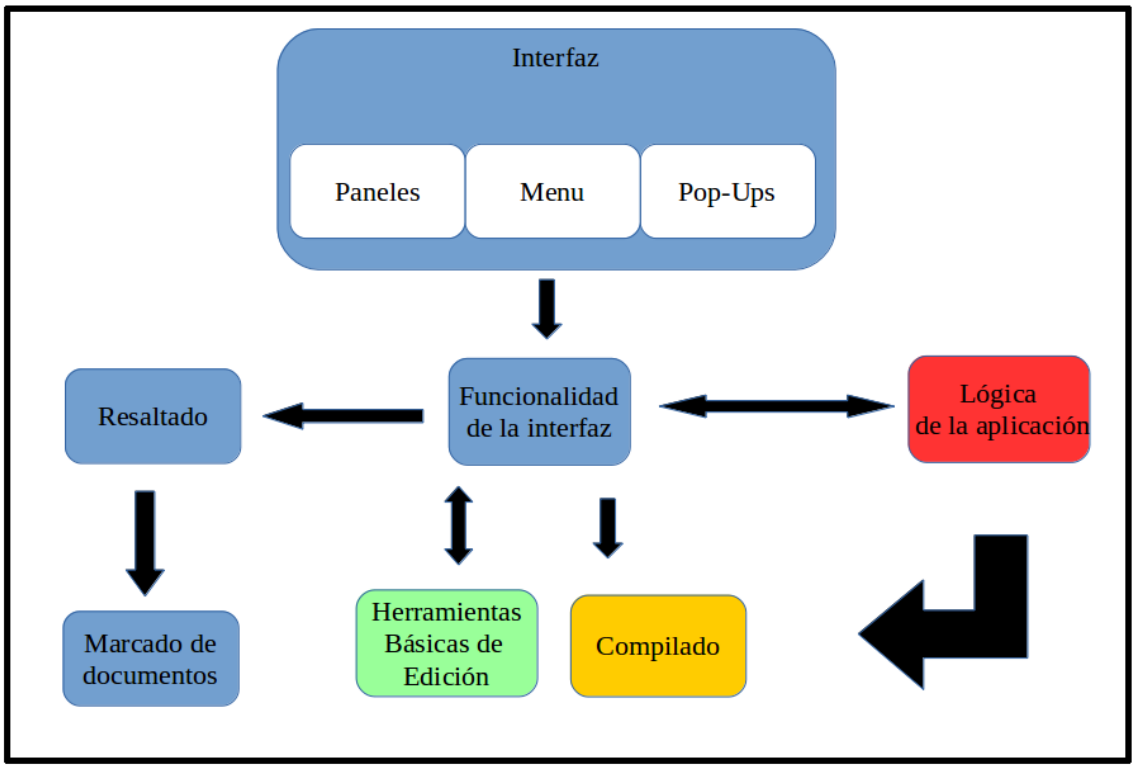

Figura 4-1: Módulos de la aplicación

4.1 Interfaz

El objetivo de este módulo es recoger todas las indicaciones del usuario y hacer la comunicación entre el usuario y el resto de la aplicación. Además, se encarga de generar una realimentación visual hacia el usuario basándose en los datos del módulo de funcionalidad de la interfaz.

Debido a que la aplicación está diseñada para ser una herramienta para el usuario, éste será el módulo principal y ningún otro módulo tomará el control de la aplicación salvo que este módulo se lo haya cedido previamente.

La interfaz está dividida en tres zonas las cuales están separadas en sus propios submódulos: los paneles, el menú y los pop-ups.

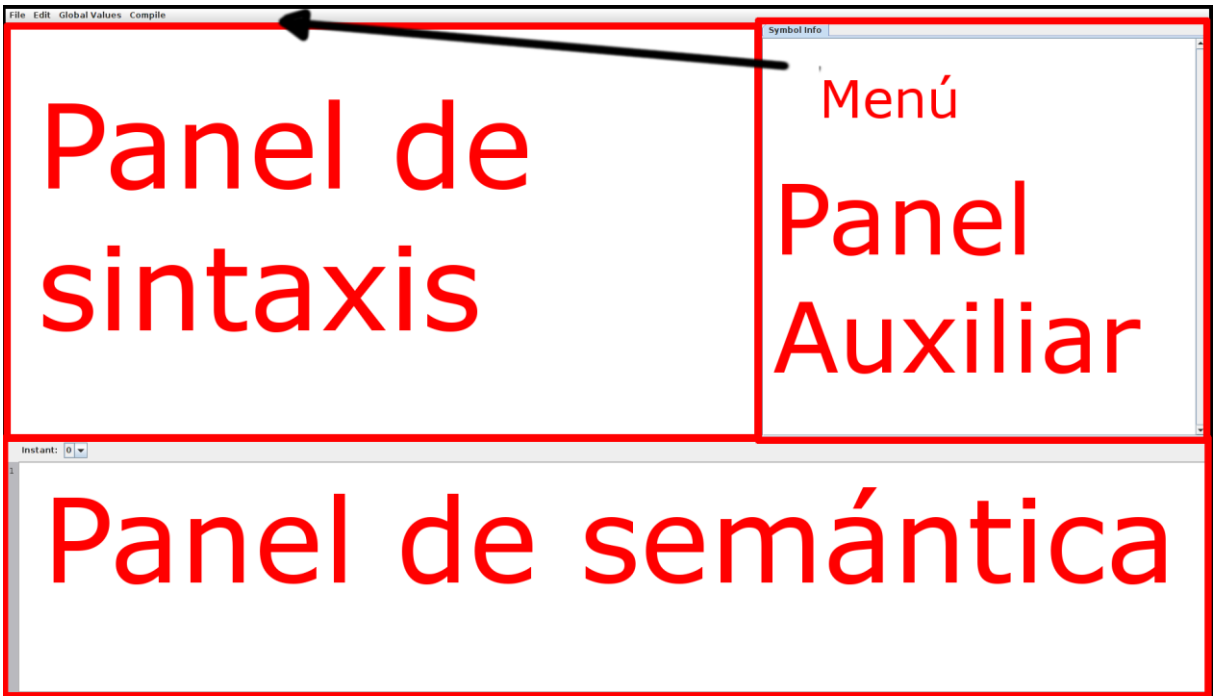

Figura 4-2:: Diseño de la interfaz

4.1.1 Submódulo de la interfaz: Paneles

Los paneles ocupan la mayor parte de la pantalla y están diseñados para funcionar como la forma principal de intercambio de información entre el usuario y la aplicación.

Visualmente los paneles están divididos en tres zonas:

-Panel de sintaxis: Situado en la zona superior izquierda, se encarga de la recogida de información sintáctica de las gramáticas.

-Panel auxiliar: Situado en la zona superior derecha, es un contenedor, que tiene una variedad de paneles de forma que sólo uno está activo y se muestra por pantalla. Los diferentes paneles pueden alternar cual es el activo a disposición del usuario.
Actualmente tiene tres subpaneles entre los que se puede alternar el foco de la aplicación:

* *Información sintáctica*: Panel únicamente de salida de la información recogida de la aplicación sobre los distintos símbolos y sus atributos. Ejemplo de contenidos del panel de información sintáctica:
"Symbol: S Type: Non Terminal
Attribute 0: ClassType[GramaticaChristiansen] Name[inheritedGrammar]
Attribute 1: ClassType[GramaticaChristiansen] Name[synthesizedGrammar]
Attribute 2: ClassType[int] Name[dotCount]

Symbol: dot Type: Terminal

Symbol: text Type: Terminal
Attribute 0: ClassType[String] Name[text]
Attribute 1: ClassType[int] Name[dotCount]"
* *Cabecera semántica*: Panel diseñado con el objetivo de recoger código Java que será utilizado para fragmentos que no cambian de los ficheros. Ejemplo del código que se le puede introducir:
"*private int variable;*

public void GlobalFunction(){
System.io.out.println("Hello World");
}"
* *Consola*: Panel únicamente de salida de la información resultante del módulo de compilado. Ejemplo de contenidos de la consola:
"*Syntax consistency compromised.*
Remember to declare every Symbol as a terminal or non terminal in the global values tab"

-Panel semántico: Situado en la zona inferior, su funcionalidad reside en la recogida de la información semántica de la aplicación (la cabecera semántica está diseñada como una extensión del panel semántico). Su contenido depende del panel sintáctico y la regla seleccionada en dicho momento. Además, se compone de un indicador de línea (que indica en qué línea está el código en ese momento) y un seleccionador de instante (el cual muestra diferentes porciones de código para distintos momentos de la derivación de una regla semántica.

4.1.2 Submódulo de la interfaz: Menú

El menú está situado en una pequeña barra en la parte superior de la aplicación. Permite la llamada a diversas acciones de la aplicación que no son tan frecuentes como las recogidas por los paneles y/o que pueden ser llamadas pulsando un solo botón.

Además, contiene "aceleradores" o combinaciones de teclas para ciertas acciones que pueden ser ejecutadas si se detecta en el usuario dicha pulsación para no tener que acceder al menú.
El menú está compuesto por los siguientes submenús:

-Submenú File:
Recoge los ítems que activan eventos relacionados con los archivos de la aplicación. Los distintos eventos que genera actualmente son:
* *"New"*: crea un nuevo proyecto, en blanco (acelerador Ctrl+N).
* *"Open"*: abre un proyecto ya existente (acelerador Ctrl+Mayús+O).

*”*Save”*: guarda el proyecto actual en el disco duro (acelerador Ctrl+S).
*”*Useless Option”*: no hace nada, mi firma personal en la aplicación.
*”*Exit”*: sale y cierra la aplicación.

-Submenú Edit:
Recoge los ítems que activan eventos relacionados con la edición del proyecto activo en la aplicación. Los eventos que genera actualmente son:
*”*Cut”*: copia en el portapapeles para uso posterior y elimina el texto seleccionado (acelerador Ctrl+x)
*”*Copy”*: copia en el portapapeles para uso posterior el texto seleccionado (acelerador Ctrl+c). Su función es igual a cortar si después de ejecutar su acción se pulsa suprimir.
*”*Paste”*: Pega texto almacenado previamente en el portapapeles (acelerador Ctrl+v).
*”*Undo”*: Deshace el último cambio en la aplicación. Puede ser utilizado varias veces para deshacer varios cambios. (acelerador Ctrl+z). (**Importante**: por razones de complejidad no se ha implementado esta funcionalidad en el Panel sintáctico).
*”*Redo”*: Rehace los cambios deshechos por “Undo” si el usuario no ha hecho ningún cambio.

-Submenú Global Values:
Recoge los ítems que activan eventos relacionados con valores globales del proyecto. Los eventos son:
*”Imports”: Activa el *pop up* de imports.
*”Symbols”: Activa el *pop up* de symbols.
*”Symbol Attributes: Activa el *pop up* de symbol attributes.
*”Precedences”: Activa el *pop up* de precedences.
Los *pop ups* se explican con mayor detalle en su apartado de la memoria.

-Botón Compile:
Activa un evento que se encarga de la compilación de la gramática y la generación de los ficheros resultantes e informa por la consola.

4.1.3 Submódulo de la interfaz: *Pop ups*

Este submódulo recoge la parte de la interfaz dedicada a la creación de ventanas emergentes de la aplicación, lo cual incluye su diseño, la recogida de los datos introducidos y su correcta expresión a partir de los datos previamente almacenados. Estas se muestran debido a alguna acción en otra parte de la interfaz (principalmente en el menú).
Los *pop ups* que hay en la aplicación son:

-*Pop up* de precedencias:

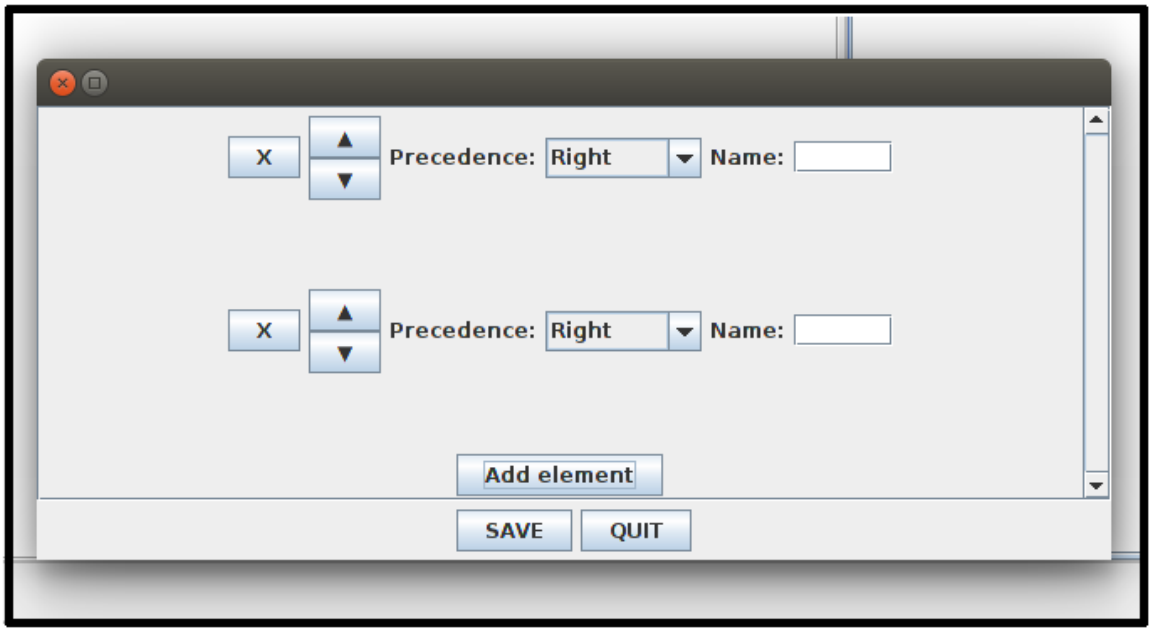

Figura 4-3: *Pop up* de precedencias

El *pop up* de precedencias se activa al pulsar el botón de menú de precedencias.
Tiene un botón para añadir elemento ("Add Element"), uno para guardar los valores introducidos ("SAVE") y otro para cerrarlo sin guardar los cambios ("QUIT").
Cada elemento añadido tras pulsar el botón de añadir elemento tiene los siguientes componentes:
*Un botón para eliminar el componente
*Dos botones para intercambiar posición con el elemento inmediatamente superior o inmediatamente posterior
*Un selector de precedencia, siendo las opciones: "Right", "Left" y "Non-asoc"
*Un campo de entrada para poner el nombre de la precedencia

Este *pop up* aunque esté dentro de la aplicación, no es utilizado en ninguna parte, se diseñó para gramáticas de atributos inicialmente pero se deja como *placeholder* por si alguna vez la aplicación de la EPS para la que se diseñó acaba implementando precedencias.

-Pop up de imports:

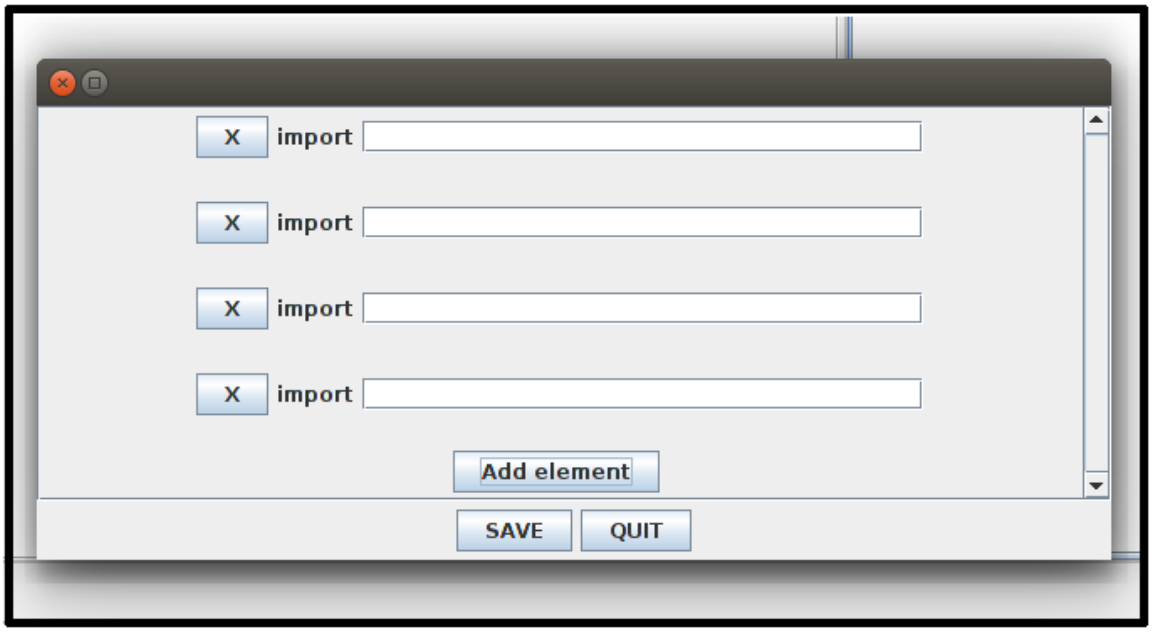

Figura 4-4: *Pop up* de imports

El *pop up* de precedencias se activa al pulsar el botón de menú de imports.
Tiene un botón para añadir elemento (“Add Element”), uno para guardar los valores introducidos (“SAVE”) y otro para cerrarlo sin guardar los cambios (“QUIT”).
Cada elemento añadido tras pulsar el botón de añadir elemento tiene los siguientes componentes:

*Un botón para eliminar el componente
*Un campo de entrada para indicar la información del import

Este *pop up* está diseñado para añadir la información relativa a los *imports* que se deben añadir en la semántica generada. La palabra *import* no hace falta ponerla (la aplicación la rellena por defecto) solo la ruta a las librerías utilizadas.

-Pop up de símbolos:

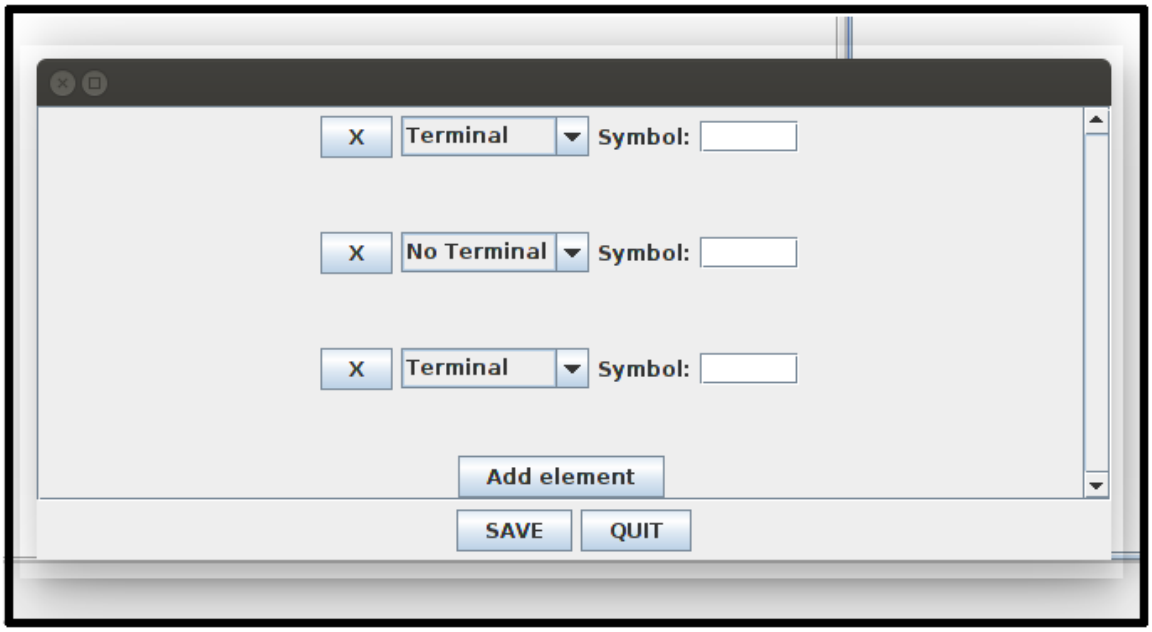

Figura 4-5: *Pop up* de símbolos

El *pop up* de símbolos se activa al pulsar el botón de menú de símbolos.
Tiene un botón para añadir elemento (“Add Element”), uno para guardar los valores introducidos (“SAVE”) y otro para cerrarlo sin guardar los cambios (“QUIT”).
Cada elemento añadido tras pulsar el botón de añadir elemento tiene los siguientes componentes:

*Un botón para eliminar el componente
*Un selector para indicar si el símbolo es terminal o no terminal.
*Un campo de texto para indicar el nombre del símbolo añadido.

Por medio de esta ventana se definen los símbolos que se usarán en la sintaxis del proyecto.
Una sintaxis que utilice símbolos no definidos dará error a la hora de intentar compilar el programa.
Si se define un símbolo dos o más veces sólo la primera vez será tomada en cuenta y la aplicación no guardará las siguientes.

-*Pop up* de atributos de símbolos:

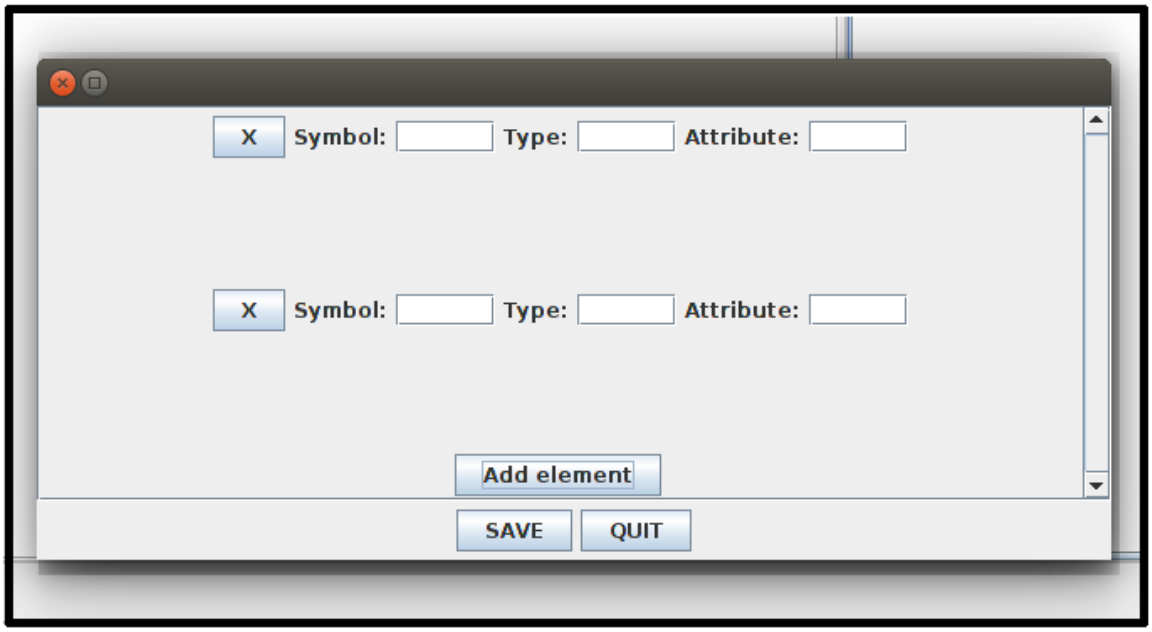

Figura 4-6: *Pop up* de atributos de símbolos

El *pop up* de atributos de símbolos se activa al pulsar el botón de menú de atributos de símbolos.
Tiene un botón para añadir elemento ("Add Element"), uno para guardar los valores introducidos ("SAVE") y otro para cerrarlo sin guardar los cambios ("QUIT").
Cada elemento añadido tras pulsar el botón de añadir elemento tiene los siguientes componentes:

*Un botón para eliminar el componente
*Un campo de texto para indicar a qué símbolo se le añade el atributo
*Un campo de texto para indicar de qué tipo de dato será el atributo
*Un campo de texto para indicar el nombre del atributo

Hay que notar que se pueden añadir atributos a símbolos que no se han definido, en dicho caso no pasará nada, pero dichos atributos no se tendrán en cuenta en la aplicación hasta que se defina el símbolo.

Si se define dos veces el mismo símbolo para el mismo atributo solo la primera vez será tomada en cuenta y la aplicación no guardará las siguientes.

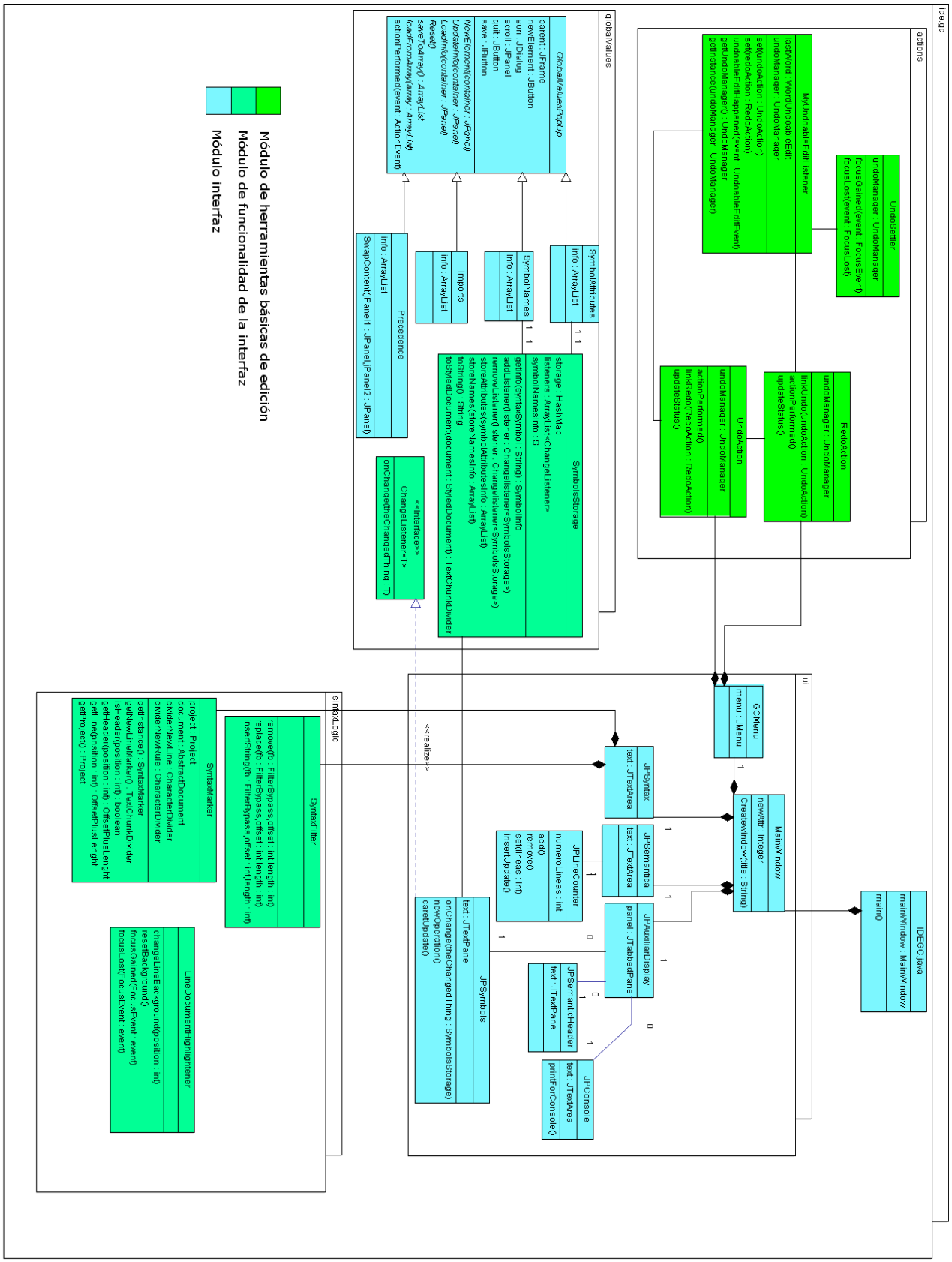

Figura 4-7: Diagrama de clases de la interfaz.

4.2 Módulo de funcionalidad interna

Este módulo se encarga de dar funcionalidad a la interfaz y conectarlo con el resto de funciones de la aplicación.
Aparte de la funcionalidad interna básica de ejecutar acciones o abrir *pop ups* al pulsar botones del menú, este módulo se encarga de las siguientes funciones también:
-Recogida de datos de los diversos *pop ups* y su paso al módulo de lógica interna.
-Mantenimiento de un código estable y utilizable por el módulo de lógica interna en el panel de sintaxis. Se ha hecho mediante la especialización de la clase Java *DocuemntFilter* (el filtro es una funcionalidad que permite o deniega la escritura de la entrada del usuario en el documento mediante la llamada a diferentes funciones previas a la escritura). Dicho filtro impedirá que en el panel queden otra cosa que no sean palabras que comiencen por una letra y compuestas por letras y números, o un solo carácter rodeado de comillas (o dos si el primer carácter es "\") o el carácter ":" tras el cual se insertará una nueva línea y el filtro se encargará de que no se escriba nada entre dicho carácter y la nueva línea.
- Marcando de puntos importantes en la sintaxis del documento (puntos indicando nueva regla ó nueva línea) y generación de eventos cuando son modificados.
-Almacenamiento de los diversos ficheros relativos a la semántica y su carga dependiendo de la regla sintáctica resaltada, así como dependiendo del instante en el que esté seleccionada dicha regla, así como la actualización y el cálculo de instantes basado en la regla seleccionada.
-Impresión de la información de los símbolos según su actualización y pegado de los atributos seleccionados en esta en caso de que el usuario clique sobre ello en la zona semántica.
-Actualización del contador de líneas en la información sintáctica cuando se introducen o se eliminan líneas.
-Almacenamiento de la información el programa manteniendo las referencias al módulo de funcionalidad interna.
-Resaltado de la semántica. Como se describirá posteriormente y se ha presentado anteriormente, existe un módulo de resaltado de texto. Se ha diseñado específicamente para que sea invocado desde el módulo de funcionalidad interna para implementar la funcionalidad de resaltado de la semántica.

4.3 Módulo de resaltado

Como su nombre indica la única función de este módulo consiste en el resaltado de un documento. Los colores en los que resalta están prefijados mediante constantes (aunque en un futuro se podrían leer desde otro lado, pero no es prioridad) y está diseñado con el objetivo de resaltar código Java con las especificaciones diseñada para esta aplicación.
El módulo de resaltado utiliza el módulo de marcado del documento para separarlo en bloques que serán analizados al resaltarlo.
Su implementación se basa en la actualización del resaltado dependiendo de las actualizaciones del documento, no obstante, debido a que es algo que debe hacer a la vez que el usuario escribe, debe ser rápido y por tanto no puede analizar todo el documento (puesto que en documentos grandes podría el usuario sentir una desconexión entre lo que escribe y el resaltado de sintaxis, ya que tendría que resaltar de nuevo todo el documento y esto bien podría tardar varios segundos en actualizar cualquier cambio hecho por el usuario).
Originalmente la idea es analizar líneas completas o comentarios multilineados completos dependiendo de si en ese momento se encuentra con un comentario multilínea o no, a parte del texto insertado en dicho momento. Esto aprovecha el hecho de que uno de los estándares del código limpio es la división de éste en líneas cortas. Si por algún casual esta

división ralentizara demasiado el funcionamiento del resaltado, está diseñado de manera que no sería complicado en exceso reestructurarlo para que analice palabras en lugar de líneas completas, no obstante, debido a la complejidad y la mayor cantidad de casos que esto produce se ha procedido inicialmente con el análisis de líneas.
El resaltado se produce en un hilo (*thread*) del programa separado para evitar que el módulo bloquee al usuario.
El módulo debe reconocer y resaltar los siguientes elementos:

-**Strings**: secuencias de distintos caracteres que comienzan al encontrarse comillas dobles y finalizan al encontrarse otras comillas dobles o al finalizar la línea.
-**Caracteres literales**: caracteres encapsulados por comillas simples.
-**Números literales**: palabras compuestas únicamente por cifras, que empiecen por 0x y compuestos por números o letras de la "A" a la "F" (*números hexadecimales*) o que lleven un punto en medio y/o acaben en "f" (*números decimales y flotantes*).

-**Comentarios:** cadenas de caracteres que empiezan por dos barras inclinadas "//" hasta llegar al final de la línea.

-**Comentarios multilineados**: cadenas de caracteres que empiezan por los caracteres "/*" y acaban con los caracteres "*/" o al llegar al final del documento.

-**Palabras reservadas**: una lista de palabras proporcionada por un fichero de texto que indican las palabras reservadas de Java, las cuales deberán de ser resaltadas.

-**Acceso a los atributos y símbolos de la sintaxis**: esta parte no es de Java, pero es conveniente resaltarla también, como se especifica en su parte correspondiente del diseño, dichos símbolos serán resaltados para facilitar su legibilidad.

Además de reconocer y resaltar dichos elementos, al analizar el documento puede encontrarse con alguno de los elementos dentro de otro. En dicho caso únicamente debe resaltarse el primero encontrado y si hay algún símbolo que indique el comienzo de otro elemento dentro de este ha de ser ignorado. Por ejemplo el string " /* " tiene dentro suya la marca que indica el comienzo de un comentario multilineado pero solo ha de resaltarse como string y no ha de considerarse que haya comenzado un comentario multilineado.

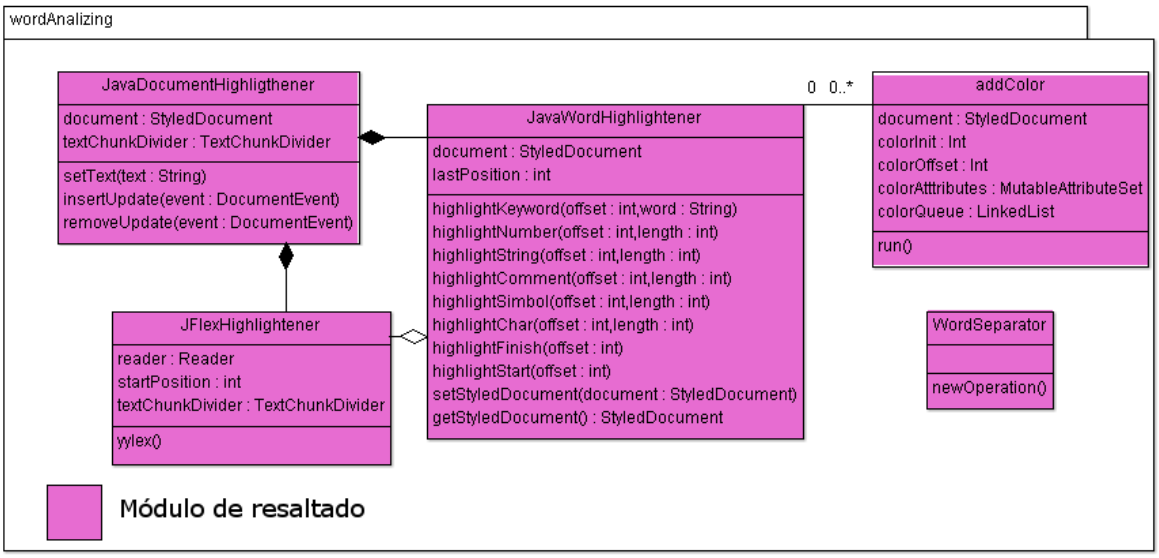

Figura 4-8: Diagrama de clases del módulo de resaltado

El módulo de resaltado esta formado por las clases *JavaDocumentHighligthener*, *JFlexHighLightener, JavaWordHighlightener,addColor* y *WordSeparator*.
La clase principal del módulo es *JavaDocumentHighligthener* la cual se encarga del analisis del documento. Para ello Hace uso de *JflexHighlightener* para identificar las distintas palabras y constantes y separarlas semánticamente. Una vez identificadas llama a *JavaWordHighlightener* para su resaltado dependiendo de qué tipo de palabra sea.

4.4 Módulo de marcado del documento

Este módulo está diseñado para introducir marcas en un documento y encontrar la marca más cercana a otra de la manera más eficiente posible. Su diseño está basado en gran parte en las cache de los ordenadores.
Para ello almacena las marcas en una lista junto con la posición suya relativa a la marca anterior. Solo guarda la posición real de la última marca consultada y en caso de necesitar la posición real de otra marca va sumando las posiciones relativas si está guardada posteriormente o va decrementándolas si está guardada anteriormente a dicha marca.
Además, está atento a los cambios que experimenta el documento de manera que los cambios producidos se reflejan en las posiciones relativas en donde ocurren dichos cambios.
Este módulo consigue de esta manera almacenar las posiciones de todas las marcas de manera que cada cambio solo tiene que actualizar una y en la mayoría de los cambios no importa la cantidad de marcas que tenga almacenada el módulo para la búsqueda o actualización de las posiciones relativas.
Las marcas han de ser introducidas externamente indicando la posición, pero se consideran como un conjunto de caracteres del documento y es eliminada automáticamente si alguno de los caracteres que conforman la marca es eliminado o modificado.
Para facilitar la implementación del programador del marcado si se vuelve a añadir la misma marca en cierta parte del documento en la que ya había una instancia de dicha marca el módulo ignora la nueva marca.
Este módulo evidentemente tiene problemas cuando se empiezan a acceder zonas alejadas de las marcas, pero como se utiliza para editado de documentos y normalmente en una ligera edición se pueden consultar varias veces las marcas de la misma zona (ya que en el módulo de resaltado consultamos todas las marcas de la misma línea con cada edición) y los usuarios suelen editar por orden el código escrito, conseguimos un marcado eficiente relativamente independiente del número de marcas del proyecto.
El módulo también añade la posibilidad de insertar listeners para indicar que una marca ha sido añadida o eliminada.
El módulo comprende el módulo, las interfaces de marcas, listener y su clase abstracta, algunas marcas utilizadas por la aplicación y la marca vacía que se añade por defecto al principio de cualquier documento que se asocie con el módulo para asegurarse de que al menos siempre hay una marca.

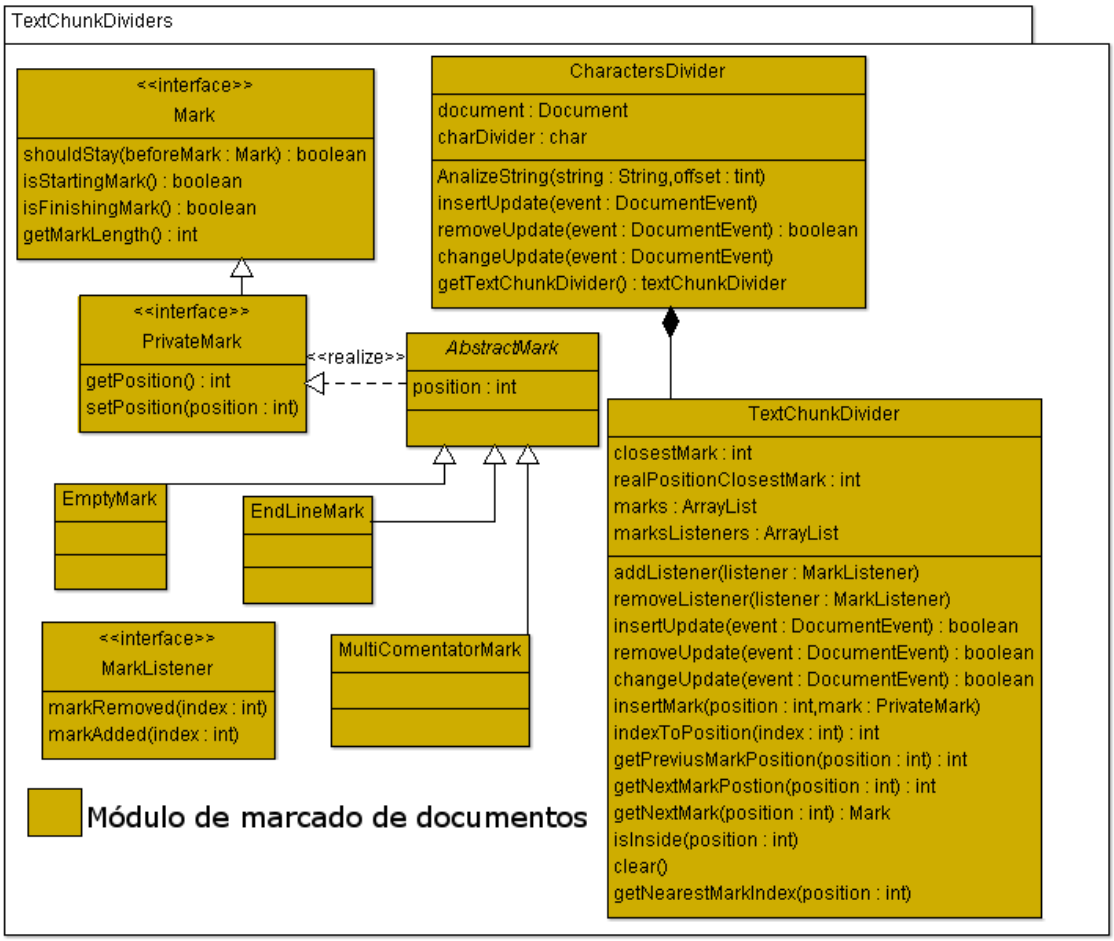

Figura 4-9: Diagrama de clases del módulo de marcado

4.4.1 Clases del módulo de marcado

-Clase *TextChunkDivider*: clase básica y el módulo en sí mismo. Se encarga del marcado del documento y la división de esta en bloques. Es además un *listener* de modificaciones de documento, y se le debe de asignar a este en el momento en que se quiera usar.
-*Interfaces Mark y PrivateMark*: son interfaces diseñadas para representar las marcas que se introducen en el documento. Private mark extiende la funcionalidad para el funcionamiento interno del documento, mientras que mark añade la funcionalidad que el código externo debería mirar.
-*AbstractMark*: implementación de *PrivateMark* de manera abstracta para facilitar su implementación.
-*EmptyMark*: marca vacía, no se debe usar más que para comprobaciones. Esta marca se añade por defecto en el documento vacío y significa el inicio de este.
-*EndLineMark* y *MultiComentatorMark*: generalizaciones de *AbstractMark* para representar las marcas de fin de línea y comentario multilineado.
-*MarkListener*: Interfaz que debe implementar cualquier listener que quiere que se le informe de un cambio en las marcas de un documento.
-*CharactersDivider*: escáner básico para detectar un único carácter y llevar marcas de dicho carácter en el documento usando *TextChunkDivider*. Sirve como ejemplo de uso de la clase *TextChunkDivider* también.

4.5 Módulo de herramientas básicas de edición

El módulo de herramientas básicas de edición es llamado por el módulo de funcionalidad interna al pulsar alguno de los botones del menú de "file" o "edit". Las funciones de edición que ahora contiene este módulo son:
-Guardar y cargar los datos del programa. Contacta con el módulo de lógica interna y almacena en disco la información necesaria o busca un fichero en el sistema he intenta que la lógica interna lo abra.
-Cortado, copiado y pegado. Aunque los botones estén implementados en la interfaz, es este módulo el que se encarga de implementar la funcionalidad, lleva cuenta del portapapeles y del panel de texto activo para insertar el texto en su lugar correspondiente.
-Deshacer y rehacer. Igual que la funcionalidad anterior, lleva cuenta del panel en que se ha de hacer el cambio y de los cambios hechos para hacerlos y deshacerlos. También junta cambios de inserción de letras en un único cambio de inserción de palabras para acelerar la edición de cambios.

4.6 Módulo de lógica interna

La lógica interna puede estructurarse informalmente en los siguientes bloques conceptuales.

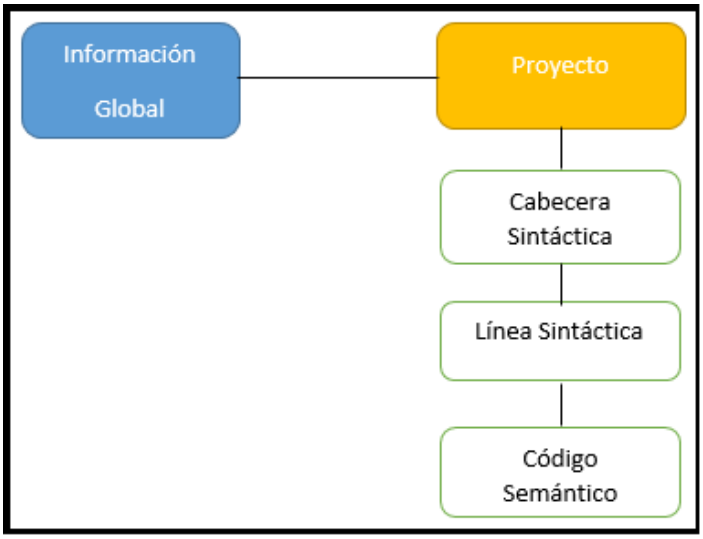

Figura 4-10: Estructura del módulo de lógica interna

El módulo de lógica interna se encarga del almacenamiento de los datos y su actualización a medida que se van utilizando en la aplicación, así como su estructuración para poder ser utilizado por el resto de la aplicación. El módulo está estructurado en diversas partes:
-**Información global**: se encarga de conectar con la información recogida por apartados globales de la aplicación (pop ups y el panel de cabecera semántica.
-**Proyecto**: parte contenedora de todas las demás partes de la lógica interna, se encarga de la recogida de toda la información del usuario.
-**Cabecera sintáctica**: simboliza la parte izquierda de una regla en el panel sintáctico. Tiene todas sus partes derechas asociadas como líneas sintácticas.
-**Línea sintáctica**: simboliza la parte derecha de una regla en el panel sintáctico. Tiene

asociado el código semántico relacionado con la regla actual, y el instante al que dicho código pertenece.
-**Código semántico**: contiene el texto asociado a un instante que se ha introducido por el panel semántico.
El protagonismo, por tanto, lo tiene el bloque *Proyecto*, diseñado para asegurarse del correcto funcionamiento entre el resto de partes de la lógica interna.
Éste mantiene una lista de las cabeceras sintácticas de la aplicación, referencias a todos los datos referentes a información global. Además, se encarga de que todas sus referencias puedan ser serializadas y deserializadas de manera que la información que almacena sea posible extraerla (para almacenarla en un fichero) o cargada desde información previamente serializada (para cargar el proyecto desde el fichero). Es necesario esta serialización para poder trabajar sobre el mismo proyecto en diferentes sesiones.
Además, la información no se actualiza constantemente, sino que espera ciertos eventos o ciertas llamadas y la va actualizando acordemente, avisando de que parte del proyecto debe actualizar la información.
Las cabeceras sintácticas simbolizan la parte izquierda de una regla sintáctica y su información es recogida desde el panel sintáctico de la aplicación. Estas tienen a su vez una lista de líneas sintácticas que simbolizan la parte derecha de las reglas que tienen la misma parte izquierda.
Si una línea sintáctica es indicada en la aplicación como una cabecera sintáctica (para ello hay que añadir en el panel sintáctico el símbolo ":"), la cabecera sintáctica que la contenía se divide en dos añadiendo en la nueva cabecera las líneas sintácticas posteriores a la nueva cabecera y eliminándolas de sí misma.
Si una cabecera sintáctica es eliminada (para ello hay que eliminar en el panel sintáctico algún símbolo ":") sus líneas sintácticas son transferidas a la cabecera sintáctica anterior. Para evitar problemas no se pueden eliminar los primeros ":" del panel de sintáxis.
Una línea sintáctica contiene su nombre y mantiene un diccionario que relaciona los instantes con los códigos semánticos que se guardan en dicha línea sintáctica. Una línea sintáctica contiene como máximo el número de palabras que contiene más uno (cada instante corresponde a un instante de la derivación de la regla).
Si una línea sintáctica es dividida en dos en la aplicación (para ello hay que introducir una nueva línea) se cuentan los nuevos instantes que se han dejado en la línea sintáctica original y se crea una nueva línea sintáctica en la que se añaden los instantes que tenga un número mayor a la línea original, con el instante restado la cantidad de instantes de la línea original.
Si se mezclan dos líneas sintácticas (para ello hay que eliminar una nueva línea entre dos líneas sintácticas) se han de añadir los instantes de la segunda línea sintáctica en la primera con el instante de la primera sumado a los de la segunda.
El código semántico representa al código que se introduce en el panel semántico. Solo guarda el texto que se está actualizando (el resto aunque estén almacenado no es visible por el usuario a no ser que seleccione la regla e instante correspondientes), el cual tiene almacenado una referencia de manera estática.

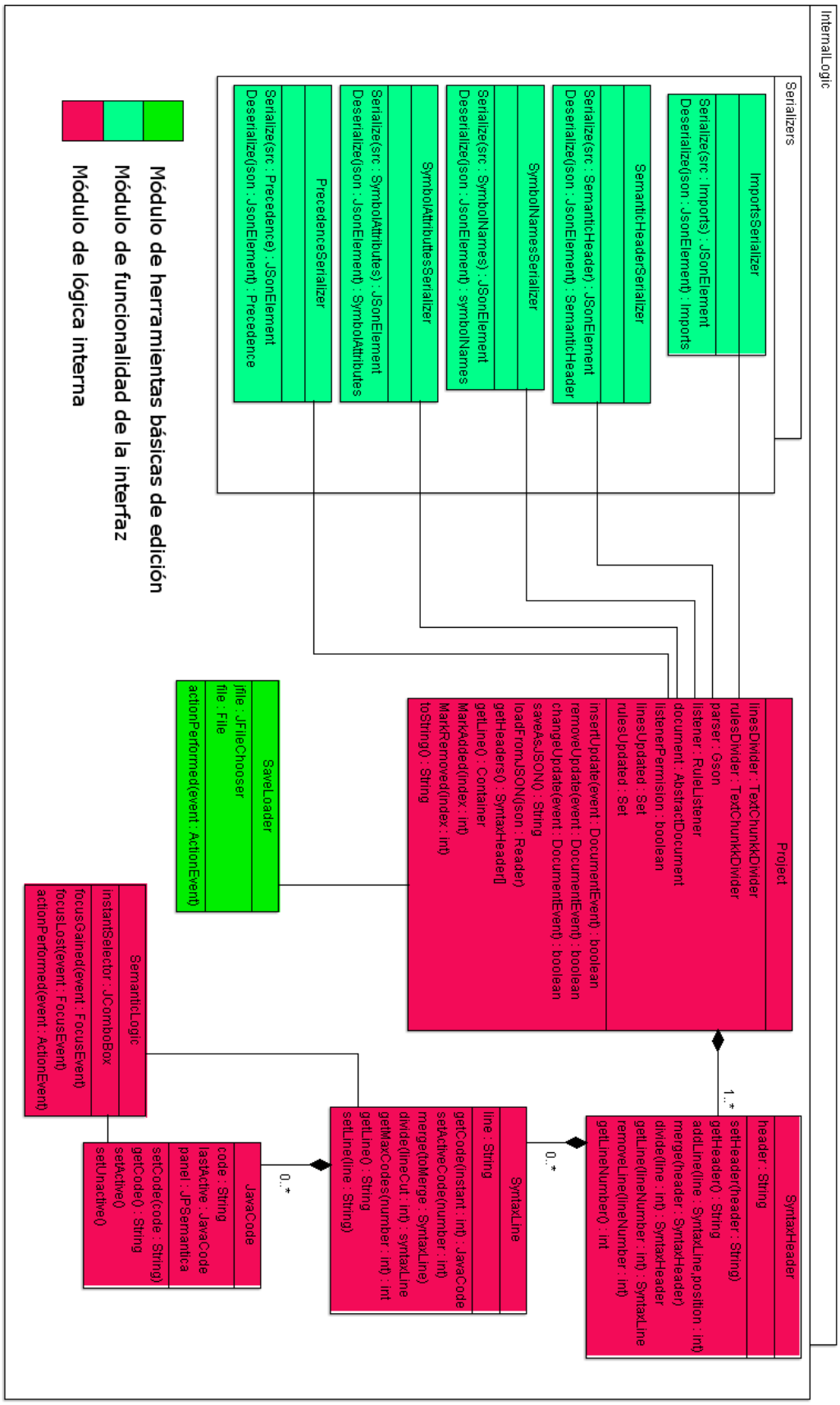

Figura 4-11: Diagrama de clases del módulo de lógica interna

4.7 Módulo de compilado

El módulo de compilado se encarga de la creación de ficheros legibles por la aplicación a la que esta está diseñada para conectarse a partir de los datos almacenados en el módulo de lógica interna, realizando de dicha manera la conexión entre aplicaciones. Lo consideramos de compilado porque implica la traducción de la gramática de Christiansen que está siendo definida en una representación distinta de la utilizada por el editor y comprensible por la aplicación a la que se va a conectar.

Dichos ficheros están divididos en una parte sintáctica y en una parte semántica. La aplicación externa utiliza referencias a las reglas por posición, y una construcción de las reglas por lo que se crea un contenedor semántico, creado a partir de una *Factory* con los datos sintácticos organizados como se requiere. Se utiliza el patrón *Factory* para la creación de la clase *SyntaxHolder* a partir de los datos del programa de manera que sea ella la encargada de la introducción correcta de los datos a partir de los parámetros mínimos necesarios.

En caso de error se para la compilación y se muestra el mensaje del error por el panel de consola.

4.7.1 Compilado de sintaxis

Se ha diseñado un nuevo método para la conexión con la aplicación externa y ficheros extra para implementar en la otra aplicación con el objetivo de leer la sintaxis.

La sintaxis requiere de los siguientes objetos:

-Un símbolo no terminal como axioma.

-Una lista de símbolos no terminales.

-Una lista de símbolos terminales.

-Una lista de reglas, donde cada regla está formada por:

*Un símbolo no terminal como parte izquierda.

*Una lista de símbolos no terminales y terminales como parte derecha.

Una vez se tienen dichos objetos en un contenedor se puede realizar fácilmente la transformación a un fichero que pueda comunicar la sintaxis entre las aplicaciones.

Para generar dicho contenedor hay que transformar los datos de la aplicación obtenidos anteriormente en el módulo de lógica interna:

-Las listas de no terminales y terminales se extraen de la parte de datos globales que se han introducido por el pop-up de símbolos. Según se introducen se comprueba que ningún terminal está en ninguna de las otras dos listas y viceversa

-Posteriormente se crean las reglas mediante las cabeceras sintácticas para crear la parte izquierda de tantas reglas como líneas sintácticas tenga asociada y por último de estas se añaden a cada una de las reglas la parte derecha. Según se va creando cada una de las reglas se comprueban que cada uno de los símbolos de la parte derecha está en una de las dos listas de símbolos y dependiendo en cuál de las dos esté se añade como terminal o no terminal. El de la parte izquierda se comprueba que esté en la lista de no terminales. Si cualquier símbolo no está en ninguna de las otras dos listas se genera un error de compilación.

El axioma es elegido entonces del símbolo no terminal de la parte izquierda de la primera regla.

4.7.2 Compilado de semántica

La aplicación externa utiliza archivos compilados de Java para la creación de la semántica de una gramática. Por lo tanto, para la conexión entre aplicaciones generamos el fichero de código fuente del ".jar" automáticamente, para que el usuario sólo tenga que compilarlo junto los archivos de la aplicación externa.

Por razones de espacio dejamos fuera del ámbito de esta memoria los detalles internos de la aplicación externa a la que debe conectarse el sistema desarrollado en este proyecto. Se ha reflejado esa relación mediante el correspondiente requisito funcional.
El formato concreto del método Java en el que se implementa la semántica en la aplicación externa es un tanto complejo y es lo que ha sido necesario gestionar en esta funcionalidad.
El lector interesado puede encontrar más detalles en la tesis doctoral donde se propone esta aplicación externa [18]

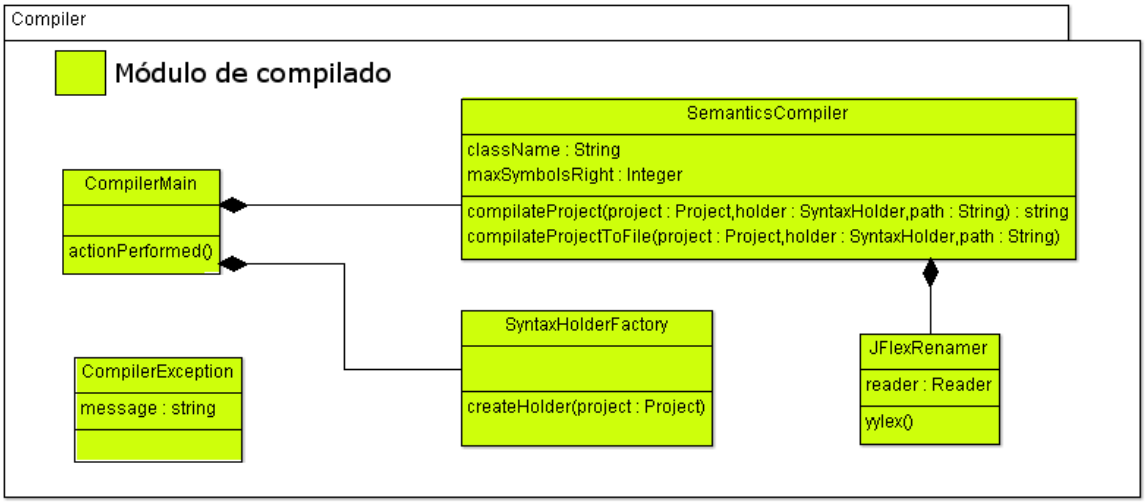

Figura 4-12: Diagrama de clases del módulo de compilado (parte principal)

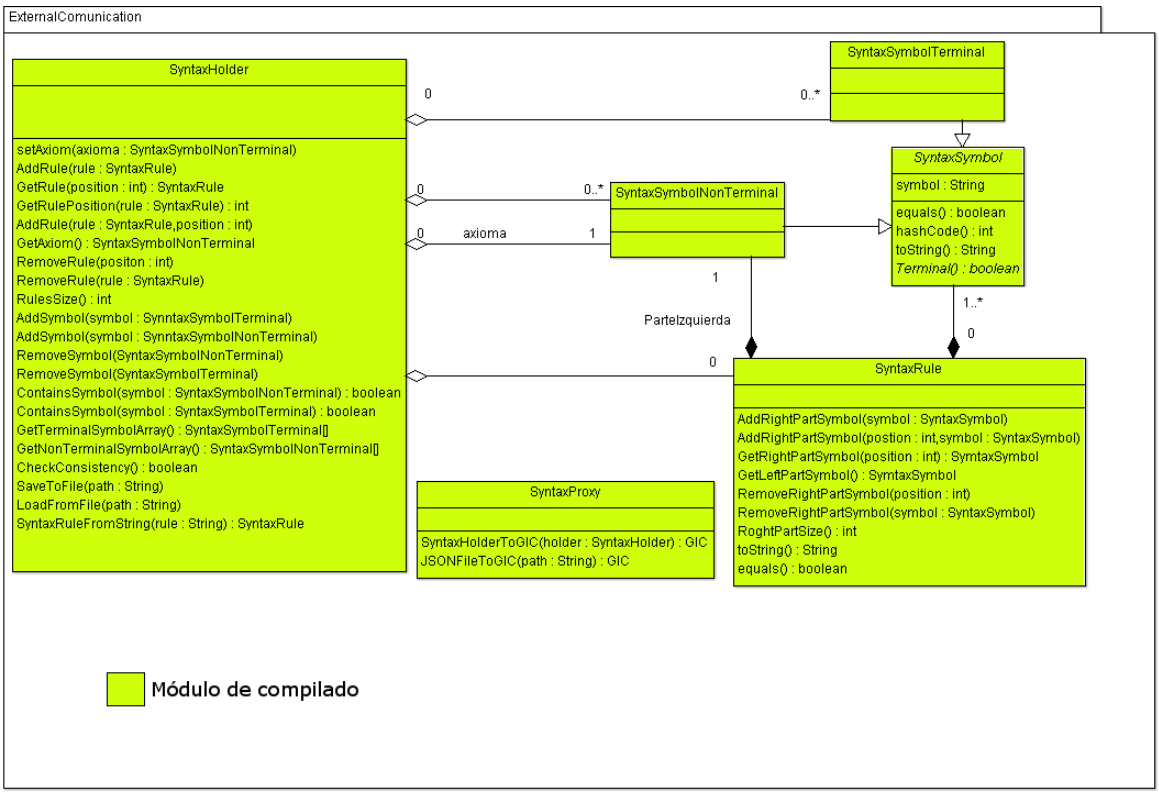

Figura 4-13: Diagrama de clases del módulo de compilado (parte de comunicación externa)

Las clases de la parte principal son *CompilerMain, SyntaxHolderFactory, SemanticsCompiler, JFlexRenamer y CompilerException*.
CompilerMain es la clase llamada para crear la compilación, esta llama a *SyntaxHolderFactory* que se encarga de crear un *SyntaxHolder* de la que puede leer la sintaxis la plataforma objetivo. Luego llama a *SemanticsCompiler* el cual se encarga de compilar la parte de las acciones semánticas para la aplicación objetivo.
Las clases de la parte de comunicación externa son *SyntaxHoder, SyntaxSymbolNonTerminal, SyntaxSymbolTerminal, SyntaxSymbol y SyntaxRule.*
La parte de comunicación externa proporciona ficheros de enlace para las dos aplicaciones, donde *SyntaxHolder* mantiene referencias a todos los símbolos y reglas (representados por *SyntaxRule* y *las* especializaciones de *SyntaxSymbol*). Por su parte *SyntaxProxy* es una clase que fue diseñada para ser incluida en la aplicación objetivo y poder leer el *SyntaxHolder* objetivo.

4.8 La cuestión de la circularidad

Como se ha expuesto inicialmente, englobamos bajo el término "cuestión de la circularidad" a los test de corrección de la gramática que el entorno de desarrollo ofrece.
En concreto estos test son los siguientes

- Detección de que un atributo sintentizado podría quedar sin valor inicial.
- Detección de que un atributo heredado podría quedar sin valor inicial.
- Detección de que el proceso de propagación de algún atributo se puede interrumpir en algún momento del proceso de las cadenas de la gramática.
- Detección de una versión simplificada del test de circularidad inspirado en las versiones originales de Knuth y posteriores [17]

Se ha diseñado un conjunto de estructuras de datos auxiliares

- Para cada regla
    - Su parte izquierda tiene un *HashTree* Java en el que cada uno de sus atributos guarda si en la regla se ha producido sólo lectura o sólo escritura
    - Su parte derecha tiene un *HashTree* Java para cada uno de sus símbolos en los que
        - Cada atributo del símbolo en la posición correspondiente guarda
            - Si ha habido una 1ª lectura antes de esta posición (idem después)
            - Si ha habido una 1ª escritura antes de esta posición (ídem después)

El rellenado de estas estructuras se realiza a la vez que se escriben las reglas.
El chequeo de las condiciones que se quiere realizar se lleva a cabo mediante el uso de 4 flags booleanos para cada atributo de cada símbolo cuyo significado y gestión se resumen a continuación:

- **internal_set**
    - Por defecto vale true
    - Se le asigna false en cuanto se detecta
        - El símbolo del atributo es terminal
        - Hay una regla en la que el símbolo considerado es el de la parte izquierda y el atributo no se ha marcado como escrito
    - En el caso de asignarle el valor false significa que ha habido una interrupción en la propagación del valor de un atributo sintetizado incluida tal vez la inicialización.

- **external_set**
  - Por defecto vale true
  - Se le asigna false en cuanto se detecta
    - Que el símbolo asociado con el atributo está en la parte derecha de alguna regla y en su *HashTree* el atributo considerado no ha sido marcado como escrito antes del instante de la regla en el que está su símbolo.
  - En el caso de valer falso significa que se ha producido acceso a un atributo heredado con la posibilidad de que no tenga valor asignado (ya sea por interrupción de la propagación o por falta de valor inicial)
- **internal_get**
  - Por defecto vale false
  - Se le asigna true en cuanto se detecta que hay una regla que tiene el símbolo considerado en la parte izquierda y tiene en su *HashTree* el atributo valor de lectura
- **external_get**
  - Por defecto vale false
  - Se le asigna true en cuanto se detecta que el símbolo asociado con el atributo está en la parte derecha de alguna regla y en su *HashTree* el atributo considerado tiene el valor de lectura

La gestión de estos flags es la siguiente: se considera que se ha producido un bucle si para un atributo de un símbolo los sets están a false y algún get está a true (si el símbolo asociado es el axioma sólo se mira el internal set excepto en el primer atributo que siempre es heredado y contiene la propia gramática de Christiansen por defecto y para éste se miran los dos sets)

4.9 Diseño de formato de sintaxis y fórmulas de la semántica

4.9.1 Formato de la sintaxis

La sintaxis se ha diseñado de manera que simplifique al máximo posible el trabajo al usuario.

La sintaxis está compuesta por la sucesión de reglas, los símbolos no terminales y terminales y el axioma. Para introducir y definir los símbolos no terminales y los símbolos terminales en la aplicación ya habíamos optado por la utilización de un pop-up con el objetivo de recoger dichos datos, por lo que se centra nuestro interés es en las reglas recogidas en el panel sintáctico.

Cada regla está formada por una parte izquierda compuesta por un no terminal y una parte derecha compuesta por varios símbolos.

Para la separación entre la parte izquierda y la parte derecha se recurre a el carácter ":" (además por términos de visibilidad y funcionamiento de la aplicación tras dicho carácter siempre habrá una nueva línea y no se puede introducir ningún carácter entre ":" y "\n".

Como muchas veces varias reglas comparten la misma parte izquierda, se ha diseñado la sintaxis de manera que se puedan añadir varias partes derechas a una sola parte izquierda, para que el programador no tenga que repetir el código referente a la parte izquierda. Cada parte derecha queda identificada como una nueva línea la cual no tiene ":" (en caso de tenerlos se identificaría como la parte izquierda de la siguiente regla).

Para separar los distintos símbolos simplemente se utilizan espacios entre ellos.

Por último, el axioma de la aplicación será elegido automáticamente por la aplicación siendo la parte izquierda de la primera regla.

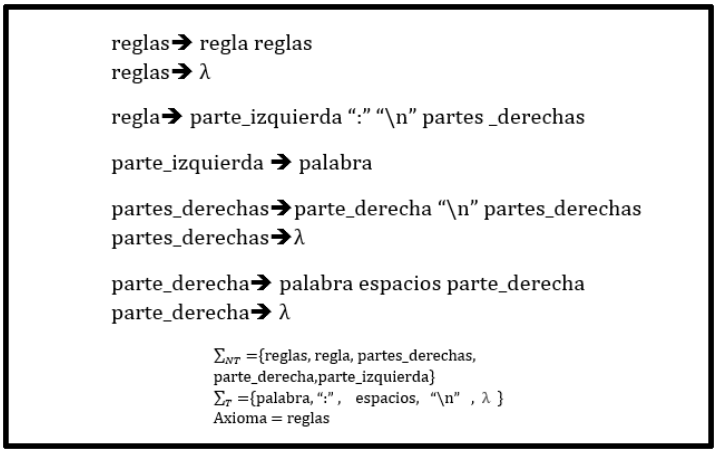

Figura 4-14: Formalización de la sintaxis

4.9.2 Fórmulas para acceso a los atributos

Debido a que el símbolo "$" no está reservado para la creación de código en Java y que se utiliza para introducir información en diversas aplicaciones de gramáticas de atributos se utilizará para acceder a los atributos en esta aplicación también en el código.
La manera por la que se accede a atributos de uno de los elementos de la parte derecha de la regla de la sintaxis sería "$X.<nombre del atributo>" donde X selecciona la posición que ocupa dicho elemento en la parte derecha y el <nombre de atributo> es el nombre de los atributos definidos para dicho símbolo (estos se pueden tener a mano con el panel de información sintáctica y clicar encima de ellos para que se escriban automáticamente en el panel de semántica).
Para seleccionar atributos del símbolo no terminal de la parte izquierda de la regla se utiliza la misma notación que antes pero con "$$" en lugar de "$X", es decir quedaría: "$$.<nombre del atributo>.

Para la funcionalidad asociada con la cuestión de la circularidad se necesita distinguir los accesos a los atributos entre consulta de su valor y escritura del mismo. Por esa razón se ha añadido una notación específica distinta para cada caso, a saber:

- Para la escritura de un valor de un atributo se utilizará la notación
  $X.<nombre del atributo>(<expresion>)
  - Si esa para un símbolo en la parte derecha de la regla
    $$.<nombre de atributo>(<nuevo valor>)
  - Si esa para un símbolo en la parte izquierda de la regla
- Para la lectura de un valor de un atributo se utilizará la notación
  $X.<nombre del atributo>
  - O
    $$.<nombre de atributo>
  - Si esa para un símbolo en la parte izquierda de la regla

4.9.3 Fórmulas para acceso a las reglas

Las gramáticas de Christiansen al ser generativas crean una dependencia entre la sintaxis y la semántica que las gramáticas de atributos no tenían. Es por esto que hay que diseñar una manera para comunicar entre dichas dos partes.
La selección de reglas en la parte de compilación se hace mediante su construcción y luego la referenciación de dicha construcción lo cual impide introducir el código directamente a

los usuarios puesto que dicho implementación violaría la encapsulación de la compilación al tener que saber el usuario como se genera la compilación en orden de querer modificar las reglas desde la semántica.
Para acceder a las reglas se han diseñado de manera que tienes que poner la regla completa utilizando otra vez el símbolo "$" utilizado en las otras partes del diseño, puesto que así nos encontramos con la ventaja de que dicho símbolo no está reservado para la creación de código en Java.
La manera de seleccionar una regla es de la siguiente manera $(<regla sintáctica>) donde la regla sintáctica es escrita en una sola línea y con una sola parte derecha de la misma manera que fue especificado como escriben reglas sintácticas en 4.8.1.

5 Desarrollo. Librerías.

Durante el desarrollo de la aplicación se han utilizado diversas librerías externas e internas que se han utilizado intensivamente durante el desarrollo de la aplicación.

5.1 Java Swing

Swing es una librería gráfica integrada en el propio *core* de Java, desarrollada por Oracle.
Incluye (aunque no se limita a ello) una gran variedad de elementos que permiten la visualización y obtención de datos por un sistema basado en eventos, como ventanas, paneles, botones, menús, campos de textos, editores de texto, etc…
Se ha elegido esta librería gráfica puesto que el autor de la aplicación ya tenía experiencia previa, con lo que no había que utilizar mucho tiempo en el aprendizaje y el funcionamiento de elementos.

Algunas otras ventajas que ofrece son:
-Gran modularidad en su arquitectura y una alta independencia entre los componentes. Su diseño está orientado y altamente adaptado al patrón de modelo-vista-controlador.
-Un número bastante elevado de elementos con los que desarrollar interfaces gráficas adaptadas a la aplicación y no al revés.
-Alta independencia entre sistemas operativos al estar desarrollada como estándar para ser utilizada por Java.
-Sin necesidad de incluir ficheros externos en el proyecto al ser una librería interna de Java.
-Mucha documentación, ejemplos de uso y tutoriales.
-Continuamente en actualización y mejora desde los inicios de Java por lo que es bastante poco probable encontrarse con algún bug no solucionado.

En cambio, al utilizar Swing nos limitamos ya que ofrece menor flexibilidad y funcionalidad que otras librerías no estándares.

No obstante, como la apariencia gráfica de la aplicación no es tan importante como la funcionalidad en este caso, tenemos suficiente con la librería swing.

5.2 JFlex

JFlex es un generador de analizadores léxicos para Java desarrollado por "JFlex Team".
Un generador analizadores léxicos utiliza como entrada una especificación sobre un conjunto de expresiones regulares y sus correspondientes acciones como entrada. A partir de esto genera un programa que lee texto de entrada y lo compara con las especificaciones y ejecuta las acciones correspondientes si la expresión regular asociada se cumple.

Estos analizadores se han requerido para analizar el texto que el usuario va escribiendo y resaltarlo en el módulo de resaltado de sintaxis, así como para el análisis del código escrito en Java en la aplicación y su integración en el compilado.

Entre las ventajas que tiene JFlex encontramos:
-Desarrollado para imitar a Flex, el generador de analizadores léxicos de C, con el cual el autor tenía experiencia previa.
-Soporte de código unicode.
-Independencia de la plataforma.
-Los analizadores léxicos generados son bastante rápidos al basarse en autómatas finitos no deterministas.
-Incluye algo de especificación sintáctica.
-Desarrollado para funcionar en conjunto con JCup (al final no resultó necesario utilizar dicha librería, pero en un principio se pensaba que sería necesaria).
-Documentado en su página web y utiliza especificaciones estándar.

Algunos inconvenientes que se han encontrado en su utilización:
-Sin actualizaciones desde hace tres años.
-El código de especificación de JFlex no tenía soporte para el entorno de desarrollo utilizado.

5.3 Gson

Gson son librerías para la serialización/deserialización de objetos en Java en formato JSON creada por Google para Java.
La serialización de objetos consiste en la transformación del objeto en datos para su almacenamiento. La deserialización es la transformación de dichos datos de nuevo en objetos.
JSON (acrónimo de JavaScript Object Notation) es un formato de texto ligero para el intercambio de datos.
La librería se ha utilizado para la serialización y la deserialización de los datos en el programa para su posterior almacenamiento, así como para la creación del fichero sintáctico que actúa como conexión entre las aplicaciones.
El autor no tenía experiencia alguna en JSON y la idea original era buscar alguna librería de serialización en C, no obstante, las ventajas de JSON como una fácil lectura del archivo generado, que es más ligero y que al estar de moda hay librerías hasta debajo de las piedras, hizo que el autor se decantara por este formato de serialización.

Las ventajas de la librería de Google con respecto a otras librerías son:
-Altamente documentada, comunidad y desarrolladores activos.
-Incluye formato para caracteres Unicode.
-Permite la creación de serializadores y deserializadores personalizados.
-Fácil de usar.

6 Pruebas y resultados

Este proyecto estaba altamente focalizado en el diseño de la aplicación y de un método para resolver el problema de la circularidad. Las pruebas se han limitado a comprobar que

cada módulo funciona y se relaciona internamente de manera correcta con cada otro módulo de la aplicación.
El correcto funcionamiento de cada módulo se ha comprobado mediante pruebas de caja blanca. Una vez escrito el módulo se han ejecutado al menos una vez cada bloque dentro de un bucle y cada ramal de bifurcación.
Cuando el módulo se creía suficientemente estable se integraba con el resto de la aplicación y se ejecutaban en todos los módulos conectados (este incluido) pruebas de caja negra relativas a casos extremos y comunes relacionados con la intercomunicación de los módulos. Estos casos se han diseñado a partir de la definición de cada módulo y de manera específica para cada uno.
Para la comunicación entre la aplicación y la plataforma, se han utilizado ficheros diseñados a mano a partir de cierta sintaxis y acciones semánticas, que la plataforma objetivo era capaz de leer. posteriormente se han introducido dicha sintaxis y acciones semánticas en la aplicación y se ha comprobado que el fichero generado automáticamente era igual al esperado. tras esto la versión de la aplicación era testeada por los creadores de la plataforma y se creaban nuevos ficheros manualmente con los casos en los que la aplicación fallara para una nueva iteración.

7 Conclusiones y trabajo futuro

7.1 Conclusiones

En este trabajo se ha desarrollado un entorno de desarrollo para gramáticas de Christiansen, con un test capaz de identificar problemas sobre circularidad de atributos.
Debido a que es el primer trabajo desarrollado de este tipo para dichas gramáticas, no ha habido punto de comparación para la aplicación, no obstante, se han podido inspirar e ir comprobando muchos de los módulos que se utilizan con otros de otras aplicaciones con funcionalidad similar.
Para la interfaz de la aplicación se ha utilizado un modelo basado en eventos para aprovechar las capacidades de Swing. Cada input del usuario genera una serie de medidas con la aplicación.
La serialización y deserialización de los datos almacenados por la aplicación, necesaria para su recuperación ha hecho uso de JSON y la gran documentación existente al respecto.
El resaltado por su parte ha sido más complicado puesto que no se han encontrado librerías reutilizables por lo que ha tenido que ser implementado utilizando métodos inspirados en otras aplicaciones.
Para lograr la comunicación entre la aplicación y la plataforma objetivo se han diseñado métodos capaces de generar los archivos requeridos por la plataforma objetivo a partir de los datos obtenidos desde la aplicación.
Finalmente se han utilizado diversas fuentes y artículos para encontrar maneras eficientes de implementar un test de la circularidad, lo cual ha permitido el desarrollo de un algoritmo propio de manera que se pueda informar al usuario sin ralentizar el compilado del proyecto de la aplicación.

7.2 Trabajo futuro

Los módulos dedicados a la edición de la aplicación son bastante básicos. Se podrían añadir otros para aumentar la funcionalidad y mejorar la velocidad de desarrollo de

proyectos basados en gramáticas de Christiansen como refactorización, soporte para má lenguajes, programación visual, etc.
Además, el resaltado es bastante básico y se podrían mejorar las capacidades que ofrece haciendo un precompilado de los archivos para resaltar variables, mostrar documentación en los atributos y ofrecer soluciones de autocompletado.
El entorno de desarrollo se ha hecho como objetivo de crear una aplicación para un problema que todavía no tenía solución, no obstante, ahora que ya hay una solución básica, se podrían realizar pruebas de usabilidad sobre la aplicación para adaptarla mejor a los usuarios que la puedan utilizar.
Aunque uno de los objetivos principales de la aplicación era conseguir la creación de un test eficaz de circularidad y el test ha sido implementado, dicho test todavía es muy básico y hay algunos puntos en los que se podría mejorar en un futuro:
-Actualmente solo detecta si ha habido una violación de circularidad, no obstante, aunque recoge mucha información para ello, no da al usuario una idea aproximada de donde se comente dicha violación. Se podría procesar dicha información para ayudar al usuario a corregir dichas violaciones.
-El test de circularidad se basa en el análisis léxico para la detección de escrituras y lecturas, por lo que la detección de la violación puede fallar si se utiliza alguna estructura condicional dependiente de la sintaxis para obviar alguna lectura (por ejemplo, si una lectura solo se encuentra en una de las ramas de una bifurcación). Se podría utilizar análisis de la sintaxis para evitar dichos problemas.

Referencias

[1] Knuth D.E. , “The genesis of attribute grammars. In: Deransart P.”, Jourdan M. (eds) Attribute Grammars and their Applications. Lecture Notes in Computer Science, vol 461. Springer, Berlin, Heidelberg, 1990

[2] A. Ortega, M. de la Cruz and M. Alfonseca, "Christiansen Grammar Evolution: Grammatical Evolution With Semantics," in IEEE Transactions on Evolutionary Computation, vol. 11, no. 1, pp. 77-90, Feb. 2007.

[3] Corbett, Robbert. “Bison”, Free Software Fundation, 1985, https://www.gnu.org/software/bison/,

[4] Paxon, Vern Edward. “Flex”, 1987, https://github.com/westes/flex

[5] Parr,Terrence . “ANTLR”, 1990, http://www.antlr.org/

[6] Peter Linz,”An Introduction to Formal Languages and Automata”, (2nd Ed.). D. C. Heath and Company, Lexington, MA, USA.,1996.

[7] Knuth, D.E. ,”Semantics of context-free languages” in Mathematical systems theory,vol. 2, no. 2 pp.127-145,Jun.1968

[8] H. Christiansen. 1990. A survey of adaptable grammars. SIGPLAN Not. 25, 11 (November 1990), 35-44.

[9] Michael Rodeh and Mooly Sagiv. 1999. Finding circular attributes in attribute grammars. J. ACM 46, 4 (July 1999), 556-ff

[10] Jazayeri, Mehdi, “The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars”,

[11] Wu, Pei-Chi & Wang, Feng-Jian. (1995). “A Worst Case of Circularity Test Algorithms for Attribute Grammars.”. ACM Trans. Program. Lang. Syst.. 17.

[12] Bruno Courcelle, Paul Franchi-Zannettacci, "Attribute grammars and recursive program schemes I", Theoretical Computer Science, Volume 17, Issue 2, 1982, pp 163-191

[13] Bruno Courcelle, Paul Franchi-Zannettacci, “Attribute grammars and recursive program schemes II”,Theoretical Computer Science,Volume 17, Issue 3,1982,pp 235-257

[14] Jourdan, Martin. An efficient recursive evaluator for strongly non-circular attribute grammars. Diss. INRIA, 1983.

[15] Matthew Belmonte. A practical attribute grammar circularity test. Technical Report TR 88-920, Department of Computer Science Cornell University, June 1988.

[16] Pei-Chi Wu and Feng-Jian Wang. 1995. A worst case of circularity test algorithms for attribute grammars. ACM Trans. Program. Lang. Syst. 17, 2

[17] Michael Rodeh and Mooly Sagiv. 1999. Finding circular attributes in attribute grammars. J. ACM 46, 4 (July 1999), 556-ff..

[18] Cruz Echeandía, Marina de la .” Evolución gramatical y semántica”, Tesis doctoral inédita. Universidad Autónoma de Madrid, Escuela Politécnica Superior, junio de 2010

Glosario

JSON	JavaScript Object Notation
IDE	Integrated Development Environment
ANTLR	ANother Tool for Language Recognition
FLEX	Fast LEXical analyzer generator
XML	EXtensive Markup Language

Anexos

A Manual de usuario

Descripción de la aplicación

La aplicación "Christenseen Grammals IDE" es una interfaz gráfica desarrollada con la intención de facilitar el desarrollo de Gramáticas de Christenseen. La aplicación está desarrollada en java (versión 1.8 o superior), por lo que en principio podría ser utilizada independientemente del sistema operativo, no obstante, se ha desarrollado y testeado unicamente en ubuntu, por lo que el autor se desentiende de cualquier problema surgido como consecuencia de usarlo en otro sistema operativo.

Ventana principal de la aplicación

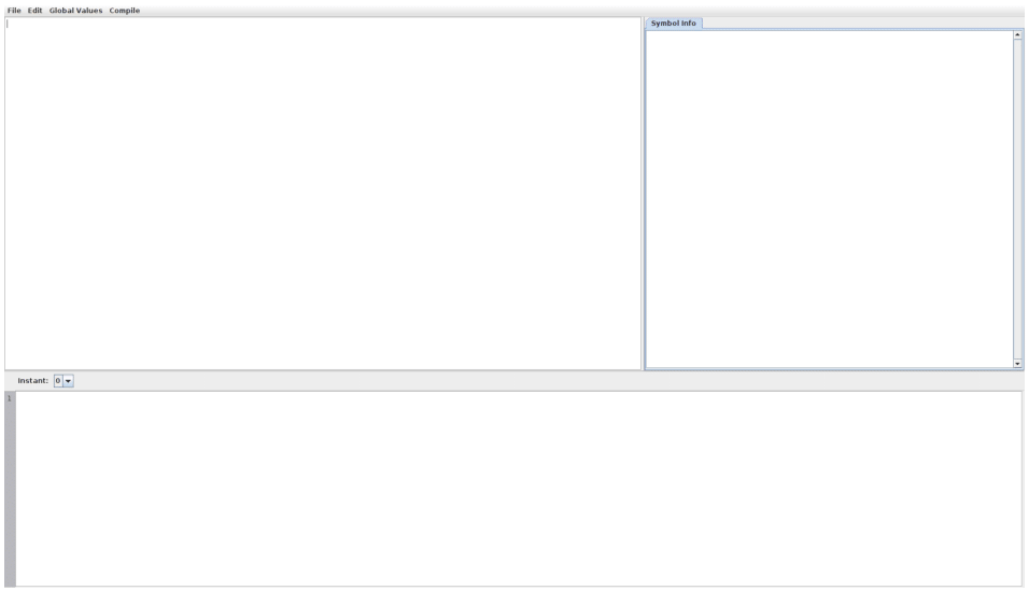

Figura: Ventana principal de la aplicación

El área de la sintáxis

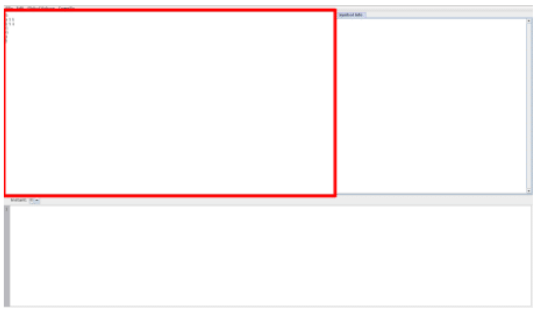

Figura: Área de sintáxis

En esta zona se escribirá la sintaxis de la gramática que estemos desarrollando.
La sintaxis en gramática de atributos está compuesta por reglas sintácticas de manéra que un símbolo no terminal esta en la parte izquierda de la regla seguido de una separador que actua como asignador y seguido de la parte derecha que son una sucesión de simbolos tanto terminales como no terminales (Por ejemplo en pseudocódigo: $S \Rightarrow aSb$).

El área de la sintáxis

La aplicación interpreta las lineas escritas en dicha ventana como partes derechas o izquierdas de la regla.
Una línea escrita como parte izquierda se indicara escribiendo " : " lo cual insertará automáticamente una nueva linea al final de esta que será interpretada como la parte derecha de la regla.
En las reglas izquierdas los simbolos no terminales deberan estar separados por espacios.
Una parte derecha puede tener varias partes izquierdas para ahorrar trabajo al programador.

Por ejemplo se puede escribir $\begin{matrix} S \Rightarrow aSb \\ S \Rightarrow bAs \end{matrix}$ como $\begin{matrix} S: \\ a\ S\ b \\ b\ A\ s \end{matrix}$

El área de la sintáxis

Por último cabe mencionar que el contenido que está en el área de sintaxis es filtrado, por lo que no aceptará carácteres en caso de que crea que deje la sintaxis insostenible y arreglará automáticamente cualquier situación insostenible que deje el usuario eliminando o insertando carácteres.
El área de sintaxis acepta:

- Palabras que comiencen por letra y solo contengan letras o numeros, para indicar los nombres de los símbolos
- Cualquier carácter rodeado de comillas simples, para indicar nombres de símbolos terminales literales
- Espacios, saltos de línea y ":"

El área de la semántica

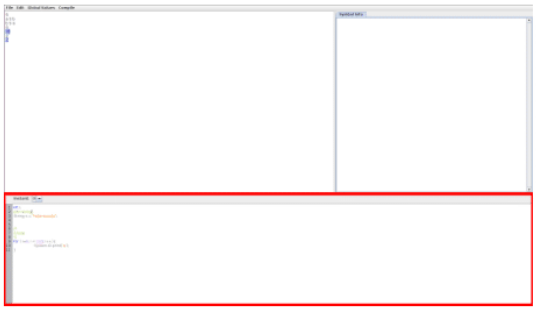

Figura: Área de semántica

Esta zona se utiliza para escribir la semántica de una regla sintáctica en código java. Para seleccionar una regla de la que queramos añadir (o modificar) semántica simplemente basta con clickar en la parte izquierda de dicha regla en la zona de sintaxis y luego clickar en cualquier en la zona de semántica.

El programa resaltará la regla seleccionada en la zona de sintáxis mientras esta no tenga el foco. Si el cursor está sobre la parte derecha de la regla, el programa no dejará editar en el área de semántica (incluyendo al abrir un nuevo documento hasta que se escriban el primer ":").

El área de la semántica

En el editor semántico se pueden detectar tres zonas diferenciadas:

- El selector de instante
- El numerador de lineas
- La zona de escritura

El área de la semántica

El selector de instante

El selector de instante es la caja de selección en la zona superior. Cada regla puede tener código asociado a distintos instantes. Este código se ejecutará en el momento en el que la derivación haya alcanzado el mismo número de símbolos que de instantes. El programa automáticamente detecta el numero de símbolos que hay en la sintaxis y deja añadir a la regla sintactica seleccionada dicho número de instantes más uno.
El código mostrado en la zona de escritura cambiará automaticamente al seleccionar otro instante, por el codigo seleccionado para dicho instante.

El área de la semántica

El numerador de líneas

El contador de líneas está situado en la zona izquierda de la zona semántica.
Simplemente es un numerador que se actualiza según el número de lineas que tenga él area de escritura, de manera que cada linea en está queda asociada a un número.

El área de la semántica

La zona de escritura

La zona de escritura ocupa la mayoría de espacio en el resto de la zona semántica.
Está diseñada para mostrar y dejar editar el código semántico asocidado a un instante de una regla sintactica.
Tiene también en funcionamiento un módulo que resalta con distinto color ciertas palabras según se va escribiendo. Resalta:

- Una lista preintroducida en un fichero con las palabras reservadas de Java.
- Números literales.
- Carácteres literales (indicados mediante comillas simples).
- Cadenas de texto literales (indicadas mediante comillas dobles).
- Comentarios (indicados por //) y comentarios multilineados (indicados por "/*" y "*/").
- Referencias al código sintáctico

El área de la semántica

Cómo referenciar a la sintáxis

Al ser las gramáticas de Christiansen de atributos y generativas, requieren que las acciones semánticas puedan referenciar la sintaxis. Actualmente se pueden referenciar de la sintaxis en el panel semántico reglas y atributos.

Referenciando reglas

Para referenciar una regla hay que escribirla en la acción semántica como $(< *Regla sin salto de linea* >) . En el siguiente ejemplo referenciamos la segunda regla de la sintaxis en semántica:

Sintaxis	*Accion semantica*
S :	
$a\ S\ b$	$(S : b\ S\ a)$
$\underline{b\ S\ a}$	

El área de semántica

Cómo referenciar atributos

Referenciando atributos para lectura

Para referenciar atributos para lectura hay que escribirlos como: $<Número del símbolo en la parte derecha>.< Nombre de atributo> si el símbolo se encuentra en la parte derecha de la regla ó $$.< Nombre de atributo> si se quiere referenciar al símbolo de la parte izquierda. El valor posteriormente se puede asignar a una variable. Por ejemplo:

```
char caracter;
caracter = $0.letter;
int cuenta;
cuenta = $$.count;
```

El área de semántica

Cómo referenciar atributos

Referenciando atributos para escritura

Para referenciar atributos para escritura hay que escribirlos como: $<Número del símbolo en la parte derecha>.< Nombre de atributo>(< Valor a asignar >) si el símbolo se encuentra en la parte derecha de la regla ó $$.< Nombre de atributo> (< Valor a asignar >) si se quiere referenciar al símbolo de la parte izquierda. Por ejemplo:

```
$0.letter('a');
int cuenta = 2;
$$.count(cuenta + 1);
```


El área auxiliar

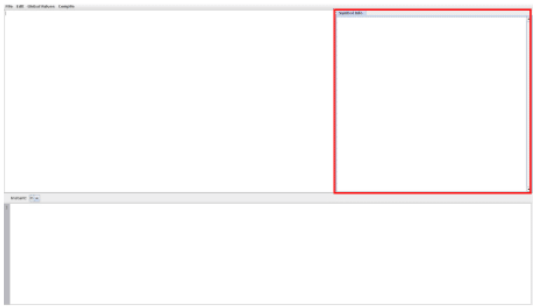

Figura: Área auxiliar

El area auxiliar está diseñada para mostrar diversas informaciones en la pantalla principal.
Para cambiar entre sus modos basta con clickar las pestañas en la parte superior.

El área auxiliar

Panel sobre información sobre símbolos

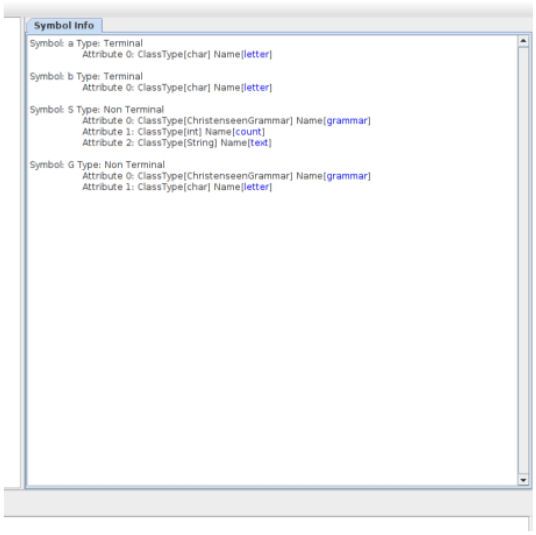

Figura: Panel de información sobre símbolos

Este panel muestra un resumen de la información introducida en la ventana de Símbolos. Los nombres de los atributos de los símbolos aparecen resaltados en azul y clickar sobre ellos los escribe automaticamente en el área de semántica (si es posible escribirlos). Al estar diseñado para gramáticas de Cristiansen, todo símbolo no terminal declarado tiene por defecto el atributo de la gramática.

El área auxiliar

Consola

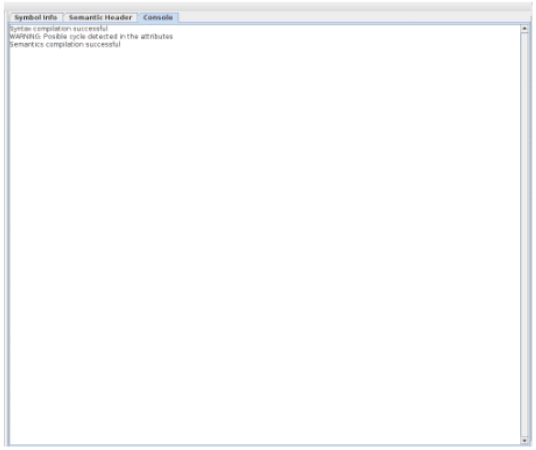

Figura: Panel de consola

Este panel muestra información relacionada con la compilación. Una vez se haya indicado al programa que compile el proyecto, mostrará si la compilación ha sido existosa o en caso contrario que error ha ocurrido y una posición aproximada del lugar dónde ha ocurrido. También avisa si se ha detectado algún bucle en la gramática diseñada.

El área auxiliar

Cabecera semántica

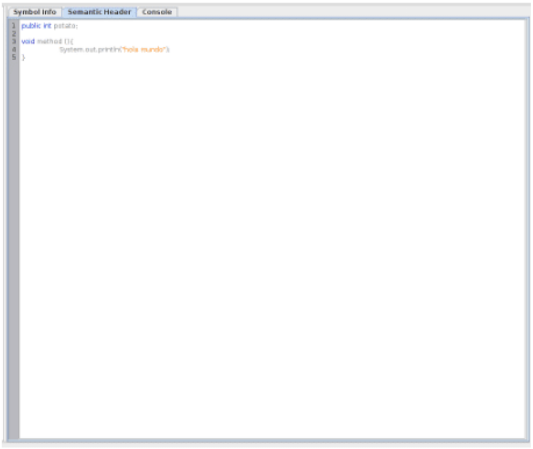

Figura: Panel de cabecera semantica

Este panel
es un panel que se puede editar.
Esta
diseñado para declarar variables
y métodos para la clase java
que se creará en la compilación. Lo
que se declare aquísera visible para
todos los métodos de las reglas.

La barra de menú

Figura: La barra de menú

La barra de menú esta situada en
la parte superior de la aplicación.
Esta ofrece diversas utilidades
mediante botones desplegables,
algunos con combinaciones de

botones de acceso rápido.

La barra de menú

Figura: La barra de menú

La barra de menú esta situada en la parte superior de la aplicación. Esta ofrece diversas utilidades mediante botones desplegables.

La barra de menú

File

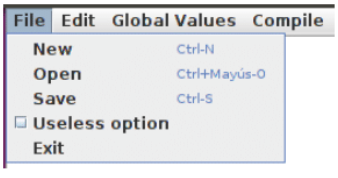

Figura: El submenú de File

El botón de File recoge las siguientes funcionalidades:

- New:
  Crea un nuevo proyecto.
- Open:
  Abre un proyecto previamente guardado desde un fichero.
- Save:
  Guarda el proyecto en un fichero (la primera vez pregunta donde, a partir de entonces actualiza dicho fichero).
- Exit:
  Sale de la aplicación

La barra de menú

Edit

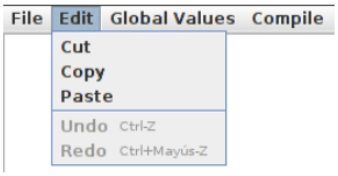

Figura: El submenú de Edit

El botón de Edit recoge las siguientes funcionalidades:

- Cut:
  Elimina el texto seleccionado tras copiarlo al portapapeles.
- Copy:
  Copia el texto seleccionado al portapapeles.
- Paste:
  Pega el texto guardado en el portapapeles.
- Undo:
  Deshace la última acción (solo funciona en el area de semántica).
- Redo:
  Rehace la última acción desecha por undo.

La barra de menú

Global Values

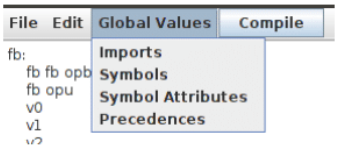

Figura: El submenú de Global Values

El submenú de Global Values incluye varios botones que abren popups con el objetivo de introducir valores globales en ellos para la aplicación.

- Symbols:
  Abre el popup de símbolos donde se declaran los símbolos.
- Symbols Attributes:
  Abre el popup de atributos de símbolos donde se declaran los atributos de cada símbolo.
- Imports:
  Abre el popup de imports donde se indican las librerías externas que se quieren usar.
- Precendeces:
  Abre el popup de precendencias donde se introducen los valores correspondientes a la prioridad precedencia de las derivaciones

La barra de menú

Global Values - Ventana de Símbolos

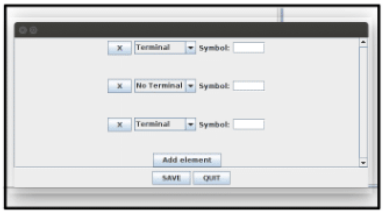

Figura: Popup de Símbolos

La ventana de símbolos tiene un botón "*SAVE*" que guarda los cambios realizados y la cierra, "*QUIT*" que los descarta y la cierra, y "*Add element*" que añade un nuevo símbolo.
Cada símbolo cuenta con (en este orden):

- Un botón para eliminarlo de la lista.
- Un selector para indicar si el símbolo terminal o no terminal.
- Un campo para poner el nombre del símbolo

La barra de menú

Global Values - Ventana de Atributos de Símbolos

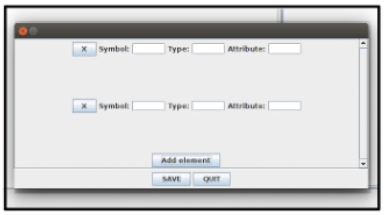

Figura: Popup de Atributos de Símbolos

La ventana de atributos de símbolos tiene un botón "*SAVE*" que guarda los cambios realizados y la cierra, "*QUIT*" que los descarta y la cierra, y "*Add element*" que añade un nuevo atributo a un símbolo.
Cada atributo cuenta con (en este orden):

- Un botón para eliminarlo de la lista.
- Un campo para escribir la referencia al símbolo del que se añadira el aributo.
- Un campo para indicar la clase o tipo de dato que será el atributo.
- Un campo para indicar el nombre del atributo

La barra de menú

Global Values - Ventana de Imports

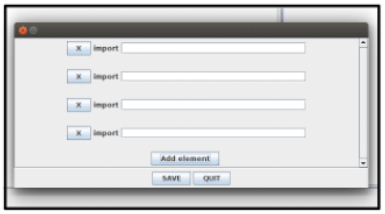

Figura: Popup de Imports

La ventana de imports tiene un botón "*SAVE*" que guarda los cambios realizados y la cierra, "*QUIT*" que los descarta y la cierra, y "*Add element*" que añade un nuevo import. Cada import cuenta con (en este orden):

- Un botón para eliminarlo de la lista.
- Un campo para indicar la librería o clase a importar (Se añadira como import en el documento java generado).

La barra de menú

Global Values - Ventana de Precedencias

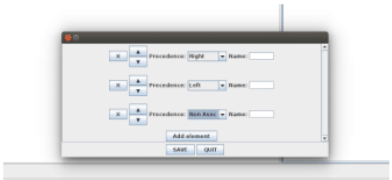

Figura: Popup de Precedencias

La ventana de precedencias tiene un botón "*SAVE*" que guarda los cambios realizados y la cierra, "*QUIT*" que los descarta y la cierra, y "*Add element*" que añade una nueva precedencia.
El orden de las precedencias cuenta para indicar su prioridad. Cada precedencia cuenta con (en este orden):

- Un botón para eliminarla de la lista.
- Un par de botones para intercambiar su posición.
- Un selector para indicar si será izquierda, derecha o no asociativa.
- Un campo para escribir la referencia al símbolo sobre el que se aplica la precedencia.

La barra de menú

Global Values - Botón de Compilación

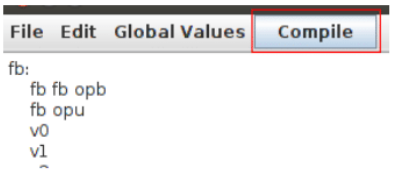

Figura: Botón de compilación

Al pulsar este botón la aplicación intenta generar los ficheros de sintaxis y semántica utilizables por la plataforma. La información sobre la generación aparecerá en el menú de consola.

B Repositorio del código de la aplicación

https://bitbucket.org/rperez01/editor-gramaticas-cristenseen