Cadena de Procesamiento Lingüístico para el Español

Proyecto Fin de Carrera

Daniel Fernández González

Área de Ciencias de la Computación e Inteligencia Artificial
Escuela Superior de Ingeniería Informática
Universidad de Vigo

Julio 2010

Cadena de Procesamiento Lingüístico para el Español

Proyecto Fin de Carrera
Departamento de Informática
ENI-316

Dirigida por el Doctor
Manuel Vilares Ferro

Área de Ciencias de la Computación e Inteligencia Artificial
Escuela Superior de Ingeniería Informática
Universidad de Vigo

Julio 2010

"Computers are incredibly fast,
accurate and stupid.
Humans are incredibly slow,
inaccurate and intelligent.
Together they are a power that
exceeds the imagination"
Albert Einstein

A mi familia

Agradecimientos

A todos los que la presente vieren y entendieren.

Inicio de las Leyes Orgánicas.
Juan Carlos I

Este es el momento de mostrar mi agradecimiento a todas las personas que han contribuido de algún modo en el desarrollo de este proyecto.

En primer lugar, a mi director, Manuel Vilares. Su excelente labor como guía y consejero han hecho posible alcanzar la conclusión del trabajo. Además, agradecerle especialmente la oportunidad brindada y la confianza depositada en mí.

Mostrar mi gratitud también al investigador Eric De La Clergerie, por su paciencia e implicación, sin la cual este proyecto no habría sido posible.

Gracias también a los miembros del grupo Cole, que no han dudado en ofrecerme su ayuda para hacer frente a cualquiera de los problemas surgidos a lo largo de esta etapa.

Por último, y no por ello menos importante, quiero dar las gracias a todas aquellas personas sin las cuales, a pesar de no estar involucradas de forma directa con el proyecto, uno no obtendría los resultados alcanzados hasta el momento. Entre ellas es necesario incluir a mi familia y amigos.

A todos y cada uno, gracias.

La realización de este trabajo se enmarca dentro del proyecto de investigación *Búsqueda de respuestas empleando metagramáticas* P.P. M709122P 6406211, financiado por el *Ministerio de Educación y Ciencias.*

Índice

Índice de figuras

Índice de Tablas

Parte I

Memoria

Capítulo 1

Introducción

1.1. Identificación del proyecto

Con el fin de facilitar la identificación unívoca de este proyecto, la tabla 1.1 muestra los datos administrativos referentes al mismo.

Código	**ENI-316**
Título	**Cadena de Procesamiento Lingüístico para el Español**
Autor	**Daniel Fernández González**
Director	**Manuel Vilares Ferro**
Departamento	**Informática**
Área	**Ciencias de la Computación e Inteligencia Artificial**
Titulación	**Ingeniería Informática**

Tabla 1.1: Identificación del proyecto.

1.2. Organización de la documentación

La documentación es una parte esencial en el desarrollo de una aplicación informática. Su objetivo fundamental es describir el funcionamiento del sistema, las distintas etapas necesarias para su desarrollo, la tecnología empleada para tal cometido y, sobre todo, señalar la aplicabilidad del *software* en construcción e indicar posibles ampliaciones. Para su mejor comprensión, el texto se ha dividido en tres bloques: la *Memoria*, el *Manual Técnico* y el *Manual de Usuario*. A continuación se detallan los contenidos de cada uno de ellos.

1.2.1. Memoria

Esta sección presenta una descripción del trabajo realizado. Además, sirve de introducción a cualquier persona que vaya a mantener o continuar su desarrollo. Recoge los siguientes capítulos:

1. Introducción: Se identifica el proyecto, se describe la organización de la documentación presentada y se ofrece una visión general del sistema objeto del mismo.

2. Descripción del sistema: Se incluye una breve descripción del problema propuesto y de los objetivos que implican su solución. También se detalla la arquitectura del sistema desarrollado, así como el entorno de implantación del mismo.

3. Desarrollo del proyecto: Se describen de forma breve la metodología, la plataforma *hardware* y las tecnologías y herramientas *software* empleadas.

4. Planificación y presupuesto: Se presenta la planificación inicial realizada y se compara con el desarrollo final del trabajo. También se detalla el presupuesto necesario para llevarlo a cabo, teniendo en cuenta tanto los recursos físicos como humanos.

5. Problemas, conclusiones y posibles ampliaciones: Contiene una relación de las dificultades encontradas durante el desarrollo del proyecto, las ventajas que aporta el sistema implementado y las posibles ampliaciones que se podrían acometer en un futuro.

1.2.2. Manual Técnico

En este documento se incluye la información necesaria para aquellas personas que se encarguen del mantenimiento y/o modificaciones del sistema. El *Manual Técnico* está compuesto por los siguientes capítulos:

1. Análisis: Contiene la especificación de los requerimientos de la aplicación desarrollada y la descripción de todos los casos de uso y clases que componen el sistema, así como los diagramas de secuencia y de colaboración obtenidos a través de los distintos escenarios.

2. Diseño: Refinamiento de los diagramas y especificaciones realizadas en el análisis, obteniendo un mayor nivel de detalle del sistema.

3. Implementación: Contiene la descripción de la solución adoptada para este proyecto, junto con ejemplos de código fuente y detalles de implementación relevantes.

4. Pruebas: Recoge las pruebas realizadas para verificar la integración, la validez y eficiencia del sistema.

5. Apéndices: Se amplían los aspectos técnicos relevantes que se usaron durante el desarrollo del proyecto y no se han podido detallar en el *Manual Técnico*. Están recogidos al final de los tres volúmenes.

1.2.3. Manual de Usuario

Este manual, dirigido al usuario final del sistema, ofrece una descripción del funcionamiento y del uso de la aplicación desarrollada. Incluye toda la información necesaria para llevar a cabo la instalación y configuración finales, así como una serie de instrucciones para su correcta ejecución. Está compuesto por los siguientes apartados:

1. Requisitos *hardware* y *software*: Descripción de los requisitos mínimos necesarios para el correcto funcionamiento de la aplicación.

2. Instalación de la aplicación: Descripción de los pasos a seguir para lograr su puesta en marcha, y del proceso de instalación.

3. Uso de la aplicación: Descripción detallada del uso de la aplicación por parte de un usuario, así como del modo de uso de la aplicación.

1.3. Contenido del DVD-ROM

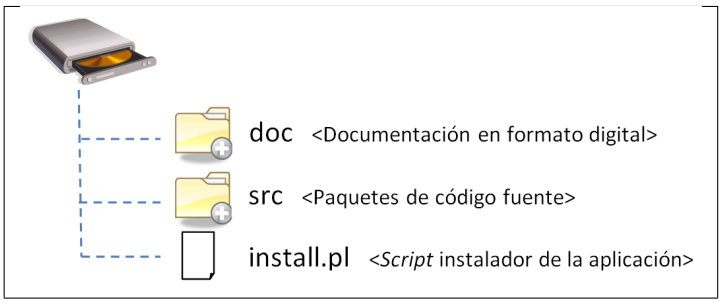


Figura 1.1: Contenido del DVD-ROM adjunto.

1.4. Visión general del sistema

Nuestro trabajo parte de la necesidad de disponer de una cadena de procesamiento lingüístico para el español. Actualmente, para lenguas como la inglesa, existe toda una variedad de recursos para aprovechar de forma relevante los beneficios y utilidad de los sistemas de *Procesamiento del*

Lenguaje Natural (PLN). Sin embargo, en el caso del castellano[1] se encuentra en un estado menos avanzado. Hasta el momento, los recursos existentes ofrecen un análisis precario e insuficiente para fundamentar el desarrollo de sistemas avanzados de PLN. La herramienta objeto de este proyecto, ha de proveer un análisis morfológico y sintáctico a un nivel profundo y de manera robusta.

Con el fin de cumplir este objetivo, fundamentamos nuestro trabajo en una herramienta ya existente: la cadena de procesamiento ALPAGE (Cabrera, 2008). Ésta ha sido implementada con el fin de obtener una cadena completa de procesamiento lingüístico para el francés e incluye toda una serie de herramientas: ALEXINA (*Atelier pour les LEXiques INformatiques et leur Acquisition*)[2], destinada al desarrollo de léxico; y *metagramáticas*[3], para la construcción de analizadores sintácticos robustos e híbridos GAR (Gramática de Adjunción de áRboles)/GIR (Gramática de Inserción de áRboles). Los principales pilares de la cadena francesa comentada son: *FRMG Lexer* (Cabrera, 2008), el analizador léxico, y *FRMG Parser* (De la Clegerie et al., 2009), el analizador sintáctico.

Concretamente, para la cadena objeto de este trabajo ha sido necesario implementar los analizadores léxico y sintáctico equivalentes para el español: *SPMG Lexer* y *SPMG Parser*, respectivamente. Estos recursos, perfectamente integrados con los componentes reutilizados de la cadena ALPAGE (Cabrera, 2008), son capaces de analizar morfosintácticamente textos en castellano.

El componente *SPMG Lexer* es el encargado de segmentar, lematizar y etiquetar con información morfosintáctica cada unidad léxica presente en el texto de entrada de la cadena. Este recurso, a su vez, incluye el *LÉxico de Formas Flexionadas del Español* (LEFFE) (Molinero et al., 2009), de amplia cobertura y basado en el formalismo lexical ALEXINA (Sagot y Danlos, 2008).

Para la implementación del componente *SPMG Parser* se ha utilizado *Metagrammar Toolkit*[4]. Se trata de un conjunto de herramientas, entre ellas MGCOMP (*Metagrammar Compiler*) (Thomasset y De la Clegerie, 2005) y DyALog (De la Clegerie, 2005a), creadas por y para la edición y compilación de metagramáticas en analizadores sintácticos tabulares, profundos e híbridos GAR/GIR (De la Clegerie, 2005b) de gran cobertura, capaces de obtener el análisis sintáctico de un texto de entrada a partir de un conjunto de sus unidades léxicas previamente etiquetadas. De este modo, para la construcción del analizador sintáctico ha sido necesario desarrollar una metagramática para el español. Ésta recoge un amplio

[1] A partir de ahora se asumirá que el idioma español es equivalente al castellano, puesto que este último es el idioma oficial del estado español.

[2] ALEXINA está recogida en el entorno de trabajo *ALEXINA-tools*.

[3] *Metagrammar Toolkit* es el entorno de compilación de las metagramáticas.

[4] `http://mgkit.gforge.inria.fr`

conjunto de estructuras sintácticas analizables y recibe el nombre de *SPanish MetaGrammar* (SPMG).

Con el objetivo de conseguir una cadena de procesamiento lingüístico con la misma funcionalidad que su equivalente francesa, se han adaptado los componentes de la cadena base que se han creído oportunos. Se puede resaltar la posibilidad de visualizar el análisis realizado por la cadena en distintos formatos: mediante un *grafo de dependencias* que muestra la estructura sintáctica de la frase de entrada de una forma más comprensible para el usuario humano del sistema, o bien en un formato estandarizado XML (eXtensible Markup Language)[5], que puede ser empleado por aplicaciones a un nivel superior.

A mayores, el sistema desarrollado permite una arquitectura cliente-servidor. Ello facilita que la cadena robusta de PLN ubicada en el servidor, dé servicio a distintos clientes que soliciten el procesamiento de textos.

Por lo tanto, nuestro trabajo ha consistido en la construcción de un recurso *software* que obtiene un análisis de la estructura morfosintáctica de un texto en español. Los resultados del análisis podrán ser interpretados tanto por un usuario humano como por una aplicación externa que accede al servidor de analizadores en busca de un procesamiento lingüístico.

[5]XML ha sido definido por el W3C. `http://www.w3.org/TR/REC-xml`

Capítulo 2

Descripción del sistema

En esta sección se especifican con más detalle cuestiones del proyecto, mostrando los objetivos perseguidos y exponiendo la estructura de la cadena de procesamiento lingüístico, así como la plataforma *hardware* y el *software* necesarios para su explotación.

2.1. Motivación

La ingente cantidad de información disponible en formato digital y en las distintas lenguas, hacen imprescindible disponer de sistemas que permitan acceder a bibliotecas cada vez de mayor talla, cuyos contenidos y organización son, además, muy heterogéneos. En este escenario, existe un interés renovado por la solución de los problemas de accesibilidad a la información y de mejora de explotación de la misma en entornos multilengua. Muchas de las bases formales para abordar adecuadamente estas necesidades han sido y siguen siendo establecidas en el marco del PLN.

El PLN es una de las ramas más importantes de la Inteligencia Artificial y, a su vez, subdisciplina de la Lingüística Computacional. El objetivo último es producir modelos y sistemas informáticos que posibiliten la comunicación hombre-computadora por medio del lenguaje natural, ya sea a través de la voz o del texto. Dado que el lenguaje humano está relacionado con la gestión inteligente de la comunicación, los modelos aplicados se enfocan no sólo a su comprensión, sino a aspectos cognitivos y a la organización de la memoria. Se trata de una disciplina tan antigua como el uso de las computadoras (años 50) (Carbonell, 1992), de gran complejidad, y que se puede aplicar a campos tan diversos e importantes como:

- Traducción automática.
- Correctores ortográficos y gramaticales.
- Recuperación de información.

- Interfaces hombre/máquina.
- Extracción de información.
- Resolución cooperativa de problemas.
- Entornos de diálogo.
- Búsqueda de respuestas.

Para remarcar la utilidad y aplicación del PLN, simplemente es necesario reflejar la frecuente problemática asociada a la imprecisión de las búsquedas en Internet. Consideremos, por ejemplo, el caso de la introducción en *Google*[1] de las consultas *"la fraga del presidente"* y *"el presidente Fraga"*, los resultados facilitados serán los mismos para ambas entradas, cuando su significado es totalmente diferente. Esto se debe a que el buscador, o sistema de interrogación e indexación, no es capaz de adquirir y diferenciar el significado (la semántica) de la consulta en cuestión. Bien al contrario simplemente lleva a cabo una comprobación de similitud entre los términos de la frase de entrada y los términos relevantes de las páginas *webs* almacenas en los distintos servidores de *Google*. Evitar este tipo de comportamiento requiere de un análisis de la entrada de cada usuario que facilite un conocimiento profundo sobre la sintaxis, la semántica, la ortografía y la gramática del idioma en cuestión. Por lo tanto, queda patente que el PLN es una herramienta necesaria en el desarrollo de los motores de búsqueda sobre Internet.

Para conseguir una relación *hombre-máquina* análoga a la relación *hombre-hombre*, esto es, que el usuario humano pueda interactuar con el ordenador de la misma forma que lo haría con un semejante, es imprescindible la comprensión y reconocimiento del lenguaje natural. Su estudio se organiza tradicionalmente en cinco niveles de complejidad que se describen a continuación:

- *Nivel fonético:* Estudia la producción y percepción de los sonidos de una lengua en sus manifestaciones físicas.
- *Nivel morfológico:* Trata la composición de cada palabra a partir de unidades de significado más pequeñas, denominadas morfemas. Dentro de ellas incluimos prefijos, sufijos, raíces y flexiones.
- *Nivel sintáctico:* Fija el papel que cada palabra juega dentro de una oración en base a una estructura gramatical.
- *Nivel semántico:* Estudia las relaciones funcionales entre las estructuras sintácticas y trata de dotarlas de significado.

[1] `https://www.google.es`

- *Nivel pragmático y del discurso:* Estudia como interpretar las estructuras semánticas en relación al contexto.

Este proyecto se centra en los niveles morfológico y sintáctico del lenguaje.

Cabe resaltar, en particular, el problema de la ambigüedad. Tanto las lenguas humanas como las formales tienen una estructura. Las primeras son el resultado de la evolución y por ello reciben el nombre de lenguas naturales. En cambio, todo lenguaje de programación ha sido creado en un momento dado con un propósito concreto y le ha sido impuesta una estructura fija, razón por la que se denominan lenguajes artificiales. Estas diferencias de origen han permitido que los lenguajes de programación rehuyan las construcciones ambiguas.

Ejemplo 1 *Como ejemplo ilustrativo tomaremos un ejemplo conocido, la frase:* "Juan vio un hombre con un telescopio en una colina". *Diferentes ubicaciones de las subestructuras correspondientes a los fragmentos* "con un telescopio" *y* "en una colina" *llevan a distintas estructuras sintagmáticas completas para la frase, todas ellas correctas, que se corresponden con los siguientes significados*[2]*:*

- *Juan vio un hombre que estaba en una colina y que tenía un telescopio. (figura 2.1)*
- *Juan estaba en una colina, desde donde miraba con un telescopio, a través del cual vio un hombre. (figura 2.2)*
- *Juan estaba en una colina, desde donde vio un hombre que tenía un telescopio. (figura 2.3)*
- *Juan miraba por un telescopio, a través del cual vio un hombre que estaba en una colina. (figura 2.4)*

■

El ejemplo comentado se refiere a la ambigüedad sintáctica, pero existen otros tipos de ambigüedades que inciden en ésta, como es el caso de la léxica. Esto es, que a una misma palabra se le puedan asignar distintas categorías léxicas. Ello puede generar diferentes interpretaciones de la oración a nivel estructural, lo que a su vez puede implicar una considerable sobrecarga para el analizador tanto a nivel léxico como, posteriormente sintáctico y semántico. Éste ha de encargarse, en particular, de decidir cuales de las categorías inicialmente asignables a una palabra son posibles dentro de las

[2]Donde en las estructuras arbóreas: `O` = Oración, `S` = Sujeto, `V` = Verbo, `OD` = Objeto Directo, `CCM` = Complemento Circunstancial de Modo, `CCL` = Complemento Circunstancial de Lugar, `SN` = Sintagma Nominal, `Det` = Determinante, `N` = Núcleo y `SPrep` = Sintagma Preposicional.

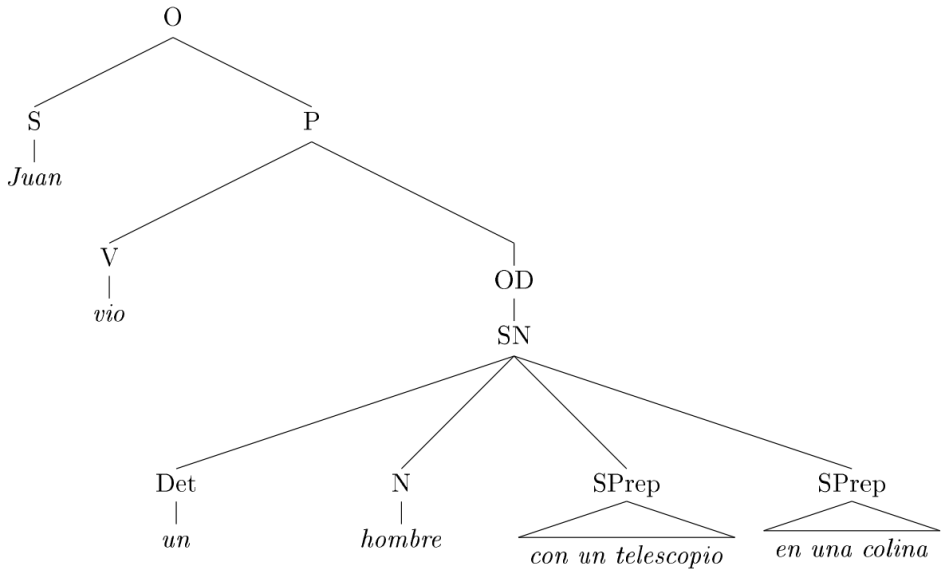


Figura 2.1: Posible estructura sintáctica de la oración *"Juan vio un hombre con un telescopio en una colina"*.

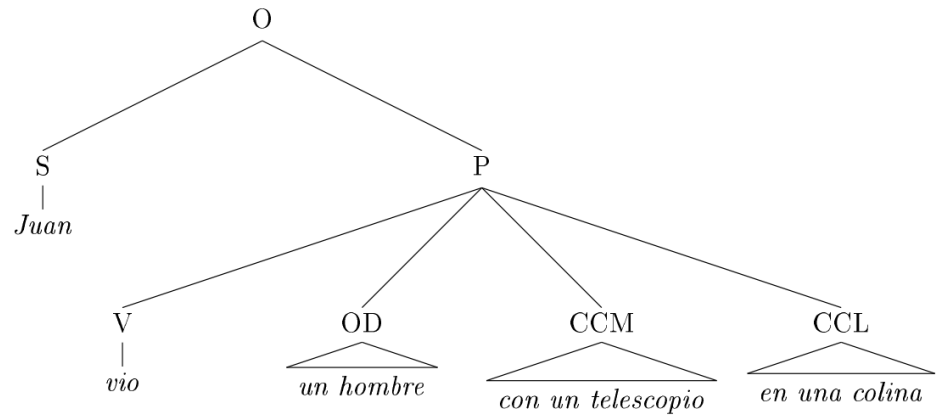


Figura 2.2: Posible estructura sintáctica de la oración *"Juan vio un hombre con un telescopio en una colina"*.

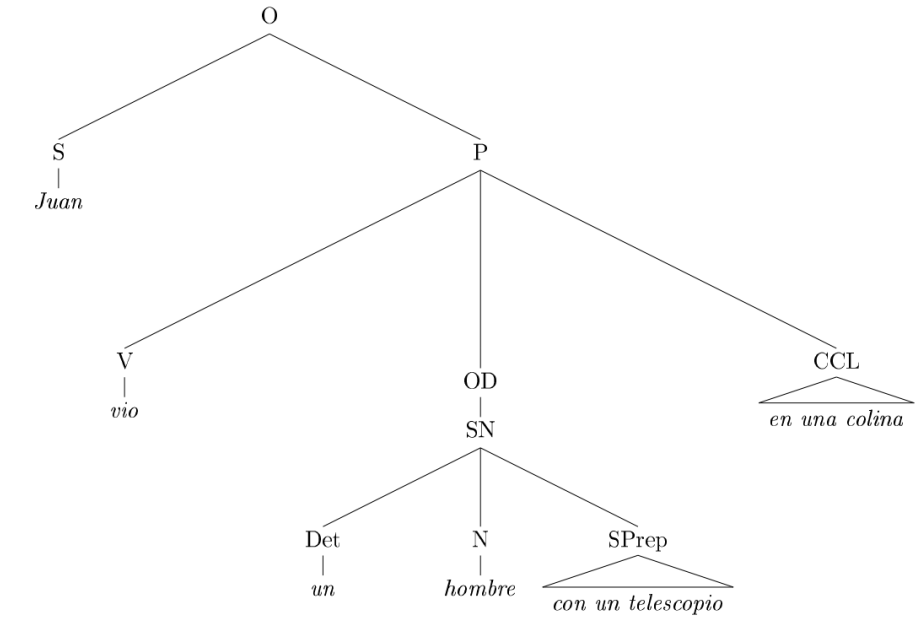


Figura 2.3: Posible estructura sintáctica de la oración *"Juan vio un hombre con un telescopio en una colina"*.

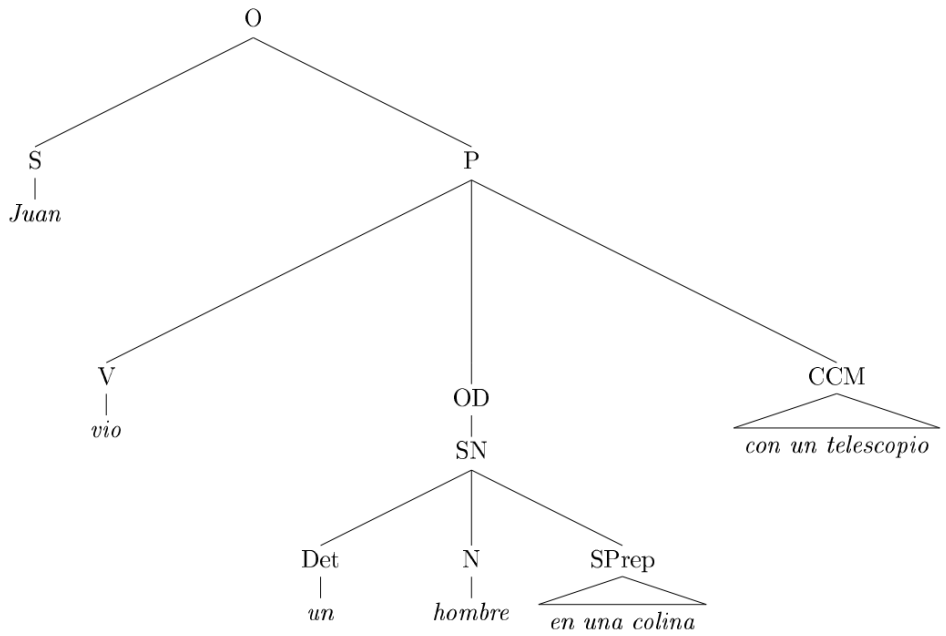


Figura 2.4: Posible estructura sintáctica de la oración *"Juan vio un hombre con un telescopio en una colina"*.

diferentes estructuras sintácticas que recoge la gramática de esa lengua para el texto analizado. Observar que dicha ambigüedad no se produce en los lenguajes de programación, donde cada palabra del lenguaje tiene un único significado.

Esta característica hace que los analizadores sintácticos diseñados para tratar el lenguaje natural sean infinitamente más complejos que sus análogos de programación. Del mismo modo que parte de la ambigüedad léxica es resuelta a nivel sintáctico, la desambiguación sintáctica queda relegada al siguiente nivel de análisis. Esto es, al nivel semántico.

A día de hoy, existen herramientas de PLN de aceptable calidad y cobertura para lenguas como la inglesa. La situación del español es, sin embargo, una de las más desalentadoras. No existe, de hecho, ninguna herramienta que ofrezca un análisis sintáctico aceptable que permita derivar al análisis semántico con un mínimo de garantías. Esto acarrea que la implementación de recursos lingüísticos sea un proceso largo, a menudo manual, que no suele alcanzar resultados visibles a corto plazo. Para intentar subsanar este problema surge el presente proyecto. Se pretende, entre otros objetivos, reunir los recursos necesarios para construir una cadena de procesamiento del español.

En esta línea, obtener análisis sintácticos profundos y robustos de manera automática es indispensable de cara a desarrollar aplicaciones que puedan hacer uso de representaciones semánticas de cualquier nivel para progresar en el análisis del castellano, en particular, en el desarrollo de aplicaciones de recuperación de información. A su vez, y a tenor de lo ya comentado, para completar un análisis sintáctico exitoso es imprescindible un eficiente etiquetado léxico del texto de entrada.

2.2. Objetivos

El objetivo principal de este proyecto es el diseño e implementación de una cadena de análisis lingüístico que permita obtener un procesamiento morfosintáctico de un texto de entrada en castellano. De esta forma, el sistema proveerá de un procesamiento lingüístico a una aplicación PLN a un nivel semántico o permitirá la visualización de los resultados obtenidos a un usuario humano.

Para ello se toma como base la ya mencionada cadena ALPAGE (Cabrera, 2008). En este sentido, se realizarán las modificaciones oportunas sobre los componentes originales, con el fin de preservar su arquitectura y funcionalidad base. A continuación, se integrarán en la nueva cadena junto con los recursos desarrollados para su adaptación al análisis de la lengua castellana. De una forma más concreta, podemos estructurar el proyecto en torno a los siguientes objetivos:

1. **Construir un analizador léxico para el castellano y agregar la información léxica pertinente:** Así, se ha retomado el desarrollo del léxico español LEFFE (LÉxico de Formas Flexionadas del Español) (Molinero et al., 2009). Éste se encontraba en un nivel preliminar de su construcción cuando se inició este proyecto. Sin embargo, a medida que se desarrolle y evolucione el analizador sintáctico, es necesario introducir cambios y mejoras en el léxico de partida. En este sentido, tanto el léxico como el analizador sintáctico evolucionarán al mismo nivel. *SPMG Lexer* será el componente de la cadena responsable de reunir los distintos recursos encargados del análisis léxico que utilizará la información morfosintáctica proporcionada por el léxico LEFFE. Su homónimo francés es *FRMG Lexer*, basado en el léxico francés LEFFF (*LExique des Formes Fléchies du Français*) (Sagot et al., 2006).

2. **Desarrollar un analizador sintáctico para la lengua española:** Proveer a la cadena de procesamiento lingüístico en construcción de un analizador sintáctico capaz de determinar la gramaticalidad del texto. Esto es, estudiar la forma en que se combinan las palabras para formar sintagmas y oraciones correctas, determinando el papel estructural de cada palabra y sintagma. El analizador encargado de reconocer la estructura sintáctica de un texto de entrada será conocido como *SPMG Parser*. El analizador sintáctico será el resultado de compilar una metagramática a desarrollar, identificada como SPMG (*SPanish Metagrammar*). Ésta describirá las diferentes estructuras sintácticas permitidas en español. La idea es crear un componente equivalente al analizador sintáctico francés *FRMG Parser* (De la Clegerie et al., 2009).

3. **Integración en la cadena en construcción:** Acoplar perfectamente los recursos tomados de la cadena ALPAGE original (Cabrera, 2008), con el fin de aprovechar todo su potencial. Ello conllevará la adaptación de los componentes reutilizados al nuevo idioma a analizar. Las funcionalidades que conservaremos de ALPAGE son:

   - Arquitectura cliente-servidor: (Cabrera, 2008) Ofrece una estructura óptima para dar servicios de procesado lingüístico a distintos clientes.
   - Preprocesador *SxPipe*: (Sagot y Boullier, 2008) Es capaz de llevar a cabo labores previas de segmentación del texto y corrección ortográfica, entre otras funcionalidades, que el analizador léxico luego complementará.
   - Visualización de los resultados: (Cabrera, 2008) La cadena ALPAGE recoge una gran variedad de recursos a la hora de manejar diferentes formatos de representación de los resultados

obtenidos como *grafos de dependencias* o XML. Además, también facilita datos estadísticos compilados durante el proceso acerca del tiempo medio empleado en el análisis de cada oración, del porcentaje de cobertura del análisis sobre un determinado *corpus* o de la tasa de ambigüedad media, entre otros.

4. **Determinar la cobertura del sistema desarrollado:** Comprobar la cobertura que ofrece la cadena de procesamiento lingüístico construida para el español. Para ello, se realizarán diferentes pruebas sobre *corpora* ya existentes. Asimismo, se llevará a cabo una comparativa de los resultados obtenidos por la nueva cadena respecto a ALPAGE. En este sentido, se emplearán los datos estadísticos proporcionados por la cadena desarrollada para obtener parámetros de referencia en común entre ambas propuestas.

Para conseguir estos objetivos ha sido necesario:

- La iniciación en el ámbito del PLN.
- El estudio detallado del formalismo lexical ALEXINA (Sagot y Danlos, 2008). Este se encuentra recogido en el entorno de trabajo *ALEXINA-Tools* destinado al desarrollo del léxico morfosintáctico LEFFE (Molinero et al., 2009).
- Estudio del lenguaje de programación *Perl*, utilizado para implementar el nuevo analizador léxico *SPMG Lexer*. Además, también se ha empleado *Perl* (Wall et al., 2000) para la adaptación de los recursos reutilizados, puesto que la mayor parte de los componentes de la cadena ALPAGE (Cabrera, 2008) se encuentran implementados en este lenguaje.
- Estudio pormenorizado del entorno de trabajo *Metagrammar Toolkit*[3], necesario para el desarrollo de la metagramática del castellano (SPMG) y para su posterior compilación en el analizador sintáctico *SPMG Parser*. Destacar la complejidad de trabajar con el lenguaje de programación *orientado a objetos*, diseñado única y exclusivamente para la construcción de metagramáticas, así como la compresión de las características y configuración de los compiladores MGCOMP (Thomasset y De la Clegerie, 2005) y DyALog (De la Clegerie, 2005a), imprescindibles para la obtención de analizadores sintácticos híbridos GAR/GIR (De la Clegerie, 2005b) a partir de una metagramática.
- Estudio del funcionamiento e implementación de la cadena ALPAGE, con el fin de integrar perfectamente los nuevos componentes desarrollados en una cadena de procesamiento lingüístico.

[3] `http://mgkit.gforge.inria.fr`

- Estudio de las *Gramáticas de Adjunción de Árboles* (Alonso, 2000), base del analizador sintáctico de la cadena objeto de este trabajo.
- Adquisición de conocimientos suficientes de la gramática española, necesarios para desarrollar una metagramática aceptable del castellano.
- Estudio del lenguaje para el procesamiento de textos LaTeX (Lamport, 1994), para la edición de la documentación de este proyecto.
- Estudio y repaso de la notación UML (*Unified Modeling Language*) (Booch et al., 2007; Larman, 2002), destinada al análisis y diseño del sistema.

2.3. Arquitectura del sistema

A continuación se muestra la estructura de la cadena de procesamiento lingüístico desarrollada, incidiendo en los componentes de la misma que han sido implementados a lo largo de este trabajo.

La herramienta a construir intenta mantener la estructura de la cadena ALPAGE (Cabrera, 2008). Se sustituyen aquellos componentes específicos del idioma francés por sus equivalentes españoles, y se integran los componentes reutilizados. En la figura 2.5 podemos distinguir los principales recursos que componen la arquitectura de la cadena de procesamiento lingüístico del español.

2.3.1. Análisis léxico: LEFFE, SxPipe y SPMG Lexer

A continuación se describen los componentes encargados del nivel morfológico del análisis dentro de la cadena de procesamiento lingüístico desarrollada.

2.3.1.1. LEFFE

Es un léxico morfológico y sintáctico de amplia cobertura y libre (bajo licencia LGPL-LR)[4]. Puede ser usado directamente en aplicaciones de PLN de alto nivel, incluso en aquellas que requieren un análisis sintáctico profundo. El LEFFE está desarrollado mediante ALEXINA (Sagot et al., 2006), un formalismo lexical previamente usado en el desarrollo de LEFFF (Sagot y Danlos, 2008).

Podemos definir un *léxico* como una lista exhaustiva de palabras que componen una lengua, acompañadas de cierta información morfológica y/o sintáctica. En efecto, para desarrollar tareas de PLN de alto nivel es necesario disponer de léxicos que efectivamente describan el comportamiento sintáctico

[4] *Lesser General Public License for Linguistic Resources.*

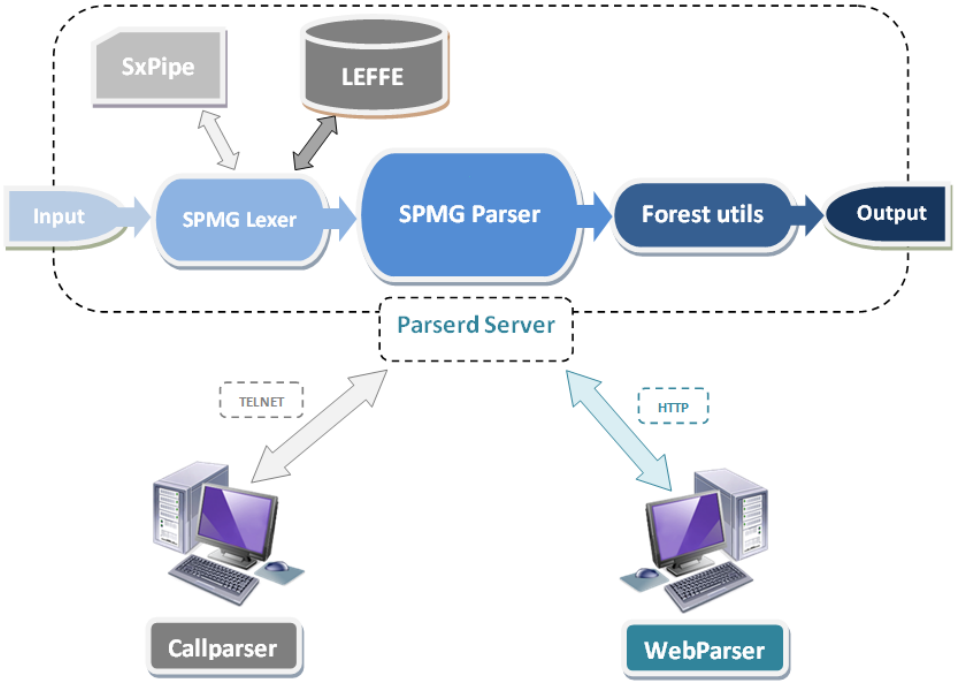


Figura 2.5: Arquitectura de la Cadena de Procesamiento Lingüístico.

de sus entradas[5]. ALEXINA (Sagot y Danlos, 2008) es un modelo que permite describir información morfológica y sintáctica de manera fácilmente legible, completa y eficiente. Su flexibilidad y calidad permiten representar un gran número de fenómenos a través de un formato sencillo que puede ser usado directamente por varios formalismos gramaticales (GLF[6] o GARL[7], entre otros.) que requieren información sintáctica detallada para todas las palabras (Sagot et al., 2006; Sagot y Danlos, 2008). El formato ha evolucionado durante los últimos cinco años junto al LEFFF y otros recursos para otras lenguas (polaco, eslovaco y otros). Esto, unido a la proximidad lingüística entre el francés y el español[8], ha permitido describir este último sin tener que modificar el formato original. La herramienta está basada en dos niveles de representación:

- Un **nivel intensional** que factoriza la información léxica, de modo

[5]Por ejemplo, para la palabra *"comer"*, además de indicar su categoría léxica (verbo) puede incluir información sintáctica como, por ejemplo, que este verbo puede ir acompañado de diferentes funciones gramaticales tal que sujeto o objeto directo, entre otras; así como qué categorías léxicas o sintagmas pueden realizar dichas funciones.

[6]Gramática Léxico-Funcional.

[7]Gramática de Adjunción de áRboles Lexicalizada.

[8]Tanto el francés como el español son lenguas flexionadas pertenecientes a la familia de lenguas Romances.

que a cada lema[9] se le asocia una clase morfológica[10] e informaciones sintácticas detalladas[11], permitiendo una gestión rápida y sencilla. La información léxica en forma intensional se encuentra organizada en diferentes ficheros `.ilex` en función de la categoría gramatical de los lemas[12].

- Un **nivel extensional**, que se genera automáticamente compilando el léxico intesional (todos los ficheros `.ilex`), en el que se asocia cada forma[13] flexionada con toda su información morfológica y sintáctica: etiqueta morfológica o el marco de subcategorización de su correspondiente redistribución, entre otros. Tras este proceso se obtienen los ficheros en forma extensional `.lex`, correspondientes a cada uno de los ficheros `.ilex` compilados[14].

Cuando el léxico intensional es compilado en uno extensional, se construyen todas las palabras pertenecientes a la familia de cada lema, usando para ello su clase morfológica[15]. Las clases morfológicas están definidas bajo un formato que cubre la mayor parte de las entradas del léxico. Tan solo los lemas que se flexionan de una forma especial (irregular) son descritos de forma manual en un fichero con extensión `.mf`.

Cada entrada en el léxico intensional se define habitualmente por un lema y una categoría léxica. Sin embargo, es posible encontrar varias entradas con el mismo lema y categoría léxica, pero en este caso difieren en la información morfológica y sintáctica. Esto permite dividir un lema en diferentes significados semánticos, los cuales implican diferentes construcciones sintácticas. Una entrada intensional recoge la siguiente información:

- Una *clase morfológica*, la cual define los patrones que construyen todas las formas flexionadas del lema.

[9] Lema es la forma canónica de la palabra, usualmente el masculino singular. Representa un conjunto de palabras con la misma raíz, la misma categoría léxica y el mismo sentido. Por categoría léxica, se entiende el tipo de palabras (por ejemplo nombre o adjetivo) mientras que sentido es el significado. Por ejemplo, para la forma *"gatitos"*, el lema es *"gato"* y el sentido *animal*.

[10] Clase que permite construir toda la familia de formas asociada a dicho lema.

[11] Marco de subcategorización, posibles reestructuraciones o atributos, entre otros.

[12] Por ejemplo, para el adjetivo es `A.ilex`, para el verbo es `V.ilex`, y así para todas las categorías léxicas recogidas. Se muestran todos los ficheros `.ilex` en el apartado 8.1 del *Manual Técnico*.

[13] Por forma, se entiende cada palabra resultante de aplicar reglas de derivación sobre la raíz de un determinado lema. Dicho de otro modo, la forma es la palabra tal como aparece. En la forma flexionada *"caserones"* el lema es *"casa"* y la raíz etimológica *"cas"*.

[14] Más adelante se muestra un ejemplo de ambos formatos.

[15] Las clases morfológicas están definidas bajo un formato descrito en Sagot, Benoît 2005. *Automatic acquisition of a Slovak lexicon from a raw corpus.* In *Lecture Notes in Artificial Intelligence 3658*, *Proceedings of TSD'05*, páginas 156-163, Karlovy Vary, Czedh Republic.

- Una *categoría léxica*, que es seleccionada de un conjunto de etiquetas posibles. Para LEFFE, las categorías léxicas pueden dividirse en dos tipos: *abiertas*[16] (también denominadas productivas) y *cerradas*[17] (también denominadas gramaticales).

- Un *marco de subcategorización*, que muestra explícitamente como el lema puede ser utilizado en una determinada construcción sintáctica. Éste lista las funciones sintácticas[18] de los posibles argumentos del lema, y la posible realización de cada una de esas funciones[19].

- Posibles *redistribuciones*, que definen como los marcos de subcategorización de sintaxis profunda[20] se transforman para construir marcos de subcategorización de sintaxis superficial[21].

Ejemplo 2 *Por ejemplo, ésta es la entrada intensional simplificada*[22] *en LEFFE para el lema* "destacar" *en el sentido de resaltar algo:*

```
destacar
        V4
        Lemma;v;
        <arg0 :Suj:cln|scompl|sinf|sn,
         arg1 :Obj:cla|scompl|sn>;
         %actif, %passif, %ppp_employé_comme_adj
```

Se trata de un verbo transitivo y recoge la siguiente información:

- *Su clase morfológica es* `V4`*, lo que se corresponde con verbos de la primera conjugación que cambian su raíz (se cambia la "c" por "qu") al formar el presente de subjuntivo.*

[16] Adjetivos, adverbios, verbos o nombres, entre otros. Mediante flexión, derivación, inclusión de neologismos, se pueden añadir nuevas categorías.

[17] Preposiciones, pronombres o conjunciones, entre otros. Las cerradas no permiten añadir más categorías.

[18] Las posibles funciones sintácticas usadas en LEFFE son las siguientes: `Suj` (sujeto), `Obj` (objeto directo), `Objde` (Objeto indirecto introducido por la preposición *"de"*), `Obja` (objeto indirecto introducido por *"a"*), `Loc` (locativo), `Dloc` (delocativo), `Att` (atributo), `Obl` and `Obl2` (oblicuos).

[19] Las posibles realizaciones son: *clíticas*, `cln`, `cla` y `cld` para los casos nominativo, acusativo y dativo; *directas*, son *sn*, *sinf*, *scompl*, *sa* y *qcompl* para los sintagmas nominal, infinitivo, completivo, adjetival y preguntas indirectas; y *preposicionales*, se construyen de la forma *prep-real*, donde *prep* es una preposición y *real* una realización directa (Ej. *con-sn*).

[20] Una sintaxis profunda ofrece el máximo nivel de detalle de la estructura sintáctica (va más allá del nivel de sintagma), mientras que una sintaxis superficial no alcanza el nivel de precisión anterior, sino que únicamente refleja las funciones sintácticas más generales (sujeto, objeto directo) sin desglosar los sintagmas que las realizan.

[21] Las redistribuciones habituales son `%actif` (para la voz activa), `%passif` (para la voz pasiva) y `%ppp_employé_comme_adj` (cuando el participio funciona como adjetivo)

[22] Se han eliminado algunas informaciones sintácticas por motivos de claridad.

- *Su predicado semántico se representa directamente con el lema (*`Lemma`*).*
- *Su categoría léxica es verbo (*`v`*).*
- *Tiene dos argumentos canónicamente realizados por las funciones sintácticas* `Suj` *(sujeto) y* `Obj` *(objeto directo). Cada función sintáctica está asociada a una lista de realizaciones alternativas.*
- *Esta entrada permite, además, tres redistribuciones:*
  - *activa (* `%actif`*).*
  - *pasiva (* `%passif`*).*
  - *participio empleado como adjetivo (* `%ppp_employé_comme_adj`*).*

El proceso de compilación construye una entrada extensional para cada una de las formas flexionadas del lema y para cada redistribución compatible, aplicando las definiciones formales de esas redistribuciones. Por ejemplo, la única forma flexionada del verbo "destacar" *que es compatible con la redistribución pasiva (* `%passif`*) es el participio. Por otra parte, la redistribución* `%ppp_employé_comme_adj` *indica que el participio de este verbo puede ser usado como adjetivo y provocará la generación de la correspondiente entrada extensional.*

La entrada extensional (simplificada) correspondiente a la redistribución pasiva del lema destacar *es la siguiente*[23]*:*

```
destacado   v
     [pred='destacar <arg1:Suj:cln|scompl|sn,
      arg$0:Obl2:(por-sn)>',@passive,@pers,@MPOOSM$];
      %passif
```

Como se puede ver, la redistribución pasiva (`%passif`*), obliga a que el objeto directo original (*`Obj`*) en posición canónica sea transformado en el sujeto pasivo; y el sujeto activo (*`Suj`*), en un complemento agente opcional (*`Obl2`*), realizado por un sintagma nominal precedido por la preposición* por *(*`por-sn`*).*

■

Una vez LEFFE se encuentre en forma extensional, es necesario someterlo a una segunda compilación para poder utilizarlo bajo analizadores sintácticos basados en metagramáticas. Mientras que la primera compilación era realizada únicamente por *ALEXINA-tools* (Sagot y Danlos, 2008), para esta última ha sido necesario implementar el paquete LEFFE-SPMG, de

[23] Donde `MPOOSM` es la etiqueta para el participio singular masculino.

estructura similar al paquete LEFFF-FRMG, y hacer uso del lexicalizador *Lexed*[24].

La funcionalidad del recurso LEFFE-SPMG es, simplemente, la de reunir toda la información morfosintáctica presente en los diferentes ficheros del LEFFE extensional bajo un único fichero. Será compilado por *Lexed* en un autómata de estados finitos recogido en el fichero `dico.xlfg.fsa`. De este modo, el conjunto de ficheros extensionales se transforman en un lexicón del castellano. Esto es, en un diccionario de fácil y rápida consulta, proveedor de información morfosintáctica. El lexicón contiene en cada línea una palabra con la información asociada a la misma. Ésta será la base de información de *SPMG Lexer*. Para tener acceso a ella, tendrá que echar mano del lexicalizador[25] *Lexed*.

Durante el proceso de lematizado[26] y etiquetado[27], a cada unidad léxica[28] presente en el texto de entrada se le asigna una etiqueta que recoge su correspondiente información morfosintáctica[29]. Ésta es la entrada correspondiente a la palabra o signo de puntuación en el LEFFE compilado, presentada bajo una estructura perfectamente compatible con el formato de etiquetas manejadas por el analizador sintáctico.

Por tanto, es necesario que *SPMG Lexer* lleve a cabo una adaptación de la información morfosintáctica del lexicón a la terminología y formato empleados en el analizador *SPMG Parser*. Su función es la de transformar en tiempo real aquella información del LEFFE que utiliza *SPMG Parser*, de tal forma que este último pueda comprenderla para llevar a cabo su tarea.

Por ejemplo, la categoría léxica *adjetivo* se designa como `a` en el lexicón LEFFE, mientras que el analizador sintáctico denota los adjetivos mediante `adj`. Lo mismo ocurre con los pronombres (`p` y `pro`) o adverbios (`r` y `adv`), por ejemplo. Además, transforma la información sintáctica del lexicón (*marco de subcategorización*) en un formato que pueda manejar el analizador. Asimismo, *SPMG Lexer* permite reducir la información que el lexicón proporciona a *SPMG Parser* durante el análisis, puesto que el analizador sintáctico puede no ser capaz de manejar o necesitar dicha información[30].

[24] *Lexed* es un recurso *software* incluido en el *framework ALEXINA-tools* y distribuido con licencia GPL. Este ha sido diseñado para construir y consultar bases de información léxica. Permite buscar de forma rápida una entrada en un diccionario en base a cadenas de caracteres. El algoritmo que emplea está basado en autómatas finitos y ofrece una buena alternativa a las tablas *hash* para implementar grandes diccionarios. `http://www.labri.fr/perso/clement/lexed/`

[25] Lexicalizador es aquella herramienta *software* capaz de organizar y manejar grandes cantidades de palabras de forma eficiente.

[26] Proceso de reducción de una forma (palabra) a su lema. Por ejemplo: si lematizamos la palabra *"cervatillos"*, el resultado será el lema de la misma, *"ciervo"*.

[27] Proceso de obtención y asignación a una palabra concreta de su categoría léxica, tipos y rasgos gramaticales de la misma.

[28] Tanto las palabras como los signos de puntuación son unidades léxicas.

[29] Nótese que incluso un signo de puntuación tiene asignada información morfosintáctica.

[30] LEFFE dispone de información morfosintáctica de un nivel más profundo del que

Ejemplo 3 *Por ejemplo, y continuando con el caso anterior, para la forma* "destacado"*, SPMG Lexer utilizaría a Lexed para que le proporcionase la información referente a esa palabra en el LEFFE compilado. Una vez obtenida la información necesaria, ésta es transformada en el siguiente formato:*

```
lemma  { lex     => destacado,
         truelex => destacado,
         lemma   => destacar,
         cat     => v,
         top     => v {aux_req => 'être',
                       diathesis => passive,
                       gender => masc,
                       mode => participle},
         anchor  => tag_anchor { name =>
                    ht {arg0 => arg {kind => [obj,(-)],
                                     pcas => prep[por,
                                     (-)]},
                        arg1 => arg {kind => subj,
                                     pcas => (-)},
                        arg2 => arg {kind => (-),
                                     pcas => (-)},
                    diathesis => passive,
                    imp => '-',
                    refl => (-)}
}
```

■

Esta estructura constituye la etiqueta que el analizador léxico le asignaría a la palabra *"destacado"*, en el supuesto de que apareciese en el texto de entrada. Se trata de un formalismo empleado por el analizador sintáctico para representar la etiqueta de cada forma, y se conoce con el nombre de *hypertag*. Como se puede apreciar, se está describiendo la misma información que en la forma extensional, modificando ligeramente el formato. La estructura de una *hypertag* es la siguiente:

- `lex`: La palabra (forma) a cual se asigna el *hypertag* tal y como aparece en el LEFFE.
- `truelex`: La palabra tal cual aparece en el texto de entrada. Ésta puede no coincidir con su forma correspondiente en LEFFE (recogida en el apartado anterior) por causas como: una contracción oral o escrita de la misma, errores ortográficos, entre otras.

SPMG Parser es capaz de utilizar actualmente. Conseguir que este último soporte esta información, mejorará en el futuro el rendimiento del analizador sintáctico.

- `lemma`: El lema de la forma en cuestión.
- `cat`: Almacena la categoría léxica de la palabra. En este caso, la categoría léxica `v` se expresa de la misma forma, tanto en LEFFE compilado como en el formato del analizador sintáctico.
- `top`: Recoge información más detallada acerca de la forma. Esta puede ser: género, número, persona, modo, diátesis[31], auxiliar requerido[32], entre otros.
- `anchor`: En este apartado se detalla la información sintáctica presente en la entrada extensional referente a argumentos del marco de subcategorización[33], redistribución, si es o no impersonal (`imp`) y si es o no reflexiva (`refl`). Se trata de información específica que sirve como enlace entre el léxico y la sintaxis. Esto es, a partir de la información recogida en el apartado `anchor` de la *hypertag* asignada a cada palabra, se consigue enlazar esta última con las estructuras sintácticas del analizador *SPMG Parser* que presenten unas características similares en su ancla[34].

Para finalizar, todo el proceso de compilación del LEFFE aparece descrito en la figura 2.6.

2.3.1.2. SxPipe

Antes de llevar a cabo un análisis léxico o sintáctico sobre un texto es imprescindible disponer de un subsistema de preprocesado. Este es el encargado de convertir texto bruto en un formato comprensible y manejable para el posterior análisis lingüístico. En este caso concreto, se encarga de convertir texto bruto en un *Grafo Acíclico Dirigido* (GAD), que será utilizado

[31] La *voz*, llamada también *diátesis* (= disposición, manera de ser), es una categoría gramatical del verbo que indica si el sujeto del proceso verbal es ajeno o se ve implicado en éste. Son dos las voces o diátesis fundamentales: la *activa*, en la cual se expresa que el sujeto permanece fuera del proceso verbal, y la *pasiva*, la cual expresa que el sujeto sufre o recibe la acción ejecutada por otro.

[32] En las construcciones verbales compuestas, el verbo principal requiere de un verbo auxiliar (*haber* o *estar*) dependiendo de si se encuentra en gerundio o participio. Por ejemplo, el verbo *"comer"* en la estructura *"ha comido"* necesita el verbo *"haber"* como auxiliar, mientras que en la construcción *"está comiendo"*, hace uso del verbo *"estar"*.

[33] `kind` y `pcas` son atributos de cada argumento del marco de subcategorización. Mientras que el primero recoge las funciones sintácticas que puede ejercer un determinado argumento del verbo, el segundo muestra las posibles preposiciones que pueden preceder a ese argumento (en caso de ternerlas). Existen dos atributos más, no presentes en el ejemplo: `real` (posibles realizaciones de las funciones sintácticas) y `extracted` (indica si el argumento se encuentra en una posición no canónica).

[34] Un ancla es un símbolo terminal de un árbol elemental. En el apartado referente al análisis sintáctico se detallará el uso de las *hypertags* como enlaces entre el léxico y la sintaxis.

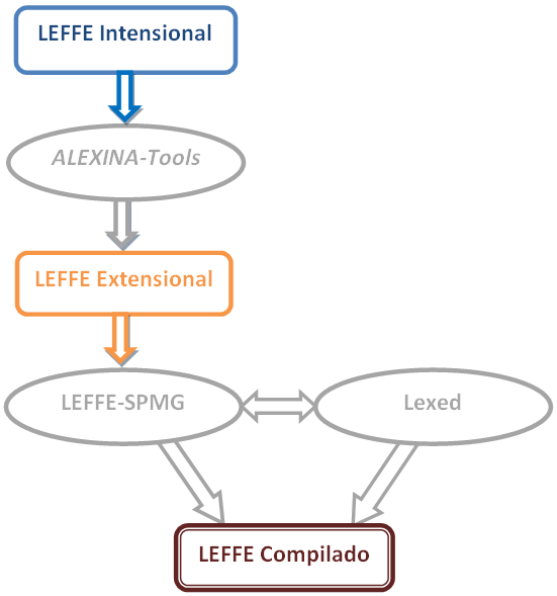


Figura 2.6: Proceso de compilación del LEFFE.

por *SPMG Lexer* para obtener las unidades léxicas que componen el texto de entrada. Se trata de un recurso que nos ofrece la cadena ALPAGE y, debido a su necesidad, se ha integrado en la cadena desarrollada. *SxPipe* (Sagot y Boullier, 2008, 2005) incluye de forma secuencial:

1. **Reconocimiento de *entidades con nombre*:** Detección y clasificación de los elementos del texto en categorías predefinidas, como nombres de personas, organizaciones, lugares, URLs, correos electrónicos, expresiones numéricas o de tiempo, entre otros, que aparecen mencionadas en un texto escrito en un determinado idioma. Esta actividad también se suele denominar como *etiquetado semántico*, puesto que el reconocimiento de las palabras se hace en base a su significado.

   La dificultad de la detección estriba en que dichas entidades pueden aparecer en diferentes formas. Por ejemplo, *"Antonio Banderas"* podría representarse como *"Banderas"*, *"A. Banderas"* o *"José Antonio Domínguez Banderas"*; así como, *"Banco Santander Central Hispano"* podría aparecer en el texto como *"Banco Santander"*, *"Santander"* o *"BSCH"*.

   Además, una vez detectadas, surge el problema de la ambigüedad léxica para su clasificación, ya sea entre diferentes categorías léxicas o incluso dentro de la misma. Por ejemplo, *"Sevilla"* puede ser la ciudad, el equipo de fútbol o el exministro *Jordi Sevilla*.

Este proceso se aplica antes de la segmentación del texto en *tokens*[35]; puesto que en caso de ser posterior, una *entidad con nombre* compuesta por varios *tokens*, podría ser segmentada erróneamente. Esto conllevaría un análisis por separado de los mismos, con un posterior análisis global erróneo desde el punto de vista semántico.

2. ***Tokenización*[36] y detección de límites entre frases:** Implica la segmentación del texto bruto en los *tokens* que lo componen, respetando las entidades con nombre reconocidas anteriormente. La complejidad de esta tarea radica en detectar y evitar convenientemente los signos de puntuación que no indican el final de una oración. Por ejemplo, las abreviaturas.

3. **Corrección ortográfica:** Los textos brutos suelen contener diversos errores ortográficos. Por lo tanto, los términos erróneos presentes en el texto no podrán ser reconocidos por el analizador léxico. Como consecuencia, se le asignara la etiqueta de *palabra desconocida*. Esto debe evitarse en la medida de lo posible, puesto que la presencia de una palabra desconocida obliga al analizador léxico a asignarle las categorías léxicas de sustantivo, verbo, adjetivo y adverbio, con el fin de poder incluirla en el posterior análisis sintáctico. Ello reducirá la precisión del analizador sintáctico e incrementará la ambigüedad sintáctica del análisis. El encargado de la corrección de los errores ortográficos es el recurso *SxSpell*, incorporado en *SxPipe*. Este componente se basa en un conjunto de reglas ortográficas y en su propio lexicón de formas correctas ortográficamente. De esta forma, una vez obtenidas todas las palabras presentes en el texto, se aplica *SxSpell* sobre cada una de ellas. Si una palabra es incorrecta ortográficamente, se corrige aplicando las reglas ortográficas pertinentes.

4. **Procesamiento no determinista de multi-palabras[37]:** Se encarga de crear la salida del subsistema de preprocesado. Ésta es el conjunto de palabras que componen el texto de entrada, representadas mediante

[35] Un *token* es un carácter o una cadena de caracteres delimitada por un espacio en blanco o otro tipo de símbolo. Cada palabra o signo de puntuación se considera un *token*. No se debe confundir este concepto con el de palabra. Una palabra puede estar formada por más de un *token*. Por ejemplo la locución *"sin embargo"*, está compuesta por dos *tokens*, *"sin"* y *"embargo"*.

[36] Proceso de segmentación del texto bruto en *tokens* o unidades léxicas mínimas con significado. Una *tokenización* simple transformará texto bruto en el conjunto de palabras y signos de puntuación que lo componen.

[37] Multi-palabra son aquellas unidades léxicas que se combinan de una manera determinada para expresar un cierto significado. La casuística de dichas combinaciones no es descrita mediante reglas gramaticales, ya que en muchos casos se producen de forma arbitraria. Por ejemplo, en español, se *dan gritos* mientras que en francés éstos se *empujan* (*pousser un cri*). El concepto de multi-palabra también incluye los aglutinados.

un GAD. También se encarga de separar los aglutinados[38] y de la descapitalización[39].

La mayor parte de sus componentes son independientes del idioma objeto del análisis. Sin embargo, existen algunos recursos del mismo que no ha sido posible adaptar de forma completa al castellano. El proceso de reconocimiento de entidades con nombre, únicamente es capaz de identificar un número bastante limitado de entidades en castellano. Por otro lado, no se puede utilizar el componente *SxSpell*, encargado de la corrección ortográfica, dado que no se encuentra adaptado al castellano. La arquitectura del sistema de preprocesado *SxPipe* se ilustra de forma más detallada en la figura 2.7.

Ejemplo 4 *Véase el siguiente ejemplo para comprender mejor el procesamiento que lleva a cabo* SxPipe. *Dada la oración de entrada* "El niño comió una manzana.", *se obtiene la representación en GAD de la figura 2.8. La representación interna del sistema para este GAD se muestra a continuación:*

```
GAD BEGIN
1   {<F id="E1F1">El</F>}     el    2
2   {<F id="E1F2">niño</F>}     niño    3
3   {<F id="E1F3">comió</F>}     comió    4
4   {<F id="E1F4">una</F>}     una     5
5   {<F id="E1F5">manzana</F>}     manzana     6
6   {<F id="E1F6">.</F>}     .     7
GAD END
```

■

2.3.1.3. SPMG Lexer

Recoge y gestiona diferentes recursos, entre ellos, LEFFE y *SxPipe*, de tal forma que en conjunto lleven a cabo el análisis léxico de un texto en castellano, esto es; identificar, lematizar y etiquetar cada palabra presente en el texto de entrada. Más concretamente, dada una oración a analizar, el recurso *SxPipe* se encarga de ofrecernos su GAD correspondiente. Como se ha visto, éste recoge las unidades léxicas presentes en el texto a analizar.

La posterior tarea de *SPMG Lexer* es la de lematizar y etiquetar cada una de las palabras presentes en el GAD. En este sentido, utiliza la información morfosintáctica recogida en el lexicón LEFFE mediante el recurso *Lexed*. Así, asigna a cada unidad léxica presente en el GAD una o varias *hypertags*[40]. Esta

[38] Estos son aquellas palabras que son el resultado de la unión de dos o más unidades léxicas. Por ejemplo, el aglutinado *"del"* compuesto por las palabras *"de"* más *"el"*.

[39] Por ejemplo, transforma *"Comieron"* en *"comieron"*.

[40] Es una etiqueta con información morfosintáctica de la forma que acompaña.

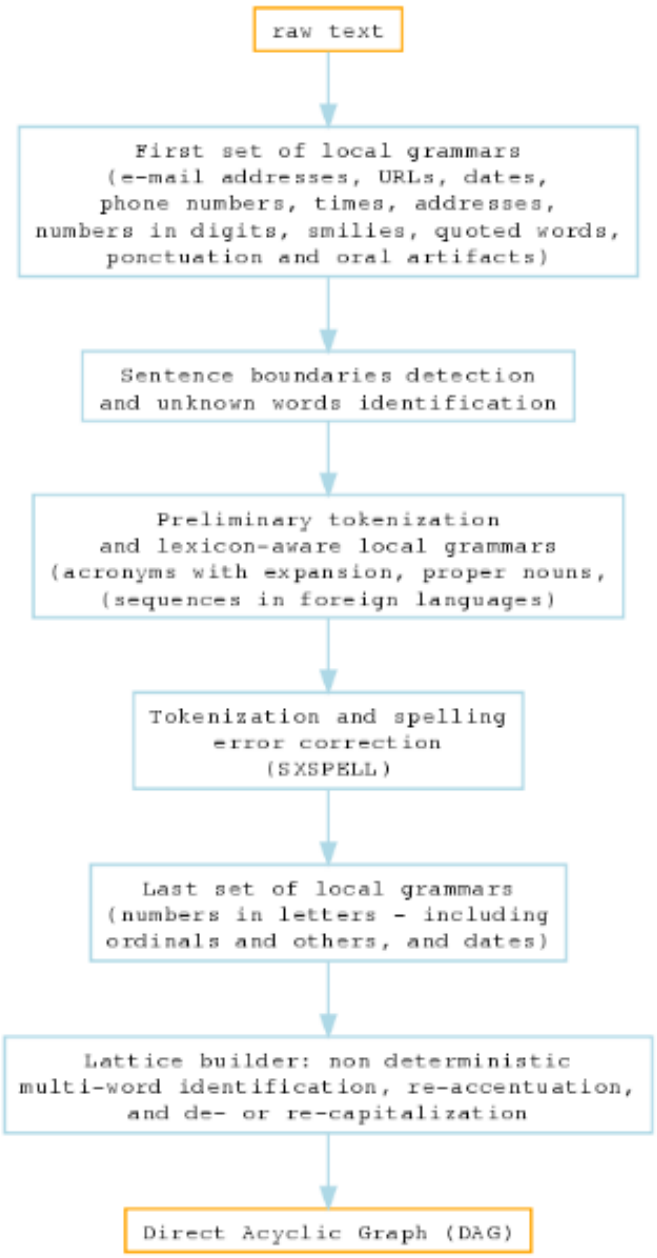


Figura 2.7: Arquitectura de *SxPipe* (Cabrera, 2008).

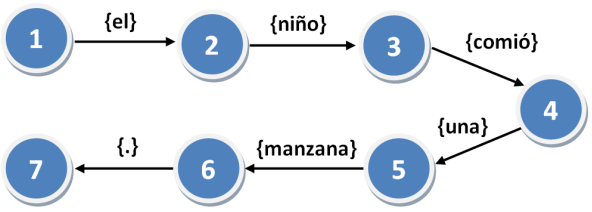


Figura 2.8: GAD para la oración *"El niño comió una manzana."*.

etiqueta es creada por el analizador léxico a partir de la información presente en la entrada o entradas correspondientes a una determinada forma dentro del léxico compilado. Una misma forma puede dar lugar a diferentes lemas, categorías léxicas o disponer de diferentes características sintácticas (marcos de subcategorización y redistribuciones). Ello implica que para una misma forma existan diferentes *hypertags*; esto es, entradas en el léxico compilado.

Ejemplo 5 *Siguiendo con el ejemplo presente en el anterior apartado para la oración* "El niño comió una manzana."*, SPMG Lexer obtendría como resultado del proceso de lematización y etiquetación la siguiente salida*[41]*:*

```
'C'(0,
    lemma{
      lex      => el,
      truelex  => el,
      lemma    => el,
      cat      => det,
      top      => det{def => (+), det => (+),
                      gender => masc},
      anchor   => tag_anchor{ name => _,
                              coanchors => []}
    },
    1
   ).

'C'(1,
    lemma{
      lex      => niño,
      truelex  => niño,
      lemma    => niño,
      cat      => adj,
```

[41] La salida mostrada ha sido simplificada con el fin de facilitar su compresión.

```
      top     => adj{gender => masc},
      anchor  => tag_anchor{ name => ht{
                     arg0 => arg{kind => kind[obj,
                                 vcomp,(-)],
                                 pcas => prep[de,
                                 (-)]},
                     arg1 => arg{kind => kind[vcomp,
                                  (-)],
                                 pcas => prep[a,(-)]},
                     arg2 => arg{kind => (-),
                                  pcas => (-)},
                     refl => (-)}, coanchors => []}
    },
    2
   ).

'C'(1,
    lemma{
      lex     => niño,
      truelex => niño,
      lemma   => niño,
      cat     => nc,
      top     => nc{gender => masc},
      anchor  => tag_anchor{ name => ht{
                     arg0 => arg{kind => kind[obj,
                               vcomp,(-)],
                               pcas => prep[de,(-)]},
                     arg1 => arg{kind => kind[vcomp,
                                         (-)],
                                  pcas => prep[a,(-)]},
                     arg2 => arg{kind => (-),
                                  pcas => (-)},
                     refl => (-)}, coanchors => []}
    },
    2
   ).

'C'(2,
    lemma{
      lex     => comió,
      truelex => comió,
      lemma   => comer,
      cat     => v,
      top     => v{diathesis => active,
```

```
                    mode => indicative,
                    person => 3,
                    tense => 'past-historic'},
      anchor  => tag_anchor{ name => ht{
                    arg0 => arg{kind => subj,
                                pcas => (-)},
                    arg1 => arg{kind => (-),
                                pcas => (-)},
                    arg2 => arg{kind => (-),
                                pcas => (-)},
                    diathesis => active, imp => '-',
                    refl => (-)}, coanchors => []}
    },
    3
   ).

'C'(2,
    lemma{
      lex     => comió,
      truelex => comió,
      lemma   => comer,
      cat     => v,
      top     => v{aux_req => avoir,
                    diathesis => active,
                    mode => indicative,
                    person => 3,
                    tense => 'past-historic'},
      anchor  => tag_anchor{ name => ht{
                    arg0 => arg{kind => subj,
                                pcas => (-)},
                    arg1 => arg{kind => obj,
                                pcas => (-)},
                    arg2 => arg{kind => (-),
                                pcas => (-)},
                    diathesis => active, imp => '-',
                    refl => (+)}, coanchors => []}
    },
    3
   ).

'C'(3,
    lemma{
      lex     => una,
      truelex => una,
```

```
      lemma   => una,
      cat     => det,
      top     => det{def => '-', det => (+),
                     gender => fem},
      anchor  => tag_anchor{ name => _,
                             coanchors => []}
    },
    4
   ).

'C'(3,
    lemma{
      lex     => una,
      truelex => una,
      lemma   => unir,
      cat     => v,
      top     => v{diathesis => active,
                   mode => imperative,
                   person => 3,
                   tense => present},
      anchor  => tag_anchor{ name => ht{
                  arg0 => arg{kind => subj,
                              pcas => (-)},
                  arg1 => arg{kind => obj,
                              pcas => (-)},
                  arg2 => arg{kind => kind[acomp,
                                           (-)],
                              pcas => prep[como,
                                           (-)]},
                  diathesis => active,
                  imp => '-', refl => (-)},
                  coanchors => []}
    },
    4
   ).

'C'(3,
    lemma{
      lex     => una,
      truelex => una,
      lemma   => unir,
      cat     => v,
      top     => v{diathesis => active,
                   mode => subjonctive,
```

```
                    person => 1,
                    tense => present},
      anchor  => tag_anchor{ name => ht{
                 arg0 => arg{kind => subj,
                             pcas => (-)},
                 arg1 => arg{kind => (-),
                             pcas => (-)},
                 arg2 => arg{kind => (-),
                             pcas => (-)},
                 diathesis => active, imp => '-',
                 refl => (-)}, coanchors => []}
    },
    4
   ).

'C'(4,
    lemma{
      lex     => manzana,
      truelex => manzana,
      lemma   => manzana,
      cat     => nc,
      top     => nc{gender => fem},
      anchor  => tag_anchor{ name => ht{
                    arg0 => arg{kind => kind[obj,
                                             vcomp,(-)],
                                pcas => prep[de,(-)]},
                    arg1 => arg{kind => kind[vcomp,
                                             (-)],
                                pcas => prep[a,(-)]},
                    arg2 => arg{kind => (-),
                                pcas => (-)},
                    refl => (-)}, coanchors => []}
    },
    5
   ).

'C'(5,
    lemma{
      lex     => '.',
      truelex => '.',
      lemma   => '.',
      cat     => poncts,
      top     => poncts,
      anchor  => tag_anchor{ name => _,
```

```
                                    coanchors => []}
    },
    6
   ).
```

Donde se aprecia que para la forma "una" *pueden existir distintas entradas en el léxico:*

- ***"una"**, con lema* "una" *y categoría léxica determinante.*
- ***"una"**, con lema* "unir" *y categoría léxica verbo, 3^a persona de imperativo.*
- ***"una"**, con lema* "unir" *y categoría léxica verbo, 1^a persona de subjuntivo*[42].

■

En el supuesto de que una forma presente en el GAD ofrecido por *SxPipe* no estuviese recogida en el LEFFE, dicha forma sería catalogada como desconocida (*Unknown Word*) y su conjunto de *hypertags* sería el propuesto en el ejemplo 6.

Ejemplo 6 *La forma* "califragilístico" *no se encuentra en el lexicón LEFFE. Ello obliga a* SPMG Lexer *a ofrecer la siguiente salida:*

```
'C'(0,
    lemma{ lex => califragilístico,
           truelex => califragilístico, cat => v },
    1
   ).
'C'(0,
    lemma{ lex => califragilístico,
           truelex => califragilístico, cat => nc },
    1
   ).
'C'(0,
    lemma{ lex => califragilístico,
           truelex => califragilístico, cat => adv },
    1
   ).
```

[42]Existen más entradas para la forma *"una"* pero no se han descrito para facilitar la legibilidad del ejemplo propuesto.

```
'C'(0,
    lemma{ lex => califragilístico,
           truelex => califragilístico, cat => adj },
    1
   ).
```

■

En el ejemplo 6, no se dispone de información acerca de la forma *"califragilístico"*. Como resultado se le asignan cuatro posibles categorías léxicas: verbo, nombre común, adjetivo y adverbio; que se tendrán en cuenta en el análisis posterior. En muchos casos, el conjunto de posibles categorías de la palabra desconocida se ve reducido gracias a la estructura gramatical del lenguaje, que las limita permitiendo la disminución de la ambigüedad en el análisis final.

2.3.2. Análisis sintáctico: SPMG y SPMG Parser

El conjunto de *hypertags* ofrecido por *SPMG Lexer* es la entrada inmediata de *SPMG Parser*, encargado de la siguiente etapa en la cadena de procesamiento lingüístico: el análisis sintáctico.

SPMG Parser es un analizador profundo de amplia cobertura para el español. Una descripción gramatical de alto nivel bajo la forma de *metagramática* (SPMG) sirve de punto de partida para la generación de una *gramática de adjunción de árboles* (GAR) mediante MGCOMP (De la Clegerie, 2005b). Está gramática GAR es transformada por el entorno DyALog (De la Clegerie, 2005a) en un analizador sintáctico.

2.3.2.1. Gramáticas de Adjunción de áRboles (GAR)

El analizador sintáctico empleado en la cadena en desarrollo se basa en las gramáticas GAR. A continuación se exponen las principales características que las definen. Para tratar automáticamente un lenguaje natural, es necesaria su formalización previa en un conjunto de reglas que denominaremos gramática (Martin, 2006).

Definición 1 *Una gramática* $\boldsymbol{G}$ *se define como una cuádrupla* $(\boldsymbol{N},\Sigma,\boldsymbol{P},\boldsymbol{S})$*:*

- $\boldsymbol{N}$*: Conjunto finito de* símbolos no-terminales *o* categorías sintácticas. *Ellos son producidos y descompuestos, a su vez, por reglas de reescritura.*
- Σ*: Conjunto de* símbolos terminales *o* categorías léxicas. *Son las palabras de la lengua no susceptibles de otras reescrituras.*

- ***P**: Conjunto de* reglas de reescritura *(o* producción*). Son de la forma* $\alpha \rightarrow \beta/\alpha$, $\beta \in (N \cup \Sigma)^*$, $\alpha \neq \epsilon$.
- ***S**:* Símbolo inicial o axioma. *Representa una frase* a priori *correctamente formada y que se reescribe en símbolos no-terminales y terminales por reglas de reescritura.*

■

Chomsky (Denning et al., 1978) establece una clasificación de cuatro tipos de gramáticas formales que, a su vez generan cuatro tipos diferentes de lenguajes formales.

Definición 2 *Las gramáticas formales se clasifican según las restricciones que se imponen a sus producciones, y la Jerarquía de Chomsky establece estos cuatro niveles:*

1. ***Gramáticas sin restricciones o de tipo 0:** Incluyen todas las gramáticas formales. Sus producciones tiene la forma general:* $\alpha \rightarrow \beta$.
2. ***Gramáticas dependientes del contexto o de tipo 1:** Sus producciones tiene la forma general* $\alpha A \beta \rightarrow \alpha \gamma \beta$ *donde* $A \in N$; α, $\beta \in (N \cup \Sigma)^*$ *y* $\gamma \in (N \cup \Sigma)^+$. *Las cadenas* α *y* β *pueden ser la cadena vacía* ϵ, *pero* γ *ha de ser distinto del vacío.*
3. ***Gramáticas independientes del contexto (GIC) o de tipo 2:** Sus reglas son de la forma* $A \rightarrow \gamma$ *con* $A \in N$ *y* $\gamma \in (N \cup \Sigma)^*$.

   Los lenguajes de programación son lenguajes independientes del contexto y deterministas.
4. ***Gramáticas regulares o de tipo 3:** Sus producciones son de la forma* $A \rightarrow a$, *o bien* $A \rightarrow aB$, $a \in \Sigma$ *y* $A,B \in N$.

■

Las *gramáticas de adjunción de árboles* se encuentran entre los tipos 1 y 2, y fueron definidas inicialmente por Joshi, Levy y Takahashi (Martin, 2006).

Definición 3 *Una* gramática de adjunción de árboles *(GAR) se define formalmente como una quíntupla* $\boldsymbol{G} = (\boldsymbol{N},\Sigma,\boldsymbol{S},\boldsymbol{I},\boldsymbol{A})$:

- $\boldsymbol{N}$: *Conjunto de símbolos no-terminales.*
- Σ: *Conjunto de símbolos terminales.*
- $\boldsymbol{S}$: *Símbolo inicial.*

- ***I**: Conjunto finito de árboles iniciales. Se caracterizan porque su raíz está etiquetada por el axioma de la gramática, sus nodos interiores están etiquetados por símbolos no-terminales y sus nodos hojas están etiquetados por terminales o por la palabra vacía, denotada por ϵ.*

- ***A**: Conjunto de árboles auxiliares. Se distinguen por su recursividad: poseen un nodo hoja marcado para la operación de adjunción (*pie*) cuya etiqueta es un símbolo no-terminal de la misma categoría que el nodo raíz. El camino desde el nodo raíz hasta el nodo pie recibe el nombre de* espina.

Los árboles $\boldsymbol{I} \cup \boldsymbol{A}$ se denominan árboles elementales *de la gramática. Éstos se pueden combinar entre sí para crear* árboles derivados.

■

Dos operaciones permiten la combinación sobre el conjunto de árboles elementales: La *sustitución* y la *adjunción*.

Definición 4 *Dados los árboles β y δ, se define la sustitución como la operación de injertar el árbol inicial β en una hoja de un árbol δ con la misma etiqueta y marcada para esta operación por el símbolo $\downarrow$.*

■

Definición 5 *Dados los árboles α y γ, la operación de adjunción se define como la inserción de un árbol auxiliar o derivado α en un nodo de la misma categoría, marcado para la adjunción con el símbolo *, dentro de un árbol elemental o derivado γ.*

■

A diferencia de las GICs, en las cuales el árbol derivado contiene toda la información necesaria para determinar qué operaciones han sido realizadas sobre qué nodos a lo largo de una derivación, en las GARs dicha información no se puede obtener directamente a partir del árbol derivado. En consecuencia, surge un objeto diferente, denominado *árbol de derivación*.

Definición 6 *Dentro del marco de las GARs, un* árbol de derivación *detalla de modo inequívoco cómo ha sido construido un árbol derivado, esto es, especifica una sucesión de adjunciones o sustituciones indicando para cada una de ellas el nodo en el cual tuvo lugar y el árbol auxiliar involucrado. La raíz de un árbol de derivación deberá estar etiquetada por un árbol inicial. Los demás nodos del árbol de derivación estarán etiquetados por un árbol auxiliar y por el nodo en el que se realizó la operación, indicando si es de adjunción o sustitución (Alonso, 2000).*

■

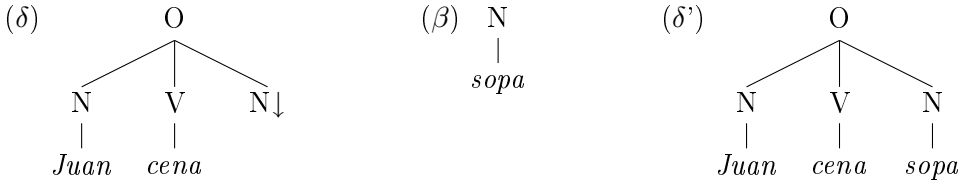


Figura 2.9: Ejemplo de sustitución.

Los árboles iniciales, sobre los cuales opera la sustitución, son utilizados, por ejemplo, para la representación de nombres propios o comunes funcionando como complemento nominal o preposicional.

Ejemplo 7 *En la figura 2.9 se presenta un ejemplo de la operación de sustitución. En él se puede ver como el árbol inicial β con el terminal* "sopa" *ha sido sustituido en el nodo N↓ del árbol derivado δ con los terminales* "Juan cena"*, dando como resultado el árbol derivado δ' (*"Juan cena sopa"*).*

■

Los árboles auxiliares, para la adjunción, son utilizados para representar los modificadores tales como los adjetivos, adverbios o subordinadas relativas, así como los verbos como completivos, verbos modales y los auxiliares.

Ejemplo 8 *En la figura 2.10 se presenta un ejemplo de la operación de adjunción. El árbol auxiliar α con el adverbio* "rápidamente" *como terminal ha sido adjuntado al nodo V del árbol derivado γ con los terminales* "Pablo corrió"*, dando como resultado el árbol derivado γ' (*"Pablo corrió rápidamente").

■

Ejemplo 9 *Para la operación realizada en la figura 2.9, el árbol de derivación correspondiente se muestra en la figura 2.11. En él se indica que sobre el tercer nodo (empezando por la izquierda) del árbol inicial δ se ha realizado una operación de sustitución en la cual ha participado el árbol auxiliar β.*

■

Existen distintas variantes de las GARs. El analizador sintáctico que emplea la cadena objeto de este proyecto integra algunas de las siguientes variantes:

- **GAR lexicalizada (GARL):** Una *gramática* se dice que está *lexicalizada* si toda estructura elemental está asociada con un símbolo

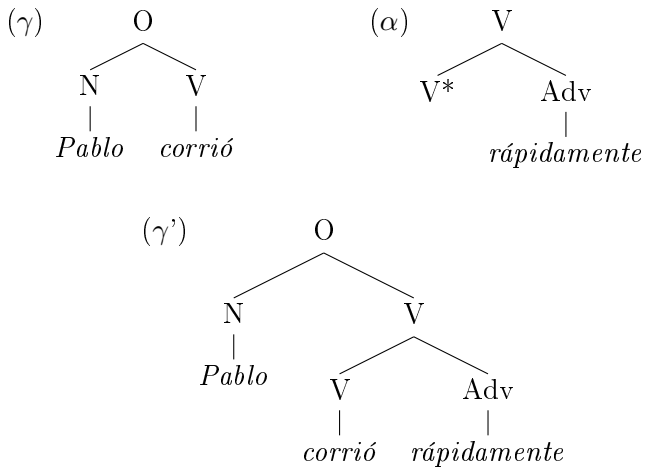


Figura 2.10: Ejemplo de adjunción.

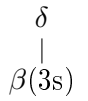


Figura 2.11: Árbol de derivación de la operación de la figura 2.9.

terminal, denominado *ancla*. En tal caso, también podemos considerar una gramática como un lexicón en el que cada palabra está asociada con un conjunto de estructuras sintácticas elementales en las que dicha palabra actúa como *ancla*. Concretamente, una GAR se dice que está lexicalizada si cada uno de los árboles elementales contiene al menos un símbolo terminal en su frontera. Dicho terminal es el *ancla* del árbol elemental (Martin, 2006).

- **GAR basada en estructuras de rasgos (GARE)**[43]**:** Son una variante de las GARs en la cual los nodos de los árboles elementales pueden estar decorados con estructuras de rasgos[44] que describen el nodo y su relación con otros nodos del mismo árbol. Las operaciones de adjunción y sustitución se definen en términos de la unificación de estructuras de rasgos, por lo que las restricciones de adjunción pueden ser modeladas a través del éxito o del fracaso de la unificación entre las estructuras de rasgos de los nodos (Martin, 2006).

[43] También conocida como GAR basada en unificación.

[44] Una *estructura de rasgos* está formada por un conjunto de pares atributo-valor, tal que un valor puede ser atómico (valor simple e indivisible) u otra estructura de rasgos. Por ejemplo, la estructura de rasgos presente en la figura 2.12.

$$\left| \begin{array}{c} superior : < cat >= SN \\ < genero >= SG \\ < persona >= 3 \end{array} \right|$$

Figura 2.12: Una estructura de rasgos.

$$\left| < genero >= SG \right| \cup \left| < genero >= PL \right| = FAIL$$
$$\left| < genero >= SG \right| \cup \left| < persona >= 3 \right| = \left| \begin{array}{c} < genero >= SG \\ < persona >= 3 \end{array} \right|$$

Figura 2.13: Ejemplos simples de unificación de estructuras de rasgos.

Ejemplo 10 *En la metagramática SPMG a cada nodo de un árbol GAR, se asocian las estructuras de rasgos* Superior[45] *e* Inferior[46]. *Estas establecen un enlace entre un nodo y su ancestro y descendientes. La figura 2.12 muestra una estructura de rasgos* Superior *asociada a un nodo concreto, que indica que aquellos nodos que lo dominan tienen la categoría de sintagma nominal, su género es singular y tienen tercera persona.*

■

Ejemplo 11 *La figura 2.13 ilustra dos unificaciones muy simples. La primera que falla puesto que los dos valores diferentes afectan al mismo rasgo, y una segunda que tiene éxito al generar la unión de dos estructuras a unificar.*

■

- **Gramática de inserción de árboles (GIR):** Las GIRs son una variante de las GARs que introducen una restricción sobre los árboles auxiliares, tal que los mismos únicamente pueden ser insertados a la izquierda o a la derecha del nodo de adjunción. Esta condición implica concretamente que los árboles auxiliares tengan su espina como frontera izquierda o derecha. El mayor interés de las GIRs proviene del hecho de que son analizables, como las GICs, con una complejidad $O(n^3)$ cuando las GARs tiene una complejidad $O(n^6)$, donde n denota la longitud de la cadena de entrada. Es más, la mayor parte de las

[45] Recoge en forma de estructura de rasgos las restricciones que debe mantener un determinado nodo con su ancestro.

[46] Recoge en forma de estructura de rasgos las restricciones que debe mantener un determinado nodo con sus descendientes.

gramáticas GAR son esencialmente GIR, siendo posible la construcción de analizadores sintácticos híbridos GAR/GIR (Alonso et al., 2002).

Las GARs y sus variantes constituyen un formalismo con propiedades atractivas para caracterizar las descripciones estructurales asociadas con las frases de las lenguas naturales, tanto para el análisis como para la generación. Entre éstas, podemos distinguir (Martin, 2006):

- **Sensibilidad suave al contexto:** La clase de lenguajes generados por las gramáticas GAR lexicalizadas contienen la clase de lenguajes independientes del contexto y está contenida dentro de la clase de los lenguajes dependientes del contexto.

- **Dominio extendido de localidad:** Las GARs poseen un dominio de localidad[47] más amplio que las GIC, de tal modo que permiten la localización de dependencias de larga distancia dentro de una misma estructura elemental. Esto se debe a que el dominio de localidad viene especificado no ya por producciones independientes del contexto sino por árboles.

  Ejemplo 12 *En la GAR lexicalizada de la figura 2.14 se observa que el árbol α especifica la dependencia entre el verbo, el sujeto y el complemento, manteniendo el nodo VP en la gramática. En una GIC no se pueden especificar las dependencias verbo-sujeto y verbo-objeto sin perder el nodo VP. En cambio, estas dependencias se pueden representar fácilmente en un árbol elemental.*

  ■

- **Factorización de la recursión del dominio:** Los árboles elementales, estructuras elementales de las GARs, son los dominios sobre los cuales se establecen dependencias tales como la concordancia, subcategorización y relleno de huecos. La operación de adjunción, mediante la inserción de árboles auxiliares dentro de árboles elementales, permite que tales dependencias sean de larga distancia, aunque hayan sido especificadas localmente en un solo árbol elemental.

  Ejemplo 13 *En la figura 2.15, el árbol γ' es el resultado de la adjunción del árbol α en el nodo VP del árbol γ. A pesar de ello, el árbol γ' mantiene las dependencias descritas por el árbol γ (verbo-sujeto y verbo-objeto). Esto se debe a que las dependencias recogidas en las estructuras arbóreas son de larga distancia.*

[47] El dominio de localidad, en este caso, hace referencia a la extensión o rango de dependencias sintácticas que es capaz de representar una gramática en una estructura elemental. Las construcciones arbóreas recogen una mayor cantidad de dependencias (y de una mayor longitud) que una regla de producción.

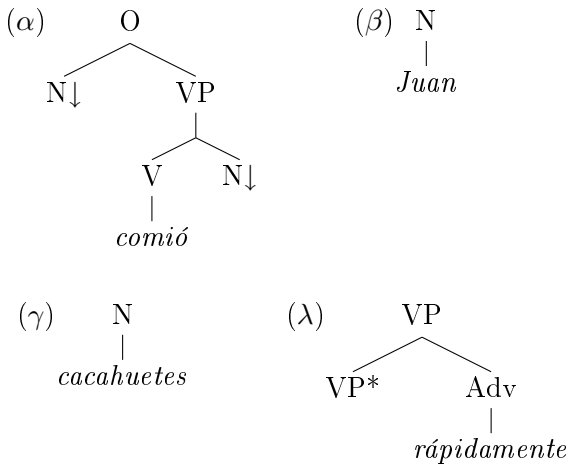


Figura 2.14: Dominio extendido de localidad de las GARs.

■

- **Gramáticas lexicalizadas:** Cada árbol de la gramática describe una realización posible de una palabra del léxico considerado. La gramática está centrada sobre el léxico.

  Ejemplo 14 *La figura 2.16 recoge los árboles GAR que describen las diferentes estructuras sintácticas en las que puede aparecer la forma* "encontró".

  ■

- **Análisis en tiempo polinomial:** La complejidad del análisis es $O(n^6)$ en general, y $O(n^3)$ para las GIRs, donde n denota la longitud de la cadena de entrada.

En concreto, el analizador sintáctico desarrollado se fundamenta en gramáticas GAR basadas en unificación, pero no lexicalizadas, puesto que no todos los árboles están anclados con léxico. De hecho, el analizador sintáctico construido es un híbrido GAR/GIR.

Del lado de los inconvenientes en GARs, podríamos referirnos a su diseño y mantenimiento. En efecto, una lengua como el castellano podría necesitar varios miles de árboles elementales para alcanzar una cobertura razonable. En nuestro caso, el problema se ha abordado a través de la utilización del potencial de las *metagramáticas* y del entorno operativo DyALog (De la Clegerie, 2005a), ambos descritos en los siguientes apartados.

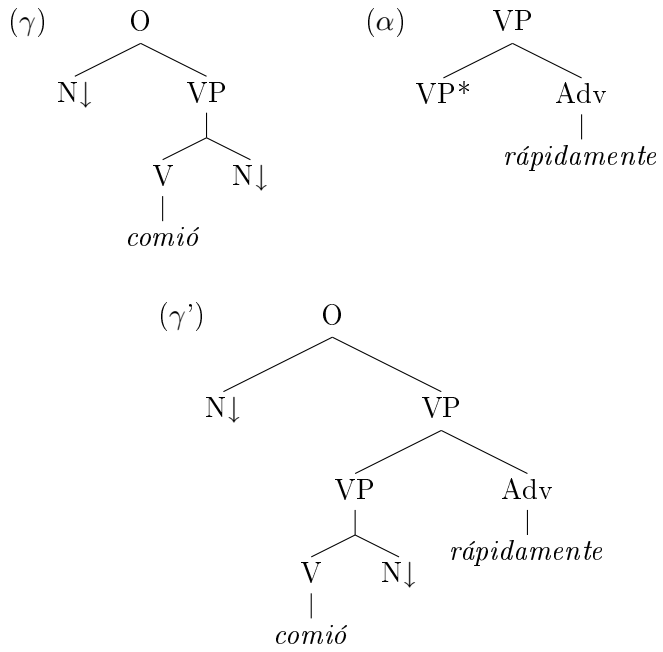


Figura 2.15: Factorización de la recursión del dominio de las GARs.

2.3.2.2. SPMG y MGCOMP

Con el fin de cubrir lo más ampliamente posible las construcciones sintácticas de una determinada lengua, es necesario describir el máximo número de estructuras elementales. El tamaño de las gramáticas de amplia cobertura convierte su diseño y mantenimiento en una difícil y ardua tarea lo que hace necesaria que ésta sea (semi-)automática. En efecto, el dominio de localidad propio de los árboles GAR entraña una explosión combinatoria del número de árboles, así como del de sus correspondientes subárboles. Por ejemplo, la estructura de un árbol verbal se va a encontrar en todos los árboles anclados por verbos. Modificar la descripción de uno de los subárboles del árbol verbal implica *a priori* la modificación de todos los árboles que contengan este árbol verbal, derivando en previsibles problemas de mantenimiento. Las metagramáticas aportan una solución elegante en este contexto, pero sin permitir la explosión combinatoria (Thomasset y De la Clegerie, 2005). Este tipo de estructuras son una de las respuestas aparecidas en la comunidad GAR para hacer frente a los problemas surgidos en el desarrollo de gramáticas GAR de gran talla (Martin, 2006).

Las metagramáticas introducen un alto nivel de abstracción en la descripción de las restricciones sobre y entre nodos que definen las estructuras sintácticas elementales de la lengua, reagrupándolas en clases relativamente simples e insertadas dentro de una jerarquía de herencia múltiple. Entre estas

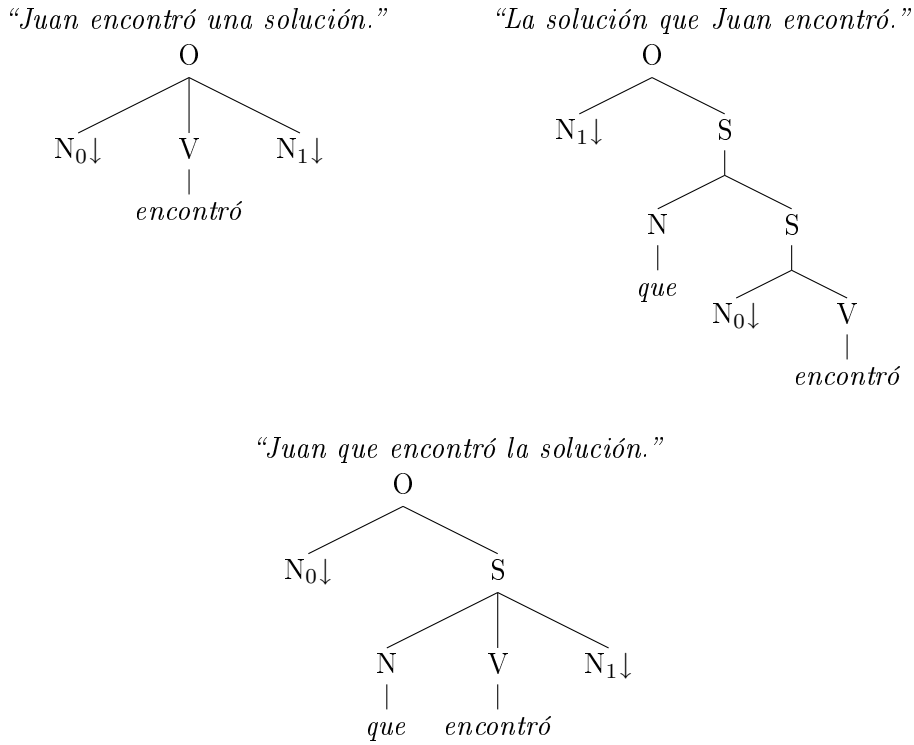


Figura 2.16: Estructuras sintácticas para la forma *"encontró"*.

restricciones se puede citar el dominio (estricto o inmediato) de un nodo sobre otro, la precedencia lineal, así como las restricciones de decoración (por las estructuras de rasgos) sobre los nodos o sobre la clase (Thomasset y De la Clegerie, 2005)[48]. Esta descripción gramatical de alto nivel sirve de punto de partida para la generación de una GAR para un lenguaje concreto. En el marco de este proyecto, se ha desarrollado una *metagramática* para el español (SPMG, SPanish MetaGrammar), con el fin de obtener una gramática GAR para nuestra lengua. Las metagramáticas, incluyendo SPMG, tienen las siguientes características (Martin, 2006):

- **Restricciones topológicas:** Cada clase de la jerarquía contiene una descripción parcial de la estructura de los árboles GAR elementales. Para ello, se emplean las relaciones siguientes: (=) *igualdad*[49], (<) *precedencia*[50], ($\gg$) *dominancia inmediata o directa*[51] y ($\gg+$) la *dominancia indirecta*[52].

- **Descripción parcial de árbol y árboles GAR minimales:** Los árboles descritos son a menudo *cuasi-árboles*. Un *cuasi-árbol* es una descripción que permite construir un número infinito de árboles que no violan las restricciones.

  Ejemplo 15 *La figura 2.17 ofrece dos ejemplos de cuasi-árboles a partir de una misma descripción. Si en la descripción se utilizan relaciones de dominancia indirectas como es el caso del ejemplo propuesto, existe un número arbitrariamente grande de árboles que pueden ser construidos a partir de la misma. En la figura 2.17 se muestran dos posibles árboles que cumplen con las restricciones descritas y que además permiten introducir nuevos nodos, por ejemplo, entre la pareja de cuasi-nodos X y Z.*

  ■

  Para solucionar el problema anterior entra en juego el concepto de *árbol minimal*. Éste es un cuasi-árbol para el cual se reducen las relaciones (de dominancia, u otras) indirectas por relaciones directas, evitando de esta forma el incremento de interpretaciones de una misma descripción topológica y prohibiendo, además, la inserción infinita de nodos entre dos cuasi-nodos.

[48] Más adelante se detallan estos conceptos.

[49] Dos identificadores de nodos relacionados mediante el operador de igualdad equivale a afirmar que ambos identificadores se refieren al mismo nodo.

[50] Mantiene el orden entre dos nodos hermanos. Uno ha de preceder al otro.

[51] Un nodo domina directamente a otro cuando el primero es padre del segundo.

[52] Un nodo domina indirectamente a otro cuando el primero es ancestro no directo del segundo. Por ancestro no directo se entiende que el nodo ubicado en un nivel superior del árbol (nodo dominante), no es padre del nodo dominado.

description:

(X <Y) ∧ (Z ≫+ Y) ∧ (P ≫+ X) ∧ (P ≫+ Z)

cuasi-árboles:

Figura 2.17: Una descripción y sus dos cuasi-árboles.

description:

(X <Y) ∧ (Z ≫ Y) ∧ (P ≫ X) ∧ (P ≫ Z)

cuasi-árboles:

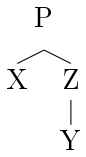


Figura 2.18: Una descripción y su árbol minimal.

Ejemplo 16 *En la figura 2.18 aparece una descripción topológica que únicamente da lugar a una interpretación y además, gracias a las relaciones de dominancia directa, no permite introducir más nodos entre sus nodos. El cuasi-árbol presente en la figura 2.18, es un árbol minimal. Se trata de la misma descripción utilizada en el ejemplo anterior. Únicamente se han sustituido las dominancias indirectas por las directas. Ello origina un único árbol y, además, minimal.*

■

- **Restricciones de unificación:** Otro conjunto de restricciones está determinado por las declaraciones o ecuaciones de estructuras de rasgos[53] asignadas a ciertos nodos o directamente a una clase, y sujetos a unificación.
- **Recursos y necesidades:** Por otro lado, cada clase puede ser *consumidora*[54] o *proveedora*[55] de uno o varios recursos particulares interpretados como funciones sintácticas (la concordancia, por

[53] Las estructuras de rasgos han sido detalladas anteriormente.
[54] Clase que solicita un recurso a otra clase de la metagramática.
[55] Clase que ofrece un recurso a otra clase de la metagramática.

ejemplo). Cada recurso puede ser consumido directamente a nivel de clase o bien a través de un *espacio de nombres*[56]. Los espacios de nombres permiten a una clase demandar varias veces un mismo recurso siempre y cuando se ubique la petición en diferentes *espacios de nombres* para evitar conflictos. Por ejemplo, un recurso de *concordancia de género y número* entre un nodo y su padre puede ser requerido varias veces por distintas clases[57].

- **Guardas:** Una clase puede contener *guardas* sobre ciertos nodos. Una *guarda* es una restricción condicional que dependiendo de la existencia o no de un determinado nodo conllevará la validación o no de las ecuaciones de estructuras de rasgos descritas en la parte derecha de la guarda. Una guarda se expresa mediante ecuaciones sobre caminos de rasgos[58]. Son de la forma:

  ```
  ~X =>node(Y).un.camino.de.rasgos = valeur(v1), ...;
  ```

  para expresar el caso de la no existencia del nodo `X`, o

  ```
  X =>node(Z).un.camino.de.rasgos = valeur(v2), ...;
  ```

  para expresar el caso de la presencia de este mismo nodo `X`. Un ejemplo de las mismas se expone en la siguiente propiedad de las metagramáticas.

- **Lenguaje concreto de la metagramática:** Finalmente, para implementar las clases de la metagramática que describen un determinado lenguaje, se utiliza un formalismo concreto (De la Clegerie, 2005b), descrito en el ejemplo 17.

  Ejemplo 17 *A continuación se presenta, la clase representante de los nombres comunes para el castellano en el lenguaje que describe una metagramática:*

```
1 class cnoun{
2    <: noun;
```

[56] Un espacio de nombres es un contexto en el que un grupo de uno o más identificadores unívocos pueden existir. De tal forma que, un mismo identificador puede independientemente ser definido en múltiples espacios de nombres. Para acceder a cada identificador es necesario indicar a qué espacio de nombres nos estamos refiriendo.

[57] Un caso concreto de ello se puede ver en el ejemplo 17.

[58] Un camino de rasgos, es la forma de indicar a que rasgo o atributo concreto de una estructura de rasgos nos estamos refiriendo. Por ejemplo, si queremos indicar que el valor *número* de la estructura de rasgos *nodo(género, número, persona)* es igual a *plural* tendremos que añadir la siguiente ecuación *nodo.número = plural*.

```
3
4     N2 >> N; N >> Nc;
5     N2 >> det;
6     det < N;
7     Nc = Ancre;
8
9     node N : [cat: N];
10    node det : [cat: det, type: subst];
11
12    - nc::agreement; Nc = nc::N;
13    - n::agreement; N = n::N;
14
15    node(det).top.number = node(N2).bot.number;
16    node(det).top.gender = node(N2).bot.gender;
17    node(det).top.wh = node(N2).bot.wh;
18    node(Ancre).bot.person = value(3);
19
20    det =>
21       node(N2).bot.sat = value(+);
22    ~ det =>
23       node(N2).bot.wh = value(-);
24 }
```

La línea 2 indica que la clase `cnoun` *hereda de la clase padre* `noun` *y viene a completarla.*

Las líneas 4, 5, 6 y 7 declaran las relaciones topológicas que deben mantener los nodos implicados en el fragmento del árbol descrito. Nótese que el nodo `det` *debe preceder al nodo* `N` *(línea 6). El árbol al que nos referimos aparece en la figura 2.19.*

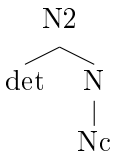


Figura 2.19: Estructura sintáctica de un sintagma nominal.

Las líneas 9 y 10 declaran directamente las estructuras de rasgos para los nodos `N` *y* `det`*. Además, las líneas de la 15 a la 18 expresan restricciones de unificación a través de ecuaciones entre caminos de rasgos. Concretamente, en estas condiciones se establece la concordancia de género y número entre los nodos* `det` *y* `N2`*. Además,*

también se comprueba si ambos pertenecen a una oración interrogativa (`wh`*) y se determina que la persona de un nombre común es la tercera.*

Las líneas 12 y 13 indican que la clase `cnoun` *requieren dos veces el recurso* `agreement` *(concordancia) sobre espacios de nombres diferentes. Este recurso está proveído por otra clase de la metagramática que tiene un nodo local denominado* `N`*. El nodo local se emplea para poder aplicar las restricciones proveídas por ese recurso sobre los de la clase que lo demandan*[59]*.*

La primera guarda, presente en las líneas 20 y 21, indica que si el sustantivo que modela la clase tiene un determinante (`det`*), entonces el sintagma nominal que reúne a ambos (*`N2`*) es saturado*[60]*.*

La segunda guarda, que ocupa las líneas 22 y 23, describe que si el sustantivo no esta acompañado por un determinante[61]*, entonces no se trata de un sintagma nominal interrogativo, esto es, un sintagma nominal dentro de una oración interrogativa. Esta guarda se basa en la idea de que un sustantivo en una oración interrogativa siempre está acompañado por un determinante interrogativo. Por ejemplo, el determinante* "cuántas" *acompaña al sustantivo* "manzanas" *en la siguiente oración:* "¿Cuántas manzanas comieron?"*.*

■

El siguiente paso en la construcción del analizador sintáctico es la compilación y transformación de la metagramática española, SPMG, en una gramática GAR equivalente. Esta tarea es encomendada al compilador de metagramáticas MGCOMP (Thomasset y De la Clegerie, 2005). El proceso de compilación de SPMG se desarrolla a lo largo de cuatro etapas principales (Martin, 2006):

1. Acumulación de restricciones de clases de la jerarquía de herencia en las hojas de la metagramática, cuyo conjunto es la conjunción de las propias y las de sus ancestros.

2. El conjunto de *clases neutras* es calculado a partir del conjunto de terminales. Estas últimas, en ocasiones, no recogen todas las restricciones y características pertenecientes al fenómeno sintáctico que modelan, sino que parte de ellas puede ser relegada a otras clases que las proveen bajo forma de recurso[62]. Por tanto, una clase neutra es

[59] Como si se tratasen de variables locales de una función ubicada en otra clase.

[60] Se dice que un sintagma nominal es saturado, cuando el núcleo del mismo está acompañado por un determinante.

[61] El símbolo $\sim$ delante de un nodo, indica la ausencia de este en el árbol.

[62] Utilizar el concepto de recurso evita la duplicación de código, puesto que un mismo recurso puede ser requerido por distintas clases.

el resultado de reunir bajo una misma estructura las restricciones y características que anteriormente son relegadas a otra u otras clases terminales a través de los recursos. De esta forma, las restricciones de las terminales implicadas son conservadas en la neutra resultante.

3. Cada clase neutra $\boldsymbol{C}$, que contiene un conjunto de nodos N_C recoge básicamente restricciones de dos naturalezas: restricciones de unificación de estructura de rasgos que decoran un subconjunto de nodos $U_C \subseteq N_C$ y restricciones topológicas sobre otro subconjunto $T_C \subseteq N_C$, cuya intersección con U_C es potencialmente no nula.

   Concretamente, el compilador MGCOMP para cada clase neutra, remonta la red de herencia de la metagramática. Para cada ancestro se operan las unificaciones de estructuras de rasgos entre nodos para comprobar si se satisfacen. Ello implica el cierre transitivo de todas las restricciones a verificar. Si al menos una verificación de restricciones falla[63] entonces el ascenso por la jerarquía se detiene, la clase neutra actual se descarta [64] y una nueva clase neutra es tratada. De esta forma, únicamente se conservan aquellas clases neutras cuyas restricciones se satisfacen.

4. Generación de *árboles elementales y minimales GAR* según las restricciones contenidas en las clases supervivientes. El conjunto de árboles producido a partir de las clases neutras constituyen la gramática GAR resultante de la compilación de la metagramática original.

2.3.2.3. DyALog y SPMG Parser

Seguidamente, a partir de la gramática GAR del español, y a partir de DyALog (De la Clegerie, 2005a), se obtiene un analizador sintáctico profundo, robusto, híbrido GAR/GIR y de amplia cobertura para el castellano, denominado *SPMG Parser*.

A diferencia de otras gramáticas GAR (De la Clegerie, 1999), *SPMG Parser* se caracteriza por disponer de un número de árboles reducido. Ello se consigue gracias a la capacidad descriptiva de las metagramáticas y los mecanismos de factorización ofrecidos por DyALog (De la Clegerie et al., 2009; De la Clegerie, 2006).

DyALog es un entorno de compilación y ejecución de analizadores sintácticos tabulares (De la Clegerie, 2005a, 1999). En particular, realiza un análisis previo de la gramática GAR SPMG para determinar, cuales son los árboles que pueden ser compilados en árboles GIR, ya que éstos ofrecen una complejidad menor. Con el fin de reducir el número de árboles de la gramática

[63] Una unificación imposible o un nodo que se precede a él mismo por transitividad.
[64] Y por relación de herencia, el conjunto de la descendencia de la clase que ha fallado.

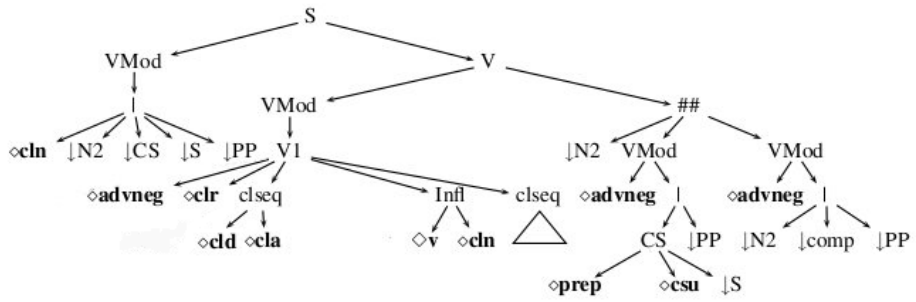


Figura 2.20: Árbol `#111` (simplificado).

GAR, DyALog realiza un proceso de factorización mediante la aplicación de operadores sobre subárboles de la gramática, tal como la disyunción, el cierre transitivo o el entrelazamiento de dos secuencias de subárboles. Estos operadores no cambian la naturaleza del formalismo gramatical, pero permiten reducir exponencialmente el número de árboles que componen la gramática GAR (De la Clegerie et al., 2009). Ello implica disponer de una gramática mucho más compacta y eficaz a la hora de construir el analizador sintáctico.

Por otra parte, DyALog no impone restricciones de lexicalización sobre los árboles de la gramática GAR, lo que implica que ciertos árboles de la GAR para el español no estén anclados.

Ejemplo 18 *La complejidad de los árboles que componen la gramática GAR se ilustra en la figura 2.20 representando una vista simplificada de un árbol verbal canónico para la voz activa. Este árbol* `#111` *resulta del cruzamiento de 25 clases terminales, comprende 43 nodos y está controlado por 35 guardas*[65]*, donde:* `S` *es la oración,* `VMod` *es un modificador verbal (incluye al sujeto de la oración),* `N2` *es un sintagma nominal,* `CS` *es una subordinada conjuntiva,* `PP` *es un sintagma preposicional,* `V1` *es un sintagma verbal,* `Infl` *incluye al verbo y a un clítico nominativo,* `comp` *es un atributo,* `cln` *es un clítico nominativo,* `cla` *es un clítico acusativo,* `cld` *es un clítico dativo,* `prep` *es una preposición,* `csu` *es una conjunción subordinada,* `advneg` *es un adverbio de negación,* `clr` *es un clítico reflexivo,* `clseq` *es una secuencia de clíticos y* `v` *es el verbo y ancla del árbol.*

■

Un árbol, el de la figura 2.20, cubre la realización de numerosas construcciones sintácticas, muchas más que las permitidas por una única

[65] Un árbol con este gran número de guardas y decoraciones sería extremadamente difícil de escribir a mano. Ello justifica más aún el empleo de las metagramáticas.

palabra del léxico. Según la idea inicial, de que cada árbol de la gramática GAR recogiese las posibles realizaciones de cada palabra del léxico, el número de árboles crecería proporcionalmente al número de palabras que forman aquél. Bajo la máxima de reducir el número de árboles de la gramática, se ha retomado la idea iniciada por Kinyon (De la Clegerie et al., 2009). Se trata de asignar una *hypertag*[66] al ancla de los árboles anclados (Thomasset y De la Clegerie, 2005). Dicha *hypertag* va a describir mediante estructuras de rasgos a un conjunto de palabras del léxico que reúnen unas determinadas características. De esta forma, un mismo árbol, como él del ejemplo propuesto, puede recoger la realización de varios verbos que comparten un comportamiento similar. Esto permite reducir considerablemente el número de árboles de la gramática. La *hypertag* del ancla de un árbol se especializa o generaliza en función de la información descrita en él. Estas etiquetas de cada árbol son creadas por DyALog (De la Clegerie, 2005a) a partir de las restricciones descritas mediante estructuras de rasgos presentes en las decoraciones de los nodos y en las ecuaciones de las guardas de la metagramática.

Por lo tanto, el anclaje entre las palabras del texto, etiquetadas previamente por *SPMG Lexer* mediante *hypertags*, y los árboles de la gramática GAR, se lleva a cabo por unificación entre las estructuras de rasgos de las etiquetas de las palabras con las asociadas a los árboles. El resultado de esta operación permitirá seleccionar aquellas construcciones (árboles) autorizadas para una determinada palabra[67].

Ejemplo 19 *De este modo, la figura 2.21 muestra la* hypertag *asociada al árbol* `#111`*, mientras que la figura 2.22 muestra la etiqueta referente al verbo* "pasear" *presente en nuestro léxico*[68]*. Entre todos los árboles presentes en la gramática GAR del español, se selecciona únicamente aquellos cuyas* hypertags *unifiquen correctamente con las asignadas a las palabras del texto de entrada. Concretamente, y en este caso, la* hypertag *de la forma* "pasear" *unificaría de forma satisfactoria con la correspondiente al árbol* `#111`*. En ambas etiquetas se recoge una construcción verbal canónica que autoriza un objeto y un grupo preposicional introducido por la preposición* "a"*. Por lo tanto, la realización del árbol propuesto está permitida para la palabra* "pasear".

[66] Se trata de una etiqueta con información morfosintáctica empleada tanto para etiquetar las palabras de entrada como para anclar los árboles de la gramática GAR. La información que recogen se encuentra representada bajo forma de estructuras de rasgos. En el apartado 2.3.1.1 de este capítulo se ha descrito su estructura.

[67] He aquí la razón por la cual es necesario que *SPMG Lexer* lleve a cabo una transformación de la información del LEFFE en *hypertags*, con el fin de unificar los formatos empleados por ambos analizadores, léxico y sintáctico.

[68] En el caso de la etiqueta asignada a la palabra *"pasear"* únicamente se muestra en la figura 2.22 el apartado `anchor` de la *hypertag*, puesto que se trata de la principal información destinada a la creación del enlace entre el léxico y las estructuras sintácticas.

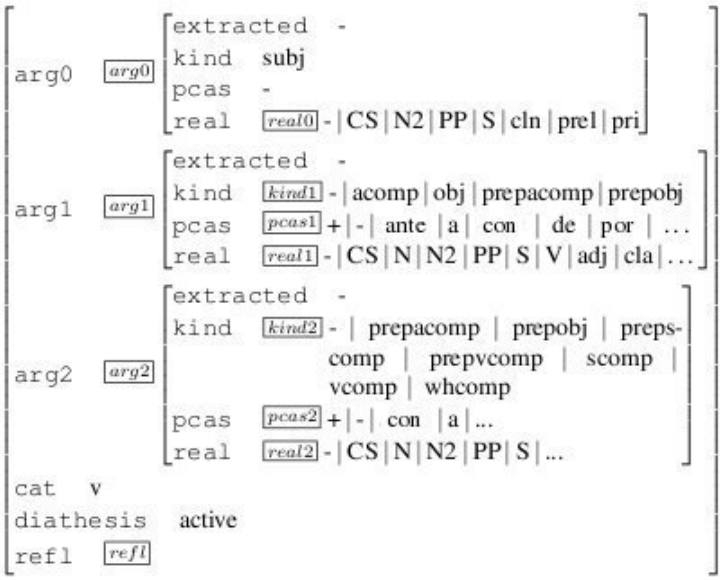


Figura 2.21: *Hypertag* del árbol `#111`.

■

A su vez, también se permiten *coanclas lexicales*. Se trata de disponer de la posibilidad de añadir en los árboles GAR lexicalizados, otros anclajes con el léxico aparte del ancla principal. Éstos son las coanclas, que a diferencia del anclaje principal mediante *hypertags*, se hacen directamente sobre el léxico.

Ejemplo 20 *Por ejemplo, el árbol que modele la comparación, tal que* "Juan es más alto que Pablo"*, puede disponer de la coancla lexical* "que"*, puesto que esta palabra se va a repetir en todas las construcciones.*

■

Tomando como base la gramática GAR factorizada para el español, DyALog crea un analizador sintáctico que se apoya sobre una estrategia de análisis tabular descendiente izquierda-derecha para el castellano: *SPMG Parser*. El proceso de análisis sintáctico se lleva a cabo en las siguientes etapas:

1. **Anclaje léxico-sintáctico:** Selecciona el conjunto de árboles de la gramática GAR para los que la unificación de las *hypertags* de sus anclas y de las palabras del texto de entrada sea exitosa. Una vez finalizada esta etapa, dispondremos de un conjunto de árboles enlazados con las palabras que cumplen funciones sintácticas

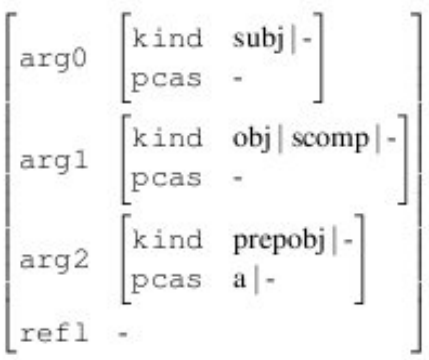


Figura 2.22: *Hypertag* de la palabra *pasear*.

principales de la oración objeto[69]. Sin embargo, palabras con papeles secundarios como determinantes o preposiciones no se enlazarán con ninguna estructura[70].

2. **Construcción de la estructura sintáctica:** A partir de este conjunto de árboles GAR, se intenta construir mediante adjunción y sustitución la estructura sintáctica que cubra el texto a procesar. En el caso ideal, al final de este proceso, se obtiene un único árbol derivado. Éste es el resultado de las combinaciones exitosas de las estructuras elementales del conjunto obtenido en la etapa inicial[71]. En caso de ambigüedad sintáctica[72], esta fase producirá más de un árbol derivado. Cada uno describirá de una forma distinta la estructura completa del texto de entrada.

3. **Decoración de la estructura sintáctica:** El árbol derivado resultante (o árboles derivados) únicamente se encuentra decorado por las anclas de los árboles elementales implicados. Por ello, en esta etapa, se insertan las palabras secundarias del texto objeto en los nodos hoja del árbol final. La decoración se establece en base al éxito o fracaso de la unificación entre las restricciones incluidas en los nodos hoja y las

[69] Por funciones sintácticas principales, se denominan aquellas categorías léxicas que desempeñan un papel relevante dentro de la estructura sintáctica, y por tanto, son anclas de los árboles GAR. Éstas pueden ser, por ejemplo sustantivos como núcleo de un sintagma nominal o verbos como núcleos de la oración.

[70] Salvo que tengan un papel decisivo en una determinada construcción. Por ejemplo, en algunas ocasiones una preposición puede ser el ancla que determina la estructura de un sintagma preposicional. La importancia de una palabra no depende de su categoría léxica sino de qué posición ocupe en la estructura objeto del análisis.

[71] En este proceso de combinación de estructuras elementales, se van descartando aquellos árboles que no es posible encajar en la estructura final.

[72] Un mismo fenómeno sintáctico puede estar modelado por diferentes construcciones de la gramática.

hypertags de las palabras secundarias involucradas, y se lleva a cabo por sustitución [73].

4. **Construcción de la salida:** El análisis sintáctico obtenido por *SPMG Parser*, es devuelto bajo la forma de un *árbol de derivaciones* que detalla las operaciones que se han llevado a cabo, y dónde se han realizado, para construir la estructura sintáctica de la oración objeto de análisis[74]. En el caso de ambigüedad sintáctica, el resultado será ofrecido bajo un *bosque compartido de derivaciones*. Éste recogerá los árboles de derivación de cada una de las estructuras sintácticas aceptadas por el texto analizado. Así, un bosque detalla la estructura común (compartida) a los árboles de derivación resultantes y en ella se indican, para cada caso, las variantes estructurales que introducen[75]. La ambigüedad sintáctica de la estructura ha de ser resuelta en un nivel superior del PLN.

Además, DyALog, permite la construcción de analizadores robustos que pueden ofrecer, en caso de no ser posible un análisis completo, el conjunto de análisis parciales que mejor cubran el texto de entrada (De la Clegerie et al., 2009). Ello significa, que este sistema siempre va a ofrecer al usuario un análisis sintáctico, bien sea completo o parcial.

A mayores, y para mejorar la eficacia del analizador sintáctico en construcción, se le dota de la posibilidad de llevar a cabo un análisis bajo restricciones temporales (*just-in-time*). Al final de un período de tiempo (*timeout*) impuesto, las respuestas encontradas son emitidas incluso si los cálculos no han terminado. Ello permite que el analizador pueda trabajar de forma fluida, sin perder excesivo tiempo en el análisis del texto.

2.3.3. Representación del análisis: *Forest utils*

El recurso *forest utils* heredado de la cadena francesa ALPAGE (Cabrera, 2008), permite tratar la salida del análisis, de tal forma que sea útil y comprensible para los posibles usuarios de la aplicación: un usuario humano o una aplicación de PLN de alto nivel.

Los resultados brutos del análisis son ofrecidos por *SPMG Parser* bajo la forma de un bosque compartido de derivaciones GAR/GIR en un formato interno propio.

Ejemplo 21 *Para la oración* "El niño comió una manzana"*, la salida de*

[73] Se puede ver, y así lo entiende el sistema, como árboles de un sólo nodo cuya ancla es la palabra que va a decorar el nodo hoja objeto de sustitución.

[74] Cada operación GAR recogida, indica el nodo y árboles involucrados.

[75] El formalismo interno empleado por *SPMG Parser* para describir el bosque compartido de derivaciones aparece descrito en el ejemplo 21.

SPMG Parser *y entrada de* forest utils *es la siguiente*[76]*:*

```
Shared Forest

*ANSWER*{answer=> [L = [],N = 5,A = 0]}
   0 <-- [0]1
S{extraction=> extraction[-, adjx], gender=> masc,
inv=> -, mode=> indicative, sat=> +, tense=>
past-historic}(0,5)
   1 <-- #2{ [subject]2 [v]3 | [subject]4 [v]3 }
[object]5 [S]6 7 N2{gender=> masc, hum=> -,
person=> 3, sat=> +, time=> -, wh=> -}(0,2)
   2 <-- [det]8 [nc]9 10
verbose!anchor(comió, 2, 3, 136 V1VMod:agreement
verb_categorization_active_acomp1,
...
tag_anchor{name=> ht{anchor=> comió, arg0=>
arg{extracted=> -, kind=> subj, pcas=> -,
real=> cat[-, CS, N2, PP, S, prel, pri]},
...
imp=> -, refl=> -}, equations=> []})
   3 <--
N2{gender=> masc, person=> 3, sat=> +, time=> -,
wh=> -}(0,2)
   4 <-- [det]8 [adj]11 12
N2{gender=> fem, hum=> -, person=> 3, sat=> +,
time=> -, wh=> -}(3,5)
   5 <-- [det]13 [nc]14 15
S{extraction=> H__1::extraction[-, adjx, cleft,
...
tense=> O__1, wh=> P__1}(5,5)
   6 <-- [Punct]16 17
verbose!struct(136 V1VMod:agreement clsubj:
agreement ante:clitic_sequence
...
imp=> -, refl=> -})
   7 <--
det{def=> +, dem=> -, det=> +, gender=> masc,
numberposs=> -, poss=> -, wh=> -}(0,1)
   8 <-- [det]18 19
verbose!anchor(niño, 1, 2, 69 n:agreement
```

[76]No se muestra toda la información de las *hypertags* presentes en la salida de *SPMG Parser*.

```
nc:agreement cnoun_leaf, nc{def=> +, gender=> masc,
hum=> -, person=> 3, time=> -}, [niño,niño],
tag_anchor{name=> ht{anchor=> niño, arg0=>
arg{extracted=> -, kind=> -, pcas=> -, real=> -},
arg1=> arg{extracted=> -, kind=> -, pcas=> -,
real=> -}, arg2=> arg{extracted=> -, kind=> -,
pcas=> -, real=> -}, cat=> nc, refl=> -},
equations=> []})
   9 <--
verbose!struct(69 n:agreement nc:agreement
cnoun_leaf, ht{anchor=> niño, arg0=> arg{
extracted=> -,
...
cat=> nc, refl=> -})
   10 <--
verbose!anchor(niño, 1, 2, 72 adj_as_cnoun n:
agreement nc:agreement, adj{gender=> masc,
...
equations=> []})
   11 <--
verbose!struct(72 adj_as_cnoun n:agreement
nc:agreement, ht{anchor=> niño, arg0=> arg{
extracted=> -,
...
kind=> -, pcas=> -, real=> -}, cat=> adj, refl=> -})
   12 <--
det{def=> -, dem=> -, det=> +, gender=> fem,
numberposs=> -,
poss=> -, wh=> -}(3,4)
   13 <-- [det]20 21
verbose!anchor(manzana, 4, 5, 69 n:agreement
nc:agreement cnoun_leaf, nc{def=> +, gender=> fem,
...
cat=> nc, refl=> -}, equations=> []})
   14 <--
verbose!struct(69 n:agreement nc:agreement
cnoun_leaf, ht{anchor=> manzana, arg0=> arg{
extracted=> -, kind=> -,
...
pcas=> -, real=> -}, cat=> nc, refl=> -})
   15 <--
end (5,5)
   16 <-- 22
verbose!struct(34 empty_spunct shallow_auxiliary, 34
```

```
empty_spunct shallow_auxiliary)
   17 <--
verbose!anchor(el, 0, 1, 0 det, det{def=> +, dem=> -,
det=> +, gender=> masc, numberposs=> -, poss=> -,
wh=> -}, [el,el], tag_anchor{name=> ht{anchor=> el,
...
real=> -}, cat=> det}, equations=> []})
   18 <--
verbose!struct(0 det, ht{anchor=> el, arg0=>
arg{extracted=> -,
...
real=> -}, cat=> det})
   19 <--
verbose!anchor(una, 3, 4, 0 det, det{def=> -,
dem=> -, det=> +, gender=> fem, numberposs=> -,
poss=> -, wh=> -}, [una,una], tag_anchor{name=>
ht{anchor=> una, arg0=> arg{extracted=> -,
...
kind=> -, pcas=> -, real=> -}, cat=> det},
equations=> []})
   20 <--
verbose!struct(0 det, ht{anchor=> una, arg0=> arg{
extracted=> -, kind=> -, pcas=> -, real=> -},
arg1=> arg{extracted=> -, kind=> -, pcas=> -,
real=> -}, arg2=> arg{extracted=> -, kind=> -,
pcas=> -, real=> -}, cat=> det})
   21 <--
verbose!struct(end, ht{cat=> -})
   22 <--
```

Para comprender mejor el formato interno de SPMG Parser*, nos centramos en el siguiente fragmento simplificado:*

```
    8 <-- [det]18
verbose!anchor(niño, 1, 2, 69 n:agreement
nc:agreement cnoun_leaf, nc{def=> +, gender=> masc,
hum=> -, person=> 3, time=> -}, [niño,niño],
tag_anchor{name=> ht{anchor=> niño, arg0=>
arg{extracted=> -, kind=> -, pcas=> -, real=> -},
arg1=> arg{extracted=> -, kind=> -, pcas=> -,
real=> -}, arg2=> arg{extracted=> -, kind=> -,
pcas=> -, real=> -}, cat=> nc, refl=> -},
equations=> []})
```

```
...

    18 <--
verbose!struct(0 det, ht{anchor=> el, arg0=> arg{
extracted=> -,
...
real=> -}, cat=> det})
```

Aquí se detalla el fragmento de la oración cubierto por el árbol `#69`. *Se trata de una estructura sintáctica perteneciente a la gramática GAR del español y cuya ancla es un sustantivo. Tras la unificación de la* hypertag *presente en el ancla del árbol* `#69`, *con la asignada por* SPMG Lexer *a la unidad léxica* "niño", *este árbol puede ser utilizado para describir parte de la estructura sintáctica del texto objeto de análisis. Concretamente, este árbol va a integrar (bajo operación de sustitución) el determinante* "el", *tal como se indica en la regla* `8 <- [det]18 verbose!anchor (niño ...)`.

Con el formato de la salida de SPMG Parser *se intenta representar el árbol de derivación*[77] *mediante reglas gramaticales. La raíz del árbol de derivación compartido es el 0. A partir de ahí, cada regla representa un fragmento del árbol de derivación, compuesta en su parte izquierda por un símbolo no-terminal que representa el nodo padre, y en su parte derecha por los nodos descendientes. Además de símbolos no-terminales los árboles parciales que forman el árbol de derivación, también tienen elementos terminales en la parte derecha de las reglas gramaticales. Estos símbolos terminales son las anclas de los árboles GAR empleados para analizar la oración objeto. Las anclas (unidades léxicas asociadas a un árbol GAR) son recogidas dentro de la etiqueta* `verbose!anchor`. *A su vez, cada no-terminal numérico puede estar etiquetado lo que resulta útil para determinar el tipo de relación sintáctica existente entre las anclas de dos árboles involucrados en una operación sobre ese nodo*[78]. *Por ejemplo, para el árbol parcial representado por la regla*

```
8 <- [det]18 verbose!anchor (niño ...),
```

se indica que la parte derecha de la regla[79]

[77] O bosque compartido de derivaciones, si existen varios árboles de derivación posibles.

[78] Esa etiqueta será utilizada luego por *forest utils* para describir la dependencia entre anclas de los árboles involucrados en una operación GAR.

[79] En este caso aparece `struct` en lugar de `anchor`. Se trata de una unidad léxica no enlazada a ningún árbol GAR en la etapa del análisis de anclaje entre el léxico y las estructuras sintácticas, sino que se encajará en algún nodo de un árbol GAR como hoja (etapa de decoración). Se puede ver, como una unidad léxica (ancla) asignada a un árbol formado por un único nodo. El sistema lo denota por `struct`.

```
18 <- verbose!struct(0 det, ...),
```

ocupará el lugar del no-terminal numérico `18` *etiquetado con* `[det]` *en la regla de parte izquierda* `8`*. Desde el punto de vista arbóreo, se puede ver que el nodo* `18`*, etiquetado por* `[det]`*, en el árbol*

```
8 <- [det]18 verbose!anchor (niño ...)
```

es el nodo objeto de una operación GAR de sustitución[80] *con el árbol (en este caso, elemento léxico)* `18 <-(verbose!struct(0 det, ...)`*, cuya ancla es el determinante* "el"*. El terminal presente en la regla de parte izquierda* `8` *es el ancla del árbol* `#69`*: el sustantivo* "niño"*. De esta forma la regla resultante*

```
8 <- [det]verbose!struct(0 det, ...) verbose!anchor (niño ...)
```

recoge el modelado concreto del fragmento "El niño" *de la oración* "El niño comió una manzana".

■

Este bosque es posteriormente tratado por el recurso *forest utils* para obtener una salida bajo forma de dependencias (un *bosque compartido de dependencias*). Las anclas de los árboles relacionadas por una operación GAR sobre un determinado nodo de etiqueta L, generan una relación de dependencia con la etiqueta L. Las etiquetas de los nodos de SPMG son generalmente elegidas para reflejar su función gramatical (`subject`, `object`, ...).

Ejemplo 22 *Por ejemplo, si un árbol GAR cuya ancla es un sustantivo es objeto de una sustitución en un nodo etiquetado por* `det`*, por otro árbol cuya ancla es un determinante, se creará la dependencia entre el lema del sustantivo y el lema del determinante etiquetada por* `det` *(con el sentido sustantivo-determinante). Ello aparece indicado en la regla resultante señalada más arriba:*

```
8 <- [det]verbose!struct(0 det, ...) verbose!anchor (niño ...)
```

Donde `[det]` *era la etiqueta del nodo objeto de la operación de sustitución y, posteriormente, se convierte en la etiqueta de la relación de dependencia sintáctica existente entre las unidades léxicas (anclas) de los árboles implicados:* "el" *y* "niño"*. Esta dependencia se puede observar gráficamente en el grafo de dependencias*[81] *de la figura 2.23 entre los lemas* "niño" *(sustantivo) y* "el" *(determinante), representados ambos mediante elipses amarillas.*

■

[80] Si fuese una operación de adjunción el no-terminal numérico estaría precedido por el símbolo #.

[81] Se trata de un grafo dirigido en el cual se intentan representar las relaciones sintácticas existentes entre las palabras presentes en una determinada estructura gramatical. Más adelante se detalla su formato.

Con el objetivo de que el análisis proveído por la cadena en construcción pueda ser utilizado por aplicaciones de PLN de alto nivel, *forest util* representa el bosque de dependencias en el formato XMLDep (Thomasset y De la Clegerie, 2005). Este formato emplea el estándar XML para describir las dependencias sintácticas fruto del análisis del texto de entrada. Los principales elementos presentes en el formato XMLDep son:

- **Nodo (*node*) y Cluster (*cluster*):** Los *clusters* o agrupaciones de nodos representan las formas presentes en la frase objeto de análisis. Dentro de cada *cluster* se encuentran nodos etiquetados con un posible lema de la forma asociada al *cluster* correspondiente. A su vez cada nodo está decorado, entre otras, con la categoría léxica del lema, el número identificador del árbol GAR del cual es ancla esa palabra[82] y un conjunto de derivaciones[83].

- **Arco (*edge*):** Relacionan un nodo origen con un nodo destino y están decoradas por una etiqueta que denota la dependencia sintáctica existente entre ambos. Cada arco está dirigido por subelementos *deriv*[84] que relacionan el nodo origen con el nodo destino.

Básicamente, lo que se intenta representar en formato XML, es un grafo de dependencias formado por nodos (lemas), agrupados en *clusters* (formas), y arcos que describen las dependencias sintácticas entre nodos.

Ejemplo 23 *Para la frase* "El niño comió una manzana"*, el análisis ofrecería los siguientes resultados en formato XMLDep:*

```
<?xml version="1.0" encoding="ISO-8859-1" standalone=
"yes"?>
<dependencies>
  <cluster left="0" right="1" id="E1c_0_1" token="el"
lex="el"/>
  <node cluster="E1c_0_1" tree="0 det" form="el"
lemma="el" xcat="det" cat="det" id="E1n001" deriv=
"E1d000001"/>
  <cluster left="1" right="2" id="E1c_1_2" token=
"niño" lex="niño"/>
```

[82] El número identificador del árbol únicamente aparecerá si la palabra en cuestión es ancla de alguno.

[83] Las derivaciones marcan las dependencias sintácticas entre dos nodos del grafo. Las derivaciones entre dos nodos son operaciones GAR realizadas durante el análisis y, por tanto, establecen una relación entre ambos.

[84] Derivaciones/operaciones GAR entre un nodo origen y otro destino. Entre dos nodos puede haber más de una derivación, ya que las estructuras sintácticas recogidas en la gramática GAR, en ocasiones, se solapan. Ello es provocado por la ambigüedad sintáctica.

```
  <node cluster="E1c_1_2" tree="69 n:agreement nc:
agreement cnoun_leaf" form="niño" lemma="niño"
xcat="N2" cat="nc" id="E1n002" deriv="E1d000000"/>
  <edge source="E1n002" target="E1n001" label="det"
type="subst" id="E1e001">
   <deriv names="E1d000000" source_op="E1o2"
target_op="E1o8"/>
  </edge>
  <node cluster="E1c_1_2" tree="72 adj_as_cnoun n:
agreement nc:agreement" form="niño" lemma="niño"
xcat="N2" cat="adj" id="E1n008" deriv="E1d000008"/>
  <edge source="E1n008" target="E1n001" label="det"
type="subst" id="E1e002">
   <deriv names="E1d000008" source_op="E1o4"
target_op="E1o8"/>
  </edge>
  <cluster left="2" right="3" id="E1c_2_3" token=
"comió" lex="comió"/>
  <node cluster="E1c_2_3" tree="136 V1VMod:
agreement clsubj:agreement ante:clitic_sequence
post:clitic_sequence arg0:collect_real_subject
arg1:real_group_comp
...
arg0:verb_argument_subject verb_canonical
verb_categorization_active_acomp1" form="comió"
lemma="comer" xcat="S" cat="v" id="E1n007"
deriv="E1d000002 E1d000007"/>
  <edge source="E1n007" target="E1n008" label=
"subject" type="subst" id="E1e003">
   <deriv names="E1d000007" source_op="E1o1"
target_op="E1o4"/>
  </edge>
  <edge source="E1n007" target="E1n002" label=
"subject" type="subst" id="E1e004">
   <deriv names="E1d000002" source_op="E1o1"
target_op="E1o2"/>
  </edge>
  <cluster left="3" right="4" id="E1c_3_4"
token="una" lex="una"/>
  <node cluster="E1c_3_4" tree="0 det"
form="una" lemma="una" xcat="det" cat="det"
id="E1n005" deriv="E1d000006"/>
  <cluster left="4" right="5" id="E1c_4_5"
token="manzana" lex="manzana"/>
```

```
  <node cluster="E1c_4_5" tree="69 n:agreement
nc:agreement cnoun_leaf" form="manzana"
lemma="manzana" xcat="N2" cat="nc" id="E1n006"
deriv="E1d000005"/>
  <edge source="E1n006" target="E1n005" label="det"
type="subst" id="E1e005">
   <deriv names="E1d000005" source_op="E1o5"
target_op="E1o13"/>
  </edge>
  <edge source="E1n007" target="E1n006" label=
"object" type="subst" id="E1e006">
   <deriv names="E1d000002 E1d000007" source_op=
"E1o1" target_op="E1o5"/>
  </edge>
  <cluster left="5" right="5" id="E1c_5_5" token=""
lex=""/>
  <node cluster="E1c_5_5" tree="34 empty_spunct
shallow_auxiliary" form="" lemma="" xcat="S" cat="S"
id="E1n004" deriv="E1d000003"/>
  <edge source="E1n007" target="E1n004" label="S"
type="adj" id="E1e007">
   <deriv names="E1d000002 E1d000007" source_op=
"E1o1" target_op="E1o6"/>
  </edge>
  <node cluster="E1c_5_5" tree="end" form="" lemma=""
xcat="end" cat="end" id="E1n003" deriv="E1d000004"/>
  <edge source="E1n004" target="E1n003" label="Punct"
type="subst" id="E1e008">
   <deriv names="E1d000003" source_op="E1o6"
target_op="E1o16"/>
  </edge>
</dependencies>
```

■

Para que la información representada en XML sea comprensible para un usuario humano, es necesario convertirla en un grafo de dependencias (De la Clegerie et al., 2009). Un ejemplo de este tipo de grafo se muestra en el ejemplo 24.

Ejemplo 24 *Siguiendo con el ejemplo 23, el grafo de dependencias correspondiente se muestra en la figura 2.23.*

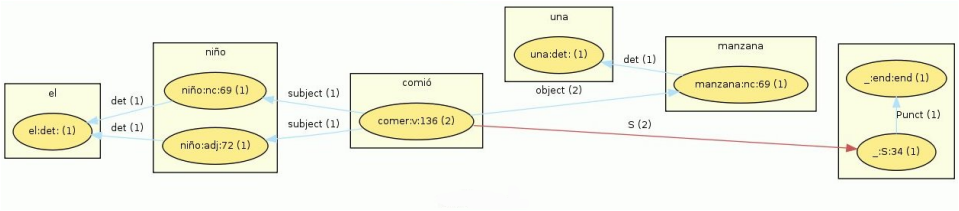


Figura 2.23: Grafo de dependencias para *"El niño comió una manzana."*

Se trata de una representación gráfica del formato XMLDep, donde los nodos se describen mediante elipses amarillas. En cada nodo se indica gráficamente la información recogida en XMLDep, esto es, el lema asociado, su categoría léxica y el número del árbol anclado que representa. Entre paréntesis aparece el número de derivaciones en ese nodo. Los nodos se encuentran incluidos dentro de un cluster[85] *que representan las formas presentes en el texto. Como se ha mencionado anteriormente, una misma forma puede tener diferentes lemas. En este caso concreto, la forma* niño *tiene dos posibles lemas, el sustantivo y el adjetivo. Dado que ambas posibilidades están permitidas por la gramática española, se trata de un caso típico de ambigüedad léxica. Algo similar ocurre con la forma* una*, ya estudiada anteriormente*[86]*. Sin embargo, la gramática diseñada únicamente permite la posibilidad de que la forma* una *sea, en este caso concreto, determinante, eliminando el resto de posibilidades. En el caso de la ambigüedad causada por la forma* niño*, ha de ser solucionada en un nivel superior del PLN, ya que a nivel sintáctico ambas están autorizadas*[87]*.*

Las dependencias sintácticas se representan mediante arcos dirigidos y etiquetados por la función sintáctica correspondiente. Entre paréntesis se indican las derivaciones del nodo que apoyan esa dependencia. Los arcos de color rojo representan una operación de adjunción sobre el árbol indicado en el nodo origen del arco dirigido. Por ejemplo, en la figura 2.23, el árbol `#34` *del nodo* `S` *se ha insertado por adjunción en el árbol* `#136` *del nodo* `comer`[88]*. Los arcos de color azul representan una operación de sustitución sobre el árbol del nodo origen del arco dirigido. Por ejemplo, el árbol* `#69` *del nodo* `manzana` *se ha insertado por sustitución en el árbol* `#136` *del nodo* `comer`*. Otro tipo de arco posible y que no aparece en el grafo de dependencias anterior, es el de color violeta, que hace referencia a las coanclas lexicales.* ■

Además de proveer el análisis realizado por el analizador sintáctico en

[85] Dibujados mediante rectángulos.

[86] La forma *una* puede lematizarse en *una* (determinante) y *unir* (verbo), entre otros.

[87] Es recomendable mostrar todas las posibles construcciones que permita la gramática, ya que cualquiera de ellas puede ser de valor a nivel semántico.

[88] Significa que el lema `comer` es el ancla del árbol `#136`.

los formatos aquí expuestos, el recurso *forest utils* es capaz de ofrecer otro tipo de representaciones e información relevante. Las demás funcionalidades de este recurso serán detalladas en el capítulo 11 del *Manual de Usuario*.

2.3.4. Arquitectura cliente-servidor: *Parserd*, *Callparser* y *WebParser*

La cadena de procesamiento lingüístico desarrollada se encuentra recogida en el servidor de analizadores *Parserd*, también utilizado por ALPAGE (Cabrera, 2008).

Para que el servidor pueda ofrecer el procesamiento lingüístico proveído por la cadena construida a todos aquellos clientes que lo soliciten, es necesario registrarla[89] en el mismo. La cadena desarrollada ejerce en su conjunto, para el servidor, como un analizador morfosintáctico. Se trata de un servidor cuyos analizadores tiene una estructura modular. Esto es, que para registrar un analizador es necesario indicar el analizador léxico (*Lexer*) y el analizador sintáctico (*Parser*) que lo componen; además de otras opciones como la activación o desactivación de la robustez del analizador. El servidor tiene establecido, por defecto, utilizar para todos los analizadores registrados el recurso *forest utils* con objeto de manejar el formato de salida de los procesamientos.

Actualmente, el servidor *Parserd* tiene registrados los analizadores `spmgtel` y `spmgtelr`. En ambos casos, se trata de la cadena implementada formada por *SPMG Lexer* y *SPMG Parser*, pero con la opción de robustez desactivada (`spmgtel`) o activada (`spmgtelr`). Para comunicarse con el servidor de analizadores se dispone de dos clientes[90]:

- **Callparser:** Este cliente usa *telnet* para comunicarse con el servidor *Parserd* a través de la línea de comandos.

- **WebParser:** Se trata de una interfaz *web* destinada principalmente a facilitar la interacción con el usuario humano. Además, este cliente accede al servidor mediante el protocolo HTTP. Ello permite que cualquier usuario humano tenga acceso, de forma sencilla, al sistema de procesamiento lingüístico desde cualquier navegador *web*. Se requiere que este cliente se ejecute en un servidor *web*, como por ejemplo *Apache*[91].

[89] Por registrar la cadena desarrollada se entiende darla de alta en el servidor.

[90] Dependiendo del cliente que utilicemos las funcionalidades que se podrán solicitar al servidor serán distintas, ello se detalla en el *Manual de Usuario*.

[91] El servidor HTTP *Apache2* para *Ubuntu* se detalla en el *Apéndice A*.

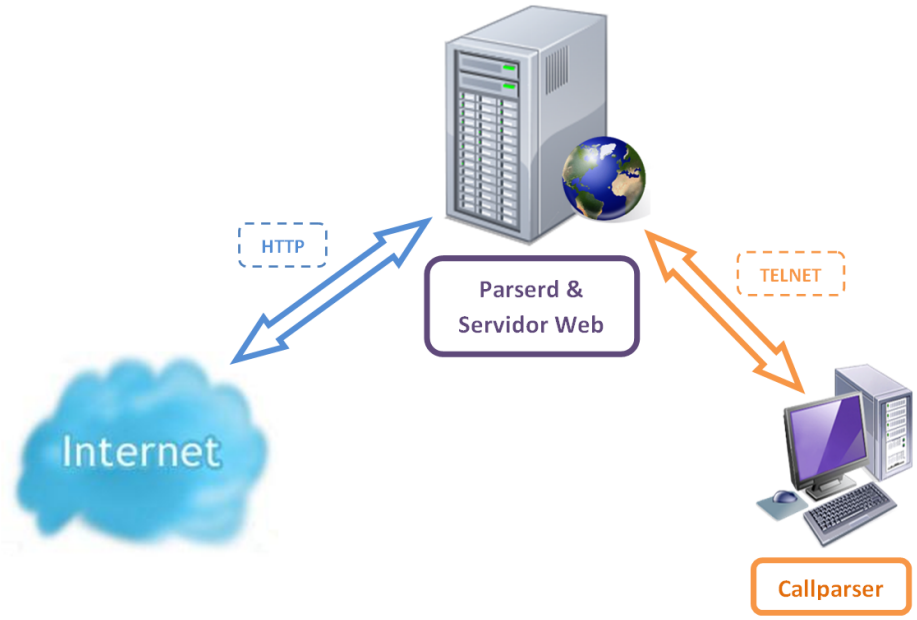


Figura 2.24: Configuración usando un único equipo servidor.

2.4. Entorno de implantación

La herramienta *software* desarrollada permite la posibilidad de implantarse bajo una arquitectura cliente-servidor. Sin embargo, existen varias configuraciones posibles en función del número de máquinas servidoras involucradas:

- **Una única máquina:** El servidor *Parserd* y el servidor *web* se encuentran instalados en el mismo equipo. Asimismo, el cliente *Callparser* se puede encontrar ubicado en la misma máquina o en una diferente. *WebParser* está instalado en el servidor *web*. En la figura 2.24 se describe esta configuración.

- **Dos máquinas:** El servidor *Parserd* está instalado en una máquina diferente de donde se encuentra el servidor *web*. El cliente *WebParser* se está ejecutando en el servidor *web* y el cliente *Callparser* puede estar en cualquiera de los equipos mencionados, o incluso en un tercero. Evidentemente, al servidor *web* se puede acceder desde cualquier punto de Internet bajo el protocolo HTTP. La figura 2.25 muestra como se comunican las diferentes partes del *software* construido.

Sea cual sea la configuración adoptada, *Parserd* ofrece servicio de procesamiento lingüístico a todos aquellos clientes que lo soliciten a través del

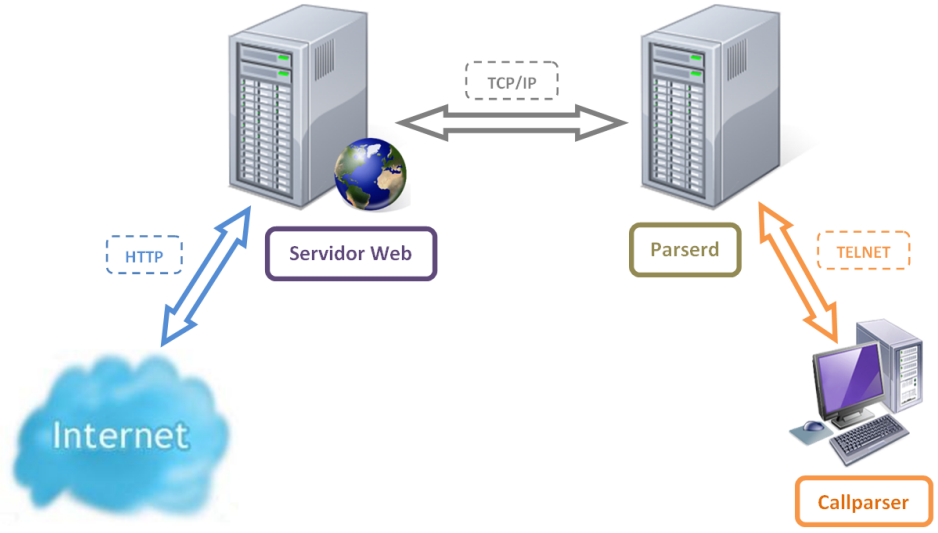


Figura 2.25: Configuración usando dos equipos servidores.

protocolo *telnet* o HTTP. Concretamente, para este proyecto se ha utilizado el servidor *web Apache2*[92] para *Ubuntu*[93].

2.4.1. Entorno hardware

Enumeraremos aquí las necesidades *hardware* del entorno de implantación del sistema:

- **Servidor:** Equipo capaz de ejecutar un sistema operativo Linux o Mac OSX. Esta restricción viene impuesta por el hecho de que el sistema desarrollado no es compatible con cualquier otro sistema operativo. Como es lógico, dependiendo de los parámetros de configuración del equipo, como los MIPS (Millón de Instrucciones Por Segundo) o la cantidad de memoria disponible, se posibilitará la concurrencia de un mayor o menor número de usuarios, así como el tiempo de respuesta.
- **Cliente:** Cualquier equipo capaz de soportar un sistema operativo.

2.4.2. Entorno software

En este subapartado se enumeran las necesidades *software* del entorno de implantación del sistema (Cabrera, 2008).

[92] `http://www.apache.org`
[93] `http://www.ubuntu.com`

- **Servidor:**
  - Perl >= 5.8
  - AppConfig
  - IPC::Run
  - pkg-config 0.15
  - g++
  - perlcc
  - recode
  - xsltproc
  - bison >2.3
  - flex
  - telnet
  - ImageMagick
  - Graphviz
  - Servidor HTTP Apache2 (con el módulo de *Perl* para Apache)
- **Cliente:** El cliente debe permitir el soporte de los ficheros implementados en *Perl* y disponer de los recursos gráficos *Graphviz* e *ImageMagick*[94], en el caso de utilizar el cliente *Callparser*. Para el cliente *web* bastará con un navegador.

[94] Ambos recursos necesarios para manejar los grafos de dependencias.

Capítulo 3

Desarrollo del proyecto

En esta sección se expone la metodología empleada en este proyecto, así como las tecnologías y herramientas utilizadas en su desarrollo.

3.1. Metodología utilizada

Se detallan diversos aspectos relacionados con el desarrollo del proyecto, como son la metodología empleada para el análisis y el diseño, así como el modelo de proceso seguido.

3.1.1. El Lenguaje Unificado de Modelado

El análisis y diseño del sistema se desarrolló utilizando el *Lenguaje Unificado de Modelado* (UML)[1]. Este lenguaje surgió a partir de diversos métodos de análisis y diseño *orientados a objetos* a finales de la década de los 80 y principios de los 90. Unifica los métodos de Rumbaugh, Jacobson y Booch (Booch et al., 2007). Hoy en día, está consolidado como el lenguaje estándar en el análisis y diseño de sistemas. Mediante UML es posible establecer la serie de requerimientos y estructuras necesarias para plasmar un sistema de *software* previo al proceso intensivo de desarrollo de código. A pesar de ser un lenguaje, posee más características visuales que de programación. Cuanto más complejo es el sistema que se desea crear, más beneficios presenta su uso. Esto se debe a dos puntos básicos (Larman, 2002):

1. Mediante una visión global resulta más fácil detectar las dependencias y dificultades implícitas del sistema.
2. Los cambios en una etapa inicial resultan más fáciles de llevar a cabo que en una etapa final del sistema.

A continuación, se exponen algunas de sus características (Larman, 2002):

[1] UML, *Unified Markup Language*.

- El sistema de *software* es diseñado y documentado antes de que sea codificado.
- Los lógicos errores en la etapa de diseño podrán ser detectados con anterioridad. El *software* se comportará de la forma esperada y surgirán menos imprevistos.
- El diseño total del sistema dicta el modo en que se desarrollará el *software*. Las decisiones finales se harán antes de que se encuentre código mal escrito. Con ello se ahorra tiempo de desarrollo.
- Cualquier modificación en el sistema será más fácil de llevar a cabo sobre la documentación UML que sobre el propio código.

En el proceso de desarrollo del proyecto se han utilizado los siguientes diagramas:

- Diagrama de casos de uso: Modela el funcionamiento esperado de la aplicación. Más detalladamente, modela los distintos requisitos funcionales que se esperan de una aplicación y cómo se relacionan entre ellos.
- Diagrama de clases: Es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases y las relaciones entre ellas.
- Diagrama de paquetes: Muestra como un sistema está dividido en agrupaciones lógicas mostrando las dependencias entre esas agrupaciones. Dado que normalmente un paquete está pensado como un directorio, los diagramas de paquetes suministran una descomposición de la jerarquía lógica de un sistema.
- Diagrama de secuencia: Es uno de los diagramas más efectivos para modelar la interacción entre objetos en un sistema. Un diagrama de secuencia muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo.
- Diagrama de actividad: Representa flujos de trabajo paso a paso, tanto de negocio como operacionales, de los componentes del sistema.
- Diagrama de componentes: Un diagrama de componentes representa cómo un sistema de software es dividido en componentes y muestra las dependencias entre estos componentes. Los componentes físicos incluyen archivos, cabeceras, bibliotecas compartidas, módulos, ejecutables, o paquetes.

3.1.1.1. Modelado

A continuación se describen las especificaciones y diagramas realizados en las distintas etapas de desarrollo del sistema:

1. **Análisis:** Descrito en el capítulo 6 del *Manual Técnico*. El primer paso para realizar un buen análisis consiste en capturar y especificar los requisitos. Esta fase, además de ser de vital importancia para el devenir del proyecto, es muy compleja, ya que los usuarios muchas veces no saben claramente lo que quieren. Una vez identificados y priorizados, los requisitos han de ser especificados formalmente mediante los correspondientes *diagramas de casos de uso* y *diagramas de secuencia*. Finalmente, se realizará el *diagrama de clases* inicial del sistema.

2. **Diseño:** En el capítulo 7 del *Manual Técnico* se detalla el diseño del sistema. En una primera etapa se desarrolla un *diagrama de componentes* y un *diagrama de clases*. A diferencia del *diagrama de clases* del análisis, este último aporta un mayor nivel de detalle al modelado del sistema. A continuación se prosigue con los *diagramas de actividad* y *diagramas de secuencia* correspondientes, y que aportan una profundidad mayor que los realizados durante el análisis, siendo más cercanos al sistema real implementado.

3. **Implementación:** En el capítulo de implementación del *Manual Técnico* se describirán varios aspectos y detalles técnicos llevados a cabo para la construcción del sistema. Se aclararán, además, varios aspectos relacionados con los ficheros de configuración.

4. **Pruebas:** Puesta en marcha del sistema implementado y evaluación de su rendimiento.

3.1.2. Ciclo de vida

Todo proyecto de *software* se compone de una serie de actividades. Estas pueden agruparse en fases que contribuyen a la obtención de un producto intermedio, necesario para continuar hacia el producto final y para facilitar su propia gestión. A este conjunto de fases se le denomina *ciclo de vida* o *modelo de proceso*. Sus objetivos son los siguientes:

- Definición de las actividades que se deberán llevar a cabo durante el proyecto.

- Proporcionar unos puntos de control y administrativos, de forma que se pueda evaluar el sistema.

Sin embargo, la forma de agrupar las actividades y los objetivos de cada fase, así como los tipos de productos intermedios que se generan; pueden ser muy diferentes dependiendo del tipo de producto o proceso a generar y de las tecnologías empleadas. En el caso concreto de este trabajo se ha optado por el modelo de proceso de desarrollo conocido con el nombre de *Proceso Unificado de Rational* (RUP)[2].

El RUP es uno de los modelos más generales de los que se utilizan actualmente. Es un proceso de ingeniería planteado por *Kruchten* (Kruchten, 2003), cuyo objetivo es producir *software* de alta calidad. Trata de cumplir con los requerimientos de los usuarios dentro de una planificación y presupuestos establecidos. RUP está dirigido por casos de uso, centrado en la arquitectura, iterativo (mini-proyectos) e incremental (versiones). Toma también en cuenta las mejores prácticas en el modelo de desarrollo de *software*, en particular las siguientes:

- Adaptar el proceso: El proceso deberá adaptarse a las características propias del proyecto. El tamaño del mismo, así como su tipo o las regulaciones que lo condicionen, influirán en su diseño específico. También se deberá tener en cuenta el alcance del proyecto.

- Balancear prioridades: Los requisitos de los diversos participantes pueden ser diferentes, contradictorios o disputarse recursos limitados. Debe encontrarse un balance que satisfaga los requerimientos generales.

- Demostrar valor iterativamente: Los proyectos se entregan, aunque sea internamente, en *etapas iteradas*. En cada una se evalúa la estabilidad y calidad del producto y se va refinando la dirección del proyecto.

- Elevar el nivel de abstracción: Este principio dominante motiva el uso de conceptos reutilizables tales como patrones del *software*, lenguajes 4GL o marcos de referencia (*frameworks*) por nombrar algunos. Esto evita que los ingenieros de *software* vayan directamente de los requisitos a la codificación a medida del cliente, sin saber con certeza qué codificar para satisfacer de la mejor manera los requerimientos y sin comenzar desde un principio pensando en la reutilización del código. Un alto nivel de abstracción también permite discusiones sobre diversos niveles y soluciones arquitectónicas. Éstas se pueden acompañar por las representaciones visuales de la arquitectura, por ejemplo con el lenguaje UML (Booch et al., 2007).

- Enfocarse en la calidad: No sólo la funcionalidad es importante. También lo son el rendimiento y la confiabilidad del sistema. El RUP ayuda a planificar, diseñar, evaluar y ejecutar pruebas que verifiquen

[2]RUP, *Rational Unified Process.*

estas cualidades durante todas las fases del proceso. Asegurar la calidad del producto final debe ser parte del proceso de desarrollo.

3.1.2.1. Ciclos y fases

El ciclo de vida RUP es una implementación del *desarrollo en espiral* (Kruchten, 2003). Divide el proceso de desarrollo en ciclos. Se obtiene un producto al final de cada ciclo. Éstos se organizan en cuatro fases:

Figura 3.1: Ciclo de vida RUP (Kruchten, 2003).

1. Inicio: Debe establecerse una base para poder comparar los gastos reales con los gastos planificados. Si el proyecto no supera este hito, conocido como *hito del objetivo del ciclo de vida*, o bien se cancela, o bien se repite después de ser rediseñado para obtener unos mejores resultados.

2. Elaboración: En esta fase se realiza el *análisis del dominio del problema* y la arquitectura del proyecto obtiene su forma básica sobrepasando el *hito de la arquitectura del ciclo de vida*. Si no se supera este hito, el trabajo aún está a tiempo de ser cancelado o rediseñado. Una vez superada esta fase, el proyecto se convierte en una operación de alto riesgo en la que es mucho más difícil responder a los cambios, por lo que es importante asegurarse de que el análisis realizado cumple con los requisitos del cliente.

3. Construcción: Se centra en el desarrollo de componentes y otros rasgos del sistema que se está diseñando. Es el momento en el que se realiza la mayor parte de la codificación. En proyectos de gran amplitud, se pueden llevar a cabo varias iteraciones con el fin de dividir los casos de uso en trozos más razonables que producen prototipos demostrables. Es aquí donde se produce la primera salida externa del *software*. Su conclusión estará marcada por el *hito de capacidad operacional* inicial.

4. Transmisión: El objetivo es llegar a obtener una versión del producto para su prueba con los usuarios. Una vez instalado surgirán nuevos elementos que implicaran nuevos ciclos. Conlleva la realización de pruebas, entrenamiento de los usuarios y encargados del mantenimiento, así como distribuir el producto.

En cada fase se realizan varias iteraciones en número variable según el proyecto, y en las que se hace un mayor o menor hincapié en las distintas actividades. En la figura 3.2 se muestra cómo varía el esfuerzo asociado a las disciplinas según el momento en el que se encuentre el proyecto RUP.

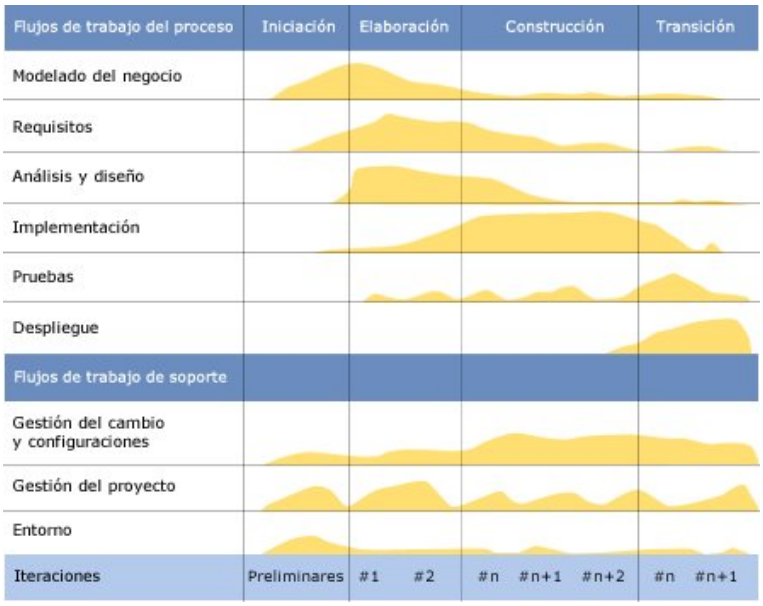


Figura 3.2: Esfuerzo en actividades según la fase del proyecto (Kruchten, 2003).

Durante la *fase de inicio* las iteraciones hacen mayor énfasis en actividades de modelado del negocio y de requerimientos. En la *fase de elaboración*, las iteraciones se orientan al desarrollo de la arquitectura, abarcan más los flujos de trabajo de requerimientos, modelo de negocios (refinamiento), análisis, diseño y una parte de implementación orientado a la arquitectura. En la *fase de construcción*, se lleva a cabo la construcción del producto por medio de una serie de iteraciones. Para cada iteración se selecciona algunos *casos de uso*, se refina su análisis y diseño, y se procede a su implementación y pruebas. Se realiza una pequeña cascada para cada ciclo. Se llevan a cabo iteraciones hasta que se termine la implementación de la nueva versión del producto. En la *fase de transición* se pretende garantizar que se tiene un producto preparado para su entrega a

la comunidad de usuarios. Como se puede observar, en cada fase participan todas las disciplinas; pero, dependiendo de la misma, el esfuerzo dedicado puede variar.

3.2. Plataforma Hardware

Para el desarrollo de este sistema se ha empleado un equipo de sobremesa con la siguiente configuración hardware:

- Procesador Intel Core 2 Quad Q9550 2,83GHz
- 4096Mb de RAM
- Unidad de CD-DVD RW
- 500 Gigabytes de Disco Duro
- Pantalla 17"

3.3. Tecnologías y herramientas empleadas

En el presente apartado se describirán brevemente las tecnologías y las herramientas empleadas para llevar a cabo este proyecto, destacando características representativas de las mismas y dejando patente los motivos que han provocado su elección frente a otras herramientas o técnicas similares.

3.3.1. Tecnologías empleadas

Para comenzar, se detallan las tecnologías que han servido como base de este trabajo.

3.3.1.1. Perl

Perl (Wall et al., 2000) es un lenguaje de programación diseñado por *Larry Wall* en 1987. *Perl* toma características del lenguaje C, del lenguaje interpretado *shell*, AWK, sed, *Lisp* y, en un grado inferior, de muchos otros lenguajes de programación. Fue ampliamente adoptado por su destreza en el procesado de texto y no tener ninguna de las limitaciones de los otros lenguajes de *script*.

El diseño de *Perl* puede ser entendido como una respuesta a tres amplias tendencias de la industria informática: rebaja de los costes en el *hardware*, aumento de los costes laborales y de las mejoras en la tecnología de compiladores. Anteriormente muchos lenguajes de ordenador como *Fortran* y C, fueron diseñados para hacer un uso eficiente de un *hardware* dado.

En contraste, Perl es diseñado para hacer un uso eficiente de los costosos programadores de ordenador. Tiene, además, muchas características que facilitan la tarea del programador a costa de unos requerimientos de CPU y memoria mayores. Éstas incluyen gestión de memoria automática, tipo de dato dinámico, *strings*, listas, tablas de *hash* y expresiones regulares. Todo esto simplifica y facilita todas las formas del análisis sintáctico, manejo de texto y tareas de gestión de datos.

El diseño de *Perl* ha sido muy aleccionado con principios lingüísticos. Ejemplos incluyen la *Codificación Huffman*[3], buena distribución[4] y una larga colección de primitivas del lenguaje. Favorece también las construcciones del lenguaje, tan naturales, como para los humanos son la lectura y la escritura, incluso si eso hace más complicado al intérprete.

Posee igualmente características que soportan una variedad de paradigmas de programación, como la imperativa, funcional y la orientada a objetos. Al mismo tiempo no obliga a seguir ningún paradigma en particular, ni obliga al programador a elegir alguna de ellas.

Hay un amplio sentido de lo práctico en este lenguaje. El prefacio de *Programming Perl* comienza con, *"Perl es un lenguaje para tener tu trabajo terminado"* (Wall et al., 2000). Una consecuencia de esto es que no es un lenguaje ordenado. Incluye características si la gente las usa, tolera excepciones a las reglas y emplea la heurística para resolver ambigüedades sintácticas.

Concretamente, se ha empleado este lenguaje de programación, tanto para el desarrollo de componentes del analizador léxico, *SPMG Lexer*, debido a su gran capacidad para el procesamiento de textos y uso de expresiones regulares, como para la integración de los componentes reutilizados. Además, su uso también posibilitó el desarrollo del cliente *web* del servidor de analizadores, ya que permite desarrollar *scripts* que se ejecutan en un servidor *web*. Por lo tanto, queda patente la polivalencia del lenguaje aquí descrito.

3.3.1.2. CGI

Se trata de una interfaz de entrada común (*Common Gateway Interface*)[5], es una importante tecnología de la *World Wide Web* que permite a un cliente solicitar datos de un programa ejecutado en el servidor *web*. De hecho, especifica un estándar para transferir datos entre el cliente y el programa. Este mecanismo de comunicación entre el servidor *web* y una aplicación externa tiene como resultado final de la ejecución a objetos

[3] Las construcciones más comunes deben ser las más cortas.
[4] La información importante debe ir primero.
[5] `http://www.w3.org/CGI/`

MIME[6].

Las aplicaciones que se ejecutan en el servidor reciben el nombre de CGIs. Éstos fueron una de las primeras maneras prácticas de crear contenidos dinámicos para páginas web. Aquí, el servidor *web* pasa las solicitudes del cliente a un programa externo. Este puede estar escrito en cualquier lenguaje que soporte el servidor, aunque por razones de portabilidad se suelen usar lenguajes de *script*. La salida de dicho programa es enviada al cliente en lugar del archivo estático tradicional. Este tipo de aplicación ha hecho posible la implementación de funciones nuevas y variadas en las páginas *web*, de tal manera que esta interfaz rápidamente se volvió un estándar, siendo implementada en todo tipo de servidores *web*.

En este proyecto se ha utilizado *Perl*, como lenguaje soportado por el servidor *web*, para implementar el cliente *WebParser*. Éste crea una interfaz *web* dinámica accesible desde cualquier navegador.

3.3.1.3. XML

XML[7] es un metalenguaje extensible de etiquetas desarrollado por el *World Wide Web Consortium* (W3C). Permite estructurar la información en un documento o, en general, en cualquier fichero que contenga texto, como por ejemplo ficheros de configuración de un programa o una tabla de datos. Ha ganado mucha popularidad en los últimos años por ser un estándar abierto y libre. A continuación se citan algunas de sus principales características, que han contribuido a convertirlo en un estándar:

- Es extensible, es decir, después de diseñado y puesto en producción, es posible extender XML con la adición de nuevas etiquetas, de modo que se pueda continuar utilizando sin complicación alguna.

- El analizador es un componente estándar, no es necesario crear un analizador específico para cada versión del lenguaje XML. Esto posibilita el empleo de cualquiera de los analizadores disponibles. De esta manera se evitan errores y se acelera el desarrollo de las aplicaciones.

- Si un tercero decide usar un documento creado en XML, es sencillo entender su estructura y procesarla. Mejora, por tanto, la compatibilidad entre aplicaciones.

- La codificación del contenido en XML consigue que la estructura de la información resulte más accesible. Además, la independencia entre

[6](*Multipurpose Internet Mail Extensions*), consiste en una serie de convenciones o especificaciones dirigidas a que se puedan intercambiar a través de Internet todo tipo de archivos (texto, audio, vídeo, etc.) de forma transparente para el usuario.

[7]XML ha sido definido por el W3C. `http://www.w3.org/TR/REC-xml`

el contenido de los datos y la presentación de los mismo hace de éste un formato adecuado para el desarrollo de un sistema como el que presentamos.

- Es un formato ideal para guardar datos de configuración de las aplicaciones.

- Presenta elementos como las DTD's o XMLSchema, que permiten verificar de forma sencilla si la estructura del documento se corresponde con la deseada.

- Existen abundantes herramientas y APIs[8] para trabajar con información representada en este formato. Estas herramientas facilitan el acceso, tratamiento y modificación de los datos. En concreto el lenguaje de programación *Perl*, empleado en la codificación de parte de este proyecto, dispone de varias APIs para tal cometido.

A lo largo de este proyecto, XML ha estado presente en el formato que almacena los resultados de los análisis sintácticos. Se trata de la mejor estrategia para permitir que cualquier sistema haga un uso estandarizado del procesamiento realizado por la cadena en desarrollo.

3.3.1.4. ALEXINA

ALEXINA[9] (Sagot y Danlos, 2008) es un modelo que permite describir información morfológica y sintáctica de manera fácilmente legible, completa, compacta y eficiente.

Su flexibilidad permite representar un gran número de fenómenos a través de un formato sencillo que puede ser usado por herramientas que se basen en diferentes formalismos gramaticales (GLF, GARL) y que requieren información sintáctica detallada para todas las palabras. Los conceptos lingüísticos en los que se basa ALEXINA son compatibles con el estándar *Lexical Markup Framework*[10].

Dadas sus características, se trata de una buena opción para describir el conjunto de palabras que componen una determinada lengua, acompañadas de información morfológica y sintáctica. Por ello se ha utilizado dicho recurso para desarrollar el lexicón de la cadena de procesamiento para el castellano[11].

[8] Una interfaz de programación de aplicaciones o API (*Application Programming Interface*) es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro *software* como una capa de abstracción.

[9] *Atelier pour les LEXiques INformatiques et leur Acquisition, Development of morphological and syntactic NLP lexicons.*

[10] *Lexical Markup Framework*, el estándar ISO/TC37 para léxicos usados en PLN.

[11] Los detalles de ALEXINA se han pormenorizado en el apartado 2.4.1.1.

3.3.1.5. Metagramáticas

Las metagramáticas (De la Clegerie, 2005b) son una de las respuestas aparecidas en la comunidad GAR para hacer frente a los problemas surgidos en el desarrollo de estructuras GAR (Martin, 2006).

Estas estructuras han renovado los métodos de concepción de gramáticas, introduciendo un nivel más abstracto en la descripción de restricciones lingüísticas, reagrupándolas en clases relativamente simples, incluidas a su vez dentro de una jerarquía de herencia múltiple (Thomasset y De la Clegerie, 2005).

El uso de este formalismo es adecuado para hacer frente al desarrollo de una gramática GAR para el español. Sin el mismo, tal cometido se convertiría en una tarea tediosa e imprecisa.

3.3.1.6. LaTeX

LaTeX (Lamport, 1994) es un lenguaje compilado para el procesamiento de textos, formado por un gran conjunto de macros de TeX (Lamport, 1994). Es muy utilizado para la composición de artículos académicos, tesis y libros técnicos, dado que la calidad tipográfica de los documentos es comparable a la de una editorial. Es, además, un *software* libre bajo licencia *Licencia Pública del Proyecto LaTeX* (LPPL). Se ha utilizado para editar esta documentación.

3.3.2. Herramientas empleadas

En esta sección se describen de forma breve las diferentes herramientas utilizadas en este trabajo. Para cada una de ellas se explicará con qué fin se ha empleado en este proyecto y el motivo que ha llevado a su selección por delante de otras herramientas similares.

3.3.2.1. Perl 5.10

Puesto que el desarrollo de parte del sistema e integración de componentes reutilizados se ha llevado a cabo en el lenguaje de programación *Perl*, se hace necesario el uso de un compilador del mismo. *Perl 5.10* (18 de diciembre de 2007)[12] es la versión que refleja la actualización más importante de este popular lenguaje de programación en 5 años y que se construye a partir de la exitosa serie 5.8.x *series*, añadiendo nuevas y potentes características al lenguaje y mejorando el interprete.

Más en concreto, *Perl 5* añadió características para soportar estructuras de datos complejas, funciones de primer orden y un modelo de programación orientada a objetos. Incluye referencias, paquetes y una ejecución de métodos basada en clases y la introducción de variables de ámbito léxico, que hizo más

[12] `http://dev.perl.org/perl5/news/2007/perl-5.10.0.html`

fácil escribir código robusto (junto con el *pragma strict*). Una característica principal es la habilidad de empaquetar código reutilizable como módulos.

Todas las versiones de *Perl* hacen el tipificado automático de datos y la gestión de memoria. El intérprete conoce el tipo y requerimientos de almacenamiento de cada objeto en el programa; reserva y libera espacio para ellos según sea necesario. Las conversiones legales de tipo se hacen de forma automática en tiempo de ejecución. Las ilegales son consideradas errores fatales.

3.3.2.2. Apache2 HTTP Server

El servidor HTTP *Apache*[13] es un servidor *web* de código abierto para plataformas *Unix*[14], *Microsoft Windows*, *Macintosh* y otras, que implementa el protocolo HTTP/1.1 (`http://www.ietf.org/rfc/rfc2616.txt`) y la noción de sitio virtual. Cuando comenzó su desarrollo en 1995 se basó inicialmente en código del popular NCSA HTTPd 1.3, pero más tarde fue reescrito por completo. Su nombre se inspira en el hecho de que *Apache* consistía en su inicio solamente en un conjunto de "parches" a aplicar al servidor de NCSA. Era, en inglés, *a patchy server*, un servidor "parcheado". El servidor *Apache* se desarrolla dentro del proyecto HTTP *Server* (httpd) de la *Apache Software Foundation*[15].

Apache presenta entre otras características altamente configurables, bases de datos de autenticación y negociado de contenido, pero fue criticado por la falta de una interfaz gráfica que ayude en su configuración. Disfruta de una amplia aceptación en la red y es el servidor HTTP más usado. Alcanzó su máxima cuota de mercado en 2005, siendo el servidor empleado en el 70 % de los sitios *web* en el mundo. Sin embargo, ha sufrido un descenso en su cuota de mercado en los últimos años[16].

La arquitectura del servidor es modular y consta de una sección *core* y diversos módulos que aportan la funcionalidad que podría considerarse básica en un servidor *web*. En este caso ha sido necesario instalar *mod_perl*, módulo imprescindible para poder ejecutar programas escritos en *Perl*. Concretamente, para este proyecto se ha utilizado *Apache2*[17].

3.3.2.3. ALEXINA-tools

Se trata del entorno de compilación necesario para poder trabajar con el formalismo lexical ALEXINA (Sagot y Danlos, 2008). En

[13] `http://www.apache.org`

[14] Por ejemplo: BSD, GNU/Linux.

[15] `http://www.apache.org`

[16] Estadísticas históricas y de uso diario proporcionadas por *Netcraft*. `http://news.netcraft.com`

[17] Se trata de la versión del servidor *Apache* para *Ubuntu*. En el *Apéndice A*, se recoge una amplia información acerca de *Apache2*.

concreto, *ALEXINA-tools*[18] es el encargado de transformar la información morfosintáctica, descrita en forma intensional mediante ALEXINA, en su forma extensional correspondiente. El léxico extensional tiene el formato adecuado para ser utilizado por terceras aplicaciones.

Incluye también el lexicalizador *Lexed*[19], distribuido con licencia GPL[20]. Éste ha sido diseñado para construir y consultar bases de información léxica. Permite buscar de forma rápida una entrada en un diccionario en base a cadenas de caracteres. El algoritmo que emplea está basado en el concepto de autómata finito y ofrece una buena alternativa a las tablas *hash* para implementar grandes diccionarios. Concretamente, en nuestro trabajo se ha desarrollado un recurso (LEFFE-SPMG) encargado de la transformación del LEFFE extensional en un lexicón de rápido y fácil acceso, utilizando para ello el recurso *Lexed*.

3.3.2.4. Metagrammar Toolkit

El fin último de este entorno de compilación y ejecución[21] es permitir la construcción de analizadores sintácticos a partir de una metagramática. En el proceso de compilación toman parte las siguientes componentes:

- **smg2xml:** Recurso capaz de organizar la metagramática de partida en formato XML.

- **MGCOMP:** Se trata de un compilador de metagramáticas. Su función es la de obtener una gramática GAR a partir de una descripción abstracta de las estructuras sintácticas permitidas en un determinado lenguaje. A partir de la metagramática representada en XML obtiene una gramática GAR en XML.

- **tag_converter:** Recurso necesario para convertir la gramática GAR (en formato XML) en el formato GAR de entrada del compilador DyALog (De la Clegerie, 2005a).

- **DyALog:** Es un entorno de compilación y ejecución de programas lógicos y analizadores sintácticos profundos, tabulares y robustos, utilizando como fuente varios formalismos gramaticales (DCG, GAR, GIR, RCG). En nuestro caso concreto, DyALog parte de la gramática GAR, fruto del trabajo de MGCOMP (De la Clegerie, 2005b), para construir un analizador sintáctico híbrido GAR/GIR (De la Clegerie, 1999).

[18] `https://gforge.inria.fr/projects/alexina/`
[19] `http://www.labri.fr/perso/clement/lexed/`
[20] GPL, *General Public License*.
[21] `http://mgkit.gforge.inria.fr`

La figura 3.3 resume el proceso de compilación de la metagramática SPMG en el analizador sintáctico *SPMG Parser*. Una vez diseñada la metagramática para el español es imprescindible disponer de los recursos de *Metagrammar Toolkit*[22] para construir el analizador sintáctico de la cadena. La razón de la selección de esta herramienta es simple y determinante.

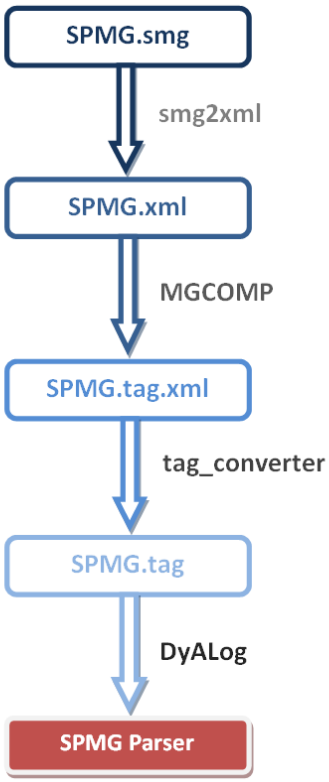


Figura 3.3: Proceso de compilación de SPMG en *SPMG Parser*.

3.3.2.5. Visual Paradigm Community Edition

Visual Paradigm[23] es una herramienta CASE[24] que utiliza UML como lenguaje de modelado. La herramienta está diseñada para una amplia gama de usuarios interesados en construir sistemas de *software* fiables con el uso del paradigma orientado a objetos, incluyendo actividades como ingeniería de *software*, análisis de sistemas y análisis de negocios. Esta herramienta ha sido utilizada en el desarrollo de este proyecto para la elaboración del análisis y del diseño.

[22] `http://mgkit.gforge.inria.fr`
[23] `http://www.visual-paradigm.com`
[24] *Computer Aided Software Engineering*, Ingeniería de Software Asistida por Ordenador.

3.3.2.6. Kile

Kile[25] es un editor de TeX/LaTeX (Lamport, 1994). Funciona conjuntamente con KDE[26] en varios sistemas operativos. Las principales características por las que se ha seleccionado *Kile* frente a otros editores son las siguientes:

- Autocompletado de comandos LaTeX.
- Coloreado de sintaxis. *Kile* automáticamente marca los comandos LaTeX y resalta los paréntesis.
- Permite trabajar con múltiples ficheros simultáneamente.
- Proporciona plantillas y patrones para facilitar la creación de documentos.
- Plegado de código.

3.3.2.7. Planner

Planner es una herramienta para la gestión de proyectos que permite, por ejemplo, realizar *diagramas de Gantt*[27] o la asignación de recursos. Forma parte del proyecto GNOME[28] y se puede instalar desde los repositorios de *Ubuntu*.

3.3.2.8. Ubuntu 9.04

Ubuntu[29] es una de las más importantes distribuciones GNU/Linux que ofrece un sistema operativo principalmente enfocado a computadoras personales. Se basa en *Debian*[30] y concentra su objetivo en la facilidad y libertad de uso, la fluida instalación y los lanzamientos regulares. El patrocinador es *Canonical Ltd.*, una empresa privada fundada y financiada por el empresario sudafricano *Mark Shuttleworth*. La versión utilizada en este proyecto es la 9.04 lanzada en abril de 2009.

[25] `http://kile.sourceforge.net`

[26] KDE es un proyecto de *software* libre para la creación de un entorno de escritorio e infraestructura de desarrollo para diversos sistemas operativos como GNU/Linux, Mac OS X, Windows, entre otros. `http://www.kde.org`

[27] El *diagrama de Gantt*, *gráfica de Gantt* o *carta Gantt* es una popular herramienta gráfica cuyo objetivo es mostrar el tiempo de dedicación previsto para diferentes tareas o actividades a lo largo de un tiempo total determinado.

[28] GNOME es un entorno de escritorio e infraestructura de desarrollo para sistemas operativos Unix y derivados Unix como GNU/Linux, BSD o Solaris; compuesto enteramente de software libre. `http://www.es.gnome.org`

[29] `http://www.ubuntu.com`

[30] `http://www.debian.org`

Capítulo 4

Planificación y presupuesto

4.1. Planificación del proyecto

A continuación se muestran las dos planificaciones realizadas para este proyecto. La primera de ellas es la prevista y fue realizada antes del inicio del desarrollo del sistema. La segunda es la real que se realizó al finalizar el proyecto para poder observar las desviaciones producidas y las causas de las mismas. Finalmente se detallan las justificaciones de las desviaciones de la planificación inicial acaecidas a lo largo de este trabajo.

4.1.1. Planificación inicial

Las fases planificadas para el desarrollo del proyecto fueron las siguientes:

1. **Estudios previos:** Será necesario familiarizarse con numerosas tecnologías novedosas. El sistema objeto de este proyecto se enmarca dentro del campo de la investigación aplicada. Como tal, para su desarrollo se recurren a herramientas innovadoras cuyo aprendizaje consume una gran parte del tiempo; bien por falta de documentación que las describa de forma detallada, o bien debido a la complejidad que entrañan. Además, también será necesaria una profunda inmersión en textos científicos referentes a la temática tratada. Por lo tanto, esta etapa será fundamental y servirá como base para las siguientes fases del desarrollo.

2. **Análisis y diseño del sistema:** Se realiza la especificación de requisitos, los casos de uso, el diagrama de clases inicial y los diagramas de secuencia con el fin de obtener una descripción detallada del sistema. Se profundizan los conceptos del análisis, detallando los diagramas de secuencia y refinando los diagramas de clases.

3. **Implementación:** Se incluye la implementación completa del sistema.

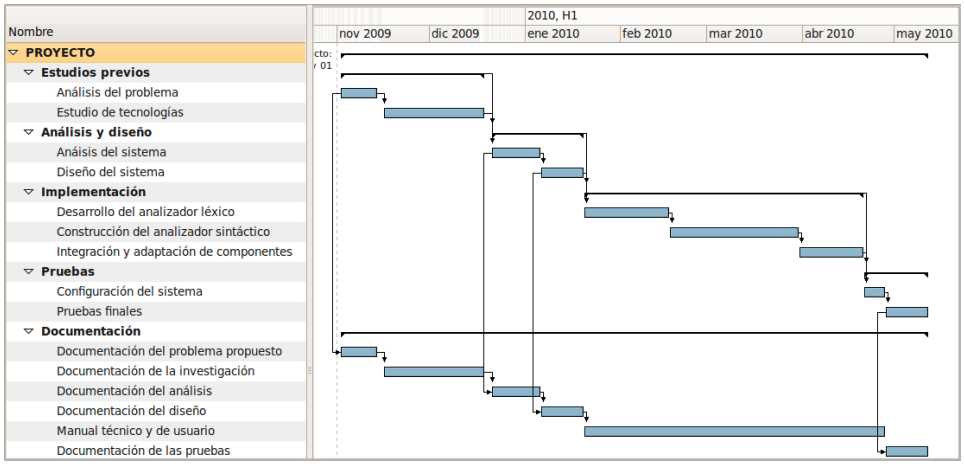


Figura 4.1: Diagrama de *Gantt* de la planificación inicial.

4. **Pruebas:** Se realizan las pruebas finales del sistema, con el objetivo de corregir los posibles errores cometidos durante la implementación. Además, se evaluará la capacidad de procesamiento lingüístico del sistema mediante una serie de *corpora* diseñados para tal fin.

5. **Documentación:** Se elaborará a lo largo de todas las fases del proyecto y fruto de ello verán la luz varios manuales y documentos.

Para la estimación temporal se suponen los siguientes datos:

- El inicio del proyecto se produce el 2 de noviembre de 2009.
- Se estima una dedicación al proyecto de 8 horas diarias, 40 horas semanales, aproximadamente 20 días por mes.
- Se destinarían aproximadamente 3 horas al día para la documentación, lo que supone una dedicación de un 35 % del tiempo diario para esta tarea.
- El final del proyecto se situaría el 11 de mayo de 2010.

Se muestra la planificación temporal mediante el diagrama de *Gantt* de la figura 4.1 y la tabla de tiempos de la figura 4.2. Esta última detalla las fechas de comienzo y fin de cada tarea.

4.1.2. Planificación real

En este apartado se va a mostrar de forma detallada el tiempo empleado realmente en la realización del proyecto. Se muestra en la figura 4.3 el

WBS	Nombre	Inicio	Fin	Duración
1	**PROYECTO**	**nov 2**	**may 11**	**137d**
1.1	**Estudios previos**	**nov 2**	**dic 18**	**35d**
1.1.1	Análisis del problema	nov 2	nov 13	10d
1.1.2	Estudio de tecnologías	nov 16	dic 18	25d
1.2	**Análisis y diseño**	**dic 21**	**ene 19**	**22d**
1.2.1	Anáisis del sistema	dic 21	ene 5	12d
1.2.2	Diseño del sistema	ene 6	ene 19	10d
1.3	**Implementación**	**ene 20**	**abr 20**	**65d**
1.3.1	Desarrollo del analizador léxico	ene 20	feb 16	20d
1.3.2	Construcción del analizador sintáctico	feb 17	mar 30	30d
1.3.3	Integración y adaptación de componentes	mar 31	abr 20	15d
1.4	**Pruebas**	**abr 21**	**may 11**	**15d**
1.4.1	Configuración del sistema	abr 21	abr 27	5d
1.4.2	Pruebas finales	abr 28	may 11	10d
1.5	**Documentación**	**nov 2**	**may 11**	**137d**
1.5.1	Documentación del problema propuesto	nov 2	nov 13	10d
1.5.2	Documentación de la investigación	nov 16	dic 18	25d
1.5.3	Documentación del análisis	dic 21	ene 5	12d
1.5.4	Documentación del diseño	ene 6	ene 19	10d
1.5.5	Manual técnico y de usuario	ene 20	abr 27	70d
1.5.6	Documentación de las pruebas	abr 28	may 11	10d

Figura 4.2: Tabla de tiempos de la planificación inicial.

diagrama de *Gantt* con la información referente a la planificación real del mismo y en la figura 4.4 las fechas de comienzo y fin de cada tarea. El proyecto se ha finalizado el día 18 de mayo de 2010.

4.1.3. Justificación de desviaciones temporales

En este apartado se van a justificar o proporcionar las razones que han ocasionado las desviaciones respecto de la planificación inicial. La finalización del proyecto estaba prevista para el 11 de mayo de 2010. No obstante, el presente trabajo no se concluyó hasta el 18 de mayo de 2010. Se puede observar que se han mantenido las fases de desarrollo previstas inicialmente. Sin embargo, se produjeron ciertas desviaciones temporales que, a pesar de resultar relevantes en algunas tareas, no han logrado retrasar significativamente el proyecto. Ello fue debido a que el excesivo tiempo empleado en algunas tareas ha sido contrarrestado por el rápido cumplimiento de otras. Las razones a las que se pueden atribuir las diferencias de la planificación real con respecto a la inicial son las siguientes:

- La principal causa de que el proyecto únicamente finalizase 5 días laborables (7 días naturales) después de lo estimado previamente, es que se trata de una planificación inicial pesimista. La ventaja de esta estrategia, es que si ocurre un retraso en una tarea crítica, hay

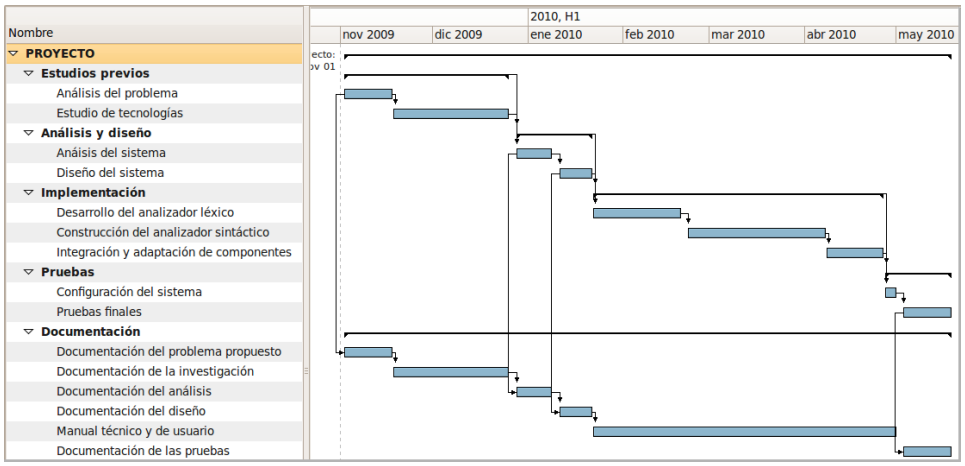


Figura 4.3: Diagrama de *Gantt* de la planificación real.

WBS	Nombre	Inicio	Fin	Duración
1	**PROYECTO**	**nov 2**	**may 18**	**142d**
1.1	**Estudios previos**	**nov 2**	**dic 25**	**40d**
1.1.1	Análisis del problema	nov 2	nov 17	12d
1.1.2	Estudio de tecnologías	nov 18	dic 25	28d
1.2	**Análisis y diseño**	**dic 28**	**ene 21**	**19d**
1.2.1	Anáisis del sistema	dic 28	ene 8	10d
1.2.2	Diseño del sistema	ene 11	ene 21	9d
1.3	**Implementación**	**ene 22**	**abr 26**	**67d**
1.3.1	Desarrollo del analizador léxico	ene 22	feb 19	21d
1.3.2	Construcción del analizador sintáctico	feb 22	abr 7	33d
1.3.3	Integración y adaptación de componentes	abr 8	abr 26	13d
1.4	**Pruebas**	**abr 27**	**may 18**	**16d**
1.4.1	Configuración del sistema	abr 27	abr 30	4d
1.4.2	Pruebas finales	may 3	may 18	12d
1.5	**Documentación**	**nov 2**	**may 18**	**142d**
1.5.1	Documentación del problema propuesto	nov 2	nov 17	12d
1.5.2	Documentación de la investigación	nov 18	dic 25	28d
1.5.3	Documentación del análisis	dic 28	ene 8	10d
1.5.4	Documentación del diseño	ene 11	ene 21	9d
1.5.5	Manual técnico y de usuario	ene 22	abr 30	71d
1.5.6	Documentación de las pruebas	may 3	may 18	12d

Figura 4.4: Tabla de tiempos de la planificación real.

más posibilidades de que este retraso no implique una demora en el proyecto.

- Sin embargo, mientras que en algunas de las tareas que componen las fases del proyecto la estimación temporal pesimista ha evitado el retraso del final del mismo, en otras, no ha sido suficiente el tiempo asignado para tal cometido.

- Concretamente, las tareas dedicadas al análisis del problema y al estudio de las tecnologías, así como la documentación de las mismas, ha superado el excesivo tiempo asignado inicialmente. Las razones del error en la planificación de estas tareas es la ausencia de experiencia personal en proyectos enmarcados dentro de la investigación. Aquí es necesario un estudio previo que consume una gran cantidad de tiempo, durante el cual no siempre se dispone de la documentación suficiente o, por el contrario, se tiene a disposición una gran cantidad de artículos cuya diversidad no permite obtener de forma precisa lo que realmente se está buscando.

- Otra posible causa del retraso producido durante la etapa de estudios previos, ha sido la necesidad de realizar una estancia de 1 mes y 20 días en el seno del equipo ALPAGE en París, Francia. La finalidad de dicho viaje ha sido el aprendizaje de los formalismos y herramientas necesarias para llevar a cabo este proyecto.

- El leve retraso ocasionado en la implementación del proyecto, a pesar de la estimación pesimista, se debe principalmente a que, dentro del ámbito de la investigación en el que se encuentra el presente proyecto, los recursos utilizados no siempre son todo lo estables o fiables que se requiere para avanzar con seguridad. Se trata de tecnologías innovadoras, en continuo desarrollo y en plena etapa experimental, y cuyos resultados y efectos aún no son del todo conocidos.

- A pesar de no exceder el tiempo asignado, la integración de los componentes reutilizados ha sido más compleja de lo esperado. Recursos, como el cliente *WebParser*, parcialmente implementado, obligaron a prácticamente consumir el tiempo destinado a tareas de integración.

- Sin embargo, la planificación pesimista ha sido útil en las tareas de análisis y diseño del sistema, así como en su documentación. Ello se debe a que la cadena ALPAGE ha servido como guía para diseñar la cadena para el español, facilitando y reduciendo el tiempo empleado en estas tareas.

- La causa que ha ocasionado el incremento del tiempo estimado para la realización de pruebas sobre los *corpora*, se debió básicamente a la

gran cantidad de tiempo empleado por el sistema desarrollado para el procesamiento de los mismos.

4.2. Presupuesto del proyecto

En este apartado se detalla el presupuesto necesario para llevar a cabo el proyecto. Para ello se han separado los costes físicos, tanto de *hardware* como de *software*, de los costes referentes a los recursos humanos.

4.2.1. Recursos físicos

Los recursos materiales empleados en este trabajo se clasifican en *hardware* y *software*.

4.2.1.1. Recursos *hardware*

El *hardware* está amortizado a diez proyectos, como regla general en el desarrollo de aplicaciones.

Hardware	**Coste**
Equipo de sobremesa con procesador Intel Core 2 Quad; 2,83 GHz; 4 Gb de RAM; HDD 500 Gb	1000 €
Total (coste/10)	100 €

Tabla 4.1: Coste de recursos *hardware*.

4.2.1.2. Recursos *software*

El hecho de que se haya utilizado exclusivamente *software* libre en el desarrollo del proyecto provoca coste cero en este apartado:

- Sistema operativo GNU/Linux. Distribución *Ubuntu* 9.04.
- *Perl 5.10*, para desarrollar el analizador léxico e integrar los componentes reutilizados.
- *Apache2*, servidor web HTTP de código abierto para *Ubuntu* (GNU/Linux)
- *ALEXINA-tools*, para mejorar el lexicón morfosintáctico.
- *Metagrammar Toolkit*, para construir el analizador sintáctico.
- Herramienta para el modelado: *Visual Paradigm Community Edition* para *Ubuntu*, versión libre disponible para alumnos de la *Escuela Superior de Ingeniería Informática* (ESEI) de la *Universidad de Vigo*.

- Editor *Kile* para llevar a cabo la documentación en LaTeX.
- Gestor de proyectos *Planner* para la realización de los diagramas de *Gantt* necesarios.

4.2.2. Recursos humanos

Si bien el desarrollador de este proyecto ha sido una sola persona, dependiendo de la tarea realizada genera costes diferentes.

Recursos Humanos	**Duración**	**Coste**
Analista (40 €/hora)	50 días X 8 horas/día	16000 €
Diseñador (35 €/hora)	9 días X 8 horas/día	2520 €
Programador (30 €/hora)	83 días X 8 horas/día	19920 €
Total		**38440 €**

Tabla 4.2: Coste de recursos humanos.

4.2.3. Coste final

Para hallar el coste final del desarrollo del proyecto, se han sumado los costes relativos a los recursos humanos a los costes relativos a los recursos físicos.

Recursos físicos	100 €
Recursos humanos	38440 €
Total	**38540 €**

Tabla 4.3: Coste final del proyecto.

Capítulo 5

Problemas, conclusiones y posibles ampliaciones

En este capítulo se describen las dificultades encontradas a lo largo del desarrollo de este trabajo, las conclusiones personales obtenidas una vez concluida la realización del proyecto, así como las posibles ampliaciones que se pueden aplicar sobre el sistema final construido.

5.1. Dificultades encontradas

A lo largo del desarrollo se han ido encontrando dificultades que se han superado con éxito. Los principales problemas surgieron durante la fase de estudio y durante la de implementación. A continuación se detallan cuales han sido los más relevantes y como se han solucionado:

- En primer lugar, destacar el marco de investigación que ha rodeado este proyecto. La falta de experiencia personal en este campo, junto con la extrema dificultad del tema tratado ha conseguido elevar considerablemente la complejidad de este trabajo. Ello ha conllevado un esfuerzo significativo para estar a la altura de la situación.
- Asimismo, la ausencia de experiencia dentro del campo del PLN ha incrementado el nivel de complejidad durante los estudios previos. La solución a este contratiempo pasó por profundizar en los conceptos de PLN necesarios para empeñar la tarea encomendada.
- Los formalismos y herramientas necesarios para llevar a cabo el presente proyecto estaban descritos en una escasa cantidad de artículos en francés. Por consiguiente, la comprensión de los mismos se ha convertido en una tarea ardua y tediosa. A su vez, la información aportada era, en no pocas ocasiones, insuficiente para poder utilizar estas nuevas tecnologías en todo su potencial.

- Concretamente, el formalismo de las metagramáticas[1] carecía de documentación que lo detallase de forma adecuada y suficiente para describir la gramática española. La dificultad de este lenguaje y la necesidad de conocer y manejar el mismo para llevar a cabo el presente proyecto, fue condición suficiente para realizar una estancia de 1 mes y 20 días en el seno del equipo ALPAGE, en el INRIA[2] de París.

- Otra dificultad importante, intrínseca al campo de la investigación, es la inestabilidad y la falta de fiabilidad de las innovadoras herramientas y tecnologías empleadas. Las mismas se encontraban en continuo estado de desarrollo, lo que ocasionaba problemas de compatibilidad y de funcionalidad. La solución a los mismos, ha sido el aislamiento de versiones locales, esto es, fijar una versión de las herramientas en la máquina local para evitar problemas futuros de compatibilidad. Ello permitía un estado local estable mientras las tecnologías desarrolladas por el equipo francés mantenían su progreso[3].

- La cadena de procesamiento lingüístico ALPAGE carecía de la suficiente documentación. La estancia en el INRIA ha subsanado este problema.

- Desarrollar una gramática para el castellano implica un conocimiento profundo de las estructuras sintácticas permitidas para este idioma concreto. Obviamente, un ingeniero informático no dispone de tales conocimientos. Ello ha implicado un aumento en el tiempo de estudios previos y ocasionado, además, problemas en la etapa de implementación de la metagramática para el español.

- Dificultad a la hora de encontrar recursos, por ejemplo *corpora*, para un idioma tan poco explotado en este campo como es el español. Únicamente se ha podido hacer uso de dos *corpora* en común con la cadena ALPAGE, objeto de la comparación.

- Durante las pruebas de procesamiento lingüístico, se ha producido una falta considerable de recursos. La complejidad y envergadura de los *corpora Europal 96* y *Europal 97*, puso de manifiesto las carencias de los recursos *hardware* y *software* destinados a este proyecto en un principio. Ello provocó que tan solo fuese posible procesar un reducido fragmento de estos *corpora*.

- La integración de los diferentes componentes reutilizados no ha sido trivial en muchos casos. Concretamente, el cliente *WebParser*

[1] Formalismo necesario para la construcción del analizador sintáctico.

[2] *Institut National de Recherche en Informatique et en Automatique.* `www.inria.fr`

[3] Evidentemente, es necesario, al cabo de un período de tiempo, comprobar las nuevas funcionalidades aportadas por las nuevas versiones de las herramientas y tecnologías, y actualizar la versión local.

únicamente se encontraba parcialmente implementado y no funcionaba de forma correcta. Ello obligó a realizar trabajos de implementación sobre el mismo.

- Se ha logrado integrar el subsistema *SxPipe* a la cadena desarrollada. Sin embargo, no ha sido posible adaptarlo al 100 % al castellano. El componente *SxSpell* encargado de la corrección ortográfica no es capaz de trabajar sobre el idioma objeto de este proyecto.

5.2. Conclusiones

Una vez finalizado el proyecto se extraen una serie de conclusiones. En primer lugar, es necesario destacar el hecho de haber cumplido los objetivos establecidos en un principio. Se ha logrado construir a lo largo de este proyecto, una cadena de procesamiento lingüístico para el español que ofrece unos resultados esperanzadores. Se ha conseguido una buena integración de los componentes reutilizados y un incremento de la información léxica de partida. Se trata de una mejora sensible para el futuro desarrollo de aplicaciones PLN en nuestro país.

Las pruebas[4] de procesamiento lingüístico llevadas a cabo empleando el sistema desarrollado han ofrecido unos buenos resultados, tanto de cobertura como de *tasa de ambigüedad*. En comparación con la cadena ALPAGE sobre el idioma francés, los resultados son ligeramente inferiores respecto a la cobertura. Sin embargo, respecto a la tasa de ambigüedad existen unas diferencias significativas entre ambas cadenas. Ello se debe, a la inmadurez del léxico y metagramática del castellano, así como a que el idioma español presenta una mayor ambigüedad sintáctica. El idioma francés tiene un nivel de rigidez mayor. Por ejemplo, la estructura sintáctica canónica[5] de una oración en francés no contempla la no existencia del sujeto, salvo en modo imperativo o impersonal. Mientras, que en el caso del castellano, el verbo en forma canónica puede no llevar sujeto de forma explícita, y encontrarse éste omitido. Este fenómeno, a parte de muchos otros, incrementa la tasa de ambigüedad del sistema desarrollado frente a la cadena ALPAGE. Tras la comparación se puede señalar que la cadena española aún se encuentra en una etapa inicial de su desarrollo, mientras que el analizador francés se fundamenta en una metagramática más especializada y eficiente.

Por otra parte, se ha observado que a pesar de la complejidad y tamaño del *corpus* del Real Jardín Botánico[6], éste presenta una tasa de ambigüedad inferior. Esto se debe a que, al primar las descripciones con grandes enumeraciones de sintagmas nominales no entran en juego los árboles

[4] Las pruebas realizadas están detalladas en el capítulo 9.3 del *Manual Técnico*.
[5] La forma canónica de una oración, es la formada por `Sujeto+Verbo+Objeto`.
[6] *Corpus* objeto de pruebas.

verbales presentes en la gramática GAR. Éstos recogen las estructuras sintácticas más complejas, arrojando al análisis una mayor ambigüedad. De ahí que su ausencia durante el procesamiento provoque una reducción en la tasa de ambigüedad.

También existen diferencias significativas entre ambos sistemas de PLN, en lo que respecta al tiempo de ejecución medio por cada oración analizada. Esto se debe principalmente, a que el equipo francés contaba con unos recursos *hardware* superiores a los empleados para las pruebas realizadas sobre el sistema objeto de este proyecto.

Frente a la ambigüedad existente en los análisis realizados, se ha decidido no aplicar ningún filtro sobre los mismos. De esta forma, se le ofrecen al usuario del sistema todas las posibles estructuras sintácticas que cubren el texto de entrada, permitiéndole decidir cuál es la adecuada. La justificación de esta medida es que dependiendo del contexto la estructura sintáctica puede variar. Para evitar la eliminación de posibles interpretaciones del lenguaje natural, ha de ser un nivel superior de PLN el encargado de desambiguarlo, no el nivel sintáctico actual. Además, hemos de señalar que el sistema se encuentra en una etapa inicial y que el futuro trabajo debería mejorar prestaciones y resultados.

Salientar igualmente la importancia de la reutilización en este trabajo. El proyecto[7] que subvencionó el trabajo promulga la reutilización de componentes costosos de PLN. Esta máxima se ha convertido en un pilar básico de este proyecto. Sería prácticamente imposible desarrollar una herramienta de las características que presenta el *software* implementado sin la reutilización de herramientas, tecnologías y componentes ya existentes. Implementar la cadena ALPAGE ha consumido una cantidad de tiempo, recursos y personal considerablemente superiores a los empleados en la cadena española.

Destacar la modularidad de la cadena desarrollada. Ello facilitará su uso en diferentes contextos. Si en un futuro fuese necesario aplicar la cadena sobre textos de temática bursátil, podría sustituirse *SPMG Lexer* por un analizador morfológico centrado en su léxico específico. Asimismo, al trabajar sobre textos botánicos, podría ser recomendable aplicar otro analizador sintáctico, diferente de *SPMG Parser* u otra versión específica del mismo, que sea capaz de manejar sintagmas nominales con enumeraciones considerablemente largas[8].

Resaltar también la importancia de los diferentes ficheros de configuración detallados en el capítulo 8 del *Manual Técnico*. Particularmente, el fichero `restrictions.txt`. Este fichero maneja la importante tarea de

[7] Proyecto de investigación *Búsqueda de respuestas empleando metagramáticas* P.P. M709122P 6406211.

[8] Los textos botánicos suelen contener amplias descripciones con largas enumeraciones sin apenas verbos.

reducir la ambigüedad, tanto léxica como sintáctica. Por ejemplo, si se está trabajando dentro del contexto de la botánica, donde la forma *"hierbas"* como sustantivo aparece un gran número de veces. Puede interesar, para reducir el nivel de ambigüedad, filtrar la forma *"hierbas"* como segunda persona singular del verbo *"herbar"*. Ello va a evitar que el analizador sintáctico eche mano de los árboles verbales de la gramática GAR, lo que implicaría un incremento de la ambigüedad y complejidad del análisis realizado.

Durante el desarrollo de este proyecto, la cadena construida ha sido integrada de forma exitosa bajo una aplicación destinada a la *recuperación de información*. Se ha conseguido que está aplicación de alto nivel de PLN, fuese capaz de llevar a cabo un análisis semántico, a partir de los resultados obtenidos por la cadena de procesamiento lingüístico del español sobre los *corpora* proporcionados por el Real Jardín Botánico y el Museo de Ciencias Naturales de Madrid.

Por otra parte, a nivel personal, el presente trabajo ha significado una oportunidad única para adentrarse en el mundo de la investigación y, más concretamente, en el campo del PLN. Los conocimientos recibidos a lo largo de este proyecto han sido de gran importancia y valor para la futura labor investigadora. Destacar la gran cantidad de dificultades encontradas durante el desarrollo de este proyecto. Solucionar todas y cada una de ellas me ha aportado una gran experiencia profesional, necesaria para hacer frente a problemas presentes en futuros desarrollos. La estancia en el extranjero, así como la lectura de artículos en otros idiomas, como pueden ser, el francés y el inglés, ha mejorado mi dominio en cada uno de ellos. Además, mi trabajo me ha permitido el aprendizaje del lenguaje de programación *Perl* y, de todas sus ventajas y potencial, desconocidas hasta el momento. También, ha propiciado el conocimiento del lenguaje de edición LaTeX para la elaboración de esta documentación.

Para finalizar, ha sido una gran satisfacción personal llevar a cabo este proyecto, a pesar de los constantes y numerosos contratiempos. Poder manejar herramientas y comprender tecnologías, cuyo potencial apenas se está vislumbrando y que únicamente se encuentran bajo el conocimiento de unos pocos (sus creadores), es la aportación más importante de este proyecto.

5.3. Posibles ampliaciones

En este proyecto se ha construido una cadena de procesamiento lingüístico para el idioma español a un nivel morfológico y sintáctico, las ampliaciones de este sistema son varias:

- En primer lugar se podría ampliar el léxico y la información morfológica y sintáctica recogida en el lexicón LEFFE. Reunir todo el léxico propio de un idioma tan complejo como el español, es una tarea difícil de

completar. Una posible solución es introducir el léxico concerniente a la temática de los textos a procesar[9].

- En este sentido, también se podría mejorar la metagramática para el español, tanto en cobertura como en especialización. Mejorar la cobertura, conllevaría describir un mayor número de fenómenos sintácticos de la gramática. Dicha tarea se centraría en aquellas estructuras sintácticas más especificas del idioma español.

  Especializar la metagramática, obligaría a detallar de forma exhaustiva los nuevos fenómenos lingüísticos y mejorar los existentes. El lexicón LEFFE está preparado para un mayor refinamiento de las estructuras sintácticas y las condiciones que las rigen. Éste ofrece una información morfosintáctica más profunda, todavía no manejada por el analizador sintáctico. Por ejemplo, el lexicón puede distinguir los diferentes tipos de determinantes o pronombres (artículos, demostrativos). Un mayor refinamiento de SPMG, provocaría la existencia de reglas más especificas que modelasen mejor el lenguaje natural y una reducción considerable de la tasa de ambigüedad ofrecida por la cadena desarrollada.

- Asimismo, podría incluirse en la cadena desarrollada un desambiguador sintáctico. Éste podría determinar cuál de las estructuras aceptadas por la gramática española para describir un determinado texto es la correcta, en base a un conjunto de *corpora* de muestra.

- Por otra parte, el sistema podría hacer frente a retos más complejos en PLN. Se podría incluir una etapa posterior al análisis sintáctico, donde se llevase a cabo un análisis semántico y/o pragmático de los textos procesados. De esta forma, no sería un mero proveedor de análisis morfosintácticos a otra aplicaciones, sino que el propio sistema se convertiría en una cadena completa de procesamiento lingüístico. Ello implicaría la construcción de un desambiguador sintáctico basado en información semántica.

- Asimismo, el incluir funcionalidades al sistema desarrollado, permitiría enfocarlo hacía disciplinas concretas como la adquisición de conocimiento, la minería de opiniones, la vigilancia tecnológica, indexación y localización documental basadas en conceptos, entre otras[10].

- Desde el punto de vista del contexto de trabajo, se podrían añadir al servidor *Parserd*, una mayor variedad de analizadores. Éstos podrían

[9] En una de las pruebas realizadas ha sido necesario introducir léxico botánico para poder procesar correctamente los *corpora* de entrada.

[10] Posiblemente, el desarrollo de estas herramientas sea el rumbo que tome el sistema propuesto en un futuro próximo.

presentar un léxico (*Lexer*) más especifico según la temática de los textos a analizar, por ejemplo botánico y bursátil. De esta forma, bajo una previa selección, el usuario podría determinar que analizador concreto le interesa usar en su marco de trabajo actual.

- Se podría diseñar una interfaz para el administrador del servidor *Parserd*.

Parte II

Manual Técnico

Capítulo 6

Análisis

En esta sección se identifican las principales funcionalidades que debe tener el sistema sin entrar en cómo se van a llevar a cabo o cual será el entorno de implantación del sistema. Utilizando uno o varios *diagramas de casos de uso* se pueden identificar cuales son las principales funcionalidades del sistema y delimitar la interacción del mismo en cuanto a otros sistemas o a usuarios, así como las relaciones entre estos y cada una de las funcionalidades del sistema.

Una vez hecho esto se procede a la creación de un primer *diagrama de clases* preliminar que especifique los componentes básicos del sistema. Las decisiones referentes al diseño se llevarán a cabo en la fase correspondiente.

Una vez identificados los componentes básicos del sistema se procederá a la identificación de una secuencia de interacciones lógicas que deberían tener estos componentes para implementar las funcionalidades especificadas en el *diagrama de casos de uso* del sistema. Se trata de establecer una coherencia lógica entre los componentes y no de determinar cuales serán en realidad los métodos y atributos de los componentes.

6.1. Requisitos funcionales

A continuación se muestran los requisitos o funcionalidades que debe cumplir el sistema. Estos son el punto de partida del desarrollo del proyecto. El objetivo final del mismo es obtener un producto que satisfaga dichos requisitos.

- Etiquetar las unidades léxicas presentes en una frase en castellano.
- Etiquetar las unidades léxicas presentes en un fichero de texto en castellano.
- Analizar morfológica y sintácticamente una frase de entrada en castellano.

- Analizar morfológica y sintácticamente un fichero de texto de entrada en castellano.
- Añadir nuevos analizadores al sistema.
- Consultar el registro de analizadores.
- Modificar el registro de analizadores del sistema.
- Visualizar el resultado del etiquetado de palabras como texto plano.
- Visualizar el resultado de un análisis morfosintáctico como bosque compartido de derivaciones.
- Visualizar el resultado de un análisis morfosintáctico en el formato XMLDep.
- Visualizar el resultado de un análisis morfosintáctico como grafo de dependencias.
- Visualizar el resultado de un análisis morfosintáctico como texto plano.
- Visualizar los datos estadísticos del análisis realizado.
- Activar la robustez del analizador.
- Iniciar el servidor de analizadores.
- Detener el servidor de analizadores.

6.2. Diagrama de casos de uso

En esta sección se presentan los *diagramas de casos de uso* del sistema. Son fiel reflejo de los requerimientos del sistema mostrando además los mecanismos o personas que interactúan con la aplicación desarrollada.

Atendiendo al contexto de las distintas funcionalidades que aporta el sistema, se ha decidido crear dos usuarios diferentes, *Administrador* y *Usuario*. Mientras que el primero, se encarga de la gestión del servidor; el segundo lleva a cabo la solicitud de procesamiento lingüístico de textos a través de los diferentes clientes del servidor *Parserd*: *Callparser* y la interfaz web *WebParser*.

En la figura 6.1 se muestra el *diagrama de casos de uso* general. A continuación se dividen y detallan estos casos de uso generales en otros casos de uso más específicos. Para facilitar la comprensión de los mismos, se ha decidido mostrar por un lado el *diagrama de casos de uso* del usuario *Administrador* (figura 6.2), y por otra parte, los correspondientes al usuario *Usuario* (figura 6.3).

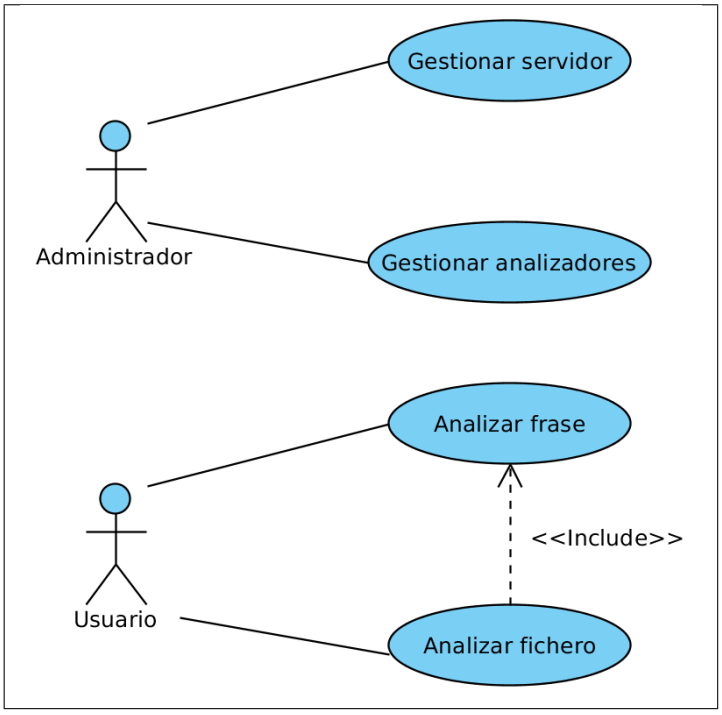


Figura 6.1: Diagrama de casos de uso general.

6.2.1. Casos de uso del Administrador

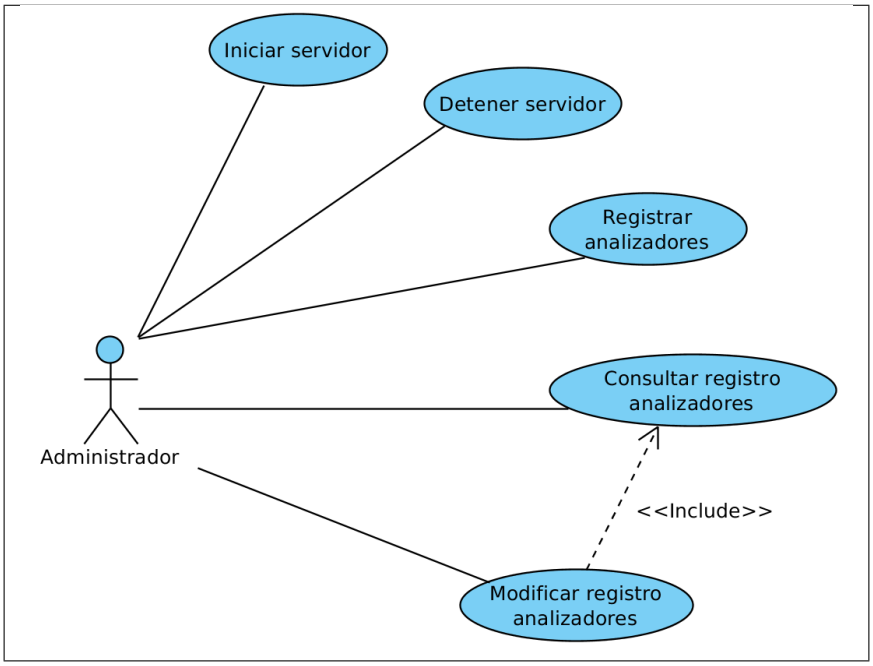


Figura 6.2: Diagrama de casos de uso del *Administrador*.

6.2.1.1. Iniciar servidor

- **Objetivo:** Arrancar el servidor de analizadores *Parserd*, para que el usuario pueda realizar peticiones de procesamiento lingüístico sobre los analizadores ubicados en él. Para ello es necesario que al menos un analizador esté registrado en el servidor.
- **Actores:** Administrador.
- **Disparadores:** Administrador.
- **Precondiciones:** El servidor *Parserd* se encuentra instalado en el equipo y no está arrancado.
- **Poscondiciones:** El servidor *Parserd* ha sido iniciado y se encuentra listo para atender las solicitudes de los clientes.
- **Flujo de Eventos:**

	Actor	Sistema
1	Solicita iniciar el servidor.	
2		Inicia el servidor.
3		Recupera la configuración del servidor.
4		Carga los analizadores registrados.
5		Muestra un mensaje de confirmación.
	Flujo Excepcional	
3a		Si no puede iniciar el servidor, muestra un mensaje de error.

Tabla 6.1: Iniciar servidor.

- **Frecuencia de uso:** Media.

6.2.1.2. Detener servidor

- **Objetivo:** Detener el servidor de analizadores *Parserd*, cuando éste se encuentre iniciado.
- **Actores:** Administrador.
- **Disparadores:** Administrador.
- **Precondiciones:** El servidor *Parserd* se encuentra instalado y arrancado en el equipo.
- **Poscondiciones:** El servidor *Parserd* ha sido detenido.
- **Flujo de Eventos:**

	Actor	Sistema
1	Solicita detener el servidor.	
2		Detiene el servidor.
3		Muestra un mensaje de confirmación.
	Flujo Excepcional	
3a		Si el servidor no está iniciado, muestra un mensaje de error.

Tabla 6.2: Detener servidor.

- **Frecuencia de uso:** Media.

6.2.1.3. Registrar analizadores

- **Objetivo:** El servidor *Parserd* exige el registro de los analizadores para poder ofrecer sus servicios de procesamiento lingüístico a los clientes. Para registrar un analizador es necesario indicar en el registro cual es su etiqueta identificativa, el analizador léxico (*Lexer*) y el analizador sintáctico (*Parser*) que lo componen, así como diferentes opciones del mismo como puede ser la activación o desactivación de la robustez. Todo ello se almacena en el fichero `parserd.conf` dedicado al registro de analizadores. En el caso de que los analizadores que se desee registrar ya se encuentren en el registro (coincidencias entre los identificadores de analizadores), éstos serán sobreescritos. Es necesario reiniciar el servidor para que los nuevos analizadores se carguen en el servidor, en el caso de que éste se encuentre ya iniciado.

- **Actores:** Administrador.

- **Disparadores:** Administrador.

- **Precondiciones:** El servidor *Parserd* y el analizador o los analizadores ha registrar se encuentran instalados en el equipo.

- **Poscondiciones:** Se ha registrado el analizador o los analizadores en el servidor *Parserd*.

- **Flujo de Eventos:**

	Actor	**Sistema**
1	Introduce la información necesaria del nuevo analizador (o analizadores).	
2	Confirma la operación.	
3		Recibe la información introducida.
4		Registra el/los analizador/es en el servidor.

Tabla 6.3: Registrar analizadores.

- **Frecuencia de uso:** Baja.

6.2.1.4. Consultar registro analizadores

- **Objetivo:** Mostrar el contenido del registro de analizadores que mantiene el servidor *Parserd*.
- **Actores:** Administrador.
- **Disparadores:** Administrador, Modificar registro analizadores.
- **Precondiciones:** El servidor *Parserd* se encuentra instalado en el equipo.
- **Poscondiciones:** Se ha consultado el registro de analizadores del servidor *Parserd*.
- **Flujo de Eventos:**

	Actor	Sistema
1	Solicita consultar el registro del servidor.	
2		Muestra la información del registro del servidor.

Tabla 6.4: Consultar registro analizadores.

- **Frecuencia de uso:** Media.

6.2.1.5. Modificar registro analizadores

- **Objetivo:** Editar la información del analizador o de los analizadores deseados entre los registrados en el servidor *Parserd*. Se obtiene una copia del registro, se introducen las modificaciones oportunas y se actualiza el registro. Es necesario reiniciar el servidor (en el caso de que esté iniciado) para que las modificaciones sean efectivas en la sesión actual. Esto es, que se actualicen los analizadores disponibles en el servidor cuyas características han sido modificadas.
- **Actores:** Administrador.
- **Disparadores:** Administrador.
- **Precondiciones:** El servidor *Parserd* se encuentra instalado en el equipo y el analizador o los analizadores ha modificar están registrados en el servidor.

- **Poscondiciones:** Se ha modificado el analizador o los analizadores seleccionados y registrados en el servidor *Parserd*.

- **Flujo de Eventos:**

	Actor	Sistema
1	Consulta los analizadores registrados. (*include* **Consultar registro analizadores**)	
2	Modifica los analizadores deseados.	
3	Guarda la operación realizada.	
4		Actualiza la información de los analizadores modificados en el registro.

Tabla 6.5: Modificar registro analizadores.

- **Frecuencia de uso:** Baja.

6.2.2. Casos de uso del Usuario

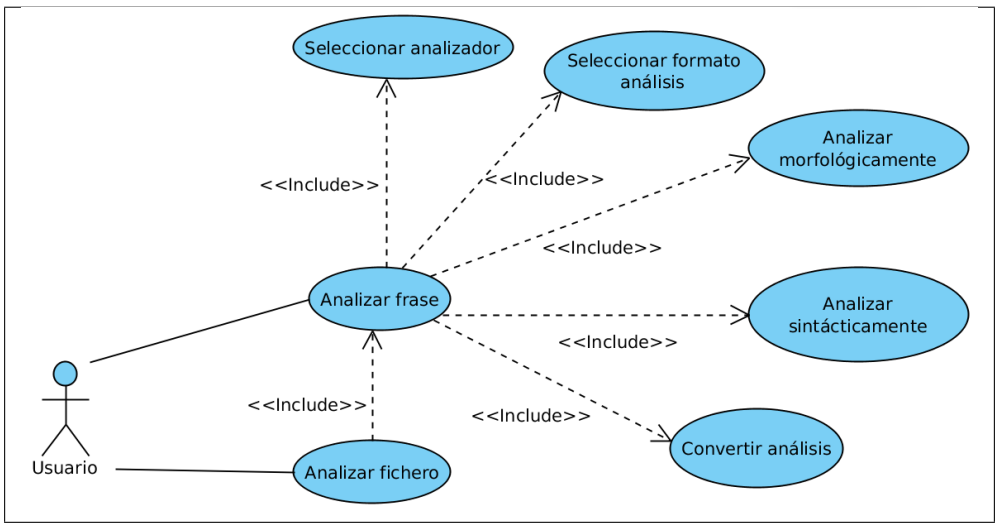


Figura 6.3: Diagrama de casos de uso del *Usuario*.

6.2.2.1. Analizar frase

- **Objetivo:** Realizar un análisis morfosintáctico sobre una frase de texto en castellano (introducida por el usuario o recogida de un fichero) mediante un analizador disponible y mostrando el análisis en un formato comprensible y seleccionado por el usuario.
- **Actores:** Usuario.
- **Disparadores:** Usuario, Analizar fichero.
- **Precondiciones:** El servidor *Parserd* está instalado en el equipo, éste se encuentra iniciado y dispone de al menos un analizador registrado.
- **Poscondiciones:** Se ha analizado morfosintácticamente la frase introducida mediante el analizador seleccionado.
- **Flujo de Eventos:**

	Actor	Sistema
1	Selecciona un analizador. (*include* **Seleccionar analizador**)	
2	Selecciona el formato de salida. (*include* **Seleccionar formato análisis**)	
3	Introduce una frase de texto.	
4	Pulsa el botón *Analizar*.	
5		Recibe la información proporcionada.
6		Realiza un etiquetado morfológico del texto mediante el analizador seleccionado. (*include* **Analizar morfológicamente**)
7		Analiza sintácticamente el texto etiquetado mediante el analizador seleccionado. (*include* **Analizar sintácticamente**)
8		Convierte el resultado del análisis en el formato seleccionado. (*include* **Convertir análisis**)

9		Muestra el resultado del análisis.
	Flujo Excepcional	
5a		Si la información introducida no es correcta, muestra un mensaje de error y vuelve al paso 1.
9a		Si se ha producido un problema durante el análisis, muestra un mensaje de error.

Tabla 6.6: Analizar frase.

- **Frecuencia de uso:** Alta.

6.2.2.2. Analizar fichero

- **Objetivo:** Realizar un análisis morfosintáctico sobre cada una de las frases en castellano que componen un fichero de texto introducido por el usuario, mostrando el análisis en un formato comprensible y seleccionado por el usuario.
- **Actores:** Usuario.
- **Disparadores:** Usuario.
- **Precondiciones:** El servidor *Parserd* está instalado en el equipo, éste se encuentra iniciado y dispone de al menos un analizador registrado.
- **Poscondiciones:** Se ha analizado morfosintácticamente las frases presentes en el fichero introducido por el usuario.
- **Flujo de Eventos:**

	Actor	**Sistema**
1	Introduce un fichero.	
2		Carga las frases presentes en el fichero.
3		Analiza cada una de las frases del fichero. (*include* **Analizar frase**)

	Flujo Excepcional	
2a		Si no es posible cargar el fichero, muestra un mensaje de error y vuelve al paso 1.

Tabla 6.7: Analizar fichero.

- **Frecuencia de uso:** Alta.

6.2.2.3. Seleccionar analizador

- **Objetivo:** Indicar qué analizador morfosintáctico se va a utilizar entre los registrados en el servidor *Parserd*.
- **Actores:** Usuario.
- **Disparadores:** Analizar frase.
- **Precondiciones:** El servidor *Parserd* está instalado en el equipo, éste se encuentra iniciado y el analizador seleccionado por el usuario está registrado en el servidor.
- **Poscondiciones:** Se ha seleccionado un analizador morfosintáctico concreto.
- **Flujo de Eventos:**

	Actor	Sistema
1	Consulta los analizadores disponibles.	
2		Muestra los analizadores disponibles.
3	Selecciona el analizador deseado.	
4		Recibe la información introducida.

Tabla 6.8: Seleccionar analizador.

- **Frecuencia de uso:** Alta.

6.2.2.4. Seleccionar formato análisis

- **Objetivo:** Indicar al sistema qué información y en qué formato va a mostrar el procesamiento realizado sobre el texto objeto de la petición. Cada analizador soporta unos determinados formatos de representación de la salida del análisis. El sistema ha de ofrecer al usuario únicamente los formatos disponibles para el analizador seleccionado.
- **Actores:** Usuario.
- **Disparadores:** Analizar frase.
- **Precondiciones:** El servidor *Parserd* está instalado en el equipo, éste se encuentra iniciado, ha sido seleccionado un analizador determinado y el formato seleccionado por el usuario está disponible en el servidor.
- **Poscondiciones:** Se ha seleccionado un formato de salida concreto.
- **Flujo de Eventos:**

	Actor	Sistema
1	Consulta los formatos disponibles.	
2		Recupera los formatos disponibles para el analizador seleccionado.
3		Muestra los formatos disponibles.
4	Selecciona el formato deseado.	
5		Recibe la información introducida.

Tabla 6.9: Seleccionar formato análisis.

- **Frecuencia de uso:** Alta.

6.2.2.5. Analizar morfológicamente

- **Objetivo:** Realiza la segmentación, lematización y etiquetación morfosintáctica de las unidades léxicas presentes en el texto mediante el analizador seleccionado.
- **Actores:** -
- **Disparadores:** Analizar frase.

- **Precondiciones:** El servidor *Parserd* está instalado en el equipo, éste se encuentra iniciado, se ha seleccionado el analizador y se ha cargado un texto a analizar.
- **Poscondiciones:** Se ha seleccionado segmentado, lematizado y etiquetado las unidades léxicas del texto de entrada mediante el analizador seleccionado.
- **Flujo de Eventos:**

	Actor	Sistema
1		Recibe la frase a analizar.
2		Segmenta la frase en sus unidades léxicas.
3		Lematiza cada unidad léxica.
4		Etiqueta cada unidad léxica.
5		Devuelve la frase segmentada y etiquetada.
	Flujo Excepcional	
5a		Si se produce algún problema en el proceso anterior, devuelve un mensaje de error.

Tabla 6.10: Analizar morfológicamente.

- **Frecuencia de uso:** Alta.

6.2.2.6. Analizar sintácticamente

- **Objetivo:** Realiza el análisis sintáctico de la frase etiquetada resultado del análisis morfológico mediante el analizador seleccionado. Esto es, obtener el bosque compartido de derivaciones que describe la estructura sintáctica de la frase de entrada.
- **Actores:** -
- **Disparadores:** Analizar frase.
- **Precondiciones:** El servidor *Parserd* está instalado en el equipo, éste se encuentra iniciado, se ha seleccionado el analizador y el texto a analizar ha sido etiquetado.
- **Poscondiciones:** Se ha analizado sintácticamente la frase etiquetada mediante el analizador seleccionado.

- **Flujo de Eventos:**

	Actor	Sistema
1		Recibe la frase etiquetada.
2		Realiza el análisis sintáctico.
3		Devuelve el bosque compartido de derivaciones que describe la estructura sintáctica de la frase.
	Flujo Excepcional	
3a		Si se produce algún problema durante el análisis, devuelve un mensaje de error.

Tabla 6.11: Analizar sintácticamente.

- **Frecuencia de uso:** Alta.

6.2.2.7. Convertir análisis

- **Objetivo:** Convierte el resultado del procesamiento realizado por el sistema bajo el formato indicado previamente por el usuario.

- **Actores:** -

- **Disparadores:** Analizar frase.

- **Precondiciones:** El servidor *Parserd* está instalado en el equipo, éste se encuentra iniciado, se ha llevado acabo el procesamiento lingüístico del texto de entrada y se ha seleccionado el formato de salida.

- **Poscondiciones:** Se ha convertido el resultado obtenido en el análisis en el formato especificado por el usuario.

- **Flujo de Eventos:**

	Actor	Sistema
1		Recibe el análisis lingüístico realizado.
2		Realiza la conversión al formato solicitado.
3		Devuelve el resultado en el formato indicado.
	Flujo Excepcional	
3a		Si se produce algún problema, devuelve un mensaje de error.

Tabla 6.12: Convertir análisis.

- **Frecuencia de uso:** Alta.

6.3. Diagrama de clases del análisis

La figura 6.4 muestra las clases o entidades identificadas en el análisis de requisitos del sistema y las relaciones entre ellas. Este *diagrama de clases* se verá modificado progresivamente según se entre en tareas de diseño del sistema. Algunas de las clases de este diagrama inicial podrán desaparecer, o verse modificadas en el *diagrama de clases* final.

6.4. Diccionario de clases del análisis

6.4.1. Parserd

Es la clase encargada de ofrecer el servicio de procesamiento lingüístico ante la solicitud de los clientes del servidor. Por lo tanto, es la clase principal que se encarga de controlar los diferentes analizadores registrados en el servidor y su interacción con los clientes que solicitan su servicio.

Cada analizador presente en el servidor está compuesto por la combinación de un analizador morfológico (*Lexer*) y un analizador sintáctico (*Parser*). Ambos tipos de componentes son cargados en el momento del arranque del servidor. En el servidor puede haber distintos analizadores que comparten *Lexers* o *Parsers*. Para localizar cada analizador, el servidor mantiene un registro en el fichero `parserd.conf` donde se detalla bajo que identificador unívoco de analizador se recoge un *Lexer* y un *Parser* concretos, además de otras opciones referentes a ese analizador.

Además también maneja los formatos de representación soportados por los analizadores registrados en el servidor.

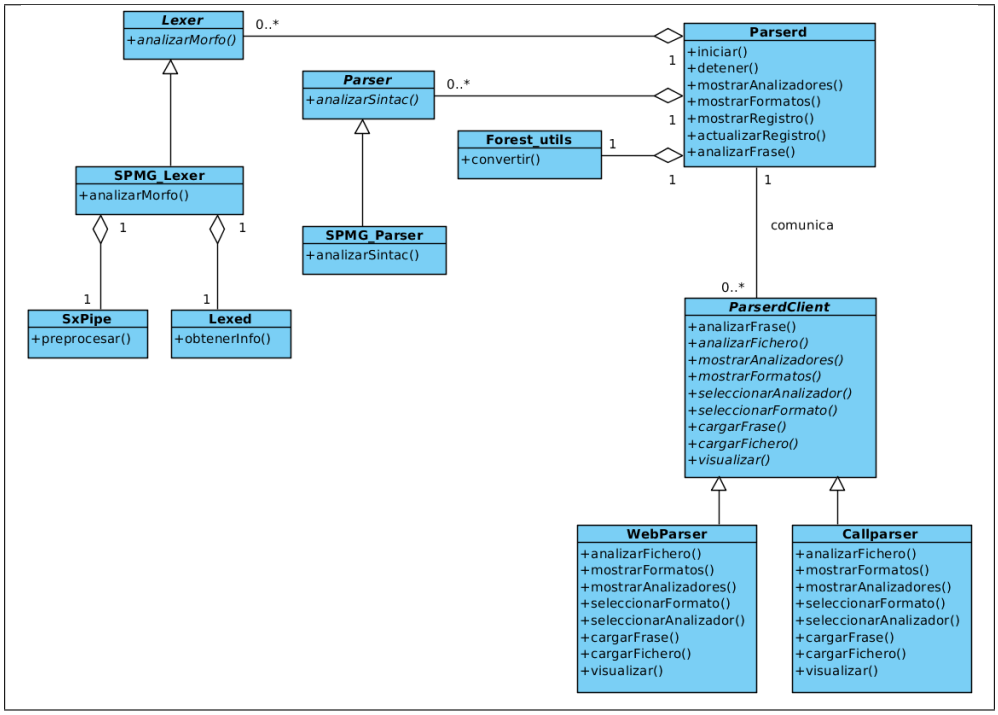


Figura 6.4: Diagrama de clases del análisis.

6.4.2. Lexer

Clase abstracta que recoge los diferentes analizadores morfológicos presentes en el servidor y que forman parte de al menos un analizador registrado en *Parserd*. Su principal función es proveer de la funcionalidad de análisis léxico a los analizadores del servidor configurados para utilizarlo como *Lexer*.

6.4.3. Parser

Clase abstracta que recoge los diferentes analizadores sintácticos presentes en el servidor y que forman parte de al menos un analizador registrado en *Parserd*. Su principal función es proveer de la funcionalidad de análisis sintáctico a los analizadores del servidor configurados para utilizarlo como *Parser*.

6.4.4. SPMG_Lexer

Componente concreto diseñado para proveer de análisis morfológico al analizador del español. Para ello utiliza los componentes *SxPipe* y *Lexed*.

6.4.5. SxPipe

Es el preprocesador encargado de segmentar la oración de entrada. Para ello construye un GAD (Grafo Dirigido Acíclico), que contiene las palabras de la oración como nodos.

6.4.6. Lexed

Es la clase encargada de recoger y manejar la información morfosintáctica del lexicón LEFFE, que permitirá al analizador léxico, lematizar y etiquetar cada una de las palabras reconocidas por el preprocesador *SxPipe*.

6.4.7. SPMG_Parser

Componente concreto que provee de análisis sintáctico al analizador del español.

6.4.8. Forest_utils

Componente encargado de manejar los diferentes formatos utilizados por los analizadores del servidor para ofrecer el procesamiento lingüístico realizado al usuario.

6.4.9. ParserdClient

Clase genérica que recoge el comportamiento común de los clientes del servidor *Parserd*.

6.4.10. Callparser

Cliente que utiliza línea de comandos para acceder al servidor vía *telnet*.

6.4.11. WebParser

Cliente *web* que accede al servidor a través del protocolo HTTP.

6.5. Diagramas de secuencia conceptuales

En este apartado se muestran los *diagramas de secuencia* de cada uno de los escenarios presentes en el *diagrama de casos de uso*. Estos *diagramas de secuencia* se han elaborado con las clases obtenidas durante la fase de análisis. Como se ha hecho previamente, se dividen los *diagramas de secuencia* de acuerdo a los dos posibles usuarios del sistema, con el fin de facilitar su legibilidad.

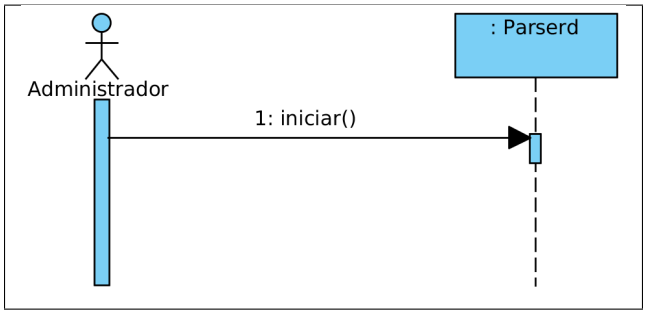


Figura 6.5: Diagrama de secuencia conceptual de *Iniciar servidor*.

Figura 6.6: Diagrama de secuencia conceptual de *Detener servidor*.

6.5.1. Diagramas de secuencia del Administrador

6.5.1.1. Iniciar servidor

La figura 6.5 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Iniciar servidor*.

6.5.1.2. Detener servidor

La figura 6.6 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Detener servidor*.

6.5.1.3. Registrar analizadores

La figura 6.7 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Registrar analizadores*.

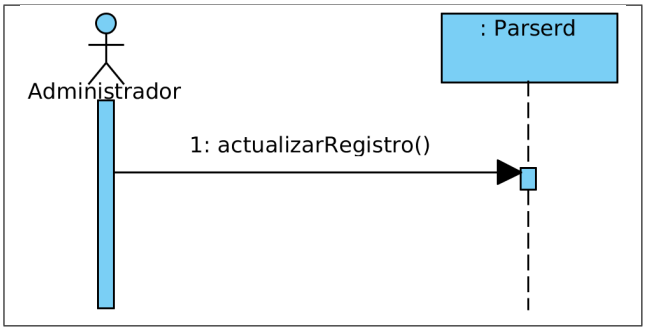


Figura 6.7: Diagrama de secuencia conceptual de *Registrar analizadores*.

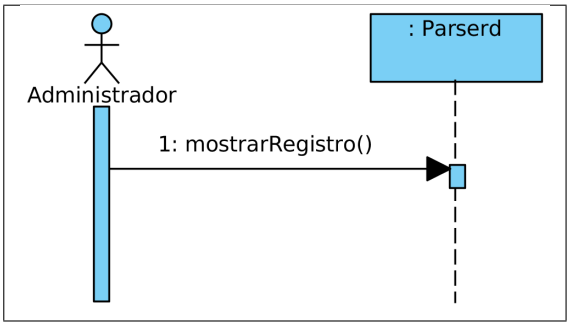


Figura 6.8: Diagrama de secuencia conceptual de *Consultar registro analizadores*.

6.5.1.4. Consultar registro analizadores

La figura 6.8 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Consultar registro analizadores*.

6.5.1.5. Modificar registro analizadores

La figura 6.9 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Modificar registro analizadores*.

6.5.2. Diagramas de secuencia del Usuario

6.5.2.1. Analizar frase

La figura 6.10 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado

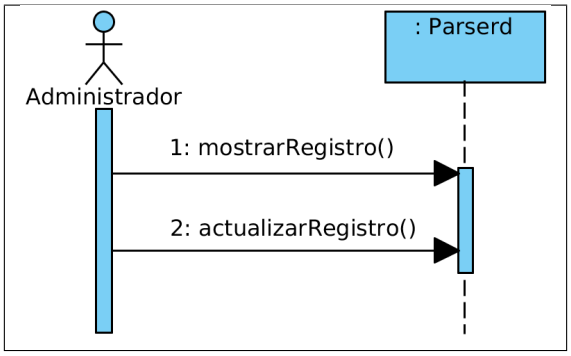


Figura 6.9: Diagrama de secuencia conceptual de *Modificar registro analizadores*.

como *Analizar frase*.

6.5.2.2. Analizar fichero

La figura 6.11 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Analizar fichero*.

6.5.2.3. Seleccionar analizador

La figura 6.12 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Seleccionar analizador*.

6.5.2.4. Seleccionar formato análisis

La figura 6.13 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Seleccionar formato análisis*.

6.5.2.5. Analizar morfológicamente

La figura 6.14 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Analizar morfológicamente*. Se detalla la interacción entre componentes para el caso concreto de la cadena para el español. Esto es, se muestra como el analizador léxico *SPMG Lexer* interactúa con el preprocesador *SxPipe* y el lexicalizador *Lexed*. Otros *Lexers* presentes en el servidor, no tienen por qué emplear los recursos utilizados por *SPMG Lexer*. De ahí que, en el siguiente

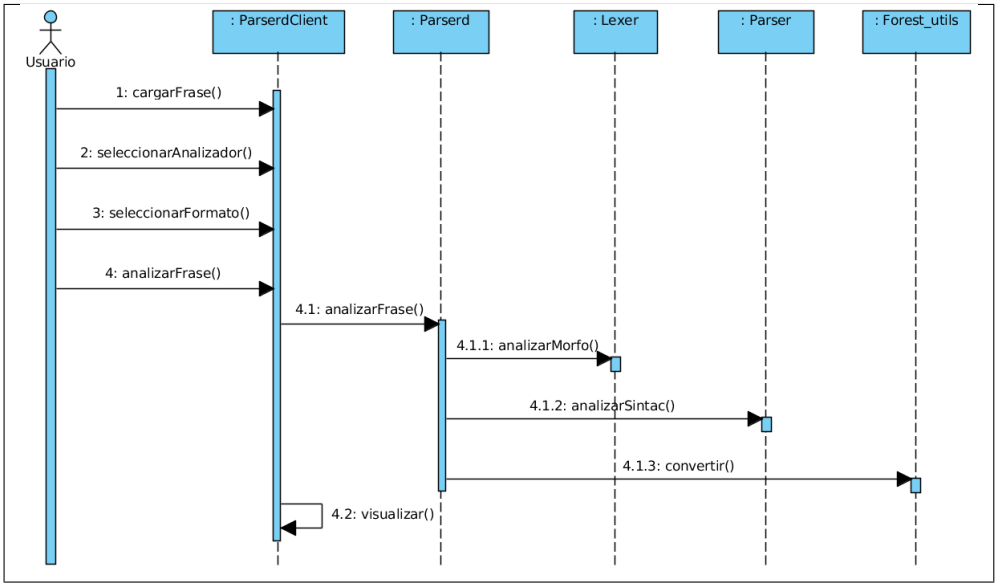


Figura 6.10: Diagrama de secuencia conceptual de *Analizar frase*.

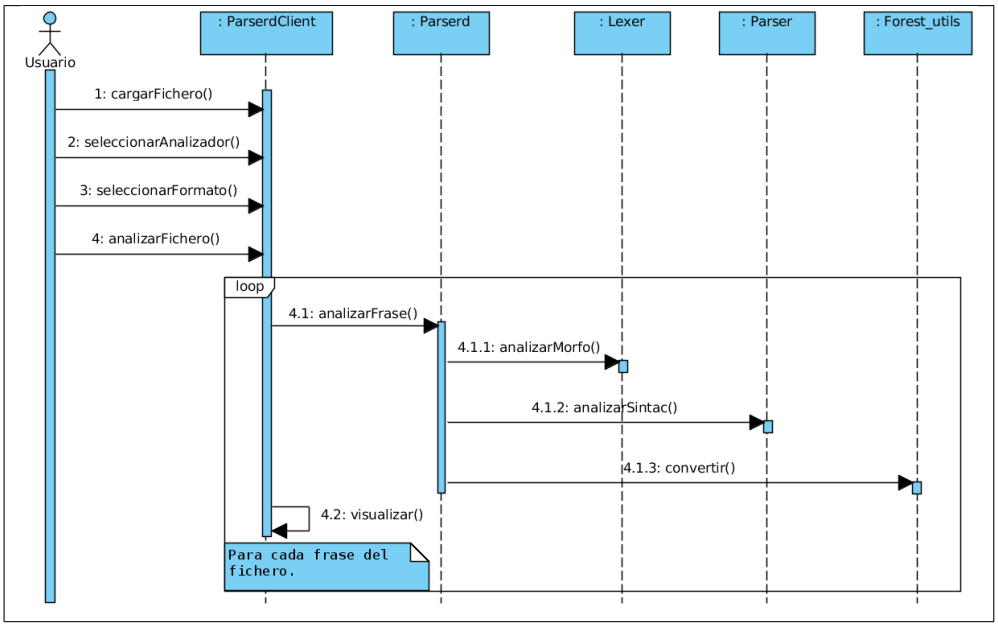


Figura 6.11: Diagrama de secuencia conceptual de *Analizar fichero*.

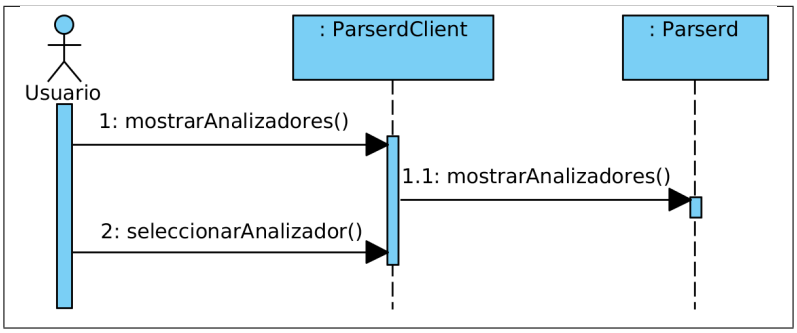


Figura 6.12: Diagrama de secuencia conceptual de *Seleccionar analizador*.

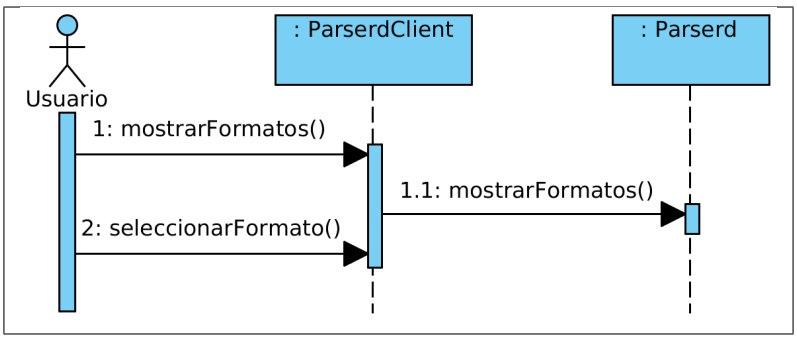


Figura 6.13: Diagrama de secuencia conceptual de *Seleccionar formato análisis*.

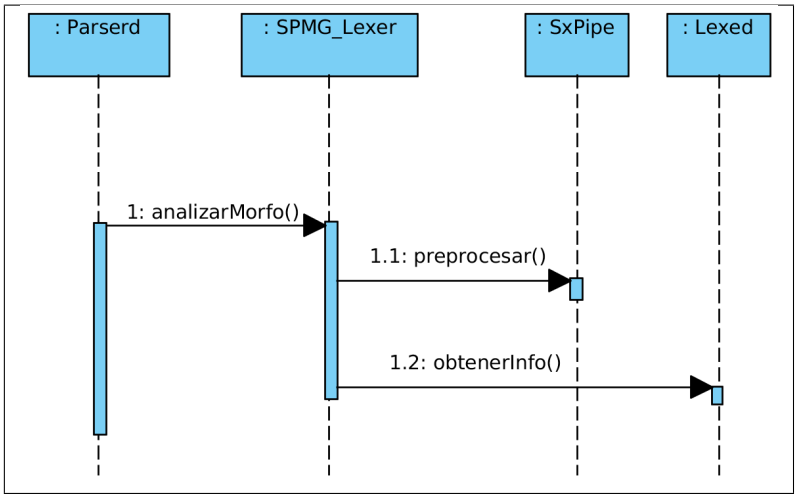


Figura 6.14: Diagrama de secuencia conceptual de *Analizar morfológicamente*.

diagrama de secuencia no se trate de la clase genérica *Lexer* sino de la clase específica *SPMG_ Lexer*.

6.5.2.6. Analizar sintácticamente

La figura 6.15 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Analizar sintácticamente*.

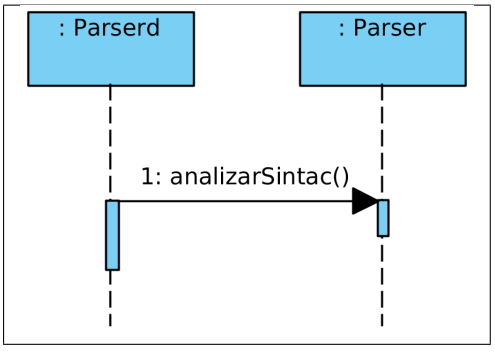


Figura 6.15: Diagrama de secuencia conceptual de *Analizar sintácticamente*.

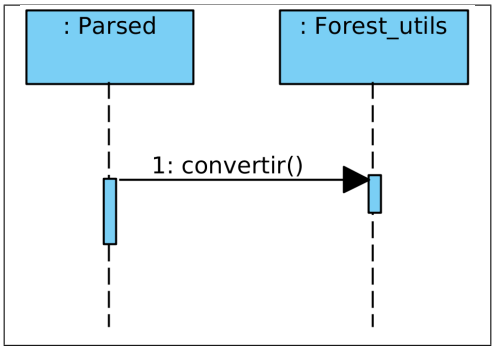


Figura 6.16: Diagrama de secuencia conceptual de *Convertir análisis*.

6.5.2.7. Convertir análisis

La figura 6.16 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Convertir análisis*.

Capítulo 7

Diseño

En esta sección se va a presentar la documentación de la fase de diseño del proyecto. En esta fase del desarrollo, se detalla como se van a construir los requisitos iniciales descritos durante la fase de análisis.

En este punto es necesario dejar claro que el presente proyecto no se rige en su totalidad bajo el paradigma orientado a objetos. Mientras que la metagramática SPMG sigue un paradigma orientado a objetos, los componentes o módulos implementados en el lenguaje *Perl* son estructurados. Ello implica llevar a cabo un análisis y diseño, por un lado estructurado y por otro orientado a objetos. Mientras que el primero emplea DFDs (Diagramas de Flujo de Datos) y *diagramas de Entidad-Relación*, el segundo utiliza los diagramas estudiados a lo largo de la carrera y que nos proporciona el conocido UML.

Con el fin de favorecer la homogeneidad del proyecto y utilizar los conocimientos adquiridos, se ha decidido emplear los diagramas que nos ofrece UML para ambos paradigmas. De este modo, se puede establecer una equivalencia para el presente trabajo, entre DFDs y los diagramas de UML, *diagramas de actividad* y *diagramas de secuencia*[1]; y entre los *diagramas de clases* y los *diagramas de Entidad-Relación*. Ello implica que algunas de las clases presentes en los diagramas no se correspondan con clases de código, sino que se asocien con entidades lógicas, componentes o módulos que forman el sistema desarrollado bajo un paradigma estructurado. Del mismo modo, los métodos y atributos presentes en los *diagramas de clases* y *diagramas de secuencia* son las principales funciones/procedimientos y variables, respectivamente, de los distintos componentes del sistema. Ello hace posible describir el flujo de datos e interacciones que se producen entre los diferentes componentes del sistema, utilizando los diagramas que UML

[1]Los *diagramas de actividad* y *diagramas de secuencia* vienen a describir la misma información que un DFD. Mientras que el *diagrama de actividad* refleja el flujo de control del programa, el *diagrama de secuencia* ofrece el flujo de datos intercambiados entre componentes, así como la interacción entre los mismos.

nos ofrece.

Inicialmente, se modela estáticamente el sistema para observar si es necesario crear nuevas clases (entidades) a partir de las identificadas en la fase de análisis. Para tal cometido, se emplea un *modelo de componentes* y un *modelo de clases del sistema*. Con ambos diagramas se intenta presentar la estructura del sistema de una forma estática. Esto es, sin mostrar las secuencias de interacciones entre los componentes del sistema ni la serie de estados por los que atraviesa un determinado componente o el sistema en general.

El *diagrama de componentes* se utiliza inicialmente para delimitar el entorno del sistema con el fin de identificar las necesidades del entorno real de ejecución y poder distinguir así entre componentes de presentación y componentes de lógica de negocio.

A continuación, se ha utilizado un *diagrama de clases* para mostrar las dependencias entre las distintas entidades que componen la aplicación. Para luego mostrar y describir de forma detallada el comportamiento esperado de cada clase y cuales van a ser sus procedimientos o funciones principales.

Una vez realizado un modelo estático se procede a la elaboración de modelos dinámicos del sistema con *diagramas de secuencia* y *diagramas de actividad*.

Los *diagramas de actividad* delimitarán cuales son las tareas básicas que tiene que realizar el sistema para completar las funcionalidades expuestas en los *diagramas de casos de uso* del análisis. Mientras que los *diagramas de secuencia* permitirán identificar las principales funciones y procedimientos necesarios para conseguir las funcionalidades del *diagrama de actividad* y mostrarán el modo en como van interaccionando e intercambiando datos los componentes del sistema para conseguir las funcionalidades deseadas.

A mayores, en esta fase del desarrollo se ha incluido el diseño de la metagramática SPMG, base del analizador sintáctico. Se trata de una gramática abstracta descrita mediante un formalismo orientado a objetos. Ésta recoge las estructuras sintácticas permitidas en el castellano mediante una jerarquía de clases, donde cada clase describe un fenómeno sintáctico concreto. Para su documentación se ha decidido utilizar *diagramas de clases*. Estos permitirán plasmar las relaciones jerárquicas existentes entre las clases de la metagramática. Para a continuación de las mismas, detallar los fenómenos sintácticos que modela cada una de las clases de la metagramática para el español[2].

[2]No habrá diagramas de modelado dinámico (*diagramas de secuencia* o *diagramas de actividad*), puesto que las clases de la metagramática no disponen de métodos para interactuar entre ellas. Una metagramática es una descripción estática de una jerarquía de clases.

7.1. Modelo de componentes

En el modelo de la figura 7.1 se muestra la arquitectura general del sistema centrando la atención en los componentes *software* que lo componen y las relaciones existentes entre ellos.

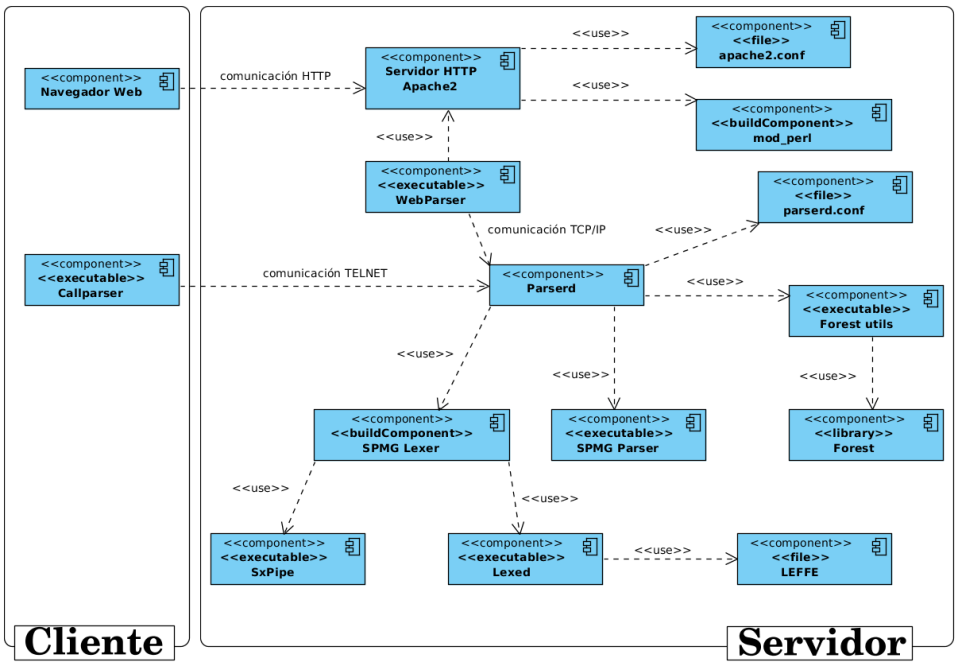


Figura 7.1: Diagrama de componentes.

Además, y saliendo de la estructura de un *diagrama de componentes*, se ha decidido incluir una línea divisoria del diagrama con el fin de crear una frontera entre el *hardware* del cliente y el *hardware* del servidor. La idea es que existan dos máquinas:

- **Servidor:** En ésta se ejecuta el servidor *Parserd* y el servidor *Apache2*[3]. Cada servidor tiene su correspondiente fichero de configuración y, además, el servidor *Apache2* dispone de un módulo (*mod_perl*) diseñado para el manejo de programas implementados en *Perl*.

  En la máquina servidora, también está presente, la cadena de procesamiento lingüístico para el español formada por los ejecutables *SPMG Lexer*, *SPMG Parser* y *Forest utils*. El componente encargado

[3]El servidor HTTP *Apache2*, versión *Apache* de *Ubuntu*, es necesario para permitir el acceso al servidor de analizadores desde navegadores *web*.

del análisis léxico (*SPMG Lexer*) está compuesto a su vez por el preprocesador *SxPipe* y por el lexicón morfosintáctico LEFFE, donde el ejecutable *Lexed* es el encargado de manejar este último.

Para poder acceder a través de un navegador *web* al servidor de analizadores, se ha diseñado el cliente *WebParser*. Éste se ejecuta en el servidor *Apache* y genera una interfaz *web* dinámica que atiende las peticiones de los usuarios a través del protocolo HTTP, para a continuación remitirlas al servidor *Parserd* mediante un *socket* TCP/IP[4].

- **Cliente:** En las máquinas cliente, será necesario disponer, bien del cliente *Callparser*, que accede al servidor *Parserd* vía *telnet*, o bien, de un navegador *web* cualquiera.

7.2. Modelo de clases del sistema

El *modelo de clases del sistema* de la figura 7.2 nos muestra la arquitectura interna del sistema. En este caso, prácticamente no ha cambiado con respecto al *diagrama de clases del análisis*. La razón principal de ello, es que el sistema desarrollado sigue una estructura muy similar a la cadena ALPAGE, herramienta que sirve de modelo para este trabajo.

7.3. Diccionario de clases del sistema

En esta sección se detallan las principales funciones o procedimientos presentes en los diferentes componentes y a través de los cuales se producen interacciones e intercambio de datos entre ellos. Además, también se detallan las principales variables de cada entidad, donde se almacenan los datos más relevantes del sistema[5].

Algunos de los componentes pertenecientes al sistema en construcción son reutilizados. Por lo tanto, no es necesario profundizar en el diseño de los mismos. Simplemente se indican cuales son sus principales funciones (y/o procedimientos) y variables, a través de los cuales interactúan con el resto de recursos del sistema. Ello permite comprender mejor el funcionamiento global de la aplicación.

[4]Podría considerarse la situación en que los servidores *Parserd* (con la cadena de procesamiento) y *Apache2* (con *WebParser*) se ubicasen en distintas máquinas. Esta posibilidad se puede ver en el apartado 2.4 de la *Memoria*.

[5]Nótese que cuando se habla de *métodos* y *atributos* en el caso de componentes estructurados se refiere a funciones/procedimientos y variables, respectivamente.

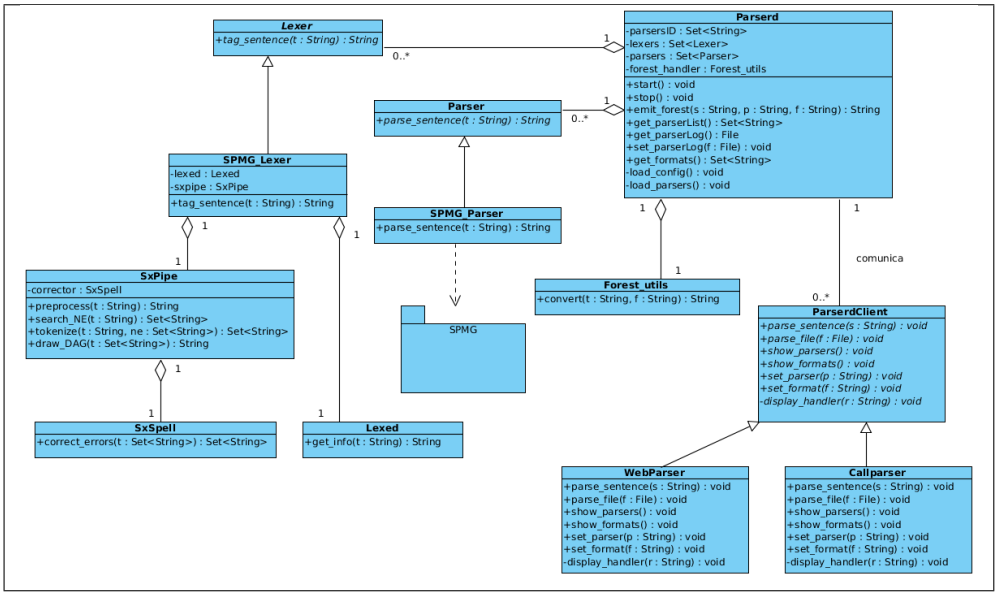


Figura 7.2: Diagrama de clases del sistema.

7.3.1. Parserd

Esta clase representa un componente que reúne los suficientes recursos para poder ofrecer una arquitectura cliente-servidor al sistema desarrollado. Su función será la de recoger las peticiones de los clientes y remitírselas al analizador correspondiente. Esto es, recibir una frase de texto dada y enviársela al *Lexer* y *Parser* registrados bajo el identificador del analizador indicado. Además, el resultado del análisis ha de ser convertido en el formato de salida solicitado por el usuario.

Tanto la configuración del servidor como el registro de analizadores se recoge en el fichero `parserd.conf`.

Para lograr esta funcionalidad dispone de los siguientes atributos y métodos:

- **Atributos:**
  - `parsersID: Set<String>`. Conjunto de identificadores de los analizadores cargados del registro.
  - `parsers: Set<Parser>`. Conjunto de *Parsers* cargados. Éstos han de pertenecer al menos a un analizador registrado.
  - `lexers: Set<Lexer>`. Conjunto de *Lexers* cargados. Éstos han de pertenecer al menos a un analizador registrado.
  - `forest_handler: Forest_utils`. Componente encargado de la conversión del análisis.

- **Métodos:**
  - `start(): void`. Su función es la de arrancar el servidor.
  - `stop(): void`. Implementa los pasos necesarios para detener el servidor.
  - `emit_forest(s:String, p:String, f:String): String`. Es el procedimiento encargado de llevar a cabo el análisis morfosintáctico solicitado. Requiere de una frase a analizar, el identificador del analizador a usar y el formato de salida en el cual se va a representar el resultado. Este método se encarga de obtener a partir del identificador del analizador cual son el *Lexer* y *Parser* que lo componen para ordenarles el análisis de la frase dada.
  - `get_parserList(): Set<String>`. Retorna los identificadores de los analizadores registrados y actualmente cargados en el servidor.
  - `get_parserLog(): File`. Devuelve la parte del fichero de configuración `parserd.conf` destinada al registro de los analizadores.
  - `set_parserLog(f:File): void`. Actualiza la zona del fichero de configuración `parserd.conf` destinada al registro de analizadores.
  - `get_formats(): Set<String>`. Obtiene una lista con los formatos disponibles en el paquete *forest utils*.
  - `load_parsers(): void`. Se utiliza durante el arranque del servidor para cargar los *Lexers* y *Parsers* de los analizadores registrados.
  - `load_config(): File`. Carga del fichero `parserd.conf` las opciones de configuración del servidor.

Se trata de un componente heredado de la cadena ALPAGE e integrado en el sistema desarrollado.

7.3.2. Lexer

Esta clase abstracta define el comportamiento general de un analizador léxico *Lexer*. A nivel de implementación, no se trata de la herencia de un lenguaje orientado a objetos, sino de unas funciones comunes que han de presentar todos los *Lexers* para que el servidor puede acceder a ellos de la misma forma.

Todos los *Lexers* disponen del siguiente método a implementar de forma especifica por cada analizador léxico concreto:

- **Métodos:**

- `tag_sentence(t:String): String`. Método encargado de devolver una frase segmentada en sus unidades léxicas[6] etiquetadas mediante información morfosintáctica.

7.3.3. Parser

Para el caso de los *Parsers*, también es necesario seguir unas pautas a la hora de implementar un *Parser* concreto para un determinado idioma. Todos ellos han de implementar el siguiente método para que el servidor pueda acceder a ellos de un modo uniforme.

- **Métodos:**
  - `parse_sentence(t:String): String`. Método encargado de obtener la estructura sintáctica de una frase segmentada con sus unidades léxicas etiquetadas mediante información morfosintáctica. Retornando en una cadena el bosque compartido de derivaciones que recoge los árboles GAR empleados para cubrir el texto de entrada.

7.3.4. SPMG_Lexer

Este componente ha sido desarrollado a lo largo de este proyecto para proveer de análisis morfológico al analizador o cadena de procesamiento lingüístico del español. Para ello utiliza los componentes *SxPipe* y *Lexed*.

- **Atributos:**
  - `lexed: Lexed`. Se trata de un componente empleado por el analizador léxico *SPMG_Lexer* para tener acceso al lexicón morfosintáctico LEFFE.
  - `sxpipe: SxPipe`. Es el recurso encargado del preprocesado de la frase de entrada.
- **Métodos:**
  - `tag_sentence(t:String): String`. Método encargado de retornar una frase segmentada con sus unidades léxicas etiquetadas mediante información morfosintáctica. Más concretamente, el analizador léxico del español lleva a cabo la lematización y segmentación de cada unidad léxica presente en el GAD obtenido por *SxPipe* mediante la información presente en el lexicón LEFFE. La lematización consiste en encontrar cual es el lema de la forma actual de la palabra. LEFFE contiene todas las formas de una palabra [7]. A

[6] Una unidad léxica incluye una palabra o un signo de puntuación.
[7] En el caso de que el lema de esa palabra esté almacenado en sus ficheros

partir de ellas, se puede obtener, cual son los posibles lemas para esa forma[8]. Por otro lado, etiquetación, es la asignación a cada palabra de la información morfosintáctica recogida en el LEFFE para esa forma concreta.

7.3.5. SxPipe

Es el preprocesador encargado de segmentar la oración de entrada. Para ello construye un GAD (Grafo Dirigido Acíclico) que contiene las palabras de la oración como nodos.

- **Atributos:**
  - `corrector: SxSpell`. Es un recurso empleado por *SxPipe* para llevar a cabo la corrección de unidades léxicas erróneas.
- **Métodos:**
  - `preprocess(t:String): String`. Es el procedimiento invocado por *SMPG_Lexer* para solicitar el preprocesado de una frase dada. Este se compone de cuatro métodos principales, tres de ellos descritos a continuación y otro presente en *SxSpell*.
  - `search_NE(t:String): Set<String>`. Antes de llevar a cabo la segmentación de la oración se buscan las entidades con nombre para evitar que sean segmentadas erróneamente. De ello se obtiene un conjunto de entidades con nombre presentes en la oración.
  - `tokenize(t:String, ne: Set<String>): Set<String>`. Recibe las entidades con nombre localizadas en la etapa anterior, y evitando separarlas, realiza la segmentación de las unidades léxicas (palabras y signos de puntuación) presentes en la frase de entrada. Retorna el conjunto de palabras y signos de puntuación resultantes del proceso.
  - `draw_DAG(t:Set<String>): String`. La salida que ofrece *SxPipe* al componente *SPMG_Lexer* es un grafo GAD. Por ello es necesario dibujar y retornar un grafo, donde los nodos sean las unidades léxicas de la oración, para que la cadena continúe con su procesamiento.

Recurso reutilizado de la cadena ALPAGE e integrado en el sistema en desarrollo.

[8] Nótese que una forma puede tener distintos lemas. La forma *una*, puede tener el lema *una* (determinante) o *unir* (verbo), entre otros.

7.3.6. SxSpell

Componente de *SxPipe* encargado de corregir las palabras erróneas de la frase de entrada. Ante una palabra erróneamente escrita, el analizador léxico, la marcará como *palabra desconocida*. Ello implicará que a la hora de asignarle una categoría morfológica, se indique que puede tener varias posibles categorías. Como consecuencia, se producirá un incremento de ambigüedad léxica, lo que derivará también en una mayor ambigüedad sintáctica. Este componente ha sido perfectamente integrado en la cadena en desarrollo. Sin embargo, no ha sido adaptado al idioma español. Por lo tanto, su aplicación no aportará efecto alguno sobre el análisis de textos en castellano.

- **Métodos:**
  - `correct_errors(t:Set<String>): Set<String>`. Recibe los *tokens* obtenidos en la fase de segmentación y trata de buscar aquellos que tengan algún error ortográfico con el fin de corregirlo. Retorna la misma lista de palabras de entrada, corregidas aquellas que lo requerían.

7.3.7. Lexed

El lexicón LEFFE es un diccionario de rápido y fácil acceso con información morfosintáctica perfectamente organizada. Para permitirle a *SPMG_Lexer* acceder a dicha información de una manera sencilla, se ha utilizado el manejador de información léxica *Lexed*. Se trata de un recurso perteneciente al analizador léxico *SPMG_Lexer*, capaz de consultar la información recogida en el lexicón. Dicha información va a ser necesaria para que el analizador léxico lleve a cabo el proceso de lematización y etiquetación.

- **Métodos:**
  - `get_info(w:String): String`. Recupera la información morfosintáctica presente en el lexicón LEFFE para la palabra dada.

7.3.8. SPMG_Parser

Componente concreto que provee de análisis sintáctico al analizador del español. Este recurso es el resultado de un proceso de compilación sobre la metagramática SPMG detallada en el apartado 7.4 del *Manual Técnico*.

- **Métodos:**
  - `parse_sentence(t:String): String`. Método encargado de obtener la estructura sintáctica de una frase segmentada en sus unidades léxicas etiquetadas mediante información

morfosintáctica. Ofreciendo un bosque compartido de derivaciones que describe la estructura del texto de entrada.

7.3.9. Forest_utils

Componente encargado de manejar los diferentes formatos utilizados por los analizadores del servidor para ofrecer el procesamiento lingüístico realizado al usuario.

- **Métodos:**
  - `convert(t:String,f:String): String`. Convierte el bosque compartido de derivaciones resultante del análisis en el formato indicado.

7.3.10. ParserdClient, Callparser y WebParser

ParserdClient es la clase genérica que recoge el comportamiento común de los clientes del servidor *Parserd*. Existen dos clientes concretos que tratan de implementar las funcionalidades presente en la clase genérica: *Callparser* y *WebParser*. Mientras que *Callparser* utiliza línea de comandos para acceder al servidor vía *telnet*; *WebParser*, es una interfaz *web* que accede al servidor a través del protocolo HTTP.

Ambos clientes comparten funcionalidades similares:

- **Métodos:**
  - `parse_sentence(s:String): void`. Método encargado de la solicitud de análisis sobre una frase determinada. Previamente han de estar seleccionados el analizador y formato de salida para poder remitirle la petición al servidor. Éste ha de ser accionado mediante algún elemento gráfico de la interfaz en el caso de *WebParser*.
  - `parse_file(f:File): void`. Se encarga de manejar el procesamiento sobre las frase presentes en un fichero de texto. Para ello se recupera una a una las frases del fichero y se analizan de forma individual.
  - `show_parsers(): void`. Muestra los identificadores de los analizadores presentes en el servidor *Parserd*.
  - `show_formats(): void`. Muestra los formatos disponibles para ofrecer la salida del análisis.
  - `set_parser(p:String): void`. Indica cual es el analizador a utilizar en el análisis. Se indica mediante el identificador del analizador en cuestión.

- `set_format(f:String): void`. Selecciona un formato concreto de los disponibles para representar el resultado del análisis.

- `display_handler(r:String): void`. Se encarga de la visualización de los resultados obtenidos.

7.4. SPMG: Metagramática de la lengua española

SPMG recoge las clases que forman la metagramática del castellano. Dichas clases no presentan un estado y un comportamiento como puede ocurrir en otros lenguajes de programación orientados a objetos, sino que describen la topología arbórea y características de un fenómeno sintáctico concreto. Además, las clases de SPMG no estarán presentes en los *diagramas de secuencia* ya que no toman parte en los casos de uso del sistema, sino que simplemente depende de ellas el componente *SPMG Parser*.

Una metagramática describe jerarquías de clases, donde cada una de las clases hijas va a dar lugar a un árbol de la gramática GAR y, en donde, cada clase de esa jerarquía, va realizando una descripción parcial del fenómeno sintáctico recogido en el árbol resultante. Por descripción parcial de un fenómeno sintáctico se entiende, indicar la topología parcial del árbol, las condiciones o restricciones a mantener entre los nodos de ese árbol parcial, así como un fragmento de las características que van a aparecer en el *hypertag* correspondiente al ancla del árbol resultante de la descendencia de esa clase. Cada clase de la jerarquía aporta parte de esta información, para que la clase final describa el árbol completo de un fenómeno sintáctico concreto[9].

Además de la herencia, en la metagramática también aparece el concepto de *recurso*. Se trata de un mecanismo similar a la herencia para reutilizar las condiciones y características de otra clase en una clase local. Ello evita, al igual que la herencia, la redundancia de código. Por tanto, hay clases que proporcionan recursos y otras que los consumen. Una misma clase puede ser de ambos tipos.

Motivado por la gran cantidad de clases presentes en SPMG, se ha decidido encapsularlas en paquetes. De tal forma que, previamente se muestra el *diagrama de paquetes* en la figura 7.3 y después se detallan los *diagramas de clases* incluidos en cada paquete. Además, para cada clase se describe el fenómeno sintáctico que recoge, incluyendo ejemplos en caso de ser necesarios.

[9]Nótese que los árboles resultantes de la metagramática no van a estar todos presentes en la gramática GAR final, o al menos no con la misma estructura, ya que DyALog lleva a cabo un proceso de optimización, provocando la eliminación, fusión o reducción de muchos de ellos.

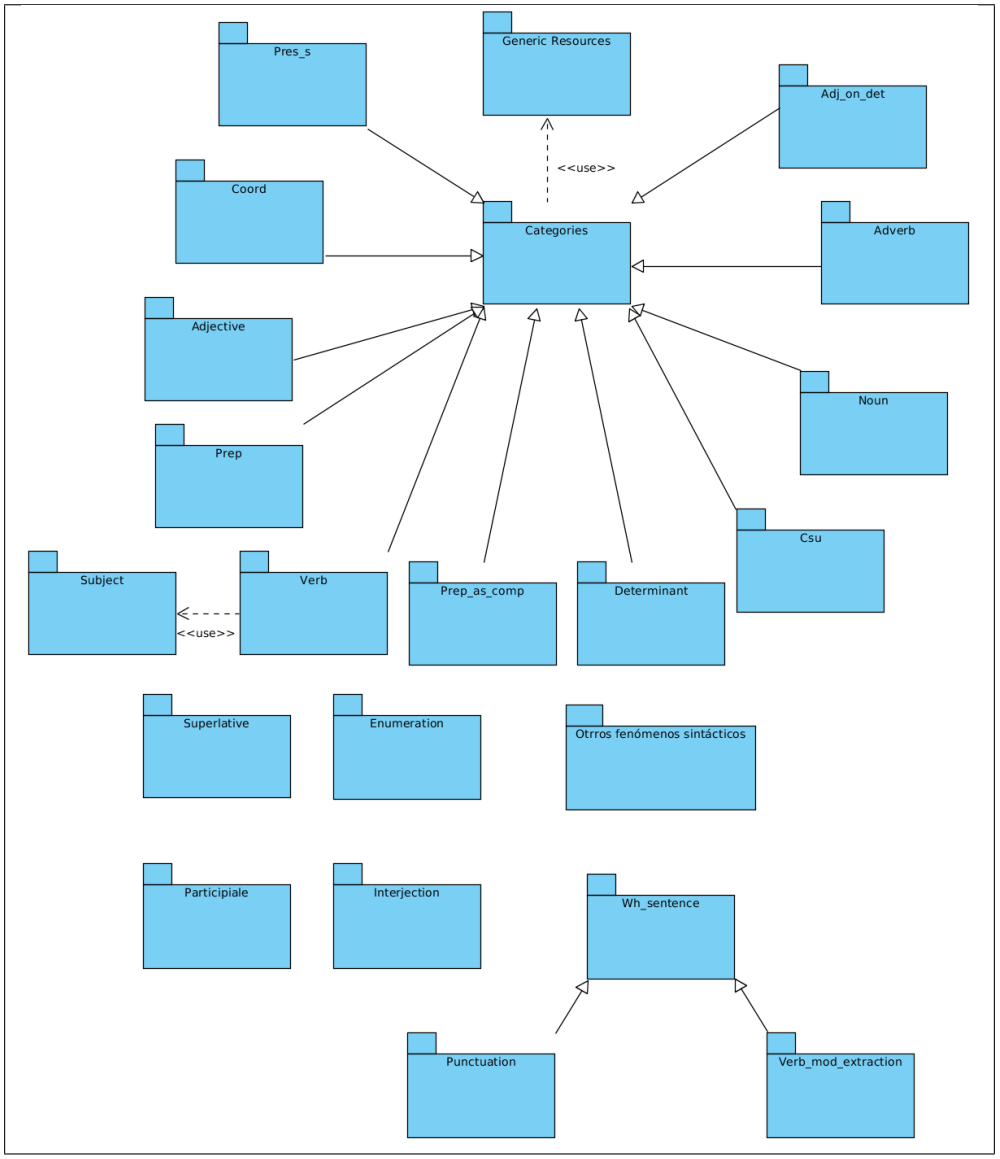


Figura 7.3: Diagrama de paquetes de SPMG.

7.4.1. Paquete *Categories*

Este paquete únicamente contiene la clase *categories* (figura 7.4) que define las características genéricas de una categoría léxica (nombre, adjetivo, adverbio, verbo, entre otras). Resaltar la condición que mantiene que todos los árboles resultantes de la jerarquía iniciada por esta clase tienen como categoría léxica de su ancla aquélla especificada en sus descendientes inmediatos. Por ejemplo, si la clase *adjective* hereda de *categories*, entonces

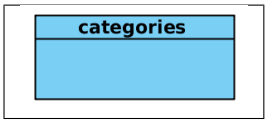


Figura 7.4: Diagrama de clases del paquete *Categories*.

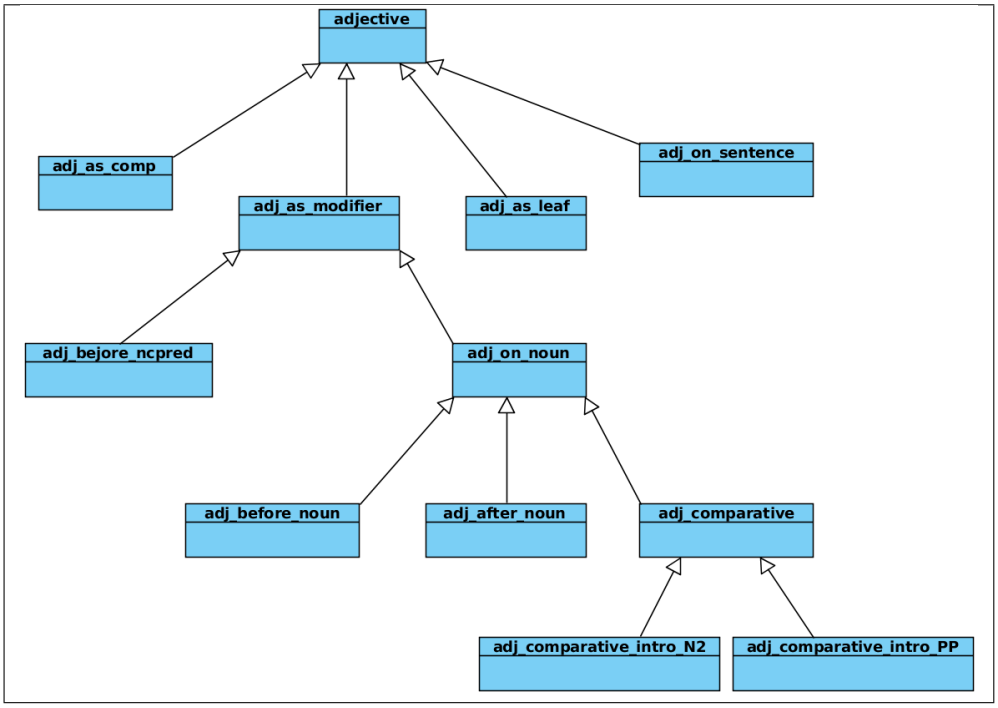


Figura 7.5: Diagrama de clases del paquete *Adjective*.

el adjetivo será la categoría léxica del ancla de los árboles resultantes de los descendientes de *adjective*.

7.4.2. Paquete *Adjective*

Este paquete recoge las clases encargadas de modelar el adjetivo y sus diferentes comportamientos sintácticos. Se compone de las clases presentes en la figura 7.5.

7.4.2.1. *adjective*

Recoge las características generales de un adjetivo. Hereda las características descritas por la clase *categories* ya que se trata de una categoría léxica. Por lo tanto, va a ser ancla de los árboles resultantes de

sus descendientes.

7.4.2.2. *adj_as_comp*

Describe el comportamiento de un adjetivo cuando funciona como atributo en una oración copulativa[10]. Por ejemplo, el adjetivo *"inteligente"* en la frase *"El niño es inteligente"*.

7.4.2.3. *adj_as_modifier*

Recoge el comportamiento de un adjetivo cuando éste es modificador de otra categoría léxica. Ello implica la existencia de restricciones de coordinación entre el modificador y lo modificado.

7.4.2.4. *adj_before_ncpred*

Se trata de un adjetivo modificador de un complemento predicativo. Además, el adjetivo se encuentra delante de lo modificado. Por ejemplo, el adjetivo *"plena"* en la oración *"Cogieron plena consciencia del problema"*.

7.4.2.5. *adj_on_noun*

Esta clase hereda de *adj_as_modifier*, pero concreta que la modificación se va a realizar sobre un sustantivo.

7.4.2.6. *adj_before_noun*

Se trata de un adjetivo modificador del nombre, y especifica que el adjetivo precede al sustantivo. Por ejemplo el adjetivo *"joven"* en la oración *"El joven hombre cruzó el valle"*.

7.4.2.7. *adj_after_noun*

Se trata de un adjetivo modificador del nombre, y especifica que el adjetivo se encuentra en posición posterior al sustantivo. Por ejemplo el adjetivo *"corpulento"* en la oración *"El hombre corpulento cruzó el valle"*.

7.4.2.8. *adj_comparative*

Esta clase ha sido diseñada para manejar el adjetivo comparativo *"mismo"* y sus diferentes formas flexionadas en la estructura comparativa *"mismo" + [sustantivo] + "que"*. El fenómeno sintáctico que describe se puede ver en el siguiente ejemplo: *"Tiene los mismos zapatos que su hermano"*.

[10] Oración con los verbos copulativos *ser*, *estar* o *parecer*.

7.4.2.9. *adj_comparative_intro_N2*

Concreta el comportamiento anterior, determinando que el adjetivo *"mismo"* o sus formas flexionadas acompañan a un sustantivo, como ocurría en el ejemplo anterior: *"Tiene los mismos zapatos que su hermano"*.

7.4.2.10. *adj_comparative_intro_PP*

Describe el fenómeno sintáctico cuando el adjetivo *"mismo"* o sus formas flexionadas introducen a un sintagma preposicional. Por ejemplo, *"Tiene los mismos de su hermano"*. En este caso desaparece la partícula comparativa *"que"*.

7.4.2.11. *adj_leaf*

Se trata de una clase hoja que no aporta ninguna información a mayores que la recogida en la clase padre *adjective*. Su función es la de proporcionar un árbol donde se defina el adjetivo de forma genérica. Las demás clases que heredaron y especializaron la clase *adjective*, únicamente son capaces de manejar las estructuras sintácticas concretas que describen. Sin embargo, el adjetivo puede estar presente en otras muchas construcciones sintácticas no recogidas en los casos anteriores, que podrán ser analizadas gracias a esta clase menos restrictiva. El árbol resultante de esta clase se utilizará particularmente en operaciones GAR de sustitución en la etapa de decoración, cuando el adjetivo no tenga un papel principal en la estructura objeto del análisis.

7.4.2.12. *adj_on_sentence*

Define la estructura sintáctica donde un adjetivo modifica a una oración. Por ejemplo, el adjetivo *"cansado"* en *"Cansado, abandonó la habitación"*.

7.4.3. Paquete *Determinant*

Este paquete recoge la clase *det* (figura 7.6) encargada de modelar el determinante. Se trata de una categoría léxica, que como tal hereda de la clase *categories*.

Figura 7.6: Diagrama de clases del paquete *Determinant*.

7.4.4. Paquete *Adj_ on_ det*

Este paquete recoge la clase *adj_ on_ det* (figura 7.7) encargada de modelar el comportamiento de ciertos determinantes que actúan como modificadores de otros determinantes. Se trata de una categoría léxica, que como tal hereda de la clase *categories*. A esta categoría pertenecería, por ejemplo, el lema *"todo"* y sus correspondientes formas flexionadas. Véase la forma *"todas"* en la oración *"Se recogieron todas las manzanas"*.

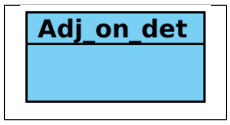


Figura 7.7: Diagrama de clases del paquete *Adj_ on_ det*.

7.4.5. Paquete *Noun*

Este paquete recoge las clases (figura 7.8) encargadas de definir las estructuras sintácticas donde el sustantivo (común o propio) o el pronombre son el ancla de las mismas.

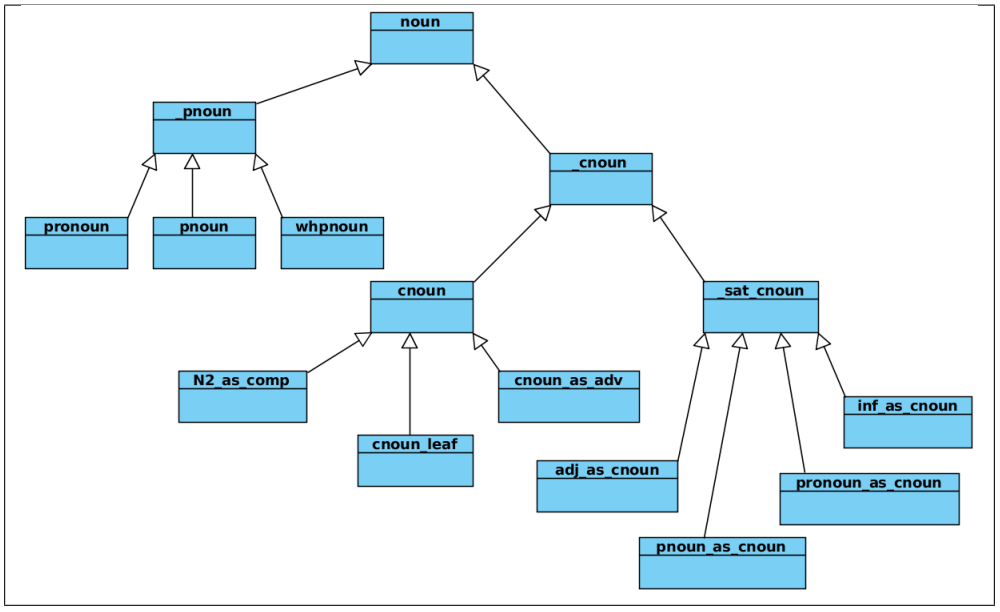


Figura 7.8: Diagrama de clases del paquete *Noun*.

7.4.5.1. *noun*

Simplemente marca el comienzo de la categoría léxica del sustantivo. Es el padre de todos los posibles fenómenos sintácticos en los que intervenga el sustantivo como núcleo o ancla. Sus dos descendientes inmediatos son las clases encargadas de modelar los nombres propios (*_pnoun*) y comunes (*_cnoun*).

7.4.5.2. *_pnoun*

Esta clase comienza la especialización de la categoría de los sustantivos centrándose en los nombres propios. Dentro de las estructuras sintácticas donde los nombres propios son anclas, comprende también los pronombres. Esto se debe a que tanto un nombre propio como un pronombre tiene comportamientos similares. Además, se distingue entre pronombre y pronombre interrogativo, puesto que las construcciones sintácticas en las que aparecen son diferentes[11].

7.4.5.3. *pnoun*

Esta clase concreta los nombres propios.

7.4.5.4. *pronoun*

Describe los pronombres dentro de las estructuras sintácticas propias de un nombre común. Esta clase contempla todos los pronombres excepto los interrogativos, pertenecientes a la siguiente clase.

7.4.5.5. *whpnoun*

Recoge los pronombres interrogativos. Por ejemplo, *"quién"* en la oración interrogativa *"¿Quién ha comido la tarta?"*.

7.4.5.6. *_cnoun*

Recoge las primeras características de los nombres comunes. Sus dos descendientes inmediatos, *cnoun* y *_sat_cnoun*, marcan dos caminos de herencia diferentes.

7.4.5.7. *cnoun*

Esta clase concreta la estructura sintáctica que acompaña a un sustantivo, es decir, modela un sintagma nominal donde el nombre es el

[11] Los pronombres interrogativos no aparecen en oraciones canónicas, sino que una oración interrogativa es considerada una extracción. Esto es, que los componentes de la oración no siguen el orden canónico.

núcleo del mismo.

7.4.5.8. *N2_as_comp*

Esta clase maneja la situación en la que un sintagma nominal tiene el papel de atributo en una oración copulativa. Por ejemplo, *"la autoridad"* en la oración *"La policía es la autoridad"*.

7.4.5.9. *cnoun_as_adv*

Describe un sintagma nominal como expresión adverbial de tiempo. Por ejemplo, el sintagma nominal *"el otro día"* en la oración *"Vimos a tu hermano el otro día"*.

7.4.5.10. *cnoun_leaf*

Al igual que en el caso de los adjetivos, es necesario mantener una clase hija, y por consiguiente un árbol en la gramática GAR, que describa un nombre común menos restrictivo que en el caso de los demás descendientes de *cnoun*. Además añade la posibilidad de que un nombre propio este acompañado de otro sustantivo (común o propio). Por ejemplo, los sustantivos común y propio *"presidente Zapatero"*, respectivamente, presentes en la oración *"El presidente Zapatero tomó la palabra"*.

7.4.5.11. *_sat_cnoun*

Esta clase modela el comportamiento sintáctico de un sintagma nominal *saturado*. Esto es, un grupo nominal donde el núcleo del mismo se encuentra precedido por un determinante. Además, también puede incluir opcionalmente otros componentes de una frase nominal como pueden ser, adjetivos o sintagmas preposicionales. Por ejemplo, *"el niño"* en la oración *"El niño está merendando"*. A partir de un sintagma nominal saturado se pueden concretar diferentes situaciones descritas en sus clases hijas.

7.4.5.12. *adj_as_cnoun*

Puesto que se trata de un sintagma nominal con un determinante (saturado), el núcleo del mismo (el sustantivo), puede ser sustituido por un adjetivo sustantivado. Por ejemplo, el grupo nominal *"el inteligente"* en la oración *"El inteligente acertó la respuesta"*.

7.4.5.13. *pnoun_as_cnoun*

Como ocurre en el fenómeno sintáctico anterior, también es posible sustituir el núcleo de un sintagma nominal saturado por un nombre propio.

Véase en el siguiente ejemplo el sintagma nominal *"la pequeña María"* en la oración *"La pequeña María vino a verte"*.

7.4.5.14. *pronoun_as_cnoun*

Asimismo, también es posible sustituir el sustantivo en una frase nominal saturada por un pronombre que haga la función de núcleo del mismo. Esto sucede en español con algún tipo de pronombres. Por ejemplo el sintagma nominal *"los otros"* en la oración *"Los otros pagan"*.

7.4.5.15. *inf_as_cnoun*

Otra posible configuración de un sintagma nominal saturado en SPMG es la de emplear como núcleo del mismo un verbo en modo infinitivo. Por ejemplo, la frase nominal *"el fumar"* en la oración *"El fumar mata"*.

7.4.6. Paquete *Verb*

El paquete de la figura 7.9 recoge las clases encargadas de definir las estructuras sintácticas del núcleo de la oración: el verbo.

7.4.6.1. *_verb_or_aux*

Se trata de la clase más genérica de la categoría léxica del verbo, y como tal, establece las bases de la misma. Sus descendientes inmediatos distinguen entre verbos y verbos auxiliares. Además hace uso de los recursos proveídos por la clase *clitics*, detallada a continuación.

7.4.6.2. *clitics*

Esta clase recoge el comportamiento de los clíticos nominativos, acusativos, dativos y reflexivos. Los clíticos son un grupo de pronombres personales que no pueden usarse independientemente, sino que necesitan ir acompañados de un verbo para tener sentido. A continuación, se exponen algunos ejemplos de los diferentes clíticos tratados:

- **Clíticos nominativos:** *"él"* en la oración *"la comida se la comió él"*. Un pronombre personal nominativo se convierte en clítico de un verbo cuando se encuentra en posición posverbal.
- **Clíticos acusativos:** *"lo"* en la oración *"lo recibe hoy"*.
- **Clíticos dativos:** *"le"* en la oración *"le vió el otro día"*.
- **Clíticos reflexivos:** *"se"* en la oración *"se vende este coche"*.

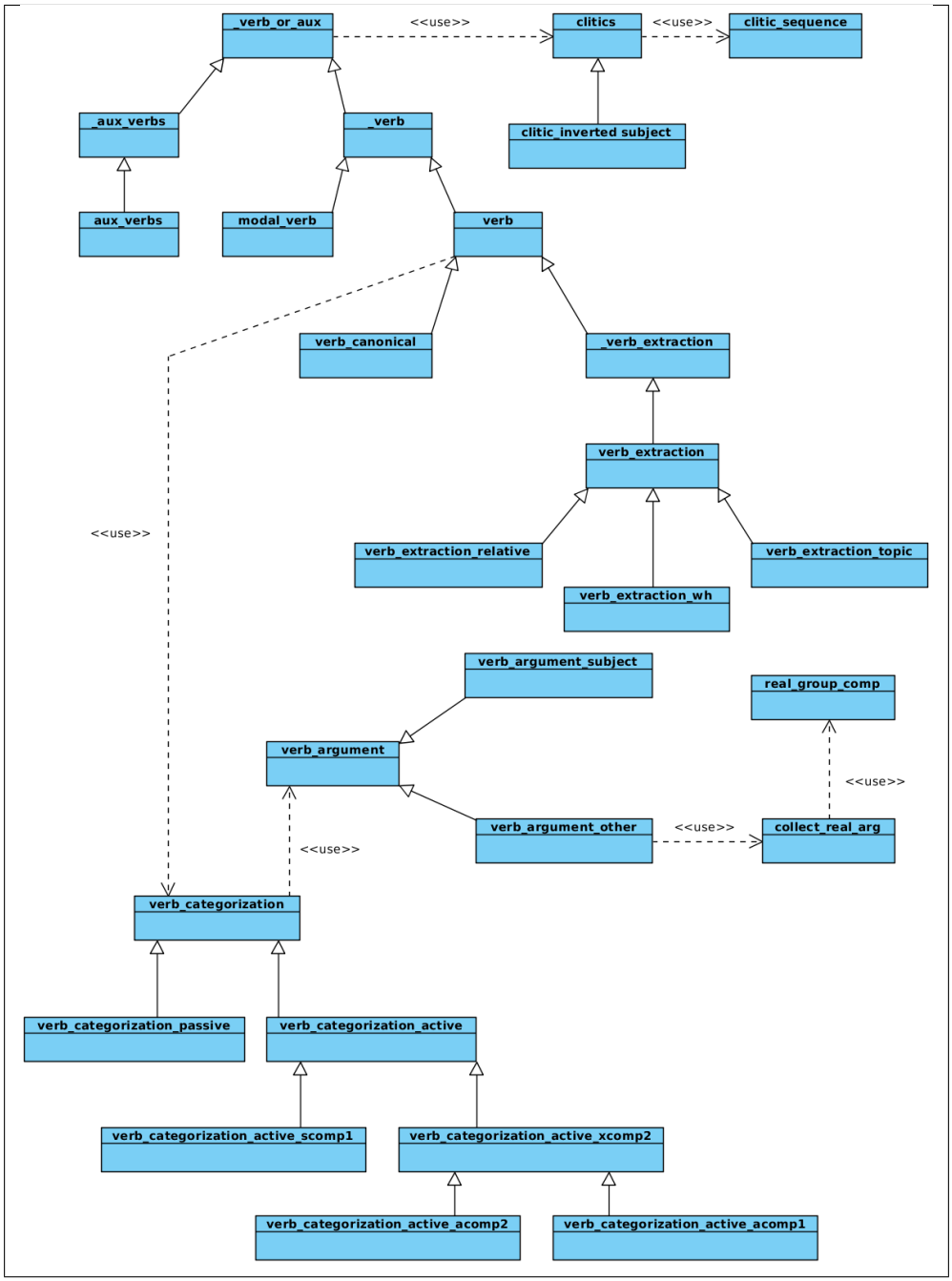


Figura 7.9: Diagrama de clases del paquete *Verb*.

7.4.6.3. *clitic_sequence*

Esta clase modela la colocación de los diferentes clíticos en el caso de acompañar al mismo verbo, así como la posición que ocuparía el adverbio de negación dentro de esa construcción. Por ejemplo, los clíticos *"se"* y *"lo"* en la oración *"No se lo dije todavía"*. En este ejemplo, queda reflejado que el adverbio de negación (*"no"*) debe preceder a la secuencia clítica y que el clítico dativo (*"se"*) precede al acusativo (*"lo"*). Las condiciones establecidas en esta clase son utilizadas como recurso por la clase *clitics*.

7.4.6.4. *clitic_inverted_subject*

Esta clase concreta la clase *clitics* para el caso de los clíticos nominativos posverbales actuando como sujetos de la oración. Se trata de un orden no canónico de la oración, ya que el verbo precede al sujeto.

7.4.6.5. *_aux_verbs* y *aux_verbs*

Ambas clases modelan los verbos auxiliares. Estos son los verbos *haber* y *estar* seguidos de participio o gerundio. Véase ejemplos de la estructura *"haber" + participio* y *"estar" + gerundio* en las oraciones *"Hemos entrado sigilosamente"* y *"Estoy cenando todavía"*, respectivamente.

7.4.6.6. *_verbs*

Es la clase común de los verbos no auxiliares que agrupa verbos modales y el resto de verbos.

7.4.6.7. *modal_verb*

Recoge el comportamiento de los verbos modales y las construcciones verbales donde aparecen. Estos son verbos que preceden al verbo principal en modo infinitivo. Algunos verbos modales son *deber, tener, poder, desear* y *querer*. Por ejemplo, el verbo modal *"deber"* en la oración *"Usted debe salir ahora"*.

7.4.6.8. *verb*

Se trata de la clase encargada de describir el verbo principal de la oración (ni modales ni auxiliares). Como núcleo de la oración, se encarga de estructurar todos los complementos de la misma, es decir, sujeto, objeto directo, objeto indirecto y el resto de modificadores verbales. Sus descendientes recogen las posibles organizaciones de los complementos del verbo.

7.4.6.9. *verb_ canonical*

Recoge el orden canónico de la oración, esto es, *Sujeto + Verbo + Objeto Directo*. El posicionamiento del resto de modificadores verbales no tiene importancia. Si el orden canónico es alterado se habla de *extracción del verbo*.

7.4.6.10. *_ verb_ extraction* y *verb_ extraction*

Describe las posibles extracciones del verbo en castellano. Dentro de SPMG, se entiende por extracción el no cumplimiento del orden canónico o la existencia de un fenómeno sintáctico no manejable en la estructura canónica. Existen 3 posibles extracciones verbales en SPMG y se describen a continuación.

7.4.6.11. *_ verb_ extraction_ wh*

En una oración interrogativa en ocasiones se produce un incumplimiento del orden canónico. Por tanto, estas se tratan como un tipo de extracción verbal. Por ejemplo, en la oración *"¿Qué libro ha leído Juan?"*, el objeto directo (*"qué libro"*) precede al verbo y el sujeto (*"Juan"*) se encuentra en posición posverbal no canónica.

7.4.6.12. *_ verb_ extraction_ topic*

Esta clase recoge el fenómeno sintáctico en donde se intercambia el sujeto por el objeto directo en el orden canónico. Por ejemplo, en la oración *raras son las personas ahí sentadas*, el atributo[12] precede al verbo mientras que el sujeto se encuentra en posición posverbal no canónica.

7.4.6.13. *_ verb_ extraction_ relative*

En SPMG las subordinadas de relativo se manejan como un tipo de extracción, puesto que éstas, a diferencia de las subordinadas sustantivas, pueden aparecer en diferentes posiciones de la estructura verbal sin tener función de sujeto u objeto directo[13]. Un ejemplo de subordinada de relativo es la cláusulas *"que vimos ayer"* en la oración *"Ese es el hombre que vimos ayer"*[14].

7.4.6.14. *verb_ categorization*

La clase *verb* utiliza los recursos proveídos por la clase *verb_ categorization*. En la *hypertag* correspondiente a un árbol donde el verbo

[12] No es un objeto directo ya que se trata de una oración copulativa.

[13] Funciones relegadas a las subordinadas sustantivas.

[14] Una subordinada de relativo se comporta como un adjetivo por ello también son conocidas como subordinadas adjetivas.

es el ancla[15] se incluye el *marco de subcategorización* del verbo. Por *marco de categorización* se entiende los argumentos que acompañan al verbo. En SPMG, un verbo únicamente dispone de 3 posibles argumentos como máximo. Estos pueden ser: el sujeto, el objeto directo o cualquier otro modificador del verbo. Se distinguen principalmente dos tipos de subcategorización o redistribuciones: activa y pasiva. El recurso que provee esta clase no se utiliza para describir las estructuras sintácticas en las que aparecen los complementos del verbo, sino que su uso está destinado a describir el *hypertag* del árbol resultante.

7.4.6.15. *verb_categorization_active*

Modela la subcategorización verbal en voz activa. Para esta situación se mantiene que el primer argumento del verbo va a ser el sujeto (o sujeto elíptico), el segundo argumento no puede ser sujeto, y el tercer argumento no puede ser ni sujeto ni objeto. En la figura 7.10 se puede observar cuál sería el *hypertag* resultante para el árbol verbal `#111` con voz (diátesis) activa. Nótese que los valores para un mismo argumento del verbo pueden ser varios[16].

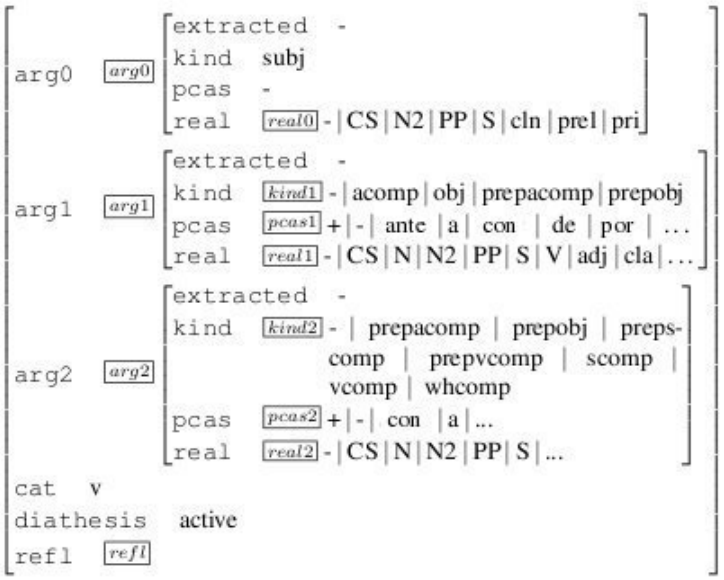


Figura 7.10: *Hypertag* del árbol `#111`.

Además se crean diferentes clases hijas que especializan el manejo de la

[15] Lo que significa que ese árbol se ha obtenido a partir de una clase hija del paquete *Verb*.

[16] El formato de una *hypertag* ya ha sido detallado en el apartado 2.3.2 de la *Memoria*.

subcategorización activa.

7.4.6.16. *verb_categorization_active_scomp1*

Describe la subcategorización verbal en oraciones activas cuando el segundo argumento del verbo es un complemento infinitivo[17], un complemento preposicional con infinitivo[18] o una subordinada sustantiva.

7.4.6.17. *verb_categorization_active_xcomp2*

Modela la subcategorización verbal en oraciones que no presentan un segundo argumento con los complementos indicados en la clase anterior. A su vez, se realiza una segunda diferenciación dentro de esta clase: *verb_categorization_active_acomp1* y *verb_categorization_active_acomp2*

7.4.6.18. *verb_categorization_active_acomp1*

Esta clase describe todos aquellos marcos de subcategorización verbal de voz activa donde el tercer argumento no es atributo de una oración copulativa.

7.4.6.19. *verb_categorization_active_acomp2*

Esta clase describe únicamente aquellos marcos de subcategorización verbal de voz activa donde el segundo argumento no es atributo de una oración copulativa, pero el tercer argumento sí lo es.

7.4.6.20. *verb_categorization_passive*

Del mismo modo que ocurre para la voz activa, sucede con la voz pasiva. En este caso, el argumento cero o primer argumento es el complemento agente, mientras que el sujeto es el segundo argumento. Se trata de una estrategia comprensible, ya que el complemento agente es el que realiza la acción en la oración pasiva y es distinto del sujeto de la misma, que es quien la recibe.

7.4.6.21. *verb_argument*

La clase *verb_categorization* emplea los recursos ofrecidos por esta clase. Su función es la de describir los atributos de los argumentos verbales en la *hypertag*. Como se puede ver en la figura 7.10, cada argumento dispone de una serie de atributos (`extracted`, `kind`, `pcas` y `real`) que se detallan

[17] Verbo en modo infinitivo que acompaña a un verbo modal.
[18] Verbo en modo infinitivo con la estructura *preposición + infinitivo*.

en el apartado 2.3.2 de la *Memoria*. Esta clase tiene dos especificaciones: *verb_ argument_ subject* y *verb_ argument_ other*.

7.4.6.22. *verb_ argument_ subject*

Modela los argumentos del verbo que actúan como sujeto en la oración.

7.4.6.23. *verb_ argument_ other*, *collect_ real_ arg* y *real_ group_ comp*

verb_ argument_ other modela cualquier argumento del verbo que no actúe como sujeto en la oración. Además hace uso de los recursos proveídos por la clase *collect_ real_ arg*. Ésta trata de describir cual son los posibles valores que puede tomar un argumento no sujeto y las restricciones que sobre ellos pesan. *collect_ real_ arg*, a su vez, hace uso de los recursos de *real_ group_ comp*. Esta última aporta a la primera información acerca de los complementos verbales que pueden ocupar el puesto de argumento verbal.

7.4.7. Paquete *Verb_ mod_ extraction*

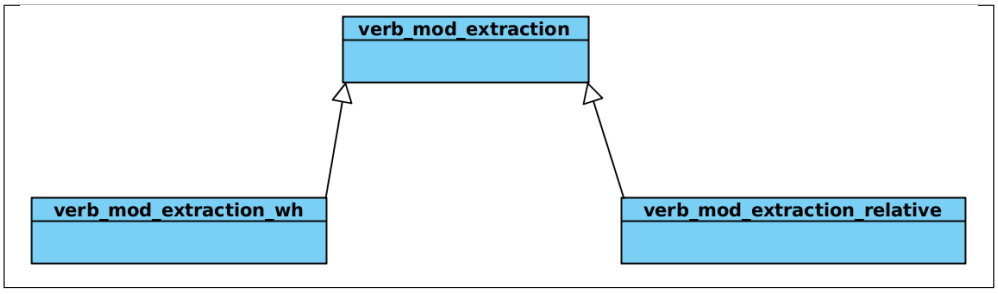


Figura 7.11: Diagrama de clases del paquete *Verb_ mod_ extraction*.

verb_ mod_ extraction es la clase principal del paquete *Verb_ mod_ extraction* descrito en la figura 7.11. Su finalidad es la de modelar las extracciones de la estructura sintáctica oracional. SPMG tiene distintas formas de modelar los modificadores o complementos del verbo. Una de ellas, expuesta anteriormente, es manejar los complementos del verbo como argumentos del mismo que rompen el orden canónico. Sin embargo, con este planteamiento no se puede hacer frente al gran número de modificadores verbales que pueden aparecer y a todas las posiciones en donde pueden situarse. En este sentido, se ha diseñado esta clase. La idea es describir los modificadores verbales, no como argumentos sujetos a la estructura rígida del verbo, sino como complementos independientes que se encuentran a un nivel de estructura oracional. Además, se especializa para

los casos de extracciones producidas en oraciones interrogativas y provocadas por subordinadas de relativo en las clases *verb_mod_extraction_wh* y *verb_mod_extraction_relative*, respectivamente.

7.4.8. Paquete *Prep_as_comp*

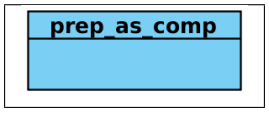


Figura 7.12: Diagrama de clases del paquete *Prep_as_comp*.

Este paquete únicamente contiene una clase identificada bajo el mismo nombre (figura 7.12). Su finalidad es describir el fenómeno sintáctico donde un sintagma preposicional con infinitivo (*preposición + infinitivo*) actúa como atributo de una oración copulativa. Por razones de funcionalidad se ha considerado una categoría léxica, siendo la preposición el ancla del árbol resultante. Por ejemplo, el complemento *"de fiar"* en la oración *"Ese hombre es de fiar"*.

7.4.9. Paquete *Prep*

Este paquete, detallado en la figura 7.13, recoge las clases encargadas de definir las estructuras sintácticas donde la preposición es el ancla de las mismas. En la mayoría de los casos se estará hablando de sintagmas preposicionales. Estos son, aquellos sintagmas introducidos por una preposición.

7.4.9.1. *prep*

Modela las principales bases de las preposiciones y de los sintagmas preposicionales, indicando para estos últimos que tipos de sintagmas pueden formar parte de ellos.

7.4.9.2. *prep_argument*

Tiene el mismo cometido que las clases *leaf* comentadas anteriormente para el adjetivo y el sustantivo: hacer llegar a un árbol resultante la información genérica detallada en la clase padre *prep*, creando de esta forma una estructura idónea para la etapa de decoración del análisis.

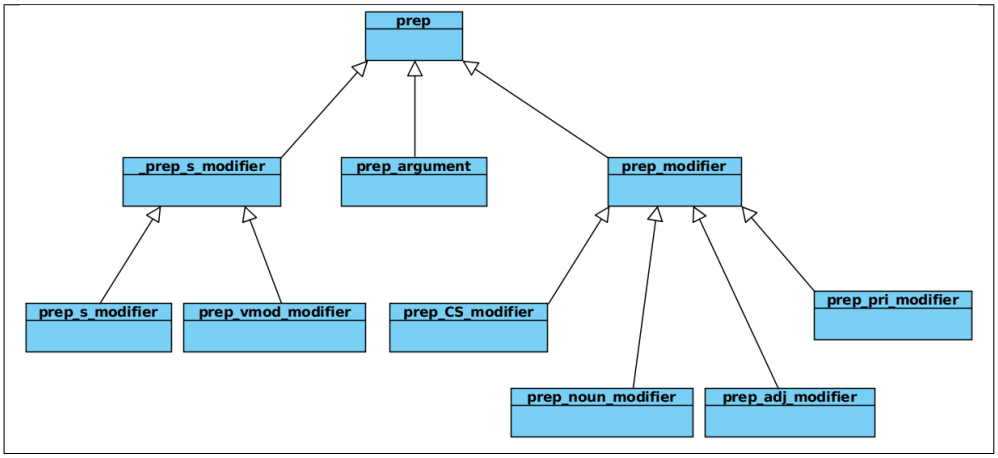


Figura 7.13: Diagrama de clases del paquete *Prep*.

7.4.9.3. *prep_ modifier*

Esta clase especializa la preposición como un modificador o acompañante de diferentes sintagmas o categorías léxicas: subordinadas sustantivas, adjetivos, sustantivos o pronombres interrogativos.

7.4.9.4. *prep_ CS_ modifier*

La preposición modifica a una subordinada sustantiva. Por ejemplo, la preposición *"de"* y la subordinada sustantiva *"que vinieses"* en la oración *"Me alegro de que vinieses"*.

7.4.9.5. *prep_ noun_ modifier*

La preposición acompaña a un sustantivo. Se trata del sintagma preposicional básico formado por una preposición más un sintagma nominal. Por ejemplo, el sintagma preposicional *"a esa casa"* en la oración *"Fueron a esa casa"*.

7.4.9.6. *prep_ adj_ modifier*

La preposición acompaña a un adjetivo. Por ejemplo, la preposición *"de"* y el adjetivo *"impresentables"* en la oración *"Ese edificio está lleno de impresentables"*.

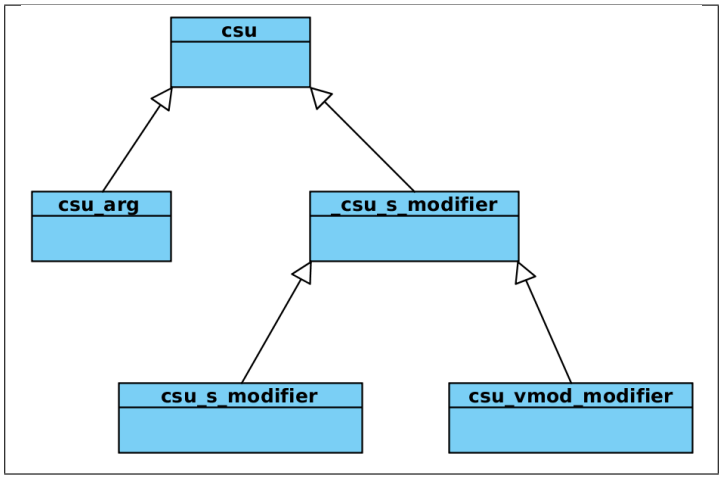


Figura 7.14: Diagrama de clases del paquete *Csu*.

7.4.9.7. *prep_pri_modifier*

La preposición acompaña a un pronombre interrogativo. Por ejemplo, la preposición *"de"* y el pronombre interrogativo *"quién"* en la oración *"¿De quién es esta cartera?"*.

7.4.10. Paquete *Csu*

Este paquete (figura 7.14) recoge las clases encargadas de definir las estructuras sintácticas donde la conjunción subordinada es el ancla de las mismas. En la mayoría de los casos se estará hablando de subordinadas sustantivas o adverbiales. Éstas son, aquellas oraciones introducidas por una conjunción subordinada.

7.4.10.1. *csu*

Modela las principales bases de las conjunciones subordinadas y de las subordinadas sustantivas o adverbiales, indicando para estas últimas que tipo de construcciones son posibles.

7.4.10.2. *csu_argument*

Tiene el mismo cometido que las clases *leaf* en los adjetivos y sustantivos: hacer llegar a un árbol resultante la información genérica detallada en la clase padre *csu*. Aunque a mayores, indica que la conjunción subordinada *"que"* es la única que puede aparecer en las construcciones fruto de esta clase.

7.4.10.3. *_csu_s_modifier*

Describe el fenómeno sintáctico donde una subordinada sustantiva o adverbial es introducida por una conjunción subordinada[19]. Por ejemplo, la conjunción subordinada *"que"* y la clausula *"vinieses"* componen una subordinada sustantiva actuando como sujeto en la oración *"Me alegra que vinieses"*.

7.4.10.4. *csu_s_modifier*

Concreta el fenómeno anterior para manejar los casos en los que estén presentes signos de puntuación que denoten incisos de tiempo en la lectura de una oración: comas, paréntesis, guiones, ... En el ejemplo anterior, se podría haber realizado el siguiente cambio de posición en los complementos: *"Que vinieses, me alegra"*. Esta clase es capaz de manejar los signos de puntuación que marcan incisos en el caso de que la subordinada se encuentre al inicio de la frase.

7.4.10.5. *csu_vmod_modifier*

Si por el contrario, la subordinada sustantiva es una aclaración incluida dentro de una oración entre signos de inciso, entonces será esta clase la encargada de modelar ese fenómeno. Especialmente, se utiliza para manejar las subordinadas adverbiales, que al igual que los adverbios, se manejan como modificadores verbales que pueden aparecer en cualquier parte de la frase. Por ejemplo, la subordinada adverbial *"cuando se quedo sin combustible"*, donde *"cuando"* es una conjunción subordinada de tiempo, en la oración *"El coche, cuando se quedó sin combustible, se detuvo"*.

7.4.11. Paquete *Adverb*

Este paquete (figura 7.15) se encarga de modelar la categoría léxica del adverbio y recoge las clases encargadas de definir las estructuras sintácticas donde éste es el ancla de las mismas.

7.4.11.1. *adv*

Sienta las bases de la categoría léxica del adverbio.

7.4.11.2. *adv_leaf*

Tiene el mismo cometido que las clases *leaf* en los adjetivos y sustantivos: hacer llegar a un árbol resultante la información genérica detallada en la clase

[19] Nótese que las subordinadas de relativo no están introducidas por una conjunción subordinada, sino por un pronombre relativo.

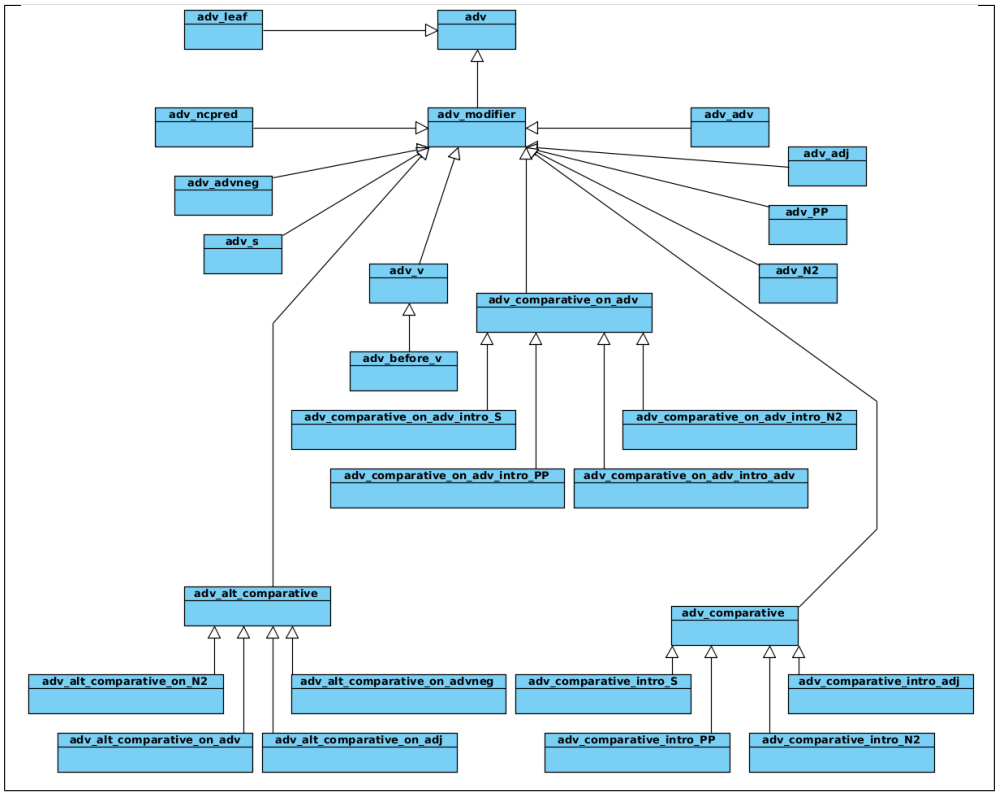


Figura 7.15: Diagrama de clases del paquete *Adverb*.

padre *adv*.

7.4.11.3. *adv_ modifier*

Describe las estructuras y condiciones sintácticas generales en las que el adverbio realiza su principal cometido: ser un modificador. Será especializada por todos sus descendientes en la jerarquía.

7.4.11.4. *adv_ ncpred*

Maneja la estructura sintáctica donde un adverbio modifica a un complemento predicativo. Por ejemplo, el adverbio *"muy"* y el complemento predicativo *"alegres"* en la oración *"Los niños pasean muy alegres"*.

7.4.11.5. *adv_ advneg*

Cuando un adverbio modifica a un adverbio de negación. Por ejemplo, el adverbio *"definitivamente"* y el adverbio de negación *"no"* en la oración *"Definitivamente no puede hacerlo"*.

7.4.11.6. *adv_ s*

Describe las estructuras sintácticas cuando un adverbio modifica a una oración. Por ejemplo, el adverbio *"hoy"* en la oración *"Hoy, el señor se fue a su casa"*.

7.4.11.7. *adv_ v*

Maneja el papel modificador del adverbio más común, acompañando a un verbo. Por ejemplo, el adverbio *"poco"* en la oración *"Durmieron poco"*[20].

7.4.11.8. *adv_ before_ v*

Concreta el caso del adverbio como modificador y predecesor del verbo. Por ejemplo, el adverbio *"poco"* en la oración *"Poco comieron hoy"*.

7.4.11.9. *adv_ adv*

Cuando un adverbio es modificador de otro adverbio. Por ejemplo, el adverbio *"muy"* en la oración *"Se alejaron muy rápidamente"*.

[20] Aunque a nivel gramatical el adverbio esté modificando al verbo en la clase *adv_ s*, se maneja a nivel oracional porque no se encuentra en las proximidades del verbo y, por lo tanto, no puede ser recogido dentro de sus estructuras sintácticas.

7.4.11.10. *adv_adj*

Cuando un adverbio es modificador de un adjetivo. Por ejemplo, el adverbio *"muy"* en la oración *"Esa chica es muy guapa"*.

7.4.11.11. *adv_PP*

Cuando un adverbio es modificador de un sintagma preposicional. Por ejemplo, el adverbio *"muy"* en la oración *"Ellos son muy de fiar"*.

7.4.11.12. *adv_N2*

Cuando un adverbio es modificador de un sintagma nominal. Por ejemplo, el adverbio *"aproximadamente"* en la oración *"Entraron cinco personas aproximadamente"*.

7.4.11.13. *adv_comparative_on_adv*

Modela la estructura sintáctica de la construcción comparativa adverbial (*adverbio de comparación* + *adverbio* +*"que"/"de"/"como"*). Por ejemplo, el adverbio *"rápidamente"* dentro de la estructura comparativa *"más rápidamente que"* en la oración *"Terminó más rápidamente que tú"*.

7.4.11.14. *adv_comparative_on_adv_intro_S*

Se especializa el comportamiento de la estructura comparativa adverbial cuando introduce a una oración. Por ejemplo, la oración *"Más rápidamente que Pablo no lo hace nadie"*.

7.4.11.15. *adv_comparative_on_adv_intro_PP*

Se especializa el comportamiento de la estructura comparativa adverbial cuando introduce a un sintagma preposicional. Por ejemplo, la oración *"Lo hizo más rápidamente que con su ayuda"*.

7.4.11.16. *adv_comparative_on_adv_intro_N2*

Se especializa el comportamiento de la estructura comparativa adverbial cuando introduce a un sintagma nominal. Por ejemplo, la oración *"Terminó más rápidamente que sus hermanos"*.

7.4.11.17. *adv_comparative_on_adv_intro_adv*

Se especializa el comportamiento de la estructura comparativa adverbial cuando introduce a un adverbio. Por ejemplo, la oración *"Actúo más brutalmente que inteligentemente"*.

7.4.11.18. *adv_comparative*

Modela las oraciones comparativas (con la estructura *adverbio de comparación + adjetivo + "que"/"de"/"como"*). Por ejemplo, el adjetivo *"alto"* dentro de la estructura comparativa *"más alto que"* en la oración *"Es más alto que tú"*.

7.4.11.19. *adv_comparative_intro_S*

Se especializa el comportamiento de la estructura comparativa con adjetivo cuando introduce a una oración. Por ejemplo, la oración *"Más roja que esta flor no la vas a encontrar"*.

7.4.11.20. *adv_comparative_intro_PP*

Se especializa el comportamiento de la estructura comparativa con adjetivo cuando introduce a un sintagma preposicional. Por ejemplo, la oración *"Estás más bella que de costumbre"*.

7.4.11.21. *adv_comparative_intro_N2*

Se especializa el comportamiento de la estructura comparativa con adjetivo cuando introduce a un sintagma nominal. Por ejemplo, la oración *"Es más bella que su hermana"*.

7.4.11.22. *adv_comparative_intro_adj*

Se especializa el comportamiento de la estructura comparativa con adjetivo cuando introduce a un adjetivo. Por ejemplo, la oración *"Es más roja que azul"*.

7.4.11.23. *adv_alt_comparative*

Modela la estructura comparativa (*adverbio de comparación + "que"/"de"/"como"*). Por ejemplo, la estructura comparativa *"más" + "que"* en la oración *"Por más que corrieses, no llegarías"*.

7.4.11.24. *adv_alt_comparative_on_adv*

Se especializa el comportamiento de la estructura comparativa sin adverbio ni adjetivo cuando introduce a un adverbio. Por ejemplo, la oración *"Comieron más que rápidamente"*.

7.4.11.25. *adv_alt_comparative_on_N2*

Se especializa el comportamiento de la estructura comparativa sin adverbio ni adjetivo cuando introduce a un sintagma nominal. Por ejemplo, la oración *"Él es más que un amigo"*.

7.4.11.26. *adv_alt_comparative_on_adj*

Se especializa el comportamiento de la estructura comparativa sin adverbio ni adjetivo cuando introduce a un adjetivo. Por ejemplo, la oración *"Él es más que pobre"*.

7.4.11.27. *adv_alt_comparative_on_advneg*

Se especializa el comportamiento de la estructura comparativa sin adverbio ni adjetivo cuando introduce a un adverbio de negación. Por ejemplo, la oración *"Eso es más que nada"*.

7.4.12. Paquete *Participiale*

Este paquete (figura 7.16) se encarga de modelar las posibles construcciones sintácticas donde aparezca un participio actuando como un adjetivo o una subordinada introducida por participio o gerundio.

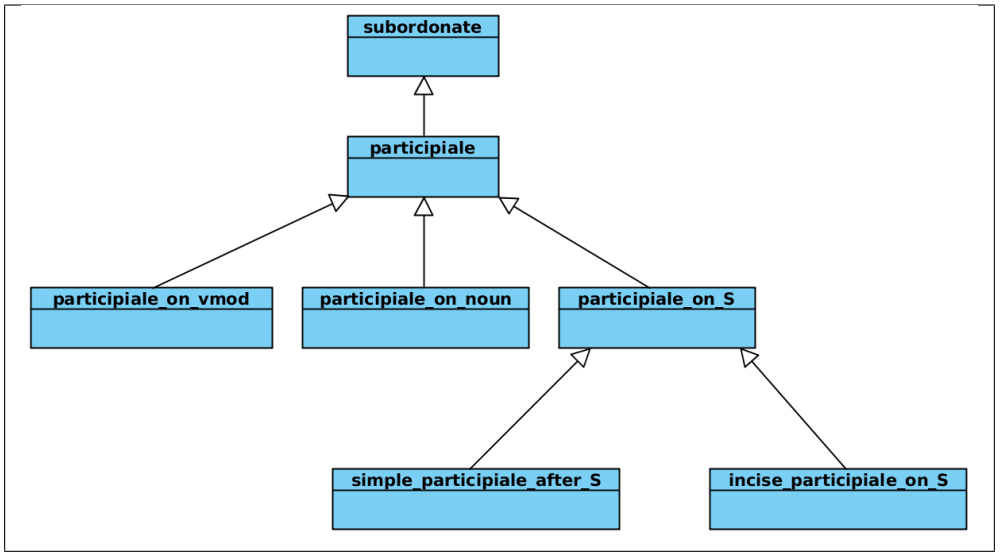


Figura 7.16: Diagrama de clases del paquete *Participiale*.

7.4.12.1. *subordonate*

Esta clase modela las oraciones subordinadas introducidas por participio o gerundio. Por ejemplo, la subordinada introducida por participio *"interrumpida el jueves"* en la oración *"Declaró reanudada la reunión, interrumpida el jueves"*.

7.4.12.2. *participiale*

Amplía la información heredada, sentando las bases de un participio o una subordinada con participio o gerundio como modificador de ciertas categorías léxicas que se detallan a continuación.

7.4.12.3. *participiale_on_noun*

Cuando un participio modifica a un sustantivo. Por ejemplo, el sustantivo *"tema"* y el participio *"cantado"* en la oración *"Ese es el tema cantado"*.

7.4.12.4. *participiale_on_S*

Cuando un participio o subordinada introducida por participio o gerundio modifica a una oración. Por ejemplo, la oración anteriormente mencionada *"Declaró reanudada la reunión, interrumpida el jueves"*.

7.4.12.5. *simple_participiale_after_S*

Esta clase se encarga de modelar los participios o gerundios posteriores a un verbo auxiliar[21]. Por ejemplo, el gerundio *"comiendo"* en la oración *"Ahora están comiendo"*.

7.4.12.6. *incise_participiale_on_S*

Se trata de la clase concreta que posibilita que las subordinadas introducidas por participios o gerundios se encuentren en cualquier posición de la oración. Para ello, es necesario que maneje los signos de inciso (comas, paréntesis, guiones, etc) como ocurría en el ejemplo *"Declaró reanudada la reunión, interrumpida el jueves"*.

7.4.12.7. *participiale_on_vmod*

Reúne unas características similares al caso anterior, donde participio, gerundio o subordinada acompañaba a una oración; pero en esta clase se

[21] En ocasiones se entienden los modificadores de una oración como modificadores del verbo, siempre y cuando, el modificador se encuentre próximo al verbo en la oración, como es el caso de los participios y gerundios que acompañan a un verbo auxiliar.

manejan modificadores verbales que no se encuentren en las estructuras próximas al verbo.

7.4.13. Paquete *Punctuation*

Este paquete (figura 7.17) se encarga de modelar las posibles construcciones sintácticas donde aparezca un signo de puntuación.

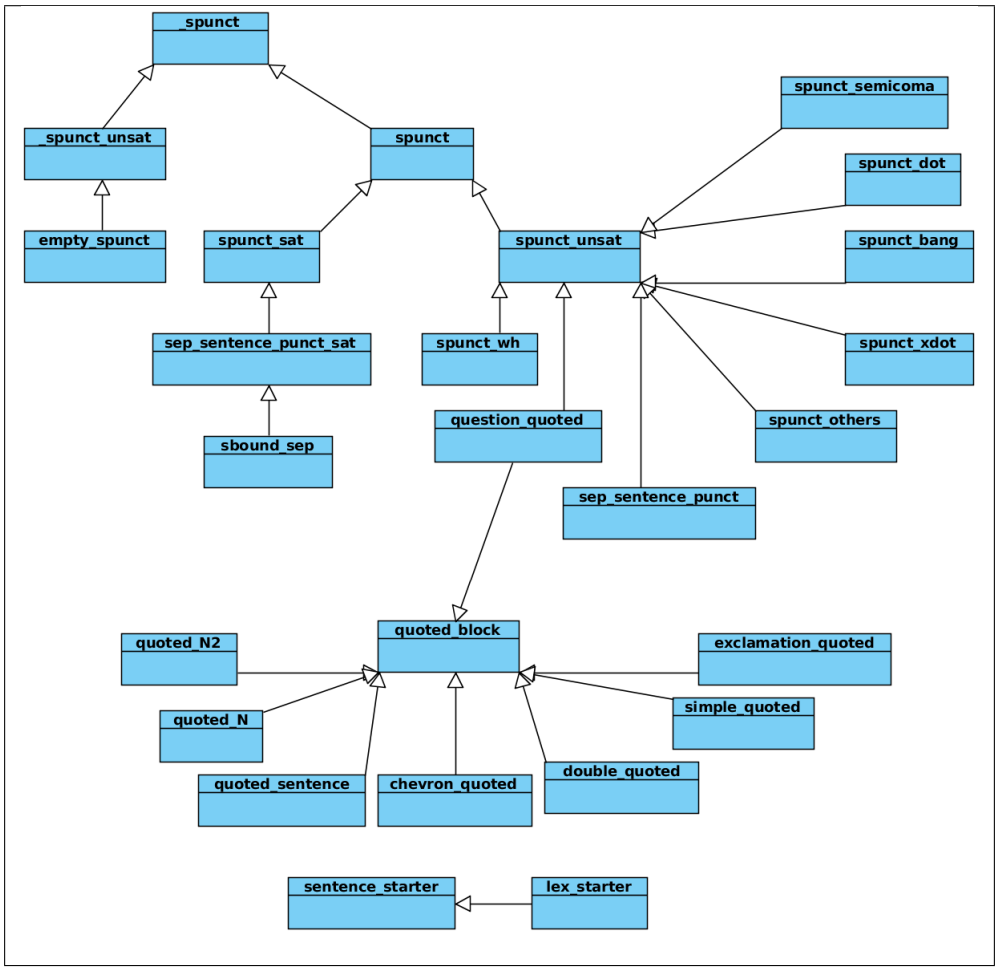


Figura 7.17: Diagrama de clases del paquete *Punctuation*.

7.4.13.1. *_spunct*

Establece las bases de las estructuras sintácticas donde aparecen signos de puntuación.

7.4.13.2. *_spunct_unsat*

Hereda las características de un signo de puntuación, pero concreta que los signos que recoge esta clase no son saturados. Un signo de puntuación es saturado cuando marca el final de una oración, por ejemplo un punto final. Una coma sería un signo no saturado.

7.4.13.3. *empty_spunct*

Esta clase se emplea para marcar el final de oraciones sin puntuación final. Si al sistema le es solicitado el análisis de una oración incompleta o sin puntuación final, esta clase asigna un nodo final para marcar el fin del texto a analizar. De lo contrario, cualquier frase sin un punto final, se tacharía de agramatical.

7.4.13.4. *spunct*

Esta clase va a comenzar una jerarquía de clases donde se manejará el léxico concreto para cada puntuación. Sus descendientes inmediatas distinguen entre puntuación saturada y no saturada.

7.4.13.5. *spunct_sat*, *sep_sentence_punct_sat* y *sbound_sep*

Este fragmento de la jerarquía se encarga de controlar las construcciones con puntuación saturada.

7.4.13.6. *spunct_unsat*

Maneja la puntuación no saturada. Sus descendientes describen los siguientes signos de puntuación:

- ***spunct_unsat*:** Maneja el punto y coma (;).
- ***spunct_dot*:** Maneja el punto (.). En algunas ocasiones, como en las abreviaturas, este signo no marca el final de una oración.
- ***spunct_bang*:** Maneja el signo de exclamación final (!).
- ***spunct_wh*:** Maneja el signo de interrogación final (?).
- ***spunct_others*:** Maneja otros signos de puntuación como: *!!*, *!!!*, *!?*, *?!*, *!?!?*, *?!?!*, ...

7.4.13.7. *quoted_block*

Controla aquellos signos de puntuación que acotan las oraciones o fragmentos de ellas. Sus descendientes describen los siguientes signos de acotación:

- ***question_quoted*:** Describe la estructura acotada entre los signos de interrogación.
- ***exclamation_quoted*:** Describe la estructura acotada entre los signos de exclamación.
- ***simple_quoted*:** Describe la estructura acotada entre las comillas simples.
- ***double_quoted*:** Describe la estructura acotada entre las comillas dobles.
- ***chevron_quoted*:** Describe la estructura acotada entre los signos *"«"* y *"»"*.

7.4.13.8. *quoted_N2*, *quoted_N* y *quoted_sentence*

Maneja la acotación de los sintagmas nominales, los nombres y las oraciones, respectivamente.

7.4.13.9. *sentence_starter* y *lex_starter*

Manejan oraciones que comienzan por signos de puntuación tales como: -, *, +, ...

7.4.14. Paquete *Coord*

Este paquete, detallado en la figura 7.18, se encarga de modelar la categoría léxica de las conjunciones coordinantes y recoge las clases encargadas de definir las estructuras sintácticas donde éstas son el ancla de las mismas.

7.4.14.1. *coord*

Clase raíz de la jerarquía que establece las características y restricciones base de una conjunción coordinante. Sus descendientes inmediatos manejan las construcciones sintácticas donde una conjunción coordinante une dos o más categorías léxicas o sintagmas.

7.4.14.2. *PP_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es un sintagma preposicional. Por ejemplo, la conjunción coordinante *"y"* y los sintagmas preposicionales *"de chocolate"* y *"de nata"* en la oración *"Está cubierto de chocolate y de nata"*.

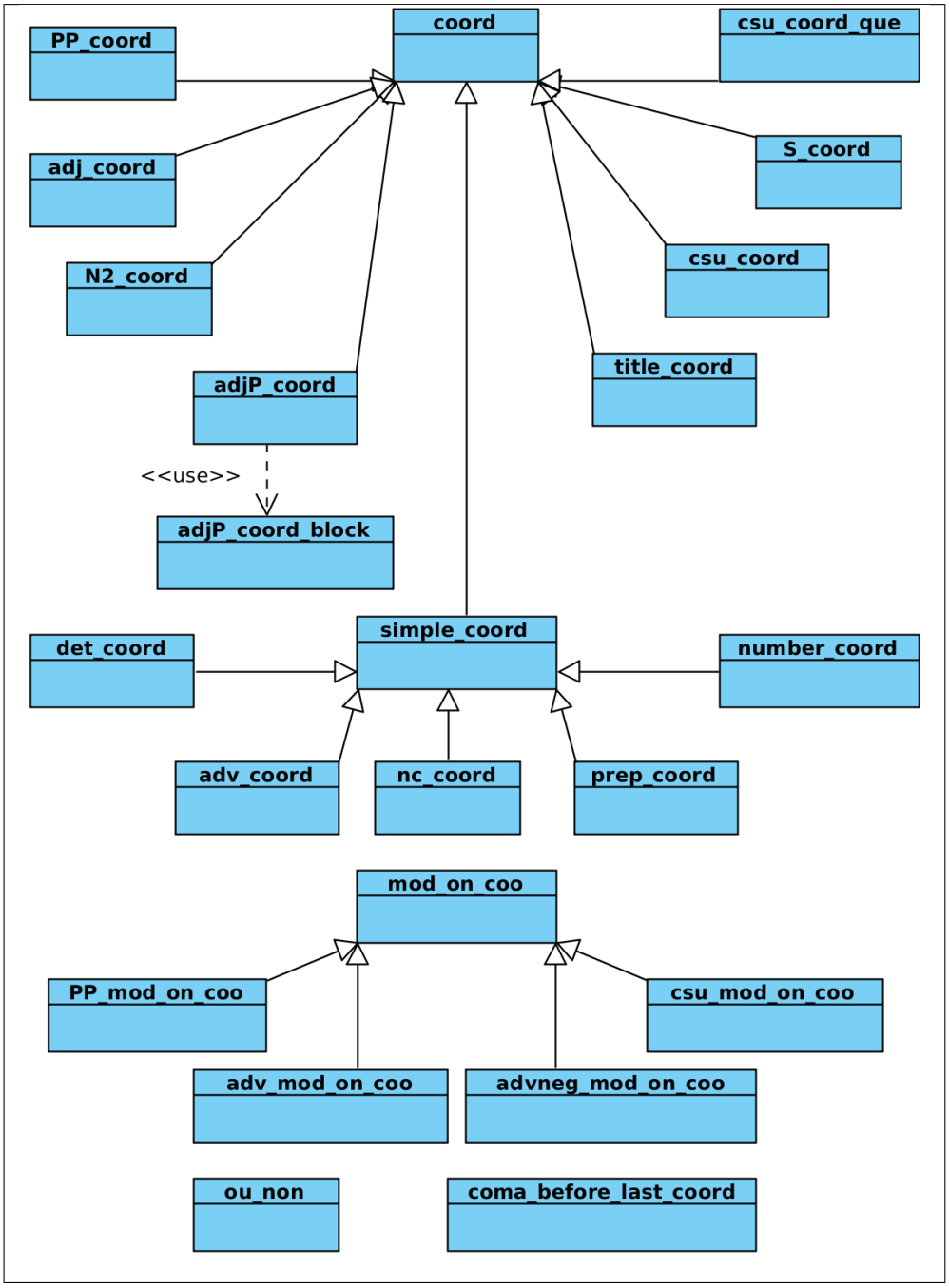


Figura 7.18: Diagrama de clases del paquete *Coord*.

7.4.14.3. *adj_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es un adjetivo. Por ejemplo, la conjunción coordinante *"y"* y los adjetivos *"corto"* y *"estrecho"* en la oración *"El pasillo es corto y estrecho"*.

7.4.14.4. *N2_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es un sintagma nominal. Por ejemplo, la conjunción coordinante *"o"* y los sintagmas nominales *"el paraguas"* y *"el chubasquero"* en la oración *"Cogemos el paraguas o el chubasquero"*.

7.4.14.5. *csu_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es una subordinada introducida por una conjunción subordinada. Por ejemplo, la conjunción coordinante *"y"* y las subordinadas adverbiales *"cuando yo quiera"* y *"donde yo quiera"* en la oración *"Iremos cuando yo quiera y donde yo quiera"*.

7.4.14.6. *csu_coord_que*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es una subordinada introducida por la conjunción subordinada *"que"*. Por ejemplo, la conjunción coordinante *"o"* y las subordinadas sustantivas *"que comas"* y *"que duermas"* en la oración *"Que comas o que duermas no me interesa"*.

7.4.14.7. *S_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es una oración. Por ejemplo, la conjunción coordinante *"pero"* y las oraciones *"él vino"* y *"ya se fue"* en la oración *"Él vino, pero ya se fue"*.

7.4.14.8. *title_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es una fórmula de tratamiento. Por ejemplo, la conjunción coordinante *"y"* y las fórmulas de tratamiento *"señor"* y *"señora"* en la oración *"Pasen señor y señora Zapatero"*.

7.4.14.9. *adjP_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es un sintagma adjetival. Por ejemplo, la conjunción

coordinante *"pero"* y los sintagmas adjetivales *"muy guapo"* y *"poco inteligente"* en la oración *"Es muy guapo, pero poco inteligente"*.

7.4.14.10. *adjP_coord_block*

La clase anterior utiliza los recursos ofrecidos por esta clase. Entre ellos se encuentran los que posibilitan la coordinación de gerundios y participios como sintagmas adjetivales. Por ejemplo, la conjunción coordinante *"y"* y los gerundios *"corriendo"* y *"nadando"* en la oración *"Corriendo y nadando llegaron a ese lugar"*.

7.4.14.11. *simple_coord*

Reúne las estructuras sintácticas donde uno de los elementos de la coordinación es una categoría léxica simple como: determinantes, adverbios, sustantivos, entre otros.

7.4.14.12. *det_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es un determinante. Por ejemplo, la conjunción coordinante *"y"* y los determinantes *"los"* y *"las"* en la oración *"Los y las tripulantes decidieron desembarcar"*.

7.4.14.13. *adv_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es un adverbio. Por ejemplo, la conjunción coordinante *"o"* y los adverbios *"hoy"* y *"mañana"* en la oración *"Lo tendrás para hoy o mañana"*.

7.4.14.14. *nc_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es un sustantivo. Por ejemplo, la conjunción coordinante *"y"* y los sustantivos *"pollo"* y *"ensalada"* en la oración *"Comimos pollo y ensalada"*.

7.4.14.15. *prep_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es una preposición. Por ejemplo, la conjunción coordinante *"y"* y las preposiciones *"por"* y *"para"* en la oración *"Vino por y para verte"*.

7.4.14.16. *number_coord*

Controla las estructuras sintácticas donde uno de los elementos de la coordinación es un determinante numérico. Por ejemplo, la conjunción coordinante *"y"* y los determinantes numéricos *"10"* y *"15"* en la oración *"Entraron entre 10 y 15"*.

7.4.14.17. *mod_on_coo*

Maneja las estructuras generales donde un modificador acompaña a la conjunción coordinante.

7.4.14.18. *PP_mod_on_coo*

Maneja las estructuras sintácticas donde un sintagma preposicional modifica a la conjunción coordinante. Por ejemplo, la conjunción coordinante *"y"* y el sintagma preposicional *"de hecho"* en la oración *"Unos libros y, de hecho, unas enciclopedias fueron vendidas"*.

7.4.14.19. *adv_mod_on_coo*

Maneja las estructuras sintácticas donde un adverbio modifica a la conjunción coordinante. Por ejemplo, la conjunción coordinante *"y"* y el adverbio *"además"* en la oración *"Esa casa está destruida y, además, deshabitada"*.

7.4.14.20. *advneg_mod_on_coo*

Maneja las estructuras sintácticas donde un adverbio de negación modifica a la conjunción coordinante. Por ejemplo, la conjunción coordinante *"pero"* y el adverbio de negación *"tampoco"* en la oración *"No pagó, pero tampoco comió"*.

7.4.14.21. *csu_mod_on_coo*

Maneja las estructuras sintácticas donde una conjunción subordinada modifica a la conjunción coordinante. Por ejemplo, la conjunción coordinante *"pero"* y la conjunción subordinada *"que"* en la oración *"Traedlo, pero que no moleste"*.

7.4.14.22. *coma_before_last_coord*

Esta clase se encarga de manejar las comas presentes en las coordinaciones cuando los elementos que las componen son mayores de dos. Sin embargo, si se trata de una enumeración muy larga, será el paquete *Enumeration* el encargado de modelarla.

7.4.14.23. *ou_non*

Esta clase ha sido diseñada para permitir el fenómeno sintáctico *"o" + adverbio de negación* presente en una interrogación. Véase estos ejemplos:*"¿Quieres venir, o no?"* y *"¿Él va, o no, a venir?"*.

7.4.15. Paquete *Enumeration*

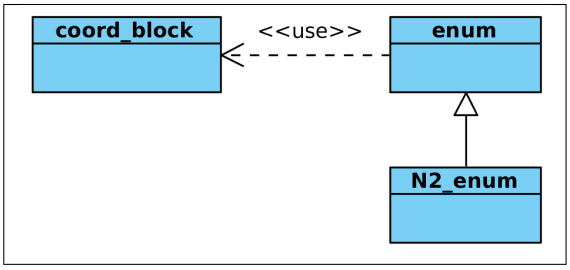


Figura 7.19: Diagrama de clases del paquete *Enumeration*.

enum es la clase principal del paquete descrito en la figura 7.19. Su principal cometido es el de manejar las enumeraciones largas. Se concreta el caso de enumeraciones de sintagmas nominales en la clase *N2_enum*, ya que pueden ser considerablemente más largas. La clase *coord_block* ofrece recursos para manejar los signos de puntuación y la coordinación de las enumeraciones.

7.4.16. Paquete *Superlative*

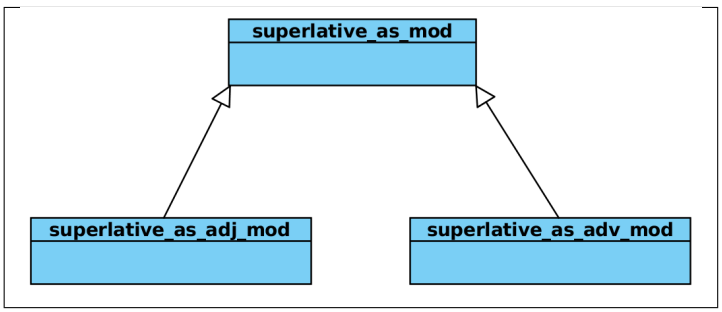


Figura 7.20: Diagrama de clases del paquete *Superlative*.

Este paquete (figura 7.20) se trata de una primera aproximación para controlar las estructuras de comparación superlativas. Existen dos

casos diferentes: cuando el superlativo se construye sobre un adverbio (*superlative_ as_ adv_ mod*) o sobre un adjetivo (*superlative_ as_ adj_ mod*). Ejemplos: *"Es el que menos rápidamente lo hace"* y *"Es el más listo de la clase"*.

7.4.17. Paquete *Subject*

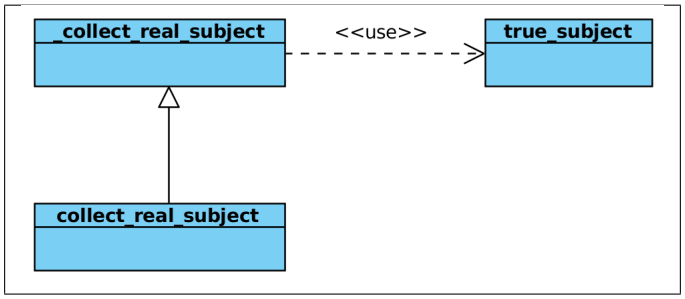


Figura 7.21: Diagrama de clases del paquete *Subject*.

La jerarquía formada por las clases *_ collect_ real_ subject* y *collect_ real_ subject*, detallada en la figura 7.21, trata de modelar los posibles sujetos de una oración, en el caso de no ser omitidos. Un sujeto puede estar formado por:

- **Sintagma nominal**[22]**:** *"El niño baila con su hermana"*.
- **Subordinadas sustantivas:** *"Que vinieses me molestó"*.
- **Verbos en modo infinitivo:** *"Fumar mata"*.
- **Clítico nominativo posverbal:** *"¿Qué libros compró él?"*.

La clase *true_ subject* provee recursos de coordinación entre el sujeto y el predicado.

7.4.18. Paquete *Generic Resources*

El paquete de la figura 7.22 recoge las clases que ofrecen recursos genéricos a prácticamente todos los paquetes de SPMG.

7.4.18.1. *modifier_ on_ S*, *modifier_ before_ S* y *modifier_ after_ S*

modifier_ on_ S reúne las características generales propias de un modificador de una oración a un nivel inferior al nodo S dentro del árbol.

[22]Este tipo de sujeto incluye los sintagmas nominales formados por adjetivos o pronombres, entre otros.

Esto significa que se trata de un modificador cercano al verbo de la oración, y por lo tanto, puede recogerse dentro de su estructura. Dentro de este tipo de modificadores, se distingue entre aquellos que preceden (*modifier_before_S*) o van después (*modifier_after_S*) del verbo que modifican.

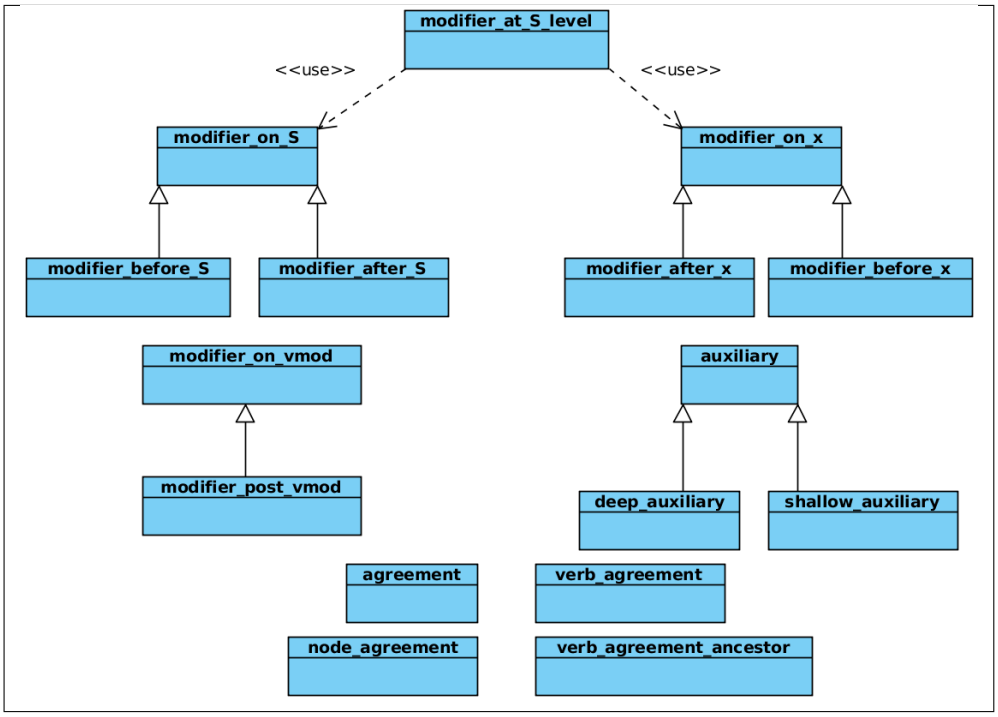


Figura 7.22: Diagrama de clases del paquete *Generic Resources*.

7.4.18.2. *modifier_on_x*, *modifier_before_x* y *modifier_after_x*

modifier_on_x reúne las características generales propias de cualquier modificador de una categoría léxica, excepto de los verbos, recogidos en la clase *modifier_on_S*. Dentro de este tipo de modificadores, se distingue entre aquellos que preceden (*modifier_before_x*) o van después (*modifier_after_x*) del elemento que modifican.

7.4.18.3. *modifier_at_level_S*

Se trata de una clase que maneja los modificadores a nivel oracional, esto es, a un nivel superior a las posiciones cercanas al verbo (controladas por *modifier_on_S*). Se utiliza particularmente para modificadores verbales situados lejos del verbo en la oración.

7.4.18.4. *modifier_on_vmod* y *modifier_post_vmod*

modifier_on_vmod reúne las características de aquellas categorías léxicas que modifican a los modificadores verbales. Concretando, *modifier_post_vmod*, restricciones propias de los modificadores posteriores de este tipo.

7.4.18.5. *auxiliary*, *deep_auxiliary* y *sallow_auxiliary*

Esta jerarquía de clases ofrece recursos para manejar las restricciones entre un nodo y su padre dentro de la estructura arbórea.

7.4.18.6. *agreement*

Mantiene la concordancia de género, número y persona entre un nodo y su padre dentro del árbol.

7.4.18.7. *node_agreement*

Mantiene la concordancia de género, número y persona entre un nodo y su antecedente no inmediato dentro del árbol.

7.4.18.8. *verb_agreement*

Controla la concordancia de género, número, persona, modo y tiempo verbal entre un nodo verbal y su antecedente inmediato (su padre) dentro del árbol.

7.4.18.9. *verb_agreement_ancestor*

Controla la concordancia de género, número, persona, modo y tiempo verbal entre un nodo verbal y su antecedente no inmediato dentro del árbol.

7.4.19. Paquete *Pres_s*

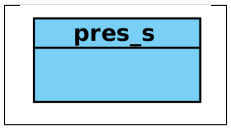


Figura 7.23: Diagrama de clases del paquete *Pres_s*.

Este paquete (figura 7.23) contiene una única clase. Ésta se encarga de manejar las interjecciones acompañando oraciones. En SPMG las

interjecciones son consideradas una categoría léxica[23]. Por ejemplo, *"oye"* en la oración *"¡Oye!, ¿pero qué estás haciendo?"*.

7.4.20. Paquete *Interjection*

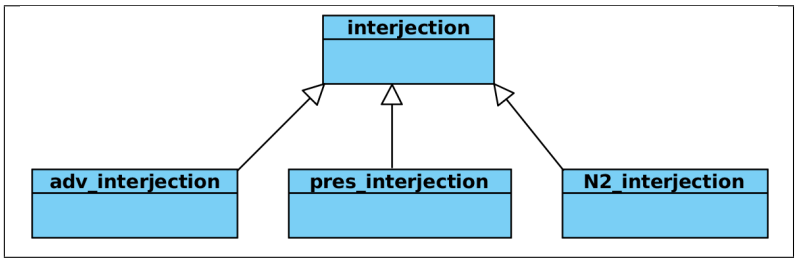


Figura 7.24: Diagrama de clases del paquete *Interjection*.

El paquete descrito en la figura 7.24 se encarga de manejar la situación de una interjección en una oración y de los signos de puntuación que deben acompañarla (signos de exclamación). Una interjección puede estar realizada por:

- **Sintagma nominal:** *"¡Cielos!, ¿pero qué haces?"* (***N2_interjection***)
- **Interjección (categoría léxica propia):** *"¡Eh!, ¿pero qué haces?"* (***pres_interjection***)
- **Adverbio:** *"¡Bien!, así se hace."* (***adv_interjection***)

7.4.21. Paquete *Wh_sentence*

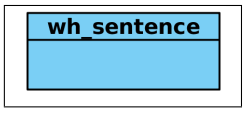


Figura 7.25: Diagrama de clases del paquete *Wh_sentence*.

Únicamente contiene una clase (figura 7.25) con el mismo nombre que se encarga de manejar las características propias de una oración interrogativa. Además, controla los signos de interrogación propios de este tipo de oraciones.

[23]En ocasiones, considerar un fenómeno sintáctico como una categoría léxica en SPMG viene marcado por la existencia de dicha categoría en el léxico LEFFE. Ello obliga a que el analizador sintáctico se adapte a la información morfosintáctica manejada, con el fin de aprovechar todo su potencial.

7.4.22. Paquete *Otros fenómenos sintácticos*

El paquete descrito en la figura 7.26 recoge otros fenómenos sintácticos presentes en la metagramática SPMG y que no se agruparon en los paquetes ya mencionados.

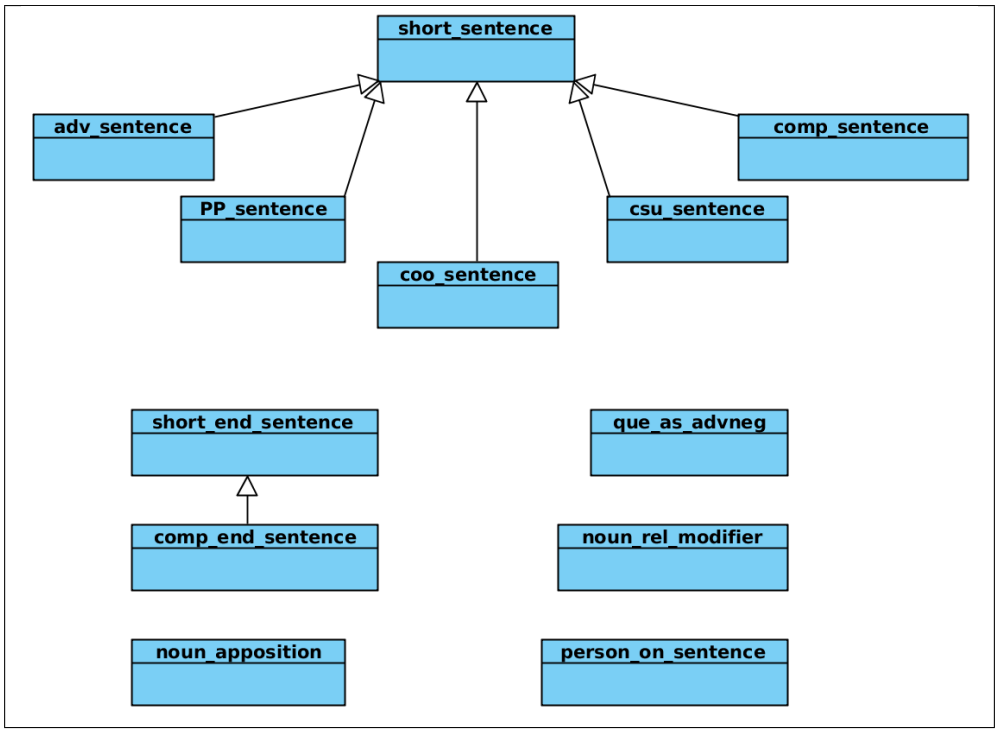


Figura 7.26: Diagrama de clases del paquete *Otros fenómenos sintácticos*.

7.4.22.1. *short_sentence*

Maneja el fenómeno sintáctico donde una oración se encuentra introducida por una sentencia corta separada de la oración principal por una coma. Por ejemplo, la frase *"sin embargo"* en la oración *"Sin embargo, vinieron"*. Sus descendientes indican las diferentes sentencias cortas introductorias dentro de SPMG.

7.4.22.2. *adv_sentence*

La sentencia introductoria está compuesta por uno o varios adverbios. Por ejemplo, los adverbios *"Siempre rápidamente"* en la oración *"Siempre rápidamente, cogen sus cosas y se van"*.

7.4.22.3. *PP_ sentence*

La sentencia introductoria está compuesta por un sintagma preposicional. Por ejemplo, el sintagma preposicional *"de repente"* en la oración *"De repente, se cayó el árbol"*.

7.4.22.4. *coo_ sentence*

La sentencia introductoria está compuesta por un sintagma preposicional coordinativo. Por ejemplo, la frase *"sin embargo"* en la oración *"Sin embargo, vinieron"*.

7.4.22.5. *csu_ sentence*

La sentencia introductoria está compuesta por una subordinada sustantiva o adverbial. Por ejemplo, la frase *"donde entramos"* en la oración *"Donde entramos, no había demasiada gente"*.

7.4.22.6. *comp_ sentence*

La sentencia introductoria está compuesta por un sintagma nominal. Por ejemplo, la frase *"el otro día"* en la oración *"El otro día, no había demasiada gente"*.

7.4.22.7. *short_ end_ sentence* y *comp_ end_ sentence*

Maneja el fenómeno sintáctico donde una oración precede a una sentencia corta separada de la oración principal por una coma. Únicamente se concreta para el caso en el que la sentencia corta sea un sintagma nominal. Por ejemplo, la frase *"señor presidente"* en la oración *"Entre en la habitación, señor presidente"*.

7.4.22.8. *que_ as_ advneg*

Maneja el fenómeno sintáctico donde en una oración negativa se incluye la construcción *adverbio de negación + verbo + "más que"*[24]. Por ejemplo, la oración *"No hace más que dormir"*.

7.4.22.9. *noun_ rel_ modifier*

Se encarga de controlar la relación de modificador entre una subordinada de relativo y su antecedente.

[24]Para conseguir mejorar la cobertura de una metagramática es necesario modelar fenómenos sintácticos muy particulares.

7.4.22.10. *noun_apposition*

Maneja las aposiciones presentes en una oración. Por ejemplo, la aposición *"el presidente"* en la oración *"Zapatero, el presidente, entró en la sala"*.

7.4.22.11. *person_on_sentence*

Modela las oraciones imperativas introducidas por un nombre propio. Por ejemplo, el nombre propio *"Juan"* en la oración *"¡Juan, vamos!"*.

7.5. Diagramas de actividad

Se muestran ahora los *diagramas de actividad* que especifican qué tareas debe llevar a cabo cada componente del sistema para cubrir las funcionalidades especificadas en el *diagrama de casos de uso*. Estos diagramas son fundamentales para determinar como interaccionarán luego estos componentes en los *diagramas de secuencia*.

7.5.1. Diagramas de actividad del Administrador

7.5.1.1. Iniciar servidor

La figura 7.27 presenta el *diagrama de actividad* del caso de uso especificado como *Iniciar servidor*.

7.5.1.2. Detener servidor

La figura 7.28 presenta el *diagrama de actividad* del caso de uso especificado como *Detener servidor*.

7.5.1.3. Registrar analizadores

La figura 7.29 presenta el *diagrama de actividad* del caso de uso especificado como *Registrar analizadores*.

7.5.1.4. Consultar registro analizadores

La figura 7.30 presenta el *diagrama de actividad* del caso de uso especificado como *Consultar registro analizadores*.

7.5.1.5. Modificar registro analizadores

La figura 7.31 presenta el *diagrama de actividad* del caso de uso especificado como *Modificar registro analizadores*.

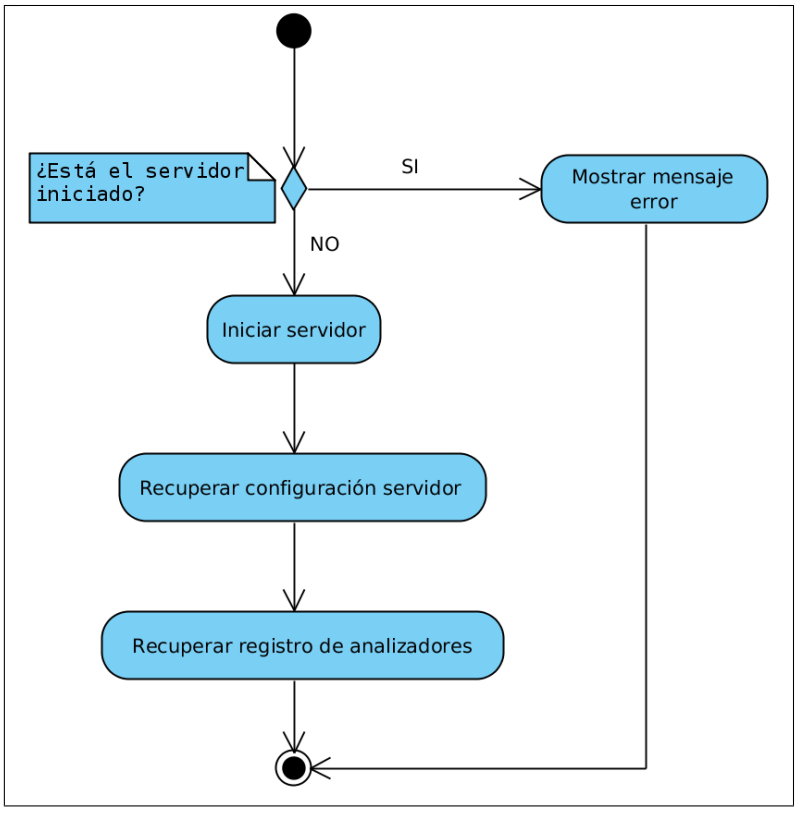


Figura 7.27: Diagrama de actividad de *Iniciar servidor*.

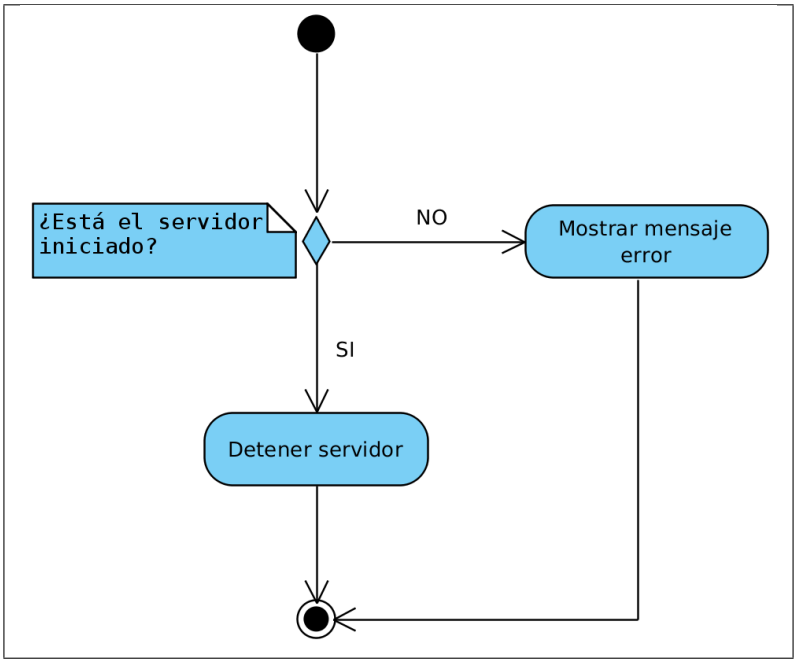


Figura 7.28: Diagrama de actividad de *Detener servidor*.

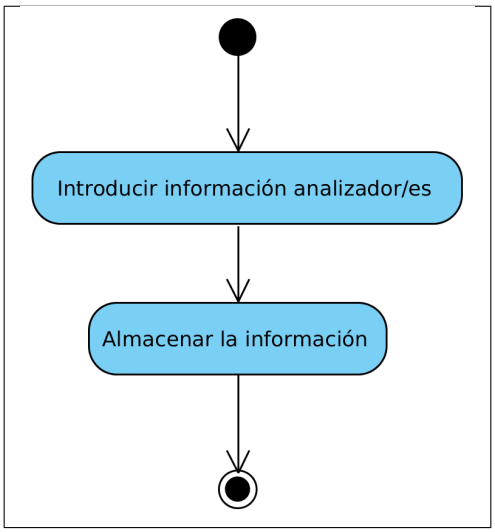


Figura 7.29: Diagrama de actividad de *Registrar analizadores*.

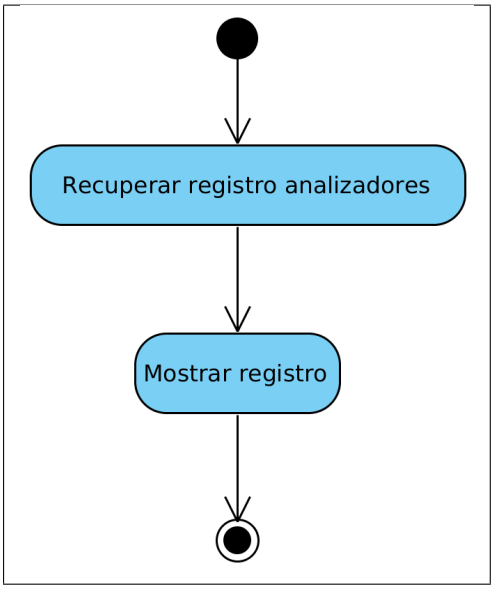


Figura 7.30: Diagrama de actividad de *Consultar registro analizadores*.

7.5.2. Diagramas de actividad del Usuario

7.5.2.1. Analizar frase

La figura 7.32 presenta el *diagrama de actividad* del caso de uso especificado como *Analizar frase*.

7.5.2.2. Analizar fichero

La figura 7.33 presenta el *diagrama de actividad* del caso de uso especificado como *Analizar fichero*.

7.5.2.3. Seleccionar analizador

La figura 7.34 presenta el *diagrama de actividad* del caso de uso especificado como *Seleccionar analizador*.

7.5.2.4. Seleccionar formato análisis

La figura 7.35 presenta el *diagrama de actividad* del caso de uso especificado como *Seleccionar formato análisis*.

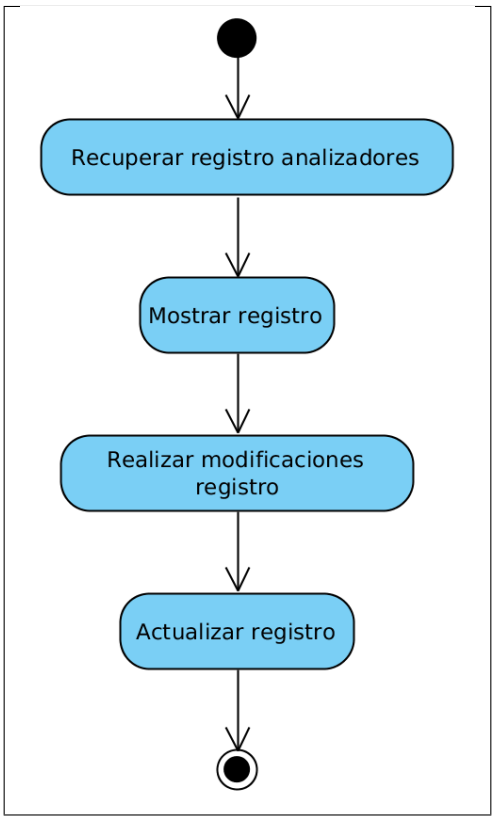


Figura 7.31: Diagrama de actividad de *Modificar registro analizadores*.

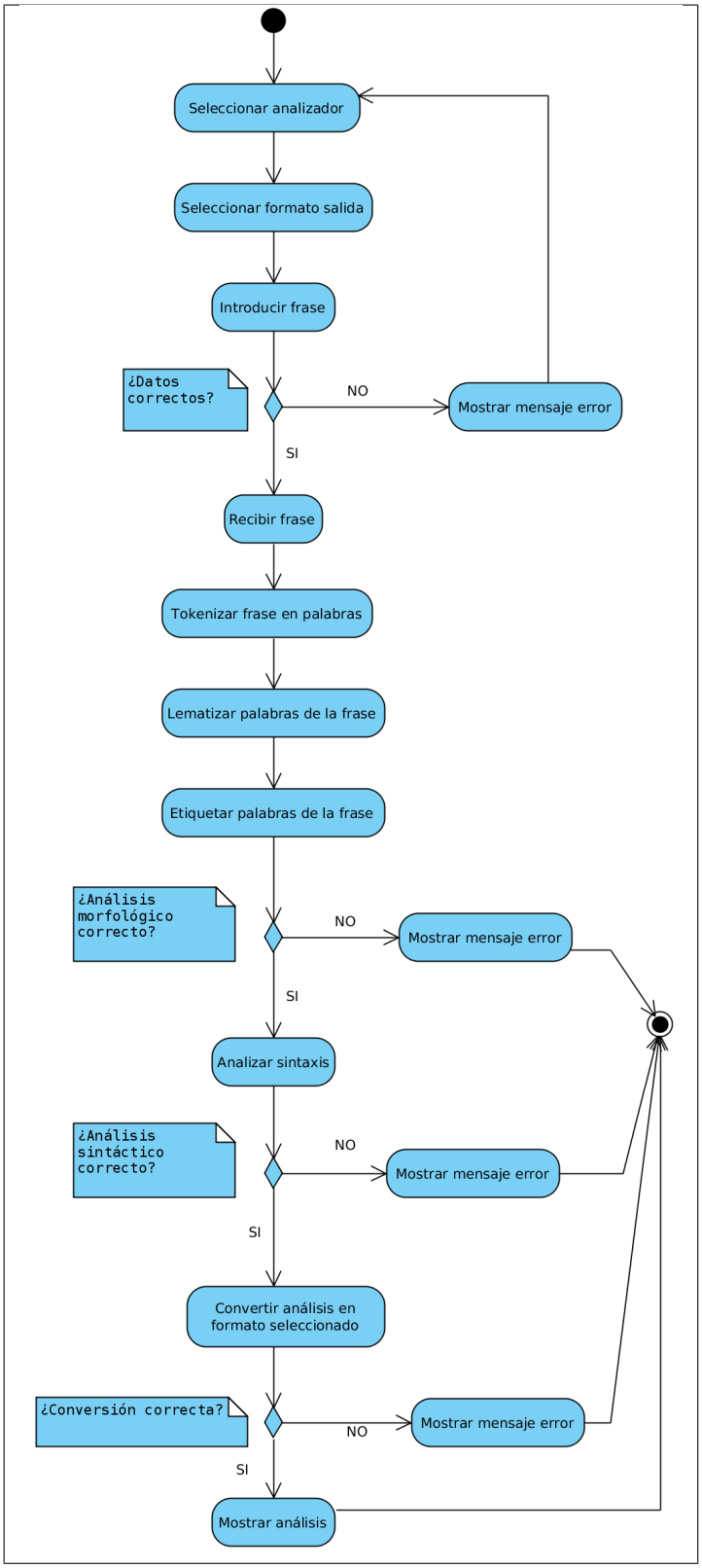


Figura 7.32: Diagrama de actividad de *Analizar frase*.

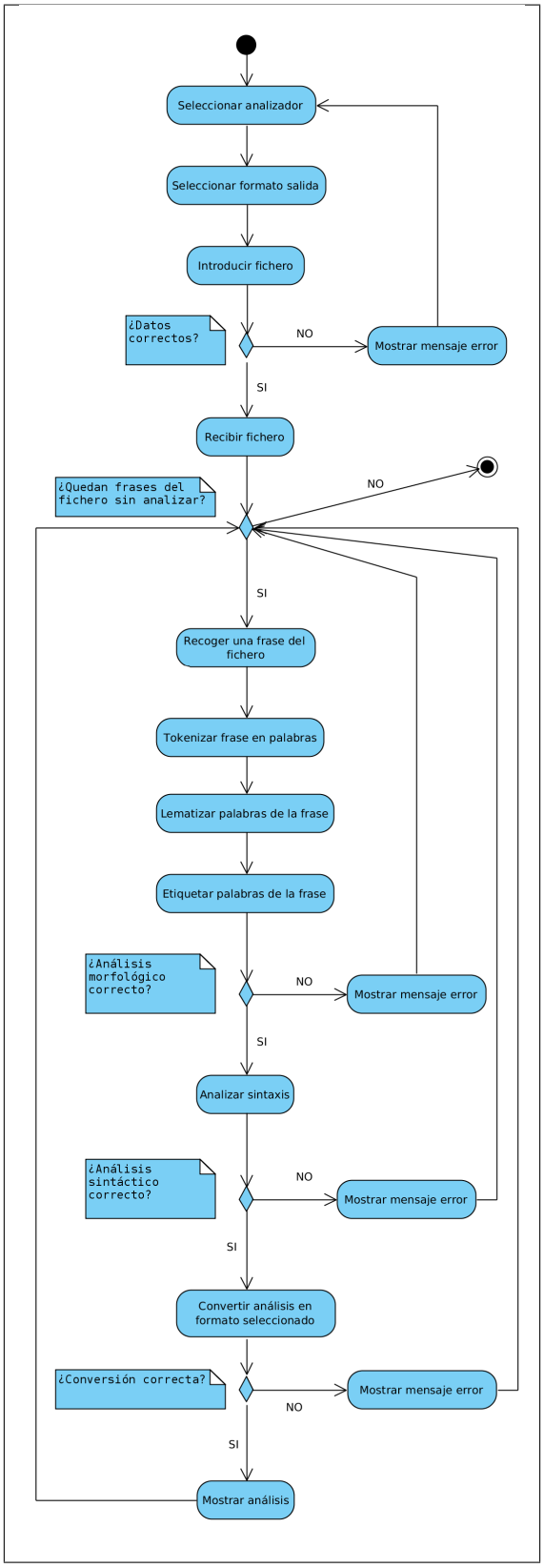


Figura 7.33: Diagrama de actividad de *Analizar fichero*.

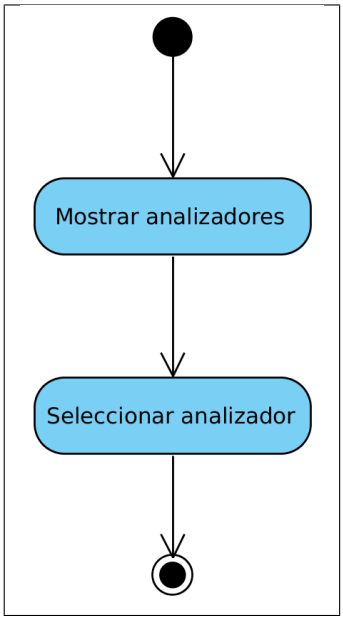


Figura 7.34: Diagrama de actividad de *Seleccionar analizador*.

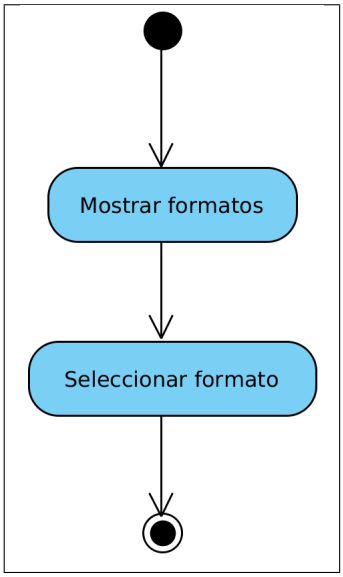


Figura 7.35: Diagrama de actividad de *Seleccionar formato análisis*.

7.5.2.5. Analizar morfológicamente

La figura 7.36 presenta el *diagrama de actividad* del caso de uso especificado como *Analizar morfológicamente*.

7.5.2.6. Analizar sintácticamente

La figura 7.37 presenta el *diagrama de actividad* del caso de uso especificado como *Analizar sintácticamente*.

7.5.2.7. Convertir análisis

La figura 7.38 presenta el *diagrama de actividad* del caso de uso especificado como *Convertir análisis*.

7.6. Diagramas de secuencia de diseño

En esta parte se va a disponer el modo en que interaccionan los componentes en tiempo de ejecución. Para ello se usarán los *diagramas de secuencia*. Se ha decidido no incluir los *diagramas de colaboración* puesto que representan, salvo pequeños matices, lo mismo que los *diagramas de secuencia*.

7.6.1. Diagramas de secuencia del Administrador

7.6.1.1. Iniciar servidor

La figura 7.39 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Iniciar servidor*.

7.6.1.2. Detener servidor

La figura 7.40 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Detener servidor*.

7.6.1.3. Registrar analizadores

La figura 7.41 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Registrar analizadores*.

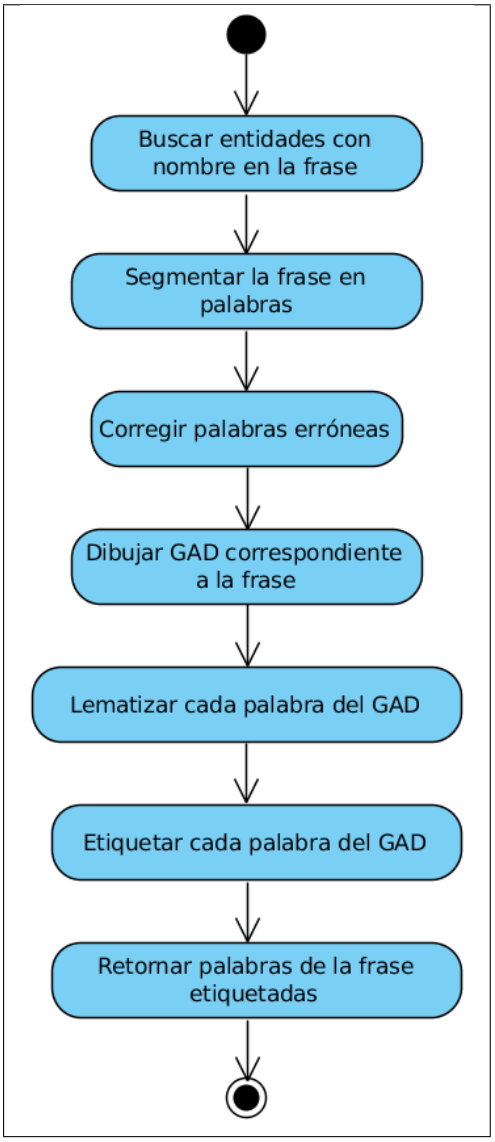


Figura 7.36: Diagrama de actividad de *Analizar morfológicamente*.

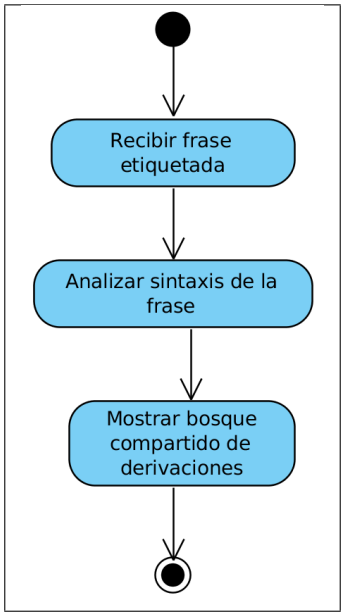


Figura 7.37: Diagrama de actividad de *Analizar sintácticamente*.

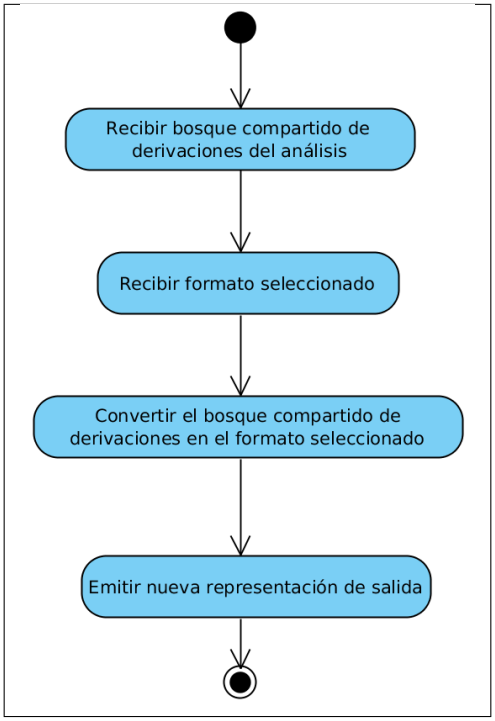


Figura 7.38: Diagrama de actividad de *Convertir análisis*.

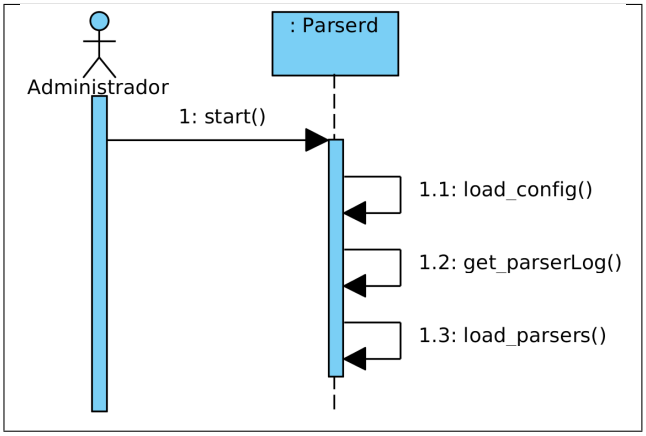


Figura 7.39: Diagrama de secuencia de *Iniciar servidor*.

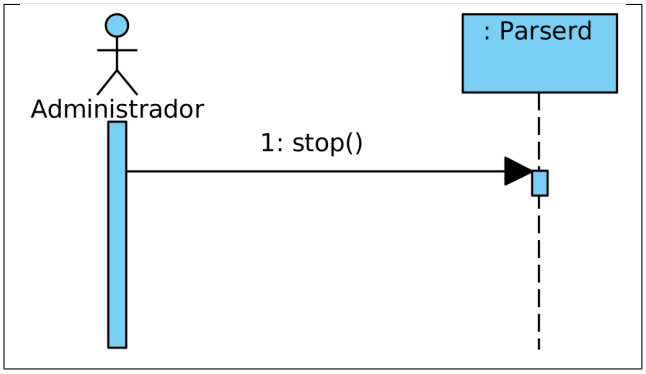


Figura 7.40: Diagrama de secuencia de *Detener servidor*.

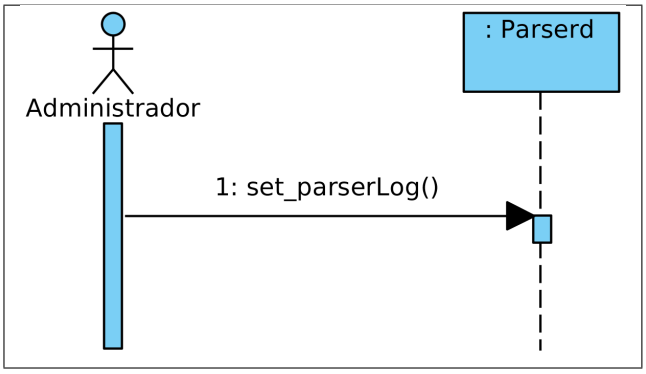


Figura 7.41: Diagrama de secuencia de *Registrar analizadores*.

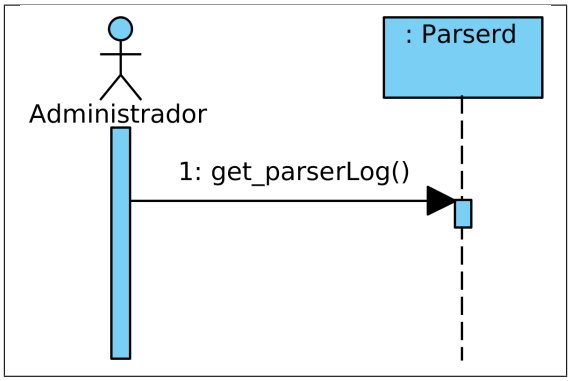


Figura 7.42: Diagrama de secuencia de *Consultar registro analizadores*.

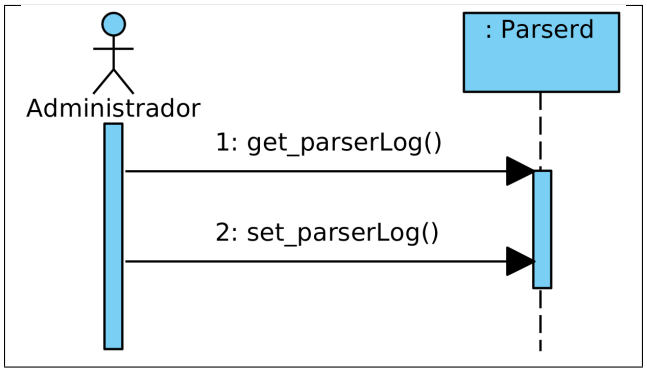


Figura 7.43: Diagrama de secuencia de *Modificar registro analizadores*.

7.6.1.4. Consultar registro analizadores

La figura 7.42 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Consultar registro analizadores*.

7.6.1.5. Modificar registro analizadores

La figura 7.43 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Modificar registro analizadores*.

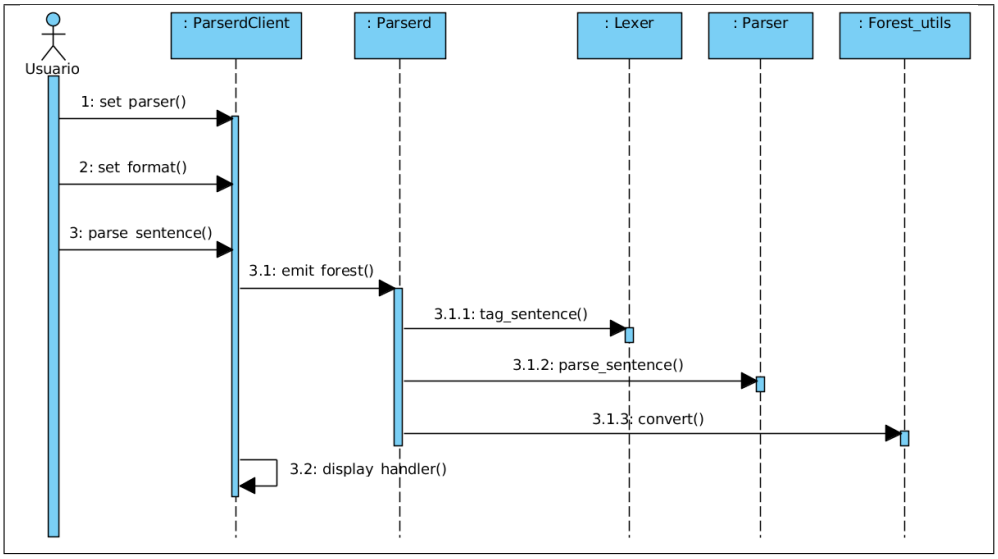


Figura 7.44: Diagrama de secuencia de *Analizar frase*.

7.6.2. Diagramas de secuencia del Usuario

7.6.2.1. Analizar frase

La figura 7.44 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Analizar frase*.

7.6.2.2. Analizar fichero

La figura 7.45 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Analizar fichero*.

7.6.2.3. Seleccionar analizador

La figura 7.46 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Seleccionar analizador*.

7.6.2.4. Seleccionar formato análisis

La figura 7.47 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Seleccionar formato análisis*.

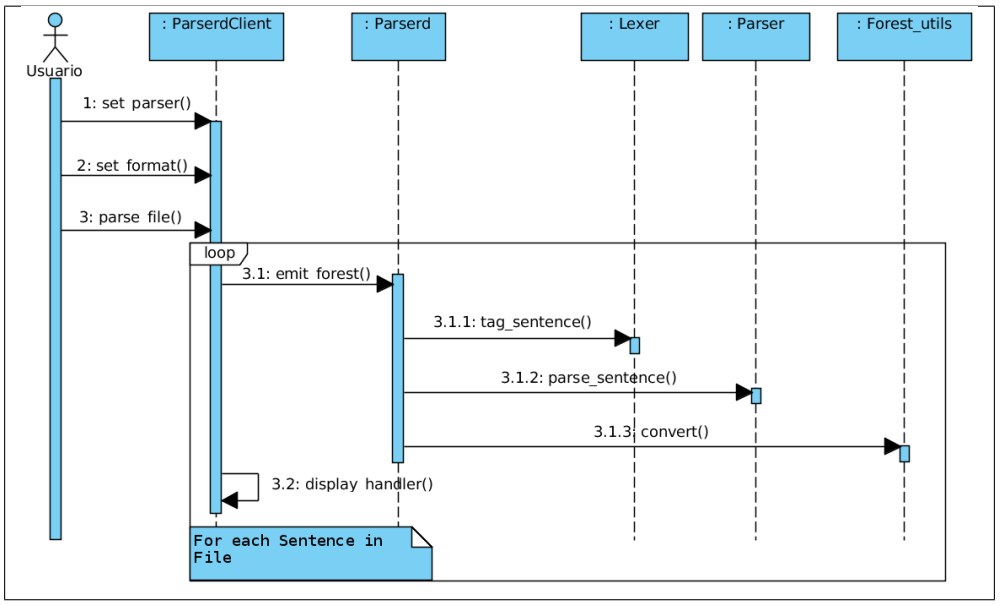


Figura 7.45: Diagrama de secuencia de *Analizar fichero*.

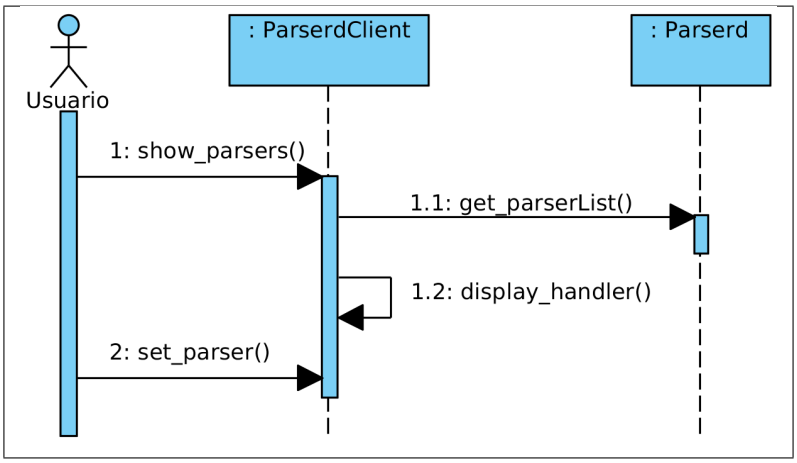


Figura 7.46: Diagrama de secuencia de *Seleccionar analizador*.

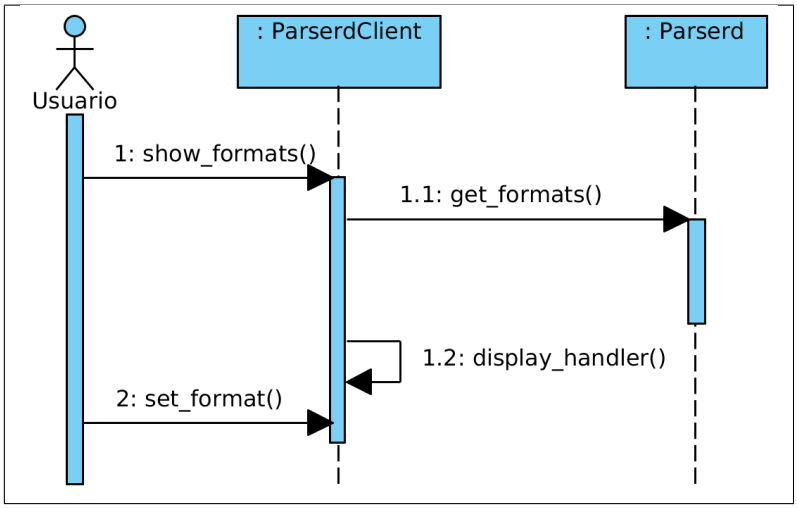


Figura 7.47: Diagrama de secuencia de *Seleccionar formato análisis*.

7.6.2.5. Analizar morfológicamente

La figura 7.48 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Analizar morfológicamente*. Se detalla la interacción entre componentes para el caso concreto de la cadena para el español. Esto es, se muestra como el analizador léxico *SPMG Lexer* interactúa con el preprocesador *SxPipe* y el lexicalizador *Lexed*. Otros *Lexers* presentes en el servidor no necesariamente deben emplear los recursos utilizados por *SPMG Lexer*. De ahí que, en el siguiente *diagrama de secuencia* no se trate de la clase genérica *Lexer* sino de la clase específica *SPMG_ Lexer*.

7.6.2.6. Analizar sintácticamente

La figura 7.49 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Analizar sintácticamente*.

7.6.2.7. Convertir análisis

La figura 7.50 presenta el *diagrama de secuencia* que muestra la interacción de las clases del sistema para realizar el caso de uso especificado como *Convertir análisis*.

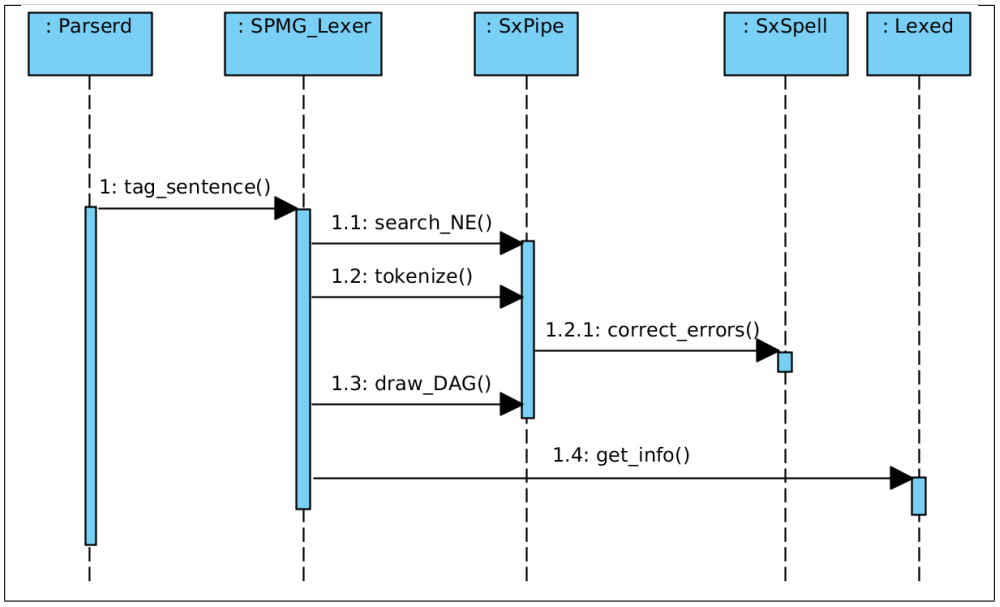


Figura 7.48: Diagrama de secuencia de *Analizar morfológicamente*.

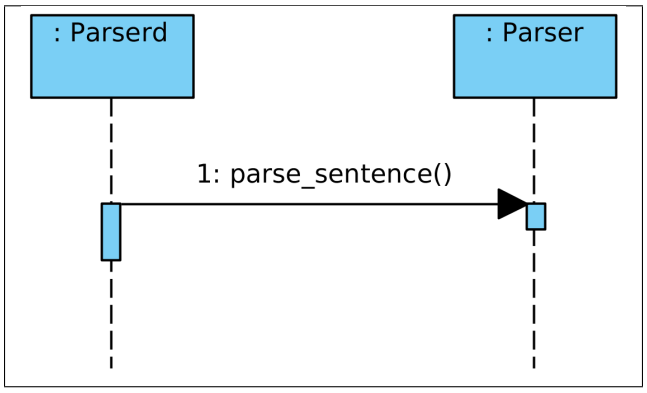


Figura 7.49: Diagrama de secuencia de *Analizar sintácticamente*.

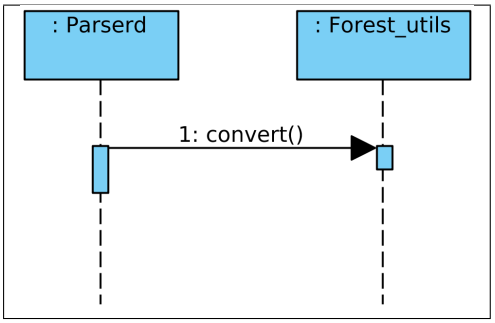


Figura 7.50: Diagrama de secuencia de *Convertir análisis*.

Capítulo 8

Implementación

En esta fase del desarrollo se traduce el modelo que se obtiene en el diseño en código fuente. En ella se reseñaran aspectos importantes de la implementación del sistema propuesto para cada componente del mismo. Principalmente, se detallará la interpretación de los ficheros de configuración existentes.

8.1. Analizador léxico

Para la implementación de este proyecto ha sido necesario hacer uso de diferentes tecnologías y herramientas. Concretamente, para construir el nivel morfológico de la cadena, se ha utilizado el lenguaje de programación *Perl* y el formalismo lexical ALEXINA.

Por un lado, la implementación del componente *SPMG Lexer* se ha llevado a cabo sobre el lenguaje de programación *Perl*, lenguaje base del resto de componentes reutilizados de la cadena ALPAGE. Por otro lado, tenemos el enriquecimiento de la base de información morfosintáctica LEFFE. Ésta se encontraba ya en un estado avanzado de desarrollo. El avance de la metagramática ha obligado a mejorar la información léxica presente en el LEFFE. Es necesario recordar que se trata de un léxico morfosintáctico, ello quiere decir que éste provee al analizador sintáctico, además de información léxica, de información de las estructuras gramaticales permitidas para el léxico recogido. De este modo, las modificaciones en las estructuras de la metágramatica implican cambios y mejoras en la información morfosintáctica del léxico recogido en LEFFE. Además, el lexicón no disponía de algunos vocablos de uso común necesarios para desarrollar una cadena de procesamiento lingüístico de amplia cobertura del español. Ello ha implicado el enriquecimiento de vocabulario de este recurso lexical.

La información recogida en el LEFFE se encuentra organizada en diferentes ficheros de acuerdo a la categoría léxica. Estos ficheros,

implementados mediante el formalismo ALEXINA, son los siguientes.

- **A.ilex:** Recoge adjetivos del castellano.
- **C.ilex:** Recoge conjunciones del castellano.
- **D.ilex:** Recoge determinantes del castellano.
- **C.ilex:** Recoge interjecciones del castellano[1].
- **N.ilex:** Recoge sustantivos del castellano.
- **P.ilex:** Recoge pronombres del castellano.
- **R.ilex:** Recoge adverbios del castellano.
- **S.ilex:** Recoge preposiciones del castellano.
- **V.ilex:** Recoge verbos del castellano.

Una vez compilados mediante *ALEXINA-tools* se obtienen los correspondientes ficheros `.lex` para cada categoría léxica de las indicadas anteriormente. Estos se pueden encontrar bajo el directorio de la aplicación `exportbuild/share/leffe`.

En la segunda compilación del LEFFE entra en juego el paquete LEFFE-SPMG. Este paquete desarrollado tomando como modelo el paquete LEFFF-FRMG, se encarga de reunir toda la información presente en los diferentes ficheros `.lex` del LEFFE bajo un único fichero (`dico.xlfg`). El fichero `dico.xlfg` va a ser posteriormente compilado por el recurso *Lexed* en un autómata finito contenido en el fichero `dico.xlfg.fsa`. Esta nueva estructura de autómata del lexicón permite un acceso más eficiente a la información morfosintáctica contenida en él. En esta compilación realizada por *Lexed* también se utilizan los ficheros `dico.xlfg.cmpnd` y `dico.xlfg.pref` proveídos por el paquete LEFFE-SPMG. El primero recoge frases hechas y nombres propios compuestos que serán incluidos en el LEFFE compilado. Asimismo, `dico.xlfg.pref` inyecta en el lexicón LEFFE una lista de prefijos del castellano. Éstos van a permitir un aumento de la cobertura del lexicón. Ante una palabra formada por uno de estos prefijos, el sistema será capaz de reconocerlo y extraer el lema de la misma, posiblemente presente en el LEFFE. De otro modo, sin esta lista de prefijos, la palabra sería tachada de desconocida. Estos ficheros están ubicados en el directorio `exportbuild/lib/leffe-spmg`.

Por otro lado, el recurso *SPMG Lexer* dispone de una serie de ficheros de configuración ubicados bajo el directorio `/exportbuild/share/spmg` que le aportan una mayor funcionalidad. Éstos se detallan a continuación.

[1] Nótese que en el LEFFE las interjecciones son consideradas categorías léxicas.

8.1.1. spmg.yml

Este fichero recoge las conversiones de nomenclaturas que *SPMG Lexer* debe efectuar sobre la información morfosintáctica que recibe del LEFFE, con el fin de traducirla en un formato comprensible para el analizador sintáctico. Éste presenta el siguiente aspecto:

```
cat_transfer:
   a: adj
   advm: adv
   advp: adv
   auxAvoir: aux
   auxEtre: aux
   c: coo
   cf: ncpred
   cfi: ncpred
   d: det
   i: pres
   n: nc
   p: pro
   parentf: ponctw
   parento: ponctw
   r: adv
   s: prep
value_transfer:
   false: -
   intens: intensive
   plural: pl
   sing: sg
   singular: sg
   ssubj: subj
   true: +
   vsubj: subj
exclude_feature:
   RDexplication: 1
   advGP: 1
   advi: 1
   clgen: 1
   clivee: 1
   clla: 1
   clle: 1
   clles: 1
   clloc: 1
   clneg: 1
```

```
   detach: 1
   detach_neg: 1
   form: 1
   loc: 1
   mod_kind: 1
   persposs: 1
   number: 1
exclude_cat_feature:
   adj:
      chunk_type: 1
      diathesis: 1
      mode: 1
      type: 1
      adj_used_as_participle: 1
      def: 1
      det: 1
      time: 1
   adv:
      adv_parlant: 1
      chunk_type: 1
      mode: 1
      time: 1
   coo:
      arg2: 1
   prep:
      chunk_type: 1
   pres:
      chunk_type: 1
   pro:
      polite: 1
      exclamative: 1
      wh: 1
   det:
      exclamative: 1
   prel:
      def: 1
   cla:
      adj: 1
   cln:
      polite: 1
feature_transfer:
   aux-req: aux_req
   define: def
   demonstrative: dem
```

```
form-aux: form_aux
inflnumber: number
inflperson: person
locative: loc
mood: mode
possessive: poss
qu: wh
synt_head: lightverb
v-form: mode
weekday: time
year: time
```

Donde, las distintas secciones indican:

- `cat_transfer:` En esta sección se recoge cuál es la conversión de nomenclaturas de categorías léxicas realizada. Se indica la nomenclatura en el LEFFE (`a`) y despues su denominación correspondiente en el analizador sintáctico (`adj`), como es el caso de la traducción de la nomenclatura mediante la cual el lexicón denomina a los adjetivos, `a: adj`. Si el nombre de la categoría léxica no varía, no es necesario indicarlo.

- `value_transfer:` Al igual que en el caso de las categorías léxicas, es necesario traducir los valores presentes en la información morfosintáctica. Por ejemplo, `plural: pl` donde se indica la conversión del valor `plural`, empleado para marcar el número de una determinada forma en el LEFFE, en la nomenclatura `pl`, empleada en *SPMG Parser*.

- `exclude_feature:` Existen categorías léxicas presentes en LEFFE que el analizador sintáctico no es capaz de manejar actualmente. Por ello, se niega su transferencia al nivel sintáctico para evitar problemas de incompatibilidades. Por ejemplo, la línea `clneg: 1` evita que un clítico negativo se envíe al analizador sintáctico, puesto que no existen clíticos de negación para el español[2].

- `exclude_cat_feature:` Hay información morfosintáctica aportada por el LEFFE que excede las necesidades de *SPMG Parser*, ocasionándole problemas. Por ello es necesario filtrar la información proporcionada. Por ejemplo, la línea *time: 1* para el adjetivo, prescinde de la información que nos indica si se trata de un adjetivo temporal (por ejemplo: *"viejo"*, *"nuevo"*)[3].

[2]Se trata de una categoría léxica heredada del LEFFF.
[3]El uso de esta información permitiría construir una mejor metagramática.

- `feature_transfer`: Traducción de la información de las categorías léxicas. Cada categoría léxica tiene asociada información morfosintáctica y ésta puede denominarse de distinta forma en el LEFFE que en el analizador sintáctico, por ello se realiza la conversión. Por ejemplo, en el lexicón la información referente al auxiliar requerido por un verbo se indica en el atributo `aux-req`, mientras que el analizador sintáctico lo denomina como `aux_req`.

8.1.2. restrictions.txt

Se ha comentado como evitar la llegada de categorías léxicas y determinada información del LEFFE al analizador sintáctico. Sin embargo, este fichero permite realizar otro tipo de filtrado. Su aspecto es el siguiente:

```
## To remove some rare or non pertinent Leffe entries

se ilimp
una nc
ve nc
la nc
las nc
qué adv
comer nc
no nc
no adv
a nc
de nc
cantar nc
por adj
vivir nc
según adv
dormido nc
feo nc
dormidos nc
mantenido nc
feo nc
querido nc
```

En este fichero se indican aquellas formas presentes en el LEFFE que son extrañas o que no se suelen utilizar en el contexto de trabajo actual. Es necesario indicar la forma y la categoría léxica, puesto que una misma forma puede pertenecer a distintas categorías léxicas. Por ejemplo, se puede ver una gran cantidad de verbos y adjetivos descartados como sustantivos. Esto se debe a que LEFFE considera que un adjetivo o un verbo en modo

infinitivo sustantivado tienen la categoría léxica de sustantivo, cuando desde la metagramática no se sigue esta norma.

Otra consecuencia importante de emplear este fichero, es la de reducir la ambigüedad, tanto léxica como sintáctica. Si, por ejemplo, estamos trabajando dentro del contexto de la botánica, donde la forma *"hierbas"* como sustantivo aparece un gran número de veces. Puede interesarnos filtrar la forma *"hierbas"* como segunda persona singular del verbo *"herbar"*. Ello va a evitar que el analizador sintáctico eche mano de los árboles verbales de la gramática GAR, lo que implicaría un considerable incremento de la ambigüedad y complejidad del análisis realizado.

Evidentemente, esta información podría ser modificada directamente en el LEFFE. Pero ello provocaría la eliminación de la misma de forma definitiva, cuando podría ser necesaria en otro contexto de trabajo.

8.1.3. complete.lex y missing.lex

Ambos ficheros ofrecen una funcionalidad similar. Tienen la extensión del nivel extensional del LEFFE y se utilizan para añadir información morfosintáctica no recogida en el lexicón[4].

Ambos ficheros pueden ser empleados para la misma finalidad. Sin embargo, se mantiene una ligera diferencia entre ellos. El fichero `complete.lex` se utiliza para incluir información morfosintáctica de lemas ya existentes en LEFFE; mientras que `missing.lex`, recoge formas de lemas no presentes en él.

Del mismo modo que en el caso de `restrictions.txt`, se podrían incorporar las modificaciones directamente en el LEFFE. Pero si, siguiendo con el supuesto anterior, nos encontrásemos en un contexto de trabajo botánico, podría ser necesario incorporar palabras específicas de este campo sin engordar nuestro LEFFE genérico. Esto es, introducir las formas necesarias para realizar el trabajo actual sin saturar el lexicón con información que no vaya a ser necesaria en los siguientes análisis.

8.2. Analizador sintáctico

Una vez diseñadas e implementadas las clases de la metagramática para el español, ésta es compilada obteniendo como resultado el analizador sintáctico híbrido GAR/GIR *SPMG Parser*.

Cada metagramática da lugar a uno o varios analizadores basados en el analizador sintáctico resultante de su compilación. Con el fin de poder registrar estos analizadores en el servidor *Parserd* se crea el fichero `register_parserd.conf`. Éste recoge los parámetros necesarios para que el

[4]El formato extensional de ALEXINA ha sido detallado en la sección 2.3.1.1 de la *Memoria*.

servidor *Parserd* pueda hacer uso de los mismos. Cada analizador o cadena de procesamiento registrada en este servidor debe de estar compuesto de un etiquetador (*Lexer*) y de un analizador sintáctico (*Parser*).

El fichero `register_parserd.conf`, ubicado bajo el directorio `/exportbuild/share/spmg`, es el correspondiente a los analizadores basados en SPMG y, por lo tanto, formados por el analizador sintáctico *SPMG Parser* y, opcionalmente, por el etiquetador *SPMG Lexer*. Concretamente, presenta el siguiente aspecto:

```
## To register these parsers whithin server parserd
## use 'register_parsers' from parserd distribution
## register_parsers -a /home/dani/exportbuild/share
## /spmg/register_parserd.conf <path_to_parserd.conf>

## parserd may be found on http://alpage.inria.fr/
## catalogue.en.html#parserd
## or by contacting Eric de la Clergerie
## <Eric.De_La_Clergerie@inria.fr>

[spmgtel]
    label = SPMG TAG/TIG - Hybrid Strategy
            - SPMG Lexer
    cmd = /home/dani/exportbuild/bin/spmg_parser
    options = -forest
    forest = lp
    examples = /home/dani/exportbuild/share/spmg/
               tests.txt
    lexer = persistent /home/dani/exportbuild/bin/
            spmg_lexer
    origin = spmg-0.1
    language = spanish

[spmgtelr]
    label = SPMG TAG/TIG - Hybrid Strategy
            - SPMG Lexer - Robust
    cmd = /home/dani/exportbuild/bin/spmg_parser
    options = -forest -robust
    forest = lp
    examples = /home/dani/exportbuild/share/spmg/
               tests.txt
    lexer = persistent /home/dani/exportbuild/bin/
            spmg_lexer
    origin = spmg-0.1
    language = spanish
```

Donde:

- `[spmgtel]:` Se trata del identificador del analizador.
- `label:` Es la etiqueta que describe el analizador.
- `cmd:` Indica la ruta donde se encuentra el ejecutable del analizador sintáctico.
- `options:` Recoge las opciones del analizador sintáctico. `-forest` indica que el resultado del análisis va a ser un bosque compartido de derivaciones[5] y `-robust`, que el analizador se encuentra en modo robusto.
- `forest:` Le indica al recurso *forest utils* cuál va a ser el formato empleado por *SPMG Parser* para mostrar el resultado del análisis. `lp` es un formato de representación soportado como entrada por el recurso *forest utils*, encargado de la conversión del resultado en los formatos deseados.
- `examples:` Recoge la ruta de un fichero de texto que contiene frases predeterminadas para testear el procesamiento lingüístico del analizador.
- `lexer:` Indica la ruta donde se encuentra el ejecutable del analizador léxico o etiquetador.
- `origin:` Recoge cuál es la metagramática base del analizador sintáctico y cuál es su versión.
- `language:` Idioma objeto del procesamiento lingüístico.

Para el analizador o cadena de procesamiento lingüístico para el español se han descrito dos analizadores diferentes. Se trata de la combinación del mismo etiquetador (*SPMG Lexer*) y analizador sintáctico (*SPMG Parser*), pero con la opción de robustez activada (`spmgtelr`) o desactivada (`spmgtel`).

Se podría haber descrito un analizador en este fichero, incluyendo la ruta de un ejecutable *Lexer* diferente de *SPMG Lexer*. Ello permite hacer uso, por ejemplo, de un *Lexer* concreto que trabaje sobre un LEFFE específico para un contexto de trabajo bursátil. Asimismo, podría existir otro *Lexer* centrado en vocabulario botánico. De esta forma, bajo una previa selección, el usuario podría determinar que analizador concreto le interesa usar en su marco de trabajo actual.

[5] Es necesario indicárselo, ya que el analizador sintáctico por defecto no ofrece salida alguna.

8.3. Componentes reutilizados

La reutilización de componentes de la cadena ALPAGE se ha realizado en *Perl*.

En el caso del cliente *web* ha sido necesario llevar a cabo parte de la implementación del mismo, puesto que estaba incompleto y no funcionaba de forma correcta.

En el caso del servidor *Parserd* y *SxPipe* tenemos, para ambos recursos reutilizados de la cadena ALPAGE, ficheros de configuración propios.

El fichero `sxpipe.conf` de *SxPipe* se ubica bajo el directorio `/exportbuild/etc` y presenta el siguiente aspecto:

```
# sxpipe reloaded

# Language
lang    = es

# Path to different
sxpipe  = @datadir@/sxpipe
bindir = @bindir@

# Verbose flag
verbose

# Component list
components = \
   tag_meta \
   email \
   url \
   date \
   tel \
   heure \
   adresse \
   mtponct \
   numprefix \
   number \
   smiley \
   formattage \
   remove_inner_1 \
   segment \
   sigles \
   np \
   remove_inner_2 \
```

```
   normalize \
   text2dag \
   epsilonize

# raw text to tokens
[tag_meta]
cmd = $sxpipe/txt2tok/tag_meta.pl
desc = protection des meta-caractères

[email]
cmd = $sxpipe/txt2tok/gl_email.pl
desc = reconnaissance des email

[url]
cmd = $sxpipe/txt2tok/gl_url.pl
desc = reconnaissance des url
depend = email

[date]
cmd = $sxpipe/txt2tok/gl_date.pl
options = -l=$lang
desc = reconnaissance des dates

[tel]
cmd = $sxpipe/txt2tok/gl_tel.pl
desc = reconnaissance des n° de téléphone

[heure]
cmd = $sxpipe/txt2tok/gl_heure.pl
desc = reconnaissance des heures

[adresse]
cmd = $sxpipe/txt2tok/gl_adresses.pl
desc = reconnaissance des adresses

[numprefix]
cmd = $sxpipe/txt2tok/gl_numtruc.pl
desc = reconnaissance des prefixes numeriques
(genre 4-place)

[number]
cmd = $sxpipe/txt2tok/gl_number.pl
desc = reconnaissance des nombres
```

```
[smiley]
cmd = $sxpipe/txt2tok/gl_smiley.pl
desc = reconnaissance des smileys

[qword]
cmd = $sxpipe/txt2tok/gl_qword.pl
desc = reconnaissance des mots cités

[mtponct]
cmd = $sxpipe/txt2tok/gl_mtponct.pl
options = -l=$lang
desc = reconnaissance des ponctuations multitokens

[formattage]
cmd = $sxpipe/txt2tok/gl_format.pl
desc = reconnaissance des mots formattés (pseudo-gras
 type _mot_ ou *mot*)

[segment]
cmd = $sxpipe/txt2tok/segmenteur.pl
options = -l=$lang -af=@lefffdir@/pctabr -p=r $*
desc = segmentation

# tokens to compound components
[sigles]
cmd = $sxpipe/tok2cc/gl_sigles.pl
desc = reconnaissance des sigles

[np]
cmd = $sxpipe/tok2cc/gl_np.pl
desc = reconnaissance des noms propres

[etr]
cmd = $sxpipe/tok2cc/gl_etr.pl
options = -no_gl -l=$lang
desc = reconnaissance des expressions langue
êtrangère

[remove_inner_1]
cmd = $sxpipe/tok2cc/remove_inner_ne.pl
desc = desencapsulation des entitées nommées

[remove_inner_2]
cmd = $sxpipe/tok2cc/remove_inner_ne.pl
```

```
desc = desencapsulation des entitées nommées

[cardinal]
cmd = $sxpipe/cc2dag/gl_card.pl
desc = reconnaissance des cardinaux

[ordinal]
cmd = $sxpipe/cc2dag/gl_ieme.pl
desc = reconnaissance des dates et des ordinaux

[normalize]
cmd = $sxpipe/tok2cc/normalize_blanks.pl
desc = normalisation des espaces et de pourcentages

# compound components to DAG
[text2dag]
cmd = $bindir/text2dag$lang
options = -cl -pl -cq -ch -pp -ps -slw 20 -lw 40
-sc -mw 300 -sww 140 -mcn 2 -mcwn 1 -g
desc = correction ambiguë et reconnaissance ambiguë
des mots composés, amalgames, et combinaisons
d'icelles

[epsilonize]
cmd = $sxpipe/cc2dag/epsilonize_metawords.pl
desc = met en alternative avec _EPSILON les
métamots (quotes, crochets...)

[unfold]
cmd = $sxpipe/cc2dag/wrapunfolddag.pl
desc = dépliage des DAGs
```

Donde en él se indica previamente el lenguaje objeto del preprocesado (`lang = es`) y a continuación se detallan los distintos *scripts Perl* que lo componen. Estos son los siguientes:

- `tag_meta`: La función de este componente es la de procesar los metacaracteres.
- `email`: Se encarga del procesamiento de las direcciones de correos electrónicos.
- `url`: Reconoce las URLs.
- `date`: Se encarga de reconocer las fechas presentes en el texto.

- `tel:` Reconoce los números de teléfono.
- `heure:` Procesa los distintos formatos horarios existentes.
- `adresse:` Reconoce direcciones en el texto.
- `mtponct:` Reconoce signos de puntuación compuestos por más de un *token*.
- `numprefix:` Reconoce prefijos numéricos. Por ejemplo *"4"* en el texto *"tiene 4-más items"*.
- `number:` Se encarga del reconocimiento de números.
- `smiley:` Reconocimiento de los símbolos gestuales *smileys* (*";)"*,*"xD"*).
- `formatGARe:` Reconocimiento de palabras formateadas (*"pseudo-ciencia"*, *"_palabra_"* o *"*palabra**"*).
- `remove_inner_1:` Encapsulamiento de entidades con nombre.
- `segment:` Es el segmentador de *tokens*.
- `sigles:` Reconoce las siglas.
- `np:` Reconocimiento de nombres propios.
- `etr:` Reconocimiento de expresiones extranjeras.
- `remove_inner_2:` Encapsulamiento de entidades con nombre.
- `qword:` Reconocimiento de palabras citadas.
- `ordinal:` Reconocimiento de ordinales.
- `cardinal:` Reconocimiento de cardinales.
- `normalize:` Normalización de espacios y de porcentajes.
- `text2dag:` Lleva a cabo la transformación del texto de entrada completamente tokenizado en un GAD.
- `epsilonize:` Sustituye las metapalabras (*""*,*"/"*,*"«"*) por la etiqueta _EPSILON.
- `unfold:` Despliega el GAD.

Estos recursos de *SxPipe* pueden estar o no activados. Ello se indica en el apartado `# Component list` del fichero de configuración. Además el orden que presenten los componentes en esa sección, será el orden de aplicación de los mismos sobre el texto a procesar.

Por otro lado, tenemos el fichero de configuración del servidor *Parserd*, denominado `parserd.conf` y ubicado bajo el directorio `/exportbuild/etc`. Éste contiene la siguiente información:

```
## Local options

# Port of the server [default = 8999]
port = 8999
# User of the server [default = nobody]
user = dani
# Group of the server [default = nogroup]
group = dani
# Max number of clients [default = 1]
# maxclients = 2
# Log level [default = 3] set higher or lower
# to get more or less info in parserd.log
# log_level = 3
log_file = /home/dani/exportbuild/var/log/parserd.log
pid_file = /home/dani/exportbuild/var/run/parserd.pid
path  = /home/dani/exportbuild/bin
path  = /usr/local/bin

## Parser list
parsers = spmgtel spmgtelr

[spmgtel]
    label = SPMG TAG/TIG - Hybrid Strategy
            - SPMG Lexer
    cmd = /home/dani/exportbuild/bin/spmg_parser
    options = -forest
    forest = lp
    examples = /home/dani/exportbuild/share/spmg/
               tests.txt
    lexer = persistent /home/dani/exportbuild/bin/
            spmg_lexer
    origin = spmg-0.1
    language = spanish

[spmgtelr]
    label = SPMG TAG/TIG - Hybrid Strategy
            - SPMG Lexer - Robust
    cmd = /home/dani/exportbuild/bin/spmg_parser
    options = -forest -robust
    forest = lp
```

```
examples = /home/dani/exportbuild/share/spmg/
           tests.txt
lexer = persistent /home/dani/exportbuild/bin/
        spmg_lexer
origin = spmg-0.1
language = spanish
```

Donde, en la sección `## Local options` se recoge:

- `port`: Este parámetro indica el puerto de escucha del servidor. El puerto por defecto es el 8999.
- `user`: Recoge el usuario con permisos para manejar el servidor. Por defecto, el servidor no tiene ningún usuario.
- `group`: Recoge el grupo con permisos para gestionar el servidor. Por defecto, el servidor no tiene ningún grupo concreto .
- `maxclients`: Indica cual es el número máximo de clientes que pueden acceder de forma simultánea al servidor. Por defecto, el número máximo de clientes simultáneos es 2.
- `log_level`: Indica el nivel de detalle registrado en el *log* del servidor. Por defecto, tiene un detalle de nivel 3, el máximo permitido.
- `log_file`: Recoge la ruta donde se ubica el fichero de *log* del servidor y cuál es su nombre. Por defecto, el fichero de *log* se denominará `parserd.log` y se ubicará bajo el directorio `/home/dani/exportbuild/var/log`.
- `pid_file`: Recoge la ruta donde se ubica el fichero que recoge el PID (*Process IDentification*) del servidor y cuál es su nombre. Por defecto, este fichero se denominará `parserd.pid` y se ubicará bajo el directorio `/home/dani/exportbuild/var/run`.
- `path`: Recoge la ruta donde se encuentra el fichero binario del servidor, así como el directorio `bin` del usuario local.

En la sección `## Parser list` se listan los analizadores registrados en el servidor. Concretamente, en el parámetro `parsers` de esta sección, se indican los identificadores de los analizadores registrados, que posteriormente son detallados. La información presente en el fichero `register_parserd.conf`, expuesto anteriormente, es utilizada para registrar en el servidor *Parserd* los analizadores descritos en él[6].

[6] En el apartado 11.1 del *Manual de Usuario* se detalla el uso de este fichero.

8.4. Otros detalles de implementación

Un dato relevante acerca del sistema desarrollado es el juego de codificación de caracteres empleado. En este caso ha sido ISO-8859-1.

ISO-8859-1 es una norma de la ISO[7] que define la codificación del alfabeto latino, incluyendo los diacríticos (como letras acentuadas, *"ñ"*, *"ç"*) y letras especiales (como *" β"*, *"ϕ"*), necesarios para la escritura de la mayor parte de las lenguas originarias de Europa occidental como por ejemplo: español, gallego, catalán, vasco, alemán, francés, inglés, italiano, portugués, entre otros. También se conoce como *Alfabeto Latino nº1* o *ISO Latín 1*.

Esta norma pertenece al grupo de juegos de caracteres de la ISO conocidos como ISO/DEC 8859. Éste se caracteriza por poseer la codificación ASCII en su rango inicial (128 caracteres) y otros 128 caracteres para cada codificación, con lo que en total utilizan 8 bits.

Ubuntu[8] usa UTF-8[9] como juego de caracteres predeterminado, así que se debe cambiar la configuración para establecer por defecto la codificación ISO-8859-1.

[7] `http://www.iso.org`
[8] `http://www.ubuntu.com`
[9] `http://www.utf8.com`

Capítulo 9

Pruebas

En esta sección se detallan las pruebas llevadas a cabo para validar el correcto funcionamiento e integración del *software* desarrollado. Estas pruebas han sido realizadas paralelamente con el proceso de implementación del sistema.

Una prueba se puede definir como una ejecución del programa con la intención de encontrar errores previos a la entrega al usuario. Se trata de buscar defectos en nuestro *software*.

Una vez comprobado el correcto funcionamiento del sistema, es necesario evaluar su rendimiento durante la tarea para la que fue diseñado: el procesamiento del lenguaje natural. En este sentido, se realizaron una serie de pruebas sobre *corpora* destinados a este fin, para a continuación, llevar a cabo una comparación con los resultados obtenidos por el sistema equivalente francés.

9.1. Pruebas de unidad

Estas pruebas se centran en cada unidad de *software* o módulo individualmente, realizando pruebas de caja blanca y de caja negra. Se tiene que probar que la información que fluye entre los módulos del sistema se hace de forma adecuada hacia y desde la unidad de programa que está siendo probada. Es necesario, por lo tanto, testear cada componente de forma individual. Las pruebas de unidad se llevaron a cabo combinando pruebas de caja negra, las cuales se realizaron sobre la interfaz del *software*, con pruebas de caja blanca, que se basan en el análisis del código fuente.

9.1.1. Pruebas de caja blanca

Las pruebas de caja blanca se encargan de validar la lógica interna de un módulo. Su realización debe garantizar que:

- Se recorren por lo menos todos los caminos de ejecución independientes de cada módulo.
- Se ejecutan todas las decisiones lógicas en su parte verdadera y en su parte falsa.
- Se recorren todos sus bucles.
- Se utilizan las estructuras de datos internas para garantizar su validez.

Todas estas pruebas fueron realizadas durante la implementación del sistema, comprobando que los módulos implementados cumplían los requisitos que se acaban de describir.

Este tipo de pruebas resulta muy costoso y no se suelen llevar a cabo todos los caminos lógicos del programa, es por esto que se combinan con la utilización de pruebas de caja negra.

9.1.2. Pruebas de caja negra

Las pruebas de caja negra se centran en probar que se verifican los requisitos funcionales del *software*, es decir, que las funciones para las cuales se ha diseñado el sistema se cumplen sin errores. Además, con ellas también se pretende detectar errores de interacción y respuesta de la interfaz de usuario. Los errores que se pretenden detectar mediante las técnicas de caja negra son:

- Funciones incorrectas o ausentes.
- Errores de rendimiento.
- Errores de inicialización o terminación.

Este tipo de pruebas se reduce a comprobar la correcta introducción de datos en el sistema y verificar que las respuestas del sistema son adecuadas a estas entradas.

Las pruebas de caja negra realizadas se han centrado en comprobar la integridad de los datos de entrada del sistema. Este puede recibir datos de dos interfaces diferentes: bajo línea de comandos (*Callparser*) y mediante una interfaz *web* (*WebParser*). Dado que son dos interfaces de distinta naturaleza, las pruebas planteadas sobre cada una de ellas serán diferentes. En este sentido, se han verificado los siguientes aspectos para cada interfaz.

9.1.2.1. Pruebas de caja negra sobre Callparser

Sobre el cliente *Callparser* se han realizado las pruebas descritas desde la tabla 9.1 hasta la tabla 9.6.

Título	**No seleccionar un analizador**
Resultado	Existe un analizador seleccionado por defecto en el sistema.
Evaluación	Correcto

Tabla 9.1: Prueba de caja negra *No seleccionar un analizador*.

Título	**Seleccionar un analizador no registrado**
Resultado	Se muestra un mensaje de error, indicando que el analizador seleccionado no es correcto.
Evaluación	Correcto

Tabla 9.2: Prueba de caja negra *Seleccionar un analizador no registrado*.

Título	**No seleccionar un formato de salida**
Resultado	El sistema no ofrece ninguna representación del análisis.
Evaluación	Correcto

Tabla 9.3: Prueba de caja negra *No seleccionar un formato de salida*.

Título	**Seleccionar un formato de salida o una opción no disponible**
Resultado	Se muestra un mensaje de error, indicando que el formato o la opción seleccionada no es correcta.
Evaluación	Correcto

Tabla 9.4: Prueba de caja negra *Seleccionar un formato de salida o una opción no disponible*.

Título	**No introducir una frase o texto a analizar**
Resultado	El cliente no realiza solicitud al servidor si no se introduce ningún texto a analizar.
Evaluación	Correcto

Tabla 9.5: Prueba de caja negra *No introducir una frase o texto a analizar*.

Título	**Introducir una ruta de fichero incorrecta**
Resultado	Se muestra un mensaje de error, indicando que la ruta introducida no es correcta.
Evaluación	Correcto

Tabla 9.6: Prueba de caja negra *Introducir una ruta de fichero incorrecta.*

9.1.2.2. Pruebas de caja negra sobre WebParser

Sobre el cliente *WebParser* se han realizado las pruebas descritas desde la tabla 9.7 hasta la tabla 9.10.

Título	**No seleccionar un analizador**
Resultado	Existe un analizador seleccionado por defecto en el sistema.
Evaluación	Correcto

Tabla 9.7: Prueba de caja negra *No seleccionar un analizador.*

Título	**No seleccionar un formato de salida**
Resultado	Existe un formato de salida seleccionado por defecto en el sistema.
Evaluación	Correcto

Tabla 9.8: Prueba de caja negra *No seleccionar un formato de salida.*

Título	**No introducir una frase o texto a analizar**
Resultado	El cliente no realiza solicitud al servidor si no se introduce ningún texto a analizar.
Evaluación	Correcto

Tabla 9.9: Prueba de caja negra *No introducir una frase o texto a analizar.*

Título	**Cargar un fichero de texto con un formato erróneo**
Resultado	Se muestra un mensaje de error, indicando que se debe introducir un fichero con el formato correcto.
Evaluación	Correcto

Tabla 9.10: Prueba de caja negra *Carga un fichero de texto con un formato erróneo.*

9.2. Pruebas de integración

Las pruebas de integración se llevan a cabo durante la construcción del sistema e involucran los diferentes módulos empleados y su cometido final es la verificación del sistema como conjunto.

Dado que uno de los objetivos de este proyecto es la integración de varios recursos se ha comprobado, de forma individual y paralela a la incorporación de cada módulo, que el funcionamiento era satisfactorio. Finalmente, se han hecho una serie de pruebas globales obteniendo los resultados esperados.

9.3. Pruebas de procesamiento lingüístico

Dentro del marco de investigación en el que se encuentra ubicado el presente proyecto tienen una mayor relevancia aquellas pruebas que nos proporcionan resultados sobre la actividad del sistema, es decir, los resultados obtenidos durante el procesamiento del español.

9.3.1. Metodología de evaluación

Para llevar a cabo la presente evaluación se ha empleado un equipo informático con las siguientes características:

- Procesador Intel Core 2 Quad Q9550 2,83GHz
- 4096Mb de RAM
- 500 Gigabytes de Disco Duro
- Ubuntu 9.04

En las pruebas realizadas se pretende evaluar el rendimiento del sistema durante el análisis de textos en castellano. Para ello se han utilizado diferentes *corpora* en castellano, donde cada fichero que los compone dispone

de una única oración por cada línea del mismo. Los *corpora* utilizados en las pruebas han sido los siguientes:

- **Corpus propio:** Se trata de un conjunto de oraciones seleccionadas personalmente. Trata de recoger una gran variedad de fenómenos sintácticos con el fin de testear la cobertura del analizador sintáctico. Contiene 123 oraciones.
- **EUROTRA:** Es un *corpus* originario del proyecto EUROTRA, establecido y financiado por la Comisión Europea desde finales de los 70 hasta 1994. Su finalidad era la de crear un sistema de traducción automática entre las siete, y después, nueve lenguas de la Comisión Europea. Contiene 194 oraciones.
- **Europarl 96:** Se trata del *Europarl parallel corpus*[1]. Este ha sido extraído de los procedimientos que se llevan a cabo en el Parlamento Europeo. Incluye versiones en 11 lenguas europeas: Románicas (francés, italiano, español, portugués), Germánicas (ingles, holandés, alemán, danés, sueco), griego y finlandés. El objetivo del mismo es proveer de oraciones de texto alineadas para sistemas estadísticos de traducción automática. En este caso concreto, se trata del conjunto de documentos en lengua española y del año 1996. Contiene 208.223 oraciones. Las oraciones presentes en este *corpus* son especialmente complejas, puesto que son considerablemente extensas (40 palabras/oración aproximadamente) y su vocabulario y estructura sintáctica no son triviales.
- **Europarl 97:** Fragmento perteneciente al *Europarl parallel corpus* del año 1997. Contiene 254.575 oraciones.
- **Corpus del Real Jardín Botánico:** Este contiene diversos textos científicos que recogen información relativa a diferentes familias y especies de plantas. En él priman las descripciones compuestas de oraciones con una gran cantidad de enumeraciones de sintagmas nominales. Además el vocabulario específico del mismo, conllevó un importante incremento de nuevos adjetivos y, particularmente, sustantivos en el lexicón LEFFE. Contiene 528 oraciones.
- **Corpus del Museo de Ciencias Naturales:** Recoge descripciones de diferentes familias y especies animales. Del mismo modo que para el *corpus* anterior, para su procesado ha sido necesario enriquecer el léxico base. Contiene 335 oraciones.

Concretamente para *corpora* excesivamente grandes, como es el caso de *Europarl 96* y *Europarl 97*, no fue posible analizar la totalidad de los mismos.

[1] `http://www.statmt.org/europarl/`

Para conseguir esta tarea, sería necesario disponer de un *cluster* de máquinas destinadas exclusivamente al procesamiento de estos *corpora*. Por ello se ha seleccionado en los *corpora* implicados, una carga de trabajo soportada por el equipo utilizado en las pruebas. Para *Europarl 96* se han tenido en cuenta 282 oraciones y para *Europarl 97*, 455 oraciones.

Los criterios de evaluación utilizados durante las pruebas realizadas, con el fin de testear el rendimiento del sistema construido, han sido los siguientes:

- **Cobertura:** Número de frases del *corpus* que el sistema es capaz de analizar de forma correcta. Por un análisis correcto de una oración, se entiende que todos los fenómenos sintácticos presentes en la oración estén recogidos en su totalidad en la metagramática SPMG. Esta medida se puede expresar mediante el número de oraciones del *corpus* analizadas correctamente e incorrectamente; o indicando el porcentaje de oraciones pertenecientes a un determinado *corpus* que es capaz de procesar con éxito. La medición de la cobertura se hace a nivel sintáctico, puesto que si el sistema desconoce una palabra del *corpus*, le serán asignadas cuatro posibles categorías léxicas (sustantivo, verbo, adjetivo y adverbio) y el sistema continuará con el procesamiento. Por lo tanto, el analizador sintáctico siempre maneja una cobertura sobre el léxico del 100 %, a pesar de que existan palabras desconocidas en el texto. Ello provoca que en algunos casos, el fallo del análisis sintáctico sea originado por el no etiquetado correcto de una determinada palabra clave en una estructura sintáctica concreta. En este sentido, se ha comprobado previamente que el lexicón ofreciese una cobertura del 100 % del léxico presente en los distintos *corpora* analizados.

- **Tiempo medio de ejecución por oración:** Tiempo medio empleado por la cadena para procesar cada una de las oraciones pertenecientes a un *corpus*.

- **Tasa media de ambigüedad:** La tasa de ambigüedad para una palabra α, se define como el número medio de arcos de dependencias que llegan a una palabra (nodo en el grafo de dependencias) menos 1. Para un análisis no ambiguo, a todas las palabras (nodos), salvo la *cabeza*[2] de la frase, debe llegar una sola dependencia. En este caso, la tasa de ambigüedad es cero. Posteriormente, para obtener la ambigüedad de una oración, se calcula la media de las tasas de ambigüedad correspondientes a cada palabra de la misma.

  Esta medida permite evaluar el grado de especialización de la metagramática SPMG para el español, esto es, a *grosso modo*, tratar de medir, ante un determinado fenómeno sintáctico, cuántas estructuras sintácticas recogidas en SPMG se podrían aplicar en ese caso concreto.

[2]Nodo raíz del árbol que cubre el texto de entrada completo.

Lo ideal, y así ocurre en los lenguajes de programación (no ambiguos), sería que cada fenómeno sintáctico del texto a analizar requiriese unívocamente de una estructura concreta del analizador sintáctico.

Esta medida se realiza a nivel de oración, por lo tanto, en el caso de procesar un *corpus*, se ofrece la media resultante de las tasas medias de ambigüedad correspondientes a cada una de las oraciones que componen el *corpus* objeto de análisis.

Consideraciones a tener en cuenta:

- La evaluación de la cadena se hace a nivel sintáctico, es decir, el principal componente, cuyos resultados se están evaluando, es el analizador sintáctico *SPMG Parser*.
- Desde el punto de vista de la cobertura, es indiferente que se emplee el analizador `spmgtel` o `spmgtelr`, ya que a pesar de tener la robustez activada, el sistema cuenta una oración parcialmente analizada como una oración procesada erróneamente. Sin embargo, centrándose en el tiempo de ejecución, se ha seleccionado para las pruebas el analizador `spmgtel`, que consume menos tiempo y recursos en el análisis.
- De los dos clientes disponibles del sistema, se ha utilizado *Callparser*. Este ofrece el procesamiento de todas las frases de un fichero de forma automática.
- Las pruebas aplicadas sobre los *corpora* previamente presentados, han consistido en utilizar el cliente *Callparser* para solicitar al servidor *Parserd* el procesamiento lingüístico de los mismos.
- Además, se le indica al cliente *Callparser* que únicamente ofrezca como salida las estadísticas del análisis realizado. Éstas serán la fuente de los resultados expuestos a continuación.

9.3.2. Resultados obtenidos

En la tabla 9.11 se muestran los resultados del análisis lingüístico realizado sobre los *corpora* reunidos para este fin, donde:

- **Nº frases:** Es el número de frases del *corpus* en concreto.
- **% Cov.:** El porcentaje de oraciones analizadas correctamente de un determinado *corpus*.
- **T. medio (seg.):** Es el tiempo medio empleado para analizar cada frase de un *corpus*.
- **T. amb.:** Tasa media de ambigüedad de cada oración del *corpus*.

Corpus	Nº frases	% Cov.	T. medio (seg.)	T. amb.
Corpus Propio	123	100	0.60	4.4
EUROTRA	194	100	2.29	5.35
Europarl 96	282	68.71	50.48	5.4
Europarl 97	455	67.58	54.17	6.1
R. J. Botánico	528	100	5.34	4.6
M. C. Naturales	335	100	37.32	7.2

Tabla 9.11: Resultados de las pruebas de procesamiento lingüístico.

9.3.3. SPMG vs FRMG

Con el fin de comprender en que punto se encuentra la cadena de procesamiento española en comparación con la francesa, es necesario disponer de los resultados de esta última para realizar una comparación con la primera. En la tabla 9.12 se muestran los resultados obtenidos por la cadena ALPAGE en 2009 (De la Clegerie et al., 2009)[3]:

Corpus	Nº frases	% Cov.	T. medio (seg.)	T. amb.
EUROTRA	334	100	0.15	0.63
TSNLP	1161	95.07	0.07	0.46
EasyDev	3879	64.73	0.93	1.04
JRCacquis	1.1M	51.26	1.41	1.1
Europarl	0.8M	70.19	1.69	1.36
EstRep	1.6M	67.05	0.69	0.92
Wikipedia	2.2M	69.11	0.49	0.87
Wikisource	1.5M	61.08	0.71	0.89
AFP	1.6M	52.15	0.51	1.06

Tabla 9.12: Resultados de la cadena ALPAGE.

Para realizar estas pruebas, el equipo ALPAGE ha dispuesto de una configuración de *cluster* entre varias máquinas destinadas exclusivamente al procesamiento de textos. Por ello, además de conseguir mejores resultados en cuanto a tiempo de ejecución, también hace posible que puedan procesar *corpora* de mayor tamaño. Es por ello, que la comparación entre ambas cadenas no va a ser excesivamente estricta. Simplemente, se tratará de esbozar aquellas diferencias observables a simple vista entre los resultados obtenidos por cada herramienta de PLN.

[3]Donde 1.1M son 1.1 millones de oraciones.

En este sentido, se basa la comparación en aquellos *corpora* que han sido analizados por ambas cadenas de procesamiento lingüístico. Éstos son los *corpora* EUROTRA y *Europarl*. No ha sido posible procesar un número mayor de *corpora* comunes, puesto que muchos de ellos no estaban ya disponibles, o como es el caso del TSLNP, no para el idioma español. Además, aquellos *corpora* analizados en común, no presentan el mismo número de frases, pero si la misma naturaleza y complejidad sintáctica en sus oraciones. Y esto último es el punto de partida de la comparación: comprobar la efectividad de ambos sistemas ante oraciones de complejidad similar.

Cabe, por lo tanto, destacar que las pruebas realizadas para determinar la efectividad de procesamiento del sistema desarrollado no son excesivamente rigurosas. Mientras que los resultados del equipo francés se obtienen en algunos casos sobre *corpora* con un tamaño que ronda el millón de oraciones, para el sistema español, los textos no superan las 550 oraciones en ningún caso, bien por falta de recursos lingüísticos, o bien por carencias de rendimiento del equipo en el caso del *corpus Europarl*. Sin embargo, se ha considerado que, dado que las oraciones presentan una complejidad similar, no es relevante que el número de oraciones sea distinto, ya que los resultados no cambiarían significativamente[4].

En la tabla 9.13 se incluyen los resultados de ambas cadenas para el procesamiento del *corpus* EUROTRA. Se identifican las cadenas a comparar por el nombre de la metagramática base de cada una, FRMG (*FRench Metagrammar*) para el francés y SPMG (*SPanish Metagrammar*) para el castellano.

Analizador	**Nº frases**	**% Cov.**	**T. medio (seg.)**	**T. amb.**
FRMG	334	100	0.15	0.63
SPMG	194	100	2.29	5.35

Tabla 9.13: Resultados de procesamiento para el *corpus* EUROTRA.

En la tabla 9.14 se incluyen los resultados de ambas cadenas para el procesamiento del *corpus Europarl*. En este caso, en SPMG se han procesado dos versiones de este *corpus*, *Europarl 96* y *Europarl 97*, denominados en la tabla como SPMG 96 y SPMG 97, respectivamente.

[4]Los resultados obtenidos para una muestra de 500 elementos (oraciones), en este caso concreto, no se van a ver modificados significativamente, si el análisis se realiza sobre 5 millones de frases.

Analizador	**Nº frases**	**% Cov.**	**T. medio (seg.)**	**T. amb.**
FRMG	0.8M	70.19	1.69	1.36
SPMG 96	282	68.71	50.48	5.4
SPMG 97	455	67.58	54.17	6.1

Tabla 9.14: Resultados de procesamiento para el *corpus Europarl*.

Las conclusiones de la comparativa realizada están presentes en el apartado 5.2 de la *Memoria*.

Parte III

Manual de Usuario

Capítulo 10

Instalación de la aplicación

En este capítulo se detalla la información referente a la instalación y la puesta en funcionamiento del sistema, así como los requisitos *software* y *hardware* necesarios para el correcto funcionamiento del mismo.

10.1. Requisitos

En esta sección se describen los requisitos, tanto *hardware* como *software*, necesarios para poner en marcha el sistema.

10.1.1. Requisitos hardware

Los requisitos *hardware* mínimos para el correcto funcionamiento del sistema son los siguientes:

- **CPU:** La familia del procesador utilizada es indiferente, pero la frecuencia de reloj debe ser igual o superior a 500 GHz (recomendable 1.5 GHz).
- **Memoria RAM:** 512 MB (recomendable 1 GB).
- **Disco duro:** 20 GB.
- **Unidad de DVD-ROM:** Necesaria para instalar la aplicación.

Para la máquina cliente, basta que permita ejecutar un *script* en *Perl* y los recursos *Graphviz* e *ImageMagick* (para *Callparser*), o un navegador *web* (para *WebParser*).

10.1.2. Requisitos software

Los requisitos *software* mínimos para el correcto funcionamiento del sistema son los siguientes:

- Sistema operativo GNU/Linux. La mayoría de herramientas empleadas han sido diseñadas para estos sistemas. Se recomienda el uso de una distribución basada en *Debian*, tal como *Ubuntu* o *Kubuntu*, puesto que ésta ha sido la utilizada durante el desarrollo de este sistema. Es indiferente la arquitectura del sistema operativo. El sistema desarrollado funciona tanto en arquitectura de 32 bits como de 64 bits.
- Perl >= 5.8
- AppConfig
- IPC::Run
- pkg-config 0.15
- g++
- perlcc
- recode
- xsltproc
- bison >2.3
- flex
- telnet
- ImageMagick
- Graphviz
- Servidor HTTP Apache2 (con el módulo de *Perl* para Apache2)

El cliente debe permitir el soporte de los ficheros implementados en *Perl* y disponer de los recursos gráficos *Graphviz* e *ImageMagick*[1], en el caso de utilizar el cliente *Callparser*. Para el cliente *web*, *WebParser*, bastará con un navegador.

10.2. Instalación

Para la instalación de los diferentes paquetes que componen el sistema desarrollado, se ha construido un *script Perl* encargado de ese cometido. Éste está presente en el DVD-ROM adjunto a esta documentación. Se trata de `install.pl`. Su función es la de instalar automáticamente cada paquete y las dependencias del mismo para su correcto funcionamiento.

La instalación de la cadena de procesamiento lingüístico pasa por los siguientes pasos:

[1] Ambos recursos necesarios para manejar los grafos de dependencias.

1. Introducir el DVD-ROM adjunto.
2. Copiar la carpeta `/exportbuild`, ubicada en el directorio `/src` del DVD-ROM, en el directorio personal del usuario.
3. Copiar el *script* `install.pl` presente en el DVD-ROM en el directorio personal del usuario.
4. Ejecutar el *script* `install.pl` de la siguiente forma:

```
./install.pl
```

Por defecto, la aplicación se instala bajo el directorio `${HOME}/exportbuild`. En el caso de que la carpeta personal fuese `/home/usuario`, la aplicación se instalaría en el directorio `/home/usuario/exportbuild`.

Cuando finaliza la instalación, `install.pl` restablece las variables de entorno del usuario. De modo que, para hacer uso de la aplicación es necesario teclear:

```
source ${PREFIX}/sbin/setenv.sh
```

Donde ${PREFIX} es el directorio bajo el cual se encuentra instalada la aplicación: `${HOME}/exportbuild`, es decir, `/home/usuario/exportbuild`.

Está acción configurará las variables de entorno del usuario para poder utilizar correctamente el sistema previamente instalado. Este entorno únicamente se mantiene para la sesión actual. Si se desea establecerlo de forma permanente, es necesario añadir el contenido del fichero `setenv.sh` en el fichero `.bashrc` presente en la carpeta personal del usuario.

Si se produce cualquier error durante la instalación, es recomendable acudir al fichero `install.log`, que se encuentra en el mismo directorio que el *script* `install.pl`. Éste muestra de forma detallada cuál ha sido el causante del error producido durante la instalación.

El *script* `install.pl` presenta las siguientes opciones:

- `-prefix = <`*`path`*`>`. Con esta opción se puede indicar el directorio donde deseamos que se produzca la instalación de la aplicación. Previamente, será necesario copiar la carpeta del DVD-ROM `/exportbuild` en el directorio seleccionado y renombrarla en caso de ser necesario, ya que, por defecto, la aplicación se instala en `$HOME/exportbuild`.
- `-port|p = <`*`number`*`>`. Con ello se indica el puerto de escucha que va a utilizar el servidor *Parserd*. Por defecto, usa el puerto 8999.
- `-user|u = <`*`user`*`>` y `-group|g = <`*`group`*`>`. Estas opciones se utilizan para señalar el usuario y grupo bajo el que se encuentra el servidor. Por defecto, se utiliza el usuario que esta ejecutando el *script* de instalación y el primer grupo al que éste pertenece.

- `-package|pkg = <package>`. Esta opción permite instalar los paquetes de forma individual. Ello puede ser útil si se ha producido alguna actualización de un paquete concreto. Esta opción se encarga de instalar también los paquetes de los que depende. Por ejemplo:

```
./install.pl -pkg=DyALog -pkg=spmg
```

  Se instalan únicamente los paquetes `DyALog`, `spmg` y aquellos de los que dependan.

- `-skippkg = <package>`. Si el número de paquetes concretos que se desea instalar es considerable, se puede emplear esta opción. Ella permite indicar qué paquete no se desea instalar. El resto de paquetes incluidos en `install.pl` serán todos instalados. Por ejemplo:

```
./install.pl -skippkg=DyALog
```

  Se instalarán todos los paquetes excepto el paquete `DyALog`.

- `-skipdep = <package>`. Si se desea actualizar un paquete, pero no modificar los paquetes de los que es dependiente, se debe utilizar esta opción. Por ejemplo:

```
./install.pl -pkg=spmg -skipdep
```

  Se actualiza el paquete `spmg`, pero no se vuelven a instalar sus dependencias.

- `-linux32`. El sistema desarrollado se basa en una tecnología no compatible con máquinas de 64 bits. Sin embargo, si se permite ejecutar la aplicación sobre éstas mediante un módulo de emulación de 32 bits. Para activarlo debemos indicar esta opción en el *script* `install.pl`.

- `-dependencies|dep`. Muestra los requisitos *software* que todavía no se encuentran instalados en nuestro equipo para el correcto funcionamiento de la aplicación.

- `-version|v`. Muestra la versión de los paquetes que componen el sistema.

- `-info`. Ofrece una descripción de los paquetes que integran el sistema.

- `-log = <log file>`. Sirve para indicar cuál va a ser el fichero utilizado como *log* de la instalación. Por defecto, es `install.log`.

10.2.1. Instalación de la interfaz Web

El *script* `install.pl` no recoge la instalación de los recursos necesarios para que el cliente *WebParser* funcione de forma correcta. Con el fin de que un navegador *web* tenga acceso a la cadena de procesamiento lingüístico para el español, es necesario que el *script* `webparser.pl` se encuentre en un servidor *web*. De tal forma, que este cliente recoja las peticiones vía HTTP que llegan al servidor *web* y las transmita al servidor *Parserd* encargado del análisis.

Para ello, en primer lugar, es necesario instalar un servidor *web*. En el desarrollo de este proyecto se ha utilizado *Apache2*. Se incluye de forma detallada en el *Apéndice A*, la instalación y puesta en marcha del servidor *Apache2* para *Ubuntu*.

A continuación, es imprescindible instalar el módulo de *Apache*, `mod_perl`, que hace posible que el servidor maneje *scripts* desarrollados en *Perl*.

Una vez instalado el servidor *web Apache2* y el módulo `mod_perl`, se debe crear un fichero de configuración, denominado `www-parserd.conf`. Éste debe encontrarse bajo el directorio `/etc/apache2/mods-available` y contener el siguiente código:

```
<IfModule mod_perl.c>
   Alias /cgi-bin/ "${PREFIX}/var/www/perl/"
   PerlModule ModPerl::Registry
   <Directory "${PREFIX}/var/www/perl/">
     <FilesMatch "\.(pl|cgi)$">
       SetHandler perl-script
       PerlHandler ModPerl::Registry
       PerlSetEnv PERL5LIB ${PREFIX}/lib/perl5/
site_perl:${PREFIX}/lib/perl5:${PREFIX}/lib/perl5/
site_perl/5.10.0:${PREFIX}/lib/perl/5.10.0/auto:
${PREFIX}/share/automake:${PREFIX}/share/perl/
5.10.0:${PREFIX}/lib/perl/5.10.0:${PREFIX}/lib/
perl:${PREFIX}/lib/perl/5.8:${PREFIX}/share/
perl/5.8::/usr/lib/perl5/vendor_perl/5.8.8
          PerlSetEnv DOTCMD /usr/bin/dot
          PerlSetEnv XLTPROC /usr/bin/xsltproc
          PerlSendHeader On
    </FilesMatch>
    Options ExecCGI -Indexes MultiViews
    Order deny,allow
    Allow from all
</Directory>
```

Donde `${PREFIX}` es la ruta de la instalación local de la cadena de procesamiento lingüístico.

Para saber cual es el contenido de la variable `PERL5LIB`, puede usar el siguiente comando:

```
echo $PERL5LIB
```

Además, es necesario indicarle al servidor el directorio local donde se encuentran los *scripts Perl* a ejecutar y cuál va a ser el directorio púbico de los mismos. Ello se incluye en el fichero `default` presente en el directorio `/etc/apache2/sites-available`. En el caso de que se ejecute, tanto el cliente *web* (en el servidor *Apache2*) como el servidor *Parserd* en la misma máquina[2], el directorio de ejecución se encuentra dentro del directorio de la aplicación `${PREFIX}/var/www/perl/`. Y el directorio público puede ser, por ejemplo, `/cgi-bin`. En este sentido, se debe introducir la siguiente línea en el mencionado fichero `default`:

```
ScriptAlias /cgi-bin/ ${PREFIX}/var/www/perl/
```

Quedando el fichero `default` con el siguiente aspecto:

```
<VirtualHost *:80>
   ServerAdmin webmaster@localhost

   DocumentRoot ${PREFIX}/var/www
   <Directory />
      Options FollowSymLinks
      AllowOverride None
   </Directory>
   <Directory /var/www/>
      Options Indexes FollowSymLinks MultiViews
      AllowOverride None
      Order allow,deny
      allow from all
   </Directory>

   ScriptAlias /cgi-bin/ ${PREFIX}/var/www/perl/
   <Directory "${PREFIX}/var/www/perl/">
      AllowOverride None
      Options +ExecCGI -MultiViews
              +SymLinksIfOwnerMatch
```

[2]Existe la posibilidad de que el servidor *Apache2* (con el cliente `webparser.pl`) se encuentre instalado en otra máquina distinta del servidor *Parserd*. Esta configuración se puede ver en el apartado 2.4 de la *Memoria*.

```
      Order allow,deny
      Allow from all
   </Directory>


   ErrorLog /var/log/apache2/error.log

   # Possible values include: debug, info,
        # notice, warn, error, crit,
   # alert, emerg.
   LogLevel warn

   CustomLog /var/log/apache2/access.log combined

    Alias /doc/ "/usr/share/doc/"
    <Directory "/usr/share/doc/">
        Options Indexes MultiViews FollowSymLinks
        AllowOverride None
        Order deny,allow
        Deny from all
        Allow from 127.0.0.0/255.0.0.0 ::1/128
    </Directory>

</VirtualHost>
```

En este fichero además se indica que el directorio público del servidor para documentos HTML[3] (`DocumentRoot`) es el directorio `${PREFIX}/var/www`.

De tal forma que, para ejecutar el cliente `webparser.pl` presente en el directorio local `${PREFIX}/var/www/perl/`, se debe introducir en el navegador la siguiente URL, indicando la dirección IP de la máquina donde se encuentra instalado el servidor *Apache2*:

`http://<`*dirección ip*`>/cgi-bin/webparser.pl`

En el supuesto de que se acceda desde la misma máquina, se introduce la siguiente URL:

`http://localhost/cgi-bin/webparser.pl`

Obteniendo en el navegador la interfaz del cliente *WebParser* tal y como aparece en la figura 10.1.

[3] *HyperText Markup Language*, Lenguaje de Marcado de Hipertexto.

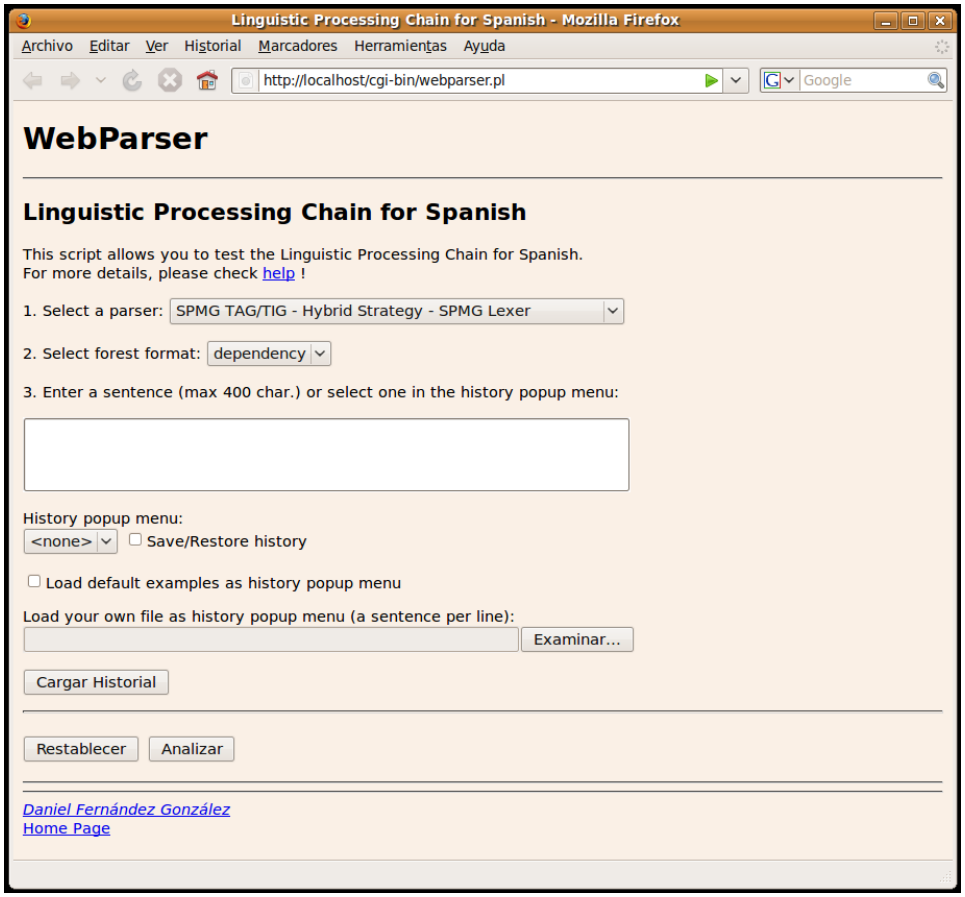


Figura 10.1: Interfaz *web* del cliente *WebParser*.

Capítulo 11

Uso de la aplicación

En este capítulo se describe de manera detallada la forma correcta de uso de la aplicación. Las explicaciones irán acompañadas de ilustraciones para facilitar la comprensión de las mismas. Éste se divide en el servidor *Parserd* y los dos principales clientes que el usuario tiene a su disposición: *Callparser* y *WebParser*.

Antes de utilizar el *software* implementado, es necesario establecer las variables de entorno mediante el siguiente comando:

```
source ${PREFIX}/sbin/setenv.sh
```

Donde `${PREFIX}`, siempre va a ser el directorio donde se encuentra instalada la aplicación. Por defecto es `${HOME}/exportbuild`.

11.1. Parserd

Ambos clientes acceden al servidor de analizadores con el fin de conseguir el procesamiento lingüístico solicitado. Por tanto, inicialmente, es necesario arrancar el servidor. Ello se consigue mediante la orden:

```
${PREFIX}/sbin/parserd_setenv start
```

El servidor se habrá iniciado de forma correcta, si la respuesta del sistema a la orden anterior es la siguiente:

```
Starting parserd: Running the server user=<user>group=<group> [ OK ]
```

Para detener el servidor, se debe teclear la siguiente orden:

```
${PREFIX}/sbin/parserd_setenv stop
```

Y se obtendrá como respuesta:

```
Stopping parserd: [ OK ]
```

En el supuesto de que el servidor se haya cerrado de forma incorrecta, será necesario eliminar el fichero `parserd` presente en el directorio `${PREFIX}/var/lock/subsys/` antes de volver a arrancarlo.

Para registrar o modificar nuevos analizadores en el servidor *Parserd* se utiliza el comando `register_parsers`. La orden tiene la siguiente estructura:

```
register_parsers -a ${PREFIX}/share/<dirMG>/register_parserd.conf
${PREFIX}/etc/parserd.conf
```

Donde *<dirMG>* es el directorio que recoge la metagramática base del analizador a registrar. En el caso de la cadena del español es el directorio `/spmg`. Cada analizador ha de tener un fichero `register_parserd.conf` con la información necesaria para su registro en el servidor [1].

Para modificar cualquier parámetro de configuración del servidor *Parserd* es necesario acceder directamente al fichero de configuración del servidor: `parserd.conf`[2]. Además, si el cambio se produce durante la ejecución, es necesario reiniciarlo para que éste tenga efecto.

11.2. Callparser

Este cliente viene implementado por el *script* en *Perl* `callparser.pl`. Éste no dispone de interfaz gráfica, sino que se utiliza bajo línea de comandos para conectarse por *telnet* al servidor *Parserd*. Inicialmente se encuentra en el directorio `${PREFIX}/bin`.

Las estructuras básicas para ejecutar este cliente, y solicitar procesamiento lingüístico a través de él, son las siguientes:

```
echo <frase>| callparser -in - [options]

cat <fichero>| callparser -in - [options]
```

La primera se utiliza para analizar una única oración, mientras que la segunda procesa todas las oraciones presentes en el fichero indicado.

Para la orden:

```
echo "El niño comió una manzana" | callparser -in -
```

[1] El apartado 8.2 del *Manual Técnico* detalla la interpretación del fichero `register_parserd.conf` de los analizadores fruto de la metragramática SPMG.

[2] El apartado 8.3 del *Manual Técnico* detalla la interpretación del fichero `parserd.conf`.

Se obtendría:

```
ok 1    El niño comió una manzana

Stats: 1 tried 0 failed 0 skipped 100.00% coverage
0.00s avtime

All tests successful !
```

Donde se indica que la oración introducida ha sido analizada de forma correcta, y además se muestran las estadísticas del análisis:

- `tried:` Indica el número de oraciones analizadas.
- `failed:` Muestra el número de oraciones analizadas de forma incorrecta.
- `skipped:` Recoge el número de oraciones no analizadas.
- `coverage:` Cobertura del análisis. Al tratarse de una única oración analizada de forma correcta, la cobertura es del 100 %.
- `avtime:` Recoge el tiempo empleado en analizar la oración. Se encuentra desactivado por defecto. Para activarlo es necesario usar la opción `-time`, descrita más adelante.

Para la orden:

```
cat frases.txt | callparser -in -
```

Se obtendría:

```
ok 1    El niño comió una manzana
ok 1    El hombre cantaba demasiado
ok 1    La casa de la izquierda es roja

Stats: 3 tried 0 failed 0 skipped 100.00% coverage
0.00s avtime

All tests successful !
```

Las opciones del cliente *Callparser* se pueden obtener mediante el comando:

```
perldoc callparser
```

Éstas se detallan en los siguientes subapartados.

11.2.1. date

Esta opción muestra la hora y la fecha actual. Por ejemplo, introduciendo la siguiente orden:

```
echo "El niño comió una manzana" | callparser -in - -date
```

Se obtiene como resultado:

```
ok 1    El niño comió una manzana

<date> 2010-05-03 19:49:09
Stats: 1 tried 0 failed 0 skipped 100.00% coverage
0.00s avtime

All tests successful !
```

11.2.2. errfile

Con esta opción se indica el fichero de error del cliente *Callparser*. En él se recogen las frases cuyo análisis no ha sido correcto. Por ejemplo:

```
echo "El niño comió una manzana" | callparser -in -
-errfile=myErrors
```

Por defecto, el fichero de error se denomina `errors` y se encuentra en el directorio `${PREFIX}/bin` junto con el *script Perl* `callparser.pl`.

11.2.3. log_file

Con la orden siguiente se indica cuál es el fichero de *log* del cliente *Callparser*:

```
echo "El niño comió una manzana" | callparser -in -
-log_file=F<file>
```

Por defecto, el fichero *log* es `callparser.log` y se encuentra bajo el directorio `/tmp`.

11.2.4. parser

El cliente *Callparser* accede por defecto al analizador `spmgtel` registrado en el servidor *Parserd*. Si deseamos que el análisis de la oración o fichero de entrada sea llevado a cabo por otro analizador registrado en el servidor, se debe indicar con la opción `-parser=<name>`, tal y como aparece en el siguiente ejemplo:

```
echo "El niño comió una manzana" | callparser -in -
-parser=miParser
```

11.2.5. host|h y port|p

Por defecto, el servidor se encuentra en la máquina local (`-host=localhost`) y el puerto 8999 (`-port=8999`). Si se desea acceder a un servidor *Parserd* ubicado en una máquina diferente y/o escuchando en otro puerto, debe indicarse de forma explícita con las opciones `-host|h=<host>` y `-port|p=<port>`. Por ejemplo:

```
echo "El niño comió una manzana" | callparser -in -
-host=192.147.87.3 -port=8080
```

11.2.6. range|r

Dado un fichero de entrada, si se desea que únicamente se analice un determinado número de oraciones presentes en el mismo, se utiliza la opción `-range|r=<num>`.

```
cat frases.txt | callparser -in - -range=2
```

Con esta orden únicamente se analizarían las dos primeras oraciones del fichero `frases.txt`. Obteniendo como salida:

```
ok 1    El niño comió una manzana
ok 1    El hombre cantaba demasiado

Stats: 2 tried 0 failed 0 skipped 100.00% coverage
0.00s avtime

All tests successful !
```

11.2.7. stats|s

Con esta opción, el cliente *Callparser* ofrece la tasa media de ambigüedad del análisis realizado. Por ejemplo:

```
echo "El niño comió una manzana" | callparser -in - -stats
```

Esta orden obtiene como salida:

```
ok 1    El niño comió una manzana

Stats: 2 tried 0 failed 0 skipped 100.00% coverage
```

```
0.00s avtime
Stats: 0.4 ambiguity

All tests successful !
```

11.2.8. tagger

Con esta opción se obtienen las palabras que componen la oración de entrada etiquetadas mediante información morfosintáctica. Funcionaría como si de un etiquetador se tratase. Por ejemplo, para la frase presente en la siguiente orden:

```
echo "El niño comió una manzana." | callparser -in - -tagger
```

Se obtiene[3]:

```
ok 1    El niño comió una manzana.

------------  START TAGGER ---------
0  1  det  El  [def : +, dem : -, det : +,
                gender : masc, numberposs : -,
                poss : -, wh : -]
1  2  adj  niño  [gender : masc, person : 3,
                 pre\_det : -]
1  2  nc  niño  [def : +, gender : masc, hum : -,
                person : 3, time : -]
2  3  v  comió  [diathesis : active, lightverb : -,
                mode : indicative, person : 3,
                tense : past-historic]
3  4  det  una  [def : -, dem : -, det : +,
                gender : fem, numberposs : -,
                poss : -, wh : -]
4  5  nc  manzana  [def : +, gender : fem, hum : -,
                   person : 3, time : -]
5  6  \_  .  []
------------  END TAGGER ---------

Stats: 1 tried 0 failed 0 skipped 100.00% coverage
0.00s avtime

All tests successful !
```

[3]Nótese que se intenta eliminar la ambigüedad léxica existente en el texto de entrada. Por ejemplo, únicamente se contempla la posibilidad de que *una* sea determinante y no verbo (lema *unir*).

11.2.9. time

Esta opción activa el dato estadístico `avtime` que recoge el tiempo medio empleado para el análisis de las oraciones de entrada. En el caso de que se trate de una única oración, recogerá el tiempo exacto empleado para procesar esa oración. Mientras que si se trata de un fichero de texto, se indica el tiempo individual empleado para procesar cada frase del mismo y cuál es el tiempo medio necesario para el análisis de cada oración. Por ejemplo, para la orden:

```
echo "El niño comió una manzana" | callparser -in - -time
```

Se obtiene como resultado:

```
ok 1    El niño comió una manzana

Stats: 1 tried 0 failed 0 skipped 100.00% coverage
0.09s avtime

All tests successful !
```

Por otra lado, si la orden contiene un fichero:

```
cat frases.txt | callparser -in - -time
```

El resultado es el siguiente:

```
ok 1    El niño comió una manzana
<time> 0.552
ok 1    El hombre cantaba demasiado
<time> 0.052
ok 1    La casa de la izquierda es roja
<time> 0.352

Stats: 3 tried 0 failed 0 skipped 100.00% coverage
0.32s avtime

All tests successful !
```

11.2.10. timeout

Por defecto, el servidor *Parserd* asigna un tiempo de 200 milisegundos a cada oración. Si ésta excede ese tiempo, se salta a la siguiente oración. Mediante la opción `-timeout=<i>` podemos modificar el tiempo dedicado a cada oración. Por ejemplo, si se quiere que el servidor le conceda un margen de 400 milisegundos a cada oración del fichero a analizar, se escribirá lo siguiente:

```
cat frases.txt | callparser -in - -timeout=400
```

11.2.11. forest

Dada la siguiente orden:

```
echo "El niño comió una manzana" | callparser -in - -forest
```

Se consigue que *Callparser* ofrezca, en formato XML, el bosque compartido de derivaciones que recoge las estructuras sintácticas que describen la oración de entrada. A continuación se muestra un fragmento de la salida:

```
ok 1    El niño comió una manzana
------------  START FOREST ---------
<?xml version="1.0" encoding="ISO-8859-1"
standalone="yes"?>
<!DOCTYPE forest PUBLIC "-//INRIA//DTD TAG
Derivation Forest 1.0
EN" "http://alpage.inria.fr/~clerger/
forest.dtd,xml"><forest>
  <op cat="S" id="1" span="0 6" type="subst">
    <narg type="top">
      <fs>
        <f name="extraction">
          <minus />
          <val>adjx</val>
        </f>
        <f name="gender">
          <val>masc</val>
        </f>
        <f name="inv">
          <minus />
        </f>
        <f name="mode">
          <val>indicative</val>
        </f>
        <f name="sat">
          <plus />
        </f>
        <f name="tense">
          <val>past-historic</val>
        </f>
        <f name="wh">
```

```
          <minus />
        </f>
      </fs>
    </narg>
  <deriv tree="136 V1VMod:agreement clsubj:agreement
ante:clitic_sequence post:clitic_sequence clitics
arg1:collect_real_arg arg2:collect_real_arg arg0:
collect_real_subject arg1:real_group_comp arg2:
real_group_comp ncpred:real_group_comp arg0:
noun:true_subject arg0:post_PP:true_subject arg0:
post_noun:true_subject arg0:post_s:true_subject arg0:
post_v:true_subject arg0:s:true_subject arg0:v:
true_subject Infl:verb_agreement V:verb_agreement
v:verb_agreement V1:verb_agreement_ancestor
arg1:verb_argument_other arg2:verb_argument_other
arg0:verb_argument_subject verb_canonical
verb_categorization_active_acomp1">
      <hypertag>
        <fs>
          <f name="anchor">
            <val>comió</val>
          </f>
          <f name="arg0">
            <fs type="arg">
              <f name="kind">
                <val>subj</val>
              </f>
            </fs>
          </f>
          <f name="arg1">
            <fs type="arg">
              <f name="kind">
                <val>obj</val>
              </f>
              <f name="pcas">
                <minus />
                <val>de</val>
              </f>
            </fs>
          </f>
          <f name="cat">
            <val>v</val>
          </f>
          <f name="diathesis">
```

```
            <val>active</val>
          </f>
        </fs>
      </hypertag>
      <node name="S">
        <op cat="S" id="8" span="5 6 6 6" type="adj">
          <narg type="top">
            <fs>
              <f name="extraction">
                <minus />
                <val>adjx</val>
                <val>cleft</val>
                <val>topic</val>
                <val>wh</val>
              </f>
               ....
```

Se trata de un formato utilizado internamente por el sistema para ofrecer el resultado del análisis. No es de mucha utilidad para los usuarios de la aplicación. Por ello, se transforma esta salida en un grafo de dependencias de fácil comprensión. Sin embargo, a los desarrolladores de una metagramática puede ser les útil comprobar los árboles de derivaciones de salida, con el fin de introducir mejoras sobre la misma.

11.2.12. display|d dep

Ofrece el grafo de dependencias correspondiente al análisis realizado. Por ejemplo:

```
echo "El niño comió una manzana" | callparser -in - -d dep
```

Para la oración *"El niño comió una manzana"*, su grafo de dependencias[4] se muestra en la figura 11.1.

[4]La interpretación del grafo de dependencias se ha detallado en el apartado 2.3.3 de la *Memoria*.

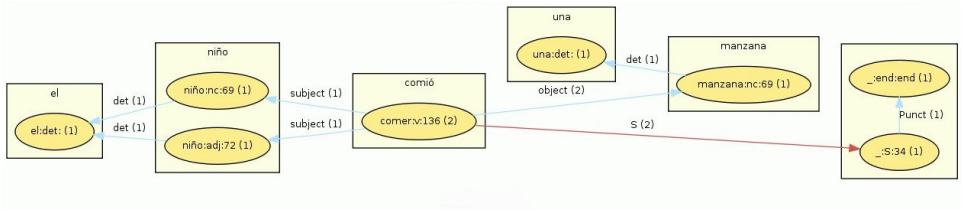


Figura 11.1: Grafo de dependencias de la oración *"El niño comió una manzana"*.

Además, el recurso *ImageMagick*, ofrece la posibilidad de gestionar[5] y editar[6] el grafo de dependencias mediante la ventana de comandos disponible (figura 11.2).

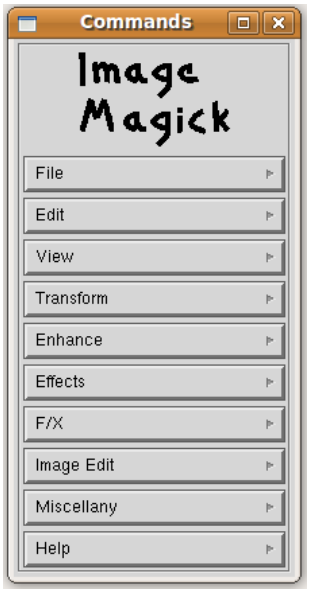


Figura 11.2: Ventana de comandos de *ImageMagick*.

11.2.13. xmldep

Esta opción muestra las dependencias recogidas en el grafo de dependencias en formato XML[7]. Esto es útil para que un sistema de nivel de

[5] Almacenar, imprimir, modificar el formato.

[6] Copiar, cortar, pegar, añadir efectos, dibujar sobre la imagen, entre otros.

[7] La interpretación de las etiquetas de este documento XML ha sido detallada en el apartado 2.3.3 de la *Memoria*.

PLN superior pueda trabajar sobre un formato estandarizado como es XML. Dada la orden:

```
echo "El niño comió una manzana" | callparser -in - -xmldep
```

El resultado es la representación en XML del grafo de dependencias correspondiente a la oración *"El niño comió una manzana"*[8]:

```
ok 1    El niño comió una manzana
------------  START XMLDEP ---------
<?xml version="1.0" encoding="ISO-8859-1"
standalone="yes"?>
<dependencies grammar="/perl/spmg/tree.pl">
  <cluster left="0" right="1" id="E1c_0_1" token="el"
   lex="El"/>
  <node cluster="E1c_0_1" tree="0 det" form="el"
   lemma="el" xcat="det" cat="det" id="E1n001"
   deriv="E1d000001"/>
  <cluster left="1" right="2" id="E1c_1_2"
   token="niño" lex="niño"/>
  <node cluster="E1c_1_2" tree="69 n:agreement
   nc:agreement cnoun_leaf" form="niño"
   lemma="niño" xcat="N2" cat="nc"
   id="E1n002" deriv="E1d000000"/>
  <edge source="E1n002" target="E1n001" label="det"
   type="subst" id="E1e001">
   <deriv names="E1d000000" source_op="E1o2"
    target_op="E1o9" span="0 1"/>
  </edge>
  <node cluster="E1c_1_2" tree="72 adj_as_cnoun
   n:agreement nc:agreement" form="niño"
   lemma="niño" xcat="N2" cat="adj"
   id="E1n008" deriv="E1d000007"/>
  <edge source="E1n008" target="E1n001" label="det"
   type="subst" id="E1e002">
  <deriv names="E1d000007" source_op="E1o4"
   target_op="E1o9" span="0 1"/>
  </edge>
  <cluster left="2" right="3" id="E1c_2_3"
   token="comió" lex="comió"/>
  <node cluster="E1c_2_3" tree="136 V1VMod:agreement
   clsubj:agreement ante:clitic_sequence
```

[8] Únicamente se muestra un fragmento del mismo.

```
 post:clitic_sequence clitics arg1:collect_real_arg
 arg2:collect_real_arg arg0:collect_real_subject
 arg1:real_group_comp arg2:real_group_comp
 ncpred:real_group_comp arg0:noun:true_subject
 arg0:post_PP:true_subject arg0:post_noun:
 true_subject arg0:post_s:true_subject arg0:post_v:
 true_subject arg0:s:true_subject arg0:v:
 true_subject Infl:verb_agreement V:verb_agreement
 v:verb_agreement V1:verb_agreement_ancestor arg1:
 verb_argument_other arg2:verb_argument_other arg0:
 verb_argument_subject verb_canonical
 verb_categorization_active_acomp1"
 form="comió" lemma="comer" xcat="S"
 cat="v" id="E1n007" deriv="E1d000002 E1d000006
 E1d000008 E1d000010"/>
<edge source="E1n007" target="E1n008"
 label="subject" type="subst" id="E1e003">
<deriv names="E1d000010 E1d000006"
 source_op="E1o1" target_op="E1o4" span="0 2"/>
</edge>
<edge source="E1n007" target="E1n002"
 label="subject" type="subst" id="E1e004">
 <deriv names="E1d000002 E1d000008"
  source_op="E1o1" target_op="E1o2"
  span="0 2"/>
</edge>
<cluster left="3" right="4" id="E1c_3_4"
 token="una" lex="una"/>
<node cluster="E1c_3_4" tree="0 det" form="una"
 lemma="una" xcat="det" cat="det" id="E1n005"
 deriv="E1d000005"/>
<cluster left="4" right="5" id="E1c_4_5"
 token="manzana" lex="manzana"/>
<node cluster="E1c_4_5" tree="69 n:agreement
 nc:agreement cnoun_leaf" form="manzana"
 lemma="manzana" xcat="N2" cat="nc" id="E1n006"
 deriv="E1d000004"/>
<edge source="E1n006" target="E1n005" label="det"
 type="subst" id="E1e005">
 <deriv names="E1d000004" source_op="E1o5"
  target_op="E1o14" span="3 4"/>
</edge>
<edge source="E1n007" target="E1n006"
 label="object" type="subst" id="E1e006">
```

```
 <deriv names="E1d000010 E1d000002 E1d000008
  E1d000006" source_op="E1o1" target_op="E1o5"
  span="3 5"/>
</edge>
<cluster left="5" right="5" id="E1c_5_5" token=""
 lex=""/>
<node cluster="E1c_5_5" tree="30 shallow_auxiliary
 spunct_others" form=""
 lemma="" xcat="S" cat="S" id="E1n004"
 deriv="E1d000003"/>
<edge source="E1n007" target="E1n004" label="S"
 type="adj" id="E1e007">
 <deriv names="E1d000002 E1d000006"
  source_op="E1o1" target_op="E1o8"
  span="5 6 6 6"/>
</edge>
<node cluster="E1c_5_5" tree="33 shallow_auxiliary
 spunct_dot" form="" lemma="" xcat="S" cat="S"
 id="E1n009" deriv="E1d000009"/>
<edge source="E1n007" target="E1n009" label="S"
 type="adj" id="E1e008">
 <deriv names="E1d000010 E1d000008"
  source_op="E1o1" target_op="E1o6"
  span="5 6 6 6"/>
</edge>
<cluster left="5" right="6" id="E1c_5_6" token="."
 lex="."/>
 <node cat="_" cluster="E1c_5_6" tree="lexical"
  form="." lemma="." id="E1n003"/>
<edge source="E1n004" target="E1n003" label="void"
 type="lexical" id="E1e009">
 <deriv names="E1d000003" source_op="E1o8"
  target_op="E1o17" span="5 6"/>
</edge>
<edge source="E1n009" target="E1n003" label="void"
 type="lexical" id="E1e010">
 <deriv names="E1d000009" source_op="E1o6"
  target_op="E1o17" span="5 6"/>
</edge>
...
```

11.2.14. txtdep

Del mismo modo que el grafo de dependencias puede ser detallado en el formato estandarizado XML, también puede ser representado en texto plano. Se trata de un formato más comprensible para el usuario humano. Por ejemplo, dada la orden:

```
echo "El niño comió una manzana" | callparser -in - -txtdep
```

El resultado es la representación en texto plano del grafo de dependencias correspondiente a la oración *"El niño comió una manzana"*:

```
------------  START TXTDEP ---------
META: Grammar=/perl/spmg/tree.pl
1:2/niño/niño/nc/69
       --{det:subst:left}--> 0:1/el/el/det/0
1:2/niño/niño/adj/72
       --{det:subst:left}--> 0:1/el/el/det/0
2:3/comió/comer/v/136
       --{subject:subst:left}--> 1:2/niño/niño/adj/72
2:3/comió/comer/v/136
       --{subject:subst:left}--> 1:2/niño/niño/nc/69
4:5/manzana/manzana/nc/69
       --{det:subst:left}--> 3:4/una/una/det/0
2:3/comió/comer/v/136
 --{object:subst:right}--> 4:5/manzana/manzana/nc/69
2:3/comió/comer/v/136--{S:adj:right}--> 5:5///S/30
2:3/comió/comer/v/136 --{S:adj:right}-->  5:5///S/33
5:5///S/30 --{void:lexical:left}--> 5:6/././_/lexical
5:5///S/33 --{void:lexical:left}--> 5:6/././_/lexical
tree 69 => 69 n:agreement nc:agreement cnoun_leaf
tree lexical => lexical
tree 33 => 33 shallow_auxiliary spunct_dot
tree 136 => 136 V1VMod:agreement clsubj:agreement
ante:clitic_sequence post:clitic_sequence clitics
arg1:collect_real_arg arg2:collect_real_arg
arg0:collect_real_subject arg1:real_group_comp
arg2:real_group_comp ncpred:real_group_comp
arg0:noun:true_subject arg0:post_PP:true_subject
arg0:post_noun:true_subject arg0:post_s:true_subject
arg0:post_v:true_subject arg0:s:true_subject
arg0:v:true_subject Infl:verb_agreement V:
verb_agreement v:verb_agreement V1:
verb_agreement_ancestor arg1:verb_argument_other
```

```
arg2:verb_argument_other arg0:verb_argument_subject
verb_canonical verb_categorization_active_acomp1
tree 72 => 72 adj_as_cnoun n:agreement nc:agreement
tree 0 => 0 det
tree 30 => 30 shallow_auxiliary spunct_others
------------  END TXTDEP ---------
```

En él se indica para cada unidad léxica que forma la frase de entrada, cuál es el lema de la misma, con qué árbol GAR se ha enlazado y qué otras palabras de la oración se encuentran dentro de ese mismo árbol.

Por ejemplo la línea:

```
1:2/niño/niño/nc/69 -{det:subst:left}->0:1/el/el/det/0
```

En ella aparece que la forma *"niño"* tiene como lema a *"niño"*. Además, se trata de un nombre común, y el árbol de la gramática GAR con el cual enlaza a través de su *hypertag* es el `#69`. A continuación, expone que el árbol GAR número `#69` ha sido objeto de una operación de sustitución por el nodo `det`, (no es un árbol, de ahí el `0`). Este nodo recoge la forma lexical *"el"*. Ello marca la dependencia entre el sustantivo y el determinante que se puede observar en el grafo de dependencias de la figura 11.1.

Lo que nos está ofreciendo esta opción, además de las dependencias entre unidades léxicas, es el detalle de un árbol de derivaciones. Explica qué árboles han intervenido y cuál y dónde se han producido las operaciones GAR de adjunción (`adj`) o sustitución (`subst`).

11.2.15. robust

Con esta opción se solicita al analizador seleccionado del servidor *Parserd* que trabaje en modo robusto. Por defecto, se encuentra seleccionado el analizador `spmgtelr` registrado en el servidor. Se trata del mismo analizador que `spmgtel` pero registrado con la opción de robustez activada.

El modo de robustez activada, obliga al analizador a, en caso de no ser capaz de analizar la oración en su totalidad, devolver por separado los trozos de la frase que es capaz de cubrir. Por ejemplo, para la oración sintácticamente incorrecta *"El el niño comió una manzana"* en la siguiente orden:

```
echo "El el niño comió una manzana" | callparser -in -
-robust -d dep
```

Se obtendrían los grafos de dependencias de la figura 11.3.

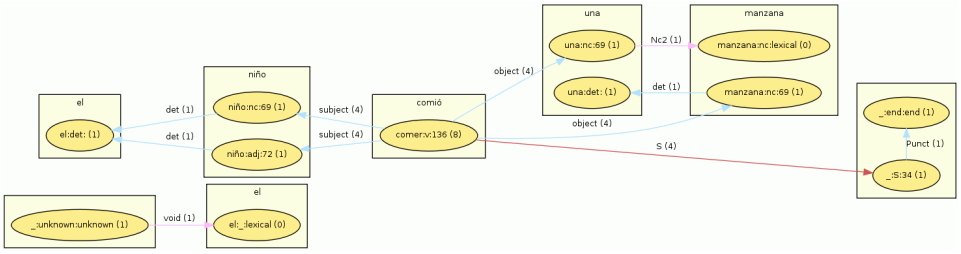


Figura 11.3: Grafos de dependencias de la oración *"El el niño comió una manzana"*.

En la figura 11.3, aparecen dos grafos de dependencias. Mientras que el primero cubre prácticamente toda la oración sintácticamente correcta, el segundo recoge el fenómeno sintáctico desconocido para el analizador.

11.2.16. rparser

Como se ha comentado en el anterior subapartado, el analizador `spmgtelr` es el analizador robusto por defecto. Si se desea indicar otro analizador con robustez activada y registrado en el servidor, ha de emplearse la opción `-rparser=<name>`. Por ejemplo:

```
echo "El el niño comió una manzana" | callparser -in - -robust
-rparser=MiParserRobusto
```

11.3. WebParser

Además de *Callparser*, el servidor *Parserd* dispone de un cliente *web* implementado por el *script Perl* `webparser.pl`. Este cliente se encuentra bajo un servidor *web*[9], desde el cual accede al servidor *Parserd* mediante el protocolo HTTP.

La interfaz inicial de *WebParser* se muestra en la figura 11.4.

Para acceder a la interfaz *web*, es necesario indicar la dirección IP de la máquina donde se encuentra el cliente `webparser.pl`:

```
http://<dirección ip>/cgi-bin/webparser.pl
```

Esta interfaz *web* presenta las características que se detallan a continuación.

[9] En este caso, *Apache2* para *Ubuntu*.

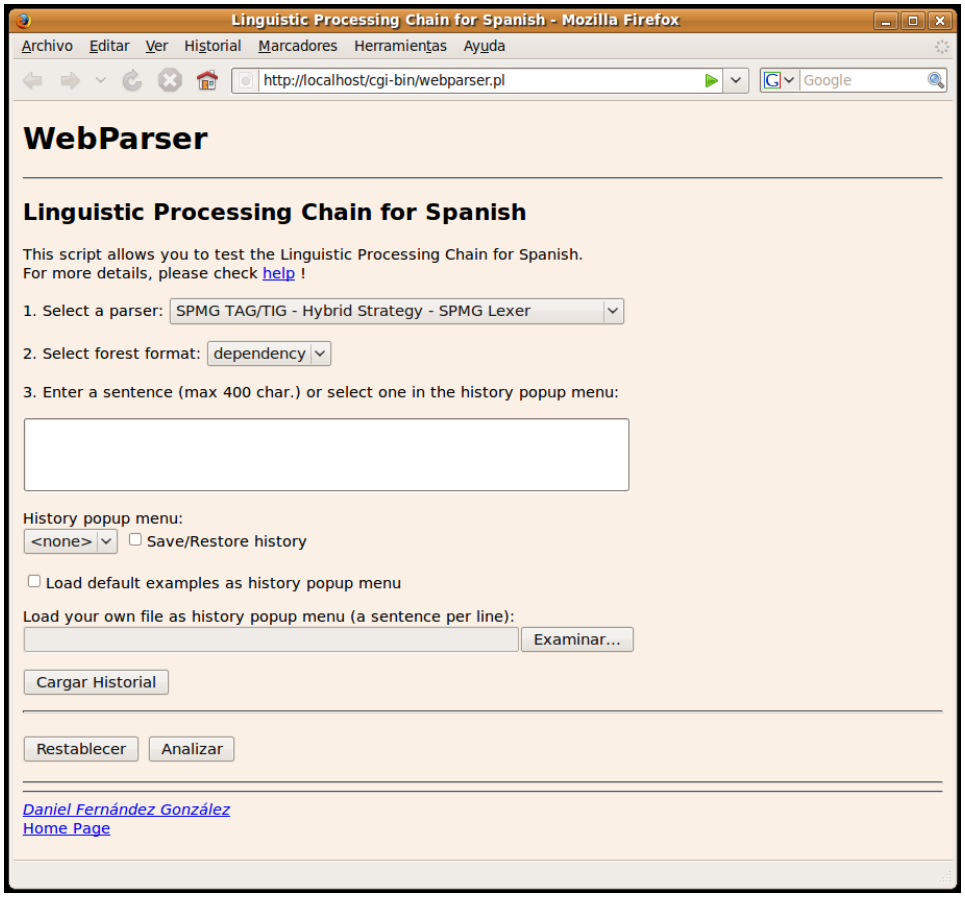


Figura 11.4: Interfaz *web* del cliente *WebParser*.

11.3.1. Guía de uso

En el enlace `help` se accede a una página *web* que ofrece una guía acerca del uso correcto de la interfaz *web*.

11.3.2. Seleccionar analizador

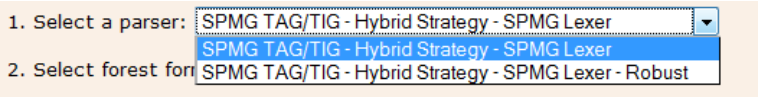


Figura 11.5: Analizadores disponibles en el cliente *WebParser*.

Para llevar a cabo el procesamiento lingüístico de una frase o fichero dados, es necesario seleccionar un analizador concreto o de lo contrario empleará el asignado por defecto. Los analizadores registrados en el servidor *Parserd* se presentan en un menú despegable (figura 11.5). Para este proyecto se han registrado en el servidor dos analizadores:

- **SPMG GAR/GIR - Hybrid Strategy - SPMG Lexer:** Es el analizador cuyo identificador es `spmgtel`. Está compuesto por el analizador sintáctico *SPMG Parser*, que sigue una estrategia híbrida GAR/GIR, y el etiquetador morfológico *SPMG Lexer*.
- **SPMG GAR/GIR - Hybrid Strategy - SPMG Lexer - Robust:** Se trata del analizador `spmgtelr`. Este tiene los mismos componentes que el analizador anterior pero con el modo de robustez activado.

11.3.3. Seleccionar formato de salida

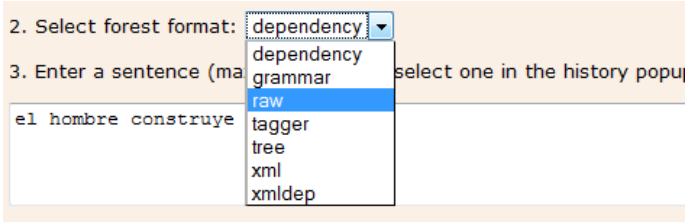


Figura 11.6: Formatos disponibles en el cliente *WebParser*.

Cuando se realiza un análisis se obtiene un resultado. Este puede representarse en distintos formatos dependiendo de la información deseada

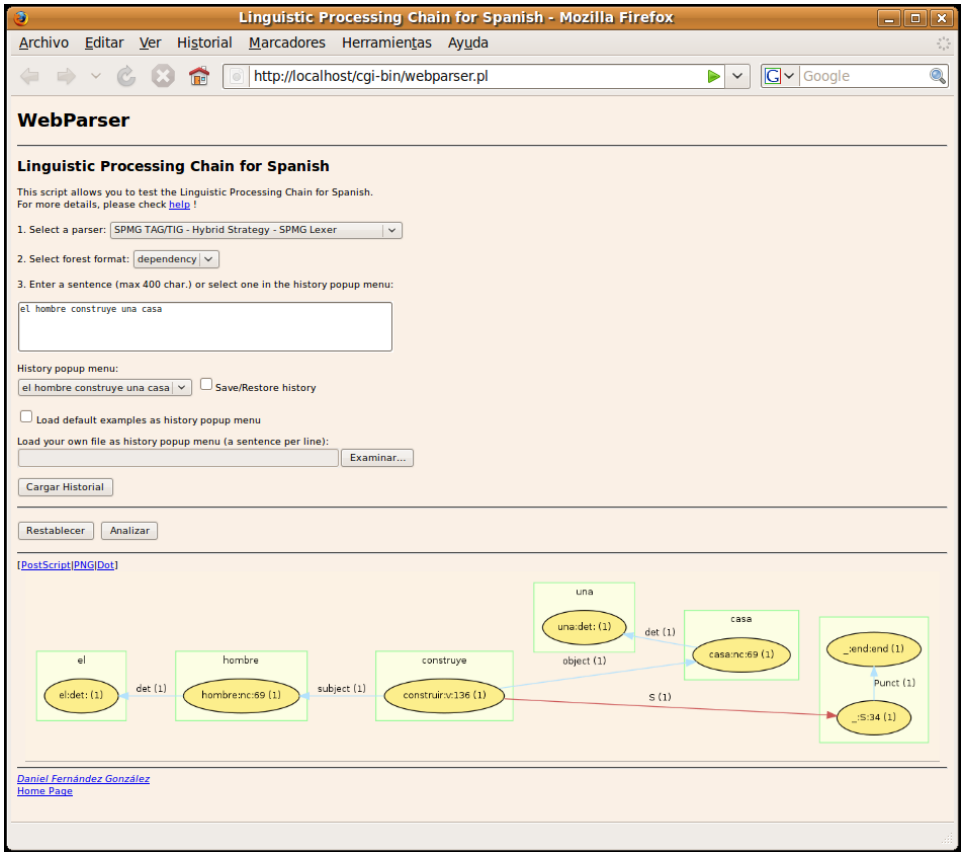


Figura 11.7: Opción *dependency* del cliente *WebParser*.

y del destinatario del resultado (usuario humano o sistema externo). Los formatos disponibles aparecen en el menú despegable de la figura 11.6.

Dada la oración *"El hombre construye una casa"* a procesar lingüísticamente, en los siguientes subapartados se van a mostrar los distintos resultados que se pueden obtener dependiendo del formato de salida seleccionado.

11.3.3.1. Dependency

Bajo esta opción, el sistema mostrará el grafo de dependencias correspondiente a la oración *"El hombre construye una casa"*, como aparece en la figura 11.7.

11.3.3.2. Grammar

Con este formato de salida se obtiene una representación del conjunto de reglas gramaticales empleadas para describir la estructura de la oración de entrada[10]. Además, también incluye una representación de las restricciones impuestas sobre cada símbolo terminal y no-terminal. Para la oración *"El hombre construye una casa"* se obtiene el resultado presente en la figura 11.8.

11.3.3.3. Raw

Con esta opción se muestra el bosque compartido de derivaciones resultado del análisis en su formato interno. Para la oración *"El hombre construye una casa"* se obtiene el resultado presente en la figura 11.9.

11.3.3.4. Tagger

Si seleccionamos esta opción como formato de salida, el sistema se convertirá en un etiquetador. De tal forma, que únicamente ofrecerá las unidades léxicas presentes en la oración de entrada etiquetadas mediante información morfosintáctica. Para la oración *"El hombre construye una casa."* se obtiene el resultado presente en la figura 11.10.

11.3.3.5. Tree

Con esta opción seleccionada, el sistema mostrará gráficamente como se combinan los árboles empleados para describir la estructura objeto de análisis[11]. Para la oración *"El hombre construye una casa"* se muestra el resultado en la figura 11.11. Los arcos relacionan dos árboles involucrados en una operación GAR. El arco dirigido parte del árbol destino de la operación, hacia el árbol que es adherido en el anterior. El color azul de la arista indica que la operación realizada es de sustitución, mientras que el color rojo señala que se trata de una operación de adjunción.

11.3.3.6. XML

Con este formato de salida seleccionado, el sistema muestra el bosque compartido de derivaciones en formato XML. Para la oración *"El hombre construye una casa"* se obtiene el resultado presente en la figura 11.12.

[10] Se trata de un formato similar al utilizado internamente por el analizador sintáctico recogido en el subapartado 11.3.3.3 del *Manual de Usuario*.

[11] Muestra gráficamente la misma información que un árbol de derivaciones.

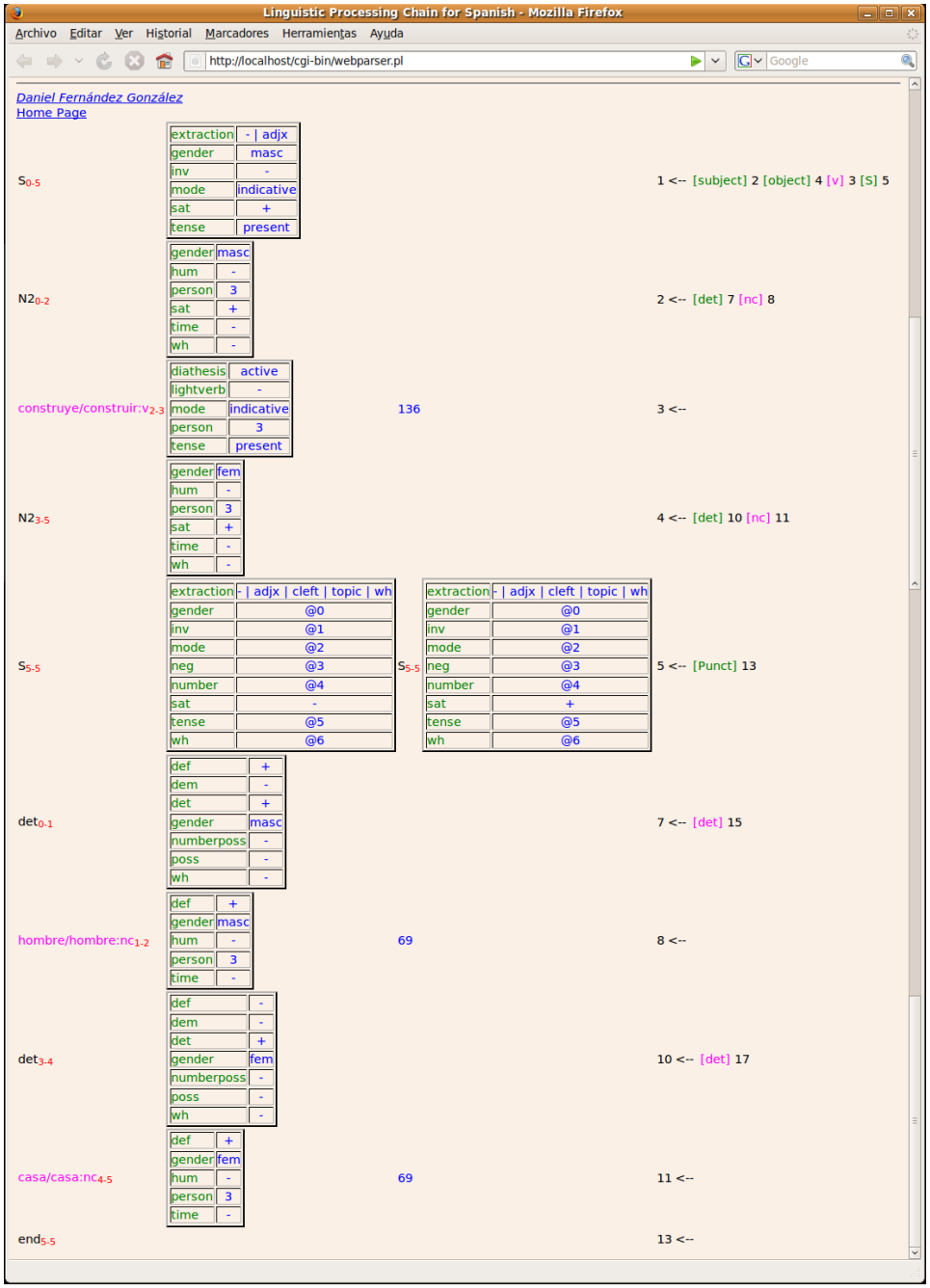


Figura 11.8: Opción *grammar* del cliente *WebParser*.

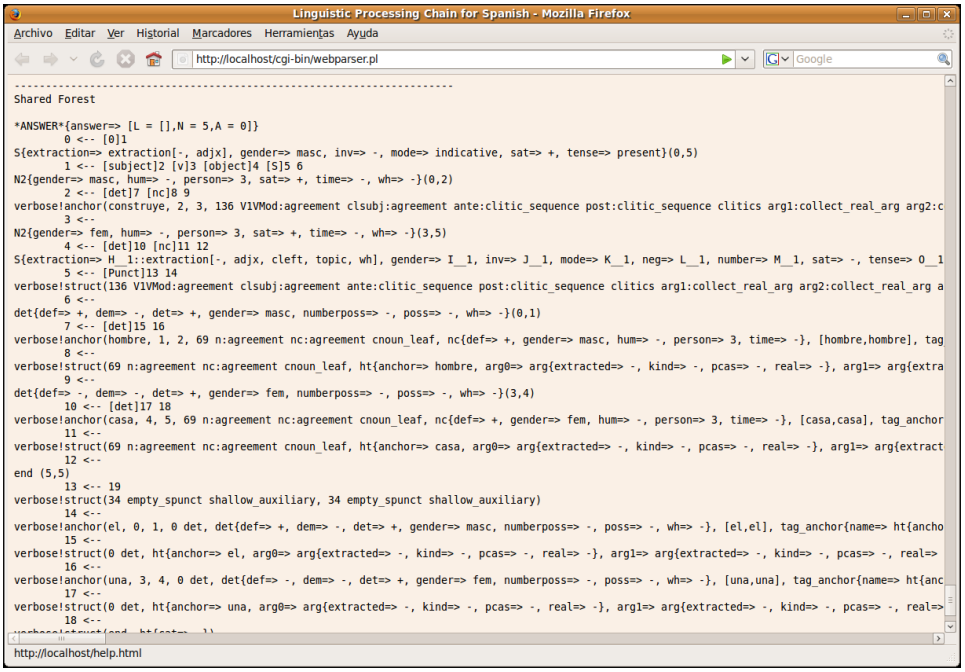


Figura 11.9: Opción *raw* del cliente *WebParser*.

11.3.3.7. XMLDep

Muestra el grafo de dependencias correspondiente a la oración de entrada representado en formato XMLDep. Para la oración *"El hombre construye una casa"* se obtiene el resultado presente en la figura 11.13.

11.3.4. Gestión de imágenes

Cuando el resultado del análisis es una imagen se puede gestionar mediante los enlaces presentes en la figura 11.14.

[PostScript|PNG|Dot]

Figura 11.14: Gestión de imágenes del cliente *WebParser*.

- **PostScript:** Este enlace permite descargar la imagen obtenida en formato *postscript*.
- **PNG:** Tras pinchar en este enlace, se obtiene la imagen en formato PNG.
- **Dot:** Esta opción ofrece la imagen en formato *dot*.

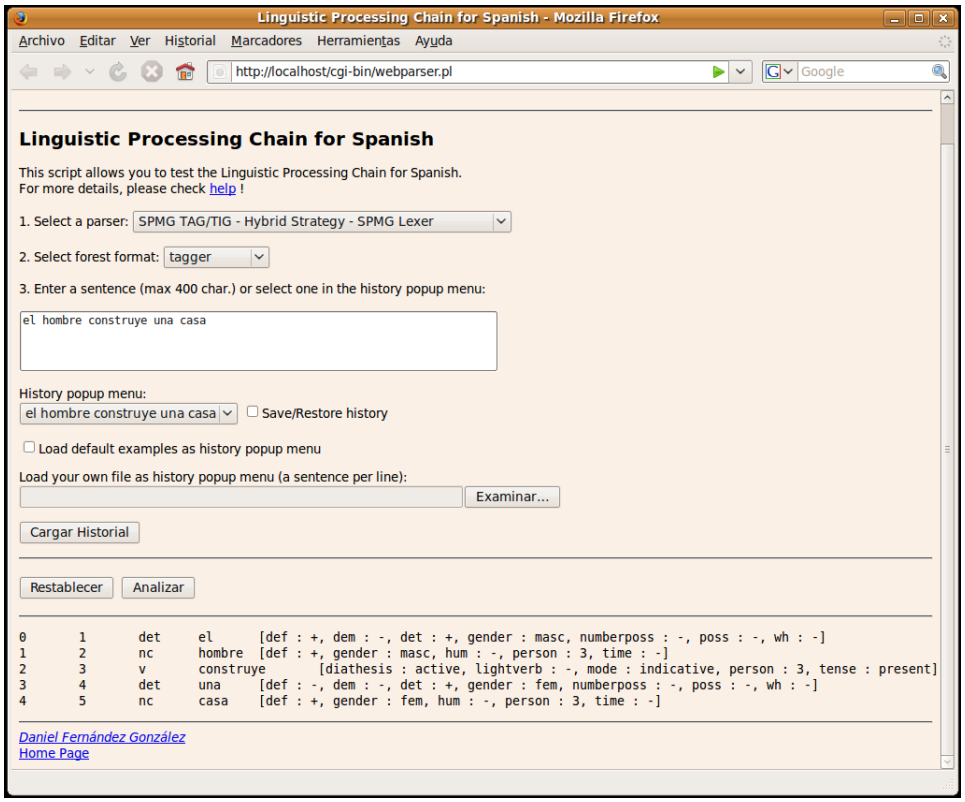


Figura 11.10: Opción *tagger* del cliente *WebParser*.

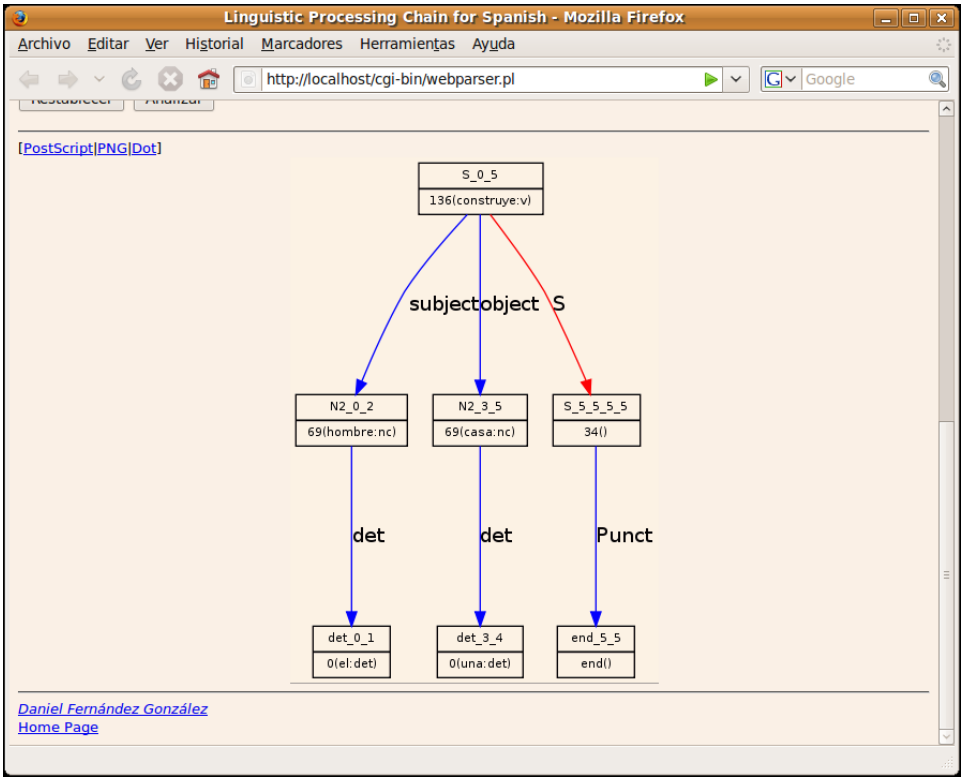


Figura 11.11: Opción *tree* del cliente *WebParser*.

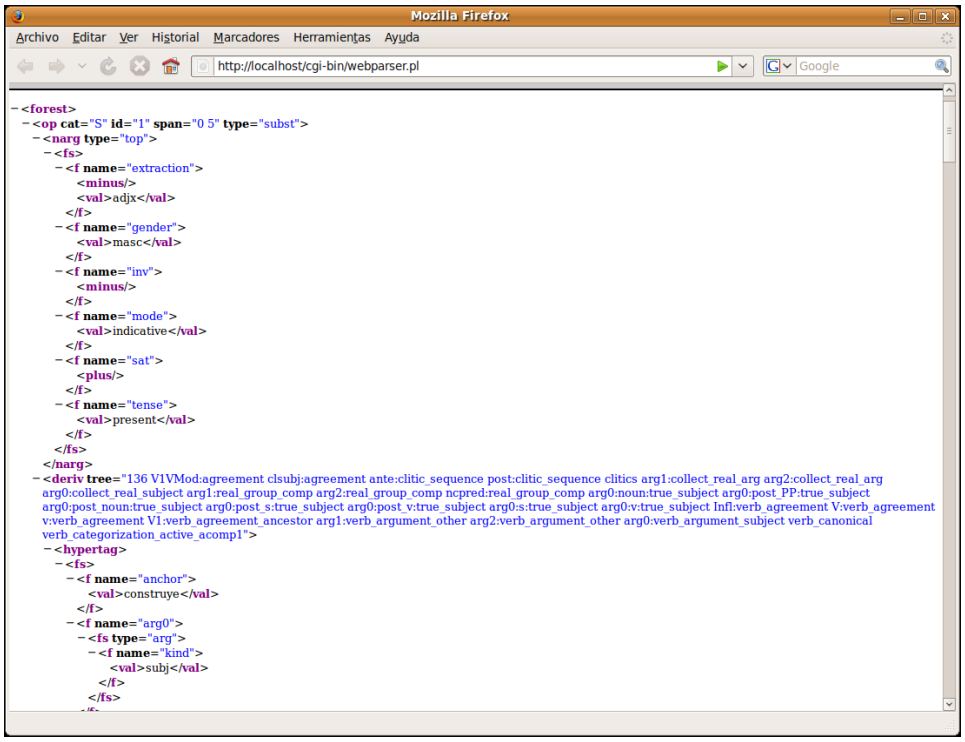
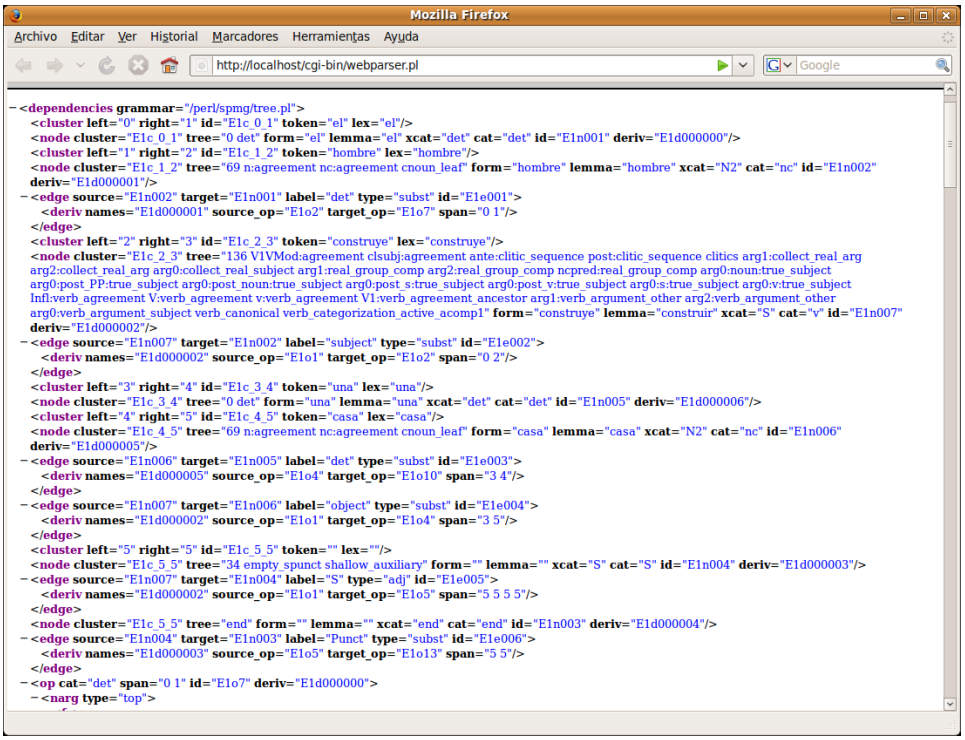


Figura 11.12: Opción *XML* del cliente *WebParser*.




```
- <dependencies grammar="/perl/spmg/tree.pl">
    <cluster left="0" right="1" id="E1c_0_1" token="el" lex="el"/>
    <node cluster="E1c_0_1" tree="0 det" form="el" lemma="el" xcat="det" cat="det" id="E1n001" deriv="E1d000000"/>
    <cluster left="1" right="2" id="E1c_1_2" token="hombre" lex="hombre"/>
    <node cluster="E1c_1_2" tree="69 n:agreement nc:agreement cnoun_leaf" form="hombre" lemma="hombre" xcat="N2" cat="nc" id="E1n002"
    deriv="E1d000001"/>
  - <edge source="E1n002" target="E1n001" label="det" type="subst" id="E1e001">
      <deriv names="E1d000001" source_op="E1o2" target_op="E1o7" span="0 1"/>
    </edge>
    <cluster left="2" right="3" id="E1c_2_3" token="construye" lex="construye"/>
    <node cluster="E1c_2_3" tree="136 VIVMod:agreement clsubj:agreement ante:clitic_sequence post:clitic_sequence clitics arg1:collect_real_arg
    arg2:collect_real_arg arg0:collect_real_subject arg1:real_group_comp arg2:real_group_comp ncpred:real_group_comp arg0:noun:true_subject
    arg0:post_PP:true_subject arg0:post_noun:true_subject arg0:post_s:true_subject arg0:post_v:true_subject arg0:s:true_subject arg0:v:true_subject
    Infl:verb_agreement V:verb_agreement v:verb_agreement V1:verb_agreement_ancestor arg1:verb_argument_other arg2:verb_argument_other
    arg0:verb_argument_subject verb_canonical verb_categorization_active_acomp1" form="construye" lemma="construir" xcat="S" cat="v" id="E1n007"
    deriv="E1d000002"/>
  - <edge source="E1n007" target="E1n002" label="subject" type="subst" id="E1e002">
      <deriv names="E1d000002" source_op="E1o1" target_op="E1o2" span="0 2"/>
    </edge>
    <cluster left="3" right="4" id="E1c_3_4" token="una" lex="una"/>
    <node cluster="E1c_3_4" tree="0 det" form="una" lemma="una" xcat="det" cat="det" id="E1n005" deriv="E1d000006"/>
    <cluster left="4" right="5" id="E1c_4_5" token="casa" lex="casa"/>
    <node cluster="E1c_4_5" tree="69 n:agreement nc:agreement cnoun_leaf" form="casa" lemma="casa" xcat="N2" cat="nc" id="E1n006"
    deriv="E1d000005"/>
  - <edge source="E1n006" target="E1n005" label="det" type="subst" id="E1e003">
      <deriv names="E1d000005" source_op="E1o4" target_op="E1o10" span="3 4"/>
    </edge>
  - <edge source="E1n007" target="E1n006" label="object" type="subst" id="E1e004">
      <deriv names="E1d000002" source_op="E1o1" target_op="E1o4" span="3 5"/>
    </edge>
    <cluster left="5" right="5" id="E1c_5_5" token="" lex=""/>
    <node cluster="E1c_5_5" tree="34 empty_spunct shallow_auxiliary" form="" lemma="" xcat="S" cat="S" id="E1n004" deriv="E1d000003"/>
  - <edge source="E1n007" target="E1n004" label="S" type="adj" id="E1e005">
      <deriv names="E1d000002" source_op="E1o1" target_op="E1o5" span="5 5 5 5"/>
    </edge>
    <node cluster="E1c_5_5" tree="end" form="" lemma="" xcat="end" cat="end" id="E1n003" deriv="E1d000004"/>
  - <edge source="E1n004" target="E1n003" label="Punct" type="subst" id="E1e006">
      <deriv names="E1d000003" source_op="E1o5" target_op="E1o13" span="5 5"/>
    </edge>
  - <op cat="det" span="0 1" id="E1o7" deriv="E1d000000">
    - <narg type="top">
```


Figura 11.13: Opción *XMLDep* del cliente *WebParser*.

11.3.5. Introducción de texto a analizar

A diferencia del cliente *Callparser*, la interfaz *web* está limitada al análisis de una única oración. Sin embargo, ello no impide cargar varias oraciones de un fichero de texto, para luego ser analizadas de forma independiente cada una de ellas.

La interfaz *web* dispone de un cuadro de texto (figura 11.15). En él se permite introducir una única oración de un máximo de 400 caracteres.

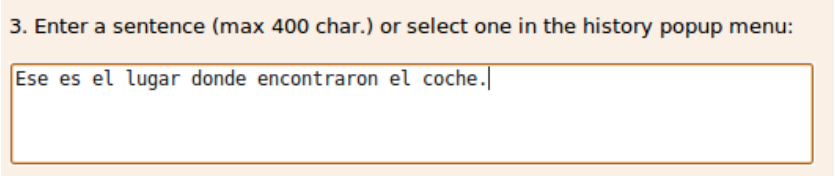


Figura 11.15: Cuadro de texto del cliente *WebParser*.

Además también es posible cargar en el cuadro de texto de la interfaz *web* una oración presente en el menú desplegable denominado *historial* (figura 11.16).

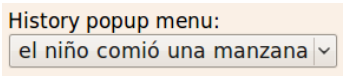


Figura 11.16: Historial del cliente *WebParser*.

Existen diferentes formas de cargar el historial:

- **Automáticamente:** El historial almacena de forma automática las oraciones que el usuario introduce en el cuadro de texto correspondiente. Además, si cuando se produce el almacenado automático la casilla `Save/Restore History` (figura 11.17) se encuentra activada, se estará guardando el historial de esa sesión del navegador. De tal forma que, cuando se inicie una nueva sesión, únicamente se debe marcar la casilla `Save/Restore History` y pulsar el botón `Cargar Historial` para que el historial almacenado se cargue en el historial de la sesión actual.

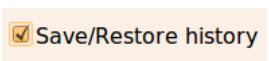


Figura 11.17: Opción `Save/Restore History` del cliente *WebParser*.

- **Desde el fichero de ejemplos:** Cada analizador registrado en el servidor *Parserd* tiene asignado un fichero de oraciones ejemplo. Si marcamos la casilla de la figura 11.18 y, a continuación, pulsamos el botón `Cargar Historial`, las oraciones del fichero ejemplo se cargarán en el historial de *WebParser*.

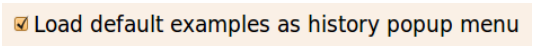


Figura 11.18: Opción `Load default examples as history popup menu` del cliente *WebParser*.

- **Desde un fichero externo:** La interfaz de *WebParser* permite subir un fichero de texto para ser objeto de análisis. Las oraciones presentes en el fichero seleccionado serán incluidas en el historial de *WebParser*. Para seleccionar el fichero se debe hacer uso del *explorador de archivos* presente en la interfaz *web* (figura 11.19) y, seguidamente, se debe pulsar el botón `Cargar Historial`.

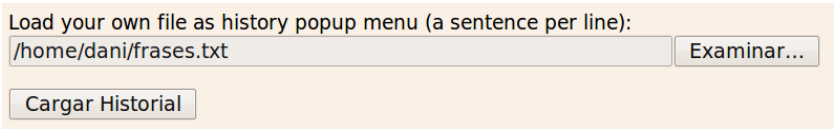


Figura 11.19: Opción `Load your own file as history popup menu` del cliente *WebParser*.

Parte IV

Apéndices

Apéndice A

Servidor HTTP Apache2

Apache[1] es el servidor *web* más usado en sistemas *Linux*. Los servidores *web* se usan para servir páginas *web* solicitadas por equipos cliente. Los clientes normalmente solicitan y muestran páginas *web* mediante el uso de navegadores como *Firefox*, *Opera* o *Mozilla*.

Los usuarios introducen un Localizador de Recursos Uniforme[2] para señalar a un servidor *web* por medio de su Nombre de Dominio Totalmente Cualificado[3] y de una ruta al recurso solicitado. Por ejemplo, para ver la página *web* del sitio de *Ubuntu*, un usuario debería introducir únicamente el FQDN. Para solicitar información específica acerca del soporte de pago, un usuario deberá introducir el FQDN seguido de una ruta.

El protocolo más comúnmente utilizado para ver páginas *web* es el *Hyper Text Transfer Protocol* (HTTP). Protocolos como el *Hyper Text Transfer Protocol* sobre *Secure Sockets Layer* (HTTPS) y *File Transfer Protocol* (FTP), un protocolo para subir y descargar archivos, también están soportados.

Los servidores *web Apache* a menudo se usan en combinación con el motor de bases de datos MySQL, el lenguaje de *scripting* PHP, y otros lenguajes populares como *Python* y *Perl*. Esta configuración se denomina LAMP[4] y conforma una potente y robusta plataforma para el desarrollo y distribución de aplicaciones basadas en la *web*.

A.1. Instalación

El servidor *web Apache2* está disponible en *Ubuntu*. Para instalar *Apache2*, introduzca el siguiente comando en un terminal:

[1] `http://doc.ubuntu-es.org/HTTPD_Servidor_web_Apache2`
[2] *Uniform Resource Locator*, URL.
[3] *Fully Qualified Domain Name*, FQDN.
[4] *Linux*, *Apache*, MySQL y *Perl*/*Python*/PHP.

```
sudo apt-get install apache2
```

A.2. Configuración

El archivo de configuración predeterminado de *Apache2* es `apache2.conf` ubicado en el directorio `/etc/apache2`. Puede editar este archivo para configurar el servidor *Apache2*. Podrá configurar el número de puerto, la raíz de documentos, los módulos, los archivos de registros, los *hosts* virtuales, entre otros.

A.2.1. Opciones básicas

Esta sección explica los parámetros de configuración esenciales para el servidor *Apache2*. Remítase a la documentación de *Apache2* para más detalles.

- *Apache2* trae una configuración predeterminada preparada para servidores virtuales. Viene configurado con un único servidor virtual predeterminado (usando la directiva `VirtualHost`). Éste se puede dejar tal cual si sólo se dispone de un único sitio *web*, o usarlo como plantilla para servidores virtuales adicionales si se tiene varios sitios *web*. Para modificar el servidor virtual predeterminado, edite el archivo `/etc/apache2/sites-available/default`. Si desea configurar un nuevo servidor o sitio virtual, copie ese archivo dentro del mismo directorio con el nombre que haya elegido. Por ejemplo:

```
sudo cp /etc/apache2/sites-available/default
/etc/apache2/sites-available/minuevositio
```

- La directiva `ServerAdmin` especifica la dirección de correo del administrador del servidor. El valor por omisión es `webmaster@localhost`. Cambie esta dirección por alguna a la que le puedan llegar los mensajes que le envíen (si es el administrador del servicio). Si su sitio *web* tiene algún problema, *Apache2* mostrará un mensaje de error en el que aparecerá esta dirección de correo para que la gente pueda enviar un informe del mismo. La directiva se encuentra en el fichero de configuración de su sitio en `/etc/apache2/sites-available`.

- La directiva `Listen` especifica el puerto (y, opcionalmente, la dirección IP) por el que escuchará *Apache2*. Si no se especifica la dirección IP, *Apache2* escuchará por todas las direcciones IP asignadas a la máquina en la que se ejecute. El valor predeterminado de la directiva `Listen` es `80`. Cambiarlo a `127.0.0.1:80` provoca que *Apache2* sólo escuche por su dispositivo *loopback*, de forma que no estará disponible para

Internet. La directiva se puede encontrar y cambiar en su propio archivo de configuración, `/etc/apache2/ports.conf`.

- La directiva `ServerName` es opcional, y especifica con cuál FQDN responderá su sitio *web*. El servidor virtual predeterminado no especifica ninguna directiva `ServerName`, por lo que responderá a todas las peticiones que no se ajusten a ninguna directiva `ServerName` en otro servidor virtual. Si acaba de adquirir el dominio `ubuntumola.com`, y desea asociar a él su servidor *Ubuntu*, el valor de la directiva `ServerName` en el archivo de configuración de su servidor virtual debería ser `ubuntumola.com`. Añada esta directiva al nuevo archivo de configuración virtual que creó previamente (`/etc/apache2/sites-available/minuevositio`).

  También puede desear que su sitio responda a `www.ubuntumola.com`, ya que muchos usuarios asumen que el prefijo `www` es apropiado. Para ello, use la directiva `ServerAlias`. Puede usar comodines en la directiva `ServerAlias`. Por ejemplo, `ServerAlias *.ubuntumola.com` hará que su sitio responda a cualquier solicitud de dominio que termine en `.ubuntumola.com`.

- La directiva `DocumentRoot` especifica dónde debe buscar *Apache* los archivos que forman el sitio. El valor predeterminado es `/var/www`. No hay ningún sitio configurado allí, pero si descomenta la directiva `RedirectMatch` en `/etc/apache2/apache2.conf`, las peticiones se redirigirán a `/var/www/apache2-default`, que es donde reside el sitio predeterminado de *Apache2*. Cambie este valor en el archivo de *host* virtual de su sitio y recuerde crear ese directorio si fuese necesario.

- La directiva `LockFile` establece la ruta al archivo de bloqueo usado cuando el servidor se compila con la opción:

  `USE_FCNTL_SERIALIZED_ACCEPT` o `USE_FLOCK_SERIALIZED_ACCEPT`

  Debe almacenarse en el disco local y dejarse su valor predeterminado, a menos que el directorio de registros esté ubicado en un directorio NFS compartido. Si es éste el caso, debería cambiarse el valor predeterminado a una ubicación en el disco local y a un directorio en el que sólo tenga permisos de lectura el usuario `root`.

- La directiva `PidFile` establece el archivo en el que el servidor registrará su identificador de proceso (PID). Sobre este archivo sólo debe tener permisos de lectura el usuario `root`. En la mayoría de los casos, debería dejarse su valor predeterminado.

- La directiva `User` establece el identificador de usuario utilizado por el servidor para responder a las peticiones. Esta opción determina el

acceso del servidor. Todos los archivos inaccesibles para este usuario serán también inaccesibles para los visitantes de su sitio *web*. El valor predeterminado para `User` es `www-data`.

Hasta que no sepa exactamente lo que está haciendo, no ponga en la directiva `User` al `root`. Usar el `root` como usuario puede crear grandes agujeros de seguridad en su servidor *web*.

La directiva `Group` es similar a la directiva `User`. `Group` establece el grupo sobre el que el servidor aceptará las peticiones. El grupo por defecto es también `www-data`.

A.2.2. Opciones predeterminadas

Esta sección explica la configuración de las opciones predeterminadas del servidor *Apache2*. Por ejemplo, si desea añadir un *host* virtual, las opciones que configure para el *host* virtual tienen prioridad para ese *host* virtual. Para las directivas no definidas dentro de las opciones del *host* virtual, se usan los valores predeterminados.

- El `DirectoryIndex` es la página servida por defecto por el servidor cuando un usuario solicita el índice de un directorio añadiendo la barra de división (`/`) al final del nombre del directorio.

  Por ejemplo, cuando un usuario solicite la página:

  ```
  http://www.ejemplo.com/este_directorio/
  ```

  El servidor intentará buscar uno de los archivos listados en la directiva `DirectoryIndex` y devolverá el primero que encuentre. Si no encuentra ninguno de esos archivos, y está establecida la opción `Options Indexes` para ese directorio, el servidor generará y devolverá una lista, en formato HTML, de los subdirectorios y archivos del directorio. El valor predeterminado, almacenado en `/etc/apache2/apache2.conf`, es `«index.html index.cgi index.pl index.php index.xhtml»`. Por tanto, si *Apache2* encuentra un archivo en un directorio solicitado que se ajusta a alguno de esos nombres, se mostrará el primero de todos.

- La directiva `ErrorDocument` te permite especificar un archivo que usará *Apache2* para los eventos de error específicos. Por ejemplo, si un usuario solicita un recurso que no existe, se producirá un error 404, y (en base a la configuración predeterminada de *Apache2*), se mostrará el archivo `/usr/share/apache2/error/HTTP_NOT_FOUND.html.var`. Ese archivo no está en el `DocumentRoot` del servidor, sino que existe una directiva `Alias` en `/etc/apache2/apache2.conf` que redirige hacia `/usr/share/apache2/error/` las solicitudes dirigidas al

directorio `/error`. Para ver una lista de las directivas `ErrorDocument` predeterminadas, use la orden:

```
grep ErrorDocument /etc/apache2/apache2.conf
```

- De forma predeterminada, el servidor escribe los registros de las transferencias en el archivo `/var/log/apache2/access.log`. Ello se puede modificar usando la directiva `CustomLog`, u omitirla para aceptar la opción predeterminada, especificada en `/etc/apache2/apache2.conf`. También puede especificar el archivo en el que se registrarán los errores, por medio de la directiva `ErrorLog`, cuyo valor predeterminado es `/var/log/apache2/error.log`. Estos se mantienen separados de los registros de transferencias para ayudar en la resolución de problemas con su servidor *Apache2*. También puede especificar la directiva `LogFormat`[5].

- Algunas opciones son especificadas por directorio en lugar de por servidor. Una de estas directivas es `Option`. Un párrafo `Directory` está encerrado entre etiquetas XML, como estas:

```
<Directory /var/www/mynewsite>
...
<Directory>
```

  La directiva `Options` dentro del párrafo `Directory` acepta uno o más de los siguientes valores (entre otros), separados por espacios:

  - `ExecCGI` - Permite la ejecución de *scripts* CGI. Éstos no serán ejecutados si esta opción no fue seleccionada. Muchos archivos no deberían ser ejecutados como *scripts* CGI, ya que podría resultar muy peligroso. Deberían mantenerse en un directorio separado fuera de su `DocumentRoot`, y dicho directorio debería ser el único que tuviese activada la opción `ExecCGI`. Así está establecido desde el principio, y la ubicación predeterminada para los *scripts* CGI es `/usr/lib/cgi-bin`.
  - `Includes` - Permite `server-side includes`. Éstos hacen posible que un fichero HTML pueda incluir otros ficheros. No es una opción muy común, consulte el *Apache2* SSI (Server Side Includes)[6] para más información.
  - `IncludesNOEXEC` - Permite `server-side includes`, pero deshabilita los `#exec` y `#include` en los *scripts* CGI.

[5] Consulte en `/etc/apache2/apache2.conf` su valor predeterminado.
[6] `http://httpd.apache.org/docs/2.0/howto/ssi.html`

- `Indexes` - Muestra una lista formateada del contenido de los directorios, si no existe el `DirectoryIndex` (como el `index.html`) en el directorio solicitado.

  Por razones de seguridad, esto no debería establecerse, particularmente en el directorio indicado por `DocumentRoot`. Habilite esta opción con cuidado (y sólo para ciertos directorios) sólo si esta seguro de querer que los usuarios vean todo el contenido del directorio.
- `Vistas múltiples` - Soporte para vistas múltiples negociadas por contenido; esta opción está desactivada de forma predeterminada por motivos de seguridad[7].
- `SymLinksIfOwnerMatch` - Solo seguirá enlaces simbólicos si el directorio de destino es del mismo usuario que el del enlace.

A.2.3. Configuración de Servidores Virtuales

Los servidores virtuales te permiten ejecutar, en la misma máquina, diferentes servidores para diferentes direcciones IP, diferentes nombres de máquina o diferentes puertos. Por ejemplo, puedes tener los sitios *web* `http://www.ejemplo.com` y `http://www.otroejemplo.com` en el mismo servidor *web* usando servidores virtuales. Esta opción se corresponde con la directiva `<VirtualHost>`, usada para el servidor virtual predeterminado y para los servidores virtuales basados en IP. Para un servidor virtual basado en el nombre se usa la directiva `<NameVirtualHost>` .

A.3. Módulos de Apache

Apache es un servidor modular. Esto supone que en el núcleo del servidor sólo está incluida la funcionalidad más básica. Las características extendidas están disponibles a través de módulos que se pueden cargar en *Apache*. De forma predeterminada, durante la compilación se incluye un juego básico de módulos en el servidor. Si el servidor se compila para que use módulos cargables dinámicamente, los módulos se podrán compilar por separado y se podrán añadir posteriormente usando la directiva `LoadModule`. En caso contrario, habrá que recompilar *Apache* para añadir o quitar módulos. *Ubuntu* compila *Apache2* para que permita la carga dinámica de módulos. Las directivas de configuración se pueden incluir condicionalmente en base a la presencia de un módulo en particular, encerrándolas en un bloque `<IfModule>`. Puede instalar módulos adicionales de *Apache2* (mediante la orden `apt-get`) y usarlos con su servidor *web*. Por ejemplo, para instalar el módulo de *Apache2* que proporciona autenticación por MySQL, puede ejecutar lo siguiente en la línea de órdenes de un terminal:

[7] Consulte la documentación de *Apache2* sobre esta opción.

```
sudo apt-get install libapache2-mod-auth-mysql
```

Una vez instalado el módulo, este estará disponible en el directorio `/etc/apache2/mods-available`. Puede utilizar el comando `a2enmod` para activar el módulo, o el comando `a2dismod` para desactivarlo. Una vez que active el módulo, éste estará disponible en el directorio `/etc/apache2/mods-enabled`.

A.4. Arrancando el servidor

Para iniciar el servidor debe ejecutar la siguiente orden:

```
sudo /etc/init.d/apache2 start
```

Otras ordenes útiles son:

```
sudo /etc/init.d/apache2 restart
```

```
sudo /etc/init.d/apache2 stop
```

La primera reinicia el servidor, mientras que la segunda lo detiene.

Bibliografía

Y así, del mucho leer y del poco dormir,
se le secó el celebro de manera que vino
a perder el juicio.

Miguel de Cervantes Saavedra

Alonso, M. A. Phd thesis. Interpretación tabular de autómatas para lenguajes de adjunción de árboles. *Departamento de Computación, Universidade da Coruña*, 2000.

Alonso, M. A., Carrillo, V. y Díaz, V. J. Análisis sintáctico combinado de gramáticas de adjunción de árboles y de gramáticas de inserción de árboles. *Procesamiento del Lenguaje Natural*, (29), páginas 65–72, 2002.

Booch, G., Rumbaugh, J. y Jacobson, I. *El Lenguaje Unificado de Modelado*. Addison-Wesley, 2007.

Cabrera, I. Alpage Linguistic Processing Chain for French. Disponible en `http://alpage.inria.fr/docs/alpchain-doc.pdf`.

Carbonell, J. El procesamiento del lenguaje natural, tecnología en transición. *Congreso de la Lengua Española*, 1992.

De la Clegerie, E. Tabular algorithms for TAG parsing. *EACL'99. Ninth Conf of European Chapter of the Association for Computational Linguistics*, páginas 150–157, 1999.

De la Clegerie, E. DyALog: A tabular logic programming based environment for NLP. 2005a.

De la Clegerie, E. From metagrammars to factorized TAG/TIG parsers. *IWPT'05*, páginas 190–192, 2005b.

De la Clegerie, E. From meta-grammars to factorized grammars. *Slides presented at Univ. of La Coruña*, 2006.

DE LA CLEGERIE, E., SAGOT, B., NICOLAS, L. y GUÉNOT, M.-L. FRMG: évolutions dún analyseur syntaxique TAG du français. *Journeé ATALA. Quels analyseurs syntaxiques pour le français?*, 2009.

DENNING, P. J., DENNIS, J. B. y QUALITZ, J. E. *Machines, Languages and Computation*. Prentice Hall, 1978.

GRANA, J. Phd thesis. Técnicas de análisis sintáctico robusto para la etiquetación del lenguaje natural. 2000.

KRUCHTEN, P. *The Rational Unified Process: An Introduction*. Addison-Wesley, segunda edición edición, 2003.

LAMPORT, L. *LaTeX: A Document Preparation System*. Addison-Wesley, segunda edición edición, 1994.

LARMAN, C. *UML y patrones: Una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Prentice Hall, segunda edición edición, 2002.

MARTIN, J. Phd thesis. Mieux comprendre les Méta-grammaires. 2006.

MOLINERO, M. A., SAGOT, B. y NICOLAS, L. A morphological and syntactic wide-coverage lexicon for Spanish: The Leffe. *Procesamiento del Lenguaje Natural*, (43), páginas 335–343, 2009.

PRESSMAN, R. S. *Ingeniería de Software. Un enfoque práctico*. McGraw-Hill, quinta edición, 2001.

SAGOT, B. y BOULLIER, P. From raw corpus to world lattices: Robust pre-parsing processing with SxPipe. *Archives of Control Sciences*, vol. 15(LI)(3), páginas 251–261, 2005.

SAGOT, B. y BOULLIER, P. SxPipe 2: architecture pour le traitement pré-syntaxique de corpus bruts. *Traitement Automatique des Langues*, (49), páginas 155–188, 2008.

SAGOT, B., CLÉMENT, L., DE LA CLERGERIE, E. y BOULLIER, P. The Lefff 2 syntactic lexicon for French: architecture, acquisition, use. *LREC'06*, 2006.

SAGOT, B. y DANLOS, L. Méthodologie lexicographique de constitution dún lexique syntaxique de référence pour le français. *Actes du colloque Lexicographie et informatique: bilan et perspectives*, 2008.

THOMASSET, F. y DE LA CLEGERIE, E. Comment obtenir plus des Méta-grammaires. *TALN'05.*, 2005.

WALL, L., CHRISTIANSEN, T. y ORWANT, J. *Programming Perl*. O'Reilly Media, tercera edición, 2000.

Lista de acrónimos

ALEXINA .... *Atelier pour les LEXiques INformatiques et leur Acquisition*, Taller para Léxicos INformáticos y su Adquisición

API ..... *Application Programming Interface*, Interfaz de Programación de Aplicaciones

CASE ..... *Computer Aided Software Engineering*, Ingeniería de Software Asistida por Ordenador

CPU ..... *Central Processing Unit*, Unidad Central de Procesamiento

DFD ..... Diagrama de Flujo de Datos

DTD ..... *Document Type Definition*, Definición de Tipo de Documento

ESEI ..... Escuela Superior de Ingeniería Informática

FQDN ..... *Fully Qualified Domain Name*, Nombre de Dominio Totalmente Cualificado

FRMG ..... *FRench MetaGrammar*, Metagramática Francesa

FTP ..... *File Transfer Protocol*, Protocolo de Transferencia de archivos

GAD ..... Grafo Acíclico Dirigido

GAR ..... Gramática de Adjunción de áRboles

GARE ..... Gramática de Adjunción de áRboles basada en Estructuras de rasgos

GARL ..... Gramática de Adjunción de áRboles Lexicalizada

GIC ..... Gramáticas Independientes del Contexto

GIR ..... Gramática de Inserción de áRboles

GLF ..... Gramática Léxico-Funcional

GPL ..... *General Public License*, Licencia Pública General

HTML ..... *HyperText Markup Language*, Lenguaje de Etiquetado de Hipertexto

HTTP ..... *HyperText Transfer Protocol*, Protocolo de Transferencia de Hipertexto

HTTPS ..... *HyperText Transfer Protocol Secure*, Protocolo Seguro de Transferencia de Hipertexto

INRIA ..... *Institut National de Recherche en Informatique et Automatique*, Instituto Nacional de Investigación en Informática y Automática

LAMP ..... *Linux*, *Apache*, MySQL y *Perl*/*Python*/PHP

LEFFE ..... LÉxico de Formas Flexionadas del Español

LEFFF ..... *LExique des Formes Fléchies du Français*, LÉxico de Formas Flexionadas del Francés

LGPL-LR ..... *Lesser General Public License for Linguistic Resources*, Licencia Pública General Reducida de Recursos Lingüísticos

LPPL ..... Licencia Pública del Proyecto LaTeX

MGCOMP .... *MetaGrammar Compiler*, Compilador de Metagramáticas

MIME ..... *Multipurpose Internet Mail Extensions*, Extensiones Multipropósito de Correo de Internet

PHP ..... *PHP Hypertext Pre-processor*, Preprocesador de Hipertexto PHP

PID ..... *Process IDentification*, Identificación del Proceso

PLN ..... Procesamiento del Lenguaje Natural

RAM ..... *Random-Access Memory*, Memoria de Acceso Aleatorio

RUP ..... *Rational Unified Process*, Proceso Unificado de Rational

UML ..... *Unified Modeling Language*, Lenguaje Unificado de Modelado

URL ..... *Uniform Resource Locator*, Localizador Uniforme de Recursos

XML ..... *eXtensible Markup Language*, Lenguaje de Etiquetado eXtensible