Universidad Carlos III de Madrid

Proyecto: GrammaTeX

Corrector gramatical para documentos LaTeX

Alumno: Julen Beloki Gómez – 100047309

Dirigido por Luis Martí Orosa

2008/2009

TABLA DE CONTENIDO

1. INTRODUCCIÓN

1.1 MOTIVACIÓN

La razón que lleva a iniciar el proyecto que se expondrá a lo largo de este documento ha sido la gran utilidad que se auguraba a la unión de dos herramientas de uso común en el mundo informático hoy en día: el lenguaje LaTeX y los correctores gramaticales.

LaTeX es una herramienta de preparación de documentos multiplataforma ampliamente utilizada, especialmente en el ámbito académico y científico. El éxito de LaTeX respecto de otros procesadores de texto radica en que permite la elaboración de documentos complejos con calidad tipográfica profesional sin excesivo conocimiento acerca del maquetado de los mismos. Los documentos LaTeX son escritos mediante un lenguaje de marcado similar al HTML (salvaguardando ciertas diferencias), donde las etiquetas o comandos sirven para informar al programa en el momento de compilar el documento el significado y formato de la porción de texto al que hacen referencia. Por tanto, LaTeX tomará estos comandos escritos por el usuario y restringirá con ellos la manera que tiene de formatear documentos por defecto.

El problema que presenta LaTeX, además del aprendizaje necesario para manejar diferentes comandos y poder comenzar a escribir documentos con él, resulta en la incapacidad de comprobar y corregir de manera automática y mediante un programa el texto plano que incluye el fichero fuente, pues este tipo de software no existe todavía en el mercado.

Por otro lado, los correctores gramaticales se han vuelto una herramienta casi imprescindible en todos los ámbitos, ya que permiten una escritura de documentos más eficaz. Esta eficacia surge de los diversos usos que se le pueden dar a este tipo de software, como por ejemplo:

- Permiten una escritura más rápida a los usuarios que conocen un determinado idioma, ya que los errores provocados por la velocidad serán revisados por el corrector al final del proceso de escritura.
- Permiten solucionar simples despistes del usuario cuando está escribiendo el documento.
- Permiten a un usuario escribir un documento decente en un idioma que no le es natural, siempre y cuando tenga ciertas nociones del mismo, ya que el software le servirá de apoyo y consejero a la hora de elaborar las frases.

No obstante, los correctores gramaticales del mercado están orientados a la corrección de ficheros o porciones de texto plano, por lo que era necesario crear una herramienta software intermedia que permitiese la conexión entre las dos anteriores. La labor que

deberá desempeñar esta nueva aplicación es la de analizar el documento LaTeX, y transformarlo en algo que pueda ser enviado a un corrector gramatical para que este lo valide.

El motor de este proyecto llamado *GrammaTeX* es por tanto el proveer a los usuarios de LaTeX de una herramienta que les permita utilizar correctores gramaticales sobre sus archivos, pensando especialmente en aquellos que necesitan escribir sus documentos en lenguas que no dominan.

1.2 OBJETIVOS

El objetivo del proyecto es demostrar que se puede crear una herramienta que permita corregir gramaticalmente documentos LaTeX. Para el desarrollo del proyecto GrammaTeX se ha escogido un ciclo de vida iterativo incremental, y la iteración del proyecto que se explica en el presente documento es la primera, el punto de partida, por lo que se ha realizado una versión muy básica pero funcional de GrammaTeX.

En cuanto al tratamiento de los documentos de LaTeX, era necesario fijarse una primera meta relacionada con los comandos que pudiese tratar el programa en desarrollo. La infinidad de comandos propios del núcleo de LaTeX, más un sinfín de paquetes de apoyo para documentos de campos más específicos, nos han obligado a acotar aquellos que iban a ser manejados por esta primera versión de la aplicación. Se ha decidido abordar los comandos más básicos y usados de LaTeX, que son la mayoría de los tratados por Leslie Lamport en los primeros cinco capítulos de la segunda edición de su libro "LaTeX – *A document preparation system*" (Lamport, 1994). La decisión de tratar estos comandos en la aplicación se ha tomado porque son los que mayor impacto van a tener sobre la funcionalidad final de esta, ya que aparecen en documentos tanto de usuarios avanzados como de aquellos que son noveles. De todas maneras, según se iba avanzando en el desarrollo de esta primera versión de la aplicación, se ha detectado que el tratamiento de los diferentes comandos es bastante parecido en muchos de los casos, por lo que estos pueden agruparse en conjuntos que recibirán el mismo tratamiento por parte de la aplicación. Aprovechando este hecho, sería interesante la posibilidad de elaborar alguna forma de añadir nuevos comandos a la aplicación con el mínimo esfuerzo, integrándolos a ser posible dentro de algún tipo de tratamiento ya creado.

Otro de los objetivos de la presente versión, relacionado también con el tratamiento de comandos, y más concretamente con la recién comentada necesidad de agrupar su tratamiento, era realizar el manejo de estos mediante información externa al propio programa. Ya que queremos que el programa siga su desarrollo tras esta primera aproximación a la solución completa, es de vital importancia desarrollar un sistema de tratamiento de comandos fácilmente extensible, y a ser posible, sin necesidad de tocar la implementación del programa. Por ello, se ha escogido almacenar la información necesaria

sobre la sustitución de comandos en archivos XML que funcionen a modo de base de datos. Este segundo objetivo no trata solamente de la creación de un documento XML con la información necesaria, sino que trata también de que la aplicación sea capaz de "digerir" convenientemente esta información a través de un parser que se ajuste a las necesidades concretas de este proyecto.

Mediante el tratamiento de los comandos de LaTeX se consigue transformar un documento de este tipo a otro en texto plano. Es entonces cuando nos vemos en situación de establecer otro de los objetivos fundamentales de la aplicación, que no es otro que la selección de un corrector gramatical acorde a nuestras necesidades, que son:

- Un corrector gramatical de código abierto.
- Una integración en nuestra aplicación que no sea excesivamente costosa, ni en términos de tiempo, ni en términos de conocimientos de programación.
- Un corrector que soporte como mínimo el inglés, ya que es probablemente la lengua más utilizada entre los usuarios de LaTeX cuando escriben con esta herramienta documentos para los ámbitos científico-técnico y académico, que es por otro lado el uso principal de la herramienta. Por ello, se pensó en elaborar este proyecto únicamente para el inglés en esta primera iteración del ciclo de vida, aunque es obviamente deseo nuestro poder hacerlo extensible a otras lenguas en fases posteriores del desarrollo.

Una vez se encuentren herramientas con estas características, habrá que decidir que características de las propias de cada uno de los casos hacen más atractivo un software u otro, por lo que tampoco se parte con una idea clara de que tipo de corrector se usará finalmente, decisión que será tomada a lo largo del proyecto.

Por último, y una vez hayamos solventado y desarrollado la serie de objetivos recién comentados, nos encontramos con la necesidad de pensar en cómo este software va a ser utilizado por sus futuros usuarios. En un principio se han barajado dos opciones: desarrollarla como aplicación "standalone" o independiente, o como plugin del IDE Eclipse. Realizar un plugin de Eclipse parecía originalmente la mejor opción, por ser más elegante y usable, pero presentaba unos costes en tiempo difícilmente asumibles, por lo que fue descartada, y se tomó la decisión de seguir adelante con la aplicación "standalone".

Una vez decididos a realizar la aplicación independiente, se ha decidido que lo primordial es crear una interfaz usable, sencilla de manejar, algo que no tenga prácticamente la necesidad de aprender cómo funciona. Es por ello que la GUI de la aplicación ha sido reducida a la mínima expresión, e implementada pensando en que sea similar a los demás softwares de este tipo.

Sintetizando todo lo comentado anteriormente en esta sección, los objetivos para el desarrollo de la aplicación en esta primera fase son:

- Dar un tratamiento adecuado a los comandos más básicos del lenguaje LaTeX que conoce la aplicación, y uno genérico a aquellos que desconoce, permitiendo en todos los casos obtener un texto lo más adecuado para la corrección gramatical posible.
- Analizar los diferentes comandos que será capaz de tratar la aplicación, detectando grupos para los que se establecerá un tratamiento común.
- Dar la posibilidad a los usuarios de extender el número de comandos tratados por la aplicación, incluyendo a aquellos que no tienen nociones de programación.
- La información sobre la sustitución de comandos debe estar almacenada fuera del programa, en un archivo XML. La aplicación debe ser capaz de acceder a dicha fuente y "digerir" su contenido a través de un parser.
- Buscar, escoger e integrar un corrector gramatical en la aplicación (deberá por lo tanto ser software de código abierto). Se le exige que pueda corregir como mínimo texto en inglés.
- Desarrollar una interfaz de usuario simple e intuitiva.

1.3 ESTRUCTURA DEL DOCUMENTO

En el presente documento se van a explicar todos los detalles importantes referentes al corrector gramatical de documentos LaTeX que hemos denominado como GrammaTeX.

En primer lugar, y a lo largo del segundo apartado de este documento, denominado *Estado del arte*, se va a dar una visión global de las herramientas software que se intentan conjugar en el proyecto. En él, hablaremos en primer lugar de cuál es el origen del sistema de preparación de documentos LaTeX, y de cuáles son las características que hacen de él un software tan utilizado y para el cuál merece la pena desarrollar software de apoyo. A continuación, se hablará también de la evolución histórica de los correctores gramaticales de texto plano, que han pasado de ser herramientas extremadamente básicas e independientes, a asimilar una cantidad inmensa de reglas y ser ciertamente complejas, además de estar plenamente integradas en los procesadores de texto, donde dan consejos al usuario en tiempo real. Se comentarán los diferentes tipos de correctores que han sido candidatos a formar parte del proyecto y sus características. Por último, explicaremos en qué consiste la tecnología XML (Extensible Markup Language) que nos ha servido para guardar la información acerca de las sustituciones de comandos de LaTeX que utiliza GrammaTeX.

En el tercer apartado, denominado *Herramientas*, se explicará brevemente cuales han sido las herramientas utilizadas para el desarrollo del proyecto, tales como Eclipse y sus plugins, Java,… de manera que aquellas personas que continúen con el desarrollo de GrammaTeX, puedan utilizar estas mismas y no perder tiempo en decidirse por uno u otro software, siempre y cuando no encuentren otras herramientas que les parezcan mejores.

El cuarto apartado del documento, denominado *Diseño de la aplicación*, es probablemente la parte más importante. A lo largo de él, se explicará el funcionamiento interno de la aplicación, comentando cuáles han sido las decisiones de diseño que se han tomado y por qué. Todo ello irá acompañado de diagramas que ayuden a entender visualmente y de una manera más sencilla lo que se está leyendo, convirtiéndose en una guía de referencia y consulta básica para quienes deseen entender el proyecto o continuar con él. Este apartado contiene también información acerca del documento XML que guarda las sustituciones LaTeX, donde se pueden encontrar comentarios sobre la forma en que los comandos son sustituidos y sobre el formato que deben tener estos para que la aplicación pueda utilizarlos. Por último, este apartado contiene una sección denominada *Pruebas*, en la que se explicará a que pruebas ha sido sometida la aplicación para verificar su funcionamiento, y cuáles han sido los resultados.

El quinto apartado del documento se llama *Manual de usuario*, y en él se explicará a los futuros usuarios de la aplicación cuáles son los pasos a seguir si se desea realizar una corrección gramatical de algún documento LaTeX. Se han incluido capturas de pantalla de la aplicación en funcionamiento, que se presentan conjuntamente con las instrucciones para facilitar y agilizar el proceso de aprendizaje en el uso de la aplicación.

El sexto y penúltimo apartado del documento son los *Comentarios finales*, donde se realizarán una serie de comentarios acerca de cómo se ha visto el proyecto desde dentro mientras estaba en desarrollo, y una evaluación del mismo una vez terminado. Dentro de este apartado hay una sección llamada *Líneas futuras*, donde se podrá encontrar información importante de cara al desarrollo del proyecto más allá de la versión actual. En él se ha dejado constancia de aquellos aspectos de la aplicación que han sido desarrollados pensando en posibles cambios o extensiones, y de las nuevas funcionalidades que se podrían incorporar al actual proyecto para hacerlo más completo. Este apartado contiene también una sección llamada alternativas, donde se expondrán como su propio nombre indica otras formas de implementar la funcionalidad de la aplicación existente. Por último, se comentarán las conclusiones obtenidas al término del proyecto acerca del trabajo realizado, y una evaluación de la aplicación en función de los objetivos fijados al inicio del mismo.

El séptimo y último apartado del documento es la *Bibliografía*, que contiene referencias a la información consultada tanto durante el desarrollo del proyecto, como en la elaboración del presente documento. En ella se podrán encontrar referencias a libros y enlaces a sitios web.

Por último, y fuera ya del cuerpo principal del documento, se puede encontrar un apéndice que contiene el glosario de términos técnicos que se han utilizado a lo largo de este documento.

2. ESTADO DEL ARTE

2.1 LaTeX

2.1.1 ORIGEN

LaTeX (LaTeX Project) es un sistema de preparación de documentos multiplataforma destinado principalmente a la creación de libros y documentos de carácter técnico o científico. Es diseñado originariamente por Leslie Lamport en 1985 (Lamport, 1994), dando lugar a la primera versión numerada como 2.09. Está formado principalmente por un gran conjunto de macros u órdenes construidas a partir de comandos de TeX, que no es sino otro lenguaje de composición tipográfica creado por Donald E. Knuth (Knuth, 1984).

TeX es un lenguaje de "bajo nivel", en el sentido de que sus acciones son fragmentadas en sub-acciones más simples y elementales que pueden ser realizadas atómicamente. No obstante, aunque TeX fuese una herramienta tipográfica muy potente, carecía de la facilidad de uso que le permitiría acceder a un público más masivo y convertirse en una herramienta fundamental y estándar de facto para la comunidad científico-técnica a la hora de escribir documentos. LaTeX fue ideado para dar solución a este problema, pues cimentándose en TeX, utiliza toda su potencia, pero con una forma de uso mucho más simple, y por ende, también más práctica y atractiva para los usuarios.

Estas características hicieron que LaTeX se extendiese rápidamente entre un amplio sector científico-técnico, pasando prácticamente a ser de uso obligado en comunicaciones, congresos e investigación, además de ser requerido por determinadas revistas a la hora de entregar artículos académicos.

Hoy en día, el mantenimiento y desarrollo del núcleo de LaTeX descansa en manos del LaTeX3 Project (The LaTeX3 project) (Mittlebach & Rowley, 1999). Los creadores de LaTeX se encuentran al frente de este equipo de desarrollo que con la ayuda de voluntarios están creando la próxima versión de LaTeX que será distribuida, denominada LaTeX3.

2.1.2 CARACTERÍSTICAS

2.1.2.1 SOFTWARE LIBRE

LaTeX se distribuye bajo licencia LPPL (The LaTeX project public license), por lo que es software libre pero con ciertas restricciones que impiden distribuirla como GPL o General Public License (Free Software Foundation, 2007).

El código abierto de LaTeX provocó que en sus orígenes, diversos usuarios manipulasen el código extendiendo sus capacidades en función de sus propias necesidades. Esto provocó un

sinfín de versiones diferentes que en muchas ocasiones eran incompatibles entre sí. Leslie Lamport conjuntamente con otros desarrolladores (Frank Mittlebach, Johannes Braams, Chris Rowley y Sebastian Rahtz) crea el llamado "LaTeX3 Project" en 1989 para solventar este problema (The LaTeX3 project).

En otoño de 1993, "LaTeX3 Project" anuncia que va a realizar una reestandarización completa de LaTeX mediante una nueva versión que incluía la gran mayoría de las extensiones adicionales creadas por los usuarios, de manera que se pudiese evitar la fragmentación entre versiones incompatibles de LaTeX 2.09, mientras se daba uniformidad al conjunto convirtiéndolo en una herramienta mucho más potente. A esta nueva versión de LaTeX, y las sucesivas versiones que actualizarán esta hasta llegar a la versión 3, se les denominará LaTeX 2ε.

Además de todas estas nuevas extensiones, la característica más relevante de este esfuerzo de reestandarización fue la arquitectura modular que se le confirió a LaTeX. Esta nueva arquitectura establecía un núcleo central (compilador) que mantenía las funcionalidades de la versión anterior, pero permitiendo incrementar su potencia y versatilidad por medio de diferentes paquetes que solo se cargaban si eran necesarios. De esta forma, LaTeX dispone ahora de infinidad de paquetes destinados a otros tantos objetivos, muchos ofrecidos como parte integrante de la distribución oficial, y otros realizados por terceros para usos más especializados.

Está arquitectura modular permite que la funcionalidad de LaTeX pueda ser extendida según las nuevas necesidades que le surjan a cada usuario, por lo que la ya actual potente base de la distribución LaTeX, unida a su fácil extensibilidad mediante paquetes permite que esta herramienta pueda ser aplicada a cualquier campo del saber. Todo lo acabamos de relatar en este apartado, por sí solo, ya es suficiente para justificar software de apoyo para LaTeX como el que se presenta en el actual proyecto.

2.1.2.2 MULTIPLATAFORMA

Una de las grandes ventajas de LaTeX redunda en que puede funcionar bajo diferentes plataformas. El resultado de la compilación y posterior visualización será siempre la misma, tanto en diferentes sistemas operativos (Windows, Unix, Mac OS, etc.), como con independencia del dispositivo (pantalla, impresora, etc.). Además, un fichero LaTeX puede ser exportado fácilmente a numerosos formatos diferentes como pueden ser: Postcript, PDF, SGML, HTML, RTF, etc.

Más allá de la teórica posibilidad de utilizar LaTeX en cualquier plataforma como acabamos de explicar, a día de hoy, ya existen numerosas distribuciones e IDEs (Integrated Development Environment) que permiten trabajar con él y que funcionan en los sistemas operativos más extendidos en el mercado: TeXnicCenter para Windows (TeXnic Center, 2008), Kile para Linux (Kile - an Integrated LaTeX Environment), TeXShop para MacOS

(TeXShop), e incluso un editor multiplataforma llamado Texmaker (Texmaker: Free LaTeX Editor) que funciona en los tres SSOO recién mencionados. Todos ellos son distribuidos bajo la Licencia GPL, por lo que cualquier usuario puede descargarlos y utilizarlos.

Esta "universalidad" en el uso de LaTeX fundamenta otra de las razones por las cuales resulta interesante desarrollar software de apoyo como GrammaTeX, ya que se está ayudando a una comunidad grande de usuarios a desarrollar mejores documentos.

2.1.2.3 USO

El uso de LaTeX presenta ciertas particularidades respecto a los procesadores de texto habituales

En primer lugar, LaTeX presupone una filosofía de trabajo diferente a la hora de elaborar un documento, ya que con él, un usuario puede centrarse exclusivamente en el contenido del documento sin tener que preocuparse en absoluto de los detalles referentes a la presentación del mismo, al contrario de lo que ocurre con los procesadores de texto habituales, donde el usuario tendrá que ajustar contenido y presentación para obtener exactamente lo que quiere. Esta diferencia ha dado lugar a la terminología WYSIWYG/WYSIWUM, donde el primero de los dos acrónimos ("*What you see is what you get" o "Lo que ves es lo que obtienes")* representa el grupo de procesadores de texto en los que un usuario participa activamente en la presentación del documento, viendo el texto tal y como será presentado, mientras que dentro del segundo grupo ("What you see is what you mean" o "Lo que ves es lo que quieres decir") están los procesadores tipo LaTeX, en los cuales el usuario introduce la información que desea que contenga el documento, mientras que las decisiones sobre formato y presentación son tomadas por el propio programa.

Esta forma de trabajo, implica dividir el proceso de obtención de un documento en dos fases: en la primera fase, deberemos crear un fichero fuente en texto plano que contenga la información que deseamos, además de una serie de comandos o etiquetas con los que informamos a LaTeX sobre ciertos aspectos referentes al significado y formato de cada parte del documento, pasando en la segunda parte del proceso este fichero fuente a LaTeX, que lo compilará y dejará preparado para ser enviado a la salida correspondiente. En este proceso de compilación, LaTeX da al texto escrito una estructura y formato predefinidos que permiten una mayor legibilidad, ya que se usan estilos de composición de texto utilizados durante años en el ámbito de la maquetación e impresión, proceso que además podrá ser alterado en determinados aspectos en función de nuestras necesidades y mediante el uso de las ya mencionadas etiquetas.

Esta forma de trabajar propia de programas como LaTeX tiene evidentemente sus pros y sus contras. La problemática principal es que un usuario novel se encontrará con que necesita asimilar una serie de comandos o etiquetas básicos con los que poder comenzar a escribir documentos, con que deberá compilar de nuevo un documento si desea añadir

cambios al mismo, y con la necesidad de abstraerse de la presentación durante la escritura. De todas formas, cada día existe más software (LyX, TeXmacs, …) que permite trabajar de una forma más visual con LaTeX, pudiendo ver el resultado de la compilación de nuestro documento casi en tiempo real, ahorrándonos tiempo en corregir errores, y permitiéndonos visualizar el documento tal y como será una vez compilado.

No obstante, aunque este funcionamiento basado en comandos puede parecer poco práctico *a priori*, resulta después en una capacidad "casi infinita" de edición de documentos de cualquier tipo con una calidad tipográfica profesional. Esto se debe a las siguientes razones:

- Contiene comandos que permiten estructurar fácilmente el documento (capítulos, secciones, notas, bibliografía,…), haciéndolo más útil y cómodo para artículos académicos, tesis, y libros y documentos técnicos y científicos.
- Tiene capacidades gráficas para el tratamiento de expresiones matemáticas como ecuaciones, notación científica, …
- Cuenta con una gran comunidad de usuarios que aportan ayuda y recursos online, además de extensiones para campos específicos (que abarcan desde la música o la química molecular hasta los circuitos eléctricos o el ajedrez) o nuevas funcionalidades (documentos con un número variable de columnas, transparencias, colores,…).
- Cuenta con comandos que dan soporte a los diferentes lenguajes (humanos), permitiendo elaborar la mayoría de los símbolos o acentos de los mismos.

2.2 CORRECTORES GRAMATICALES

La corrección de un texto desde el punto de vista gramatical es absolutamente esencial, por lo menos si esperamos que el texto sea leído y entendido por alguien que no lo haya escrito. Incluso cuando hablamos de escritores muy capacitados, hay infinidad de factores que provocan que un documento presente errores gramaticales, que van desde el desconocimiento de la totalidad de las reglas gramaticales que constituyen un lenguaje (por ser este un conjunto inmenso), hasta el ritmo de vida y trabajo imperante en la sociedad, que provoca la escritura a contratiempo y un mayor índice de errores gramaticales, pasando entremedias por los simples despistes comunes a todos los mortales. Si a todo ello añadimos la circunstancia de que hoy en día la mayoría de la gente escribe sus documentos en procesadores de texto, estaremos en situación de afirmar que los correctores gramaticales se están volviendo imprescindibles, permitiendo a los usuarios escribir sus textos de una forma más rápida y segura, no teniendo que comprobar todas y cada una de las líneas que escriben, sin comprometer por ello la calidad del documento final, eso sí, sin olvidar que la capacidad de corrección de este tipo de software por el momento no es absoluta, y que por tanto no asegura la obtención de un documento completamente libre de errores.

2.2.1 ORIGEN

Los primeros correctores gramaticales se dedicaban más bien a buscar errores concretos, como fallos de puntuación, en vez de realizar un análisis gramatical como lo hacen hoy en día. El primer sistema de este tipo conocido se llamaba *Writer's Workbench*, estaba formado por varias herramientas de escritura, y se incluía en los sistemas Unix en la década de los setenta. Cada una de las diferentes herramientas que conformaba el *Writer's Workbench* se dedicaba a analizar un aspecto diferente de la escritura. Destacan entre ellas la denominada como *diction* (dicción en español), que se encargaba de realizar un análisis del texto en busca de frases típicas mal usadas o construidas, y elaboraba una lista con aquellas sospechosas de estar mal escritas con sus posibles soluciones; o la denominada como *style* (estilo en español), que realizaba una serie de pruebas de legibilidad cuyos resultados eran mostrados al usuario, con información estadística acerca de lo que se había encontrado en el texto.

Aspen Software fue la encargada en 1981 de lanzar al mercado *Grammatik*, el primer corrector de dicción y estilo desarrollado para ordenadores personales (Radio Shack, TRS-80, CP/M e IBM PC). El desarrollo de *Grammatik* tras su compra a manos de Reference Software lo convirtió también en el primer software capaz de analizar un texto más allá de simples comparaciones con patrones de escritura y frases almacenados en el programa.

A partir de este momento, todos los programas, que hasta ahora solo se habían encargado de la dicción y el estilo, comenzaron a contener y utilizar diferentes niveles de procesado de lenguaje, desarrollando por fin la capacidad de analizar realmente la gramática de un texto, por muy sencillo que este análisis fuese.

Hasta 1992, los correctores gramaticales eran comercializados como "add-on"s, es decir, como programas dependientes de otro principal, que en este caso era un procesador de texto. Por aquella época existían todavía multitud de procesadores de texto, con *WordPerfect* y *Microsoft Word* liderando el mercado. Pero en 1992, *Microsoft* decide incluir un corrector gramatical dentro de *Word*, licenciando y comercializando además como aplicación independiente *CorrecText*, otro corrector de la empresa *Houghton Mifflin*. *WordPerfect* decide entonces contraatacar con la adquisición de *Reference Software*, incluyendo en su software una versión del *Grammatik*.

Hoy en día, hay varios proyectos de software de código abierto que están también desarrollando tecnología para la corrección gramatical, como por ejemplo *Abiword* (AbiSource Community), que es distribuido con un corrector gramatical basado en el *link grammar*, y *LanguageTool* (Naber), utilizado como herramienta que se puede añadir a *OpenOffice* y *Lyx*. Existen también correctores gramaticales comerciales como *Grammar Check Anywhere* y *Grammatica*.

2.2.2 ASPECTOS TÉCNICOS

Los primeros correctores gramaticales se basaban en la comparación del texto a analizar con una serie de patrones contenidos dentro de la aplicación, buscando como hemos dicho en el apartado anterior frases típicas de un idioma concreto, por poner un ejemplo. En estos casos, el núcleo del programa estaba formado por una lista de varios cientos o miles de frases que eran consideradas como lenguaje de un nivel pobre. La lista de frases sospechosas incluía una formación alternativa para cada frase. Por tanto, el proceso llevado a cabo por el corrector gramatical era dividir el texto en frases, comparar cada una de esas frases con patrones similares encontrados dentro del diccionario de frases, y dar a conocer cuáles de ellas se consideraban sospechosas mostrando alternativas que pudiese escoger el usuario en caso de estar de acuerdo con el programa en que la frase era incorrecta. Estos programas, podían de todas formas realizar también algunas comprobaciones de carácter simple, como puede ser el encontrar palabras repetidas, signos de puntuación repetidos, o errores de mayúsculas y minúsculas entre otros.

Sin embargo, el verdadero análisis gramatical de un texto es bastante más complicado. Mientras que los lenguajes de programación utilizados en los ordenadores tienen una sintaxis y gramática muy específicas, los lenguajes naturales tienen una variabilidad prácticamente infinita. Aunque es posible definir una gramática formal que especifique el uso de un lenguaje natural, hay por regla general tantas excepciones en el uso real de la misma, que una gramática formal no es suficiente para implementar un corrector gramatical.

La parte más importante de un corrector gramatical para lenguajes naturales es un diccionario que contenga la totalidad de las palabras propias de dicho lenguaje, conjuntamente con las posiciones que pueden ocupar dentro de una frase. El hecho de que las palabras puedan ser encontradas en diferentes partes de las frases incrementa sobremanera la complejidad de un corrector gramatical. Por tanto, el proceso a seguir por el corrector será identificar cada una de las frases del texto para tratarlas por separado, comprobando todas y cada una de las palabras que las conforman en el diccionario, e intentando entonces analizarla de forma que coincida con las reglas gramaticales propias de un lenguaje concreto. Es entonces cuando a través de la comprobación de esas reglas, puede el programa detectar errores diversos como la concordancia de número o de persona, el orden correcto de las palabras, etc.

Es posible también detectar algunos problemas de estilo en el texto, como puede ser el uso excesivo de la voz pasiva, considerado como mala práctica en la escritura. Después de que cada una de las palabras haya sido contrastada con el diccionario y por tanto hayamos obtenido la estructura que pueden seguir dentro de una frase, es posible detectar la voz pasiva en dicha frase, y reformularla pasándola a voz activa.

2.2.3 CORRECTORES GRAMATICALES CANDIDATOS AL PROYECTO

Se explicarán a continuación los correctores gramaticales que han sido candidatos a formar parte del proyecto, y comentaremos también para cada uno de ellos las ventajas y desventajas que presentaban en relación a GrammaTeX.

2.2.3.1 LINK GRAMAR PARSER

El *Link Grammar Parser* (Temperlay, Sleator, & Lafferty, 2004) es un parser sintáctico del Inglés desarrollado por Davy Temperley, Daniel Sleator y John Lafferty. Está basado en una teoría de la sintaxis inglesa que se conoce como gramáticas *Link Grammar.*

BASE FUNCIONAL: LÓGICA Y NOTACIÓN DE LOS LINK GRAMMARS

La idea básica detrás de este tipo de gramática reside en imaginarse las palabras como elementos de los que salen una serie de conectores. Estos conectores pueden ser de diferentes tipos, y además, cada uno de ellos puede encontrarse apuntando a izquierda o derecha. De esta forma, la unión entre dos conectores puede darse si en una palabra existe un conector de determinado tipo apuntando a la derecha, y en otra palabra tenemos el mismo tipo de conector pero apuntando a la izquierda. A esta unión se le denomina *enlace*. Para representar conectores que apuntan a la derecha, se escribe el tipo de enlace que forma, acompañado del símbolo "+", y los que apuntan a la izquierda con el símbolo "-".

Cada palabra tiene determinadas reglas propias acerca de cómo puede ser conectada a otras, es decir, reglas que constituyen los usos válidos que se le pueden dar. Estas reglas propias no son más que una serie de representaciones de diferentes conectores (tipo y dirección), agrupados en expresiones lógicas, de las cuales podemos obtener todas las posibles combinaciones válidas de uso de conectores para esa palabra. Al conjunto compuesto por todas y cada una de las diferentes combinaciones de conectores válidos de una palabra se le llama "disjunct" (disjunto en español), que es un elemento fundamental para el funcionamiento interno del parser.

De todas formas, para que una frase sea válida no basta solo con que todas y cada una de las palabras que la conforman sean utilizadas acorde a sus propias reglas de uso, sino que además se deben cumplir ciertas reglas globales. Estas reglas globales son dos:

- La planaridad: consiste en que los enlaces no se pueden cruzar, es decir, que solo podrán conectarse dos palabras o grupos de palabras que se encuentren al lado.
- La conectividad: consiste en que todas las palabras de la frase deben estar unidas aunque sea indirectamente, no pudiendo encontrar dos grupos de palabras aislados el uno del otro

La estructura de las frases que maneja link grammar es por tanto algo poco habitual entre los sistemas gramaticales, aunque utilice conceptos bastante cercanos a las gramáticas de

dependencia. En vez de pensar en términos de funciones sintácticas, como pueden ser el sujeto, el objeto, o el predicado, uno debe pensar en términos de la relación existente entre pares de palabras. De todas formas, aunque la forma habitual de analizar gramaticalmente una frase no sea del todo explícita con este sistema, es fácil ver que se puede deducir rápidamente el objeto de una frase, o elementos del tipo de los complementos circunstanciales con una rápida mirada a los enlaces.

CARACTERÍSTICAS DEL PARSER

El parser fue originalmente escrito en código C, y por tanto puede ser ejecutado en cualquier plataforma que posea un compilador de este lenguaje de programación, aunque ya existen hoy en día versiones para Perl, Python, Ruby, Java, Ocaml y .NET. El hecho de que el programa esté disponible para Java, además de ser distribuido bajo licencia BSD (Berkeley Software Distribution), compatible con la GNU General Public License (Free Software Foundation, 2007), hace de este software un candidato a formar parte del proyecto. Existe además una API bien documentada que posibilitaría una inserción más rápida en nuestro sistema.

En segundo lugar, el parser es notablemente robusto gracias a:

- Un diccionario con más de 60000 palabras y conocimiento de una gran variedad de construcciones sintácticas que incluyen algunas idiomáticas y de uso minoritario.
- Un sistema de post-procesado encargado de cubrir la funcionalidad que los link grammar no abarcan por sí mismos, que incluye un sistema llamado “null-link system”, que dota al software de la capacidad de analizar frases en las cuales existe la necesidad de obviar una parte para posibilitar el análisis del resto.
- Es capaz de manejar vocabulario desconocido, y realizar suposiciones inteligentes a partir del contexto y la ortografía sobre las categorías sintácticas de esas palabras desconocidas.
- Conocimiento del uso de mayúsculas, expresiones numéricas, y de los diferentes signos de puntuación.

Quizás el único, aunque importante inconveniente de este parser es que no tiene la extensibilidad lingüística que esperamos que maneje GrammaTeX en versiones futuras, aunque sí es cierto que posee diccionarios para varios idiomas además del inglés, como pueden ser el persa, lenguas árabes o el ruso.

El parser y su desarrollo están siendo mantenidos ahora por el proyecto Abiword (AbiSource Community), un procesador de textos que se distribuye gratuitamente en la red y que utiliza el Link Grammar Parser (Temperlay, Sleator, & Lafferty, 2004) como corrector gramatical integrado en su interfaz

2.2.3.2 LANGUAGETOOL

LanguageTool es un corrector gramatical creado por Daniel Naber (Naber, style_and_grammar_checker.pdf, 2003), y mantenido actualmente por él mismo y Marcin Milkowski. Utiliza reglas contenidas en archivos XML y clases Java para realizar el análisis gramatical.

BASE FUNCIONAL

El primer paso que realiza el LanguageTool a la hora de analizar una frase, es intentar asignar a cada una de las palabras que la conforman una etiqueta que identifique la función que estas desempeñan: nombre, verbo, adjetivo, adverbio, etc. Este proceso de asignación de etiquetas se vuelve bastante complicado teniendo en cuenta que en idiomas como el inglés una misma palabra puede ser utilizada con diferentes objetivos (Ex: house como nombre o verbo). Por ello, se intenta asignar a cada palabra su etiqueta apoyándose en un diccionario que tiene almacenada información probabilística acerca de lo habitual de los diferentes usos. Usa además otros dos diccionarios que guardan esta misma información pero relacionada con tríos de palabras, utilizándose a través de ellos también información de contexto para la asignación. En caso de que las palabras del texto analizado no se encuentren en los diccionarios, se trata de averiguar su etiqueta mediante el análisis de los sufijos y prefijos de las palabras, comienzo con mayúscula, etc.

Por otro lado, se realiza la división del texto en frases, ya que los correctores gramaticales funcionan a este nivel. En primer lugar, se buscan en el texto todas las ocurrencias de los caracteres “.”+“ ”, que en la mayoría de los casos nos proveerán de los límites correctos de las frases. Posteriormente se realiza una revisión de estos límites, ya que habrá que eliminar de este conjunto elementos como las abreviaturas y los acrónimos, además de comprobar algunas otras situaciones de carácter muy específico que se dan en algunos idiomas.

La asignación de etiquetas y división del texto en frases es el paso previo a la división de estas en elementos más significativos para la corrección gramatical, como pueden ser el sujeto y el predicado. Para ello, el languageTool implementa un divisor de oraciones basado en reglas, de forma que las etiquetas asignadas previamente serán agrupadas procurando que cumplan alguno de los formatos establecidos para la correcta formación de estos elementos intermedios.

Es en este punto donde la aplicación cuenta con la información suficiente acerca del texto para comenzar con el análisis. La aplicación cuenta con un archivo .xml que contiene reglas que guardan información acerca de formaciones sintácticas incorrectas. Los elementos principales que forman cada una de estas reglas son:

- El patrón del error, que estará formado por una combinación de etiquetas (aunque pueden ser palabras concretas)

- Un mensaje con información para el usuario acerca del error.
- El idioma para el cual la regla es válida (pueden ser varios).

Hay cierto tipo de reglas que no es posible gestionar desde el archivo xml, por lo que son gestionadas externamente mediante código Java. Esto puede deberse a: incapacidad de auto-referenciación de las reglas en xml (palabras repetidas), de la necesidad del uso de heurísticas basadas en elementos diferentes a los utilizados en la corrección (uso de a/an en el inglés en función de la primera letra de la siguiente palabra, que además tiene sendas excepciones), imposibilidad de referenciación de los espacios en blanco desde el xml, etc.

El languageTool es además capaz de realizar sugerencias acerca del uso de un idioma concreto en cuanto a estilo se refiere, utilizando para ello los dos tipos de reglas comentados anteriormente, tanto xml como Java. Entran dentro de este grupo reglas que corrigen por ejemplo el uso de abreviaciones ampliamente utilizadas, como por ejemplo “don’t” o “wouldn’t”, que aunque plenamente válidos, son poco recomendables para documentos académicos o empresariales.

La base de reglas, tanto xml como Java, puede ser ampliada por los propios usuarios fácilmente, ya que la aplicación ha sido pensada para que uno mismo pueda extenderla. Tiene un sistema de entrenamiento para la asignación de etiquetas a nuevas palabras e idiomas, además de estar pensada para que un usuario no versado en las artes de la programación pueda añadir reglas siguiendo unos sencillos pasos.

Originalmente la aplicación tenía partes dependientes del idioma, del inglés para ser más exactos, pues por ejemplo la asignación de tags basándose en sufijos y prefijos estaba solo habilitada para ese idioma. La versión actual está implementada con mayor cantidad de módulos independientes del idioma, aunque esté optimizada para funcionar con lenguajes como el inglés, francés, alemán y polaco.

CARACTERÍSTICAS

Aunque originariamente este software estaba escrito con Python, la versión actual está escrita en Java, por lo que podría ser integrado fácilmente en nuestra aplicación. Además, es distribuido bajo la licencia LGPL (Free Software Foundation, 2007), y por tanto cumple también en este aspecto los requerimientos de nuestro proyecto por ser software libre. Además, nuestra aplicación está pensado que sea distribuida también como software libre y por tanto no se violan las normas relacionadas con el uso de ninguna de las licencias de la Free Software Foundation.

En cuanto al funcionamiento del LanguageTool, es necesario comenzar diciendo que la calidad de su corrección depende directamente del idioma utilizado. En nuestro caso, el objetivo para esta primera versión del GrammaTeX era tener un corrector para el inglés, y aunque el rendimiento del LanguageTool con él no es del todo malo, idiomas como el polaco y el francés lo duplican y triplican en cantidad de reglas XML respectivamente. De todas

formas, aunque el LanguageTool no esté preparado para realizar un análisis gramatical en inglés demasiado profundo, también es verdad que aparecen nuevas reglas realizadas por gente que colabora en el proyecto de manera regular, por lo que se prevé que próximamente ofrezca un servicio eficaz a nuestra aplicación.

Por otro lado, aún teniendo esta pequeña limitación en el plano funcional, este software representa una gran opción para nuestra aplicación, ya que tiene la posibilidad de ser configurado para trabajar con diferentes idiomas de una manera muy sencilla, simplemente activando la aplicación con un grupo de reglas diferente. Por ello, si la aplicación es desarrollada finalmente con el LanguageTool, la incorporación de otros lenguajes en futuras versiones de GrammaTeX será prácticamente instantánea.

2.3 XML

2.3.1 INTRODUCCIÓN

XML son las siglas en inglés de E**x**tensible **M**arkup **L**anguage (W3C), un metalenguaje extensible de etiquetas que aparece en 1998 desarrollado por un equipo encabezado por Jon Bosack y bajo la tutela del W3C (World Wide Web Consortium).

XML proviene en origen de un lenguaje inventado por IBM llamado GML (Generalized Markup Language). Fue desarrollado para el almacenamiento masivo de información, y como gusto a la ISO, fue normalizado en 1986 creando SGML (Standard Generalized Markup Language). En 1989 se define HTML a partir de SGML, lenguaje simple pero potente que dio origen a Internet, pero que presentaba ciertos problemas de adaptabilidad y compatibilidad. Se intentó definir entonces un subconjunto de SGML que permitiese mezclar elementos de diferentes lenguajes (lenguajes extensibles), la creación de analizadores simples y genéricos, y que utilizase documentos con una sintaxis estructurada.

Nace entonces XML como simplificación y adaptación de SGML, dando cobertura a la mayor cantidad posible de funcionalidad de SGML, y reduciendo en gran medida su complejidad. Para ello, se dejaron de lado muchas características de SGML que estaban pensadas para facilitar la escritura manual de documentos, centrándose en la interpretación y procesado automático de los documentos XML.

2.3.2 DEFINICIÓN

XML se puede definir como una tecnología destinada a la gestión y organización de datos, no centrándose solo en la visualización como hacia HTML, sino dando también soporte a la estructura y a la organización de la información, añadiéndole en definitiva un mayor contenido semántico al documento. Se propone como estándar para el intercambio de información estructurada entre diferentes plataformas, permitiendo la compatibilidad entre

sistemas, y un uso compartido de la información fácil, fiable y seguro. Puede utilizarse como base de datos, como fichero de configuración, puede ser tratado mediante simples editores de texto, etc. XML es una tecnología sencilla, muy fácil de conjuntar con otras que tiene alrededor y que la complementan, dando lugar a posibilidades mucho mayores, además de aumentar la capacidad descriptiva de los lenguajes de marcado actuales.

XML no es un lenguaje, sino un metalenguaje. Esto significa que XML es un lenguaje orientado a establecer las normas y reglas que permiten definir otros lenguajes, siempre que sigan un patrón común. Esta es una de las características que comparte con su predecesor SGML, permitiendo una mayor interoperatividad entre sistemas y una mayor estandarización de los lenguajes de marcado.

La principal ventaja de XML es su capacidad para representar una misma información de diferentes maneras. Esto es lo que se conoce como "extensibilidad" de un lenguaje y se basa en los siguientes conceptos:

- Podemos representar la misma información de diferentes formas en función de los intereses específicos que tenga una organización o una determinada aplicación. Gracias a esto, los documentos XML nos servirán para interconectar diferentes aplicaciones con diferentes estructuras de datos, reestructurando la información para que uno u otro sistema concreto la pueda "digerir" más fácilmente.
- Permite la posibilidad de añadir nuevas etiquetas a un documento XML, extendiéndolo tiempo después de haberlo diseñado y puesto en producción, de modo que se pueda continuar utilizando sin complicación alguna.

Otra de las principales ventajas de XML es que está diseñado de forma genérica. Su cometido principal es definir lenguajes de marcado que sigan unas determinadas pautas, por lo que un analizador de XML es un componente estándar, sin necesidad de crear un analizador específico para cada versión del lenguaje XML. Esto posibilita el empleo de cualquiera de los analizadores ya disponibles, que al estar optimizados, evitan *bugs* (errores en la programación) y aceleran el desarrollo de aplicaciones.

Al no ser un lenguaje en sí mismo, requiere de estructuras auxiliares para definir lenguajes basados en él. Dichas estructuras son las DTD y los Esquemas, que son una manera de definir reglas, haciendo que un documento no solo sea correcto (cumpla las especificaciones del estándar XML), sino que además sea válido (siga una serie de pautas que ha determinado el desarrollador).

Existen diversas formas de procesar documentos XML. Una está basada en la creación de un árbol de objetos referente al documento XML conocida como DOM (Document Object Model), y la otra en eventos que ocurren durante la lectura del mismo conocida como SAX (Simple API for XML). Ambas formas de manipulación de un documento XML tienen una serie de ventajas e inconvenientes que serán comentados más adelante.

Otra característica clave de XML es que es independiente del lenguaje de programación que se utilice y de la plataforma. Puede ser manipulado y escrito por cualquier lenguaje de programación que soporte ficheros de texto plano, extendiendo su utilidad a otros sistemas electrónicos además de los ordenadores.

2.3.3 DEFINICIÓN DEL TIPO DE DOCUMENTO

Cuando definimos vocabularios o tipos de documento en XML, estamos indicando que tipo de elementos podemos encontrarnos dentro del mismo. De esta manera dotamos al documento de la estructura necesaria para que no solo sea correcto desde el punto de vista sintáctico, sino que además sea válido desde el punto de vista estructural.

La diferencia entre un documento correcto y un documento válido puede ser definido de la siguiente manera:

- Un documento correcto es aquel que está definido según las especificaciones de XML, es decir, que respeta las normas de construcción básicas de este metalenguaje.
- Un documento válido, además de cumplir el anterior requisito, debe de estar asociado a un DTD o a un esquema-XML, cumpliendo lo que dichos sistemas le impongan.

2.3.3.1 ESQUEMAS DTD

Los DTD son la forma más sencilla de definir estructuras para XML. Están compuestos de elementos, atributos y entidades.

Los elementos del DTD se declaran mediante el nombre y el tipo de contenido que puede tener dicho elemento. De esta forma, estamos diciendo a los analizadores cuáles deben ser los nombres de las etiquetas, y que otras etiquetas puede contener.

Los atributos del DTD nos sirven para definir los tipos de datos y la necesidad de que estos datos aparezcan o no, dando una mayor facilidad a las aplicaciones para que “digieran” de forma eficaz esa información. Una de las principales desventajas de los DTD frente a otros sistemas de definición de vocabularios XML es el limitado rango de tipos de datos que ofrece.

Las entidades almacenan información que se declara una sola vez. Una vez declarada la entidad se puede llamar a la información que contiene utilizando el identificador de la misma. Esto es realmente útil, sobre todo si se repite algún fragmento de información muchas veces o si se debe de hacer referencia a una información no soportada inicialmente por XML.

Un DTD puede ser declarado externa y/o internamente, realizándose en un fichero aparte o dentro del propio documento. Ambas contienen las reglas principales del lenguaje que se va

a aplicar a ese tipo de documento, y son complementarias, de manera que puede incluirse una, la otra, o ambas simultáneamente.

Una de las grandes ventajas de las DTD es que son un medio ideal para facilitar la comunicación entre aplicaciones. Se puede definir una determinada DTD para la comunicación, en base a la cual las diferentes aplicaciones formatearán la información que quieran compartir antes de enviarla. Con esto conseguimos una vía de comunicación homogénea entre sistemas con organización de la información heterogénea.

2.3.3.2 XML-ESQUEMAS

A diferencia de los DTD, los esquemas ofrecen una estructura más flexible y adaptada al mundo real, dando la opción de definir los vocabularios de una forma más compleja. Los esquemas XML están más cercanos al paradigma de orientación a objetos. Esto hace que hereden algunas de sus características, como las relaciones. Además, nos permite unas mayores posibilidades de reciclado de código, mediante etiquetas que permiten incluir o importar código de otros documentos.

Los esquemas XML están basados en el propio XML, lo que permite utilizar toda la potencia descriptiva de este para la definición de los propios esquemas. Los esquemas están orientados a la definición de estructuras de datos más complejas que las DTD.

En primer lugar, estos esquemas permiten un mayor abanico de tipos de datos, definiendo de una forma más exhaustiva la información que contiene, lo que redunda en una mejora considerable del contenido semántico del documento. Mejora el rendimiento y la calidad de las búsquedas de información. Además dispone de un amplio soporte para estructuras de datos complejas, cosa que los DTD no tienen.

En segundo lugar, permite referenciar múltiples esquemas dentro de un mismo documento. Esta es una de las principales limitaciones de los DTD, que tan solo nos dejaban definir una por documento. Utilizando XML-esquemas podremos definir tantos esquemas como necesitemos por documento.

Los esquemas están compuestos de definiciones y declaraciones. Las definiciones hacen referencia a los tipos de datos. Las declaraciones se refieren a la estructura del documento y a los elementos que pueden incluirse en dicha estructura.

2.3.3.3 ESPACIO DE NOMBRES

Los espacios de nombres (Namespaces) nacen a raíz de un problema que surge cuando hay necesidad de combinar diferentes estructuras basadas en XML dentro de un mismo documento. El conflicto surge cuando dichas estructuras poseen etiquetas idénticas para definir conceptos diferentes. Para solucionar esto disponemos de los Espacios de Nombres.

Para ello, hemos de definir la estructura (DTD) junto al Espacio de Nombres que le corresponde.

Los Espacios de Nombres han de ser únicos, esto quiere decir que no se deben de poder referenciar dos espacios de nombres diferentes con la misma etiqueta identificadora. Esto supone un problema ya que puede haber diferentes compañías o personas que decidan nombrar su espacio de nombres con una misma etiqueta. Para resolver este problema se opto por la creación de un sistema que combina dos factores:

1. Por un lado tenemos la utilización de una etiqueta identificativa del espacio de nombres en cuestión, que se coloca delante de cada elemento. De esta forma podemos saber al espacio de nombres al que pertenece la etiqueta.
2. Por otro lado, mediante una identificación unívoca que actualmente ya está en uso, como son las URL, le damos el complemento necesario a la etiqueta para que no haya problemas con la repetición de identificadores.

La declaración de un espacio de nombres en un elemento implica que se aplicará dicha declaración a ese elemento y a todos los elementos que contenga.

2.3.4 MANIPULACIÓN PROGRAMÁTICA DE XML

2.3.4.1 DOM (DOCUMENT OBJECT MODEL)

DOM (W3C Document Object Model) es una API que permite trabajar con documentos etiquetados. Es compatible con otros lenguajes además de XML, siendo por ejemplo conocido por todos aquellos que han “trasteado” con páginas web el uso de “Javascript” para trabajar con DOM sobre documentos HTML.

Su funcionamiento está basado en la creación de un mapa en memoria con forma de árbol que contiene todos los objetos y estructuras que componen nuestro documento XML. De esta forma, nos es posible acceder a cualquier etiqueta o a su contenido, así como trabajar con grupos de etiquetas o de atributos de forma programática y sin tener que implementar un analizador de código. Además, nos proporciona dicha interfaz de forma estándar, asegurándonos la independencia de plataforma y lenguaje, además de ayudar a la estandarización de los analizadores.

El principal problema que presenta DOM es su falta de eficiencia en la gestión de recursos. Esto se debe a dos factores, la gran cantidad de memoria que se consume si el documento es muy extenso (varios Gigabytes), y el tiempo que tarda una aplicación que utiliza DOM en procesar el documento, ya que se ha de analizar y construir el árbol antes de poder acceder a la información.

Por otro lado, la gran ventaja que presenta DOM es la posibilidad de alterar la información del árbol durante la ejecución del programa que lo utiliza, lo que permite volcar

posteriormente la información de nuevo a un fichero XML, guardando por tanto los cambios realizados.

DOM funciona mediante la definición de jerarquías. Sigue el esquema del documento creando un árbol jerarquizado en memoria. A su vez, el sistema funciona a modo de capa, o de API, entre la interfaz del programador y el propio documento ofreciendo un servicio de acceso a la información altamente simplificado.

2.3.4.2 SAX (SIMPLE API FOR XML)

SAX (SAX Project, 2004) es otra API para trabajar con documentos XML basada en eventos. El analizador simplemente se dedica a ir leyendo el documento y realizar alguna rutina de tratamiento cuando se lanza un evento, es decir, cuando se encuentra una determinada etiqueta o elemento dentro de la estructura, encargándose el sistema que lo utiliza de su gestión. Esto aumenta la eficiencia del sistema frente a DOM, que necesita como hemos dicho antes de la creación de árboles en memoria con el contenido del documento. Por otro lado, esta forma de tratamiento de los documentos también implica ciertas limitaciones para esta API, ya que la interfaz de SAX no nos ofrece las posibilidades que nos ofrecía DOM para desplazarnos por la jerarquía de elementos, cambiar su contenido, y volcar después la nueva estructura sobre un fichero XML.

SAX nace por tanto como respuesta a la necesidad de procesar documentos XML de mayor tamaño, cosa que hemos visto que DOM hace de forma altamente ineficiente. A día de hoy, SAX no es un estándar coordinado por el W3C, lo cual hace que su uso sea más restringido y específico.

3. HERRAMIENTAS

Las herramientas son un aspecto fundamental en el desempeño de una labor, y como tal, se ha procurado seleccionar aquellas que permitiesen un desarrollo eficaz de nuestra aplicación. Se explican a continuación las tecnologías escogidas, y un razonamiento acerca del beneficio que aportaban al proyecto.

3.1 JAVA

La herramienta más importante del proyecto, por ser la base de todo el desarrollo, es el lenguaje de programación **Java** (Sun Microsystems), que presenta ciertas características que hacen de él un lenguaje propicio para este proyecto.

La primera característica importante de Java para este proyecto reside en que es un lenguaje orientado a objetos, lo cual permite que en desarrollos con un ciclo de vida iterativo incremental sea más fácil añadir funcionalidad o alterar la ya existente sin excesivas complicaciones. Esto tiene que ver con que el código esté dividido en objetos coherentes e independientes, que pueden ser transformados o sustituidos individualmente, ofreciendo por ello una base más estable para el diseño de un sistema software

Otra característica favorable de este lenguaje relacionado también con la orientación a objetos, es que permite la reutilización de código a través de clases genéricas desarrolladas por otros usuarios u organizaciones, premisa por otro lado fundamental en la Ingeniería del Software. De esta manera, podremos realizar un desarrollo más eficaz, ya que además de reducir drásticamente el tiempo de desarrollo, estaremos también incorporando a nuestro sistema componentes de comprobada calidad para tareas genéricas.

Por último, pero no por ello menos importante, sino todo lo contrario, reside en la posibilidad de ejecución de proyectos Java con independencia de la plataforma. Esto significa que una vez hayamos implementado la totalidad del sistema, cualquier usuario independientemente del sistema operativo que utilice, o el hardware sobre el que este funcione, podrá ejecutar nuestra aplicación si tiene la máquina virtual de Java (JVM) que puede descargarse directamente desde la página de Sun. GrammaTeX ha sido concebido para el uso en diferentes plataformas, pues consideramos importante poder brindar la ayuda que supone a una comunidad de usuarios cuanto más amplia mejor.

Además de las características propias de Java, que lo hacían interesante por sí mismo, este lenguaje de programación suponía no tener problemas de integración con los correctores gramaticales, que también están implementados con él, sin olvidar que LaTeX es también una herramienta multiplataforma.

3.2 ECLIPSE

Una vez seleccionado Java como lenguaje de programación a utilizar en el proyecto, quedaba escoger el entorno que siendo compatible con este lenguaje, proporcionase una forma de trabajar lo más cómoda posible. De entre los posibles candidatos se ha seleccionado el IDE Eclipse (The Eclipse Foundation), ya que aportaba soluciones para todos los aspectos relacionados con nuestro proyecto, y que son explicados a continuación.

En primer lugar, Eclipse es un entorno de desarrollo disponible para todo aquel que lo quiera descargar desde su página oficial, ya que se distribuye gratuitamente. Fue desarrollado originalmente por IBM en 2001, y desde entonces vienen dándose nuevas versiones que lo hacen cada vez más robusto y eficaz, hasta llegar a la versión 3.5, también conocida como Galileo y con la cual se ha desarrollado el proyecto. A día de hoy, Eclipse viene siendo sinónimo de una buena ingeniería del software, siendo ejemplo de utilidad, usabilidad y extensibilidad.

Una característica apreciada por los usuarios de Eclipse es el hecho de que esté pensado para facilitar el trabajo a estos. Da infinidad de facilidades al usuario para que pueda personalizar el entorno de trabajo según sus preferencias, ofreciendo diferentes perspectivas en función del tipo de trabajo que vayamos a desempeñar (implementación, debugging,…). Tiene además muchos elementos gráficos que ayudan al usuario a programar mejor y más rápido, con un asistente de código fantástico que incluso incluye información sobre la API de Java, y elementos gráficos que permiten detectar rápidamente errores, variables, notas, etc.

Eclipse goza de la posibilidad de incorporarle un sinfín de plugins (Eclipse Plugin Central), destinados a facilitar el uso de diferentes tecnologías que pueden ser utilizadas conjuntamente dentro de un mismo proyecto, facilitando el desarrollo y gestión del mismo a través de una herramienta común. En nuestro caso particular, esta ha sido una de las principales razones que nos ha llevado a decantarnos por este software, ya que además de desarrollar en Java, necesitábamos hacer uso de una serie de plugins que permitiesen manejar las diferentes tecnologías que formaban parte del proyecto:

- Manejo de archivos LaTeX
- Manejo de archivos XML
- Manejo de clases visuales Swing para la interfaz de la aplicación
- Mantener a través del subversión un control de versiones, permitiendo al tutor del proyecto controlar el avance de la aplicación a través de las actualizaciones realizadas periódicamente.

3.2.1 PLUGINS DE ECLIPSE UTILIZADOS

3.2.1.1 TEXLIPSE

TeXlipse (TeXlapse-team) es un plugin que añade soporte para LaTeX al Eclipse IDE. En principio está enfocado a usuarios que ya conocen el funcionamiento básico de LaTeX, pero es también una ayuda muy útil para el novato a la hora de escribir documentos. Se encuentra disponible a través de las actualizaciones de Eclipse en el repositorio: http://texlipse.sourceforge.net; donde además se podrá encontrar documentación que facilitará su uso.

En nuestro caso el uso principal que se le ha dado ha sido el de la edición de documentos LaTeX que servían de prueba para la aplicación, aunque con él se pueden incluso compilar estos archivos si se tiene una distribución de TeX instalada en el equipo. Las características principales que lo han hecho útil para esta primera fase del proyecto han sido el resaltado de sintaxis, el asistente de código (completa comandos y referencias), el corrector ortográfico, y un esquema en forma de árbol que permite la rápida navegación dentro del documento, aunque tiene muchísimas otras más potentes y útiles para usuarios avanzados.

3.2.1.2 SUBCLIPSE

Subclipse (Tigris.org) es un plugin que añade soporte para el subversion mediante nuevos comandos naturales a este, y una interfaz gráfica para su uso que se integra en la habitual del IDE Eclipse. El subversion, también conocido como SVN, es un sistema de control de versiones mediante el cual se facilita la gestión de un proyecto en el que trabajan diferentes personas, de manera que los cambios que estos realizan no se "pisen" los unos a los otros.

En nuestro caso, al haber un único programador, el sentido de utilizar SVN radica en la posibilidad de realizar "commits" (actualización de la versión del proyecto almacenada en el servidor con la copia local) periódicamente para que el tutor del proyecto pueda seguir más de cerca los avances, pudiendo así devolver este sus impresiones y mejorar el proceso de implementación.

El software está disponible a través de dos medios:

- Puede ser descargado desde la página http://subclipse.tigris.org y posteriormente instalado manualmente.
- Puede instalarse directamente en Eclipse desde la dirección http://subclipse.tigris.org/update_1.6.x mediante el gestor de actualizaciones.

3.2.1.3 VISUAL EDITOR

El Visual Editor (The Eclipse Foundation) es un plugin de código abierto para el Eclipse IDE que añade a este herramientas para editar visualmente clases Java, con clases visuales en particular, y desarrollar así interfaces gráficas. El desarrollo y mantenimiento de este software lo gestiona el mismo "Eclipse Project", por lo que es distribuido también de manera gratuita bajo la licencia "Eclipse Public License" siguiendo la misma filosofía que el IDE, lo cual lo ha hecho más atractivo que otros plugins populares para desarrollo de GUIs como *WindowsBuilder Pro* y *Jigloo SWT/Swing GUI Builder*.

Otro de los principales atractivos que tiene este plugin es su fácil manejo y aprendizaje. Quizás resulte un poco simple para usuarios expertos con metas complejas a la hora de desarrollar interfaces gráficas, pero es desde luego una herramienta perfecta si se tienen conocimientos básicos de SWING. Su uso se realiza mediante el manejo de una paleta donde están contenidos todos los elementos SWING y SWT que pueden ser utilizados, y una vista que permite configurar las propiedades de estos. Hay además una vista que muestra la aplicación asociada al código que implementamos actualizada en todo momento, de manera que los cambios en el código o la vista son inmediatamente reflejados en ambos.

Hay que reseñar para aquellos que pretendan descargarlo y utilizarlo, que hay versiones del IDE Eclipse que presentan conflictos con las dependencias que tiene el Visual Editor sobre los "EMF Model Utilities", además de otros conflictos que se dan con ciertos plugins. En mi caso concreto, estos problemos fueron resueltos tras utilizar como base la distribución *Eclipse Classic 3.5.1*, a la que antes de nada se le instalo el Visual Editor versión 1.4, para seguir después con todos los demás plugins necesarios.

El software puede obtenerse de las siguientes formas:

- Puede ser descargado desde la página http://www.eclipse.org/vep/downloads (hay otras pero esta es la más segura y rápida) y posteriormente instalado manualmente.
- Puede instalarse directamente en Eclipse desde la dirección http://download.eclipse.org/tools/ve/updates/1.4 una vez dentro de *Help > Install New Software*.

4. DISEÑO DE LA APLICACIÓN

4.1 FASE DE ANÁLISIS

En esta sección se va a describir qué requisitos fueron considerados como básicos a la hora de iniciar el proyecto. Con la especificación de estos requisitos se obtiene el panorama básico sobre el cual se ha desarrollado posteriormente la aplicación, verificando que lo aquí comentado era correcto, e introduciendo cambios en función de las necesidades encontradas según se iba implementando la aplicación.

4.1.1 REQUISITOS FUNCIONALES

La aplicación debe ser capaz de recibir como entrada un documento LaTeX y transformarlo a texto plano para posteriormente ser analizado gramaticalmente por una aplicación externa.

La corrección gramatical es una función de la que no se va a encargar nuestra aplicación, sino que el documento a analizar va a ser enviado a una aplicación externa que nos devolverá los errores encontrados. Los correctores gramaticales solo son capaces de analizar texto plano, ya que no conocen el lenguaje que utiliza LaTeX. Esto implica que el primer paso que debe dar la aplicación es realizar una conversión de código LaTeX a texto plano, por lo que será necesario establecer unas pautas de transformación de uno a otro. La aplicación debe ser capaz de entender lo que significa cada tag, y realizar una sustitución de este por texto que desempeñe la misma función dentro de la frase origen, de manera que seamos capaces de dejar el máximo número posible de frases analizables en el texto.

Una vez realizado este paso, el texto deberá de ser analizado gramaticalmente, por lo que es necesario escoger un corrector gramatical que se adecúe a nuestras necesidades, y ver la manera en que podemos conectar ambas aplicaciones. Será necesario elaborar una serie de métodos y clases que permitan aprovechar al máximo la información de la que provee el corrector, para poder dar el máximo de facilidades en la corrección al usuario final de nuestra aplicación.

La aplicación debe ser fácil de manejar y permitir al usuario familiarizarse con ella en el menor tiempo posible. Este requisito puede ser cumplimentado de dos maneras: mediante una aplicación independiente que tenga una interfaz de usuario simple y manejable, o integrando nuestra aplicación dentro de otra aplicación que trate documentos LaTeX. La elaboración de una aplicación independiente presenta una curva de aprendizaje menos pronunciada, ya que simplemente tendríamos que desarrollar una interfaz SWING sin restricciones en la implementación. La opción de integrar nuestra aplicación en un procesador de texto o en una IDE como pueda ser Eclipse, ya sea tanto a través de algún plugin ya creado como TeXlipse o de uno creado para la ocasión, implicaría un desarrollo más lento, sin posibilidad de establecer un coste temporal aproximado, pero por otro lado

volvería más atractiva nuestra aplicación para los usuarios, ya que estos no tendrían que ejecutar la aplicación por separado cada vez que quieran corregir un documento, posibilitando además la corrección “on the fly”.

Por último, es necesario basar la transformación del documento original en información que los mismos usuarios puedan generar e introducir en la aplicación sin necesidad de tener conocimientos de programación. Esto no solo da facilidades a los usuarios, sino que además permite separar el tratamiento de la información de cara a poder realizar cambios en la aplicación con menor esfuerzo. Por ambos motivos es necesario que dicha información no esté implementada con código, sino que se guarde en algún elemento externo. La tecnología XML representa una opción ideal en este sentido, ya que es muy fácil de utilizar y además existen numerosos parsers y clases en la propia API de Java que han sido desarrollados para manejar este tipo de documentos. Por lo tanto, será un requisito de la aplicación el incorporar alguna forma de obtener y almacenar información a través de este medio.

4.1.2 REQUISITOS DE RENDIMIENTO

No hay restricciones importantes en este sentido. La aplicación debe de mantener un nivel de respuesta aceptable, y no se puede hablar de los tiempos de una manera más concreta ya que el tiempo de análisis depende de diversos factores, como pueden ser el tamaño del documento, la cantidad de comandos LaTeX que contiene, el anidamiento de estos, …

Uno de los factores relacionados al rendimiento que si puede ser medido es el tiempo que tarda nuestra aplicación en inicializar el corrector y que este pueda ser utilizado. La aplicación debe de ser capaz de iniciar el corrector en menos de dos segundos si queremos que pueda ser integrado en otro software para correcciones “on the fly”, ya sea en la versión actual, o en futuras iteraciones del ciclo de vida.

4.1.3 REQUISITOS DE INTERFAZ

Se prevé que la interfaz necesaria para el manejo de la aplicación vaya a ser muy simple, ya que no se requieren muchos controles para su uso. Debe facilitar en la medida de lo posible las acciones del usuario, permitiendo ser utilizada casi instintivamente. Esto es aplicable tanto al caso de la integración en otro software mediante la adición de los controles propios de nuestra aplicación, como al caso de la creación de una interfaz propia.

Por otro lado, y debido al público al que va destinado, sería preferible mantener un aspecto serio, formal, libre de florituras. La maquetación debe de estar pensada para resaltar los elementos y controles principales. En este sentido, será necesario además que los controles estén situados de manera que se agrupen acciones relacionadas a las distintas funciones, y permitir así un uso más eficaz. Sería deseable que se incluyesen además ayudas visuales, de manera que un usuario pueda distinguir en todo momento que controles son susceptibles de ser manipulados y cuáles no.

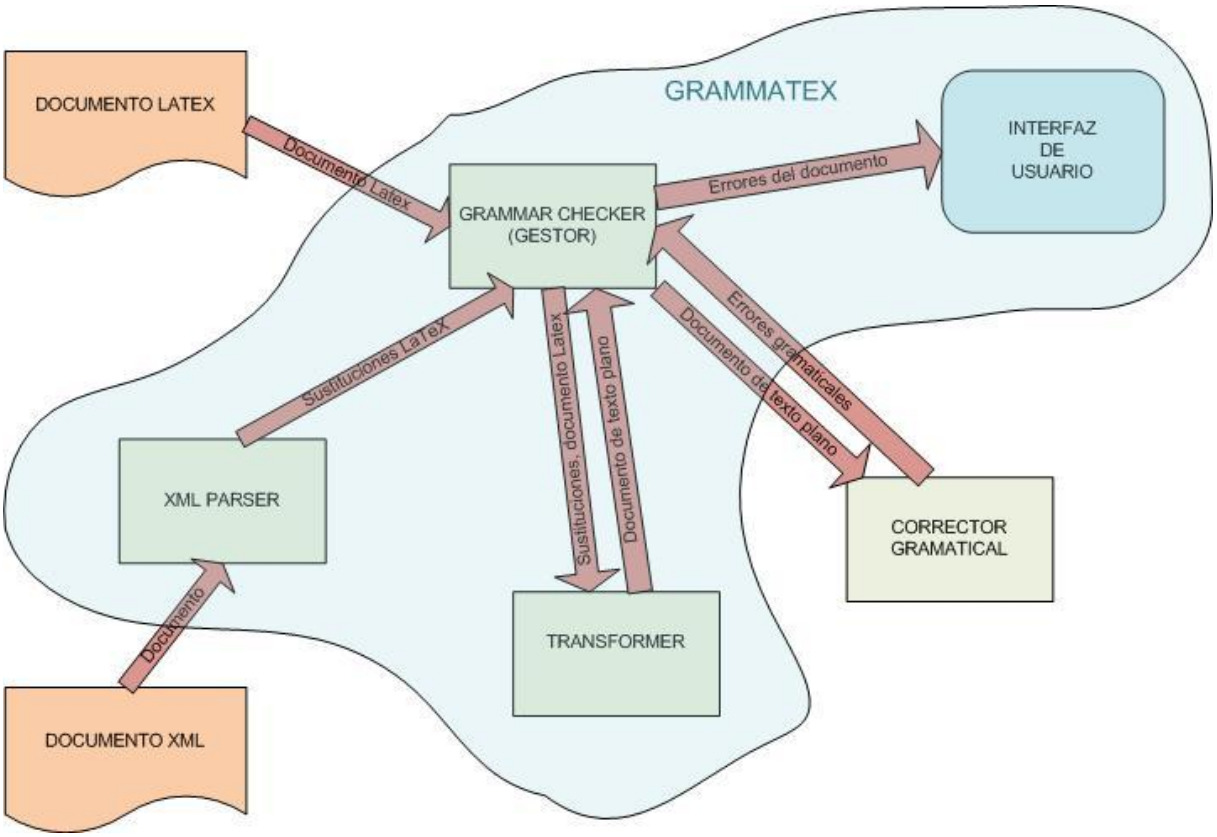
4.1.4 REQUISITOS SOFTWARE

En cuanto a los requisitos software, la aplicación debe estar desarrollada en el lenguaje de programación Java por varios motivos:

- El desarrollo en Java va a permitir que la aplicación pueda ser utilizada en diferentes plataformas.
- Hay disponibles recursos en Java que permitirían acelerar parte del proceso de desarrollo.
- Es el lenguaje de programación que permitiría una conexión más sencilla con los correctores gramaticales preseleccionados para el proyecto.

4.2 ARQUITECTURA DEL SISTEMA

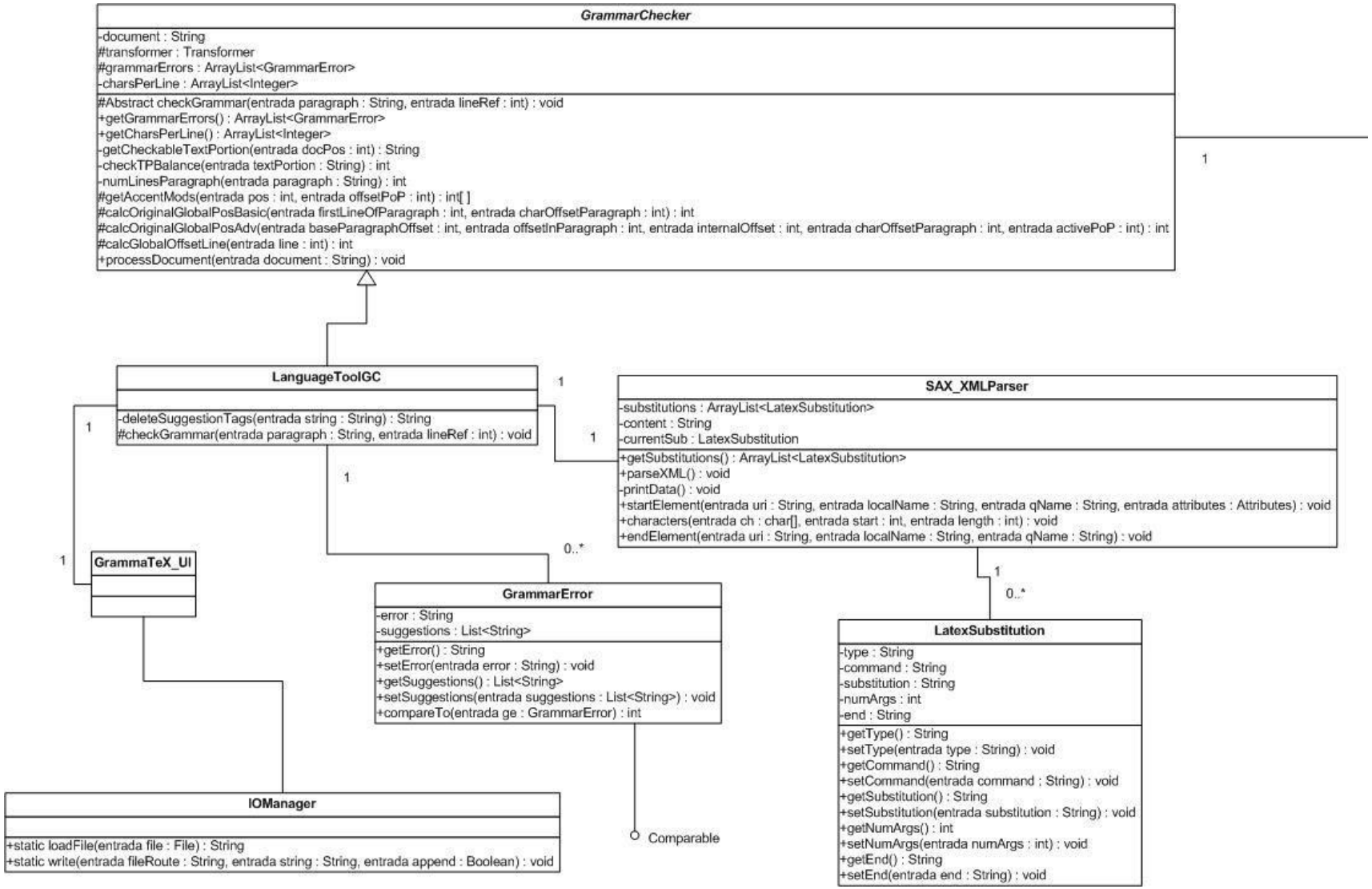
Se muestra a continuación el diagrama de la arquitectura que finalmente tendrá la aplicación:

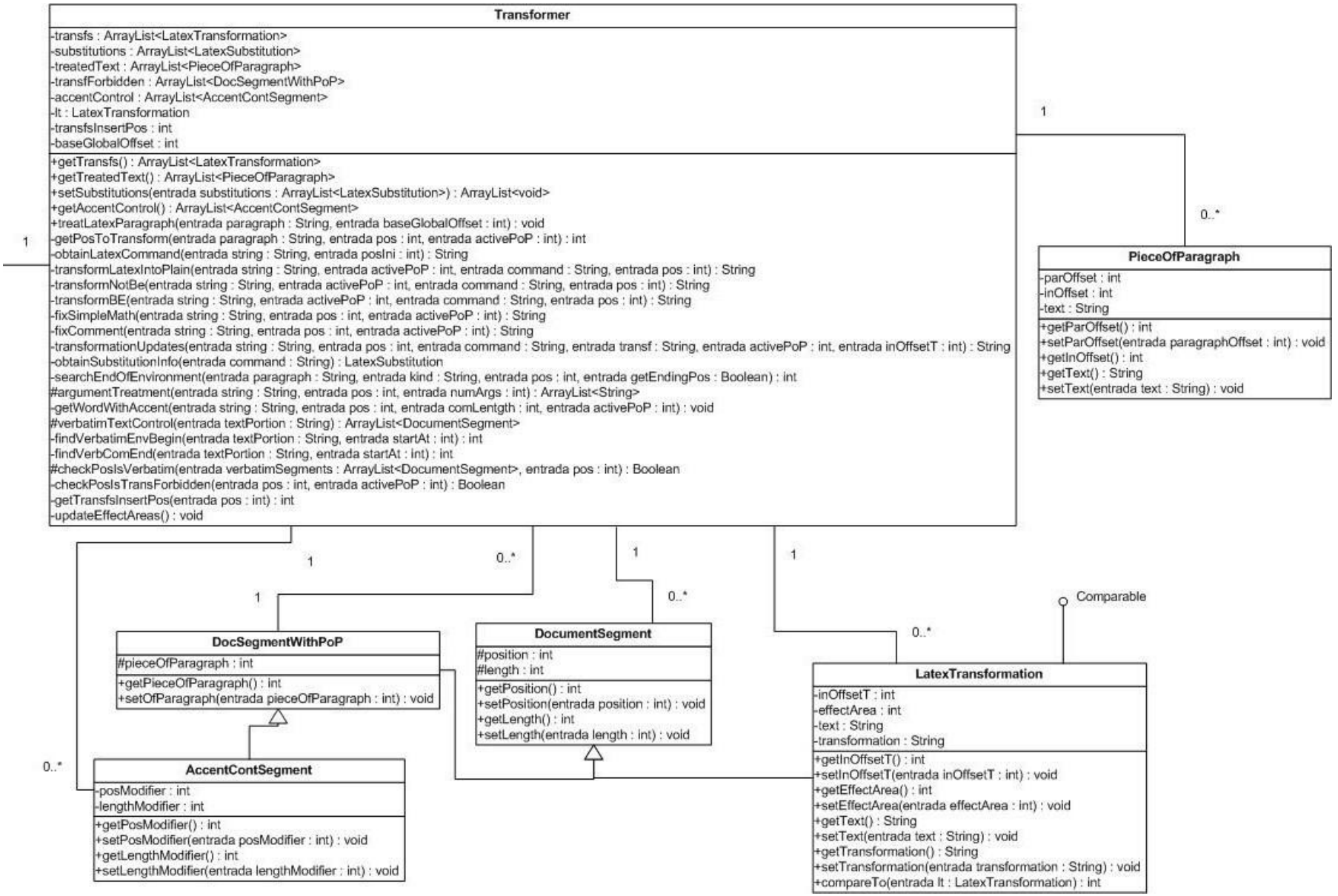


El proceso comienza con la recepción del documento original en el GrammarChecker, que recibe también la información que el parser de XML ha generado con el documento de sustitución de comandos. Esta información es enviada al transformer, que contiene los

métodos responsables de la transformación del documento, los cuales saben interpretar la información de las sustituciones. Una vez finalizado el proceso de transformación, el documento está listo para ser enviado de nuevo a través del GrammarChecker hasta el corrector gramatical que documentará los posibles errores que encuentre en las porciones de texto que va recibiendo. Finalmente, el GrammarChecker tomará los errores gramaticales, y con información contenida en el Transformer calculará las posiciones reales de los errores para que sean mostrados en el documento de la interfaz de usuario.

4.3 DIAGRAMA DE CLASES DE DISEÑO




Transformer
-transfs : ArrayList<LatexTransformation>
-substitutions : ArrayList<LatexSubstitution>
-treatedText : ArrayList<PieceOfParagraph>
-transfForbidden : ArrayList<DocSegmentWithPoP>
-accentControl : ArrayList<AccentContSegment>
-lt : LatexTransformation
-transfsInsertPos : int
-baseGlobalOffset : int
+getTransfs() : ArrayList<LatexTransformation>
+getTreatedText() : ArrayList<PieceOfParagraph>
+setSubstitutions(entrada substitutions : ArrayList<LatexSubstitution>) : ArrayList<void>
+getAccentControl() : ArrayList<AccentContSegment>
+treatLatexParagraph(entrada paragraph : String, entrada baseGlobalOffset : int) : void
-getPosToTransform(entrada paragraph : String, entrada pos : int, entrada activePoP : int) : int
-obtainLatexCommand(entrada string : String, entrada posIni : int) : String
-transformLatexIntoPlain(entrada string : String, entrada activePoP : int, entrada command : String, entrada pos : int) : String
-transformNotBe(entrada string : String, entrada activePoP : int, entrada command : String, entrada pos : int) : String
-transformBE(entrada string : String, entrada activePoP : int, entrada command : String, entrada pos : int) : String
-fixSimpleMath(entrada string : String, entrada pos : int, entrada activePoP : int) : String
-fixComment(entrada string : String, entrada pos : int, entrada activePoP : int) : String
-transformationUpdates(entrada string : String, entrada pos : int, entrada command : String, entrada transf : String, entrada activePoP : int, entrada inOffsetT : int) : String
-obtainSubstitutionInfo(entrada command : String) : LatexSubstitution
-searchEndOfEnvironment(entrada paragraph : String, entrada kind : String, entrada pos : int, entrada getEndingPos : Boolean) : int
#argumentTreatment(entrada string : String, entrada pos : int, entrada numArgs : int) : ArrayList<String>
-getWordWithAccent(entrada string : String, entrada pos : int, entrada comLentgth : int, entrada activePoP : int) : void
#verbatimTextControl(entrada textPortion : String) : ArrayList<DocumentSegment>
-findVerbatimEnvBegin(entrada textPortion : String, entrada startAt : int) : int
-findVerbComEnd(entrada textPortion : String, entrada startAt : int) : int
#checkPosIsVerbatim(entrada verbatimSegments : ArrayList<DocumentSegment>, entrada pos : int) : Boolean
-checkPosIsTransForbidden(entrada pos : int, entrada activePoP : int) : Boolean
-getTransfsInsertPos(entrada pos : int) : int
-updateEffectAreas() : void
1
0..*
PieceOfParagraph
-parOffset : int
-inOffset : int
-text : String
+getParOffset() : int
+setParOffset(entrada paragraphOffset : int) : void
+getInOffset() : int
+getText() : String
+setText(entrada text : String) : void
DocSegmentWithPoP
#pieceOfParagraph : int
+getPieceOfParagraph() : int
+setOfParagraph(entrada pieceOfParagraph : int) : void
DocumentSegment
#position : int
#length : int
+getPosition() : int
+setPosition(entrada position : int) : void
+getLength() : int
+setLength(entrada length : int) : void
Comparable
LatexTransformation
-inOffsetT : int
-effectArea : int
-text : String
-transformation : String
+getInOffsetT() : int
+setInOffsetT(entrada inOffsetT : int) : void
+getEffectArea() : int
+setEffectArea(entrada effectArea : int) : void
+getText() : String
+setText(entrada text : String) : void
+getTransformation() : String
+setTransformation(entrada transformation : String) : void
+compareTo(entrada lt : LatexTransformation) : int
AccentContSegment
-posModifier : int
-lengthModifier : int
+getPosModifier() : int
+setPosModifier(entrada posModifier : int) : void
+getLengthModifier() : int
+setLengthModifier(entrada lengthModifier : int) : void

4.4 CLASES DE DISEÑO

A continuación se explican en detalle las clases que se han implementado como solución al problema planteado cumpliendo con los requisitos obtenidos en la fase de análisis. Para ello, se citarán los aspectos más relevantes de cada clase, dándose información algo más extensa en el caso de los atributos y métodos para facilitar la comprensión de su funcionamiento.

En las declaraciones de los atributos se ha utilizado notación UML para denotar la privacidad, y en el caso de los métodos se muestran las cabeceras tal y como han sido declaradas en Java. Hay algunos métodos que no han sido incluidos por su simpleza y cotidianidad, como el método toString, y los getters y setters (solo se muestran si no hay ningún otro método en la clase).

Es necesario saber antes de la lectura de las clases que en el programa se manejan dos tipos de posicionamiento. Tenemos por un lado el posicionamiento real, que alude a la posición que guarda un elemento en el documento original, y por otro lado, el posicionamiento relativo, que son las posiciones que ocupan los elementos dentro del texto transformado, las cuales varían en función de las transformaciones realizadas. Las posiciones relativas pueden ser convertidas a posiciones reales mediante el cálculo de los offsets producto de las transformaciones, ya que estas se guardan en una lista en la clase Transformer.

4.4.1 ACCENT CONT SEGMENT

Archivo: AccentContSegment.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s): DocSegmentWithPoP, DocumentSegment (no directa)

Subclase(s): No

Descripción: Clase que solo sirve para guardar información acerca de los acentos que han sido tratados al convertir el archivo LaTeX a texto plano. Esto es necesario a efectos de mostrar correctamente los errores que contienen acentos en la vista, ya que estos son los únicos comandos LaTeX que alteran el tamaño de una palabra que el corrector puede detectar como errónea.

Atributos:

- **# int position**: almacena la posición en la que comienza una palabra que contiene algún acento. La posición es relativa a la transformación a texto plano que se ha realizado sobre la porción de texto que la contenía. Es un atributo heredado.

- **# int length**: almacena la longitud de la palabra que contiene el acento una vez convertida a texto plano. Es un atributo heredado.
- **# int pieceOfParagraph**: referencia hacia el pieceOfParagraph del texto transformado dentro del cuál se encontró la palabra acentuada. Es un atributo heredado.
- **- int posModifier**: modificador que se añade a la posición donde encuentra un error un corrector gramatical.
- **- int lengthModifier**: modificador que se añade a la longitud de la palabra en la que encuentra un error un corrector gramatical.

Métodos: únicamente **getters & setters**.

4.4.2 DOC SEGMENT WITH POP

Archivo: DocSegmentWithPoP.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s): DocumentSegment

Subclase(s): AccentContSegment

Descripción: Clase que sirve para guardar información acerca de segmentos de texto durante el tratamiento de conversión del archivo LaTeX a texto plano. Añade a su superclase DocumentSegment el atributo pieceOfPragraph para poder controlar que transformaciones afectan su posicionamiento y cuáles no.

Atributos:

- **# int position**: almacena la posición en la que comienza el fragmento de texto cuyo control deseamos realizar. Es un atributo heredado.
- **# int length**: almacena la longitud del fragmento de texto. Es un atributo heredado.
- **# int pieceOfParagraph**: referencia hacia el pieceOfParagraph del texto transformado dentro del cual se encontró el fragmento de texto.

Métodos: únicamente **getters & setters**.

4.4.3 DOCUMENT SEGMENT

Archivo: DocumentSegment.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s): No

Subclase(s): DocSegmentWithPoP, AccentContSegment (no directa)

Descripción: Clase padre de todas aquellas otras clases que guardan información acerca de segmentos de texto que requieren de un tratamiento especial a lo largo del proceso de transformación de una porción de documento LaTeX a texto plano.

Atributos:

- **# int position**: almacena la posición en la que comienza el trozo de documento al que hace referencia. Se utiliza tanto para almacenar posiciones reales (interfaz gráfica), como para relativas al proceso de transformación.
- **# int length**: almacena la longitud del trozo de documento al que hace referencia.

Métodos: únicamente **getters & setters**.

4.4.4 GRAMMAR CHECKER

Archivo: GrammarChecker.java

Paquete: grammaTeX.project

Tipo de clase: Abstracta

Superclase(s): No

Subclase(s): LanguageToolGC

Descripción: Es la clase que organiza todo el proceso que lleva a cabo la aplicación. Es la clase que recibe la solicitud de analizar un documento desde la interfaz gráfica y decide la porción del mismo que será analizada en cada iteración, para posteriormente delegar en una de las subclases creadas a partir de su plantilla la conversión a texto plano y búsqueda de potenciales errores gramaticales. Posee también los métodos que permiten posicionar correctamente dentro del texto original un error, partiendo de la posición del error encontrado por los analizadores gramaticales.

Atributos:

- **- String document**: guarda una copia de la variable que se le manda al GrammarChecker como parámetro desde la interfaz gráfica con el contenido del documento activo tal y como puede verse en la vista, es decir, con los cambios que se le hayan realizado.
- **# Transformer transformer**: esta instancia de la clase Transformer asociada al GrammarChecker es la encargada de la transformación del documento LaTeX a

texto plano. Guarda toda la información necesaria que utilizan el GrammarChecker y sus subclases para la gestión de los errores gramaticales.

- **# ArrayList<GrammarError> grammarErrors**: en esta lista se van añadiendo los errores gramaticales encontrados.
- **- ArrayList<Integer> charsPerLine**: para cada n, donde n es una posición cualquiera de la lista, se guardará el número de caracteres que alberga esa línea en el texto original. Se utiliza para calcular posiciones en el documento.

Métodos:

- **abstract protected void checkGrammar(String paragraph, int lineRef) throws IOException:** este método es el encargado de la transformación y análisis de cada fragmento de documento, y por tanto serán las subclases de GrammarChecker las encargadas de definir su funcionamiento según las características de cada corrector gramatical.
- **private String getCheckableTextPortion(int docPos)**: este método es el encargado de ir obteniendo párrafos del texto original desde la variable de instancia documento a partir de la posición obtenida como parámetro. El proceso se lleva a cabo por líneas para poder guardar una referencia del número de caracteres que alberga cada una, uniéndose al final del proceso para ser devueltas en conjunto. Para que un conjunto de líneas se consideré como apto para ser devuelto al método llamante, debe de estar balanceado, teniendo el mismo número de comandos "\begin", que de comandos "\end" (por razones de tratamiento).
- **private int checkTPBalance(String textPortion)**: este metodo es el que calcula para el getCheckableTextPortion si una determinada porción de documento está balanceado. Para ello primeramente realiza un análisis de los entornos y comandos de tipo verbatim, contrastando con los resultados las posiciones de los "\begin" y "\end" encontrados, para determinar si cuentan o no. Utiliza evidentemente texto no transformado.
- **private int numLinesParagraph(String paragraph)**: calcula cuantas lineas hay dentro de una porción de texto recibido por parámetro. Se utiliza para mantener actualizada la posición de lectura del documento.
- **protected int[] getAccentMods(int pos, int offsetPoP)**: la necesidad de tener este método lo provoca el cambio que ejercen los comandos de acentos sobre la posición y longitud de una palabra que las contiene. Contrastando la posición (relativa) calculada mediante los parámetros obtenidos en la llamada con la relación de acentos transformados por el transformer, se deduce si corresponde aplicar alguna modificación sobre la posición o longitud del error encontrado por lo analizadores, devolviendo ambos modificadores en un array.
- **protected int calcOriginalGlobalPosBasic(int firstLineOfParagraph, int charOffsetParagraph)**: este es el método que calcula la posición real de un error gramatical a partir de su posición relativa (tras transformación) y de la posición global

que ocupa la primera línea del fragmento de texto analizado en el documento original. Para calcular esa posición, se obtiene primeramente el offset que generan las transformaciones realizadas entre el primer carácter del texto transformado en la presente iteración y de la posición donde se encontró el error, siempre y cuando pertenezcan también al nivel principal (PoP = 0) del párrafo. Después, solo queda realizar la suma de la posición que ocupa el primer elemento del texto tratado con la posición que los correctores proporcionan, y añadirles el offset generado dentro del párrafo.

Solo sirve para calcular posiciones reales sobre el fragmento de texto principal\padre (PoP = 0, posición 0 de la variable de instancia treatedText de la clase Transformer) en cada iteración, ya que para los demás es necesario el uso de más información (offsets de los PieceOfParagraph que los contienen).

- **protected int calcOriginalGlobalPosAdv(int baseParagraphOffset, int offsetInParagraph, int internalOffset, int charOffsetParagraph, int activePoP)**: el funcionamiento de este método es similar al utilizado por la versión básica del mismo, solo que en este caso buscamos la posición real de un error encontrado dentro de una porción de texto que fue desalojada de su posición original (el por qué se explica más adelante en la subsección 4.5.5 dedicada a los comandos select_Out). El fragmento de texto sustraído ha de ser guardado con una referencia que indica en que parte del texto original estaba situado, y donde comienza el texto analizable dentro del mismo. Por tanto, la posición real se calcula mediante el sumatorio de:
  1. posición que ocupa el comando que generó la porción de texto independiente dentro del texto transformado.
  2. El offset generado por las transformaciones entre la posición que ocupa el primer carácter del texto en proceso de transformación y 1.
  3. La posición en la que comienza el texto analizable dentro del comando a partir del propio comando.
  4. La posición del error obtenida desde el corrector (que se calcula a partir de 3)
  5. El offset de las transformaciones con posiciones entre 3 y 4.

  Para calcular 2 se utilizarán las transformaciones que pertenecían al fragmento de texto externo (PoP = 0), y para 5 aquellas que compartan el mismo PoP que los errores.

- **protected int calcGlobalOffsetLine(int line)**: este método es la base del posicionamiento, y ayudado de la lista con el número de caracteres por línea devuelve la posición que ocupa el primer elemento de una determinada línea en el texto original.
- **public void processDocument(String document) throws IOException, SAXException, ParserConfigurationException**: este método es el gestor de todo el proceso llevado a cabo por la aplicación.

En primer lugar crea y ejecuta el parser que obtendrá la información sobre sustituciones LaTeX desde el archivo XML que las contiene, almacenando dicha información dentro de una instancia de Transformer que llevará a cabo la transformación del documento LaTeX a texto plano. Una vez realizado este paso, toma porciones de texto balanceadas del documento original para que sean transformadas y analizadas, rellenando la lista de grammarErrors con los errores encontrados. Este proceso se inicia tras encontrar el comando "\begin{document}" en el documento, que da comienzo al texto analizable.

4.4.5 GRAMMAR ERROR

Archivo: GrammarError.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s): DocumentSegment

Subclase(s): No

Interface(s): Comparable

Descripción: Clase que almacena la información sobre los errores gramaticales encontrados por un determinado corrector gramatical. La información que muestra la interfaz gráfica es la que almacenan las instancias de esta clase.

Atributos:

- **# int position**: almacena la posición en la que comienza el error detectado por un corrector gramatical. Es un atributo heredado.
- **# int length**: almacena la longitud del error detectado por un corrector gramatical. Es un atributo heredado.
- **- String error**: se guarda aquí la información acerca del tipo de fallo encontrado del que proveen al usuario los correctores gramaticales.
- **- List<String> suggestions**: cada posición de esta lista será rellenada con las sugerencias de corrección que aporten los correctores gramaticales.

Métodos:

- **public int compareTo(GrammarError ge)**: este es el método heredado de la interfaz Comparable que nos permite ordenar una lista de GrammarErrors según su atributo *position*. Se utiliza para poder mostrar en la interfaz gráfica los errores en orden según la precedencia que tengan en el texto.

4.4.6 GRAMMATEX_UI

Archivo: GrammaTeX_UI.java

Paquete: grammaTeX.ui

Tipo de clase: Concreta

Superclase(s) y subclase(s): No

Descripción: esta es la clase que contiene la interfaz de usuario de la aplicación.

Atributos: además de los elementos explicados a continuación, se han utilizado una serie de clases típicas de interfaces SWING en las que no se hará hincapié por su obviedad.

- **- grammaTeXTableModel gErrorTableModel, gSuggestTableModel**: instancias del modelo de tabla creado expresamente para la interfaz. Es una subclase del DefaultTableModel proporcionado por Java que elimina la posibilidad de editar los elementos de las tablas para poder hacer clic sobre ellas sin dicho efecto.
- **- GrammarChecker gc**: clase que gestiona la corrección gramatical.
- **- GrammarError grammarError:** contiene información sobre un error gramatical.
- **- StyledDocument styledDoc:** clase que gestiona todo lo relacionado con el documento visualizado en la interfaz.
- **- Style normal, bold, boldRed, highlight**: estilos creados para formatear el texto del styleDoc.
- **- DocumentSegment highlightedGE:** segmento de documento que guarda la posición y longitud del error activo en la interfaz (subrayado).
- **- Color activeCorrection, nonActiveField, nonActiveScroll**: colores creados para manejar la ayuda visual de los elementos de corrección manual.
- **- AbstractDocument doc**: clase abstracta asociada a la instancia de StyledDocument.
- **- HashMap<Object, Action> actions**: tabla que nos permite utilizar elementos del DefaultEditorKit a través del nombre de las acciones que queremos realizar con ellos.
- **# UndoAction undoAction**: clase creada para controlar la opción *deshacer* del *Edit*.
- **# RedoAction redoAction**: clase creada para controlar la opción *rehacer* del *Edit*.
- **# UndoManager undo**: clase que gestiona las opciones de *deshacer* y *rehacer*.

Métodos:

- **private JScrollPane getErrorTableScroll()**: método que inicializa el scroll pane en el que está situada la tabla de errores. Se le añade un "listener" que escucha el uso del ratón sobre ella.
- **private JTable getErrorTable()**: método que inicializa la tabla de errores. En este método se añade un "listener" a la tabla para que al clickear sobre alguno de los errores que contiene, la tabla de sugerencias muestre aquellas opciones de

corrección asociadas al error seleccionado, además de cambiar el error subrayado en el documento por el actual y realizar un focus sobre él.

- **private JScrollPane getSuggestTableScroll()**: método que inicializa el scroll pane en el que está situada la tabla de sugerencias. Se le añade un "listener" que escucha el uso del ratón sobre ella.
- **private JTable getSuggestTable()**: método que inicializa la tabla de sugerencias. En este método se añade un "listener" a la tabla para que al clickear sobre alguna de las sugerencias que contiene, se guarde una referencia a la fila seleccionada.
- **private JScrollPane getOriginalLatexTextScroll()**: método que inicializa el scroll pane en el que está situado el texto del documento cargado en la interfaz, en cuyo borde se muestra una referencia a la ruta del archivo activo.
- **private JTextPane getOriginalLatexText()**: método que inicializa el recuadro de texto de la interfaz asociándole el StyledDocument, y añadiéndole los estilos creados para este.
- **private JButton getCheckGrammar()**: este método es el encargado de crear el botón que da luz verde a todo el proceso de corrección del documento LaTeX cargado en la interfaz. Se encarga de inicializar el GrammarChecker y ponerlo en funcionamiento, rellenando las tablas de errores y sugerencias con los resultados, y activando el resto de la interfaz.
- **private JButton getCorrectButton()**: método encargado de crear el botón de corrección. Este botón detecta si está activada la corrección automática o manual, y en función de ello aplica el cambio desde la tabla de sugerencias o el campo de texto. Una vez sustituido el texto en el documento, elimina de la lista de errores y de la tabla de errores los elementos asociados al error corregido, y propaga el offset generado por el cambio a los errores que vienen después.
- **private JButton getIgnoreButton()**: método encargado de crear el botón que ignora el error activo. Elimina directamente el error seleccionado de la lista y tabla de errores, activando el siguiente error en la tabla y en el documento.
- **private JTextField getManualCorrectionField()**: inicialización del campo de texto para la corrección manual de errores.
- **private JCheckBox getManualCheckBox()**: inicialización del checkbox que sirve de switch entre la corrección manual y la asistida.
- **private void deactivateAllInterfaceOptions()**: desactiva todos los elementos de la interfaz (excepto documento y botón checkGrammar)**.**
- **private String formatPath(String path)**: devuelve la ruta recibida por parámetro con una longitud susceptible de ser mostrada correctamente en la interfaz.
- **public static void main(String[] args)**: método main.
- **private JFrame getJFrame()**: método que crea la ventana de la aplicación y pone en marcha la creación de todos los demás elementos. Inicia los "listeners" del documento y deshacer/rehacer.

- **private JPanel getJContentPane()**: inicialización del JPanel asociado al JFrame anterior. Contiene todos los elementos a excepción de los menús.
- **private JMenuBar getJJMenuBar(), JMenu getFileMenu(), JMenu getEditMenu()**: puesta en marcha del menú de la aplicación y sus dos grupos de acciones.
- **private JMenuItem getExitMenuItem()**: opción de salir de la aplicación.
- **private JMenuItem getCutMenuItem(), getCopyMenuItem(), getPasteMenuItem()**: creación de las opciones de cortar, copiar y pegar del menú *Edit*. Simplemente se toman del DefaultEditorKit que proporciona Java y se pegan en nuestra aplicación.
- **private JMenuItem getSaveAsMenuItem()**: creación de la opción de menú que permite realizar un "Guardar como". Comprueba si se desea sobreescribir el fichero.
- **private JMenuItem getSaveMenuItem()**: creación de la opción de menú que guarda el documento activo en la ruta desde la cual se cargó.
- **private JMenuItem getOpenMenuItem()**: creación de la opción de menú que nos permite seleccionar y cargar un archivo del sistema, mostrando su contenido en el recuadro de texto y activando la opción de corrección gramatical.
- **private HashMap<Object, Action> createActionTable(JTextComponent textComponent)**: este método es el encargado de la creación y rellenado de la tabla de acciones. Recibe un componente de texto SWING y almacena las acciones asociadas a este en la tabla.
- **private Action getActionByName(String name)**: este método recibe el nombre de una acción como parámetro, y devuelve el elemento de la tabla de acciones con dicho nombre.
- **private int posAtGrammarErrors(int pos, int length)**: este método es utilizado para calcular a partir de qué elemento de la lista de grammarErrors es necesario realizar actualizaciones de la posición en el documento. Suele deberse a algún cambio, inserción, o borrado de texto directamente en la vista del documento de la interfaz.

Clases internas: se han creado una serie de clases dentro de la interfaz gráfica para dar soporte a la escritura en la vista.

- **protected class GrammaTeXUndoEditListener implements UndoableEditListener**: clase que gestiona las opciones de deshacer y rehacer del menú de edición.
  - **public void undoableEditHappened(UndoableEditEvent e)**: se guarda la edición que dispara este método, y se actualizan las listas de ediciones que se pueden deshacer y el rehacer.
- **protected class GrammaTeXDocListener implements DocumentListener**: clase que gestiona los cambios que provocan la escritura, borrado y sustitución de texto en la vista.
  - **public void insertUpdate(DocumentEvent e), removeUpdate(DocumentEvent e), changedUpdate(DocumentEvent e)**: estos tres métodos se disparán cuando se realiza algún cambio en el

documento. En ellos se aplica el offset necesario a los errores gramaticales y al error activo de la vista, ya que el cambio realizado en el documento provoca que las posiciones de estos queden obsoletas.

- **class UndoAction extends AbstractAction**: clase que contiene la acción deshacer del menú de edición.
  - **public UndoAction()**: constructor.
  - **public void actionPerformed(ActionEvent e)**: realiza la primera acción de la lista deshacer, y actualiza su propia lista y la del rehacer.
  - **protected void updateUndoState()**: actualiza la lista de acciones a deshacer y el nombre mostrado en el menú edición.
- **class RedoAction extends AbstractAction**: clase que contiene la acción rehacer del menú de edición.
  - **public RedoAction()**: constructor.
  - **public void actionPerformed(ActionEvent e)**: realiza la primera acción de la lista rehacer, y actualiza su propia lista y la del deshacer.
  - **protected void updateRedoState()**:actualiza la lista de acciones a rehacer y el nombre mostrado en el menú edición.

4.4.7 IO MANAGER

Archivo: IOManager.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s) y subclase(s): No

Descripción: Clase que se encarga del tratamiento de ficheros externos a la aplicación. Permite cargar ficheros y guardarlos desde la interfaz gráfica.

Atributos: No

Métodos:

- **public static String loadFile(File file) throws IOException**: método estático que recibe como parámetro un objeto de la clase File, volcando su contenido a una variable de tipo String que será devuelta. Un fallo al manipular el fichero proporcionado puede provocar una excepción de entrada/salida.
- **public static void write(String fileRoute, String string, boolean append) throws IOException**: método estático que se encarga de volcar una cadena de caracteres recibida por parámetro a un fichero con la ruta de acceso recibida también por parámetro . El parámetro append controla la reescritura del fichero desde el

comienzo o el añadido a partir del final del original. Un fallo al manipular el fichero puede provocar una excepción de entrada/salida.

4.4.8 LANGUAGE TOOL GC

Archivo: LanguageToolGC.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s): GrammarChecker

Subclase(s): No

Descripción: Esta es la clase que realiza la transformación y posterior envío del nuevo texto a un corrector gramatical para recibir los errores que este encuentre y almacenarlos. Esta clase está adaptada para trabajar conjuntamente con el corrector gramatical escogido para el proyecto, el *LanguageTool*. Si se desea utilizar la aplicación con otro corrector, habrá que realizar una clase de este tipo que se ajuste a él.

Atributos:

- **# Transformer transformer**: esta instancia de la clase Transformer heredada del GrammarChecker es la encargada de la transformación del documento LaTeX a texto plano. Guarda toda la información necesaria que utilizan el GrammarChecker y sus subclases para la gestión de los errores gramaticales. Es un atributo heredado.
- **# ArrayList<GrammarError> grammarErrors**: en esta lista se van añadiendo los errores gramaticales encontrados. Es un atributo heredado.

Métodos:

- **private String deleteSuggestionTags(String string)**: elimina los tags <suggestion> del mensaje con información acerca del error que proporciona el *LanguageTool*.
- **protected void checkGrammar(String paragraph, int lineRef) throws IOException**: en primer lugar se manda el texto a analizar al transformer para que este lo transforme en texto plano, y guarde información acerca de los tags encontrados. Una vez terminada la labor del transformer, se inicializa el *LanguageTool* y sus reglas gramaticales, que recibe después el texto plano y lo analiza devolviendo información acerca de los potenciales errores gramaticales encontrados. Finalmente, la fusión de la información recibida desde el *LanguageTool* y la almacenada sobre las transformaciones realizadas en el transformer, nos permite rellenar la lista grammarErrors que contiene toda la información que necesita la interfaz gráfica para asistir al usuario en la corrección de su documento.

4.4.9 LATEX SUBSTITUTION

Archivo: LatexSubstitution.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s) y subclase(s): No

Descripción: esta clase sirve para almacenar la información que el parser de XML obtiene del fichero de sustitución de comandos para la clase transformer.

Atributos:

- **- String type**: este atributo representa los diferentes tipos de tratamiento que pueden recibir los tags de LaTeX por parte de la aplicación. Más adelante (sección 4.5) se explican cuales son, y que tratamiento provocan.
- **- String command**: este atributo almacena el nombre de cada uno de los comandos que trata la aplicación.
- **- String substitution**: este atributo guarda la información sobre el cambio a realizar. Puede indicar directamente el cambio, o servir como parámetro numérico (mediante casting) para la aplicación.
- **- int numArgs**: este atributo indica el número de argumentos entre llaves (“{“ y “}”) que tiene un determinado tag.
- **- String end**: es caso de no poder deducir el final de un entorno a partir de su comienzo, este atributo guarda dicho final.

Métodos: únicamente **getters & setters**.

4.4.10 LATEX TRANSFORMATION

Archivo: LatexTransformation.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s): DocSegmentWithPoP, DocumentSegment (no directa)

Subclase(s): No

Interface(s): Comparable

Descripción: esta clase guarda información acerca de cada uno de los cambios que se van realizando a lo largo del proceso de transformación de los documentos LaTeX a texto plano.

Atributos:

- **# int position**: este atributo guarda la posición relativa en la que fue realizada una transformación. Es un atributo heredado.
- **# int length**: en este caso concreto, la longitud se utiliza para guardar el offset que provoca la transformación, es decir, cuanto (valor) y hacia dónde (signo) se mueven las posiciones relativas posteriores a la actual. Es un atributo heredado.
- **# int PieceOfParagraph**: este atributo indica en que PieceOfParagraph (trozo de texto) se ha realizado la transformación. Es un atributo heredado.
- **- int inOffsetT**: este atributo indica a qué distancia de la "\" que comienza un comando está el texto chequeable que este contiene, de manera que se pueda aplicar solo este offset en caso de que un error sea hallado dentro de dicho texto.
- **- int EffectArea**: este atributo indica la longitud del segmento de texto dentro del cual habrá que aplicar el inOffsetT como desplazamiento en vez del length.
- **- String text**: aquí se guarda el texto que había originariamente en el documento antes de la transformación.
- **- String transformation**: aquí se guarda el texto que ha sustituido al original en el documento, lo que se deja tras la transformación.

Métodos:

- El **constructor** principal de esta clase acepta cinco argumentos, ya que:
  - el atributo *length* se obtiene de restar la longitud de la transformación a la del texto original.
  - El atributo *effectArea* será inicialmente igual a la longitud de la sustitución insertada en el documento. Solo tiene sentido calcularla si el comando transformado contiene texto que servirá como sustitución del mismo, y en el cual los errores deberán utilizar solo el desplazamiento inOffsetT para calcular su posición real.
- **public int compareTo(LatexTransformation lt)**: este es el método heredado de la interfaz Comparable que nos permite ordenar una lista que contiene LatexTransformations en función del valor de su atributo *position*. Se utiliza para mantener organizada la lista de transformaciones sobre la cual se calculan las posiciones reales.

4.4.11 PIECE OF PARAGRAPH

Archivo: PieceOfParagraph.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s) y subclase(s): No

Descripción: una vez balanceado un texto, todavía es posible que requiera de la extracción de una parte del mismo para poder ser analizado, y para poder analizar también esta parte extraída por separado. Esta clase contiene cada una de las porciones de texto que serán analizadas, y guarda información de posicionamiento sobre ellas.

Atributos:

- **- int parOffset**: guarda la posición relativa que ocupaba el primer carácter del texto extraído dentro del párrafo del que fue extraído.
- **- int inOffset**: guarda la distancia que existe entre el primer carácter del comando que provoca la extracción y del primer carácter del texto analizable que contiene ese comando.
- **- String text**: guarda el texto analizable del comando extraído.

Métodos: únicamente **getters & setters**.

4.4.12 SAX_XML PARSER

Archivo: SAX_XMLParser.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s): DefaultHandler

Subclase(s): No

Descripción: esta clase es la encargada de realizar la lectura del archivo XML que contiene las sustituciones de comandos LaTeX y elaborar una lista con la información recabada.

Atributos:

- **- ArrayList<LatexSubstitution> substitutions**: en esta lista será guardada la información acerca de cómo manipular cada comando de LaTeX.
- **- String content**: aquí se guarda la información contenida entre dos tags de un mismo tipo (de inicio y final), para después ser volcado al atributo correspondiente de un LatexSubstitution.
- **- LatexSubstitution currentSub**: almacena toda la información necesaria sobre cada uno de los comandos obtenidos del archivo XML.

Métodos:

- **public void parseXML() throws SAXException, ParserConfigurationException, IOException**: se encarga de la inicialización del parser SAX, de su puesta en funcionamiento sobre el archivo de sustituciones,y del registro de la clase como destino de los "callbacks" del parser (eventos generados por el parser durante la lectura del fichero cuando analiza los tags).
- **private void printData()**: simplemente muestra en pantalla la información obtenida desde el fichero al final de la lectura. Apoyo para la implementación.
- **public void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException**: su contenido se ejecuta cuando se recibe un evento relacionado con la lectura de un tag de apertura en el fichero. En nuestro caso solo nos interesa realizar acciones si lo que se ha leído es el comienzo de una sustitución (no la transformación, sino el tratamiento completo), en cuyo caso se crea una nueva instancia de LatexSubstitution para albergar la información que se irá leyendo, y se busca y almacena el tipo de sustitución que se va a leer.
- **public void characters(char[] ch, int start, int length) throws SAXException**: rellena el atributo content con la información contenida entre dos tags de un mismo tipo (de inicio y final).
- **public void endElement(String uri, String localName, String qName) throws SAXException**: su contenido se ejecuta cuando se recibe un evento relacionado con la lectura de un tag de cierre en el fichero. Aquí se completa la información del LatexSubstitution con los tags que aparezcan, y en caso de leer el tag de cierre de la sustitución, se añade la información obtenida a la lista de sustituciones.

4.4.13 TRANSFORMER

Archivo: Transformer.java

Paquete: grammaTeX.project

Tipo de clase: Concreta

Superclase(s) y subclase(s): No

Descripción: esta clase es la que alberga la mayor parte de la funcionalidad de la aplicación, y se encarga de llevar a cabo todo el proceso de transformación de documentos LaTeX a texto plano en un formato procesable por los correctores gramaticales. Toma una porción del documento LaTeX (recibido desde el GrammarChecker), y va realizando transformaciones de manera iterativa hasta conseguir obtener una lista de porciones de texto analizables y la información necesaria para que se pueda recolocar cualquier posible error en el texto, tal y como este fue recibido.

Atributos:

- **- ArrayList<LatexTransformation> transfs**: es la lista en la que se va guardando información sobre las transformaciones realizadas.
- **- ArrayList<LatexSubstitution> substitutions**: es la lista en la que está almacenada la información sobre las sustituciones contempladas por la aplicación, y es proporcionada por el GrammarChecker antes de comenzar el proceso de transformación.
- **- ArrayList<PieceOfParagraph> treatedText**: es una lista que contiene los strings que van siendo transformados y que serán posteriormente analizados por el corrector. Inicialmente solo contiene su primera posición rellena con el texto recibido desde el GrammarChecker, pero a lo largo de la transformación puede ser que se le realicen a este extracciones que serán almacenadas en índices posteriores de la lista, los cuales también serán analizados una vez terminemos con el primero.
- **- ArrayList<DocSegmentWithPoP> transfForbidden**: contiene referencias a los fragmentos de texto dentro de los cuales se le prohíbe a la aplicación buscar y utilizar elementos que formen parte de una transformación, como literales o elementos ya transformados.
- **- ArrayList<AccentContSegment> accentControl**: los acentos alteran la posición y longitud de las palabras que pueden verse implicadas en un error gramatical, y por tanto, es necesario guardar en esta lista referencias a la posición de las palabras que los contienen.
- **- LatexTransformation lt**: sirve para guardar la información de la transformación que se está llevando a cabo antes de añadirla a la lista de transformaciones.
- **- int transfsInsertPos**: al necesitar el modo de funcionamiento de la aplicación que las transformaciones se mantengan ordenadas por posición en todo momento, la inserción dentro de la lista se realiza ordenadamente en la posición marcada por este atributo.
- **- int baseGlobalOffset**: cada vez que el transformer es puesto en funcionamiento con una determinada porción del documento LaTeX, esta variable se actualiza guardando en ella la posición global que ocupa su primer elemento dentro del original. Las posiciones de los demás elementos se calculan a partir de él.

Métodos:

- **public void treatLatexParagraph(String paragraph, int baseGlobalOffset)**: este método es el encargado de gestionar la transformación de una porción de documento LaTeX recibida como parámetro a una secuencia de cadenas de caracteres con texto plano que serán almacenadas en una variable de instancia propia a la que podrá acceder el GrammarChecker.

  El método comienza vaciando las listas de control y las transformaciones, para que estas solo almacenen los elementos que pertenecen a la iteración actual, agilizando y simplificando su uso. Después, se analizan los sucesivos índices de la variable

treatedText (teniendo en cuenta que la transformación de uno de los índices puede conllevar sustracciones de texto que se almacenan en posiciones posteriores) de uno en uno mientras se encuentre algún elemento en ellos que pueda ser transformado. Si en el texto del índice en proceso de transformación se encuentra algún elemento susceptible de formar parte de un comando, se válida si la posición forma parte de un segmento prohibido, y si no es así, se aplica la transformación pertinente (comandos, fórmulas matemáticas, o comentarios).

- **private int getPosToTransform(String paragraph, int pos, int activePoP)**: este método se encarga de buscar la posición donde comienza el siguiente comando LaTeX que debe ser transformado. Para ello, se buscan en el texto recibido por parámetro las primeras posiciones válidas (fuera de los segmentos prohibidos) de cada uno de los símbolos con los que comienzan los comandos ("\" para los normales, "$" para los matemáticos, y "%" para los comentarios). Después, si se encuentran posiciones válidas para alguno de los diferentes símbolos, estas son agregadas a un conjunto cuyo mínimo será devuelto por la función, pues al tener precedencia en el texto respecto de las demás, será la próxima que deba ser transformada. Si no se encuentra ninguna posición válida para alguno de los símbolos, se devuelve -1.
- **private String obtainLatexCommand(String string, int posIni):** este método recibe una cadena de caracteres y una posición dentro de la misma en la cual se ha encontrado el comienzo de un comando de LaTeX común (por común se entienden aquellos comandos que comienzan con una "\"), y busca el final del mismo devolviendo una cadena con el comando completo.
- **private String transformLatexIntoPlain(String string, int activePoP, String command, int pos)**: existen dos métodos de transformación de comandos de LaTeX que comienzan por "\", los simples y los entornos con begin/end. Este método simplemente direcciona el flujo del programa por uno u otro según el comando recibido por parámetro.
- **private String transformNotBE(String string, int activePoP, String command, int pos)**: este método se encarga de la transformación de comandos simples de LaTeX, que son aquellos que comienzan por "\", seguidos o por un único carácter distinto de una letra, o por una serie de caracteres letra. Busca en la información de la lista con las sustituciones el comando obtenido, y aplica el tratamiento correspondiente al tipo de comando en el texto en función de dicha información. Si no encuentra información acerca del comando lo elimina, ya que es más probable que los correctores gramaticales puedan reconocer fallos en la frase de esta manera. El procedimiento termina con la devolución de la cadena de caracteres transformada. El tratamiento realizado a cada comando se expone en la siguiente sección (4.5) de este mismo apartado.

- **private String transformBE(String string, int activePoP, String command, int pos)**: este método funciona de la misma manera que el anterior, pero con la salvedad de que transforma entornos begin/end en vez de comandos simples. En este caso el comando recibido por parámetro siempre será el mismo, un "\begin", por lo que primeramente habrá que obtener el argumento que acompaña al mismo para realizar la búsqueda en la lista de sustituciones. Posteriormente se realiza el tratamiento y devolución de la cadena transformada.
- **private String fixSimpleMath(String string, int pos, int activePoP)**: este método es análogo a los dos anteriores, solo que se encarga de la transformación de fórmulas matemáticas delimitadas por símbolos "$", y no necesita de ninguna información del archivo XML, pues las sustituciones son siempre las mismas. Los demás tipos de fórmulas matemáticas son transformadas como comandos normales, pues comienzan también por el símbolo "\". Por lo demás, funciona de la misma manera, busca el final de la fórmula (si no se encuentra se detiene el proceso de transformación), se sustituye la fórmula encontrada en el texto y se devuelve la cadena sin la fórmula.
- **private String fixComment(String string, int pos, int activePoP)**: este método es el encargado de tratar los comentarios LaTeX. Recibe un fragmento de texto y una posición dentro de él en la que hay un símbolo "%" válido para ser transformado. Se extrae a partir de él el texto que resta en la línea y los espacios que comienzan la siguiente línea, tal y como hace el propio LaTeX. Con el texto extraído se forma un nuevo PieceOfParagraph que será almacenado para un análisis gramatical posterior e independiente. El fragmento de texto original es transformado y devuelto.
- **private String transformationUpdates(String string, int pos, String command, String transf, int activePoP, int inOffsetT)**: este es el método responsable de llevar a cabo la transformación de un comando en el texto y de almacenar la información asociada a dicha transformación.
- **private LatexSubstitution obtainSubstitutionInfo(String command)**: este método es el encargado de proporcionar a los métodos de transformación (los que utilizan transformaciones basadas en el archivo XML) la información que necesitan acerca de la manera en que debe de ser sustituido un comando recibido por parámetro.
- **private int searchEndOfEnvironment(String paragraph, String kind, int pos, boolean getEndingPos)**: este método recibe una cadena de caracteres, y una posición dentro de la misma donde comienza un entorno de tipo *kind* cuyo final se debe devolver. En primer lugar, se busca en el texto recibido el comando de cierre asociado a este entorno, devolviendo después la posición sobre la cual este comando tiene su primer o último carácter, en función del valor de la variable booleana *getEndingPos* recibida por parámetro que sirve a modo de selector entre ambas opciones. Si no se encuentra el comando de cierre, se devuelve la posición recibida por parámetro.
- **protected ArrayList<String> argumentTreatment(String string, int pos, int numArgs)**: este método sirve para obtener los argumentos de un comando. Recibe

un string con el comando y sus argumentos, la posición donde comienza este, y el número de argumentos que se han de buscar. El método devuelve una lista de cadenas de caracteres donde la primera posición contendrá el comando conjuntamente con sus argumentos tal y como pueden ser encontrados en el texto. Las demás posiciones almacenan cada uno de los argumentos por separado (lo contenido entre llaves).

- **private void getWordWithAccent(String string, int pos, int comLength, int activePoP)**: este método es el encargado de mantener el control de acentos del transformer. Cuando se encuentra un comando de acentuación, este método busca el comienzo y final de la palabra que lo contiene (para saber cuándo aplicar el control a los errores), y se calculan los modificadores de posición y longitud que se necesitarán más tarde en caso de que exista un error en la palabra.
- **protected ArrayList<DocumentSegment> verbatimTextControl(String textPortion)**: este método sirve al GrammarChecker para elaborar una lista de segmentos prohibidos (comandos verb y verb*, y entorno verbatim) que después será utilizada para contrastar la validez de los comandos "\begin" y "\end" que se utilizan para comprobar el balance de las porciones de texto enviadas al transformer.
- **private int findVerbatimEnvBegin(String textPortion, int startAt)**: este método busca dentro de una cadena de caracteres y a partir de una posición recibidas por parámetro, las posiciones donde comienzan los comandos "\begin" asociados a entornos *verbatim*.
- **private int findVerbComEnd(String textPortion, int startAt)**: este método devuelve la posición del delimitador de final de un comando "\verb" o "\verb*" a partir de una cadena de caracteres y la posición en la que comienza dicho comando.
- **protected boolean checkPosIsVerbatim(ArrayList<DocumentSegment> verbatimSegments, int pos)**: este método es utilizado por las instancias derivadas de la clase abstracta GrammarChecker para decidir si un comando "\begin" o "\end" se puede utilizar en la comprobación de si una porción de texto está balanceada, y así enviarla al transformer. Para ello, comprueba si una determinada posición recibida como parámetro se encuentra dentro de una lista con segmentos de texto también recibida por parámetro.
- **private boolean checkPosIsTransForbidden(int pos, int activePoP)**: este método es similar al anterior, solo que en este caso la lista en vez de ser recibida por parámetro es la lista de segmentos prohibidos del propio transformer. La diferencia estriba en que al contrario que en el método anterior, aquí los segmentos prohibidos se van almacenando durante la transformación, y por tanto es necesario controlar a que PieceOfParagraph pertenecen. Esto se debe a que dos segmentos prohibidos pueden pertenecer al texto base y a una porción de texto extraído del mismo, y a causa de las transformaciones presentar posiciones cercanas, incluso la misma. No obstante si nos encontramos en el texto extraído, obviamente solo el segmento prohibido que

pertenece a este debería de ser tenido en cuenta para tomar posiciones válidas de cara a la transformación.

- **private int getTransfsInsertPos(int pos)**: devuelve la posición dentro de la lista de transformaciones en la cual debería de ser insertada la actual en función de la posición que ocupa en el texto.
- **private void updateEffectAreas()**: este método es el encargado de ir actualizando las dimensiones de los segmentos sobre los cuales hay que aplicar offsets parciales. Es llamado cada vez que se realiza una transformación en el documento, ya que si dicha transformación ha sido realizada dentro del área de efecto de otra transformación con esta propiedad, habrá que redimensionarla para poder calcular correctamente la posición real de los errores que se encuentren en su interior.

4.5 XML DE SUSTITUCIONES LaTeX

A continuación se explican los diferentes tipos de comandos LaTeX que es capaz de tratar la aplicación. Para cada grupo se mostrará el formato que tienen los comandos que pertenecen a él, una explicación de cómo se desarrolla el proceso de sustitución, y una tabla que contiene todos los comandos que trata la aplicación pertenecientes a dicho grupo. Es necesario hacer mención aquí de que hay tres comandos que la aplicación es capaz de tratar y que no están presentes en esta sección: las fórmulas matemáticas delimitadas por uno y dos símbolos "$", y los comentarios que comienzan con el símbolo "%". Esto se debe a que dichos comandos no presentan el formato habitual, y por tanto, su tratamiento se realiza en dos métodos propios que contienen toda la información necesaria para su sustitución (fixSimpleMath y fixComment).

Se establecen al comienzo de los métodos de transformación (transformNotBE, transformBE, fixSimpleMath y fixComment) unos valores por defecto para algunos de los atributos que tendrán los LatexTransformation. Estos serán alterados por el tratamiento propio de cada grupo solo en caso de que sea necesario. Por ello, solo se mencionarán los valores que toman estos atributos si se realiza algún cambio sobre ellos, obviándolos si no guardan relación con el tratamiento llevado a cabo para un determinado tipo de comandos.

4.5.1 SUSTITUCIÓN DIRECTA

4.5.1.1 FORMATO XML

```
<Substitution type="direct">
    <Command>\comando</Command>
    <Change>sustitución</Change>
</Substitution>
```

4.5.1.2 TRATAMIENTO

Los comandos que pertenecen a este grupo son los más simples (en formato) que tiene LaTeX. Comienzan con un símbolo "\", seguido de un único carácter no letra, o varios caracteres letra.

Estos comandos son también los más sencillos de transformar, pues solo es necesario sustituir el comando encontrado por la cadena de caracteres del campo "Change" asociado a estos en el archivo XML. En los casos en los que el tratamiento del comando implica borrarlo sin sustituirlo por nada, el XML tiene el campo "Change" vacío (en la tabla siguiente, para estos casos se muestra la palabra "Eliminado" en el campo Sustitución alineada a la derecha).

Lo único que es necesario reseñar dentro de los comandos de este tipo es que en caso de encontrarnos con los comandos "\$" o "\%", se crea un segmento prohibido asociado a la transformación, ya que las transformaciones de estos comandos ("$" y "%") son símbolos utilizados para detectar comandos, y hay que evitar que se tomen como válidos.

4.5.1.3 TAGS

Comando LaTeX	Sustitución
\$	$
\&	&
\%	%
\#	#
_	_
\{	{
\}	}
\ldots	…
\maketitle	Eliminado
\tableofcontents	Eliminado
\listoffigures	Eliminado
\listoftables	Eliminado
\today	Sunday
\makeindex	Eliminado
\makeglossary	Eliminado
\item	Eliminado
\upshape	Eliminado
\itshape	Eliminado
\slshape	Eliminado
\scshape	Eliminado
\mdseries	Eliminado
\bfseries	Eliminado
\rmfamily	Eliminado
\sffamily	Eliminado
\ttfamily	Eliminado

4.5.2 COMANDO “VERB”

4.5.2.1 FORMATO XML

```
<Substitution type=“verb”>
      <Command>\verb</Command>
</Substitution>
```

4.5.2.2 TRATAMIENTO

Este tipo sirve para tratar dos comandos: “\verb” y “\verb*”. En ambos casos, el comando va seguido (sin espacios) de un carácter no letra que dará inicio al entorno que contiene el texto, el cual acaba cuando se encuentre de nuevo ese mismo carácter no letra. Obviamente, habrá de escogerse un delimitador que no aparezca dentro del texto del entorno.

El transformador detectará en ambos casos solo la parte común, y la pequeña diferencia que tienen sus tratamientos se realiza directamente en el código. Apoyado en la función *findVerbComEnd*, el transformador detecta la posición del delimitador de final del comando, se toma el comando entero, y se sustituye por el texto que hay entre los delimitadores. La diferencia entre los dos comandos es que hay que controlar que el transformador no tome un comando “\verb*” como un comando “\verb” con delimitador ‘*’.

Al ser un comando cuya sustitución está formada por el texto que contiene el propio comando, es necesario guardar la distancia que existe entre el comienzo del comando y el texto chequeable, para que si existe un error dentro de este, se pueda aplicar solo esta distancia como desplazamiento en vez del generado por la transformación completa.

La funcionalidad de estos comandos (y el entorno *verbatim* que se verá más adelante) es la de asegurar que el texto que contienen es escrito en el documento final (compilado) tal y como aparece en ellos. Esto significa que aunque contengan palabras reservadas y comandos LaTeX, estos no han de ser transformados ni tenidos en cuenta. Por lo tanto, cada vez que se realice una transformación de este tipo, se añade un nuevo DocSegmentWithPoP (posición, longitud, PieceOfParagraph) relacionado con la transformación a la lista de segmentos prohibidos.

4.5.2.3 TAGS

Comando LaTeX	Sustitución
\verb	Se obtiene en tiempo de ejecución con el texto entre los delimitadores
\verb*	

4.5.3 SUSTITUCIÓN DIRECTA DE ENTORNOS

4.5.3.1 FORMATO

```
<Substitution type="directEnv">
        <Command>\comando</Command>
        <End>\comando</End>
        <Change>Sustitución</Change>
</Substitution>
```

4.5.3.2 TRATAMIENTO

Pertenecen a este grupo los comandos que definiendo un entorno, no utilizan para ello las palabras reservadas "\begin" y "\end". Para ello constan de dos tag simples (uno de comienzo y otro de final) que deben ser únicos.

La sustitución se realiza sobre el entorno completo, es decir, una vez encontrado el tag de inicio, se busca en el archivo XML que tag de final le corresponde y se sustituye toda la longitud del comando por el contenido del atributo "Change" de dicho comando.

4.5.3.3 TAGS

Tag de inicio	Tag de cierre	Sustitución
\(	\)	MathFormula
\[	\]	MathFormula

4.5.4 SELECCIÓN DE ARGUMENTOS CON TRATAMIENTO INTERNO

4.5.4.1 FORMATO

```
<Substitution type="select_In">
      <Command>\comando</Command>
      <Args>nº de argumentos</Args>
      <Change>argumento seleccionado</Change>
</Substitution>
```

4.5.4.2 TRATAMIENTO

En este grupo se encuentran los comandos que están formados por un comando simple acompañado de uno o varios argumentos. Cada uno de los argumentos va separado entre llaves ("{" y "}"), y puede haber espacios entre comando y argumentos, y entre cada par de estos últimos (existe alguna excepción, como por ejemplo el comando de cierre del entorno *verbatim*).

La sustitución que se realiza sobre estos comandos está contenida dentro del propio comando, ya que el cambio se realiza seleccionando para ello alguno de sus argumentos, siempre sin realizar transformación alguna. El tratamiento comienza obteniéndose el comando completo (comando + argumentos) y cada uno de los argumentos por separado mediante la función *argumentTreatment* de la clase Transformer. La función obtiene el número de argumentos a buscar del archivo XML. La selección de uno u otro argumento se realiza también con información del XML, donde el campo "change" almacenará en esta ocasión un número (se realiza un cast dentro de la aplicación). Los comandos que deben ser eliminados por completo contendrán un 0 en este campo.

La sustitución de este comando está formada por el texto que contiene alguno de los argumentos del propio comando, y por tanto será necesario guardar la distancia que existe entre el comienzo del comando y el texto del argumento seleccionado para la sustitución. Esto nos permitirá aplicar solo esta distancia como desplazamiento en vez del generado por la transformación completa si existe un error en el texto del argumento.

4.5.4.3 TAGS

Comando LaTeX	Nº de argumentos	Argumento seleccionado
\author{…}	1	0
\date{…}	1	0

\bibliography{…}	1	0
\bibliographystyle{…}	1	0
\nocite{…}	1	0
\index{…}	1	0
\glossary{…}	1	0
\label{…}	1	0
\end{…}	1	0
\mbox{…}	1	1
\part{…}	1	1
\chapter{…}	1	1
\section{…}	1	1
\subsection{…}	1	1
\subsubsection{…}	1	1
\paragraph{…}	1	1
\subparagraph{…}	1	1
\emph{…}	1	1
\underline{…}	1	1
\title{…}	1	1
\textup{…}	1	1
\textit{…}	1	1
\textsl{…}	1	1
\textsc{…}	1	1
\textmd{…}	1	1
\textbf{…}	1	1
\textrm{…}	1	1
\textsf{…}	1	1
\texttt{…}	1	1

4.5.5 SELECCIÓN DE ARGUMENTOS CON TRATAMIENTO EXTERNO

4.5.5.1 FORMATO

```
<Substitution type="select_Out">
        <Command>\comando</Command>
        <Args>nº de argumentos</Args>
        <Change>argumento seleccionado</Change>
</Substitution>
```

4.5.5.2 TRATAMIENTO

El formato de estos comandos es exactamente igual al de los de selección interna, ya que están formados por un comando simple acompañado de uno o varios argumentos. Los argumentos también van entre llaves y puede haber espacios entre comando y argumentos,

y entre los propios argumentos. La diferencia estriba en que mientras que un comando de tipo select_In contiene entre sus argumentos texto que da sentido a la frase en la que es encontrado, los comandos de tipo select_Out deben de ser extraídos para que la frase que los contiene pueda ser analizada. Además, estos comandos suelen tener sentido por sí mismos, por lo que su gramática puede ser analizada por separado.

El tratamiento que reciben estos comandos comienza obteniendo el comando completo y sus argumentos mediante el método argumentTreatment del Transformer y la información sobre el número de argumentos existente en el XML, sustrayéndose después el comando completo del párrafo original. Se extrae después del comando el texto analizable con información también contenida en el XML, el cual es guardado conjuntamente con la posición de extracción del texto original y el offset existente entre el comienzo del comando y dicho texto, formando un nuevo PieceOfParagraph que será analizado y transformado en nuevas iteraciones del transformer. Este grupo no contiene comandos que son eliminados, ya que en tal caso se etiquetan como select_In con campo "Change" a 0 (comparten formato).

4.5.5.3 TAGS

Comando LaTeX	Nº de argumentos	Argumento seleccionado
\footnote{…}	1	1

4.5.6 ACENTOS

4.5.6.1 FORMATO

```
<Substitution type="accent">
    <Command>\comando</Command>
</Substitution>
```

4.5.6.2 TRATAMIENTO

Este grupo lo forman los 14 comandos que tiene LaTeX para el tratamiento de acentos. Los comandos son simples, formados por una "\" más un único carácter seguidos de un único argumento.

No es necesario guardar ninguna información en el archivo XML además del nombre, ya que el tratamiento es siempre el mismo. Esto se debe a que de momento la aplicación ha sido preparada única y exclusivamente para el inglés, que carece de acentos, por lo que el

tratamiento consistirá en sustituir el comando por la letra a la cual se le pretendía adherir un acento. El argumento del comando puede contener una o dos letras iguales, y en el caso de que la letra sea una i o una j aparecerán como "\i" y "\j", ya que deben perder el punto antes de recibir un acento. En cualquiera de los casos la sustitución puede obtenerse tomando el último carácter que contenga el argumento.

La peculiaridad de estos comandos radica en que son los únicos que forman parte de las palabras, y por tanto generan errores de posicionamiento (si una letra con acento comienza una palabra) y longitud de subrayado (acortan la palabra) en los errores lanzados por el corrector gramatical que son mostrados en la interfaz. Para prevenir este hecho, se analiza la palabra que contiene el acento y hallan sus límites a izquierda y derecha del comando, información que nos permite crear un AccentContSegment que guardará la posición del comienzo de la palabra, su longitud, y modificadores de posición y longitud según el caso.

4.5.6.3 TAGS

Comandos LaTeX	
\`{letra}	\u{letra}
\´{letra}	\v{letra}
\^{letra}	\H{letra}
\"{letra}	\t{letra}
\~{letra}	\c{letra}
\={letra}	\d{letra}
\.{letra}	\b{letra}

4.5.7 SUSTITUCIÓN DIRECTA DE ENTORNOS BEGIN-END

4.5.7.1 FORMATO

```
<Substitution type="directEnvBE">
    <Command>comando</Command>
    <Change>sustitución</Change>
</Substitution>
```

4.5.7.2 TRATAMIENTO

Este grupo contiene entornos en vez de comandos. Los entornos habituales en LaTeX se construyen con una pareja de comandos "\begin{}" y "\end{}" cuyos argumentos son el nombre del entorno. Su efecto se reduce al espacio contenido entre ambas expresiones.

En este caso concreto el tratamiento es muy sencillo, ya que son entornos que pueden ser sustituidos directamente por una determinada cadena de caracteres que permita el análisis de la frase origen. Primeramente se obtiene el nombre del entorno (que en el XML se guarda en el campo "Command" pero evidentemente sin la "\"), para después y mediante la utilización de funciones de apoyo que contiene el Transformer, conseguir la posición en la que se termina el comando "\end{}" asociado a un determinado comando "\begin{}". Una vez obtenida esta posición, se sustituye todo lo contenido entre ambos límites por la sustitución que dicte el archivo XML.

4.5.7.3 ENTORNOS

Nombre del entorno	Sustitución
equation	MathEquation
displaymath	DisplayMath
math	MathFormula
tabular	Eliminado
thebibliography	Eliminado
theindex	Eliminado

4.5.8 ELEMENTOS FLOTANTES

4.5.8.1 FORMATO

```
<Substitution type="floatingElems">
    <Command>comando</Command>
    <Change>sustitución</Change>
</Substitution>
```

4.5.8.2 TRATAMIENTO

Este grupo contiene dos entornos específicos de tipo begin/end, *figure* y *table*. Solo se diferencian de los demás entornos de este tipo en el hecho de que aunque la mayoría de la información que albergan puede ser eliminada directamente (pues no son construcciones gramaticales), parte de ella sí que es susceptible de ser analizada por un corrector.

El tratamiento es similar a los demás comandos de este tipo, ya que es necesario buscar la posición en la que se terminan los entornos para sustituir el comando completo. En este caso no se realiza ninguna sustitución, simplemente se elimina el entorno de la porción de texto original en la que se encontraba. Una vez eliminado el comando, se buscan dentro de

él los comandos "\caption{}" (puede haber varios) cuyo argumento es el texto analizable que buscábamos. Calcularemos el offset que tiene el comando dentro del texto original, y el offset que hay entre esta posición y el texto analizable, para conjuntamente con el propio texto crear un nuevo PieceOfParagraph que se añade a la cola de porciones de texto que deben de ser transformadas y analizadas. En el formato de estos comandos en el archivo XML se ha dejado el campo "Change" por si en algún nuevo caso que pueda ser integrado dentro de este tratamiento fuese necesario utilizarlo.

4.5.8.3 ENTORNOS

Nombre del entorno	Sustitución
figure	**Eliminado, genera nuevos PieceOfParagraph**
table	**Eliminado, genera nuevos PieceOfParagraph**

4.5.9 ENTORNO "VERBATIM"

4.5.9.1 FORMATO XML

```
<Substitution type="verbatimBE">
      <Command>comando</Command>
</Substitution>
```

4.5.9.2 TRATAMIENTO

Este comando tiene el formato de un entorno begin/end común con nombre "verbatim", solo que el tratamiento que recibe es similar al del comando "\verb". No han podido incluirse dentro del mismo tipo debido a que los métodos que transforman comandos normales y entornos begin/end están separados dentro de la aplicación.

Para tratar este tipo de entorno, se determina primero donde acaba y se extrae completo del texto. Después, una vez aplicado el método *argumentTreatment* sobre el tag de comienzo, podemos ya extraer el texto que contiene el entorno, pues sabremos donde termina el comando "\begin", y donde comienza el comando "\end" (este caso es el único en el que no puede haber espacios entre el end y su argumento), insertándolo después donde estaba el comando original.

Igual que en el caso de los comandos "\verb" y "\verb*", el texto que contiene este entorno no puede ser usado para transformaciones, debe ser conservado literalmente. Por tanto, se añade un segmento prohibido con la información de posición, longitud, y PieceOfParagraph

en el que fue realizada la transformación. El hecho de que la sustitución esté formada por texto que contiene el propio entorno provoca también en este caso que se guarde una referencia del desplazamiento que provoca el comando de comienzo de entorno. Este será el desplazamiento aplicado a los errores contenidos dentro del texto del entorno, y no el completo.

4.5.9.3 ENTORNOS

Nombre del entorno	Sustitución
verbatim	Texto que contiene el entorno

4.5.10 ENTORNOS CON BORRADO DE TAGS DE INICIO Y FINAL

4.5.10.1 FORMATO XML

```
<Substitution type="deleteTags">
    <Command>comando</Command>
</Substitution>
```

4.5.10.2 TRATAMIENTO

Este grupo está formado por los entornos begin/end en los que la sustitución se realiza con el texto contenido en los propios entornos, y por lo tanto solo será necesario eliminar del texto original los tags de comienzo y final.

Para tratar este tipo de entorno, se determina primero donde acaba este, información con la que conoceremos ya los límites del mismo a ambos lados, y por ende, el segmento de texto a sustituir. Se aplica entonces el método *argumentTreatment* sobre el tag de inicio de entorno, obteniéndose con la longitud de este el offset al cual comienza el texto que sustituirá al original. Para obtener la posición en la que finaliza el texto de sustitución, se utiliza la función searchEndOfEnvironment, que mediante un flag nos permitirá obtener la posición en la que comienza el tag de final de entorno. Una vez aquí, ya conocemos el texto a sustituir y la sustitución, por lo que solo queda ejecutar el cambio.

El hecho de que la sustitución esté formada por texto que contiene el propio entorno, genera también en este caso la necesidad de guardar una referencia del desplazamiento que provoca el comando de comienzo de entorno. Este será el desplazamiento aplicado a los errores contenidos en el texto de estos entornos.

4.5.10.3 ENTORNOS

Nombre del entorno	Sustitución
center	Texto que contiene el entorno
itemize	Texto que contiene el entorno
enumerate	Texto que contiene el entorno
description	Texto que contiene el entorno
quote	Texto que contiene el entorno
quotation	Texto que contiene el entorno
tabbing	Texto que contiene el entorno

4.6 PRUEBAS

Para llevar a cabo las pruebas, se han creado una serie de documentos LaTeX de reducido tamaño que comprueban cada una de las diferentes funciones implementadas dentro del sistema. Los segmentos de código que se han utilizado han sido exclusivamente desarrollados para llevar a cabo las pruebas, y por tanto puede ser que no tengan demasiado sentido en un documento LaTeX real.

Evidentemente solo se expondrán aquí aquellas que prueban funciones de alto nivel de la aplicación, ya que su correcto funcionamiento demuestra también el funcionamiento de elementos y métodos más simples utilizados por estos. Se muestran a continuación dichos documentos de manera individual con comentarios acerca de qué y cómo ha sido probado. En algunos casos, además del texto original se muestra como queda este tras pasar por el transformador (texto recogido desde la aplicación mediante system.out.println), para saber así que es lo que ha recibido el corrector gramatical.

Prueba nº 1: Funcionamiento del parser XML
Descripción: La prueba consiste en ver si el parser SAX que utiliza la aplicación lee y almacena correctamente la información sobre las sustituciones LaTeX que pueden encontrarse en el archivo XML.
Resultado: La prueba se ha superado satisfactoriamente. Se realizo una impresión en fichero del contenido del ArrayList substitutions para ver que todos y cada uno de los comandos aparecían en él con los atributos correctos, y efectivamente así lo era. Además, la impresión se ha realizado desde la clase Transformer, lo cual significa que la información además de ser correcta, llega a su destino.

Prueba nº 2: Funcionamiento de la interfaz
Descripción: La prueba consiste en comprobar que los controles de la interfaz funcionan correctamente.
Resultado: La prueba se ha llevado a cabo con resultados positivos, ya que todos los elementos de la interfaz funcionan correctamente. Se realiza también una captura y tratamiento de las excepciones correctamente. No se muestran aquí ni las pruebas ni los resultados por limitaciones de espacio, y por ser pruebas muy simples.

Prueba nº 3: Comprobación del IOManager
Descripción: El IOManager es la clase de Java que se ha creado tanto para cargar en la interfaz el contenido de un documento LaTeX, como para guardar si se desea el documento de la vista en un archivo.
Resultado: La prueba se ha resuelto correctamente, ya que al cargar un archivo LaTeX de prueba la aplicación a mantenido el formato exacto del fichero, y al guardarlo ha ocurrido lo mismo.

Prueba nº 4: Comprobación de la selección de segmentos de texto para el análisis
Descripción: Se comprobará si la función getCheckableTextPortion del GrammarChecker funciona adecuadamente, ya que la transformación de entornos depende directamente del funcionamiento de ella.
Documento LaTeX original:

```
\documentstyle[11pt]{article}
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

\begin{enumerate}
\item
This paragraph has +1 balance.

Items in a list can contain multiple paragraphs.
  \begin{itemize}
  \item This is an item from an itemize environment
    \begin{description}
    \item[A] description item.
    \item[This is another] description item.
    \end{description}
This paragraph has +1 balance.

  \item This is another item from an itemize environment.
    \begin{itemize}
    \item Itemize element.
   \end{itemize}
  \end{itemize}
\item \verb+\end{...}+ \begin{verbatim}\end{...} \end{verbatim}
\end{enumerate}
This paragraph has -2 balance. This one balances the previous paragraphs.

\end{document}
```

Resultado: La aplicación ha resuelto la prueba correctamente, ya que ha dividido este documento en tres porciones: párrafo uno, párrafos 2-4, párrafo 5.

- El primer párrafo tiene balance 0 porque el único begin que contiene es referente al entorno *document*, que no debe ser tenido en cuenta.
- Los párrafos 2 a 4 forman en conjunto una porción de documento con balance 0, ya que el begin{enumérate} del párrafo 2 tiene su final en el cuarto párrafo. Con esta selección se demuestra también que el contenido literal del comando *verb* y el entorno *verbatim* se respetan y no forman parte del balance.
- El párrafo 5 que solo contiene el *\end{document}* también se considerá balanceado por las mismas razones que el primero.

Prueba nº 5: Comprobación del funcionamiento del LT

Descripción: esta prueba consiste en comprobar que la aplicación es capaz de inicializar el corrector gramatical que se ha utilizado en el proyecto (LanguageTool), proporcionarle parte del texto que contiene un documento LaTeX, y que este sea capaz de encontrar y devolver errores.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
I were in the kitchen
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

This is a test test LaTeX document.
I were in the kitchen.

\end{document}
```

Resultado: al pasar el corrector a este documento LaTeX, la interfaz ha encontrado dos errores:

- Determina que en la primera frase del párrafo central la palabra “test” está repetida.
- Determina que el verbo “were” de la segunda frase del párrafo central no corresponde a la frase empleada.

Por tanto, comprobamos que el LanguageTool ha sido puesto correctamente en funcionamiento, que recibió el texto a corregir, y que fue capaz de encontrar errores en él, devolviendo referencias correctas a los mismos.

Prueba nº 6: Comprobación de donde se empieza a checkear el LaTeX

Descripción: La aplicación está pensada para trabajar con el documento a partir del comando \begin{document}. Antes puede haber algo de texto susceptible de ser corregido, pero como no aparece dentro del documento final (compilado), se ha preferida pasarlo por alto y no tener que contemplar todos los comandos que son habituales en esa parte del documento.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
I were in the kitchen
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

I were in the kitchen.

\end{document}
```

Resultado: La aplicación ha resuelto la prueba satisfactoriamente, ya que la interfaz solo muestra como error el "were" de la frase que forma el párrafo central, pasando por alto el error en la misma frase antes del "\begin{document}".

Prueba nº 7: Comprobación de la correcta captura de comandos LaTeX que poseen argumentos

Descripción: se comprobará aquí el funcionamiento del método argumentTreatment que posee el transformer para obtener un comando completo con sus argumentos, y estos últimos por separado, respetando el formato que pueden tener en el lenguaje LaTeX. El comando se guarda en la aplicación dentro de un ArrayList<String>.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

Did the US government intentionally leave the terrorists destroy
the Empire State buildings, \comando   {primer argumento}
   {segundo argumento   }       {   tercer argumento}{\em,} or they
actually did it themselves?

\end{document}
```

Comando formateado por el *argumentTreatment*:

completeCommand.get(0) = "`\comando   {primer argumento}\n   {segundo argumento }       {   tercer argumento}`"
completeCommand.get(1) = "`primer argumento`"
completeCommand.get(2) = "`segundo argumento   `"
completeCommand.get(3) = "`    tercer argumento`"

Resultado: la prueba ha sido superada, ya que dada la posición del símbolo "\" dentro del párrafo central, el método objeto de la prueba ha sido capaz de detectar cuales eran los límites del comando con la información acerca del número de argumentos recibida desde el archivo XML. Para la prueba se había creado un comando "\comando", con tres argumentos, de tipo select_In, y con tratamiento de eliminación.
La primera posición del array guarda correctamente el comando completo con espacios entre comando y argumentos, y entre argumentos (este formato es correcto para LaTeX), lo cual es necesario porque una futura sustitución se realizaría basada en esta cadena. Las demás posiciones del ArrayList de Strings también han sido rellenadas satisfactoriamente,

ya que los argumentos han sido almacenados tal y como aparecen en el texto, sin eliminar los espacios.

Prueba nº 8: Comprobación del proceso iterativo de transformación con segmentos de documento LaTeX.

Descripción: Esta prueba consistirá en comprobar:

- Que la aplicación es capaz de realizar correctamente el proceso iterativo de transformación de segmentos de documento LaTeX, para lo que deberá de recorrer la variable treatedText y transformar sus índices.
- Que durante este proceso se crean correctamente nuevos PieceOfParagraph dentro de la variable treatedText con el texto y offset adecuados -> comandos select_Out.

Se realizarán las pruebas con el comando \footnote.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

Let's test GrammaTeX. We're gonna type some LaTeX commands and
environments here. This \footnote{\mbox{\underline{I}}
were in the cinema.}items are useful. He goes
\footnote{I \mbox{w\`{e}r\`{e} in} the cinema.} there when she glanced
at him. I \footnote{I \mbox{were in} the cinema.}were in
the kitchen. Berta \mbox{measure} the size
\footnote{\mbox{\emph{\underline{This}}
 items} are} very useful.} of the ladder. Test over over.

\end{document}
```

Segmento de código central transformado para ser analizado:

- **treatedText[0]**

```
Let's test GrammaTeX. We're gonna type some LaTeX commands and
environments here. This items are useful. He goes
 there when she glanced
at him. I were in
the kitchen. Berta measure the size
 of the ladder. Test over over.
```

- **treatedText[1]**

```
I
were in the cinema.
```

- **treatedText[2]**

```
I were in the cinema.
```

- **treatedText[3]**

```
I were in the cinema.

    • treatedText[4]

This
 items are very useful.
```

Resultado: la prueba ha sido correctamente superada por la aplicación, ya que el hecho de que los errores presentes en cada uno de los segmentos de texto almacenados en diferentes índices hayan sido subrayados correctamente en el documento de la interfaz, demuestra que se han realizado adecuadamente las acciones que se deseaban comprobar en esta prueba.

- Se ha determinado que la aplicación itera correctamente, transformando primeramente el texto base, y después los sucesivos índices de la variable treatedText donde están almacenados los nuevos PieceOParagraph que surgen producto de las transformaciones.
- Se han extraído satisfactoriamente segmentos de texto que formaban parte del argumento de comandos select_Out, y que han sido transferidos a las diferentes posiciones de la variable treatedText. Se ha comprobado que el texto de estos comandos era seleccionado adecuadamente en tiempo de ejecución, y el offset calculado también, ya que de lo contrario el posicionamiento del subrayado de errores se hubiese realizado de manera incorrecta en la interfaz.

Prueba nº 9: Comprobación del tratamiento que reciben los comandos de sustitución directa

Descripción: esta prueba consiste en introducir en una frase que debería de lanzar un error gramatical todos los comandos de sustitución directa que son eliminados por completo por la aplicación para demostrar que se realiza dicho tratamiento correctamente y la frase puede ser corregida satisfactoriamente. Se introducirán también los demás comandos de este tipo en frases correctas para ver que la sustitución se realiza bien y las frases no dan errores.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

I \maketitle \tableofcontents were \listoffigures \listoftables in
\makeindex
\makeglossary the kitchen. A list could be made with element one, element
two, element three, \ldots
\$ \% \& \# \{ \} \_
Let's write a formula: $ a + b = c $.
This page was written on \today.
The \item first \upshape command \itshape deleted \slshape in \scshape
this
\mdseries line \bfseries should \rmfamily be \sffamily used \ttfamily
inside a list
environment.
```

```
\end{document}
```

Segmento de código central transformado para ser analizado:

```
I   were   in
 the kitchen. A list could be made with element one, element
two, element three, ...
$ %  # { } _
Let's write a formula: SDformula.
This page was written on Sunday.
The  first  command  deleted  in  this
 line  should  be  used  inside a list
environment.
```

Resultado: Todas las transformaciones se realizan correctamente, tanto las de eliminación, como las de sustitución.

- La primera frase, como puede verse, queda con el formato adecuado para poder ser analizada, y de hecho, devuelve el error en el verbo "were".
- El ldots es sustituido correctamente por puntos suspensivos.
- Los siete comandos asociados a los caracteres especiales de LaTeX se transforman correctamente. Además los segmentos prohibidos asociados al "$" y al "%" se han guardado correctamente. El "$" transformado se respeta, porque si no se hubiese realizado una transformación de fórmula con ese símbolo y el que comienza la fórmula de verdad. El "%" se ha respetado igualmente, ya que no se ha generado ningún *PieceOfParagraph* nuevo con lo que restaba de línea.
- El comando \today también ha tenido una sustitución correcta.
- El comando \item y los comandos de sustitución directa para formateo de fuentes han sido eliminados correctamente.

Prueba nº 10: Comprobación del tratamiento que reciben los comandos de tipo select_In

Descripción: en esta prueba vamos a intentar demostrar que la aplicación trata correctamente los comandos de tipo select_In, tanto en el texto base, como en otros PieceOfParagraph. Para ello, la aplicación deberá eliminar o sustituir el comando y sus argumentos, y realizar también el manejo de las áreas de efecto asociadas a estos, de manera que los errores contenidos en ellos utilicen los offsets parciales para poder ser mostrados en la interfaz en el lugar adecuado. La aplicación es capaz de tratar muchos comandos de este tipo, por lo que se utilizarán en la prueba unos pocos que demuestren los diferentes tratamientos (aunque han sido probados todos durante la implementación).

Documento LaTeX original:

```
\documentstyle[11pt]{article}
I were in the kitchen
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

He \mbox{go there when} she glanced at him. \label{select_In_test}
I \footnote{\mbox{\underline{I}} \mbox{\underline{w\`{e}r\`{e} in}} the
cinema.}were in
the kitchen.
Berta \emph{\underline{\mbox{measure}}} the size of the ladder.
\mbox{\emph{\underline{This} items} are} very useful.
```

```
I \footnote{\mbox{\underline{I }}\mbox{w\`{e}r\`{e} in} the cinema.}were
in the
kitchen.
I would like \texttt{\textbf{to live }}more closer to my friends

\end{document}
```

Segmento de código central transformado para ser analizado:

- **treatedText[0]**

```
He go there when she glanced at him.
I were in
the kitchen.
Berta measure the size of the ladder.
This items are very useful.
I were in the
kitchen.
I would like to live more closer to my friends
```

- **treatedText[0]**

```
I were in the cinema.
```

- **treatedText[0]**

```
I were in the cinema.
```

Resultado: la prueba ha sido resuelta por el programa satisfactoriamente, ya que:

- Se han introducido en el texto de la prueba comandos select_In (sin tratamiento de borrado) anidados que han sido transformados adecuadamente tanto en el texto base como en segmentos de texto que posteriormente iban a ser trasladados del texto original a otros índices de la variable treatedText.
- El correcto subrayado en la interfaz de los errores contenidos en estos comandos, indica que se realiza un adecuado tratamiento de los segmentos en los que debe aplicarse el offset parcial que estos provocan.
- Además, se ha comprobado directamente que la longitud de estos segmentos varía en función de las transformaciones que se realizan dentro de ellos. Para ello, se han colocado errores gramaticales contiguos a los argumentos de los comandos, de manera que de no ser por la reducción de los segmentos fruto de transformaciones internas, esos errores quedarían dentro de dichos segmentos, utilizando solo el offset parcial, cosa que no ha ocurrido.
- Se ha introducido el comando \label en la prueba, que demuestra que los comandos select_In con tratamiento de borrado (change = 0) son eliminados.

Prueba nº 11: Comprobación del tratamiento que reciben los comandos de acentuación

Descripción: en esta prueba se tratará de demostrar la correcta transformación de los comandos de acentuación. Se realizará la prueba sobre los diferentes tipos de comandos de este grupo y sobre los argumentos que estos pueden contener.
Finalmente habrá que comprobar también que se calculan bien los modificadores que surgen fruto de la transformación de estos comandos, comprobando que la aplicación muestra correctamente los errores situados en estas palabras.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
I were in the kitchen
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

I \`{w}ere in the kitchen.
I we\´{r}e in the kitchen.
I wer\`{e} in the kitchen.
I \`{w}e\´{r}e in the kitchen.
I w\`{e}r\´{e} in the kitchen.
I \`{w}\´{e}re in the kitchen.
He g\t{oo} there when that happened.
Th\t{\i\i}s items are useful.
\^{\j}\"{\i}\~{\i}\={\i}a\.{o}\u{o}\v{o}\H{\i}aea\t{oo}\t{\i\i}\c{\j}\d{\j}
\b{\j}lk!!!
I \footnote{I \mbox{w\`{e}r\`{e} in} the cinema.}were in the kitchen.

\end{document}
```

Segmento de código central transformado para ser analizado:

- **treatedText[0]**

```
I were in the kitchen.
I were in the kitchen.
I were in the kitchen.
I were in the kitchen.
I were in the kitchen.
I were in the kitchen.
He go there when that happened.
This items are useful.
jiiiaoooiaeaoijjjlk!!!
I were in the kitchen.
```

- **treatedText[1]**

```
I were in the cinema.
```

Resultado: la prueba ha sido resuelta por la aplicación satisfactoriamente. Cada una de las frases de la prueba tenía un fallo sobre la palabra acentuada, y esa palabra ha sido correctamente subrayada por la aplicación en el documento de la interfaz.

- Se comprueba que todos y cada uno de los comandos de este grupo son transformados correctamente en la penúltima frase del treatedText[0].
- Se comprueba la correcta transformación de los acentos sobre las letras i y j con un tratamiento especial, y el de las demás letras tratadas de forma general, estén donde estén situadas en las palabras.
- Se comprueba el correcto cálculo del modificador de posición que generan los acentos que se ponen en la primera letra de una palabra.
- Se comprueba el correcto cálculo de los modificadores de longitud para los diferentes tamaños que tienen los comandos junto con sus argumentos.
- Se comprueban estas cuatro condiciones anteriores para el tratamiento de palabras que tienen más de un acento, tanto con letras acentuadas separadas, como seguidas.
- Se comprueba el correcto cálculo de los modificadores de posición y longitud dentro

de otras transformaciones, incluso dentro de segmentos de texto diferentes al texto base.

Prueba nº 12: Comprobación del tratamiento que reciben los entornos de tipo *directEnv*

Descripción: esta prueba consiste en ver que las transformaciones de los comandos de tipo *directEnv* se realiza correctamente.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

Formulas have to be typeset in math mode:
\[
z \left( 1 \ +\ \sqrt{\omega_{i+1} + \zeta -\frac{x+1}{\Theta +1} y + 1}
\ \right)
\ \ \ =\ \ \ 1
\]

\(
z \left( 1 \ +\ \sqrt{\omega_{i+1} + \zeta -\frac{x+1}{\Theta +1} y + 1}
\ \right)
\ \ \ =\ \ \ 1
\)

\end{document}
```

Segmento de código central transformado para ser analizado:

```
Formulas have to be typeset in math mode:
MathFormula

MathFormula
```

Resultado: es obvio que las transformaciones se han realizado correctamente, y que por lo tanto, la prueba ha sido superada.

Prueba nº 13: Comprobación del tratamiento que reciben los comandos verb y verb*, y el entorno verbatim.

Descripción: en esta prueba se va a intentar demostrar que el transformer realiza el tratamiento de los comandos de tipo verbatim correctamente, realizando las sustituciones adecuadas, y llevando una gestión correcta de la lista que contiene los segmentos prohibidos (que no permiten el uso de elementos internos para otras transformaciones).

Documento LaTeX original:

```
\documentstyle[11pt]{article}
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.
```

```
\begin{verbatim}This is a verbatim environment $ \maketitle\end{verbatim}
\begin{verbatim}* $\end{verbatim}
\verb+$ in verb+
\verb*2$ in verb*2
\verb$U shouldn't see a dollar in this line$
I \mbox{were} in the kitchen. I \footnote{$ c  + c = c $.
I \mbox{were in the cinema}} were in the kitchen. I \footnote{\mbox{$ c  +
c =
c $}. I \mbox{were in the cinema} \verb$YO$ $} were in the kitchen. $

\end{document}
```

Segmento de código central transformado para ser analizado:

- **treatedText[0]**

```
This is a verbatim environment $ \maketitle
* $
$ in verb
$ in verb*
U shouldn't see a dollar in this line
I were in the kitchen. I  were in the kitchen. I  were in the kitchen.
```

- **treatedText[1]**

```
SDformula.
I were in the cinema
```

- **treatedText[2]**

```
SDformula. I were in the cinema YO
```

Resultado: prueba superada satisfactoriamente. Observaciones:

- El texto contenido en los entornos verbatim se ha transformado correctamente, y los tags "$" y comando *\maketitle* no han sido transformados, por lo que se han utilizado correctamente los segmentos prohibidos.
- Los comandos *\verb* y *\verb** también se han transformado satisfactoriamente, respetándose también los segmentos prohibidos. Se ha comprobado también el uso de otros símbolos con los que comienzan los tags como delimitadores de estos comandos sin que falle la aplicación. (El texlipse no contempla el comando \verb*, y por eso asigna mal los delimitadores en el texto original -> ver colores)
- Por último, se ha comprobado que la transformación de este tipo de comandos y el posicionamiento de los segmentos prohibidos que generan se maneja correctamente cuando hay transformaciones cuyo offset es necesario aplicarles, incluso en porciones de texto analizadas por separado.

Nota: los símbolos "$" que se han dejado al final son los que demuestran que ninguno de los otros que están contenidos dentro de los segmentos prohibidos es utilizado para formar un comando.

Prueba nº 14: Comprobación del método fixSimpleMath

Descripción: esta prueba consistirá en comprobar la correcta selección de los tags "$" que utilizan algunas formulas matemáticas, y posterior tranformación de este tipo de comandos.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
I were in the kitchen
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

Mathematical formulas can be written in LaTeX by enclosing
them in dollar sign symbols -> \$ formula \verb-$-
This is an example: $$x_n = \sqrt{a + b}$$
This is another example: $x^y$$

\end{document}
```

Segmento de código central transformado para ser analizado:

```
Mathematical formulas can be written in LaTeX by enclosing
them in dollar sign symbols -> $ formula $
This is an example: DDformula
This is another example: SDformula
```

Resultado: el resultado es satisfactorio. La aplicación determina que los dos primeros símbolos "$" están en segmentos prohibidos, de manera que los descarta para las transformaciones (si no se habrían emparejado con el primer "$" valido de la primera fórmula). Después se realizan correctamente los dos tipos de tratamiento que tiene este método: fórmulas con tags "$", y fórmulas con tags "$$". Finalmente, la aplicación descarta el símbolo "$" que queda, ya que no puede ser emparejado, y por tanto tampoco transformado.

Prueba nº 15: Comprobación del método fixComment

Descripción: esta prueba consistirá en comprobar la correcta selección de los símbolos "%" con los que comienzan los comentarios de LaTeX, y la posterior selección y transformación del propio comentario.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

When LaTeX encounters a \% character in the input, it
ignores the \% and all the characters following it on
that line. %including the carriage return
It also ignores spaces at the beginning of the next line.
I think th%I were in the kitchen
        at this command is really useful.

\end{document}
```

Segmento de código central transformado para ser analizado:

- **`treatedText[0]`**

```
When LaTeX encounters a % character in the input, it
ignores the % and all the characters following it on
that line. It also ignores spaces at the beginning of the next line.
I think that this command is really useful.

   • treatedText[1]

including the carriage return

   • treatedText[2]

I were in the kitchen
```

Resultado: el resultado de la prueba es satisfactorio. En primer lugar, los dos primeros símbolos "%", que se encuentran dentro de segmentos prohibidos, han sido correctamente pasados por alto sin llevar a cabo con ellos el tratamiento para comentarios. Los otros dos símbolos "%" sí que pertenecen a comentarios, y por tanto han sido extraídos correctamente del texto original para ser analizados de manera independiente. Se encuentran y posicionan los errores bien dentro de los PieceOfParagraph que contienen a ambos comentarios, por lo que sus desplazamientos también deben estar bien calculados. Además, se puede ver que se realiza el tratamiento de los comentarios exactamente como lo hace LaTeX, incluyendo en el comentario lo que resta de línea a partir del "%", y los primeros espacios que hay en la siguiente línea.

Prueba nº 16: Comprobación del tratamiento que reciben los entornos de tipo *directEnvBE*

Descripción: esta prueba consiste en comprobar si se realizan las transformaciones de este tipo de entornos correctamente, buscando los límites del entorno y realizando la sustitución oportuna.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

- The equation environment.
\begin{equation} \left[ {\bf X} + {\rm a} \ \geq\
\underline{\hat a} \sum_i^N \lim_{x \rightarrow k} \delta C
\right]\end{equation}
- The displaymath environment.
\begin{displaymath} |a\cos x +b \sen x|\le \sqrt{a^2+b^2}
\end{displaymath}
- The math environment.
\begin{math} \int_a^x \!\!\!\int_a^s f(y)\,dy\,ds
 = \int_a^x f(y)(x-y)\,dy \end{math}
\begin{thebibliography}
\bibitem{kn:gnus} D. E. Knudson
\emph{1966 Gnus Almanac.}
\ldots
\end{thebibliography}
\begin{theindex}
indexentry 1
indexentry 2
\end{theindex}
```

```
\begin{tabular}{|l|c|p{3.5in}|}
\multicolumn{3}{|c|}{Places to Go Backpacking}\\ \hline
Name&Driving Time&Notes\\
&(hours)&\\ \hline
Sunol&1&Technicolor green in the spring. Watch out for the cows.\\ \hline
\end{tabular}

\end{document}
```

Segmento de código central transformado para ser analizado:

```
- The equation environment.
MathEquation
- The displaymath environment.
DisplayMath
- The math environment.
MathFormula
```

Resultado: las transformaciones de todos los entornos se han realizado correctamente, prueba superada.

Prueba nº 17: Comprobación del tratamiento que reciben los *floatingElements*.

Descripción: esta prueba consiste en comprobar que cuando la aplicación encuentra un *floatingElement* es capaz de obtener la información textual que contienen y crear nuevos PieceOfParagraph con ellos, eliminando el comando completo del texto origen.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

- Figure.
\begin{figure}
\begin{minipage}{5.8cm}
\begin{tabular}{c}
\includegraphics[height=4.8cm]{totalPPFLeoTrueCSFx3_136.eps} \\
(a)
\end{tabular}
\end{minipage}
\caption{This is the the text that should be extracted and analyzed.}
\caption{I were in the kitchen}
\label{fig:ppf}
\end{figure}
- Table.
\begin{table}[h]
\vspace{3mm}
{\centering
\begin{tabular}[t]{|c|c|} \hline
\pbox[t]{20mm}{\textbf{Heading}} &
\pbox[t]{100mm}{\textbf{Heading}} \\
\hline \end{tabular}
\par} \centering
\caption{Long long table caption to appear below the table.}
\label{table:template}
\vspace{3mm}
```

```
\end{table}

\end{document}
```

Segmento de código central transformado para ser analizado:

- **treatedText[0]**

```
- Figure.

- Table.
```

- **treatedText[1]**

```
This is the the text that should be extracted and analyzed.
```

- **treatedText[2]**

```
I were in the kitchen
```

- **treatedText[3]**

```
Long long table caption to appear below the table.
```

Resultado: la prueba se considera superada por las siguientes razones:

- El código contenido dentro de los entornos de este tipo ha sido eliminado completamente de la frase origen, permitiéndose así el análisis de la misma.
- Se han creado nuevos PieceOfParagraph con el texto contenido en los caption de ambos entornos.
- Los nuevos PieceOfParagraph han sido analizados correctamente, y los errores lanzados por el corrector estaban bien colocados gracias al offset que se cálcula para los segmentos de texto extraídos.

Prueba nº 18: Comprobación del tratamiento que reciben los entornos de tipo *deleteTags.*

Descripción: en esta prueba vamos a comprobar el funcionamiento de las sustituciones que se realizan con los entornos del grupo deleteTags, donde se deberían de sustituir los entornos por el texto que contienen, eliminando los tags de comienzo y final. Se comprobará también la aplicación de los offsets parciales a los errores que se encuentren dentro del texto que albergan los entornos, para lo cual se han de gestionar correctamente las áreas de efecto asociadas a este tipo de transformaciones.

Documento LaTeX original:

```
\documentstyle[11pt]{article}
\begin{document}
\title{LaTeX testing document}
\author{Julen}
This is a LaTeX testing document.

\begin{center}\begin{enumerate}I\begin{itemize}
were\begin{description} in\begin{quote} the
\begin{quotation}kitchen.\begin{tabbing}
This items are useful.
\end{tabbing} I \end{quotation}
\end{quote}\end{description}were in the
```

```
\end{itemize} cinema.\end{enumerate}\end{center}This
items are useful

\end{document}
```

Segmento de código central transformado para ser analizado:

```
I
were in the
kitchen.
This items are useful.
 I
were in the
 cinema.This
items are useful
```

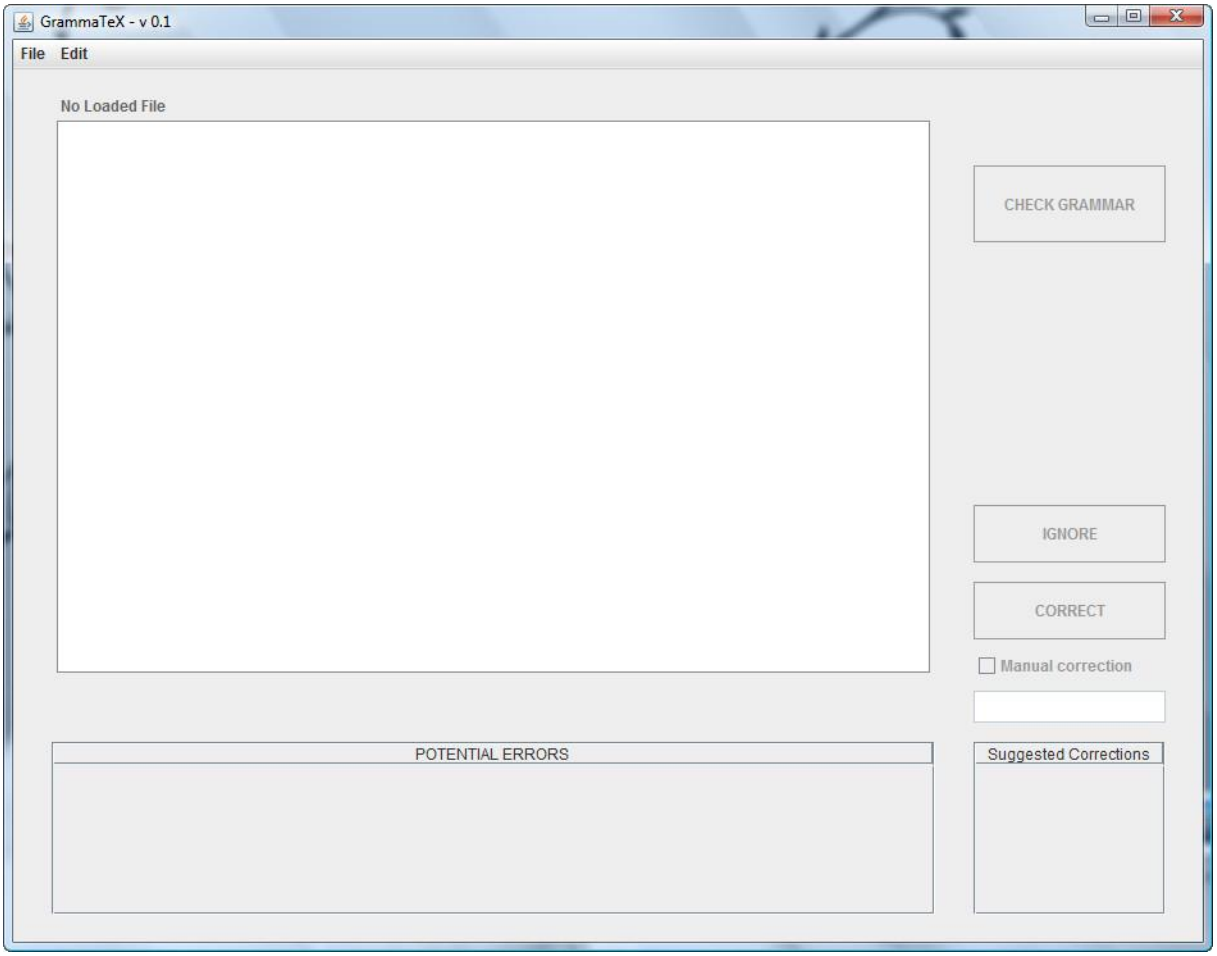
Resultado: la prueba se considera superada por las siguientes razones:

- Los tags de inicio y final de cada entorno son eliminados correctamente, dejando intacto el texto que contienen dentro.
- Se ha seguido el desarrollo de la prueba para comprobar que los \end son eliminados conjuntamente con sus \begin respectivos, y no mediante el tratamiento de eliminación independiente de estos que se realiza por seguridad.
- Los errores gramaticales que se producen dentro de los entornos son correctamente posicionados en la interfaz, lo que asegura que se manipulan bien las áreas de efecto de las transformaciones asociadas, aplicándose satisfactoriamente los offsets parciales.
- Se ha introducido en el documento un error justo al término del entorno más externo, de cara a la demostración de que el área de efecto de las transformaciones se va alterando en función de las transformaciones que se realizan dentro. Si no se hubiese reducido bien este área de efecto en el ejemplo, la última frase hubiese quedado dentro de ella, ya que se ha eliminado texto dentro del entorno *center*, pero el error que contiene ha sido posicionado correctamente.

5. MANUAL DE USUARIO

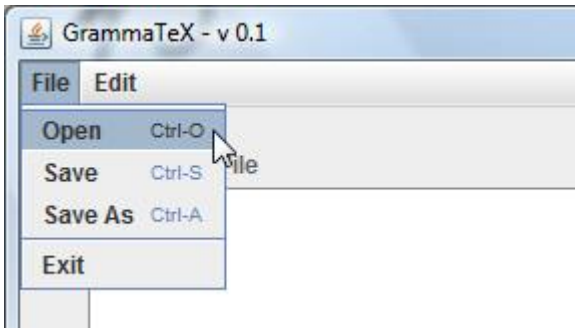
5.1 PUESTA EN MARCHA

La interfaz de la aplicación ha sido desarrollada pensando en ser lo más simple y fácil de utilizar posible, ya que el proceso de chequeo de la gramática es solo complejo a nivel de tratamiento del texto, requiriendo el programa acciones muy simples por parte del usuario para completar la labor para la que ha sido ideado. Cuando se abre la aplicación nos encontramos con la siguiente pantalla:



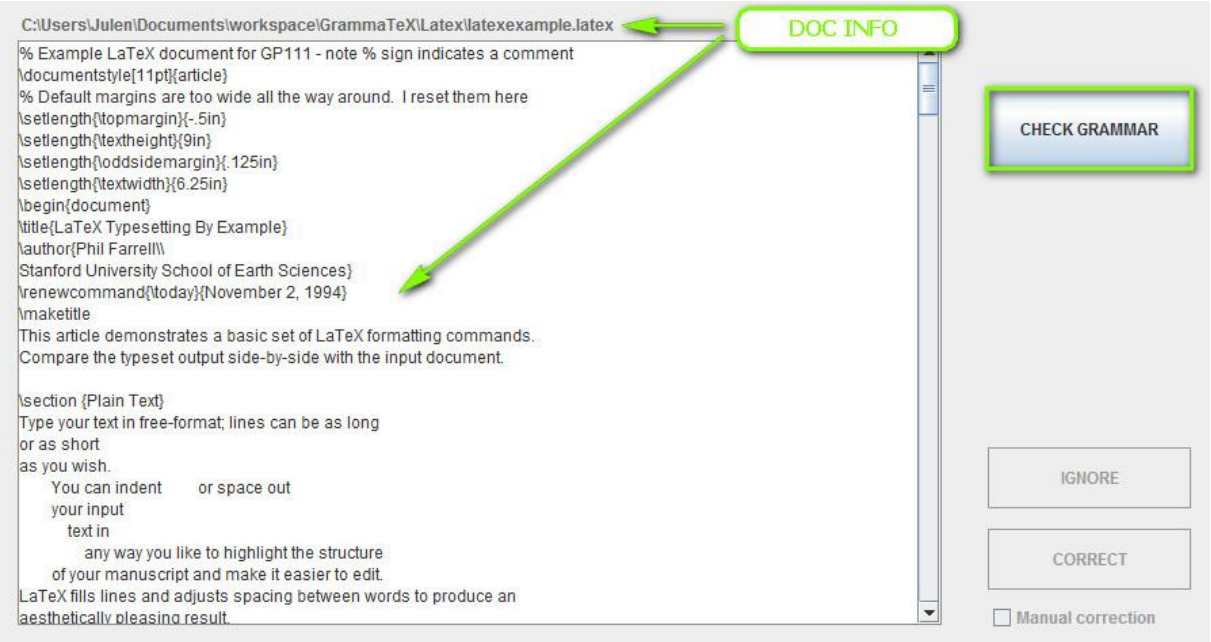
6.1 - Vista Inicial de GrammaTeX

Vemos que todos los controles están deshabilitados, lo cual se debe a que la aplicación no tiene sentido si no trabaja con un documento LaTeX cuya gramática corregir. Por tanto, el primer paso consistirá en hacer click sobre el menú "File" en la parte superior izquierda de la pantalla, que al ser desplegado mostrará el siguiente aspecto:



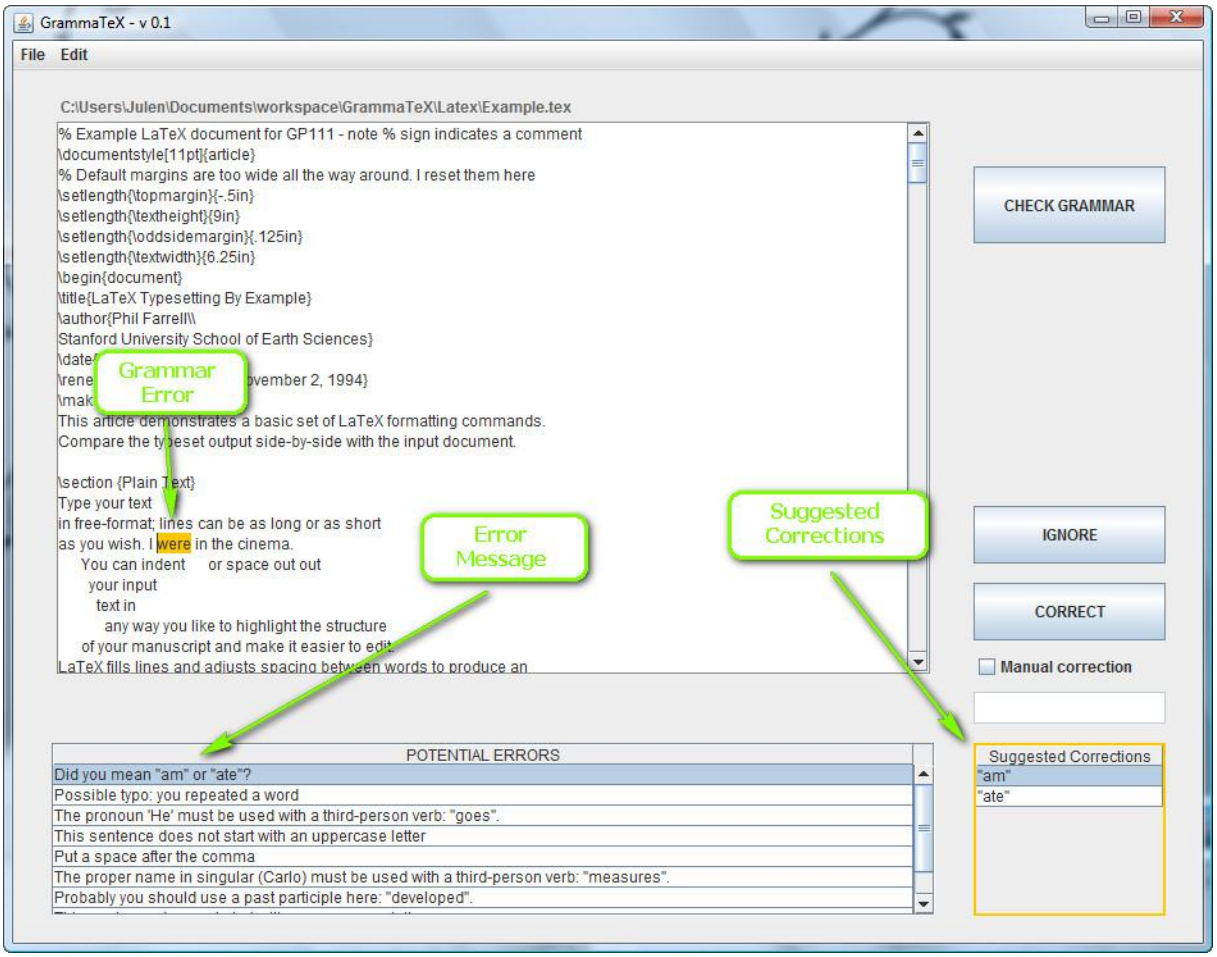
6.2 - Vista del menú desplegable "File"

Seleccionamos la opción "Open" dentro del menú, y se abrirá el clásico selector de archivo (similar al habitual en cualquier sistema operativo), dentro del cual hay que escoger el archivo LaTeX cuya gramática deseamos validar. Una vez seleccionado el archivo correspondiente, la aplicación muestra el contenido del mismo en el recuadro blanco que domina la vista, además de una referencia a la ruta del archivo justo encima de este. Se puede observar (figura 6.3) que el botón "Check Grammar" en la esquina superior derecha de la pantalla ha sido habilitado, ya que la aplicación ya tiene cargado un documento.



6.3 – Vista de GrammaTeX tras cargar un archivo

Si hacemos click sobre este botón, la aplicación comenzará a analizar el documento cargado, transformando el documento LaTeX a texto plano, y validando este mediante el corrector gramatical que utiliza GrammaTeX. En caso de que la aplicación encuentre algún error gramatical en potencia, la pantalla pasará al estado *activo*, de manera que todas las opciones de la pantalla principal quedan activadas (figura 6.4). Si por el contrario no se hallara ninguna posible sugerencia de corrección, la aplicación mantendrá sus controles desactivados hasta que se realice un cambio de documento (cambios sobre el actual o carga de otro nuevo) y compruebe su gramática encontrando errores.



6.4 – Vista de GrammaTeX (Activo)

Llegados a este punto, la aplicación nos proporciona los resultados del análisis mostrando una lista de potenciales errores gramaticales en la tabla "POTENTIAL ERRORS" de la parte inferior de la vista. Cada entrada de esta tabla representa un error gramatical, y en ella se muestra una breve explicación que ayuda al usuario a entenderlo.

La aplicación selecciona por defecto (y como ayuda al usuario) la primera entrada de la tabla. Tanto en este primer caso por defecto, como cuando el usuario selecciona por sí mismo cualquier entrada de la tabla, se producen dos eventos:

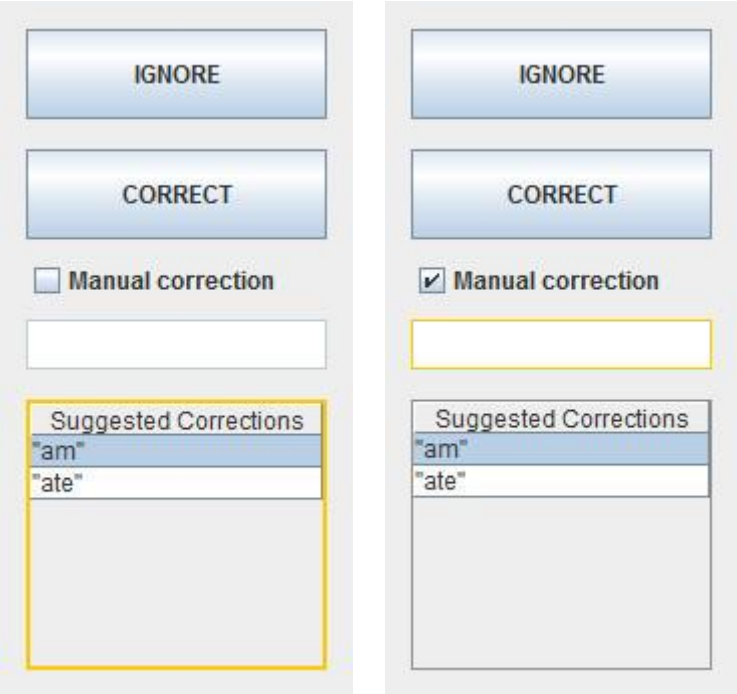
1. En primer lugar se mostrará en el documento un segmento de texto subrayado en naranja que muestra el error apuntado por la entrada de la tabla. Esta porción de texto es la que será transformada como consecuencia de la corrección automática (mediante la tabla de sugerencias) o manual seleccionada por el usuario.
2. En la esquina inferior derecha de la pantalla se rellenará la tabla "Suggested Corrections" con las sugerencias de corrección que proporciona el corrector gramatical que utiliza nuestra aplicación. Se selecciona automáticamente la primera sugerencia como ayuda al usuario, ya que en los casos en que solo exista una sugerencia, o la primera de ellas sea la que necesita el usuario, evitamos la necesidad de clickear en esta una y otra vez.

5.2 CORRECCIÓN DE ERRORES

Cuando un error de la tabla está seleccionado y vemos en el documento un segmento subrayado podemos proceder de dos maneras, ignorando el error o corrigiéndolo automática o manualmente.

En caso de que se decida que el corrector gramatical se ha equivocado a la hora de interpretar el error actual, y dicha porción de texto esté escrita como queremos que sea presentada en el documento final, basta con pulsar el botón *Ignore* situado en la parte inferior derecha de la aplicación. Esto provocará que el error que estaba activo sea eliminado de la tabla de errores y se seleccione automáticamente uno nuevo, con el consiguiente cambio en la tabla de sugerencias de corrección y de error subrayado en el documento.

Si por el contrario, resulta que el error detectado por la aplicación es en efecto un error, puede ser corregido pulsando el botón *Correct* (situado justo debajo del anterior), lo cual provocará la sustitución del segmento subrayado en la aplicación por uno de nuestra elección. El texto que será utilizado para corregir el error puede provenir de dos fuentes: de la tabla de sugerencias (por defecto), o del campo de texto de corrección manual. Debajo del botón *Correct* se puede encontrar un checkbox que funciona a modo de switch (figura 6.5) entre ambos tipos de corrección. Podemos saber cuál es el tipo de corrección activa en un momento dado mirando la ayuda visual que se proporciona en forma de halo alrededor de este. El halo es del mismo color que los errores del documento para hacerlo más intuitivo.

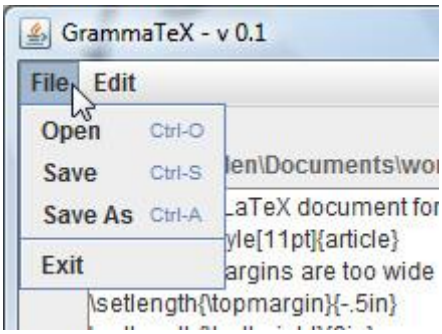


6.5 – Vista de GrammaTeX – Corrección automática/manual

En el caso de que nos decantemos por la corrección mediante sugerencias, basta con seleccionar aquella que deseamos y pulsar Correct mientras el *checkbox* está desactivado. Si deseamos realizar una corrección manual habrá que activar el checkbox, introducir el texto deseado, y pulsar *Correct*. La corrección manual es la única forma de realizar correcciones que contienen comandos LaTeX, ya que el corrector nunca ofrecerá este tipo de sugerencias (Ej.: palabras con acentos). Al igual que ocurre al ignorar un error, después de una corrección se elimina el error de la tabla, y será activado el siguiente, si es que lo hay.

5.3 MENÚ ARCHIVO

El menu "Archivo" de GrammaTeX permite guardar archivos y salir de la aplicación, además de cargar documentos en la vista. Presenta las siguientes opciones:

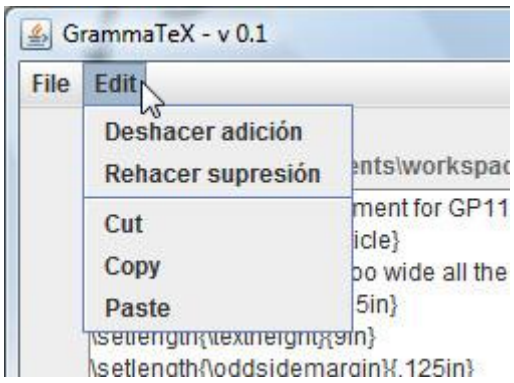


6.6 – Vista del menú "Archivo" de GrammaTeX

- **Save**: esta opción realiza un guardado del documento de la vista en la misma ruta desde la cual fue cargado. La acción puede realizarse mediante las teclas de acceso rápido CTRL+S sin acceder al menú.
- **Save As**: esta opción también realiza un guardado del documento activo en la vista, solo que en este caso se nos pedirá introducir un nombre nuevo o seleccionar un archivo ya existente. En caso de escoger un archivo existente pedirá confirmación de sobreescritura. La acción puede realizarse mediante las teclas de acceso rápido CTRL+A sin acceder al menú.
- **Exit**: en caso de seleccionar esta opción saldremos del programa directamente. Es importante saber en caso de salir del programa sin guardar el documento activo, los cambios no serán guardados.

5.4 MENÚ EDICIÓN

El menu "Edición" contiene las clásicas utilidades de los procesadores de texto. Presenta las siguientes opciones:



6.7 – Vista del menú "Edición" de GrammaTeX

- **Deshacer** (muestra la acción susceptible de ser deshecha)
- **Rehacer** (muestra la acción susceptible de ser rehecha)
- **Cortar**
- **Copiar**
- **Pegar**

6. COMENTARIOS FINALES

6.1 LINEAS FUTURAS

El proyecto, tal como se describía en la sección 1.2 de este documento, tenía como objetivo demostrar que era posible realizar una aplicación relativamente simple que pudiese corregir gramaticalmente un documento LaTeX. No obstante, solo se han implementado en esta primera fase los cimientos de lo que se espera que sea en un futuro GrammaTeX.

En primer lugar, la aplicación es capaz de tratar un número limitado de comandos de LaTeX, pero ha sido desarrollada de manera que este conjunto pueda ser ampliado fácilmente, tanto en número como en tipos. Los diferentes comandos pueden ser reducidos a un pequeño número de tratamientos diferentes, de los cuales ya hay varios implementados. Esto permitirá añadir en un futuro fácilmente comandos que compartan alguno de los tratamientos que ya están implementados, siendo únicamente necesario incluirlos en el archivo XML con el formato adecuado.

Para aquellos comandos que requieran de un tratamiento diferente a los ya implementados, será necesario incluir dicho tratamiento dentro del código de la aplicación. Hay que decir, que las rutinas de tratamiento son bastante sencillas, por lo que no supondrá un gran esfuerzo desarrollarlas (tienen en torno a 5-10 líneas de código).

En relación a la recién comentada agregación de nuevos comandos a la aplicación, sería deseable que futuras versiones de GrammaTeX fuesen capaces de ampliar el tratamiento que se les da a los comandos desconocidos, creando con ellos una lista que le sería ofrecida al usuario al término del análisis. De esta forma, el usuario sabrá qué nuevos comandos debe de introducir en la aplicación, ya que además es probable que los utilice de manera recurrente en sus documentos.

Hay también una serie de sencillos pasos que podrían ampliar el espectro de idiomas soportados por la aplicación. El corrector gramatical seleccionado para el proyecto, el LanguageTool, solo requiere ser inicializado con un lenguaje diferente para poder corregir texto plano en idiomas diferentes al inglés. Incluyendo un botón en la interfaz gráfica que seleccione de entre los diferentes idiomas soportados por esta herramienta, agregando las librerías con las reglas gramaticales para los nuevos idiomas, e inicializando en base a ello el LanguageTool, proveería a la aplicación de la capacidad de analizar textos en los dieciocho idiomas disponibles, algunos de ellos con más reglas de corrección que el inglés.

Por último, una de las limitaciones quizás más notorias de la aplicación desarrollada es que solo funciona como aplicación independiente. Hoy en día, en la búsqueda de un entorno de trabajo que permita unificar todas las herramientas y tecnologías necesarias para una labor concreta, es algo deseable que herramientas como la nuestra, de un tamaño reducido y muy

específicas, sean integradas dentro de otros programas de uso común. Así por ejemplo, la comunidad de usuarios que utilizan LaTeX desarrollan su trabajo habitualmente con herramientas como Eclipse, por lo que sería deseable poder incluir el código de GrammaTeX dentro de plugins como TeXlipse, pudiendo realizarse así la corrección gramatical "*on the fly*", ahorrando tiempo al usuario en darse cuenta de los errores cometidos, y facilitando sobremanera la labor de corrección. Por otro lado, esto no es solo aplicable al IDE Eclipse, que puede ser utilizado desde cualquier plataforma, sino que podría ser incluido en las herramientas que aún funcionando en un sistema operativo específico, son también de uso habitual, como pueda ser por ejemplo el TeXnicCenter para Windows.

6.2 ALTERNATIVAS

Se van a comentar a continuación las posibles alternativas de implementación que se han detectado a lo largo del desarrollo del proyecto.

En esta primera versión de la aplicación, la forma en que los diferentes comandos de LaTeX son encontrados y analizados es algo primitiva, ya que se carecía del tiempo suficiente para desarrollar el proyecto con un analizador sintáctico de LaTeX. No obstante, la aplicación resuelve satisfactoriamente la detección y transformación de los comandos que de momento han sido incluidos en ella. La implementación de este parser redundaría en una mejor comprensión del documento LaTeX por parte de la aplicación, lo cual facilitaría el manejo de estructuras LaTeX más complejas que las tratadas hasta el momento, como pueden ser los comandos con múltiples argumentos opcionales y no opcionales. De todas maneras, solo tiene sentido plantearse esta alternativa en el caso de que la aplicación vaya a mantenerse como "standalone", o vaya a integrarse en una herramienta que no contiene ya un parser propio de LaTeX, puesto que si se incluye en plugins como el texlipse que ya implementan uno, no sería necesario.

Otra posible alternativa a la implementación actual de la aplicación es el uso de expresiones regulares en el tratamiento de texto, y en la información contenida en el archivo XML de sustituciones LaTeX. En cuanto al uso de este tipo de expresiones en el tratamiento de texto, se podría mejorar con ellas por ejemplo la forma de capturar comandos y entornos. Las expresiones regulares nos permitirían una vez detectado el comienzo de un comando, el poder capturar el comando con sus argumentos o entorno completo directamente, sin recurrir a métodos adicionales, y calcular además los offsets y demás controles de los mismos durante este proceso.

De igual manera, si se utilizasen estas mismas expresiones en la información sobre sustituciones del archivo XML, podríamos reducir el número de atributos necesarios para especificar a la aplicación el tratamiento a realizar sobre un comando. La expresión regular asociada a un comando permitiría guardar en un solo atributo los campos *command*,

numArgs y *end*. Esta reducción del número de atributos necesarios para el tratamiento permitiría homogeneizar y por tanto simplificar el uso de estas plantillas.

Una tercera y última alternativa al código de GrammaTeX proporcionado sería la implementación de algún tipo de ordenamiento en la lista de sustituciones, y de métodos que permitan hacer un buen uso del mismo sacándole el máximo provecho. En esta primera versión de la aplicación, el número de comandos que se utiliza no es demasiado grande, y por tanto la pérdida de tiempo por recorrer la lista de sustituciones completa (en el peor de los casos) es también pequeña, pero a medida que la aplicación asimile más comandos, esta pérdida irá aumentando hasta el punto de convertirse en un serio problema de rendimiento. Es por ello preferible implementar este ordenamiento, que permitiría ahorrar mucho tiempo en la sustitución de comandos, aumentando ligeramente el tiempo que el parser tarda en analizar el documento XML y almacenar su información en la aplicación.

6.3 CONCLUSIONES

6.3.1 CONCLUSIONES SOBRE GRAMMATEX

El resultado final de la aplicación me parece satisfactorio, aunque un análisis crítico de la misma me obliga a decir que es también mejorable en muchos aspectos. Se explican a continuación las razones que respaldan la afirmación anterior mediante el análisis de la solución que el programa aporta a los objetivos que se fijaron al comienzo del proyecto.

En primer lugar, se ha incluido en la aplicación el tratamiento de 88 comandos LaTeX, los cuales cubren la mayor parte de los tratados en los primeros cinco capítulos del libro de Leslie Lamport (Lamport, 1994) tal y como se pretendía. Además, la aplicación tiene un tratamiento de eliminación por defecto para aquellos comandos que desconoce, ya que a lo largo de la implementación se ha constatado, como cabía esperar, que el corrector gramatical utilizado es capaz de detectar más errores con esta forma de operar. Se han establecido con estos comandos diez grupos de sustitución cuyo funcionamiento ha sido contrastado con varios documentos LaTeX. Estos grupos de tratamiento implementados son los más básicos, y entre ellos se encuentran algunos de los tratamientos más generales, por lo que podrán albergar gran cantidad de nuevos comandos fácilmente. Por todo ello, se puede afirmar que GrammaTeX cuenta con una base suficiente de comandos y grupos de tratamiento como para comenzar a ser utilizado.

En segundo lugar, el haber almacenado la información referente a las sustituciones en un documento XML externo permite que los usuarios puedan extender el número de comandos tratados por la aplicación fácilmente. La sencillez del formato de los comandos en este archivo permite además que incluso usuarios con unas mínimas nociones de informática puedan aumentar el número de sustituciones contempladas, lo cual amplia a su vez el espectro de usuarios que encontrarán útil la aplicación

El manejo de este archivo de sustituciones ha sido además resuelto con un parser que recoge correctamente la información almacenada en él. Se ha implementado un parser SAX para llevar a cabo esta tarea, ya que no hay necesidad de ir cambiando el documento desde la aplicación, y por tanto este tipo de parser nos ahorra recursos computacionales sin perder por ello eficacia en el tratamiento.

Existía la posibilidad de trabajar con dos correctores gramaticales de código abierto que podían ser integrados en GrammaTeX: el link grammar parser y el language tool. En este aspecto no existía una opción correcta y otra incorrecta, sino que se trataba de escoger uno de los dos correctores en función de la importancia que se le daba a sus ventajas e inconvenientes. En el caso de GrammaTeX, como ambos correctores funcionaban con el inglés, nos hemos decantado por el segundo, ya que se ha primado la posibilidad de una sencilla adaptación a otros lenguajes sobre la calidad en la corrección que presentaba el primero (solo inglés). De todas formas, la aplicación ha sido pensada para poder ser adaptada a diferentes correctores gramaticales a través de las subclases que se pueden crear para la clase abstracta GrammarChecker, la cual contiene los procedimientos generales que se necesitarán para transformar y adaptar la información obtenida de cualquier otro corrector.

Por último, y para completar las necesidades que se consideraban básicas en la aplicación, se debía desarrollar una interfaz de usuario simple e intuitiva. La interfaz que presenta GrammaTeX parece incluso demasiado simple a primera vista, pero es en realidad algo que se ha buscado intencionadamente. Para empezar, no se sabe si la aplicación seguirá siendo software independiente, por lo que no tenía mucho sentido centrarse demasiado en la interfaz. Además, el proceso de corrección gramatical es solo complicado en el procesado del texto que subyace a la interfaz de usuario, necesitando esta de muy pocas acciones por parte del usuario para llevar a cabo la tarea para la que ha sido diseñado el programa. Se ha tratado de maquetar la interfaz de manera que la mayoría de usuarios no necesiten ni leerse las instrucciones para saber utilizarla. Se han incluido también pequeñas ayudas visuales que la aplicación muestra a modo de feedback, pensando en mejorar la usabilidad de esta.

6.3.2 CONCLUSIONES PERSONALES

Comenzare diciendo que una de las grandes diferencias que he notado entre el proyecto de fin de carrera que se presenta en este documento y los trabajos realizados anteriormente a lo largo, tanto de la ingeniería técnica, como del grado en informática, ha sido la forma en la que he tenido que dar solución a los problemas que me han ido aconteciendo.

Durante los años de carrera siempre había algún compañero, profesor o libro recomendado dentro de una asignatura que podía echarte una mano a la hora de solventar los problemas derivados de las prácticas. En esta ocasión, los problemas ha habido que solventarlos a base de largas búsquedas en solitario, escudriñando en ocasiones la red en busca de foros o documentación relacionada con algún aspecto poco genérico de la aplicación, mediante la

lectura de libros de LaTeX que me permitiesen comprender como funciona este programa y poder tratar sus documentos, tomando decisiones sobre la implementación en solitario,… es decir, llevando a cabo la labor para la que son educados los ingenieros y con la cual no estaba todavía demasiado familiarizado.

Otra de las conclusiones de este proyecto es la confirmación de la necesidad de llevar a cabo un buen proceso de ingeniería del software a lo largo de todas las etapas de un proyecto software. Personalmente, he de decir que en este caso no supe realizar correctamente esta tarea, pues el problema parecía inicialmente más sencillo de lo que después se ha visto que era, tomando decisiones al principio del proyecto que me han lastrado considerablemente en etapas finales de la implementación de la herramienta.

Uno de los aspectos que me ha parecido más interesante, y a la vez más quebraderos de cabeza me ha costado, ha sido el uso de Java y su programación orientada a objetos. Durante la carrera no había estudiado nada de este lenguaje, simplemente conocía el paradigma de programación orientada a objetos, pero sin llevar nunca estos conocimientos al ámbito práctico. La diferencia entre la programación estructurada utilizada durante la carrera y este nuevo lenguaje es enorme, y sinceramente me ha parecido mejor. Buena culpa de ello la tiene el IDE Eclipse, una herramienta que facilita sobremanera la programación en este lenguaje con su *code completion*, vistas, perspectivas, plugins,… En mi caso, aunque en un principio supuso muchos problemas a la hora de hacer funcionar plugins como el *Visual Editor* dedicado a la construcción de interfaces SWING, resulto después en una herramienta con la que poder manejar conjuntamente todos los aspectos y tecnologías relacionados con mi proyecto: Java, LaTeX, XML, SVN, …

7. BIBLIOGRAFÍA

AbiSource Community. (n.d.). *AbiWord*. Retrieved Diciembre 2009, from http://www.abisource.com/

Berkeley Software Distribution. (n.d.). Retrieved Diciembre 2009, from http://es.wikipedia.org/wiki/Berkeley_Software_Distribution

Eclipse Plugin Central. (n.d.). Retrieved Diciembre 2009, from http://www.eclipseplugincentral.com/

Free Software Foundation. (2007). *GNU Lesser General Public License*. Retrieved Diciembre 2009, from http://www.gnu.org/copyleft/lesser.html

Free Software Foundation. (2007). *The GNU General Public License*. Retrieved Diciembre 2009, from http://www.gnu.org/licenses/gpl.html

Kile - an Integrated LaTeX Environment. (n.d.). Retrieved Diciembre 2009, from http://kile.sourceforge.net/

Knuth, D. E. (1984). *The TeXbook.* Reading (MA): Addison-Wesley.

Lamport, L. (1994). *LaTeX - A document praparation system (2nd Edition).* Reading (MA): Addison-Wesley Professional.

LaTeX Project. (n.d.). *LaTeX Project: LaTeX - A document praparation system*. Retrieved Diciembre 2009, from http://www.latex-project.org/

Mittlebach, F., & Rowley, C. (1999, Diciembre 12). *ltx3info.pdf.* Retrieved Diciembre 2009, from http://www.latex-project.org/guides/ltx3info.pdf

Naber, D. (n.d.). *LanguageTool Open Source language checker*. Retrieved Diciembre 2009, from http://www.languagetool.org/

Naber, D. (2003, Agosto 28). *style_and_grammar_checker.pdf*. Retrieved Diciembre 2009, from http://www.danielnaber.de/languagetool/download/style_and_grammar_checker.pdf

SAX Project. (2004). *SAX*. Retrieved Diciembre 2009, from http://www.saxproject.org/

Sun Microsystems. (n.d.). *Developer Resources for Java Technology*. Retrieved Diciembre 2009, from http://java.sun.com/

Temperlay, D., Sleator, D., & Lafferty, J. (2004). *Link Grammar*. Retrieved Diciembre 2009, from http://www.link.cs.cmu.edu/link/

TeXlapse-team. (n.d.). *TeXlipse homepage - LaTeX for Eclipse*. Retrieved Diciembre 2009, from http://texlipse.sourceforge.net/

Texmaker: Free LaTeX Editor. (n.d.). Retrieved Diciembre 2009, from http://www.xm1math.net/texmaker/

TeXnic Center. (2008). Retrieved Diciembre 2009, from http://www.texniccenter.org/

TeXShop. (n.d.). Retrieved Diciembre 2009, from http://www.uoregon.edu/~koch/texshop/

The Eclipse Foundation. (n.d.). *Eclipse.org home*. Retrieved Diciembre 2009, from http://www.eclipse.org/

The Eclipse Foundation. (n.d.). *Visual Editor Project*. Retrieved Diciembre 2009, from http://www.eclipse.org/vep/

The LaTeX project public license. (n.d.). Retrieved Diciembre 2009, from http://www.latex-project.org/lppl/

The LaTeX3 project. (n.d.). Retrieved Diciembre 2009, from http://www.latex-project.org/latex3.html

Tigris.org. (n.d.). *subclipse.tigris.org*. Retrieved Diciembre 2009, from http://subclipse.tigris.org/

W3C Document Object Model. (n.d.). Retrieved Diciembre 2009, from http://www.w3.org/DOM/

W3C. (n.d.). *Extensible Markup Language*. Retrieved Diciembre 2009, from http://www.w3.org/XML/

World Wide Web Consortium. (n.d.). *World Wide Web Consortium (W3C)*. Retrieved Diciembre 2009, from http://www.w3.org/

APÉNDICES

A) GLOSARIO

Arquitectura modular: arquitectura basada en diferentes módulos o componentes, que estando cada uno destinado a una labor concreta, pueden trabajar conjuntamente con un fin común más general.

Comando: es una instrucción u orden que el usuario proporciona a un sistema informático. Suele admitir parámetros (argumentos) de entrada, lo que permite modificar el comportamiento predeterminado del comando. En nuestra aplicación está estrechamente relacionado con los tags.

Distribución: conjunto de aplicaciones o paquetes software destinado a satisfacer las necesidades de un grupo específico de usuarios.

DOM (Document Object Model): convención multiplataforma e independiente del lenguaje que permite representar e interactuar con objetos en documentos HTML, XHTML y XML a través de la interfaz pública especificada en su API.

Fichero fuente: fichero que contiene una colección de declaraciones en algún lenguaje de programación y que permite especificar a un ordenador las acciones que debe realizar.

GUI (Graphical User Interface): artefacto tecnológico de un sistema interactivo que posibilita, a través del uso y la representación del lenguaje visual, una interacción amigable con un sistema informático.

IDE (Integrated Development Environment): aplicación software que proporciona a los programadores informáticos un paquete de herramientas integral para el desarrollo de software. Generalmente este paquete de herramientas puede ser ampliado a través de plugins.

Macros (macroinstrucciones): son instrucciones complejas formadas por un grupo de instrucciones más simples, y que se almacenan para poder ser ejecutadas de forma secuencial mediante una sola llamada u orden de ejecución. Permiten la automatización de tareas repetitivas.

Paquete: programas y material de instalación agrupados para distribuirlos conjuntamente.

Plugin: programa informático que funciona a modo de complemento de una aplicación software anfitriona, en la que una vez integrado y mediante la interacción que realiza con esta a través de la API, permite utilizar nuevas funciones, generalmente muy específicas.

SAX (Simple API for XML): Es una API de acceso secuencial que permite leer datos desde documentos XML. Los parsers que utilizan esta API están dirigidos por eventos, los cuales son recogidos y tratados por una serie de métodos que debe implementar el programador. No necesita por tanto guardar en un árbol el contenido del archivo XML como hace el DOM.

SWING: framework orientado a interfaces gráficas de usuario de tipo Modelo-Vista-Controlador para Java. Es independiente de la plataforma, extensible y personalizable.

Tag/etiqueta: marca con tipo que delimita una región en la cual lenguajes como LaTeX y XML almacenan información.