UNIVERSIDAD POLITECNICA DE VALENCIA

Escuela Técnica Superior de Ingeniería Informática

UNIVERSIDAD POLITÉCNICA DE VALENCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

PROYECTO FINAL DE CARRERA

PROCESO DE ETIQUETADO DE UN CORPUS DIGITAL

ALUMNA: Laura Vilar Bohigues

DIRECTORES: María José Labrador

Salvador Pons

Diciembre 2010

Dedicatorias

Dedico este proyecto a mi familia, a mis directores de proyecto María José Labrador y Salvador Pons, y en general a todos los profesores que he tenido, sin cuyas enseñanzas nunca habría podido llegar hasta este punto.

Mención especial merece Lorenzo Toldrá de la empresa Wera Systems, por sus valiosos consejos durante el desarrollo de este trabajo.

Contenidos

1. Introducción

1.1. Motivación

El objetivo principal del presente proyecto consiste en el diseño e implementación de una biblioteca de clases que sea capaz de etiquetar un corpus según un etiquetado *morfosintáctico y conversacional* primero, y de realizar consultas expertas sobre dicho corpus en segundo lugar.

La motivación en este proyecto proviene principalmente de tres vertientes. Por una parte, está la contribución a un proyecto real existente, donde mi trabajo supondrá una ampliación del sistema actual y sentará las bases para algunas de las fases posteriores que los usuarios finales han definido. Por otro lado, me motiva el tener la oportunidad de trabajar con tendencias tecnológicas muy actuales, pues estas son necesarias para garantizar una correcta integración con el proyecto. Por último, este trabajo supone para mí un reto personal, ya que el marco entorno al que gira, el *Lenguaje Natural* y su computación, es desconocido para mí.

1.2 Objetivos

- Estudio de los etiquetadores disponibles actualmente en red, análisis de sus funcionalidades y elección de uno de ellos.
- Diseño e implementación del sistema que permita aplicar un etiquetado morfosintáctico (utilizando el etiquetador seleccionado) y de fenómenos conversacionales al corpus.
- Corrección de algunas *deficiencias* (alargamiento vocálico, pronombres enclíticos, etc.)
- Creación de documentos *XML* para almacenar de forma estructurada cada transcripción etiquetada del corpus.
- Diseño e implementación de un motor de búsqueda con *LINQ to XML*.
- Evaluación de resultados del proceso de etiquetado inicial.

1.3 Contexto

1.3.1 La lingüística de Corpus

Para poder diseñar un sistema que interactúe con un corpus lingüístico es necesario tener claro qué es un corpus y porqué surge la necesidad de procesarlo.

Como primer paso recurrimos a las definiciones de corpus y lingüística de corpus que nos proporciona la Wikipedia: Un *Corpus lingüístico* es un conjunto, normalmente muy amplio, de ejemplos reales de uso de una lengua. Estos ejemplos pueden ser textos (típicamente), o muestras orales (normalmente transcritas). Se denomina *lingüística de corpus* a la subdisciplina de la lingüística que estudia la lengua a través de estas muestras.

Estas simples definiciones, sencillas de entender, son el punto de partida para comprender el contexto que envuelve a este proyecto.

Aun así, es interesante destacar algunas características adicionales de los corpus lingüísticos como:

- Deben estar sistematizados según determinados criterios.
- Deben ser suficientemente extensos en *amplitud y profundidad* para ser representativos.
- No son repertorios de textos, sino repertorios de datos lingüísticos, recopilados con fines de investigación y análisis.
- Deben estar dispuestos de tal modo que puedan ser procesados mediante ordenador.

Es esencial para el proyecto que estamos realizando tener en cuenta esta última característica, pues precisamente nuestro objetivo es procesar el corpus con el fin de obtener resultados varios y útiles para la descripción y el análisis del mismo.

Pero, ¿qué utilidad se puede extraer de la lingüística de corpus? No hay una respuesta directa a esta pregunta, pues esta disciplina a penas a recorrido un pequeño camino de su historia, y se espera que en un futuro cercano experimente un notable auge acompañado de una

expansión en distintas direcciones. Las previsiones de los lingüistas apuntan a que serán la pragmática, la sociolingüística, la lexicografía y el estudio de la sintaxis, la morfología y la fonética, entre otros, los que mayor beneficio obtengan de la explotación de los corpus.

Sin embargo, esta disciplina lingüística presenta una serie de limitaciones muy a tener en cuenta. En primer lugar, una lengua es un sistema de comunicación siempre abierto a nuevas modalidades y a nuevas adquisiciones, por lo que nunca se puede dar por cerrado a menos de que no queden sujetos que se valgan de él para comunicarse y deba darse por muerto. Otra de las limitaciones es la temporalidad, pues la vigencia de un corpus está sujeta a los límites de la supervivencia de las muestras recopiladas, y a la evolución de la lengua con el paso del tiempo. El siguiente problema lo encontramos reflexionando sobre si los datos lingüísticos que recopila un corpus son capaces de representar la totalidad de palabras y usos gramaticales que encierra un idioma, lo cual es prácticamente imposible que suceda en un repertorio limitado. Incluso podríamos encontrar un problema en una de las ventajas que caracterizan a los corpus, pues la cantidad de datos que representa es enorme, por lo que probablemente las formas lingüísticas más comunes en el ámbito de referencia del corpus se encuentren en decenas de miles de entradas.

Dejando atrás estas limitaciones, cabe destacar que el concepto de corpus se ha creado ligado al ordenador. Sin este último sería casi imposible obtener y clasificar los datos de manera que resultaran útiles y accesibles de manera sistematizada y abundante.

Repasemos ahora brevemente la historia de los corpus lingüísticos. Como acabamos de indicar, sin la presencia de los ordenadores nunca se hubiera podido concebir la idea de corpus tal y como es conocida hoy en día. Las posibilidades de procesamiento de los ordenadores han sido decisivas para poder fijas algunos de los parámetros que distinguen a un corpus.

El nacimiento de los corpus se sitúa en la década de los sesenta, donde empiezan a recopilarse datos formando varios corpus como el Brown Corpus del inglés americano o el Survey of English Usage, impulsados por la aparición de los ordenadores. Sin embargo, no es hasta 1975 cuando el lingüista Jan Svartvik propone por primera vez que se transcriba y digitalice un corpus oral. Estos corpus fueron pioneros y

por ello se les denomina corpus de primera generación. La segunda fase de la lingüística de corpus se alcanza en la década de los 80, marcada por los adelantos en la potencia de los ordenadores y por la posibilidad de captar ópticamente textos escritos por medios mecánicos. Los corpus creados en esta fase aumentan considerablemente el tamaño o volumen de los datos susceptibles de ser procesados automáticamente. En la actualidad la idea de corpus se ha extendido a muchos idiomas, y el tamaño de los recursos de cada corpus se ha visto incrementado. Por esta razón los estudiosos los corpus actuales como pertenecientes a la tercera generación.

Respecto a los corpus del español, algunos de los que están disponibles en la actualidad son:

- CREA: Corpus de referencia del español actual
- Cumbre: Corpus lingüístico del español contemporáneo
- Lexesp: Léxico informatizado del español

1.3.2 La Lingüística Computacional

Después de haber visto qué es la lingüística de corpus desde el punto de vista de la lingüística, es necesario conocer el enfoque que los investigadores informáticos tienen respecto al procesamiento de los datos lingüísticos que forman los corpus.

La disciplina del estudio del lenguaje que tiene en cuenta la computación es conocida como Lingüística Computacional o como Procesamiento del Lenguaje Natural y se encarga de investigar mecanismos que permitan una comunicación hombre/máquina más fluida y menos rígida que los lenguajes formales. El objetivo de ésta es desarrollar una teoría computacional del lenguaje usando las nociones de algoritmos y estructuras de datos desde la informática.

Para simular un comportamiento lingüístico humano es necesario conocer las estructuras del lenguaje y conocer el universo de discurso.

Existen dos motivaciones para desarrollar modelos del lenguaje:

- Científica: Se pretende profundizar en el estudio del lenguaje para mejorar la comprensión y producción del mismo, además

de desarrollar sistemas formales y formalismos gramaticales (GPSG, LFG, HDPS, …)

- Tecnológica: Se pretende procesar el lenguaje mediante modelos que no siguen necesariamente las pautas humanas. En esta motivación se pretende también dar soporte a aplicaciones (traducción automática, extracción de información, resumen automático, etc.)

Una de las aproximaciones de la lingüística computacional está basada en conocimiento, y se caracteriza por la codificación manual del conocimiento lingüístico y el afinamiento del sistema. En cambio otra está basada en métodos de aprendizaje, ya que utiliza técnicas de aprendizaje y métodos estadísticos, aprende conocimiento lingüístico desde corpus anotados o no y por último aprende reglas desde la interacción con el usuario.

En esta disciplina se reconocen las siguientes fases diferenciadas de análisis: Análisis morfológico-léxico, análisis sintáctico, análisis semántico y análisis contextual o de función pragmática.

La fase de análisis morfológico-léxico consiste en convertir una secuencia de caracteres de entrada en una secuencia de unidades léxicas. Para ello es necesario usar diccionarios y reglas morfológicas.

El proceso de estructurar en forma de árbol una secuencia de unidades léxicas se denomina análisis sintáctico.

En el análisis semántico se parte de las estructuras de árbol para obtener la forma lógica que representa el significado.

Por último, en la fase de análisis contextual o de función pragmática logramos a partir de la forma lógica y en función del contexto la interpretación final.

El mayor problema al que se enfrenta la lingüística computacional es el de la ambigüedad, pues en muchos casos se producen varias interpretaciones sobre un mismo objeto. Para resolver este problema es necesario definir qué estrategias se deben seguir para determinar la interpretación deseada en función del tipo de ambigüedad.

Las ambigüedades que nos podemos encontrar son de distintos tipos, y cada una de ellas se debe resolver en una fase concreta del análisis.

En concreto, las ambigüedades léxicas aparecen en la fase de análisis léxico, las estructurales y las léxicas de tipo categorial son propias de la fase de análisis sintáctico, en cambio en el análisis semántico es probable que nos encontremos con ambigüedades de carácter léxico, de ámbito de cuantificación y de función contextual. Por último, las de carácter referencial son halladas en el análisis pragmático.

Después de esta vista general por las diversas fases del análisis vamos a investigar un poco más a fondo las fases que vamos a tratar en nuestro proyecto.

El análisis léxico es el proceso que transforma el texto de entrada (caracteres) en una secuencia de unidades significativas (unidades léxicas) con información asociada. Las características léxicas, que dependerán de la aplicación final, se dividen en cuatro categorías:

- Características morfológicas: Raíz, lema, género, número, persona, tiempo, modo, etc.

- Categoría morfosintáctica o gramatical: Nombre, verbo, adjetivo, adverbio, preposición, etc.

- Información semántica: Sentido, glosa.

- Pronunciación de la palabra.

La Segmentación del texto consiste en dividir el texto en unidades lingüísticamente significativas: grafemas (letras), palabras u oraciones. En cambio, la segmentación en palabras o Tokenización es el proceso que divide la secuencia de caracteres de un texto en palabras o tokens (unidades léxicas), y la segmentación de oraciones es el proceso que divide el texto en oraciones.

Sin embargo, la segmentación tiene una serie de problemática asociada, y es que es por una parte dependiente del lenguaje según la definición de símbolos (logográficos, silábicos y alfabéticos) y del

marcado de límites de palabras y oraciones, por otra parte es dependiente del juego de caracteres y además es dependiente de la aplicación y de cada corpus.

En el caso concreto de la segmentación en palabras, también conocida como tokenización son dos los aspectos a tener muy en cuenta: Por una parte, la estructura tipográfica de la palabra (aislada, la palabra no se divide en unidades más pequeñas, aglutinativa, división clara en morfemas, flexiva, los límites entre morfemas no están claros, más de un significado gramatical) que en la mayoría de los lenguajes muestran trazas de los 3 tipos de estructura aunque tengan una tendencia por alguno de ellos, y por la otra parte, si es un lenguaje segmentado o no, es decir, si separa las palabras por blancos o se escribe como una sucesión de palabras sin límite entre ellas.

Los lenguajes segmentados por espacios se caracterizan por: utilizar símbolos de puntuación, que normalmente son tokens separados, el uso de abreviaturas, el uso de comillas, apostrofes o el genitivo del inglés, la utilización de palabras compuestas y la utilización de expresiones multipalabra (locuciones adverbiales, fechas, horas, etc.).

En cambio, en los lenguajes no segmentados, la tokenización es dependiente de la lengua, es necesario un diccionario y pueden existir varias segmentaciones válidas.

Si analizamos el proceso de segmentación en oraciones, observamos que las oraciones se delimitan normalmente por símbolos de puntuación como el punto, la exclamación o la interrogación. Sin embargo, aquí se nos presenta algunos problemas, como ambigüedad, sentencias anidadas, citas, entidades que contienen algún símbolo delimitador, etc. Además, este proceso es dependiente del corpus para el que se diseña.

Vistas las características principales de la segmentación y parte de su problemática pasemos a ver de qué forma nos interesa contar con información léxica en el contexto de la lingüística computacional.

Empecemos definiendo lexicón, que es un repositorio de información léxica, donde a cada unidad léxica se asocia un conjunto de información (categoría sintáctica, interpretación semántica a nivel

léxico) y diversas propiedades (morfológicas, sintácticas, semánticas). En la construcción de un lexicón encontramos dificultades en la representación léxica, el volumen de la información, la segmentación de la oración en palabras, la homonimia, la polisemia y el domino de la aplicación.

Pero, ¿qué tipo de información léxica interesa realmente procesar? Interesa el marcado de la categoría sintáctica con una etiqueta asociada a cada grupo de unidades léxicas, conocer las características sintácticas de concordancia, la información morfológica y semántica y algunas otras informaciones como restricciones seleccionales, el tipo de los complementos, las preposiciones admisibles, etc.

Para adquirir conocimiento léxico, en lingüística computacional se tienen en cuenta dos aproximaciones:

- Adquisición manual. Consiste en la construcción del lexicón a cargo del lexicógrafo haciendo uso de herramientas que le faciliten la estructuración de la información léxica. Esta visión no es adecuada para lexicones de gran tamaño, pues implica muchísimo trabajo.

- Adquisición semi-automática. Dentro de esta aproximación existen dos vertientes, dependiendo de las características de la base a analizar.

    - A partir de diccionarios convencionales en soporte magnético. En este proceso los diccionarios se procesarán y derivarán en Bases de Datos Léxicas. La extracción de información se realizará desde una Base de Datos Léxica para la representación de ésta en el lexicón.

    - A partir de córpora en soporte magnético acompañado de herramientas de análisis y selección de subconjuntos de tales corpus.

La morfología estudia cómo se construyen las palabras a partir de los monemas (morfemas y lexemas), que son unidades más pequeñas. Aunque el análisis morfológico en este proyecto está a cargo de un etiquetador externo, es conveniente conocer el procedimiento de composición de palabras. Así pues, existen tres técnicas diferentes de composición: La inflexión, donde se combina un lexema de una palabra con morfemas gramaticales y la palabra resultante es de la misma categoría sintáctica, la derivación, donde se combina un lexema con morfemas gramaticales para formar una palabra de categoría sintáctica diferente o de significado diferente, y la composición, donde se combinan varios lexemas resultando una palabra de la misma categoría gramatical.

El etiquetado morfosintáctico o morfológico es el proceso que asigna a cada uno de los 'tokens' de un texto etiquetas sobre las partes de la oración, sin tener en cuenta sus relaciones sintácticas. Las partes de la oración representan la categoría gramatical de cada palabra y pueden incorporar información morfológica. Las palabras, vistas de forma aislada, por lo general, son ambiguas respecto a su categoría gramatical, sin embargo, la categoría gramatical de las palabras puede desambiguarse dentro de un contexto.

Para el análisis morfológico de una palabra, se descompone la palabra en una secuencia de morfemas y se obtiene el lema asociado para poder acceder a la información léxica. Hay que tener en cuenta que almacenar todas las entradas en un diccionario es ineficiente.

Para un análisis eficaz, un analizador morfosintáctico debe tener componentes como una detección multi-palabras (puede usarse diccionarios especiales como los de locuciones adverbiales), una detección de Entidades Nombradas (Name Entity) que identifique las secuencias de palabras que pueden constituir una entidad.

En lingüística computacional existen dos aproximaciones derivadas del análisis morfológico, una basada en conocimiento y otra basada en métodos de aprendizaje.

Las características de la que está basada en conocimiento son:

- Debe existir una codificación manual de las reglas, tanto las que se definen con expresiones regulares como los patrones para el análisis de ambigüedades.
- Esta aproximación tiene el inconveniente de que para la definición de la gramática es costosa.
- Otro inconveniente es que es dependiente de la lengua para la que ha sido diseñada.
- Con esta aproximación se obtienen altas prestaciones.
- La gramática tiene una gran riqueza expresiva.

Por su parte, los métodos basados en aprendizaje, construyen el modelo a partir de características que aparecen en el contexto.

Uno de los métodos basados en aprendizaje, es el método del conjunto de desarrollo, constituido por tres fases. En la primera, se divide el corpus en tres partes (Entrenamiento, desarrollo o ajuste y prueba). A continuación se ajustan los parámetros del modelo, aprendiendo en el conjunto de entrenamiento y probando sobre el de desarrollo. La última fase consiste en evaluar el mejor modelo obtenido en la fase anterior sobre el conjunto de prueba. Una forma de medir la precisión consiste en aplicar la fórmula Palabras Acertadas / Total de Palabras.

Otro método, el de la validación cruzada, primero divide aleatoriamente el corpus en n partes, para a continuación ejecutar n veces la secuencia: n-1 partes como entrenamiento y 1 parte como prueba, donde la parte de prueba es diferente en cada ejecución. Para la evaluación del modelo se calcula la precisión media de las distintas ejecuciones.

2. El grupo Va.les.co

2.1 Descripción

El grupo Val.Es.Co (Valencia Español Coloquial) es un grupo de investigación consolidado de la Universidad de Valencia que se ha dedicado al estudio del español hablado desde 1990. En su seno se han leído una docena de tesis doctorales y dos docenas de trabajos de investigación y ha recibido numerosos reconocimientos dentro del área de investigación de Lengua Española.

Una de las tareas del grupo ha sido la elaboración de un corpus de conversaciones del español hablado. Este corpus, que comenzó a ser recogido a principios de los años noventa, consta de más de trescientas horas de grabación, de las que más de la mitad ha sido transcrita, y que, en conjunto, comprende más de medio millón de palabras. Sin embargo, su aprovechamiento, dada la fecha de grabación de las conversaciones, ha sido hasta ahora limitado. Para actualizar las conversaciones y convertirlas en una base de datos accesible por internet se les concedió en el año 2008 un proyecto de investigación gracias al cual se digitalizaron las primitivas conversaciones analógicas, se transcribió buena parte de las cintas originales y se creó una página web, en principio experimental, en la que se podía realizar una búsqueda sencilla por palabras, de la que resultan sus concordancias o los párrafos en que aparece.

Esta era la primera parte de un proyecto que pretende mejorar la estructura de su corpus digital en dos fases adicionales: en una segunda fase, se pretende etiquetar el corpus según un etiquetado morfosintáctico primero, y conversacional después; en una tercera y última fase, alinear dicho corpus con los audios correspondientes, de modo que cada búsqueda pueda devolver, no solo la transcripción, sino también las cadenas de sonidos correspondientes.

2.2 Transcripciones

Cada una de las conversaciones del corpus que ha sido transcrita, ha pasado por el filtro de varios investigadores que han incluido marcas conversacionales, tales como la entonación de los

hablantes. Las transcripciones son almacenadas en documentos Word (.doc), con su identificador como nombre. Cada transcripción tiene asociada una ficha técnica donde se detallan los datos relativos al lingüista encargado de la transcripción, a los hablantes (o interlocutores) que intervienen en ella y a las circunstancias en las que se produjo la grabación.

El modelo de ficha técnica es el siguiente:

FICHA TÉCNICA

a) **Investigador:** Nombre y apellidos del investigador
   Clave de la investigación: código numérico asociado
b) **Datos identificadores de la grabación**:
   - **Fecha de la grabación:** dd-mm-aaaa
   - **Tiempo de la grabación:** x minutos
   - **Lugar de grabación:**

c) Situación comunicativa:
   - **Tema o materia:** [cotidiano | académico | otro]
   - **Propósito o tenor funcional predominante:** **[**Interpersonal | Transaccional]
   - **Tono: [**Informal | Formal]
   - **Modo o canal:** [oral | escrito]

d) **Tipo de discurso registrado:** conversación

e) Técnica de grabación:
   - **Conversación libre**
     [Observador participante | Observador no participante]

f) Descripción de los participantes:
   - **Número de participantes: Clave:**
     activos:
     pasivos:
   - **Tipo de relación que los une:**
   - **Sexo**:
   - **Edad:**
     18-25
     26-55
     >55
   - **Nivel de estudios:**
     analfabetos:
     primarios:
     secundarios:
     medios:
     superiores:

- **Profesión:**
- **Residencia o domicilio:**
- **Nivel sociocultural:**
  alto:
  medio:
  bajo:
- **Lengua habitual:**
  monolingüe en castellano:
  bilingüe:
  - **Grado de prototipicidad coloquial: [**Coloquial prototípico | Coloquial periférico]

En el archivo .doc que representa la transcripción, cada intervención empieza con la clave correspondiente al interlocutor de dicha intervención y a continuación la transcripción. El símbolo ':' es el separador entre las dos partes nombradas. Así pues, la estructura de una transcripción completa será la siguiente:

Clave_interlocutor1:transcripción_de_la_intervención

Clave_interlocutor2:transcripción_de_la_intervención

Clave_interlocutor1:transcripción_de_la_intervención

Clave_interlocutor2:transcripción_de_la_intervención

Clave_interlocutor1:transcripción_de_la_intervención

Clave_interlocutor2:transcripción_de_la_intervención

Clave_interlocutorN-1:transcripción_de_la_intervención

Clave_interlocutorN:transcripción_de_la_intervención

Las marcas conversacionales que utiliza el grupo se intercalan en la transcripción literal de las palabras y sonidos que figuran en la grabación de la conversación.

Lista de símbolos:

§ <sucesión inmediata
= <mantenimiento del turno
[<inicio solapamiento
] <final solapamiento
- <reinicio
/ <pausa corta
// <pausa intermedia
/// <pausa larga
(*") silencio, donde *=número
↑ entonación ascendente
↓ entonación descendente
→ entonación mantenida
MAYÚSCULAS pronunciación marcada
si la ba te a do pronunciación silabeada
(()) fragmento indescifrable
((palabra**))** transcripción dudosa
((...)) interrupción de la grabacion
(en**)**tonces reconstruido
pa'l fonética sintáctica
º()º bajo
(RISAS) entre risas
en cursiva estilo directo

3. Requisitos técnicos

Al ser este trabajo una ampliación de un proyecto ya existente, y al estar pensado para servir de base a otras fases futuras de dicho proyecto, es necesario que tanto las herramientas de desarrollo como las características del sistema desarrollado sean compatibles con las utilizadas para las otras fases del proyecto.

Los desarrolladores de la fase inicial del proyecto, que además son los encargados del mantenimiento del sistema y de la base de datos utilizan tecnología Microsoft actualizada. Por este motivo, la implementación del proyecto se llevará a cabo con el lenguaje de programación c# y bajo el entorno Visual Studio 2010.

Como ya se ha visto en la sección anterior, el corpus a etiquetar está almacenado en documentos .doc, por lo que la manipulación de los mismos por código es imprescindible. Para ello será necesario incluir referencias a objetos de Microsoft Office en la solución generada con el Visual Studio.

Para estandarizar el almacenamiento de los datos procesados se crearán documentos en formato XML con los datos presentes en los documentos .doc ya estructurados y con la información derivada de la fase de etiquetado morfológico.

Para la segunda parte de este trabajo, la búsqueda avanzada, el lenguaje de consulta será LINQ to XML, un lenguaje de consulta muy potente que cuenta con la característica de que es orientado a objetos, a diferencia de la mayoría de lenguajes de consulta que son estructurados.

4. Análisis de los etiquetadores morfológicos existentes

4.1. Etiquetadores para el español

El primer objetivo de este trabajo, no es otro que el de investigar qué programas o herramientas existen actualmente en la red que sean capaces de procesar datos lingüísticos y aportar a éstos información relevante. En nuestro caso, las herramientas deben ser capaces de aportar información morfológica al corpus. Es decir, necesitamos un programa que reciba como entrada los datos lingüísticos de un corpus no anotado y que devuelva como salida la información léxica correspondiente a cada elemento del corpus. Es imprescindible que la herramienta esté disponible para procesar corpus en español.

A continuación detallaré las características más relevantes de los etiquetadores encontrados.

Thera

Thera, spin-off del Centre de Llenguatge i Computació de la Universitat de Barcelona, cuenta con la experiencia de más de 20 años de investigación en el área de la Lingüística Computacional, y con más de 6 años en el desarrollo de aplicaciones en las tecnologías del conocimiento y en la gestión de la información basadas en la ingeniería lingüística. En el sitio web de la empresa, encontramos tres demos disponibles (figura 1) referentes al análisis morfológico: Un lematizador, un Flexionador y un Etiquetador. Para nuestros intereses, el etiquetador es mucho más interesante que los otros dos.

Figura 1. Lista de demos disponibles en el sitio web del proyecto thera.

Para probar su funcionamiento accedemos a la demo del Etiquetador, lo que nos transporta a otra sección de la web con un formulario en el que nos piden que introduzcamos una frase. En la figura 2 podemos observar dicho formulario.

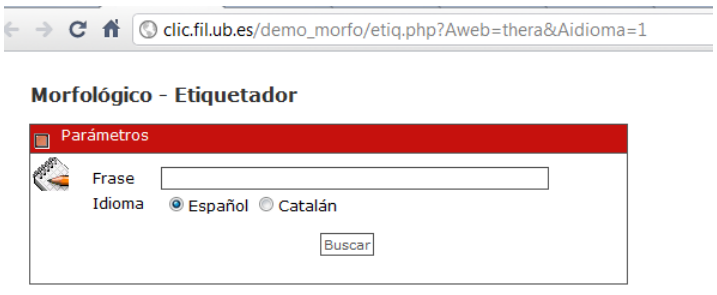

Figura 2. Demo del Etiquetador morfológico Thera.

Nos damos cuenta en este momento de que esta herramienta no va a ser útil para nuestro trabajo, pues es impensable que se

introduzcan todas las frases de nuestro corpus en esta interfaz. Necesitamos acceder a la herramienta, no a la demo. Lamentablemente ésta no está disponible, pues los analizadores lingüísticos son utilizados por esta compañía como módulos auxiliares para sus productos comerciales. Debemos pues descartar este etiquetador, no sin antes probar su precisión como se muestra en la figura 3.

Figura 3. Ejemplo de uso del etiquetador Thera.

System for automatic morphological analysis of Spanish

Este etiquetador, desarrollado por Grigori Sidorov, Alexander Gelbukh, Francisco Velázquez y Liliana Chanona, realiza la lematización y ofrece información gramatical de cada forma de la palabra en la frase. El sistema es un archivo ejecutable para Windows, y el DLL está disponible bajo petición a los autores.

El funcionamiento del sistema es de unos 5KB por segundo en ordenadores con Pentium 4. Utiliza el BDE para acceder a los datos. El diccionario tiene un tamaño de aproximadamente 25.000 palabras. Además, el sistema contiene un archivo de texto "complex.dic" con las palabras compuestas (a_partir_de,etc.). Ese mismo archivo se puede utilizar como un diccionario de usuario para una sola palabra.

Los términos de Licencia, indican que se puede utilizar libremente esta herramienta para propósitos académicos, pero que no hay ningún tipo de garantía. Además se debe informar a los desarrolladores del uso del programa y citar el correspondiente artículo científico en las publicaciones donde se haya hecho uso de la herramienta. En la figura 4 tenemos la referencia a dicho paper.

A. Gelbukh, G. Sidorov. Approach to construction of automatic morphological analysis systems for inflective languages with little effort. In: Computational Linguistics and Intelligent Text Processing (CICLing-2003), Lecture Notes in Computer Science, N 2588, Springer-Verlag, 2003, pp. 215–220.

Figura 4. Referencia a incluir en los proyectos en los que se utilice esta herramienta.

Si descargamos el sistema estamos aceptando la licencia anteriormente comentada. La última versión lanzada data del 18 de abril de 2007.

El fichero de entrada debe ser un archivo de texto (con codificación DOS o ANSI), mientras que la salida obtenida es un archivo con el mismo nombre que el de la entrada más el prefijo "c_".

FreeLing

Freeling es una herramienta de código abierto de análisis lingüístico, publicada bajo la Licencia Pública General GNU de la Free Software Foundation.

Este paquete consiste en una biblioteca que proporciona servicios de análisis de idiomas. Freeling está diseñado para ser utilizado como una biblioteca externa de cualquier aplicación que requiera este tipo de servicios. Sin embargo, un simple programa inicial también se incluye

como una interfaz básica para la biblioteca, que permite al usuario analizar archivos de texto desde la línea de comandos.

Los principales servicios ofrecidos por esta biblioteca son:

- Un tokenizador de texto
- Sentencia de separación
- Análisis morfológico
- Tratamiento de sufijo, retokenización de los pronombres enclíticos
- Reconocimiento de varias palabras flexible
- Contracción de la división
- Probabilística predicción de categorías de palabras desconocidas
- Detección de entidades nombradas
-Reconocimiento de fechas, números, relaciones, dinero, y las magnitudes físicas (velocidad, peso, temperatura, densidad, etc.)
- Clasificación de entidades nombradas
- WordNet: anotación, sentido, base y desambiguación
- Dependencia basada en reglas de análisis
- Resolución correferencia nominal

Actualmente los idiomas admitidos son el español, catalán, gallego, italiano, inglés, galés, portugués y asturiano.

La biblioteca FreeLing está enteramente implementada en C++, utilizando plantillas estándar STL, por lo que se puede compilar en casi cualquier plataforma.

A pesar de esto, el hábitat natural de FreeLing son sistemas Linux/Unix. Según los autores de Freeling, algunos usuarios han conseguido con éxito la compilación bajo MacOS y Windows, utilizando MinGW, Cygwin o MSVC.

Las versiones de Windows se proporcionan por conveniencia, pero sin control ni apoyo por parte de los desarrolladores.

En la web del proyecto está disponible una demo del sistema.

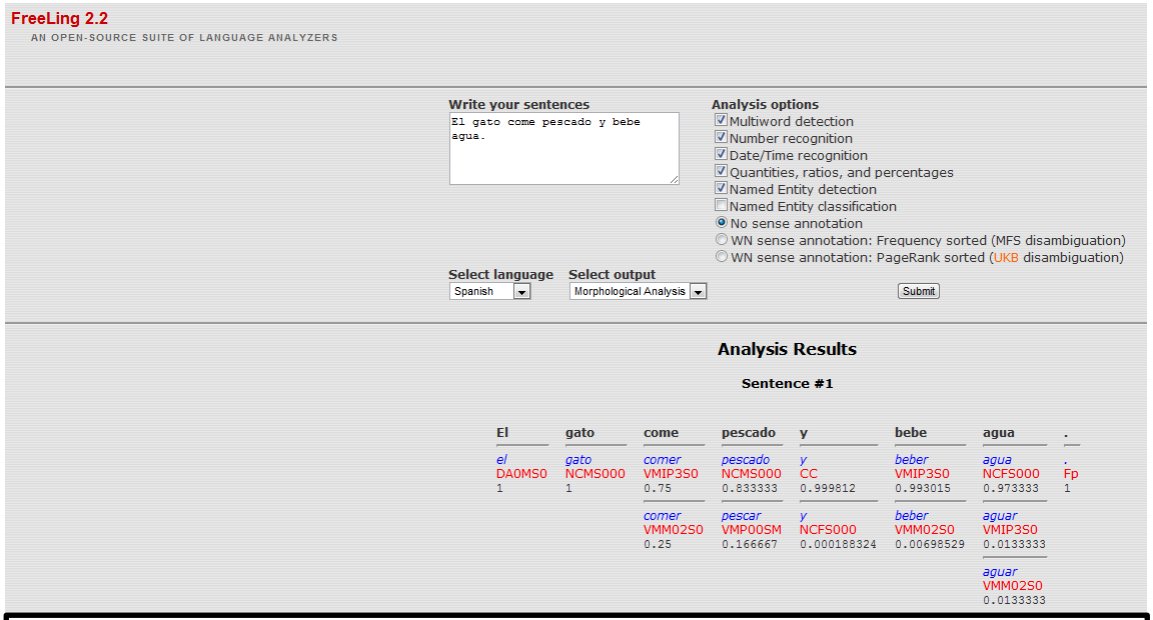

Figura 5. Ejemplo de uso de la demo de FreeLing.

En el capítulo 4.2 se comentará con más detalle el funcionamiento de este etiquetador, pues es el que ha sido seleccionado para ser utilizado en nuestro trabajo.

NLTK

NLTK es un Kit de herramientas para el lenguaje natural o, más comúnmente, es un conjunto de bibliotecas y programas de procesamiento de lenguaje natural (NLP) para el lenguaje de programación Phython. NLTK incluye demostraciones gráficas y datos de muestra. Se acompaña de una extensa documentación, incluyendo un libro que explica los conceptos subyacentes a las tareas de procesamiento del lenguaje con el apoyo de la caja de herramientas. NLTK es ideal para estudiantes que están aprendiendo la disciplina del procesamiento del lenguaje natural o alguna otra área afín, incluyendo la lingüística empírica, la ciencia cognitiva, la inteligencia artificial, la recuperación de información y el aprendizaje automático. NLTK ha sido utilizada con éxito como herramienta de enseñanza, como herramienta de estudio individual, y como una plataforma para la construcción de prototipos y sistemas de investigación. El proyecto NLTK está dirigido por Sreven Bird, Eduardo López, y Ewan Klein, y está disponible para distribuciones Windows, Mac OSX y Linux.

Para nuestro trabajo, descartamos el uso de esta biblioteca por estar escrita en phyton.

LingPipe

LingPipe es un juego de herramientas de procesamiento de texto mediante técnicas de lingüística computacional. lingPipe se utiliza para tareas como: Buscar los nombres de las personas, organizaciones o lugares en las noticias, clasificar automáticamente los resultados de la búsqueda de Twitter en categorías o sugerir la ortografía correcta de las consultas.

La arquitectura LingPipe está diseñada para ser eficiente, escalable, reutilizable, y robusta. Sus características más destacadas son el uso del API de Java con código fuente y con pruebas unitarias, la capacidad multi-idioma, el multi-dominio, los modelos de multi-género, la formación con nuevos datos para nuevas tareas, conseguir un n-mejor rendimiento con las estimaciones de confianza estadística, formación en línea, modelos seguros para subprocesos simultáneos y decodificadores para lectura exclusiva y escritura (CREW), etc.

LingPipe License Matrix				
Type	Royalty Free	Developer	Startup	Enterprise Server
Cost	Free	$450/server	Starts at $9,500/year/site*	$40,000/server
Data processed must be freely available	yes	no	no	no
Linked software must be freely available	yes	no	no	no
Research use	yes	yes	yes	yes
Production use	limited	no	yes	yes
Support Level	none	2 email	10 email	unlimited phone/email+
Upgrades	no	no	yes	yes**
Indemnification	no	no	no	yes**
Term	perpetual	perpetual	1 year	perpetual
License	View License	View License	contact us	contact us
All download links are to the same files:	**Tarball (68.1MB)**	**Jar (981KB)**		
Purchase License	Free	Add to Cart	Contact Alias-i	Contact Alias-i
Download	Download	Download	Download	Download

Figura 6. Formas de obtener la herramienta LingPipe.

4.2. Etiquetador Freeling

4.2.1 Características

Aunque en la sección anterior se han nombrado algunas de las características de este etiquetador, es necesario introducir mayor nivel de detalle. En concreto, es interesante conocer qué información lingüística trae.

La versión distribuida incluye diccionarios morfológicos. Aunque los diccionarios para algunas lenguas latinas pueden parecer pequeños, hay que tener en cuenta que muchas formas son reconocidas gracias a un módulo de análisis de sufijos poderoso, capaz de detectar pronombres enclíticos, formas verbales, sufijos diminutivos/aumentativos de sustantivos y adjetivos.

El diccionario de español contiene más de 550.000 formularios correspondientes a más de 76.000 combinaciones lema-PoS. Este diccionario morfológico se obtuvo de una fuente determinada, y tiene su licencia de distribución. Antes de utilizarlo se debe comprobar el archivo de copia en el archivo comprimido para asegurarse de que no se está violando ningún término de la licencia.

Esta versión también incluye diccionarios de sentido basado en WordNet.

Las características detalladas de la implementación, especificaciones de los módulos, ejemplos de uso y propuestas de utilización se encuentran en el manual de usuario de Freeling, que es el **Anexo 1** de este trabajo.

4.2.2 Instalación bajo Windows

Como se ha indicado en el apartado de requisitos técnicos, el entorno en el que se va a ejecutar el proyecto está ligado a tecnología Microsoft. Esto implica que el etiquetador debe ser capaz de ejecutarse bajo Windows Server.

Aquí aparece el primer problema grave en el desarrollo del proyecto, pues los desarrolladores de Freeling no dan soporte para instalaciones ni ejecuciones del programa en este entorno. Desde la sección de instalación de la página web del etiquetador ya advierten que aunque se ha reportado que es posible la ejecución bajo Windows, la migración es compleja y costosa.

En el foro de soporte de dicha web, los usuarios con interés en esta portabilidad, han ido recopilando diversas formas de intentar la migración del programa. Antes de explicar cuál ha sido la que he utilizado, voy a resumir las técnicas que se proponían.

En la primera propuesta, con el fin de obtener los archivos binarios preparados para la plataforma win32 tenemos que seguir una serie de pasos:

- Instalar MingW32, que es una implementación de los compiladores GCC para la plataforma Win32, que permite migrar la capacidad de este compilador en entornos Windows.

- Descargar las librerías Pcre, Berkeley Database, Boost, Libcfg+, Omlet, Fries y Freeling.

- Utilizar MingW32 para compilar de forma cruzada.

Otra propuesta, denominada Cygwin Pura, consiste en la construcción de Freeling para cygwin (Windows) mediante la aplicación cygwin. Las principales ventajas son:

- El proceso de construcción es el mismo que en Unix, muy sencillo.

- El software de creación entiende las rutas de unix (/)

Estas propuestas están muy detalladas en el foro de freeling, en la sección referencias se puede encontrar un enlace a dicho contenido.

En mi caso, intenté las dos versiones que he nombrado, y en ninguna conseguí tener éxito. Finalmente pude obtener el programa ya portado gracias a una entrada en la página web de Ciarán Ó Duibhín, donde además de explicar detalladamente los pasos a seguir para

conseguir versiones anteriores de Freeling para la plataforma Windows, indica cómo se debe reconfigurar el archivo de configuración del etiquetador para utilizarlo correctamente. Este es el caso de dónde se deben modificar las rutas de acceso a ficheros, pues las que vienen por defecto son incorrectas para Windows.

4.2.3 Uso de la herramienta de análisis

Para facilitar la invocación del programa, un script llamado analiza se proporciona con el paquete Freeling. Este script es capaz de localizar los archivos de configuración por defecto, definir las rutas de búsqueda de bibliotecas, y decidir si se desea el cliente-servidor o la versión recta.

El programa principal de muestra se llama con el comando:

analyze [número de puerto] [-f config-file] [opciones]

Si el número de puerto se omite el analizador se inicia en modo directo. Si un número de puerto es constante, un servidor se pondrá en marcha para aceptar conexiones de socket en ese puerto.

Si el fichero de configuración no se especifica, un archivo llamado analyzer.cfg se buscará en el directorio de trabajo actual. Si el fichero de configuración se especifica, pero no se encuentra en el directorio actual, será buscado en el directorio de instalación de FreeLing.

El fichero de entrada que se le proporciona al analizador debe tener la extensión .txt, y contener los elementos que se quieren etiquetar como si fueran un texto. El tokenizador interno reconocerá cada elemento separado por un blanco.

El fichero de salida, tendrá extensión .mrf, y contendrá una línea por cada elemento que haya etiquetado, de tal forma que en cada línea el primer elemento corresponderá al elemento que se va a etiquetar, el segundo corresponderá a la forma de cita propuesta para ese primer elemento, mientras que en el tercer lugar aparecerá la

etiqueta propuesta. Cabe destacar que las etiquetas que utiliza este etiquetador son las PAROLE, que intentan ser un estándar.

Los problemas para procesar la salida del etiquetador surgen cuando nos damos cuenta de que no siempre el número de tokens a etiquetar coincide con el número de tokens etiquetados. En el caso de las formas compuestas detectadas, en el fichero de salida solo encontraremos una línea de salida que estará asociada a varias entradas diferentes. Pero no solo existe el caso de encontrar más entradas que salidas, existen casos, como en la detección de contracciones (al, del) o en las formas verbales con pronombres enclíticos, donde una sola entrada se corresponderá con varias líneas de salida (a + el, de + el). En la descripción de la fase de implementación comentaré más detalladamente qué solución tomaré para detectar estas anomalías.

5. Diseño e implementación del sistema

5.1 Modelado de la estructura de datos

Antes de implementar el sistema es necesario modelar las estructuras de datos que vamos a reconocer y cómo van a interactuar entre ellas.

Es conveniente recordar la estructura de los documentos:

- Cada documento representa una transcripción
- Cada párrafo pertenece a una intervención
- En las intervenciones aparece marcado el interlocutor
- La transcripción está marcada con símbolos conversacionales que están repartidos por todo el documento.

A partir de estas especificaciones se propone el siguiente modelado de las estructuras de datos.

Transcripción

Contiene los siguientes elementos:

- Nombre: Se corresponde con el nombre del documento Word sin la extensión.
- Fecha de modificación: La fecha de modificación será la del sistema cada vez que se edite el contenido del XML en el que se almacenarán los datos.
- Estado: El estado indicará en qué fase se haya la transcripción. Por ejemplo en el momento de creación el estado pasará a ser "Creado", y cuando todos los elementos estén etiquetados cambiará a "Etiquetado"
- Lista de intervenciones (lineas): Colección de elementos de tipo línea.

Línea

Contiene los siguientes elementos:

- Contenido: En esta variable se almacenará el contenido literal del párrafo asociado a la línea, incluyendo la clave del interlocutor y el elemento separador ':'
- Número: La posición en la que aparece dentro de la transcripción
- Interlocutor: La clave del interlocutor de esta intervención
- Estado: El estado indicará en qué fase se haya la línea. Por ejemplo en el momento de creación el estado pasará a ser "Creado", y cuando todos sus elementos estén etiquetados cambiará a "Etiquetado". Si no contiene ningún elemento su estado será "Sin elementos"
- Lista de elementos: Colección de elementos de tipo Elemento.

Elemento

Contiene los siguientes elementos:

- Estado: El estado indicará en qué fase se haya el elemento. Por ejemplo en el momento de creación el estado pasará a ser "Creado", y cuando se le asigne una etiqueta en el módulo de

etiquetado (si es de tipo palabra) o en el módulo de tokenizado (si es un símbolo) pasará a "Etiquetado"

- Etiqueta: Etiqueta asignada al elemento
- Aparición: Sucesión de caracteres original

Palabra

Es una especialización de Elemento

Símbolo

Es una especialización de Elemento

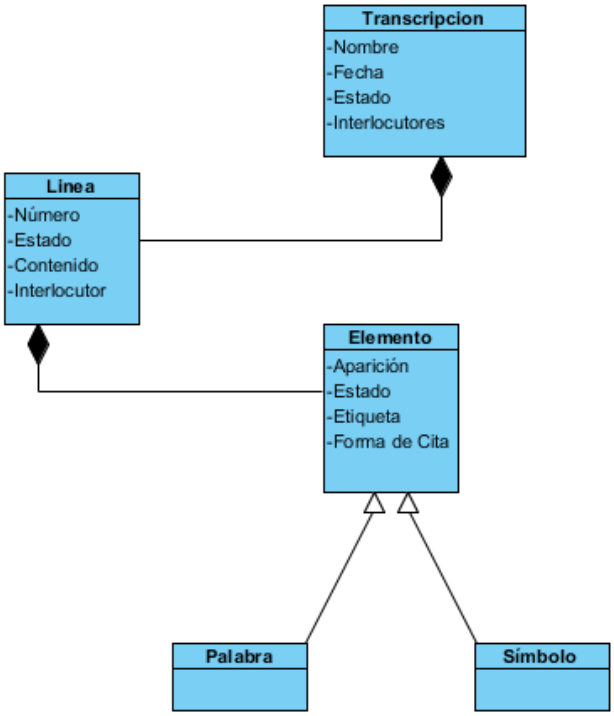

Figura 7. Detalle del sistema con el Diagrama de clases asociado

5.2 Manipulación por código de documentos .doc

Como ya he comentado anteriormente, todas las transcripciones que forman el corpus de Val.Es.Co están almacenadas en documentos

.doc. Esto hace imprescindible que se incluya código de manipulación de objetos de office en el sistema desarrollado.

Para ello se requiere de nuevo una investigación, en este caso sobre qué características se pueden controlar desde código, y cómo se deben realizar las conexiones.

Lo primero que debemos hacer es añadir a nuestra solución del Visual Studio las referencias necesarias para tener acceso a las propiedades y objetos de office, en concreto a los de Word.

Es necesario tener instalado en el equipo Microsoft office, de lo contrario no podremos referenciarlo.

En el Visual Studio, en la ventana de explorador de la solución hacemos clic en el botón derecho sobre nuestro proyecto y elegimos la opción Add Reference. La figura 8 ilustra este paso.

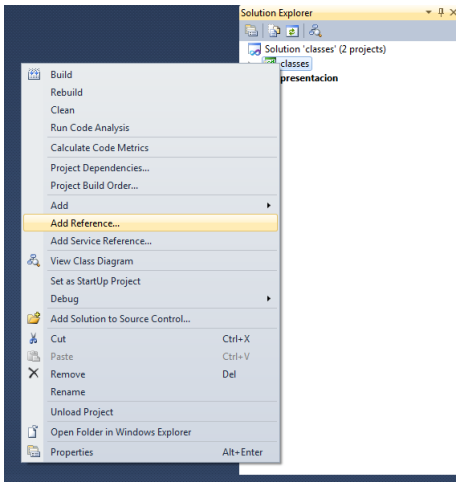

Figura 8. Captura de pantalla un submenú del Visual Studio.

Nos ubicamos en la pestaña **.NET** del cuadro de diálogo Add Reference y buscamos **Microsoft.Office.Interop.Word** y **Microsoft.Office.Core** para añadirlas. La versión de la referencia es la equivalente a la versión de office instalada en la maquina donde se esté ejecutando el Visual Studio, en este proyecto en concreto la versión utilizada es la de Office 2010. La figura 9 ilustra este paso.

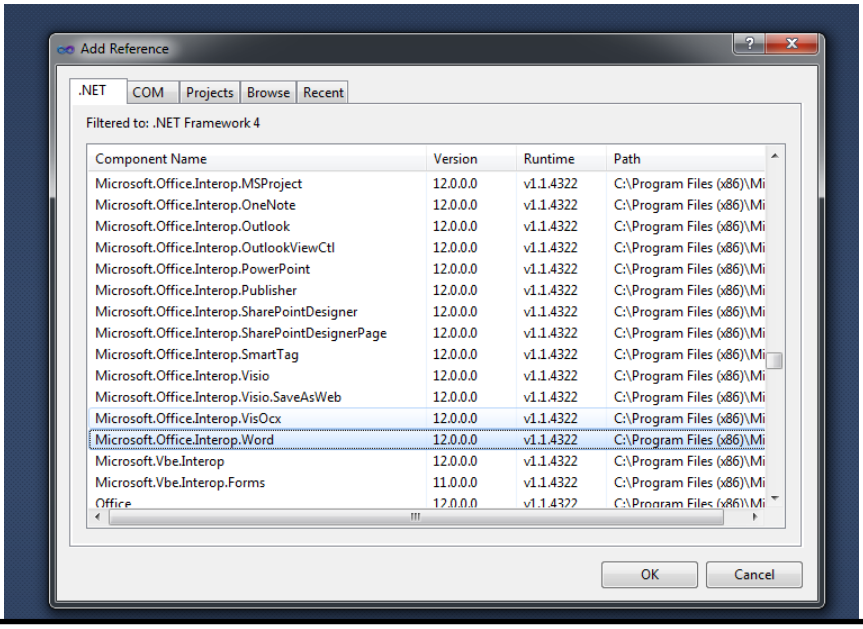

Figura 9. Detalle de la pestaña .NET del cuadro de diálogo Add Reference de Visual Studio.

Una vez ejecutado el paso anterior podemos comprobar como en la carpeta *References* de nuestro proyecto ya figuran las dos referencias que acabamos de añadir. Por último, para poder utilizar los objetos en una determinada clase, es necesario añadir la referencia con la directiva using correspondiente al inicio del documento. La figura 10 ilustra el resultado obtenido.

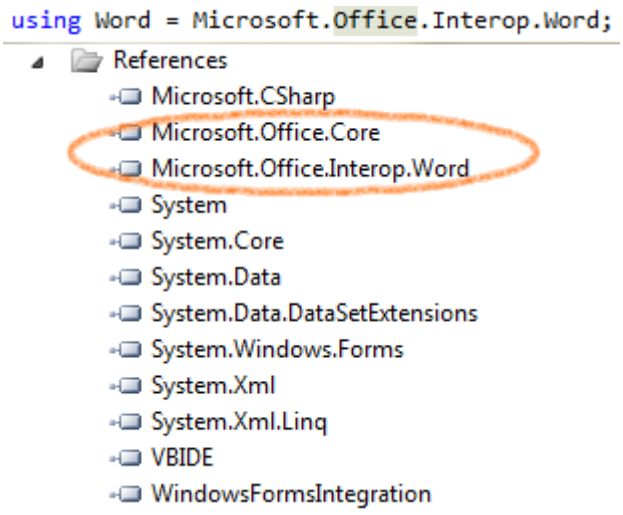

Figura 10. Estado de la carpeta **References** después de agregar las referencias.

Debemos crear una instancia de la aplicación, cuya clase es **Microsoft.Office.Interop.Word.Application**. Para abrir un documento Word necesitamos la ruta hasta el documento. Con el método Open de

la clase **Microsoft.Office.Interop.Word.Document** obtenemos la referencia a este objeto.

En nuestro caso, lo que queremos extraer de este documento es su contenido, pero organizado por párrafos, pues las estructuras de datos que hemos modelado consideran cada uno de estos párrafos una entidad diferente.

El siguiente problema con el que nos encontramos es qué clase de objeto usar para recuperar la información. La mejor opción es utilizar la interfaz **IDataObject**, pero para ello tenemos que incluir también entre las referencias de nuestro proyecto a **System.Windows.Forms** y a **System.Data.**

Deberemos pues recorrer cada párrafo del documento, seleccionándolo con la combinación de la propiedad *select* de la clase **Range** y de la propiedad *Copy* de la clase **ActiveWindow.Selection.**

De esta forma, utilizando la propiedad **GetDataObject** de la clase **Clipboard** y asignándosela a la variable de tipo IDataObject habremos recuperado la información contenida en ese párrafo.

Cuando terminemos de extraer todo el contenido que nos interese del documento utilizaremos el método **close** para cerrar el documento.

5.3 Tokenizador

En el módulo de tokenizado se produce una discriminación entre símbolos y palabras, pues es aquí donde se reconocen los elementos.

Para garantizar el correcto etiquetado de los símbolos conversacionales, es en esta fase donde se les asigna la etiqueta que les corresponde.

El algoritmo recorre cada carácter de cada línea buscando símbolos simples, comprobando con secuencias anteriores para los símbolos compuestos y para las palabras.

En la siguiente figura encontramos la descripción y el esquema del método.

```
/// <summary>
/// ProcesarLinea tokeniza una intervencion(linea) y crea las estructuras en las que se almacenan los elementos tokenizados.
/// </summary>
/// <param name="linea">La linea(intervencion).</param>
/// <returns>Retorna la string con la que se representa al interlocutor de dicha intervencion</returns>
private string ProcesarLinea(linea linea)
{
    if (linea.Contenido.Length > 0)
        {
            1: Inicializacion de variables
            2: Correspondencia de simbolos
            3: Division de la intervencion en 2 partes:
            4: Para cada caracter del contenido de la intervencion
            5: Comprobaciones adicionales
        }
        else linea.Interlocutor = null;
            6: Return
}
```

Figura 11. Detalle del esquema y la documentación de la función ProcesarLinea.

Como parámetro de entrada se recibe un objeto de tipo línea en el que previamente se ha cargado el valor de la variable contenido.

El método devuelve una string en la que se ha cargado el valor de la clave del interlocutor.

Si la variable contenido está vacía, la línea no contiene elementos, por lo que no se puede procesar nada y el método retorna null.

En los demás casos, el proceso empieza declarando las variables e inicializándolas. Definimos dos enteros, uno que servirá para almacenar la posición actual en el recorrido del *contenido,* y otro que representará cuántos elementos hemos identificado. Utilizaremos una variable de tipo carácter para almacenar el valor del carácter inmediatamente anterior al actual, y utilizaremos dos variables más, de tipo string, que usaremos de buffer auxiliar para almacenar las secuencias compuestas por más de un carácter (una para las palabras, y otra para los símbolos). Por último, definiremos una variable de tipo símbolo y otra de tipo palabra, sin instanciar.

El siguiente paso consiste en corregir algunas de las deficiencias encontradas en el texto. En concreto, se ha observado que en las transcripciones el símbolo que representa un silencio de x segundos se ha transcrito de varias formas distintas. Por ello, mediante el método

Replace de la clase string, cada vez que encontremos la secuencia ", `` ó ´´ seguida de un entero, reemplazaremos la marca encontrada por la estándar '' (dos simples comillas).

Es tiempo ahora, de separar en dos partes el *contenido* de la línea. Como habíamos descrito en la sección del *grupo Val.Es.Co,* cada intervención es almacenada originalmente con el siguiente formato:

Clave_interlocutor:transcripción_de_la_intervención

Mediante el método *Contains* de la clase string, nos aseguramos de que el elemento separado, ':' está presente en el valor de la intervención que estamos procesando, y si está, procedemos a almacenar el valor de cada una de las partes por separado.

Después de realizar estas acciones ya somos capaces de procesar esta línea para detectar los distintos elementos que tiene, y almacenarlos correctamente.

Recorreremos carácter a carácter el *contenido*. Para detectar algunos de los símbolos compuestos será necesario añadir una serie de comprobaciones que deberán ser evaluadas antes de almacenar el contenido del carácter actual, pues en gran parte dependen de los caracteres almacenados en el buffer auxiliar. Este es el caso de los símbolos compuestos que comienzan con el carácter '(' y de los símbolos relacionados con pausas (carácter '/').

La discriminación de qué caso se debe evaluar se produce mediante una instrucción switch. El orden en el que se evalúan los casos es el siguiente:

1: Símbolo "Silencio de x segundos"

```
case "(\")"
```

Si en nuestro buffer auxiliar de símbolos tenemos almacenada la secuencia ("), crearemos una nueva instancia de la clase símbolo. En este momento asignaremos la etiqueta "Silencio de "…" segundos" donde "…" será el número leído anteriormente y almacenado en el otro buffer auxiliar. A continuación, añadiremos este símbolo a nuestra lista de elementos y vaciaremos los buffers auxiliares.

2: Símbolos "Fragmento indescifrable" y "Transcripción dudosa"

case "(())"

En caso de que nuestro buffer auxiliar de símbolos almacene la secuencia (()).

2.1: Símbolo "Fragmento indescifrable"

Si nuestro otro buffer auxiliar no contiene ningún carácter o contiene el carácter vacío, tendremos el caso Fragmento indescifrable, por lo que crearemos el símbolo, lo etiquetaremos y añadiremos a la lista de elementos y vaciaremos los buffers auxiliares.

2.2: Símbolo "Transcripción dudosa"

Si en el otro buffer auxiliar tenemos algún carácter diferente del carácter vacío, crearemos el símbolo, lo etiquetaremos y añadiremos a la lista de elementos, crearemos un nuevo elemento de tipo palabra con el contenido del buffer auxiliar, lo añadiremos a la lista de elementos, y vaciaremos todos los buffers.

3: Símbolo "Interrupción de la grabación"

case "((...))"

Si en nuestro buffer auxiliar de símbolos tenemos almacenada la secuencia ((…)), crearemos una nueva instancia de la clase símbolo. En este momento asignaremos la etiqueta "Interrupción de la grabación". A continuación, añadiremos este símbolo a nuestra lista de elementos y vaciaremos los buffers auxiliares.

4: Símbolos "Entre risas" y "Reconstruido"

case "()"

En caso de que nuestro buffer auxiliar de símbolos almacene la secuencia ().

4.1: Símbolo "Entre risas"

Si en el otro buffer auxiliar tenemos almacenada la secuencia "RISAS", crearemos un nuevo símbolo con etiqueta "Entre risas", lo añadiremos a la lista de elementos y vaciaremos los buffers auxiliares.

4.2: Símbolo "Reconstruido"

En los demás casos, crearemos un nuevo símbolo con la etiqueta "Reconstruido", lo añadiremos a la lista de elementos y vaciaremos el buffer auxiliar de símbolos.

5: Símbolo "Pausa corta"

```
case "/"
```

En caso de que nuestro buffer auxiliar de símbolos almacene el carácter '/', y el carácter actual no sea '/', crearemos un nuevo símbolo con la etiqueta "Pausa corta", lo añadiremos a la lista de elementos y vaciaremos los buffers auxiliares.

6: Símbolo "Pausa intermedia"

```
case "//"
```

En caso de que nuestro buffer auxiliar de símbolos almacene la secuencia '//', y el carácter actual no sea '/', crearemos un nuevo símbolo con la etiqueta "Pausa intermedia", lo añadiremos a la lista de elementos y vaciaremos los buffers auxiliares.

Después de estas comprobaciones, y mediante otro switch, procedemos a evaluar el carácter actual, almacenándolo si corresponde su valor en alguno de los dos buffers auxiliares, o añadiéndolo directamente a la colección de elementos si de un símbolo simple se trata. El orden en el que se evalúan los casos es el siguiente:

1: Símbolo "Sucesión inmediata"

```
case '§'
```

Si el carácter que leemos en la iteración actual es '§', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo símbolo con la etiqueta "Sucesión inmediata" que representa al carácter encontrado y lo añadimos a la lista de elementos.

2: Símbolo "Mantenimiento del turno"

```
case '='
```

Si el carácter que leemos en la iteración actual es '=', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo símbolo con la etiqueta "Mantenimiento del turno" que representa al carácter encontrado y lo añadimos a la lista de elementos.

3: Símbolo "Inicio solapamiento"

```
case '['
```

Si el carácter que leemos en la iteración actual es '[', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo símbolo con la etiqueta "Inicio solapamiento" que representa al carácter encontrado y lo añadimos a la lista de elementos.

4: Símbolo "Final solapamiento"

```
case ']'
```

Si el carácter que leemos en la iteración actual es ']', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo símbolo con la etiqueta "Final solapamiento" que representa al carácter encontrado y lo añadimos a la lista de elementos.

5: Símbolo "Reinicio"

```
case '-'
```

Si el carácter que leemos en la iteración actual es '-', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo símbolo con la etiqueta "Reinicio" que representa al carácter encontrado y lo añadimos a la lista de elementos.

6: Símbolo "Pausa larga"

```
case '/'
```

Si el carácter que leemos en la iteración actual es '/', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar de palabras para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación comprobamos si en el buffer auxiliar de símbolos tenemos la cadena "//". Si esta última condición se cumple, creamos un elemento de tipo símbolo con la etiqueta "Pausa larga" y lo añadimos a la lista de elementos. Por el contrario, si la condición no se cumple, añadiremos a nuestro buffer auxiliar de símbolos el carácter '/'.

7: Símbolo '('

```
case '('
```

Si el carácter que leemos en la iteración actual es '(', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación añadiremos a nuestro buffer auxiliar de símbolos el carácter '('.

8: Símbolo "Entonación ascendente"

```
case '↑'
```

Si el carácter que leemos en la iteración actual es '↑', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo símbolo con la etiqueta "Entonación ascendente" que representa al carácter encontrado y lo añadimos a la lista de elementos.

9: Símbolo "Entonación descendente"

```
case '↓'
```

Si el carácter que leemos en la iteración actual es '↓', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo símbolo con la etiqueta "Entonación descendente" que representa al carácter encontrado y lo añadimos a la lista de elementos.

10: Símbolo "Entonación mantenida"

```
case '→'
```

Si el carácter que leemos en la iteración actual es '→', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para

crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo símbolo con la etiqueta "Entonación mantenida" que representa al carácter encontrado y lo añadimos a la lista de elementos.

11: Símbolo ')'

```
case ')'
```

Si el carácter que leemos en la iteración actual es ')', añadiremos a nuestro buffer auxiliar de símbolos este carácter.

12: Símbolo '"'

```
case '"'
```

Si el carácter que leemos en la iteración actual es '"', añadiremos a nuestro buffer auxiliar de símbolos este carácter.

13: Símbolo '.'

```
case '.'
```

Si el carácter que leemos en la iteración actual es '.', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo palabra que representará al carácter encontrado y lo añadimos a la lista de elementos.

14: Símbolo ' '

```
case ' '
```

Si el carácter que leemos en la iteración actual es ' ', comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos.

15: Símbolo "Bajo"

```
case 'º'
```

Si el carácter que leemos en la iteración actual es 'º' y el carácter que hemos leído en la iteración anterior no es '(', añadiremos al buffer auxiliar de símbolos el carácter actual. Si no se produce la anterior condición, y en el buffer auxiliar de símbolos tenemos almacenada la cadena "º()" creamos un nuevo elemento de tipo símbolo con la etiqueta "Bajo" y lo añadimos a la lista de elementos.

16: Símbolo "Interrogación" '¿'

```
case '¿'
```

Si el carácter que leemos en la iteración actual es '¿', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo palabra que representará al carácter encontrado y lo añadimos a la lista de elementos.

17: Símbolo "Exclamación" '¡'

```
case '¡'
```

Si el carácter que leemos en la iteración actual es '¡', primero comprobamos si tenemos algún elemento en nuestro buffer auxiliar para crear su elemento correspondiente y añadirlo a la lista de elementos, y a continuación creamos un elemento de tipo palabra que representará al carácter encontrado y lo añadimos a la lista de elementos.

Para terminar con la iteración actual, incrementamos en uno el valor de nuestro contador de posición, y asignamos el valor del carácter actual a la variable que representa el carácter anterior.

Cuando ya hemos recorrido todos los caracteres, volvemos a analizar que no tengamos ningún elemento guardado en los buffer auxiliares, y si lo tenemos procedemos a identificarlo y almacenarlo en la lista de elementos.

Por último, utilizamos la instrucción *return* para devolver el valor del interlocutor que habíamos detectado.
Actualización de variables

5.4 Etiquetado

Este módulo es el encargado de generar los documentos de entrada para el etiquetador Freeling, lanzarlo a ejecución, y procesar la información de salida.

Para crear el archivo de texto que se enviará al etiquetador, utilizamos la clase StreamWriter. En la fase previa, la de tokenizado, habíamos conseguido separar las palabras de los símbolos, por lo que con un simple recorrido de todas las palabras de la transcripción podemos generar el documento.

Pero antes de añadir cada palabra al texto, comprobaremos si padece alargamiento vocálico (duplica una vocal), y en caso de que lo padezca, será sustituida por una sin dicho fenómeno. Este proceso mejorará notablemente el % de acierto del etiquetado, puesto que las palabras con alargamiento vocálico no serían encontradas contra diccionario por Freeling y por tanto serían incorrectamente etiquetadas.

Una vez el fichero de entrada está listo, ya podemos ejecutar el comando para que el analizador lo procese.

La forma en la que ejecutamos dicho comando por código es la siguiente:

- Creamos un archivo con extensión .bat cuyo contenido sea una instrucción para ir al directorio raíz de freeling y otra donde se llame al método analyze con los parámetros oportunos.
- Creamos y arrancamos un proceso que ejecute el fichero .bat, utilizando el método *Start* de la clase System.Diagnostics.Process.
- Mantenemos en espera a nuestro sistema hasta que el proceso que hemos creado termine.

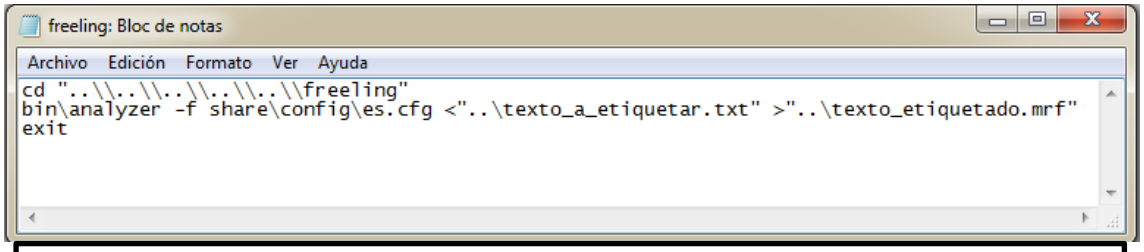

Figura 12. Contenido archivo .bat

A continuación, abriremos el nuevo fichero creado por el proceso mencionado para procesarlo. El algoritmo encargado de asignar a cada palabra presente en el fichero de entrada su correspondiente etiqueta propuesta sigue los siguientes pasos:

En primer lugar, declararemos e inicializaremos las variables auxiliares que utilizaremos más adelante.

```
StreamReader SR = new StreamReader(ruta+"texto_etiquetado.mrf",Encoding.Default);
```

En la variable SR cargaremos el archivo salida del etiquetador, con la codificación por defecto.

```
string[] words=null;
```

words es un vector de strings que contiene en su primer elemento, el token reconocido por freeling, en el segundo, la etiqueta propuesta y en el tercero, la forma de cita correspondiente.

```
string lectura;
```

La variable lectura, de tipo string, será donde almacenaremos el contenido de cada línea leída con el StreamReader.

```
bool compuesta = false; int num_composicion = 0;
```

Tanto la variable booleana compuesta, como el entero num_composicion son utilizadas para el cálculo de lecturas adicionales necesarias cuando se ha leído una palabra compuesta.

El segundo paso consiste en recorrer cada elemento de cada línea de la transcripción. Solo nos interesa analizar los elementos de tipo palabra, por tanto mediante la cláusula *is* comprobaremos el tipo del elemento.

Comprobamos que la variable auxiliar compuesta no está a true, pues si lo está esta iteración no tendrá ninguna lectura, y se asignará al elemento actual la etiqueta y la forma de cita leída en la iteración anterior.

Si no estamos ante una palabra compuesta, realizaremos una lectura, con la instrucción ReadLine, cuyo valor almacenaremos en la variable auxiliar lectura.

```
lectura = SR.ReadLine();
```

Utilizando la función Split presente en la clase string, asignaremos a la variable Word los elementos presentes en la línea leída y almacenada en lectura.

```
words = lectura.Split(' ');
```

A continuación evaluaremos si nuestro elemento actual coincide totalmente con el elemento leído. Si este es el caso, le asignaremos la etiqueta y forma de cita propuesta y terminaremos la iteración. Por el contrario, si no se produce coincidencia exacta seguiremos evaluando más casos.

Si el elemento actual es la contracción 'al' o la contracción 'del', estamos ante un caso especial, en el que el etiquetador separará en dos entidades lo que para nosotros era solo una. Realizaremos una primera asignación de etiqueta y forma de cita propuesta, y a continuación volveremos a utilizar la instrucción ReadLine para obtener una segunda lectura dentro de esta iteración. Por último, antes de terminar la iteración volveremos a asignar la nueva etiqueta y la nueva forma de cita propuesta a nuestro elemento.

El siguiente caso que evaluamos es si el elemento propuesto contiene el carácter '_', que denota que el elemento propuesto es una palabra compuesta por varios elementos. Si se da este caso, asignamos a la variable *num_composicion* el número de ocurrencias de este carácter, almacenamos la etiqueta y la forma de cita propuestas y terminamos la iteración después de poner a true la variable compuesta.

Si el elemento original contiene el carácter ', podemos encontrarnos ante dos casos especiales. El primero aparece con la opción de detección de números desactivada, pues el tokenizador del etiquetador separará los números decimales en tres partes. En este caso, deberemos asignar la etiqueta y forma de cita propuestas por la lectura inicial, y realizar dos lecturas y asignaciones adicionales. En el segundo caso, dos elementos unidos por una comilla simple son separados por el etiquetador. Ante este problema, primero asignaremos la etiqueta y forma de cita propuestas y después procederemos a realizar una segunda lectura y posterior asignación.

La función más compleja dentro de este módulo, es la encargada de detectar formas verbales encadenadas con pronombres enclíticos. Los pronombres enclíticos, son los pronombres que se unen al verbo precedente para formar una sola palabra.

El etiquetador Freeling es capaz de detectar estos pronombres, por lo que cuando un elemento de entrada está compuesto por una forma verbal junto con un pronombre enclítico, se genera en el documento salida una entrada para cada subelemento.

Nos podemos encontrar con dos tipos de casos en los que intervienen palabras con pronombres enclíticos. En primer lugar, puede coincidir que la forma verbal que se ha unido al pronombre enclítico tenga sentido por sí misma, y por tanto coincidirá con el primer elemento propuesto en el documento salida. Sin embargo, existen algunas formas verbales que se ven truncadas antes de unirse al pronombre enclítico, por lo que no podemos buscar la coincidencia exacta con la forma verbal propuesta por el etiquetador, si no que deberemos comprobar que la forma verbal original está contenida en la propuesta.

Además, es necesario tener en cuenta que los pronombres enclíticos pueden adherirse en parejas a una sola forma verbal.

El algoritmo para detectar los pronombres enclíticos evalúa, en el siguiente orden, estas características:

- Inicialización de variables
- Coincidencia exacta
    - Case 2 -> Pronombre de 2 caracteres
    - Case 3 -> Pronombre de 3 caracteres
    - Case 4 -> 2 Pronombres de 2 caracteres
    - Case 5 -> 2 Pronombres de 2 y 3 caracteres
        - Case 2 + 3
        - Case 3 + 2
        - Case 6 -> 2 Pronombres 3 caracteres
- Coincidencia parcial

5.5 Manipulación por código de documentos XML

XML es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Es una simplificación y adaptación del SGML y permite definir la gramática de lenguajes específicos. Por lo tanto XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades.

XML no ha nacido sólo para su aplicación en Internet, sino que se propone como un *estándar para el intercambio de información estructurada* entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier cosa imaginable.

Además, es una tecnología sencilla que tiene a su alrededor otras que la complementan y la hacen mucho más grande y con unas posibilidades mucho mayores. Tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.

Las principales ventajas de XML son:

- Es extensible: Después de diseñado y puesto en producción, es posible extender XML con la adición de nuevas etiquetas, de modo que se pueda continuar utilizando sin complicación alguna.
- El analizador es un componente estándar, no es necesario crear un analizador específico para cada versión de lenguaje XML. Esto posibilita el empleo de cualquiera de los analizadores disponibles. De esta manera se evitan bugs y se acelera el desarrollo de aplicaciones.
- Si un tercero decide usar un documento creado en XML, es sencillo entender su estructura y procesarla. Mejora la compatibilidad entre aplicaciones. Podemos comunicar aplicaciones de distintas plataformas, sin que importe el origen de los datos, es decir, podríamos tener una aplicación en Linux con una base de datos Postgres y comunicarla con otra aplicación en Windows y Base de Datos MS-SQL Server.
- Transformamos datos en información, pues se le añade un significado concreto y los asociamos a un contexto, con lo cual tenemos flexibilidad para estructurar documentos.

Almacenar los datos lingüísticos del corpus en este tipo de documentos, en lugar de volver a usar archivos .doc, permitirá recuperarlos, consultarlos o modificarlos con mayor facilidad.

Utilizando el editor XML del Visual Studio, y partiendo de las estructuras de datos modeladas anteriormente, creamos la estructura inicial del XML.

El siguiente paso, es obtener el XML Schema (xsd) asociado a este XML, lo que nos permitirá verificar qué documentos XML siguen correctamente la estructura que hemos definido. Para ello utilizamos, de nuevo, una herramienta que nos proporciona el Visual Studio y que automáticamente genera el XSD correspondiente al XML que hemos definido. Visualizando este nuevo archivo, podemos observar qué tipado ha asignado por defecto a cada uno de los campos del xml, además se nos permite modificar cualquiera de los tipos, siempre que el valor que asignemos sea el de un tipo soportado.

Pero lo más interesante ha sido descubrir que existe una herramienta, xsd.exe, que a partir de un XML Schema autogenera las clases C# necesarias para la manipulación de cualquier documento XML que tenga asociado dicho schema. Con esta herramienta, los programadores obtenemos de forma sencilla y eficaz la implementación de todas las estructuras de datos presentes en el schema, con sus correspondientes atributos y propiedades.

Crear un documento XML es ahora tan sencillo como después de haberle asignado a cada uno de los elementos su valor correspondiente, ejecutar estas tres instrucciones:

```
XmlSerializer serializadorXML=new XmlSerializer (typeof(nuestra_clase_auto_generada));
StreamWriter escritorXML = new StreamWriter(ruta + texto.Nombre + ".xml");
serializadorXML.Serialize(escritorXML, instancia_de_ nuestra_clase_auto_generada);
escritorXML.Close();
```

5.6 Organización y documentación del código

La llamada a cada una de las funciones descritas anteriormente se produce desde una función que recibe como parámetros la colección (SortedList) de intervenciones de una transcripción y un booleano, que indica si se debe etiquetar y almacenar o solo almacenar.

En esta función se crea la instancia de la transcripción, y se le asigna a ésta el valor del atributo nombre. Se recorre la colección de intervenciones, y para cada una de ellas se llama a la función de tokenizado (procesar_línea). Al terminar el recorrido, si el parámetro booleano está activado se realiza una llamada a la función de

etiquetado (Freeling), y por último se invocará a la función XML para almacenar los datos procesados en éste formato.

El código implementado ha sido documentado con la ayuda del complemento para Visual Studio GhostDoc para documentar las declaraciones de funciones. Los métodos han sido estructurados por regiones, para facilitar la reutilización del código aportando más claridad a la implementación. Cada variable declarada ha sido acompañada de un comentario donde se ha detallado su función.

6. Motor de búsqueda

6.1 Tecnología LINQ to XML

LINQ to XML proporciona una interfaz de programación XML en memoria que aprovecha las características de .NET Language-Integrated Query (LINQ) Framework. LINQ to XML utiliza las características más recientes del lenguaje .NET Framework y es comparable a una actualizada y rediseñada interfaz de programación XML para el Modelo de objetos de documento (DOM).

LINQ define operadores de consulta estándar que permiten a lenguajes habilitados con LINQ filtrar, enumerar y crear proyecciones de varios tipos de colecciones usando la misma sintaxis. Tales colecciones pueden incluir vectores, clases enumerables, XML, conjuntos de datos desde bases de datos relacionales y orígenes de datos de terceros. El proyecto LINQ usa características de la versión 2.0 del .NET Framework, nuevos ensamblados relacionados con LINQ, y extensiones para los lenguajes C# y Visual Basic .NET.

El objetivo de crear LINQ es permitir que todo el código hecho en Visual Studio (incluidas las llamadas a bases de datos, datasets, XMLs) sean también orientados a objetos. Antes de LINQ, la manipulación de datos externos tenía un concepto más estructurado que orientado a objetos. Además LINQ trata de facilitar y estandarizar el acceso a dichos objetos.

LIINQ usa varias características nuevas para permitir al lenguaje C# el uso de la sintaxis de consultas nativas. Estas novedades son el uso

de tipos anónimos, de métodos extensores, expresiones lambda, árboles de expresión y operadores de consulta estándar.

Aunque LINQ soporta inicialmente consultas en colecciones en memoria, bases de datos relacionales y datos XML, es una arquitectura extensible que permite a desarrolladores de orígenes de datos adicionales el uso del LINQ, implementando los operadores de consulta estándar como métodos extensores para sus orígenes de datos, o mediante la implementación de la interfaz IQueryable que permite convertir un árbol de expresión en tiempo de ejecución para transformarlo en algún lenguaje de consultas. Los operadores de consulta estándar son usados para objetos también y permiten consultar objetos en la memoria con la misma sintaxis LINQ.

En lugar de especificar explícitamente un tipo al declarar e inicializar una variable, se puede utilizar el modificador var (var es una variable de tipo implícito) para indicar al compilador que deduzca y asigne el tipo. Esta característica es muy interesante, ya que permite hacer consultas sin especificar de qué tipo deben ser los datos.

En el siguiente ejemplo se declaran variables de tipo var a las que se les asigna datos tan diversos como un entero, una string y una consulta a una colección de datos almacenada en memoria.

```
var numero = 5;

var nombre = “Laura";

var query = from str in stringArray

where str[0] == 'm'

select str;
```

Las variables declaradas como var tienen el mismo establecimiento inflexible de tipos que las variables cuyo tipo se especifica explícitamente. El uso de var permite crear tipos anónimos, pero se puede utilizar para cualquier variable local. Las matrices también se pueden declarar con tipos implícitos.

6.2 Implementación

Una vez nos es conocida la sintaxis de LINQ, definimos qué tipo de consultas vamos a implementar.

Los textos etiquetados contienen información sobre el tipo de elementos (palabra o símbolo), sobre las etiquetas y forma de cita de éstos y sobre su aparición en el documento original.

Las etiquetas de los elementos de tipo símbolo, las introducidas en el módulo de tokenizado, son cadenas de caracteres asignadas por los investigadores, por lo que para buscar en el documento XML las apariciones de estos símbolos la estrategia más factible es la de hacer coincidir las etiquetas de estos elementos con la cadena que se pasa como parámetro al método de consulta.

Por otra parte, las etiquetas correspondientes a los elementos de tipo palabra son las que el etiquetar utilizado (Freeling) ha propuesto para ese elemento. Estas etiquetas son las Parole, que se caracterizan por en cada posición de la etiqueta dar una determinada información. Por ejemplo, la etiqueta DD0MS0, correspondería a un D (determinante) D (demostrativo) 0 (impersonal) M (masculino) S (singular) 0 (sin poseedor). Con esta distribución de las etiquetas, podemos hacer consultas con mayor o menor filtrado. Podemos buscar categorías absolutas comparando solo el primer carácter de la cadena, o darle determinado valor a los caracteres con información complementaria para filtros muy concretos.

En otros casos, los elementos que interesará encontrar serán aquellos que hayan aparecido literalmente en el texto original, y por tanto el último tipo de consulta que reconoceremos será el literal.

Pero sin duda, para los investigadores, lo más valioso será poder hacer consultas combinadas donde los elementos puedan ser de cualquiera de los tipos descritos.

La descripción de los métodos es la siguiente:

El método Consulta es un método público que pretende ser la función de entrada de cualquier consulta. Recibe como parámetros la secuencia a consultar, el número de elementos intercalados permitidos, el número máximo de resultados que se desean y la ruta donde

encontrar los documentos XML sobre los que se desea consultar. En este método se recurre a otros dos métodos, en primer lugar se llama al método *tipar*, y a continuación al método *consultar.* La salida de esta función es una colección de arrays de tipo string, donde el primer elemento representa la intervención anterior, el segundo es la intervención que satisface la consulta, el tercero es la intervención posterior a ésta, el cuarto representa a la ruta al archivo xml donde se satisface la consulta y el quinto es la posición del último elemento de la secuencia a consultar.

La función *procesar_resultado,* de tipo privado, se utiliza para obtener la información deseada de cada una de las coincidencias encontradas.

Este método tiene como entrada una colección de referencias, con información de la ruta hacia el documento XML en el que aparecen y de la posición concreta dentro de dicho documento. La salida de esta función es una colección de arrays de tipo string, donde el primer elemento representa la intervención anterior, el segundo es la intervención que satisface la consulta, el tercero es la intervención posterior a ésta, el cuarto representa a la ruta al archivo xml donde se satisface la consulta y el quinto es la posición del último elemento de la secuencia a consultar.

Por su parte, la función *tipar* se encarga de determinar cuál es el tipo de cada elemento presente en la consulta. Esto es posible porque para denotar que el elemento es un símbolo o una categoría gramáticas se utiliza un carácter especial para introducirlo (@, \). Devuelve la lista de entrada tipada.

El método *consultar* puede ser utilizado de forma directa desde fuera de la biblioteca de clases (es un método público), o de forma interna, pues es utilizado por la función *Consulta*, como ya he descrito anteriormente.

Es en esta función donde se realizan las llamadas a las correspondientes funciones que consultan por tipo de elemento. Los parámetros de entrada son la secuencia a encontrar, el número máximo de resultados de la consulta y la ruta hacia los XML. La salida de esta función es una colección de arrays de tipo string, donde el primer

elemento representa la intervención anterior, el segundo es la intervención que satisface la consulta, el tercero es la intervención posterior a ésta, el cuarto representa a la ruta al archivo xml donde se satisface la consulta y el quinto es la posición del último elemento de la secuencia a consultar.

El método público *consultar_estado* recibe una lista con las rutas de los documentos XML de los que se quiere conocer el estado.

El método *símbolos* realiza una consulta en la que solo se busca la coincidencia de símbolos. Es de tipo privado, pues es invocado desde la función *Consultar* y recibe como entrada la secuencia a encontrar, el número máximo de resultados de la consulta y la ruta hacia los XML. Devuelve una colección que contiene la referencia a todas las coincidencias encontradas.

El método *literal* realiza una consulta en la que solo se busca la coincidencia literal de los elementos de la secuencia. Es de tipo privado, pues es invocado desde la función *Consultar* y recibe como entrada la secuencia a encontrar, el número máximo de resultados de la consulta y la ruta hacia los XML. Devuelve una colección que contiene la referencia a todas las coincidencias encontradas.

La función *categoria* realiza una consulta en la que solo se busca la coincidencia de la categoría gramatical de los elementos. Es de tipo privado, pues es invocado desde la función *Consultar* y recibe como entrada la secuencia a encontrar, el número máximo de resultados de la consulta y la ruta hacia los XML. Devuelve una colección que contiene la referencia a todas las coincidencias encontradas.

El método *literal_query* comprueba si coincide el siguiente elemento de una secuencia. Recibe como parámetros el XML sobre el que se está consultando, la lista de referencia a las coincidencias anteriores y el elemento de la secuencia a comprobar. Como salida devuelve la lista con las referencias a las coincidencias actualizadas.

El método simbolos_*query* comprueba si coincide el siguiente elemento de una secuencia. Recibe como parámetros el XML sobre el que se está consultando, la lista de referencia a las coincidencias anteriores y el elemento de la secuencia a comprobar. Como salida devuelve la lista con las referencias a las coincidencias actualizadas.

El método *gramatical_query* comprueba si coincide el siguiente elemento de una secuencia. Recibe como parámetros el XML sobre el que se está consultando, la lista de referencia a las coincidencias anteriores y el elemento de la secuencia a comprobar. Como salida devuelve la lista con las referencias a las coincidencias actualizadas.

La función literal_query_inicial comprueba si coincide el primer elemento de una secuencia. Devuelve la lista de referencias de las coincidencias.

La función simbolos_query_inicial comprueba si coincide el primer elemento de una secuencia. Devuelve la lista de referencias de las coincidencias.

La función gramatical_query_inicial comprueba si coincide el primer elemento de una secuencia. Devuelve la lista de referencias de las coincidencias.

7. Diseño e implementación de una demo

Como he explicado en el capítulo 2 y 3 de esta memoria, el sistema implementado en este trabajo por sí solo no está pensado para la interacción directa con el usuario. En un proyecto anterior ya se construyó la interfaz web desde la que los usuarios accederán a todas las soluciones implementadas para el grupo Va.Les.Co, tanto a las descritas e implementadas en este trabajo final de carrera, como las que han quedado fuera del contexto de éste, ya sea porque estaban ya implementadas o porque corresponden a fases que aún no están totalmente definidas por el grupo (como la alineación de los textos con su audio correspondiente).

Por este motivo, la implementación de este trabajo forma parte de una biblioteca de clases que por sí sola no puede ser ejecutada. Para una evaluación detallada de los resultados de las funciones desarrolladas, se ha construido una Demo utilizando formularios Windows que utiliza las características más básicas de las funciones implementadas y visualiza el resultado de ejecutar esos métodos.

La demo ha sido desarrollada en modo modal, y consta de 3 interfaces básicas:

- Menú principal: Donde se podrá elegir entre acceder a la parte de etiquetado, o a la de consulta.

- Formulario con funciones para el etiquetado: Permite cargar una transcripción original con extensión .doc, etiquetarla y almacenarla en XML o simplemente almacenar en XML los elementos de la transcripción sin etiquetarlos (solo se utiliza el módulo de tokenizado).

- Formulario con funciones para la consulta: Permite realizar consultas sobre los documentos XML etiquetados que se hallen almacenados en una ruta específica. Se permite búsqueda por símbolos, por literales, por categoría gramatical absoluta y combinada de cualquiera de los tipos.

En la siguiente figura se muestra el formulario principal, que se muestra después de lanzar el ejecutable. Está formado por dos Botones, si hacemos clic sobre el botón Etiquetar Transcripciones se abrirá el formulario de etiquetado en modo modal, mientras que si hacemos clic en el botón de consultar transcripciones nos aparecerá el formulario de consulta, también en modo modal. Para salir de la aplicación se debe hacer clic sobre la x de la esquina superior derecha.

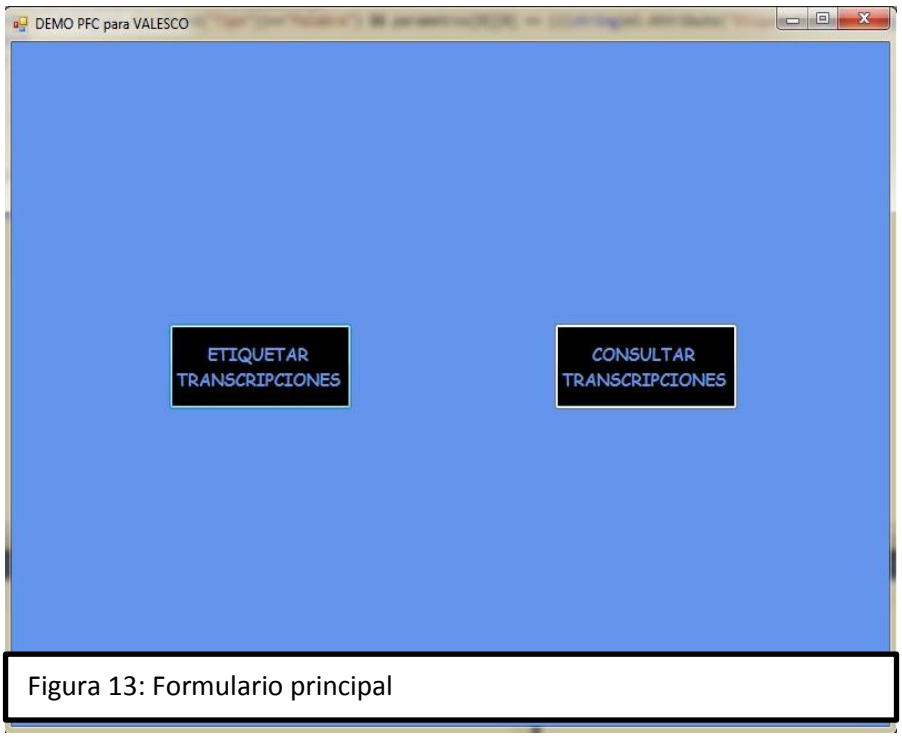

Figura 13: Formulario principal

En el formulario de etiquetado, los elementos Etiquetar y guardar en XML y Guardar en XML, están inactivos hasta que no se ha cargado un documento mediante el comando abrir.

Por tanto, vamos a empezar viendo cómo funciona el botón de abrir. En la siguiente imagen se observa dónde podemos encontrar el botón abrir. Si hacemos clic en él, se nos abrirá un cuadro de dialogo y deberemos navegar por nuestro equipo hasta encontrar el documento que queremos procesar. La figura 15 muestra este cuadro de diálogo. Una vez hemos encontrado la transcripción que nos interesa, pulsamos abrir y se nos cargará la ruta de ese documento en el cuadro de texto situado a la derecha del botón Abrir. Si nos equivocamos de documento podemos seleccionar uno nuevo tantas veces como estimemos, simplemente repitiendo el proceso que hemos seguido esta primera vez.

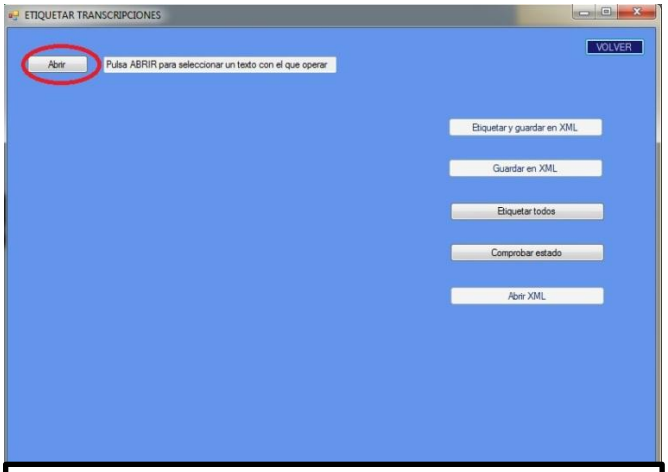

Figura 14. Botón ABRIR en el formulario de

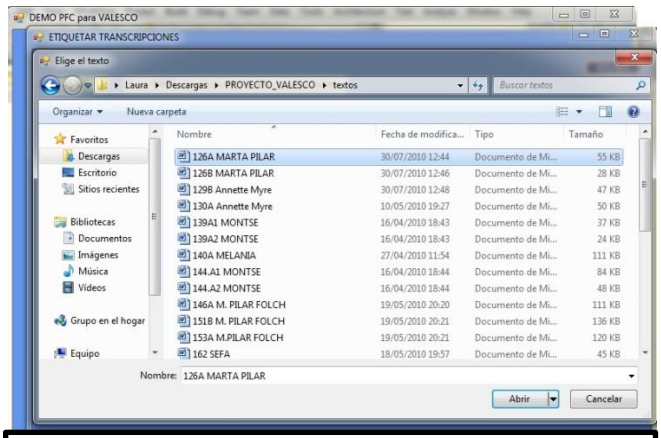

Figura 15. Cuadro de diálogo emergente.

Una vez cargado un documento, se desbloquean las opciones que no estaban disponibles antes. Para lanzar el proceso de etiquetado del texto, pulsamos el botón Etiquetar y guardar en XML, que se encuentra ubicado en la parte derecha de la interfaz, tal como indica la figura 16. Un instante después de haberse lanzado el proceso, aparecerá la ventana de símbolo de sistema, lo que significa que el etiquetador Freeling ha sido invocado y está procesando los datos. A continuación, desaparecerá de nuevo esta ventana y se mostrará en un cuadro de texto el contenido del documento Word que hemos abierto. Este paso está ilustrado por las imágenes 17 y 18.

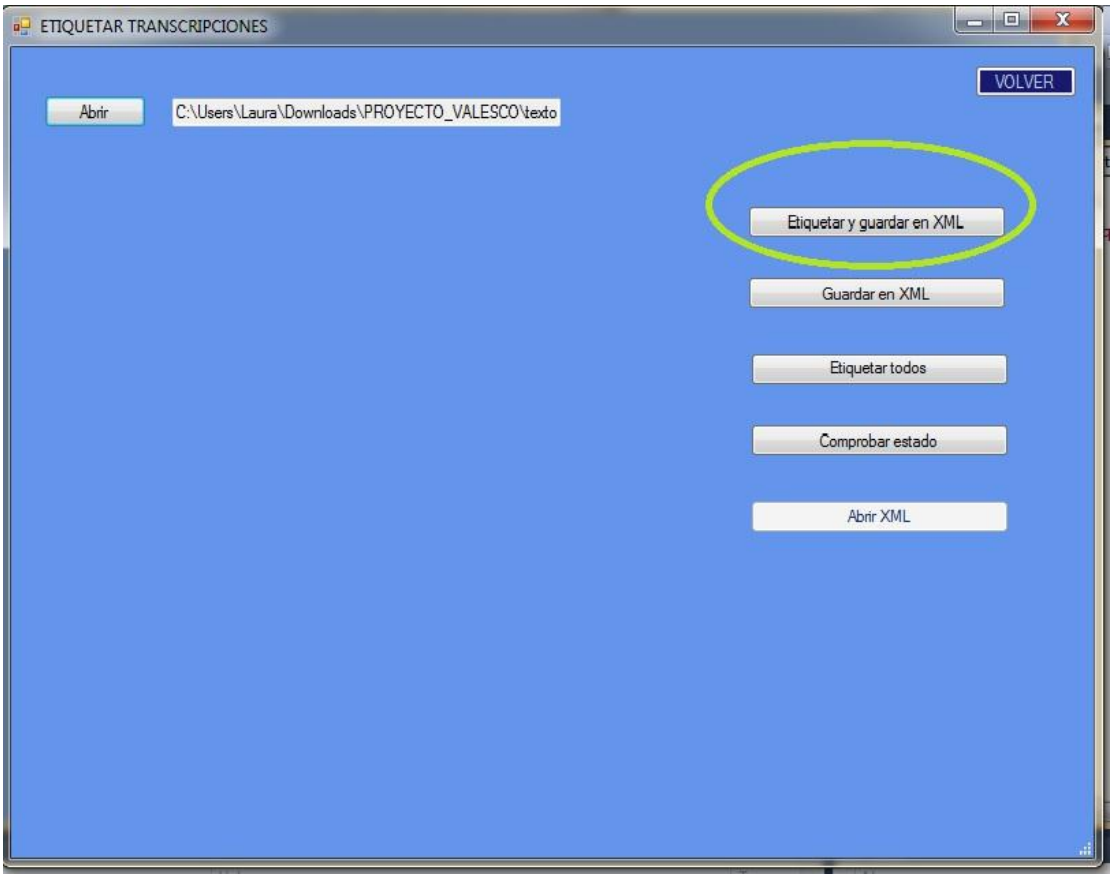

Figura 16. Botón Etiquetar y guardar en XML del formulario de Etiquetado.

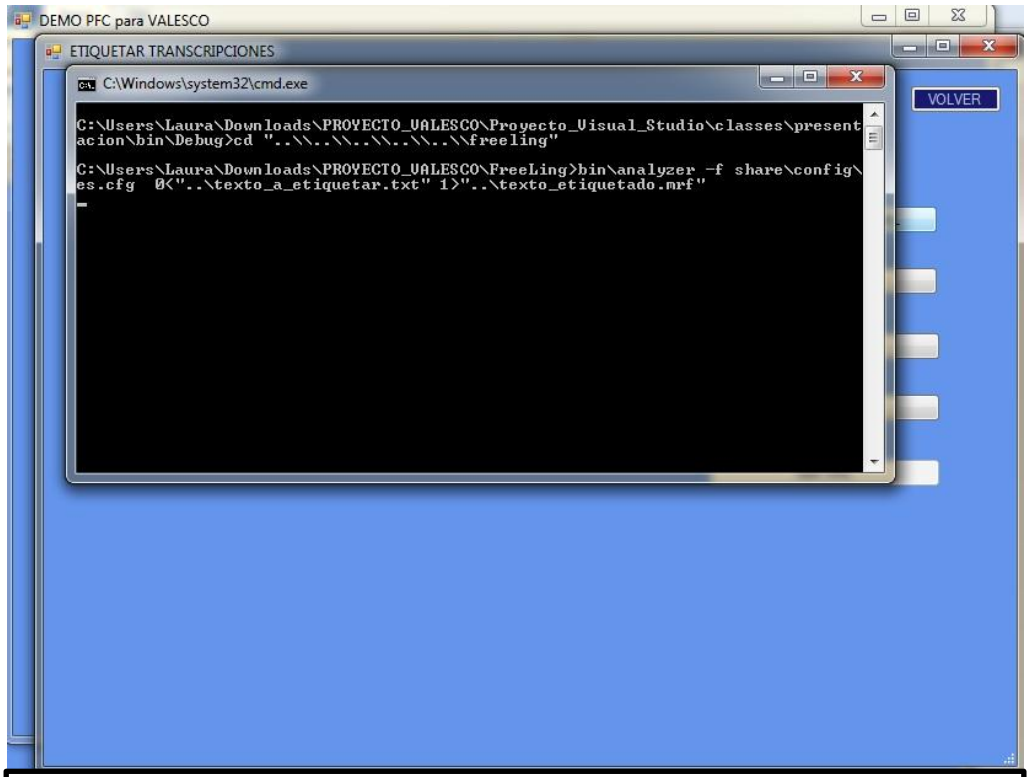

Figura 17. Símbolo del sistema.

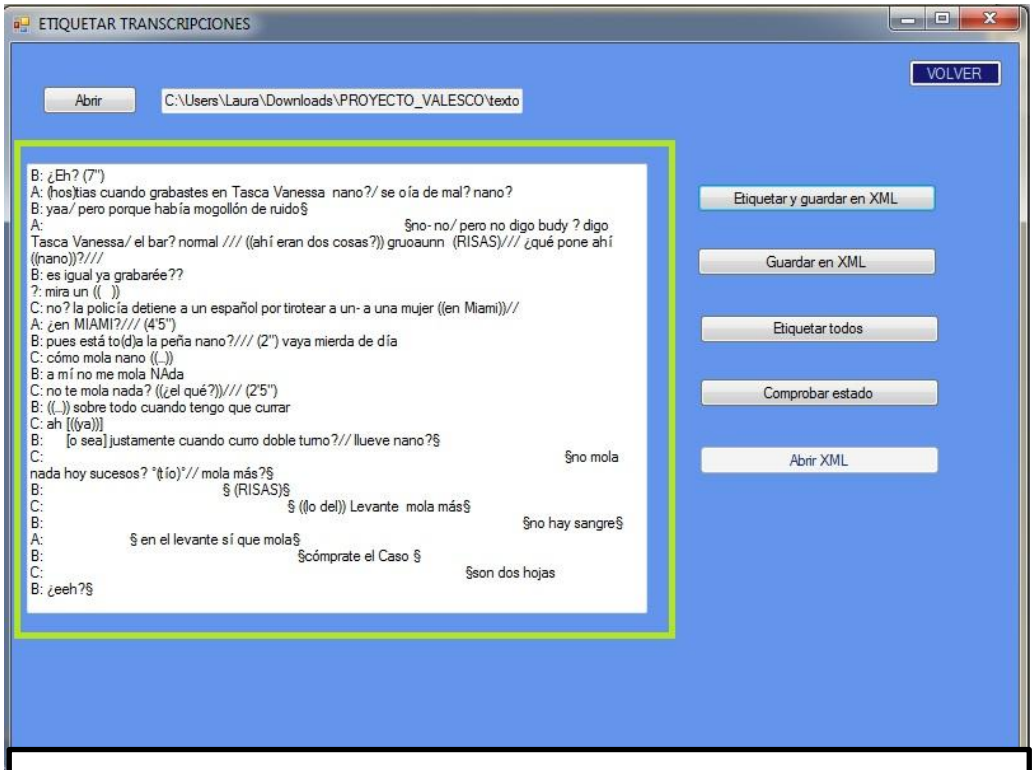

Figura 18. Cuadro de texto con el texto original cargado.

La función Comprobar Estado nos es muy útil, porque enumera la lista de documentos XML existentes en el directorio donde se

almacenan informándonos de la ruta, nombre del archivo y estado. Si el estado es etiquetado significará que todos los elementos de todas las intervenciones están etiquetados. En el apartado conclusiones veremos que esta etiqueta no tiene porqué ser correcta en todos los casos, por tanto, realmente el estado etiquetado indica que ese elemento tiene asignado *un candidato a etiqueta.*

Para ejecutarla, es necesario hacer clic sobre el botón Comprobar estado, ubicado a la derecha de la interfaz, como se puede observar en la imagen asociada. El cuadro de texto que anteriormente había mostrado el contenido original de una transcripción, ahora muestra la lista comentada. En concreto, nos fijamos en que el estado del texto que acabábamos de enviar a etiquetar se ha creado con el estado 'etiquetado', por lo que el proceso ha sido un éxito. Podemos observarlo en la figura 20.

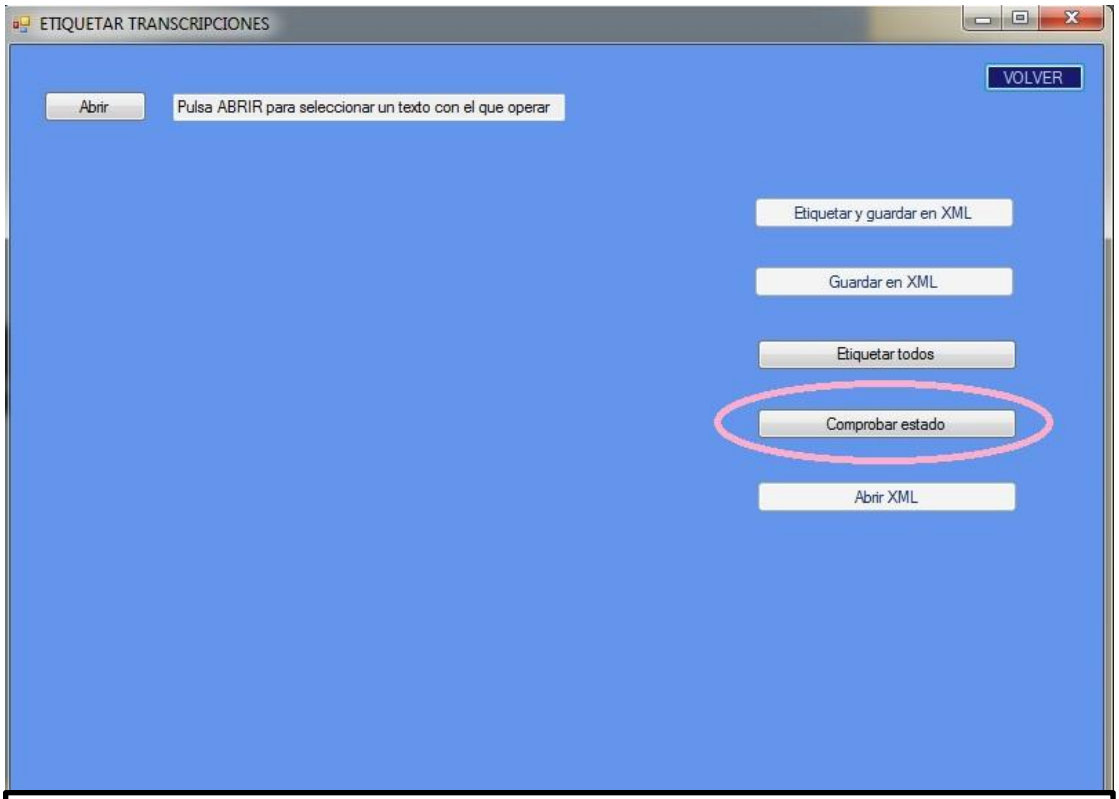

Figura 19. Botón Comprobar estado del formulario de Etiquetado.

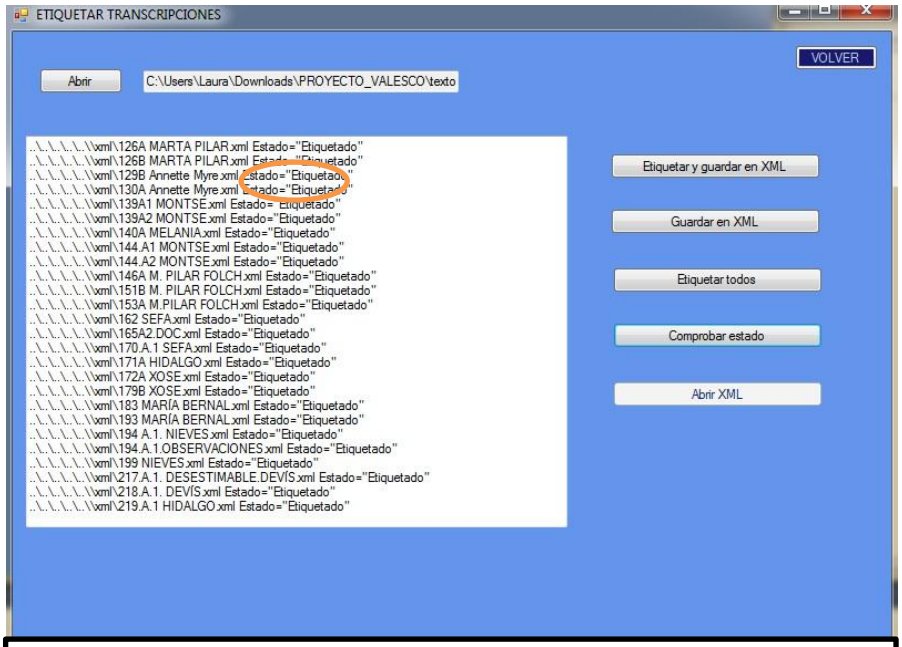

Figura 20. Detalle del estado del texto etiquetado.

Si en vez de etiquetar una transcripción eligiéramos la opción de sólo guardar en XML los elementos tokenizados, deberíamos hacer clic en el botón Guardar en XML, del menú de la derecha. El resultado de la ejecución sería el mismo cuadro de texto que se mostraba al terminar el proceso de etiquetado del texto, pero si volviéramos a ejecutar la comprobación del estado, observaríamos que el texto ahora tiene el estado 'Creado'. Las figuras 21 y 22 ilustran este proceso.

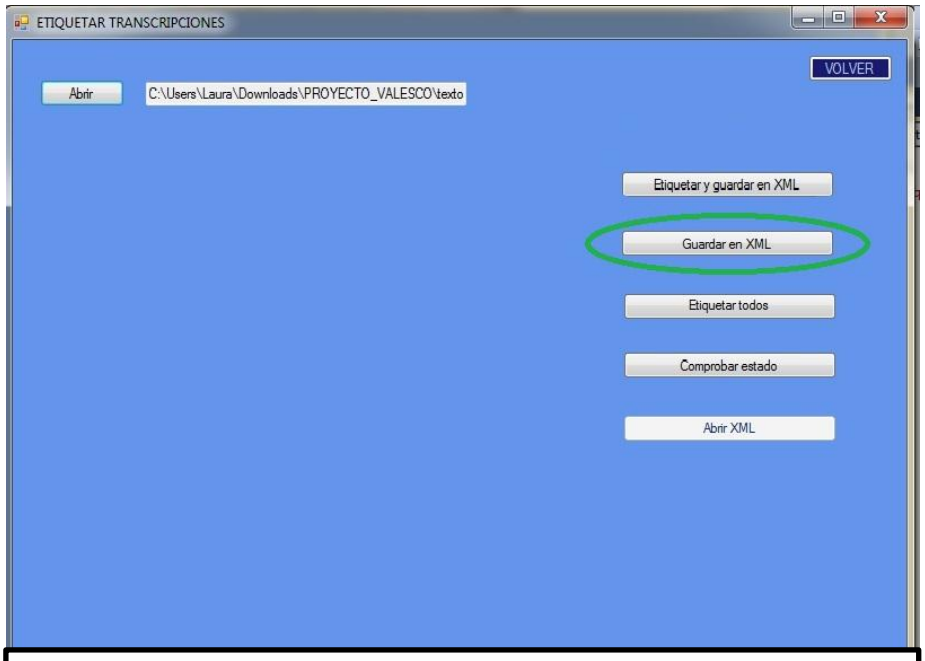

Figura 21. Botón Guardar en XML del formulario de etiquetado.

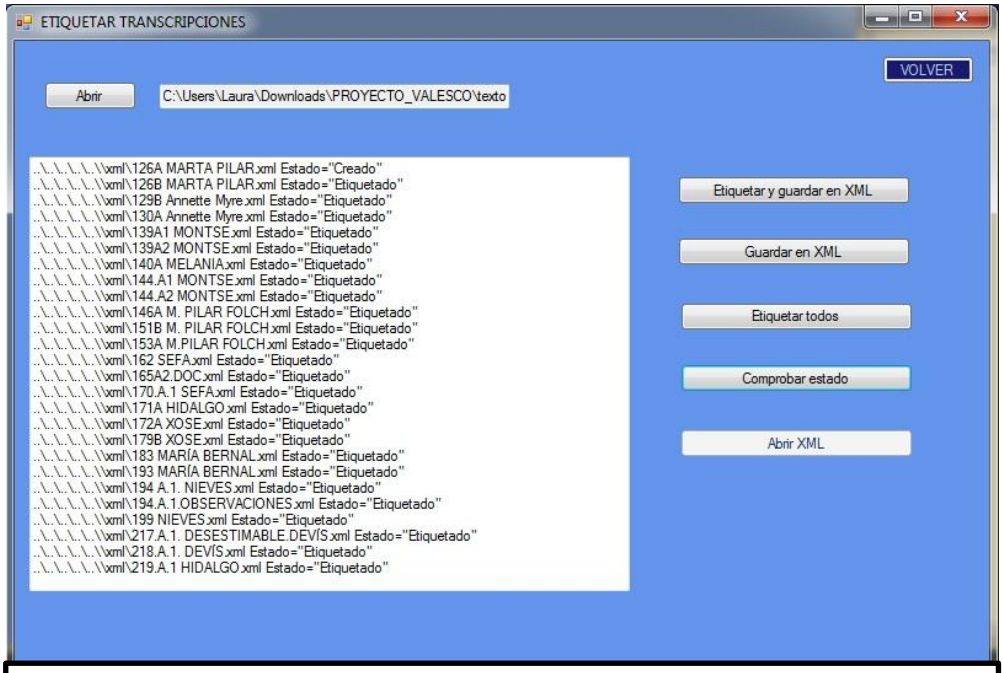

Figura 22. Detalle del estado del texto creado.

Pulsando el Botón *Volver* regresaremos al formulario principal. Elegimos ahora la opción de *Consultar transcripciones,* y nos aparece un formulario distinto con varias zonas de interacción con el usuario.

En la imagen 23, tenemos una vista global de este formulario.

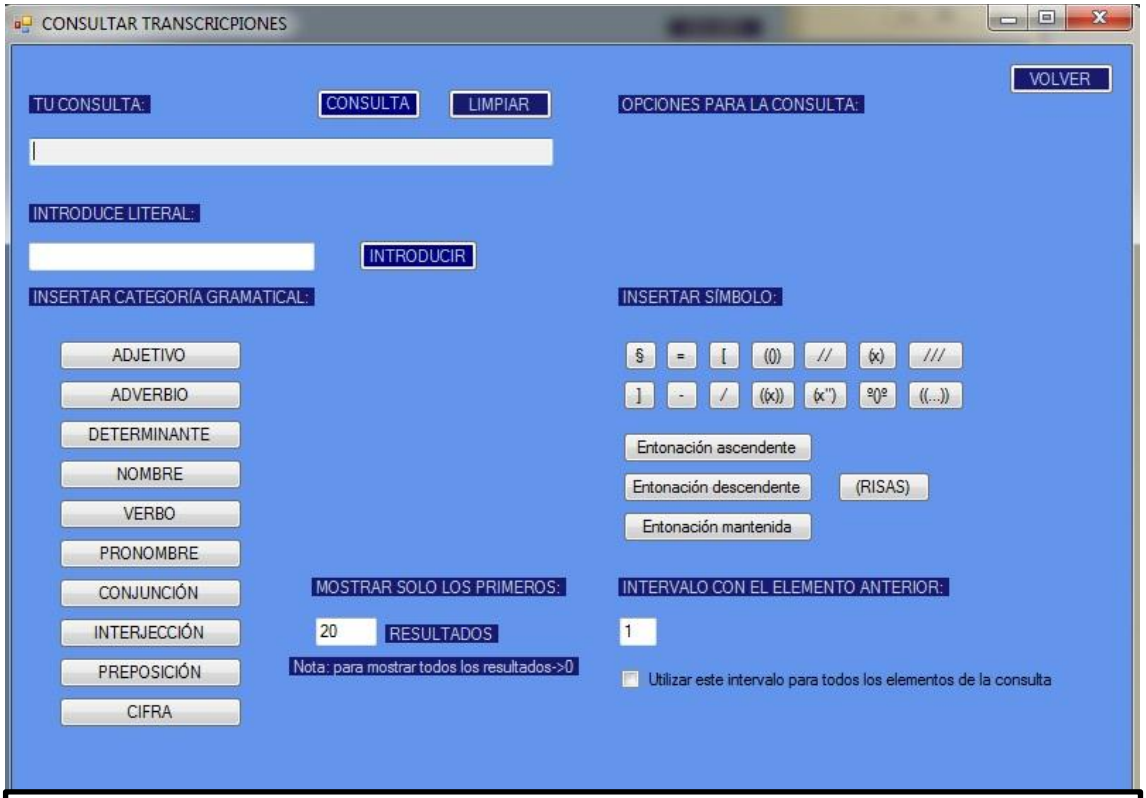

Figura 23. Formulario para Consultar Transcripciones.

En este formulario tenemos un total de 4 cuadros de texto, 32 botones, 1 check button y 9 etiquetas. Es conveniente, por tanto, revisar cada sección por separado para tener una visión clara de las opciones que proporciona esta interfaz.

Empezaremos centrándonos en la sección donde podremos incluir secuencias literales. En el cuadro de texto habilitado, introduciremos la palabra que queramos incluir en nuestra consulta, y pulsaremos el botón *Introducir* para formalizar la inclusión. Podemos añadir tantas palabras como queramos en cada consulta, siempre que sigamos el mecanismo de introducirlas individualmente. En el cuadro de texto de la consulta irán apareciendo las palabras incluidas, por orden. En la figura 24 está remarcada la ubicación de esta zona.

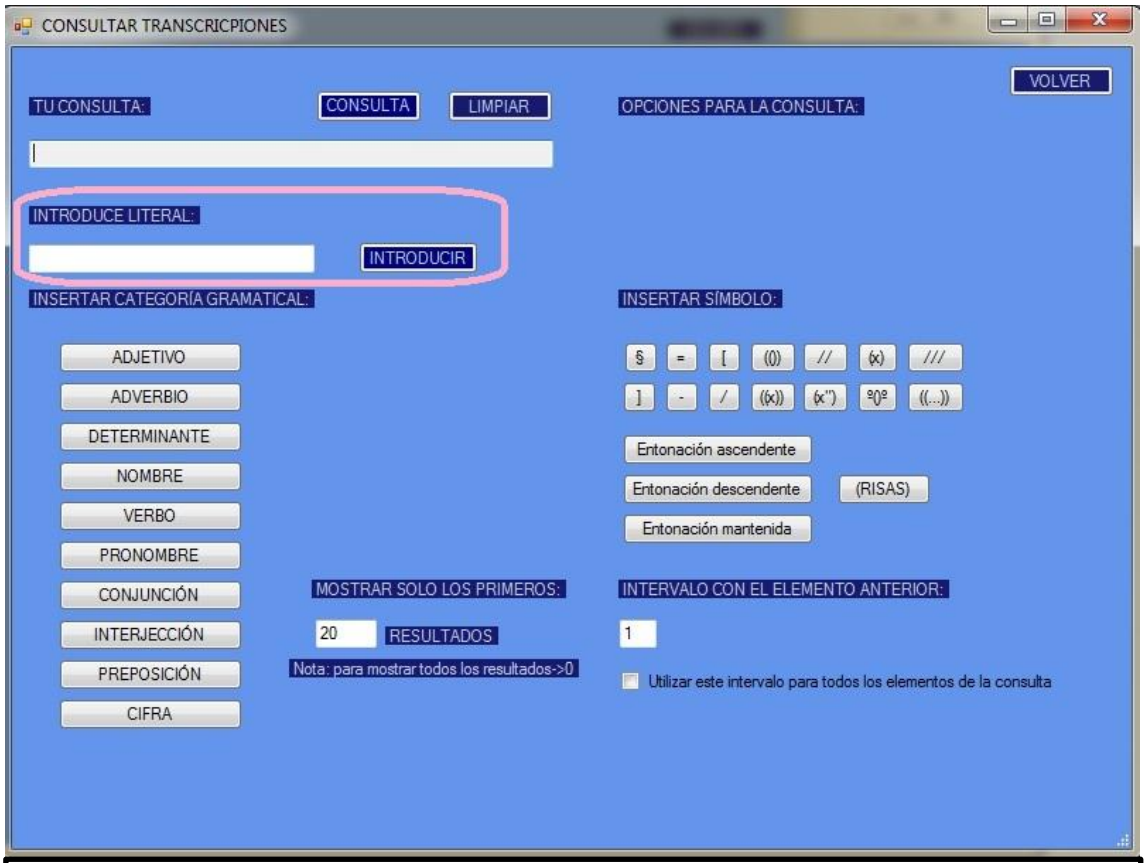

Figura 24. Detalle de la zona para introducción de literales.

En esta demo está permitido filtrar por categoría gramatical absoluta. Para introducir en la secuencia de consulta un determinado elemento representado por una categoría gramatical, bastará con pulsar el botón correspondiente a dicha categoría, y ésta se añadirá a la consulta. En la imagen 24 observamos cuál es la zona de botones para las categorías gramaticales.

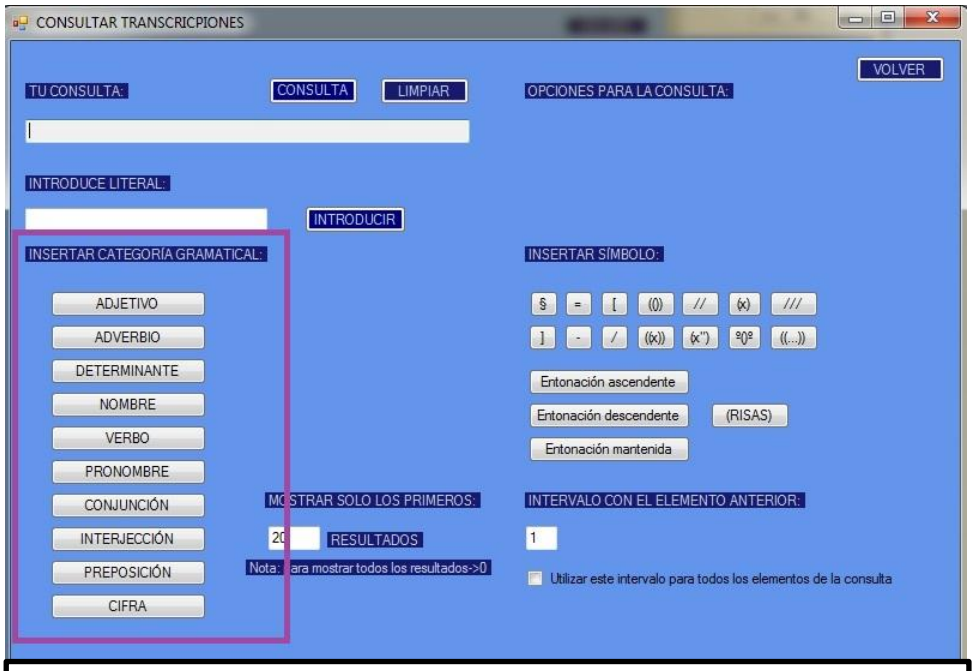

Figura 25. Detalle de la zona para seleccionar categorías gramaticales.

Vistas las opciones de consulta literal y consulta por categoría gramatical, nos queda familiarizarnos con la zona para incluir símbolos conversacionales en la consulta. Con el mismo mecanismo que para las consultas gramaticales, pulsando el botón que representa al símbolo que se pretende añadir a la consulta, éste se incluirá automáticamente. En la figura 26 aparece remarcada la zona de símbolos conversacionales.

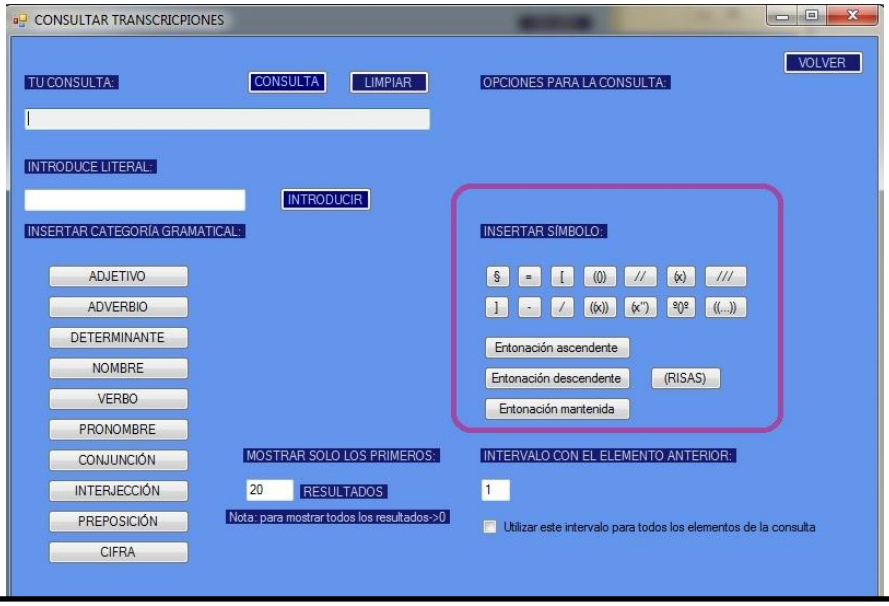

Figura 26. Detalle de la zona para seleccionar símbolos conversacionales.

Existen dos zonas de configuración adicionales, una representada por la imagen 27, y la otra representada por la imagen 28. En la primera,

se selecciona el número máximo de resultados que se desea recibir, denotando con el número 0 que se muestren todas las coincidencias. En la segunda, se debe especificar si los elementos de la consulta deben encontrarse de forma contigua o si pueden tener algún elemento intercalado. El valor del número de elementos intercalados que se permiten es el que debe introducirse en el cuadro de texto.

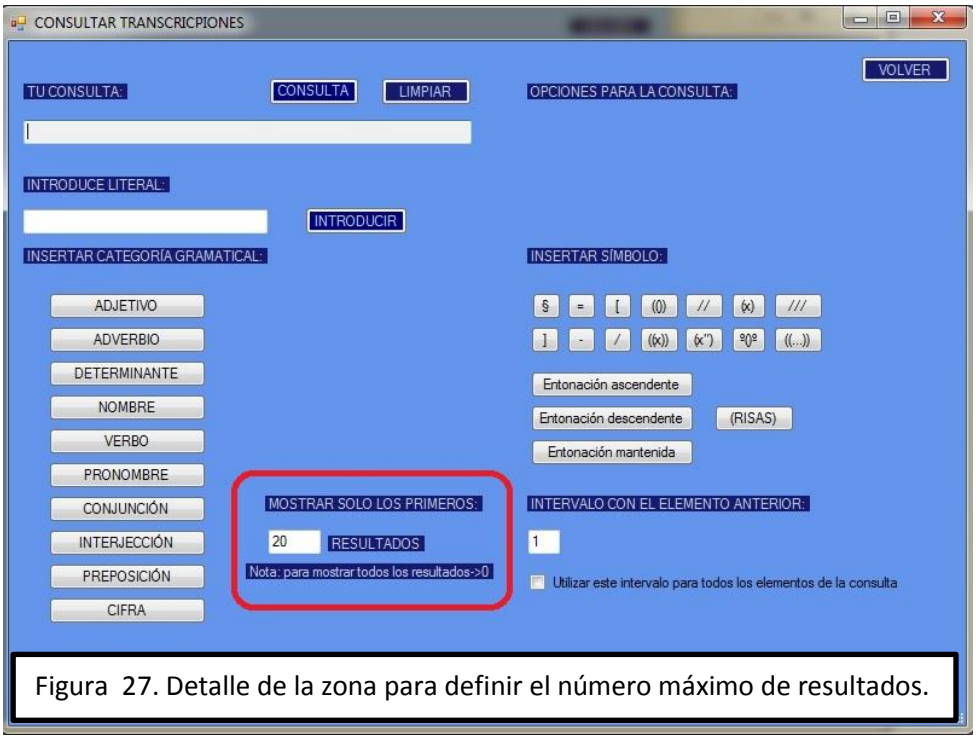

Figura 27. Detalle de la zona para definir el número máximo de resultados.

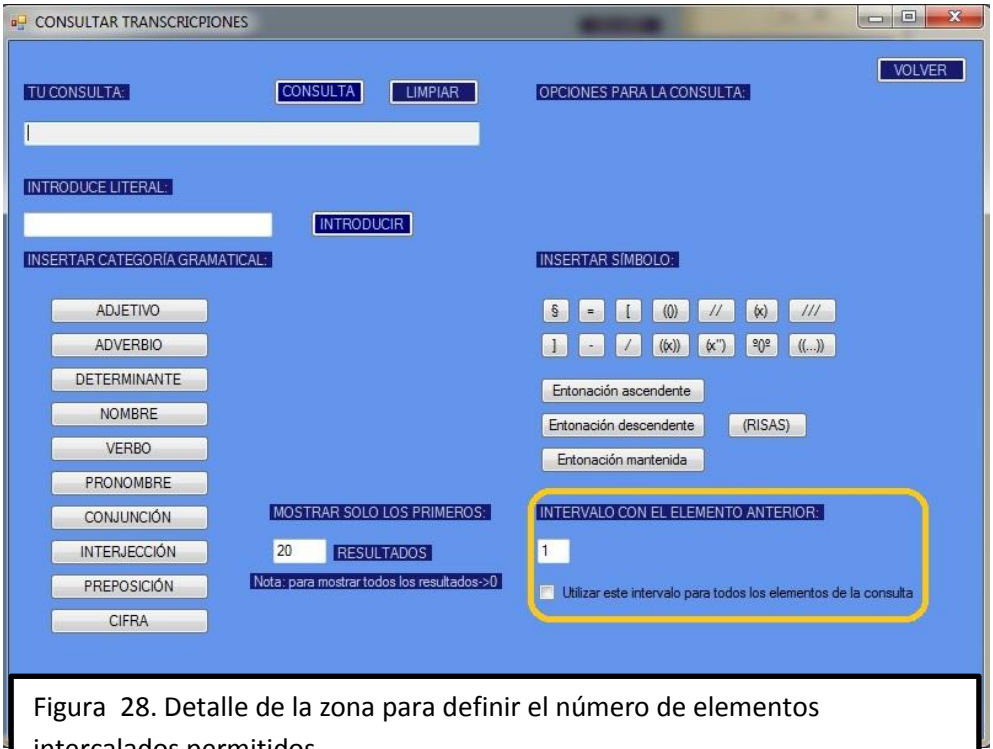

Figura 28. Detalle de la zona para definir el número de elementos intercalados permitidos.

En la última zona (figura 29) encontramos un cuadro de texto, bloqueado para escritura, en el que van apareciendo por orden de introducción los elementos que hemos introducido en las distintas zonas ya comentadas. El botón *limpiar* elimina todos los elementos introducidos hasta el momento en la consulta actual. Por otra parte, si pulsamos el botón *Consulta* formalizamos la consulta y el sistema pasa a procesarla y a mostrar los resultados obtenidos en un nuevo formulario.

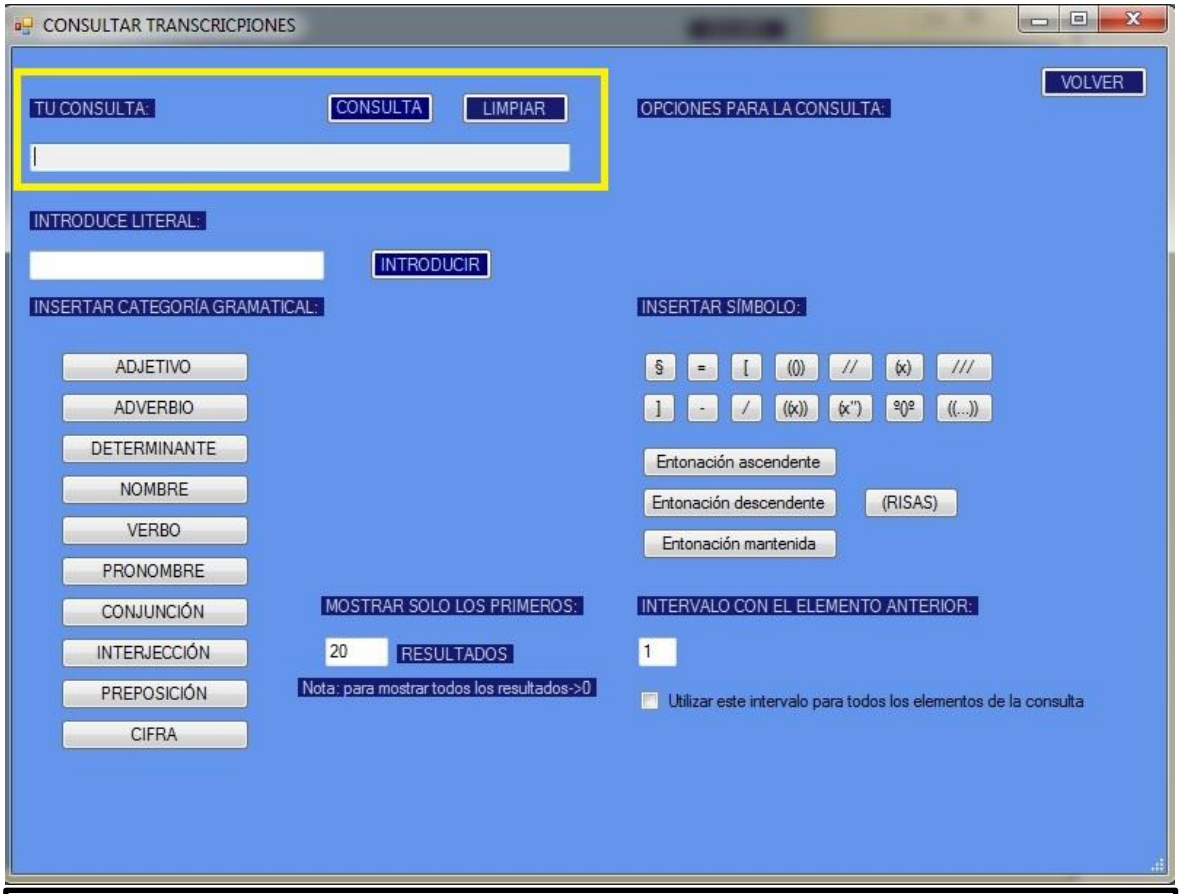

Figura 29. Detalle de la zona de visualización de la consulta actual.

Un nuevo cuadro de diálogo nos avisará si nuestra consulta no ha generado ningún resultado. Para los demás casos, se generará en la ventana resultado la sucesión de coincidencias. Cada resultado se separa del siguiente mediante una tabulación de guiones. Para la consulta combinada que aparece en la imagen 30, se genera el resultado mostrado en la figura 30.

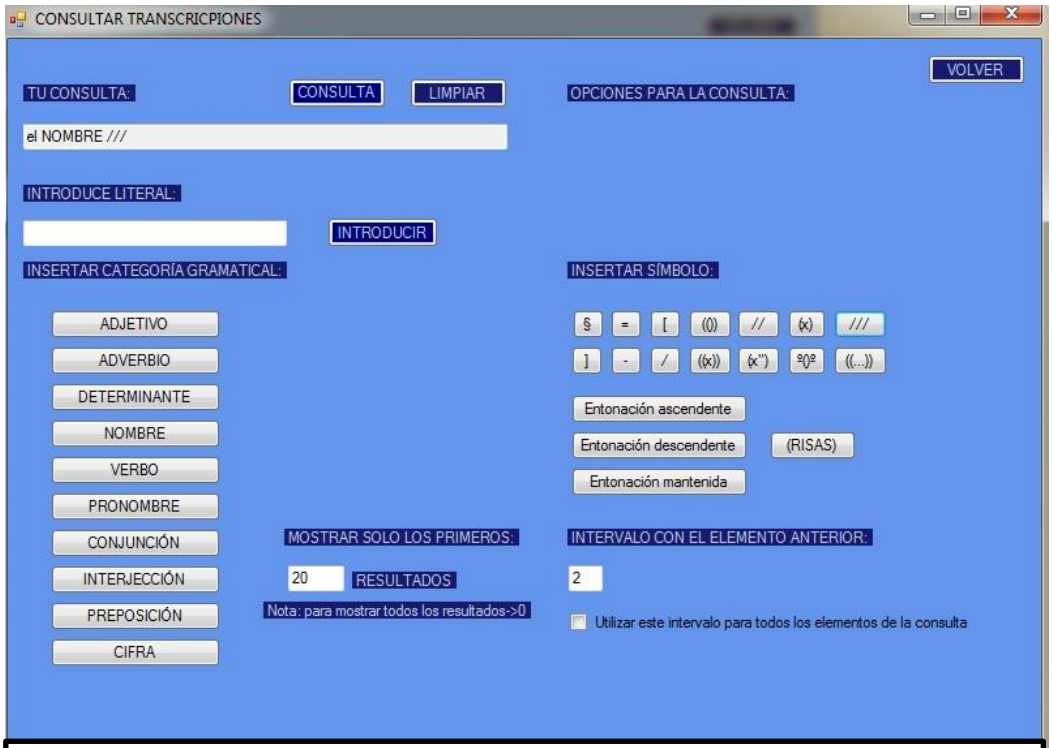

Figura 30. Ejemplo de una consulta combinada.

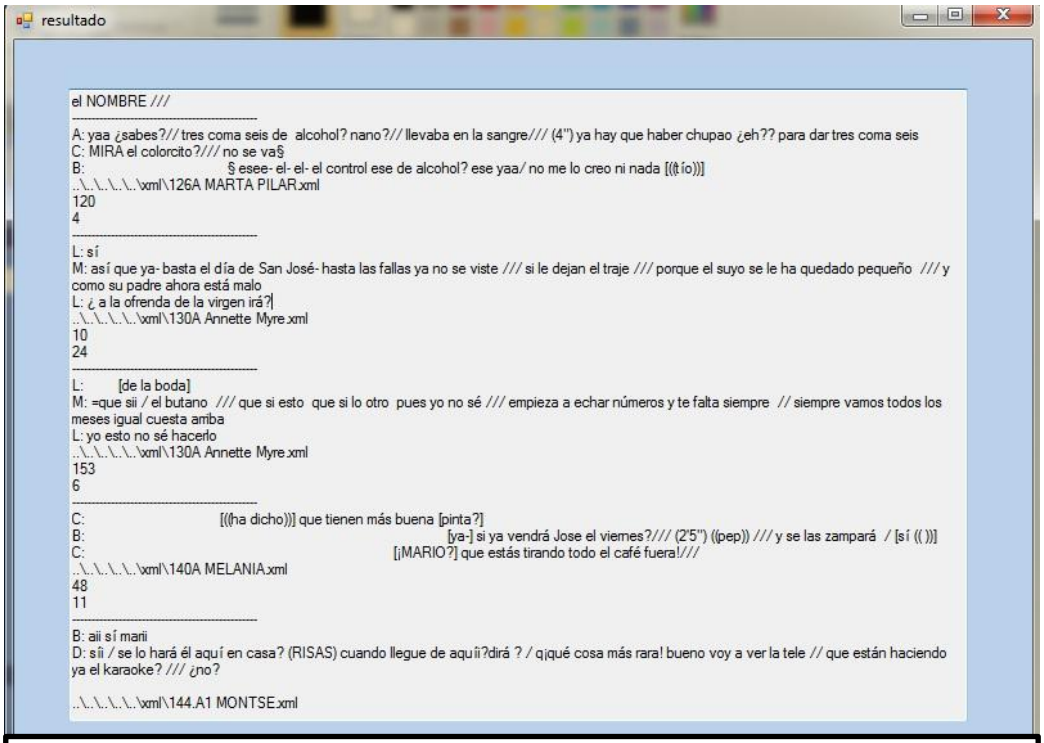

Figura 31. Vista general de la ventana de resultados.

La primera línea de la ventana de resultados representa siempre la consulta para la que se ha generado esa lista de coincidencias. En la imagen 32 comprobamos que aparece la misma información que se había introducido en el paso anterior (figura 31).

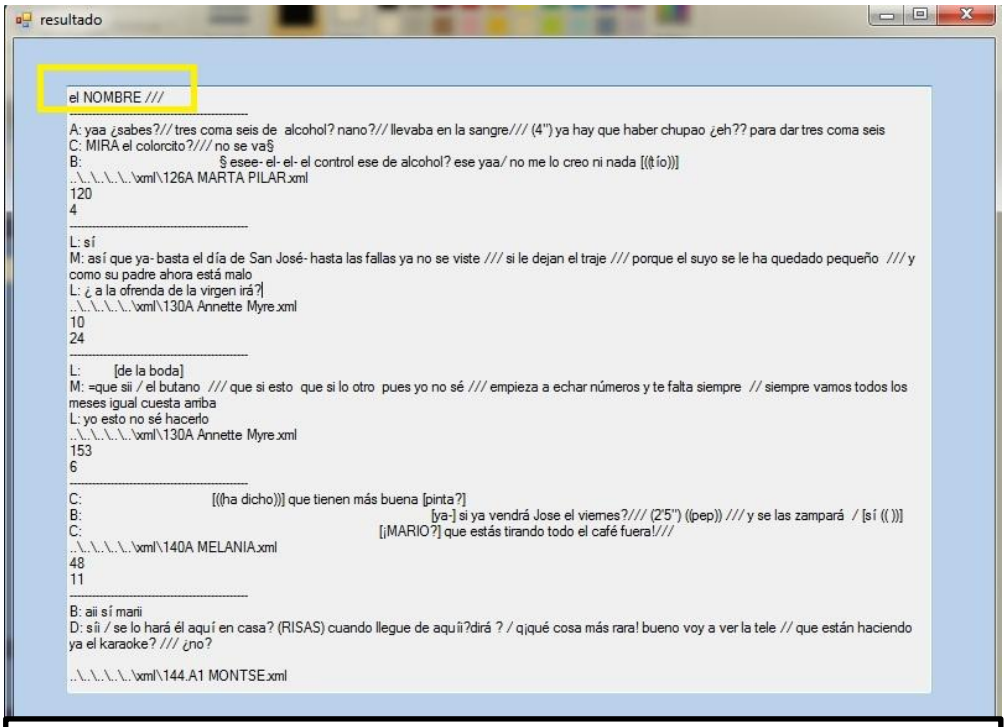

Figura 32. Detalle de la ventana de resultado.

A continuación, para cada resultado, se muestra la intervención donde se ha obtenido la coincidencia (imagen 33), acompañada de la intervención inmediatamente anterior y de la intervención inmediatamente superior (imagen 34).

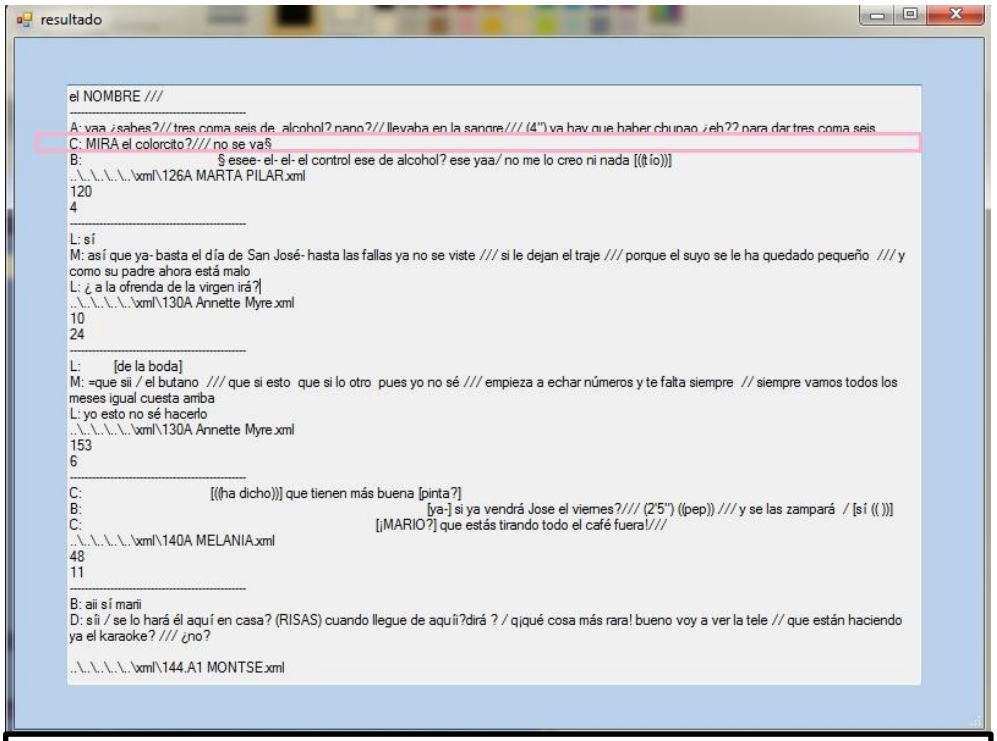

Figura 33. Detalle de la ventana de resultado.

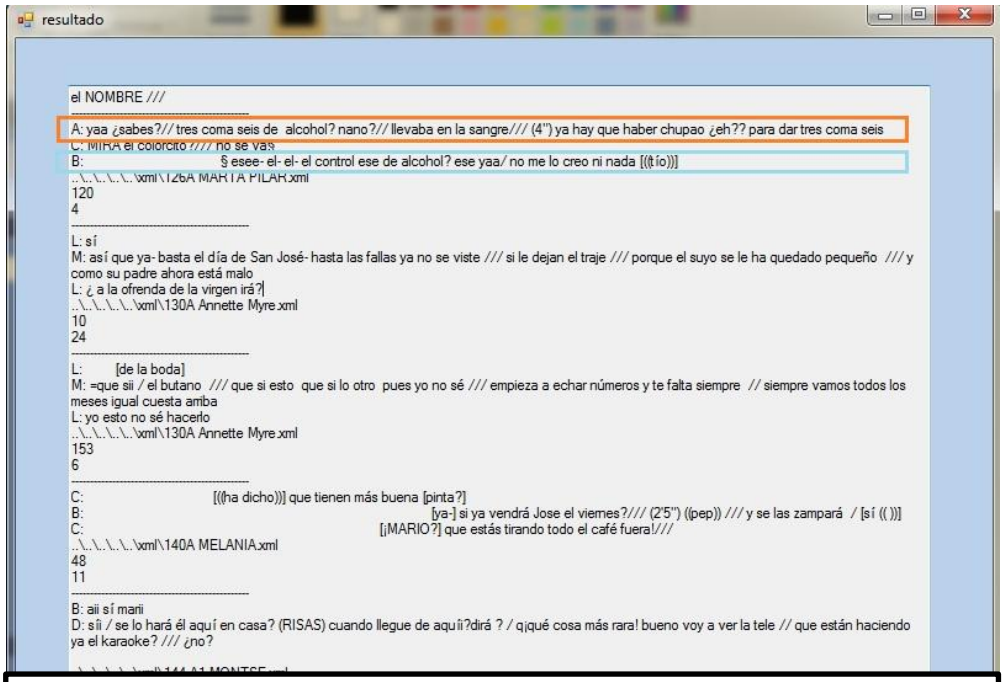

Figura 34. Detalle de la ventana de resultado.

La siguiente información se muestra con el fin de poder recuperar fácilmente información adicional, pues se proporciona la ruta al documento donde se ha encontrado esa coincidencia (imagen 33) e información para la recuperación de la intervención dentro de ese documento. En concreto, se muestra el número línea donde se ha producido la coincidencia y la posición del último elemento de la secuencia de la consulta dentro de esa línea (figura 35).

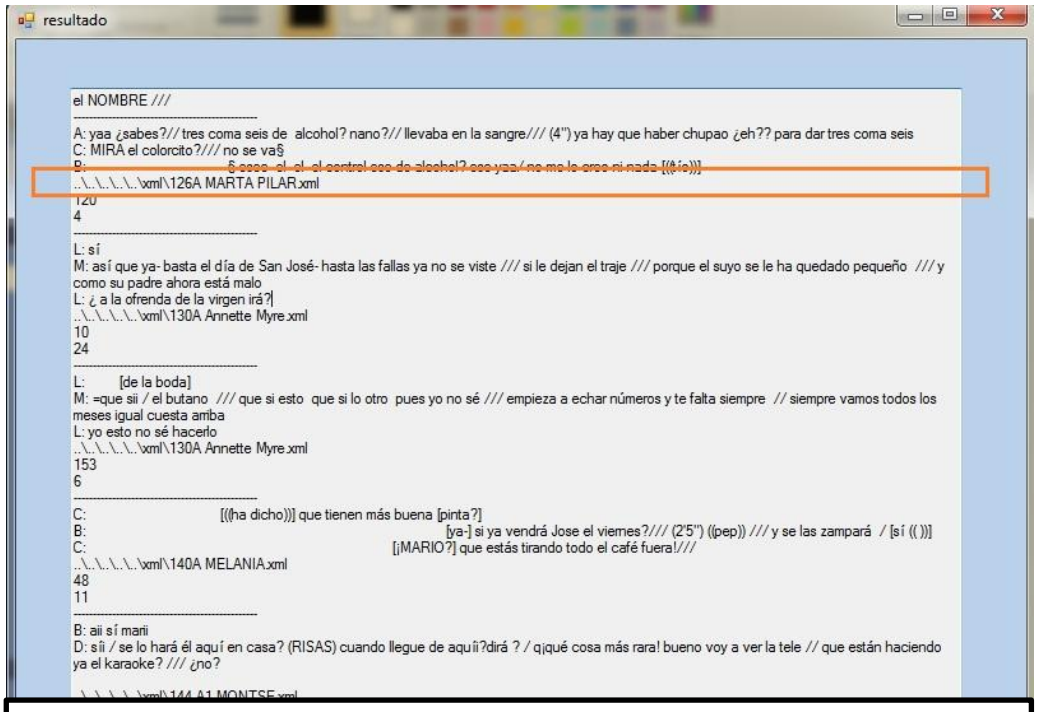

Figura 35. Detalle de la ventana de resultado.

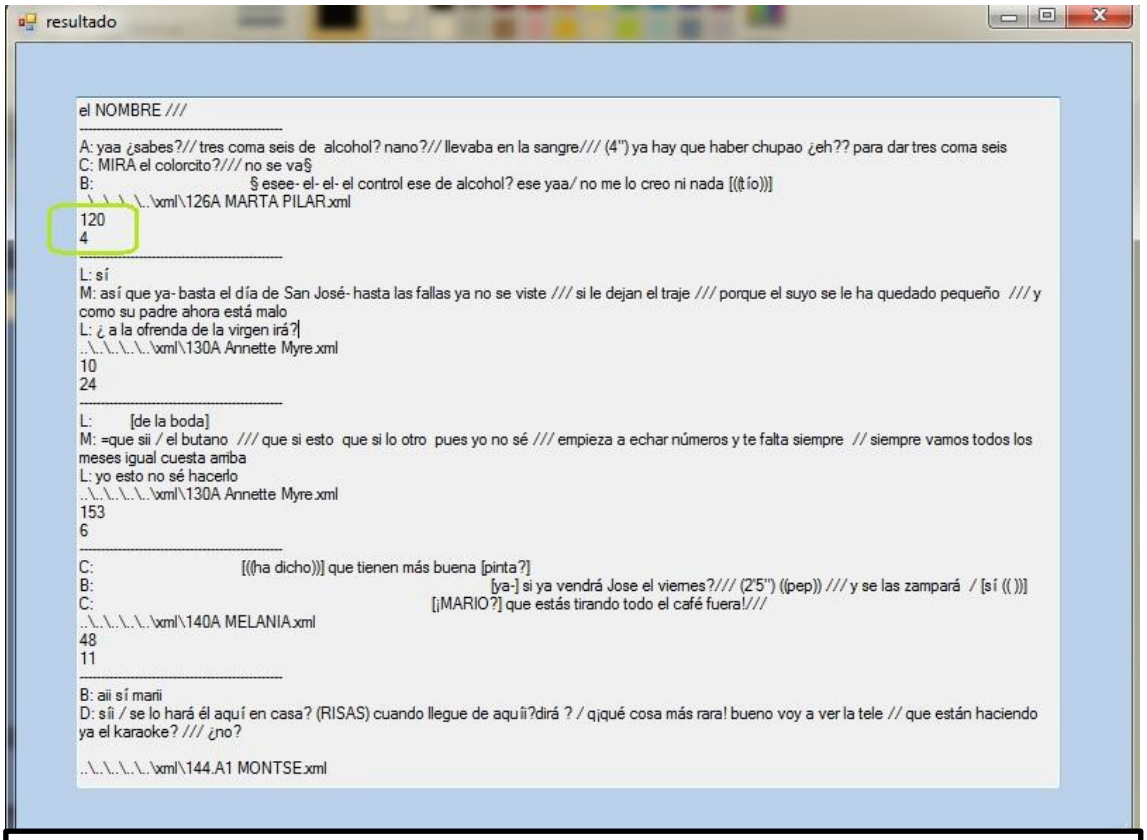

Figura 36. Detalle de la ventana de resultado.

Para cerrar esta ventana resultado y volver al formulario de Consulta es necesario un clic en el aspa roja de la parte superior.

8. Conclusiones

La conclusión principal del presente proyecto es que se han sentado las bases para poder procesar datos lingüísticos del corpus alrededor del que se ha trabajado.

De entre los etiquetadores disponibles, ninguno estaba preparado específicamente para trabajar con conversaciones orales transcritas, pues han sido entrenados con datos lingüísticos de textos literarios y periodísticos. Si en el futuro se lanzara un etiquetador que tuviera en cuenta las características de los corpus orales para etiquetarlos morfológicamente, sería conveniente hacer una revisión del trabajo aquí realizado para determinar si modificando el etiquetador externo utilizado como base se produciría una mejora significativa en el porcentaje de elementos etiquetados correctamente.

Respecto al diseño e implementación de un sistema que permitiera aplicar un etiquetado morfosintáctico y de fenómenos conversacionales al corpus, se ha conseguido satisfactoriamente reconocer cada una de las entidades que están presentes en los documentos de las transcripciones originales mediante un módulo de reconocimiento o tokenización. Además, se ha implementado con éxito una función capaz de interactuar con el etiquetador Freeling y asignar los resultados del análisis propuestos por éste a los elementos del corpus.

Mediante una función adicional se ha eliminado el alargamiento vocálico, una deficiencia muy presente en las transcripciones originales y sin la cual el análisis del etiquetador externo es significativamente más fiable.

Uno de los objetivos del proyecto, era el modificar la extensión en la que se almacenaban los datos lingüísticos. El formato utilizado hasta el momento por el grupo de investigación había sido el de almacenamiento en archivos .doc, pero en este proyecto, aparte de mantener los archivos originales, se han generado nuevos documentos estructurados en formato XML, consiguiendo así mantener la información de forma estructurada en archivos de extensión estándar.

Como se puede comprobar en la aplicación demo, se han implementado consultas avanzadas sobre los documentos XML generados, capaces de reconocer las categorías gramaticales de los elementos etiquetados, y los símbolos conversacionales presentes en las transcripciones originales.

Para concluir, como he comentado anteriormente, los resultados iniciales del etiquetado arrojan errores, en su mayor parte debidos a que el etiquetador utilizado no ha sido entrenado para trabajar con datos lingüísticos provenientes de conversaciones orales. Será necesario, por tanto, que proyectos posteriores se dediquen a perfeccionar este sistema base para aumentar la fiabilidad del etiquetado automático. Para ello será necesaria la estrecha colaboración entre los profesionales del campo de la lingüística y los de la computación, pues aún queda un largo camino por recorrer en este ámbito.

9. Referencias

- BIRD, Steven, *Natural Language Toolkit* [en línea] [consulta: septiembre 2010]. Disponible en: http://www.nltk.org/

- FREELING, *An open source suite of language analyzers* [en línea] [consulta: junio 2010]. Disponible en http://www.lsi.upc.edu/~nlp/freeling.

- LINGPIPE, *A suite of Java libraries for the linguistic analysis of human language* [en línea] [consulta: junio 2010]. Disponible en http://alias-i.com/lingpipe.

- MALTPARSER, *a system for data-driven dependency parsing* [en línea] [consulta: noviembre 2010]. Disponible en http://maltparser.org.

- MINIPAR, *a broad-coverage parser for the English language* [en línea] [consulta: noviembre 2010]. Disponible en http://webdocs.cs.ualberta.ca/~lindek/minipar.htm

- MORENO, L., PALOMAR, M., MOLINA, A., FERRÁNDEZ. A. *Introducción al procesamiento del Lenguaje Natural*. Alicante: Publicaciones de la Universidad de Alicante, 1999.

- MSDN, *Desarrollo de Office en Visual Studio*. [en línea] [consulta: julio 2010]. Disponible en http://msdn.microsoft.com/es-es/library/d2tx7z6d.aspx

- MSDN, *Herramienta de definición de esquema XML (Xsd.exe)*. [en línea] [consulta: agosto 2010]. Disponible en http://msdn.microsoft.com/es-es/library/x6c1kb0s(v=VS.100).aspx

- MSDN, *LINQ to XML*. [en línea] [Consulta: octubre 2010]. Disponible en http://msdn.microsoft.com/es-es/library/bb387098.aspx

- PALMER, D. *Tokenization and Sentence Segmentation*. Ed. Marcel Dekker, Inc. 2000.

- SÁNCHEZ, Aquilino. *Cumbre: Corpus Lingüístico del español contemporáneo. Fundamentos, metodología y aplicaciones de los corpus lingüísticos*. Alicante: Ed. SGEL, 1995

- SPROAT, R. *Lexical Analysis*. Ed. Marcel Dekker, Inc. 2000.

- VAL.ES.CO, *Valencia Español Coloquial* [en línea] [consulta: diciembre 2010]. Disponible en http://www.valesco.es/.

- WIKIPEDIA, *la enciclopedia libre* [en línea] [consulta: diciembre 2010]. Disponible en: http://es.wikipedia.org/wiki/Portada.