Desarrollo de un generador automático de ejercicios gramaticales de euskera

Dana Scarinci Zabaleta
Grado en Ingeniería Informática
Inteligencia Artificial

Emilio Sansano Sansano
Carles Ventura Royo

4 de enero de 2022

Título del trabajo:	*Desarrollo de un generador automático de ejercicios gramaticales de euskera*
Nombre del autor:	*Dana Scarinci Zabaleta*
Nombre del consultor/a:	*Emilio Sansano Sansano*
Nombre del PRA:	*Carlos Ventura Royo*
Fecha de entrega:	01/2022
Titulación:	*Grado en Ingeniería Informática*
Área del Trabajo Final:	*Inteligencia Artificial*
Idioma del trabajo:	*Castellano*
Palabras clave	*Basque language, natural language generation, functional programming*

Resumen del Trabajo

El objetivo de este proyecto es sentar las bases de un generador automático de ejercicios gramaticales de euskera de nivel inicial.

El programa genera ítems lingüísticos de tipo *fill-gap* o «rellenar los huecos», y permite practicar distintos fenómenos sintácticos y morfológicos. La característica principal de la aplicación es que construye oraciones simples pero siempre correctas y semánticamente coherentes.

El sistema se ha programado desde cero utilizando el lenguaje funcional puro Haskell, y en el presente trabajo se describen los patrones funcionales que han resultado útiles para la resolución de los problemas propios de la generación de lenguaje natural.

Abstract

The aim of this project is to lay the groundwork for an automatic generator of grammar exercises in Basque at a beginner's level.

The program generates linguistic fill-in-the-gap items, and allows the user to choose between different syntactic and morphological phenomena to practice. The main feature of the application is the fact that it builds simple sentences that are always correct and semantically coherent.

The system has been programmed from the ground up using Haskell, a pure and lazy functional programming language. The present work also describes the functional patterns that have proved useful in solving problems related with natural language generation tasks.

Índice

Índice de figuras

Índice de fragmentos de código

Índice de tablas

1 Introducción

1.1 Contexto y justificación del Trabajo

Durante el año académico 2019/20, cerca de 40.000 personas asistieron a cursos de euskera en *euskaltegis*[1] del territorio del País Vasco. De estos, un 27% tomaron clases de iniciación [1], es decir, de los niveles A1 y A2 del Marco de Europeo de Referencia para las Lenguas [2]. El perfil más habitual de este grupo es el de un adulto que ha llegado desde fuera del territorio vascoparlante (otras comunidades autónomas, otros países) para quedarse.

El sistema gramatical del euskera es muy diferente al de las lenguas vecinas, como el castellano y el francés. Por ello, en los niveles iniciales se dedica una proporción considerable del tiempo a realizar ejercicios puramente gramaticales. Podemos ver un ejemplo de estas actividades en *Ikasten* [3], un curso de euskera en línea y gratuito creado por el Departamento de Educación del Gobierno Vasco. *Ikasten* abarca un currículum desde el nivel inicial A1 hasta el nivel intermedio-alto B2, a través de unidades en las que se enseñan píldoras de gramática y vocabulario. Aunque ofrecen algunos ejercicios de comprensión oral y escrita, la gran mayoría de las tareas están enfocadas en repasar las construcciones gramaticales que se introducen en cada lección.

Estos ejercicios ilustran perfectamente los que se utilizan en los *euskaltegis*. Cada actividad consiste en una serie de ítems, y un ítem suele ser una oración sobre la que el estudiante debe realizar alguna acción. Por ejemplo, rellenar un hueco (ítems *fill-gap*) o rellenar un hueco mediante alguna de las opciones proporcionadas (ítems *multiple-choice*), como se observa en la Figura 1.

En esta plataforma, cada contenido gramatical cuenta con uno o dos ejercicios de 8-10 ítems, generados manualmente. Si el estudiante desea practicar más una estructura en particular, no dispone de medios para hacerlo. Lo mismo ocurre en el entorno tradicional del aula, donde el profesor o el libro de texto proporcionan los ejercicios.

En este contexto, un sistema automático de generación de *ítems* podría beneficiar tanto al colectivo de estudiantes como a la comunidad de profesores y creadores de recursos de aprendizaje. Para los segundos, significaría un ahorro notable de tiempo. Para los primeros, supondría la posibilidad de estudiar completamente a su ritmo, practicando más aquellos aspectos que les resultan más difíciles.

1 Escuelas de euskera

Figura 1: Ejemplos de tareas de Ikasten.net. Multiple choiche (izq) y gap-fill (dcha)

No solo eso, sino que sobre una solución de este tipo se podrían construir estrategias para optimizar el estudio, o incluso sistemas CALL (*Computer Assisted Language Learning)*. Por ejemplo, los profesores podrían obtener información personalizada sobre los puntos gramaticales que más dificultades plantean a cada alumno. También se podría implementar un método de repaso espaciado de estructuras gramaticales [4], [5]. Las opciones son interminables.

En la actualidad, ya existe una solución similar a la que proponemos en euskera, llamada ArikIturri [5]. Este sistema puede crear cuatro tipos de ejercicios: *fill-gap, multiple-choice*, corrección de errores y formación de palabras. Se engloba en una familia de generadores automáticos de ítems lingüísticos [7]–[11] que basan su funcionamiento en corpus de textos analizados morfológica y sintácticamente, y que utiliza las oraciones halladas en los textos, sin alterarlas, para crear los ejercicios.

Estos sistemas presentan algunos inconvenientes para nuestro objetivo. Por una parte, el propósito principal de este tipo de soluciones es asistir a la labor del profesor en la generación de ítems de examen, y siempre requieren un último paso humano, en el que se seleccionan manualmente los ejercicios aceptables. El porcentaje de ítems utilizables de ArikIturri, según su propia autoevaluación, es de alrededor del 85%. Otros sistemas reportan porcentajes más bajos: en torno al 50% o 60% [8], [11]. Aunque estos programas son increíblemente valiosos para reducir el tiempo invertido en diseñar ítems de examen, nosotros aspiramos a crear una herramienta que pueda ser usada de forma autónoma, con un 100% de ejercicios generados útiles.

Por otra parte, puesto que se trata de sistemas basados en textos reales, suele haber

una discrepancia entre los contenidos que se enseñan en la aulas y las construcciones disponibles en el corpus, sobre todo en los niveles iniciales [6], [12]. Asimismo, la calidad de los ejercicios depende en gran parte de la sofisticación de las herramientas utilizadas; en ArikIturri, por ejemplo, no pudieron incluir fenómenos enseñados en niveles de iniciación, como la inflexión de pronombres demostrativos o el caso genitivo, porque sus herramientas NLP no las identificaban correctamente. Por último, es prácticamente imposible seleccionar el nivel de los ejercicios, por la dificultad que entraña establecer el nivel de los textos naturales; en ArikIturri se utiliza un único corpus de nivel alto, constituido por 234 textos[2].

En un intento de adaptar el nivel de los ejercicios generados automáticamente a niveles iniciales, se ha intentado también producir un corpus de oraciones sencillas a partir de una gramática preexistente [12]. El problema de estos casos es el componente semántico: las oraciones generadas son estructuralmente correctas pero carecen de sentido.

Hasta donde hemos podido investigar, en la actualidad no existe ningún sistema capaz de generar ejercicios en euskera de forma completamente autónoma.

1.2 Objetivos del Trabajo

El objetivo de este proyecto es sentar las bases de un generador automático de ejercicios de euskera de nivel inicial (A1 y A2). Las unidades didácticas en la enseñanza de idiomas suelen organizarse en torno a un tema específico, que se trabaja mediante una o dos construcciones gramaticales seleccionadas; por ello, el sistema debe permitir elegir entre distintos fenómenos sintácticos o morfológicos para generar los ejercicios. Además, sería deseable que el sistema permitiese escoger un área de vocabulario específica para realizar los ejercicios.

Además, el sistema debería construir oraciones siempre correctas y semánticamente coherentes. Oraciones como «El cajón duerme la siesta» o «El león compra una tarta» nunca deberían presentarse.

Dados los límites temporales del proyecto, el objetivo no es crear un sistema completo que puedan generar todos los ejercicios del currículum de los niveles A1 y A2, sino sentar unos cimientos sólidos para la aplicación. En gran medida, es un trabajo de ensayo y error. A continuación detallo los objetivos específicos de este proyecto, que giran en torno a los módulos necesarios para obtener un sistema generador de ejercicios completamente funcional.

1. **Módulo sintáctico**. Formalizar algunos patrones sintáctico-semánticos tomados de un libro de texto de euskera de nivel A1, que representen fenómenos sintácticos lo más diferenciados posibles. Por ejemplo, tratar verbos transitivos e intransitivos.

2 Comprobar bien esto.

2. **Módulo semántico**. Organizar el vocabulario de varios campos semánticos del currículum del nivel A1, de forma que las estructuras de (1) puedan utilizar las unidades léxicas para generar oraciones coherentes.
3. **Módulo morfológico.**
    1. Programar un generador morfológico nominal y pronominal, que genere las declinaciones del euskera utilizadas en (1).
    2. Módulo morfológico. Programar un generador morfológico verbal, que genere las declinaciones verbales utilizadas en (1) y trate los casos irregulares pertinentes.
4. **Módulo de generación de ítems**.
    1. Integrar los componentes 1-3. Establecer el modo de acceso al sistema, de forma que, dadas unas restricciones sintácticas y semánticas, se generen oraciones que se adecúen a ellas.
    2. Implementar un generador de ejercicios *fill-gap* a partir de las oraciones generadas con el sistema. Esto equivale a identificar la palabra que debe ser eliminada en cada oración.

Por último, otro objetivo principal del trabajo es la evaluación del sistema. Este objetivo se materializará de dos formas. Por una parte, con un conjunto de tests unitarios y de integración que se irán generando a medida que avance el proyecto, y por otra parte, mediante una evaluación de todos o un subconjunto de los ejercicios generados por parte de un profesor de euskera.

1.3 Enfoque y método seguido

Por los motivos expuestos en la sección 1.1, vamos a diseñar un sistema de generación de oraciones desde los cimientos, pero no partimos desde cero. Por una parte, podemos usar como referencia las soluciones del campo de la generación automática de ítems (*Automatic Item Generation*, AIG) [13], que estudia cómo generar ejercicios de todo tipo: matemáticos, de razonamiento, de capacidades lingüísticas generales, etc.

En este campo no siempre es necesario producir acompañamientos textuales para los ejercicios. Sin embargo, en ocasiones sí lo es, por lo que también se han enfrentado al problema de la generación del lenguaje natural (*Natural Language Generation*, NLG). Debido a que en estos casos el dominio de los ítems es muy estructurado y muy restringido en cuanto a contenido, es bastante habitual el uso de plantillas lingüísticas (*templates, schemas*) sobre las que intercalar el contenido del ejercicio [14]. Este contenido, que se insertará en las distintas posiciones de las plantillas, está cuidadosamente diseñado y seleccionado para poder combinarse y así dar lugar a distintas tareas. También se usan métodos más complejos para modelar el dominio, que parten de teorías lingüísticas como la semántica de marcos [15]–[17]. Los

ejercicios de gramática de nivel inicial emplean un subconjunto bastante reducido del léxico y la gramática de una lengua; por ello, un sistema inspirado en estos modelos podría ser viable.

También podemos estudiar las soluciones halladas en el área del NLG desde una perspectiva más general. El objetivo de las aplicaciones de NLG suele ser automatizar procesos que tienen un componente lingüístico esencial, como la generación de predicciones meteorológicas o la implementación de *chatbots* de ayuda [18]. Estos sistemas son complejos y cuentan con módulos encargados de planificar un texto, generar representaciones intermedias no lingüísticas [19] y, finalmente, crear un texto a partir de estas. Este último módulo suele utilizar una gramática diseñada manualmente.

A diferencia de los *parsers*, que utilizan como punto de partida gramáticas tradicionales[3], las gramáticas de generación de textos se han basado en corrientes lingüísticas menos populares. Así, las gramáticas sistémicas funcionales, que organizan las estructuras de la lengua según su función [20]—[22] se han usado con éxito en proyectos de generación de textos [23]. Otro ejemplo son las gramáticas sintagmáticas nucleares (*head-driven phrase structure grammar,* HPSG) [24], [25] que, si bien tradicionalmente se han considerado gramáticas sintagmáticas, se adhieren bien a los preceptos de la gramática de construcciones (*Construction Grammar*, CxG) [25]. Esta última es una teoría lingüística que concibe la gramática como un conjunto de construcciones o patrones asociados a un significado, más que como un conjunto de reglas puramente formales. En cualquier caso, las gramáticas utilizadas en sistemas NLG suelen tener como objetivo cubrir la totalidad de la lengua, y diseñarlas puede llevar años a equipos de expertos. Asimismo, estas formalizaciones necesitan partir de representaciones semánticas intermedias para generar textos válidos.

En nuestro caso, ni necesitamos una gramática total del euskera, ni sería imprescindible utilizar representaciones intermedias. Esto se debe a que el contenido de la oraciones no es relevante, siempre y cuando sea coherente. Sin embargo, sí es reseñable la naturaleza de estas gramáticas: en oposición a las PSG, que establecen un límite muy claro entre estructura y semántica, las gramáticas utilizadas en NLG esta distinción es borrosa. Por un lado, una construcción se corresponde con una estructura asociada a un significado y, por otro lado, los elementos léxicos están fuertemente asociados a determinadas estructuras.

Los proyectos NLG también tienen en común la necesidad de organizar el léxico que representa el dominio. En este sentido cabe mencionar el *Generalized Upper Model* (GUM), una ontología diseñada para proporcionar una base semántica a las expresiones del lenguaje natural [27], [28].

En resumen, en este apartado y en los anteriores, hemos repasado algunos sistemas

3 *Phrase structure grammars* (PSG) o gramáticas de dependencias, normalmente

existentes tanto en el ámbito de generación de ejercicios (lingüísticos o no), como en el área más general de la generación de lenguaje natural, en busca de ideas sobre las que basar un programa de generación de ejercicios de euskera de niveles iniciales (A1, A2). Aunque ninguna propuesta se adapta a la perfección a nuestras necesidades, sí que hemos encontrado herramientas que pueden dirigir el diseño del sistema.

Por una parte, los sistemas existentes de generación de ítems lingüísticos ya disponen de soluciones para generar ejercicios a partir de oraciones correctas. Por otra parte, en las áreas de AIG y NLG en general, hemos observado la importancia de modelar la estructura gramatical no tanto como reglas, sino como plantillas o patrones asociados a una función o significado lingüístico. Por último, para la organización del léxico de forma que el sistema sea capaz de generar oraciones coherentes, podemos partir de trabajos existentes, como el GUM.

A partir de estas herramientas, aspiramos a desarrollar una solución novedosa para la generación automática de ejercicios gramaticales. Programaremos en Haskell [29], un lenguaje funcional, que utiliza un sistema de tipos estático, y que pospone la evaluación de las expresiones hasta el último momento (no estricto). El estilo de expresión declarativa de la programación funcional, mejorado con el sistema de guardas y *pattern-matching* de Haskell, puede agilizar enormemente la codificación de las reglas o patrones gramaticales.

En cuanto a la representación del componente semántico, hemos considerado dos opciones: el uso de ontologías [27], [30], [31], o el uso de un base de datos de grafos, como Neo4J. Puesto que en este caso prima la capacidad de recuperar resultados de la base de datos rápidamente, más que la necesidad de razonar sobre los elementos del dominio, utilizaremos una base de datos de grafos para representar el conocimiento.

1.4 Planificación del Trabajo

Hemos divido el proyecto en iteraciones. En cada iteración se trabajarán todos los módulos del programa, de forma que al final de cada una de ellas se obtenga una aplicación completamente funcional. Al terminar el primer ciclo, el programa podrá generar oraciones usando un único patrón sintáctico y un único campo semántico; al poner punto final al último ciclo, dispondremos de al menos tres patrones sintácticos y cuatro campos semánticos.

El periodo de trabajo en el código de la aplicación comienza el 13 de octubre de 2021 y termina el 13 de diciembre de 2021. La primera iteración comprende cinco semanas, mientras que a las iteraciones consecutivas se les han asignado menos recursos temporales. Esto se debe a que será en la primera fase cuando se establezcan los cimientos del programa y cuando se lleve a cabo la mayor reflexión en cuanto a la estructura más apropiada de cada módulo.

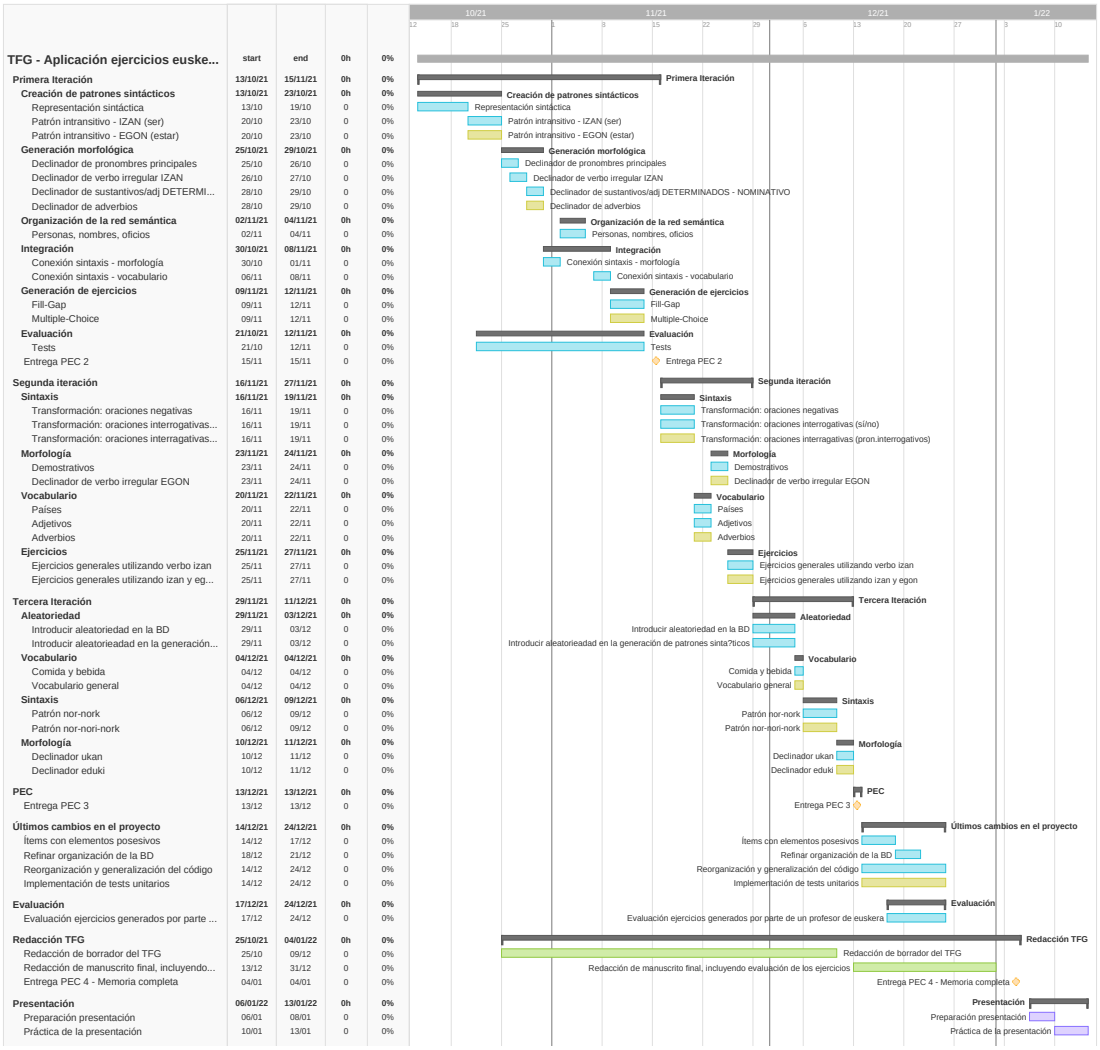

Figura 2: Diagrama de Gantt de la planificación temporal del proyecto

Una parte considerable del trabajo consistirá en hallar soluciones a problemas representacionales (cómo representar las estructuras sintácticas, cómo organizar el léxico) y operacionales (cómo acceder al sistema, cómo generar los ejercicios). Esto hace difícil estimar la duración de cada fase. Para garantizar el éxito del proyecto frente a esta incertidumbre, hemos tomado dos medidas. En primer lugar, en cada fase hemos clasificado las tareas en primarias y opcionales. Estas aparecen de color azul y amarillo, respectivamente, en el diagrama de Gantt de la Figura 2. En segundo lugar, como hemos indicado, al final de cada iteración tendremos un programa funcional; por ello, incluso eliminar una iteración completa no supondría un problema.

La redacción del Trabajo progresará a medida que avance el proyecto. Se ha organizado la escritura de los borradores de los capítulos de menor a mayor complejidad técnica. El periodo comprendido entre el 14 de diciembre y el 4 de enero se dedicará a la redacción y corrección del trabajo; además, durante este periodo llevará a cabo la evaluación, por parte de hablantes nativos de euskera, de una muestra de ejercicios lo más representativa posible.

1.5 Breve sumario de productos obtenidos

Junto con este documento se entrega el código de la aplicación de generación de ejercicios gramaticales. Aunque no era uno de los objetivos del proyecto, se ha implementado una sencilla interfaz de línea de comandos para poder demostrar los resultados obtenidos.

La aplicación se en encuentra en la carpeta `euskera-item-generation` y se entrega como un proyecto de Stack [32], una popular herramienta de creación de proyectos de Haskell.

```
Elige un ejercicio:
1  -  Izan: presente
2  -  Izan: pasado simple
3  -  Nor-nork: presente perfecto
4  -  Nor-nork: futuro
5  -  Posesivos
0  -  Salir
```

Figura 3: Interfaz de línea de comandos de la aplicación

Para ejecutarlo hay que instalar Haskell, Stack y lanzar una base de datos Neo4J. Las instrucciones detalladas para hacerlo se encuentran en el fichero `README.md`. Todos los *scripts* para preparar la base de datos se encuentran en la carpeta `db`.

1.6 Breve descripción de los capítulos de la memoria

La memoria se articula en torno a los distintos módulos de la aplicación; estos, a su vez, se han organizado en torno a los distintos campos lingüísticos implicados en una aplicación de generación de lenguaje natural: Morfología (Capítulo 3), Sintaxis (Capítulo 4) y Semántica (Capítulo 5).

En el Capítulo 2, *Visión general,* se expone la organización del código y se explica cómo explorarlo a través de la documentación.

En el Capítulo 3, *El módulo morfológico,* se explica cómo se han implementado las distintas clases de palabra y los rasgos gramaticales necesarios para derivar palabras a partir de sus raíces o lexemas.

El Capítulo 4, *El módulo sintáctico,* trata sobre las representaciones sintácticas escogidas para representar la estructura oracional, así como su motivación.

A continuación, el Capítulo 5, *El módulo semántico,* explica cómo se ha organizado el vocabulario en la base de datos, la lógica detrás de esta organización, y la motivación para la elección de una base de datos de grafos.

Por último, en el Capítulo 6, *Generación de ejercicios,* se explica cómo estos módulos funcionan en conjunto para dar lugar a oraciones y ejercicios; además, se tratan dos problemas importantes a los que nos hemos enfrentado en la generación de ítems: la generación de huecos, y la introducción de aleatoriedad.

Finalmente, en el Capítulo 7, *Conclusiones,* se hace una evaluación tanto del producto final entregado como del trabajo llevado a cabo durante el desarrollo del proyecto.

2 Visión general

Durante este proyecto hemos creado una aplicación para generar ejercicios gramaticales en euskera de tipo *fill-gap*. El objetivo no era producir una aplicación lista para ser usada, sino sentar las bases para poder crearla. Sin embargo, para poder demostrar las funcionalidades terminadas, se ha creado una sencilla interfaz de línea de comandos, en la que se pueden practicar cinco ejercicios (Figuras 3 y 14).

Los módulos de la aplicación se encuentran en la carpeta `src`, que tiene la estructura mostrada en la Figura 4. La aplicación de prueba se encuentra en la carpeta `app`.

En los capítulos que siguen explicamos la funcionalidad principal y la lógica detrás de cada uno de los módulos que componen la aplicación. También explicamos los patrones de programación funcional en general, y del lenguaje Haskell en particular, que se han utilizado para implementar las distintas funcionalidades. Sin embargo, ha sido imposible explicar la totalidad del código.

Una forma sencilla de adentrarse en el código es utilizar la documentación generada mediante la librería Haddock [33], que se encuentra en la carpeta `docs`. Está en formato html (se accede a través del fichero `index.html`) y contiene referencias cruzadas a los distintos módulos, así como enlaces al código fuente[4]. Esta documentación solo expone las funciones que se exportan en cada módulo, por lo que supone un resumen de las funciones principales del sistema.

```
src
├── BasicData.hs
├── EuskeraAppError.hs
├── Exercises
│   ├── Generation.hs
│   ├── ItemWithAnswer.hs
│   ├── RandomizeSentence.hs
│   └── UpdateSentence.hs
├── Features
│   ├── Adverbial.hs
│   ├── Nominal.hs
│   ├── Sentential.hs
│   ├── Verbal.hs
│   └── Wrappable.hs
├── Features.hs
├── Lexis
│   ├── Adjective.hs
│   ├── Adverb.hs
│   ├── Noun.hs
│   └── Verb.hs
├── Lexis.hs
├── Morphology
│   ├── Inflection.hs
│   ├── IrregularVerbs.hs
│   ├── NominalForms.hs
│   ├── Nominal.hs
│   ├── Utils.hs
│   └── Verbal.hs
├── Neo4J
│   ├── Config.hs
│   ├── Conversion
│   │   ├── Adjective.hs
│   │   ├── Noun.hs
│   │   ├── Utils.hs
│   │   └── Verb.hs
│   ├── Conversion.hs
│   └── Queries.hs
├── Neo4J.hs
├── Syntax
│   ├── Constituent.hs
│   ├── NominalPhrases.hs
│   ├── ProtoSentence.hs
│   ├── SentenceModels.hs
│   └── VerbalPhrases.hs
└── Syntax.hs
```

Figura 4: Estructura de la librería de la aplicación

4 Los enlaces a tipos y módulos de librerías externas a la aplicación no funcionan.

3 El módulo morfológico

morfología

f. Gram. Parte de la gramática que estudia la estructura de las palabras y sus elementos constitutivos

Este capítulo trata sobre el desarrollo del módulo morfológico, `Morphology`, cuya tarea principal es la derivación de palabras a partir de los lexemas —las raíces— de las palabras. Para flexionar una palabra es necesario conocer a qué clase pertenece, y qué información gramatical expresa; en las secciones que siguen explicamos cómo se ha representado esta información.

3.1 Clases de palabras

Observemos algunos hechos básicos:

La palabra «escribir» se presenta en distintas formas, como «escribí» y «escribimos». La palabra «niño», también, pero estas formas son distintas: «niña», «niños» o «niñas».

La palabra «escribir» se coloca es posiciones específicas de la oración; «niño» también. Las posiciones que puede ocupar «escribir» y las que puede ocupar «niño» no son intercambiables, como comprobamos a continuación.

(1) La niña escribió un programa de ordenador.

*(2) *La escribió niña un programa de ordenador.*[5]

La palabras «mirar», «buscar», «dormir» y «acontecer» pueden tomar las mismas formas que «escribir» y colocarse en las mismas posiciones. Y las palabras «perro», «teclado» y «soledad» pueden tomar las mismas formas[6] y ocupar las mismas posiciones que «niño».

De estos hechos básicos derivamos la necesidad de describir el vocabulario de las lenguas en clases (y subclases) de palabra[7]; en los ejemplos anteriores, estas clases son verbo y sustantivo. Cualquier herramienta NLG necesitará una representación de las clases de palabras suficientemente fina para cumplir sus objetivos; incluso las herramientas lingüísticas basadas en algoritmos de aprendizaje automático se benefician de estas representaciones [34], [35].

5 El asterisco (*) se usa en lingüística para indicar que una oración es agramatical.
6 Aunque no siempre todas. ¡La morfología es compleja!
7 Todas las lenguas tienen clases de palabras, pero la definición de estas clases es particular para cada lengua.

Nuestra aplicación define las clases de palabra con las que trabaja el programa en el módulo `Lexis`. Para cada clase de palabra se ha denifido un nuevo tipo (*type*). Lo explicaremos a través de un ejemplo, la clase de palabra `Nominal`, definida así:

```
data Nominal = Noun Noun
             | Pronoun
```

Código 1: Definición del ADT Nominal

Mediante la palabra clave `data` definimos un nuevo tipo llamado `Nominal`. Las instancias del tipo `Nominal` pueden ser sustantivos (`Noun`) o pronombres (`Pronoun`). En Haskell, cuando creamos un nuevo tipo de esta forma, creamos un tipo de dato algebraico (*algebraic data type*, ADT). Los ADT son clases constituidas por otras clases, y se utilizaron por primera vez en el lenguaje Hope, creado en la Universidad de Edimburgo en 1980 [36]. Ahora solo exploraremos el ADT tipo suma[8], utilizado para definir las categorías léxicas.

Hemos definido dos clases de palabra dentro del tipo `Nominal`: los nombres y los pronombres. Sintácticamente, estas dos clases de palabra se comportan de la misma forma, es decir, ocupan las mismas posiciones y realizan las mismas funciones gramaticales. Sin embargo, morfológicamente, son dos categorías relativamente distintas. Los pronombres son una categoría cerrada, es decir, formada por un número reducido de formas básicas y no extensible; por el contrario, los sustantivos son una categoría abierta, un grupo formado por innumerables palabras y en constante cambio. Podemos enumerar todas las formas de los pronombres en nuestro código, pero deberemos derivar las formas de los nombres programáticamente. En resumen, a ojos del módulo sintáctico desearemos tratar nombres y pronombres de la misma forma, pero desde el módulo morfológico querremos tratarlos de forma distinta.

Los ADTs de tipo suma son la herramienta ideal para esta situación, ya que nos permiten definir una clases como la unión disjunta de otras clases. En este caso, la clase `Nominal` puede ser o bien un sustantivo, o bien un pronombre. Para indicar de qué clase se trata, utilizamos los constructores, que son las palabras que aparecen en negrita en el Código 1: `Noun` y `Pronoun`. Un constructor puede o no recibir argumentos; si los reciben, debemos indicar de qué tipo serán en la definición; en este caso, `Pronoun` es un constructor sin argumentos, mientras que `Noun` toma una argumento de tipo `Noun`[9]. ¿Pero qué es `Noun`? Otro ADT que presentamos a continuación.

8 En la sección 3.2 trataremos el tipo producto.

9 No confundir el constructor `Noun` con el tipo `Noun`. Los tipos aparecen en las signaturas y del lado izquierdo de las definiciones de tipo. Los constructores se usan en el código, y aparecen en el lado derecho de las definiciones de tipo.

```haskell
data Noun = Count String Number
          | Mass String
          | Proper String
          | Anthroponym String
```

Código 2: Definición del ADT Noun

No todos los sustantivos se comportan de la misma forma. Por ejemplo, hay sustantivos contables, como «taza», con los que solemos encontrarnos tanto en singular como en plural. Sin embargo, los sustantivos incontables, como «mantequilla», tienden a aparecer solo en singular. Además, algunos sustantivos contables los preferimos en plural («gafas» o «lentejas»). Por otra parte, los nombres propios suelen tener un comportamiento algo distinto: no aceptan artículos, se declinan diferente, etc. Puesto que todos estos comportamientos son relevantes para generar oraciones, necesitamos que nuestras clases reflejen estas categorías. Así que hemos definido cuatro subclases de sustantivos:

- Sustantivos contables, creados mediante el constructor `Count`, que reciben dos argumentos, una cadena de caracteres que representa la raíz de la palabra, y un `Number`, que representa el número por defecto del sustantivo;
- sustantivos no contables, creados mediante el constructor `Mass`, que reciben un único argumento, la raíz;
- nombres propios, creados mediante el constructor `Proper`, y
- nombre propios de persona, creados mediante el constructor `Anthroponym`.

Se han generado todas clases de palabras necesarias para el sistema siguiendo una lógica similar a la que se ha expuesto para la clase `Nomina`. Presentamos en la Figura 5 un esquema de las clases de palabras definidas. Cada clase principal cuenta con su propio submódulo dentro del módulo `Lexis`. Todas las clases, aunque lingüísticamente motivadas, se han implementado únicamente si eran relevantes para algún módulo del programa (sintáctico, morfológico o de generación de ítems).

El uso de tipos de datos algebraicos por parte de Haskell, y la facilidad que proporciona para implementarlos, han sido de enorme utilidad en la definición de las clases de palabras y la posterior utilización de estas en el código. Por una parte, son muy fáciles de modificar y, sobre todo, de extender. Además, una vez definido un tipo, este se utiliza en las signaturas de las funciones, y el compilador será capaz de llevar a cabo inferencias basadas en tipos con ellas, lo cual agiliza enormemente la programación.

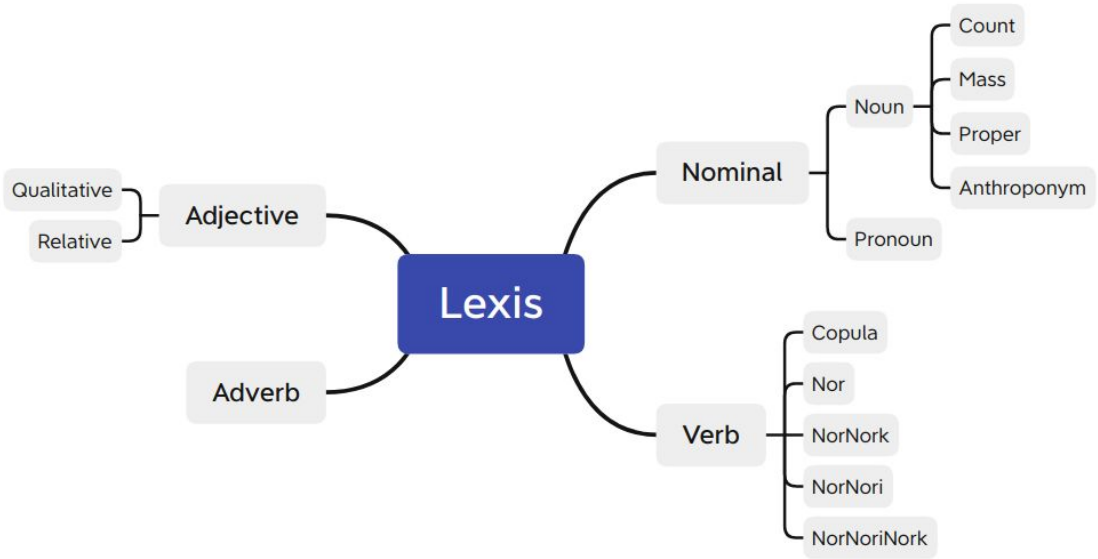

Figura 5: Clases de palabras del programa

3.2 Rasgos gramaticales

En el apartado anterior hemos mencionado que uno de los rasgos característicos de una clase de palabras son las formas que sus miembros pueden tomar. Por ejemplo, la forma de la 1ª persona del singular del presente de indicativo se crea añadiendo una «o» a la raíz del verbo: «escribo», «camino», «pienso». La segunda persona del singular en futuro simple se genera con «ás»: «escribirás», «caminarás» o «pensarás». La parte que no varía es la raíz de la palabra o lexema y las partes cambiantes son morfemas[10] o desinencias.

Los morfemas expresan significados que la lengua ha gramaticalizado, es decir, que ha dejado de expresar a través de palabras léxicas y ha incorporado en el sistema flexivo. El inventario de categorías gramaticales es particular de cada lengua, tanto en el número y tipo de categorías que cada una exhibe, como en las clases de palabras a las que estas se asocian. Por ejemplo, en castellano y en euskera el tiempo se expresa en el verbo, pero otras lenguas también lo expresan en los sustantivos.

El módulo `Features` contiene el inventario de rasgos gramaticales que utiliza el programa organizados en submódulos según la clase de palabra a la que están asociados. De nuevo, hemos utilizado ADTs de tipo suma para representarlos. En la mayoría de los casos, estos tipos son enumeraciones —ADTs definidos únicamente con constructores sin argumentos—, como la definición del rasgo de persona:

10 Morfemas flexivos, para ser exactos.

```
data Person = First
            | Second
            | Third
```

Código 3: Definición del ADT enumerativo del rasgo Person

Además de los rasgos que podemos encontrar en la gramática descriptiva del euskera, hemos añadido otras categorías necesarias para la correcta declinación de los constituyentes de la oración; tratamos este tema con más profundidad en la sección 3.2.2 *¿Palabras o rasgos gramaticales?*.

Cada clase de palabra suele expresar a través de sus morfemas un complejo de categorías gramaticales. Hemos representado estos grupos creando nuevos tipos. En concreto, hemos utilizado un ADT de tipo producto para aglutinar los rasgos gramaticales de cada clase gramatical. Veamos un ejemplo:

```
data NF = NF {
      person :: Person,
      number :: Number,
      grammCase :: Case,
      possessive :: Possessive,
      determiner :: Determiner,
      coord :: Bool
      }
```

Código 4: Definición del ADT de los rasgos nominales

El ADT `NF` (por *Noun Features*) se instancia mediante el constructor `NF`. Si el ADT suma representaba la unión disjunta, el ADT producto representa el producto cartesiano de los posibles valores de sus campos, justo lo que necesitamos para especificar los rasgos gramaticales de una palabra en una ocasión concreta. Para definir este tipo hemos usado *record syntax*, donde cada campo se define como `palabraClave :: tipoDelCampo`. La palabra clave genera funciones similares a los *getters* y *setters* que conocemos de la programación orientada a objetos, con la única diferencia de que no es posible alterar un objeto en programación funcional: si necesitamos cambiar el valor de un campo, debemos crear un nuevo valor.

Los rasgos principales se ubican al principio de la descripción, y en este caso son `Person`, `Number` y `Case`[11]. El resto de los campos, en muchos casos, tomarán un valor por defecto y serán ignorados. Esto significa que estamos usando algo más del espacio estrictamente necesario para representar los rasgos gramaticales, ya que siempre representamos *todos* los que podrían ser necesarios. Esta solución, sin embargo, simplifica muchísimo la lógica del programa. Por otra parte, nunca creamos un complejo de rasgos, como `NF`, directamente, sino que utilizamos las instancias predefinidas en el modulo `BasicData`, y modificamos los rasgos que nos interesan a

11 El lector atento percibirá que en euskera, el género no es una categoría gramatical del sustantivo.

partir de estas.

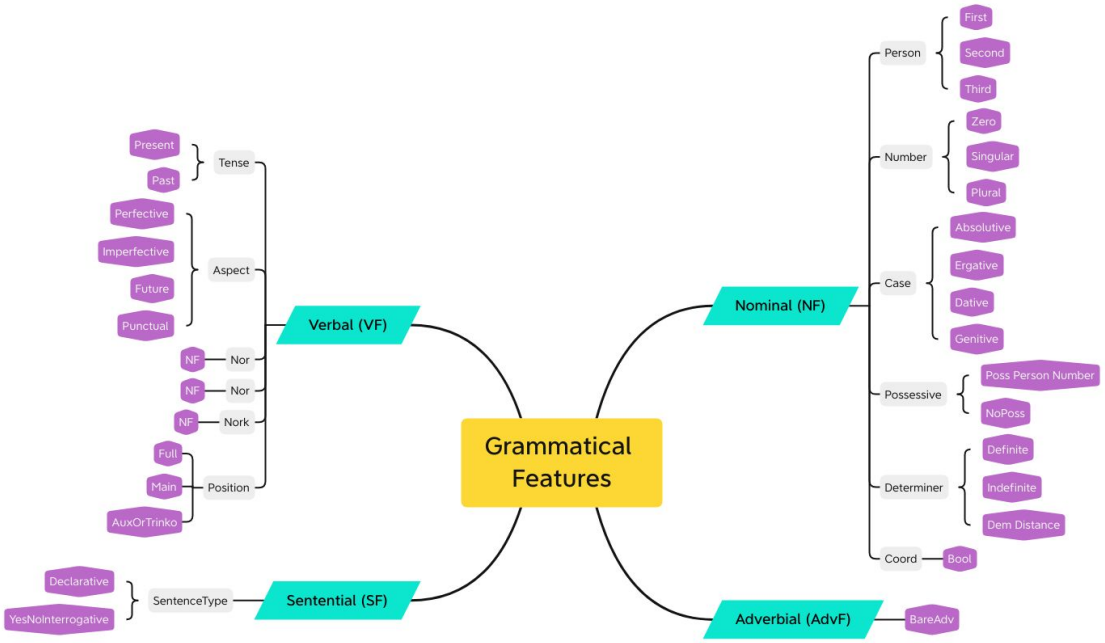

Figura 6: Rasgos gramaticales del programa hasta el momento

En la Figura 6 presentamos el esquema actual de los rasgos gramaticales introducidos en el programa. En turquesa representamos los complejos de rasgos, y entre paréntesis su nombre y su constructor (que son el mismo). En el siguiente nivel reflejamos los tipos que conforman el complejo; por último, en morado, enumeramos qué elementos pueden constituir cada tipo (sus constructores).

Como se puede observar, por el momento los adjetivos no cuentan con un complejo de rasgos gramaticales propio; esto se debe a que utilizan directamente el de la categoría nominal. Sin embargo, en un futuro habría que asignarles una categoría propia para poder transmitir información sobre el grado del adjetivo, utilizada en construcciones comparativas («Ese es **mejor**», «Este ordenador es **más rápido**») .

3.2.1 Persona y número como monoides

Observemos las siguientes oraciones:

(1) ***Tú*** *<u>has ido</u> al cine.*

(2) ***Tú y yo*** *<u>hemos ido</u> al cine.*

En (1), el sujeto «Tú» y el verbo «has ido» comparten rasgos gramaticales: 2ª persona del singular. Es un ejemplo clásico de concordancia entre sujeto y verbo, que se da muchas lenguas[12].

Sin embargo, en (2), «tú» es 2ª persona del singular y «yo» es 1ª persona singular,

12 En euskera, además, el verbo también concuerda con el complemento directo y el indirecto. Lo veremos más adelante.

mientras que el verbo «hemos ido» es 1ª persona del plural. ¿Qué ocurre aquí?

El sujeto no es «tú» y no es «yo», sino «tú y yo», que es equivalente a «nosotros», 1ª persona del plural. Podemos considerar la operación (representada por el símbolo <>) que consiste en combinar dos rasgos gramaticales del mismo tipo (persona o número, en este caso) para resumir su valor en uno solo. En el caso de persona, la regla es que si combinamos cualquier valor con 1ª persona, el resultado de la operación es 1ª persona; descartando 1ª persona, cualquier valor combinado con 2ª persona resulta en 2ª. A continuación detallamos todos los posibles casos:

- 1ª <> _ = 1ª
- _ <> 1ª = 1ª
- 2ª <> 2ª = 2ª
- 2ª <> 3ª = 2ª
- 3ª <> 2ª = 2ª
- 3ª <> 3ª = 3ª

Observamos que:

1. La operación definida es asociativa: (a <> b) <> c = a<> (b <> c)
2. La operación es conmutativa: a <> b = b <> a
3. Existe un elemento neutro (3ª)

Un semigrupo es una estructura algebraica que define una operación binaria asociativa para combinar dos valores de un tipo y obtener un valor del mismo tipo (propiedad 1). Un monoide es un semigrupo con un elemento neutro [37] (propiedad 3). La concatenación de listas y la suma son ejemplos clásicos de monoides, cuyos elementos neutros son la lista vacía y el 0, respectivamente. Pero la combinación de rasgos gramaticales de persona y número también puede representarse mediante esta estructura algebraica. En concreto, tenemos un monoide conmutativo (propiedad 2).

Haskell utiliza *type classes* para representar estructuras algebraicas útiles, como los monoides. Más adelantes veremos otras.

Type class significa, literalmente, clase de tipo. Un *type class* es muy similar a una interfaz de Java: define la signatura de los comportamientos que la definen. En Haskell estas definiciones utilizan variables de tipo (*type variables*), como la `a` o la `m` en el siguiente ejemplo:

```
class Semigroup a where
 (<>) :: a -> a -> a

class Semigroup m => Monoid m where
 mempty :: m
```

Código 5: Definición mínima de las clases de tipo Semigroup y Monoid

La clase `Semigroup a` se define por la función (<>), que resulta en un valor de tipo `a` al ser evaluada con dos valores de tipo `a`.

Cualquier función que utilice variables de tipo[13] será una función polimórfica ya que, en principio, cualquier tipo puede ocupar el lugar de la variable de tipo. Esto, por otra parte, limitará en gran medida los posibles comportamientos de dicha función. Las clases de tipo, adicionalmente, permiten el polimorfismo estático o sobrecarga de métodos: un mismo método puede tener comportamientos distintos según el tipo con el que se utilice.

En los fragmentos de Código 6 y 7 vemos cómo se han definido las instancias de `<>` y `mempty` (el elemento neutro) para `Person` y `Number`. Cabe señalar que hemos tenido que añadir la categoría de número `Zero`, para poder contar con un elemento neutro.

```
instance Semigroup Person where
 First <> _ = First
 _ <> First = First
 Second <> _ = Second
 _ <> Second = Second
 _ <> _ = Third

instance Monoid Person where
 mempty = Third
```

Código 6: Rasgo Person: Definición de typeclasses Semigrupo y Monoide

La operación `<>` (o `mappend`) se utiliza, por el momento, en la generación de sintagmas nominales coordinados. Una de las ventajas de definir las instancias de `Monoid`, es que obtenemos una operación gratis: `mconcat`, que reduce una lista de valores aplicando `mappend` recursivamente.

```
instance Semigroup Number where
Zero <> Zero = Zero
Zero <> Singular = Singular
Singular <> Zero = Singular
_ <> _ = Plural

instance Monoid Number where
mempty = Zero
```

Código 7: Rasgo Number: Definición de typeclasses Semigrupo y Monoide

3.2.2 ¿Palabras o rasgos gramaticales?

Al implementar un sistema generador del lenguaje natural, debemos tomar numerosas decisiones representacionales. Una de ellas es cómo representar las palabras que pertenecen a llamadas categorías funcionales (en oposición a las categorías léxicas).

13 En las signaturas podemos encontrar tipos y variables de tipo; los primeros siempre comienzan en mayúscula y suelen tener un nombre que describe el tipo, los segundos comienzan en minúscula y suelen consistir en una única letra.

Las categorías léxicas son aquellas que tienen contenido semántico y referencial, es decir, significados que hacen referencia a entidades (concretas o abstractas) que identificamos en el mundo que nos rodea. Por el contrario, las categorías funcionales tienen contenido únicamente gramatical.

Normalmente, las categorías léxicas vienen representadas en lexemas, y las categorías gramaticales en morfemas. Sin embargo, todas las lenguas disponen de palabras con contenido únicamente gramatical. Por ejemplo, en castellano tenemos los artículos y preposiciones.

La decisión que hay que tomar en cuanto a estas palabras es ¿dónde las representamos? ¿Junto con palabras léxicas como «sol» y «anochecer»? ¿Junto con rasgos gramaticales, como género y número?

En nuestro caso, hemos decidido representarlas como categorías gramaticales. Algunos de estos casos, son

- Los artículos y pronombres demostrativos
- Los artículos y pronombres posesivos
- La coordinación
- La negación (parcialmente)

Esto significa que la representación sintáctica contiene estas categorías como rasgos, no como palabras o constituyentes. Serán simplemente un rasgo más a tener en cuenta durante el proceso de inflexión, que exploraremos más adelante, se transformarán en palabras al instanciar la oración.

Esta decisión simplifica enormemente la aplicación en varios aspectos. Por una parte, la representación sintáctica contiene menos elementos. Además, facilita la aleatorización de estos rasgos. Por ejemplo, para generar variedad en la oraciones podemos crear sintagmas nominales determinados, indeterminados o acompañados de demostrativos («el libro», «un libro», «este libro», respectivamente). Aleatorizar el determinante como un rasgo, y posteriormente establecer cómo llevar a cabo la inflexión de ese rasgo es mucho más sencillo que insertar las unidades léxicas que los representan y programar el comportamiento flexivo de cada combinación de una palabra gramatical y palabra léxica.

Por otra parte, no es necesario almacenar y recuperar estos elementos en la base de datos, ya que no son elementos léxicos. En gran parte, esto es posible gracias a que todas las palabras gramaticales pertenecen a las categorías léxicas cerradas: grupos de número reducido y cuyos miembros son muy estables, ni se añaden ni se eliminan. Y, como sabemos, reducir el número de consultas a la base de datos se traduce mejor rendimiento.

3.3 Flexión

La flexión es el proceso a través del cual, a partir un lexema y de un conjunto de rasgos, generamos una palabra flexionada. La flexión es la tarea principal del módulo `Morphology`. Hemos dado un rodeo para llegar hasta aquí, puesto que debíamos visitar primero la justificación de la representación de las clases de palabra y los rasgos gramaticales. Hasta ahora, hemos presentado por separado las clases de palabras y los complejos de rasgos gramaticales. Necesitamos una representación que los unifique en dos aspectos:

- Desde el punto de vista sintáctico, sería deseable que todas las clases de palabra formaran parte de un mismo grupo, porque en algunos aspectos las trataremos uniformemente. Por ejemplo, serán las hojas de los árboles de los árboles sintácticos (ver 4.1 *Las representaciones sintácticas)*.
- Desde el punto de vista morfológico, necesitamos acceder a la raíz y a los rasgos simultáneamente, para poder derivar la forma adecuada.

El tipo `Lexis` cumple esta función. Su definición es simple: cada clase de palabra cuenta con un constructor y toma dos argumentos: una clase de palabra y un conjunto de rasgos gramaticales:

```
data Lexis = N Nominal NF
           | V Verb VF
           | Adj Adjective NF
           | Adv Adverb AdvF
```

Código 8: Definición del tipo Lexis

Con un sistema de tipos fuerte y coherente como el que hemos definido hasta ahora, programar la derivación morfológica se vuelve enormemente intuitivo y casi trivial. La flexión de cualquier lexema comienza con una llamada a la función `inflect`, cuya signatura simplificada es `inflect :: Lexis -> InflectedWord`.

A partir de aquí, utilizando la asociación de patrones (*pattern matching*), vamos descomponiendo cada palabra, aumentando la granularidad con la que la analizamos hasta hallar la función que generará la palabra flexionada que deseamos.

```
inflect :: Lexis -> Either EuskeraAppError InflectedWord
inflect (N noun nf) = inflectNominal nf noun
inflect (V verb vf) = inflectVerb vf verb
inflect (Adj adj nf) = inflectAdjective nf adj
inflect (Adv (A adv) advf) = Right adv
```

Código 9: Definición de inflect

Vamos a reconstruir el proceso con un ejemplo simple. Tenemos la instancia:

```
ane = N (Noun (Proper “Ane”))
        (NF Third Singular Ergative NoPoss Definite False)
```

1. Evaluamos la expresión `inflect ane`.

    1. Tenemos un *pattern match* con el caso (**N** `noun nf`). Por lo tanto, llamamos a `inflectNominal`.

```
inflectNominal :: NF -> Nominal -> Either EuskeraAppError InflectedWord
inflectNominal nf EmptyNoun = Right ""
inflectNominal nf Pronoun
        | possessive nf /= NoPoss = inflectPossPronoun nf
        | determiner nf /= Definite = inflectDemPronoun nf
        | otherwise = inflectPronoun nf
inflectNominal nf (Noun n)
        | getNoun n == "" = Left $ EmptyString "inflectNoun: Empty proper noun"
        | otherwise = inflectNoun nf n
```

Código 10: Definición de inflectNominal

2. Evaluamos la expresión `inflectNominal` con los argumentos `noun=Noun (Proper “Ane”)` y `nf=NF Third Singular Ergative NoPoss Definite False`. Puesto que tenemos un `Noun` y no un `Pronoun` o `EmptyNoun`, llamamos a `inflectNoun`.

```
inflectNoun :: NF -> Noun -> Either EuskeraAppError InflectedWord
inflectNoun nf (Proper noun) = inflectProper nf noun
inflectNoun nf (Anthroponym noun) = inflectProper nf noun
inflectNoun nf (Count noun num) = inflectCommon nf noun
inflectNoun nf (Mass noun) = inflectCommon nf noun
```

Código 11: Definición de inflectNoun

3. `InflectNoun` separa los casos según la subclase de sustantivo. En este caso, decidirá utilizar `inflectProper`.

```
inflectProper :: NF -> Lexeme -> Either EuskeraAppError InflectedWord
inflectProper nf noun
        | grammCase nf == Absolutive = absolutiveUndetermined noun
        | grammCase nf == Ergative = ergativeUndetermined noun
        | grammCase nf == Dative = dativeUndetermined noun
```

Código 12: Definición de inflectProper

4. `InflectProper` determina de qué caso se trata, y llama a la función final. Aquí podríamos hacer *pattern matching* de la clase `NF`, pero es muy compleja y sería difícil de programar y leer. Gracias a que hemos utilizado *record syntax* para definir `NF`, podemos utilizar el método accesor `grammCase`. El único rasgo que nos interesa para declinar nombres propios es su caso gramatical (los sustantivos comunes son mucho más complejos). Puesto que `ane` es caso

ergativo, llamamos a `ergativeUndetermined`.

```
ergativeUndetermined :: Lexeme -> Either EuskeraAppError InflectedWord
ergativeUndetermined =
          Right . ifEndsInVowel (++ "k") (ifEndsInR (++ "rek") (++ "ek"))
```

Código 13: Definición de ergativeUndetermined

5. `ergativeUndetermined` contiene las instrucciones para declinar `ane`. En concreto: si la palabra acaba en vocal, añadimos «k» al lexema (la raíz de la palabra); si acaba en «r», añadimos «rek»; en cualquier otro caso, añadimos «ek». Cabe mencionar que se puede llegar a esta función desde otros caminos que no sean nombres propios.

Hemos visto un ejemplo detallado de cómo el programa va desgranando la información, haciendo preguntas cada vez más precisas que le guían hasta la regla que es necesario aplicar según la clase de palabra y ciertos rasgos gramaticales. Hemos comprobado qué fácil es hacer las preguntas gracias al sistema de tipos y a la sintaxis de guardas y *pattern matching*.

Este es un ejemplo muy claro de cómo, cuando programamos funcionalmente, nos centramos en identificar qué información necesitamos y qué transformaciones hay que realizar sobre los datos.

4 El módulo sintáctico

sintaxis

f. Gram. Parte de la gramática que estudia el modo en que se combinan las palabras y los grupos que estas forman para expresar significados, así como las relaciones que se establecen entre todas esas unidades.

Del mismo modo que el módulo morfológico es el responsable de generar palabras bien formadas a partir de una raíz y unos rasgos gramaticales asociados a ellas, el módulo sintáctico es el encargado de establecer el orden de los constituyentes y las relaciones entre ellos (concordancia) a partir de las representaciones recibidas. Por lo tanto, hemos escogido las representaciones sintácticas a partir de las necesidades del programa.

Un función esencial de este módulo será la generación distintos tipos de oraciones a partir de los mismos elementos. Por ejemplo, a continuación presentamos una oración afirmativa (1), negativa (2), interrogativa (3) e interrogativa negativa (4). Estos son los cuatro tipos de oraciones que la aplicación genera actualmente.

(1) El gato ha saltado desde el tejado.

(2) El gato no ha saltado desde el tejado.

(3) ¿Ha saltado el gato desde el tejado?

(4) ¿No ha saltado el gato desde el tejado?

Hay al menos dos motivos por los que esta funcionalidad es importante. Por una parte, la transformación puede ser un ejercicio en sí mismo; por ejemplo, es muy común en los ejercicios de nivel inicial recibir una oración afirmativa y transformarla a negativa. Por otra parte, los ítems generados para un mismo ejercicio deberían ser lo más variados posible, y parte de esta variabilidad se consigue mediante estructuras oracionales diversas.

4.1 Las representaciones sintácticas

Desde el punto de vista tanto lingüístico como computacional, existen distintas opciones de representación sintáctica. Los árboles de constituyentes (*phrase structure trees*) y los de dependencias (*dependency trees*), que ilustramos en las figuras 7 y 8 son muy comunes y cada uno tiene motivaciones teóricas distintas [38], [39]. Además, ambos se han utilizado extensivamente en aplicaciones NLP.

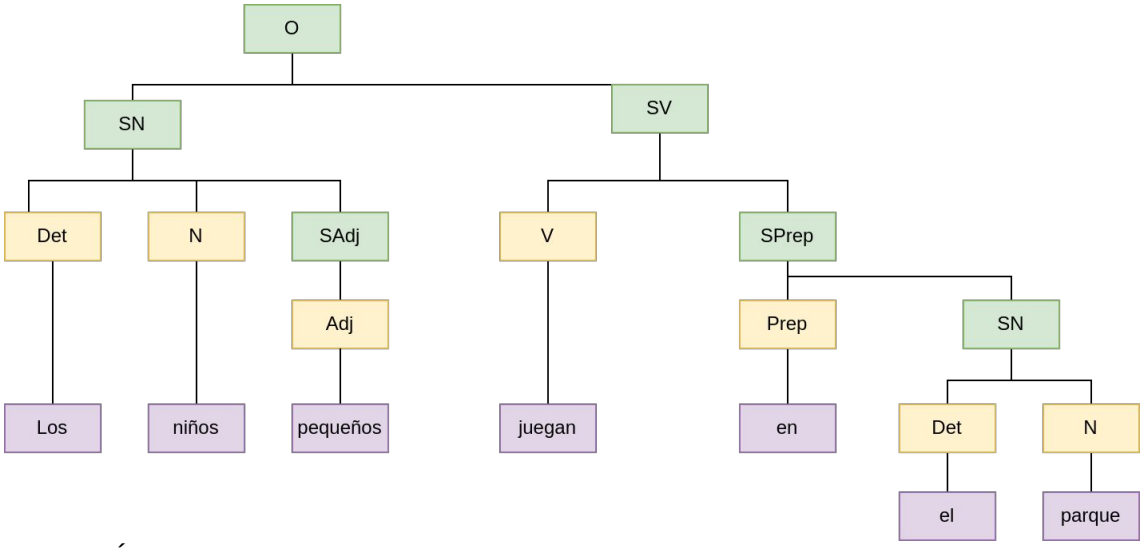

Figura 7: Árbol de constituyentes de la oración «Los niños pequeños juegan en el parque»

Los árboles de constituyentes representan la idea de que las palabras de una oración se organizan formando grupos (constituyentes), que a su vez pueden contener otros grupos, etc. Son, por lo tanto, estructuras jerárquicas y recursivas. Los árboles de dependencias representan relaciones directamente entre parejas de palabras. Estas relaciones son asimétricas —hay un elemento regente y uno dependiente— y están opcionalmente etiquetadas.

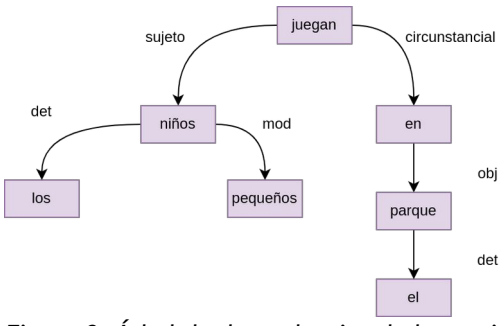

Figura 8: Árbol de dependencias de la oración «Los niños pequeños juegan en el parque»

Una diferencia importante entre los árboles de constituyentes y los de dependencias es que los primeros representan el orden de los constituyentes en la oración, mientras que los segundos no lo hacen.

Hemos optado por representar las oraciones mediante árboles de constituyentes principalmente por este motivo, ya que el objetivo final de una representación sintáctica, para el programa, es ser transformada en una una secuencia de caracteres. Si recorremos un árbol de constituyentes en profundidad primero, transformando los terminales en cadenas de caracteres a medida que lo hacemos, generaremos la oración representada. La tarea de generar una oración a partir de un árbol de dependencias, por el contrario, no es trivial, ya que el orden en que visitamos los elementos del árbol no se corresponde con el orden en el que deberán ser representados finalmente.

Sin embargo, puesto que nuestro objetivo no es la representación descriptiva de oraciones, sino la generación de oraciones y ejercicios, utilizamos árboles de constituyentes adaptados a nuestras necesidades. En la Figura 7, encontramos etiquetas de tres tipos distintos, que se corresponden a tres categorías distintas: sintagmas o constituyentes (verde), clases de palabra (amarillo) y palabras flexionadas

(morado). Veamos la definición del árbol sintáctico del programa y cómo se corresponde con esta Figura.

```haskell
data Constituent = S [Tag] SF [Constituent]
                 | VP [Tag] VF [Constituent]
                 | NP [Tag] NF [Constituent]
                 | L [Tag] Lexis
```

Código 14: Definición del árbol de constituyentes (Constituent)

En un primer vistazo nos damos cuenta de que podemos crear un `Constituent` a través de distintos constructores: `S`, `VP`, `NP`, y `L`. Los tres primeros se corresponden con los elementos verdes de la Figura 7: representan a los constituyentes Oración (`S`, *sentence*), sintagma verbal (`VP`, *verbal phrase*) y sintagma nominal (`NP`, *nominal phrase*). Estos constructores toman tres argumentos; por el momento ignoraremos el primero, una lista de etiquetas, que se usará para la generación de ejercicios. El segundo argumento es un complejo de rasgos gramaticales, y el tercero una lista de constituyentes. Gracias a este último argumento, hemos creado un tipo recursivo, es decir, un tipo que se puede definir mediante una referencia a sí mismo.

En concreto, hemos generado un *rose tree (*o árbol de rosas*)* [40], [41], un árbol que puede tener un número arbitrario y variable de ramas. Esta propiedad permitirá que cada constituyente esté formado por un número variable de elementos. Los *rose trees* no necesitan definir un caso base, puesto que se podría utilizar la lista vacía para esta función. Sin embargo, conceptualmente sí lo necesitamos.

Nuestro caso base es `L`, que sintetiza los elementos amarillos y morados del árbol de dependencias. Su elemento principal es la clase `Lexis`, que, como ya hemos visto en el apartado 3.3 *Flexión*, contiene una clase de palabra con su raíz y los rasgos gramaticales necesarios para representarla.

Existen restricciones respecto al tipo de constituyentes obligatorios y opcionales que cada tipo de constituyente debe contener. Por ejemplo, esperamos que una oración contenga un `VP`, que un `VP` contenga un verbo, que un `NP` contenga un sustantivo, etc. Sin embargo, estas restricciones no se han implementado a nivel de tipos, sino que será la lógica del programa la encargada de generar oraciones correctamente construidas. Además, esta flexibilidad es útil; veamos un ejemplo.

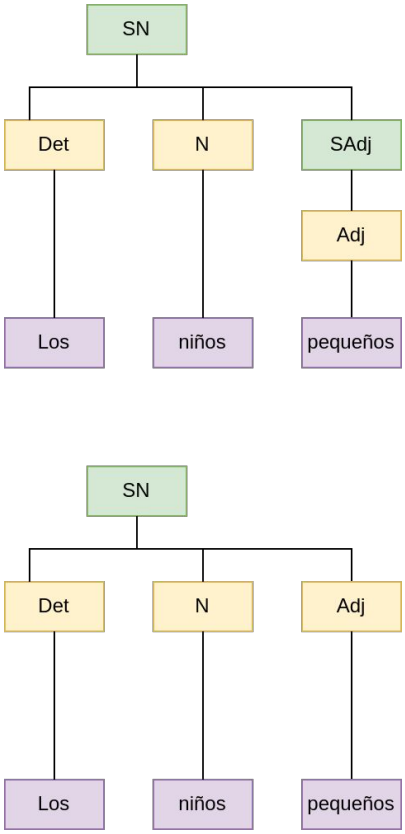

Figura 9: Representación sintáctica descriptiva vs. representación sintáctiva simplificada

Si analizamos bien la Figura 7, tal vez nos daremos cuenta de que una palabra léxica siempre genera su propio sintagma (como «pequeños» o «parque»). Aunque esta representación tiene una justificación lingüística[14], nos parece que no tiene sentido engrosar la representación con el único motivo de ser fieles a la teoría. Por ello, siempre que sea posible, representaremos los constituyentes como en árbol de abajo de la Figura 9. Estos árboles se pueden considerar aberrantes teóricamente, pero son más fáciles de generar y gestionar, además de ocupar menos memoria que sus contrapartes correctos. Debido a esto, para representar las oraciones generadas hasta el momento no ha sido necesario introducir sintagmas adjetivales o adverbiales en la definición de `Constituent`. Cuando sea necesario, la ampliación de la clase será tan sencilla como añadir los respectivos constructores.

4.2 La generación de estructuras sintácticas

En la introducción de este capítulo hemos hablado de que la sintaxis representa las relaciones que se establecen entre constituyentes, principalmente de la concordancia entre el verbo y sus argumentos. El tipo `Constituent`, aunque incorpora rasgos gramaticales, no garantiza de ninguna manera que estas relaciones se establezcan. También hemos mencionado la necesidad básica de generar distintos tipos de oraciones a partir de un mismo modelo de oración. En la mayoría de las lenguas, incluido el euskera, al transformar una oración en negativa o interrogativa, el orden de los constituyentes cambiará. El tipo `Constituent` no es flexible en este sentido.

El tipo `Constituent` es útil para generar cadenas de caracteres a partir de una representación. Pero necesitamos un componente capaz de generar estas representaciones; para ello contamos con el tipo `ProtoSentence` y el módulo `Syntax.SentenceModels`.

`ProtoSentence`, como su nombre indica, es una proto-oración, el objeto anterior a una oración. Contiene toda la información lingüística necesaria para generar una estructura sintáctica bien formada y será el cruce de caminos entre los distintos módulos del sistema.

La definición actual de `ProtoSentence` se muestra en el Código 15. Los campos `norkCons`, `norCons`, `noriCons` y `attribute` se corresponden con los argumentos verbales. Distintos verbos piden distintos argumentos (de hecho, las subclases de verbo están definidas según este criterio, ver Figura 5, pág. 14). Aunque todos los campos son obligatorios, los distintos verbos solo utilizan un subconjunto de los argumentos posibles. Esto implica que un objeto `ProtoSentence` siempre ocupa más memoria de lo estrictamente necesario; como contrapartida, ganamos en generalidad, ya que el punto de partida para la generación de un ítem será siempre un

14 Aunque «pequeños» solo sea una palabra, podría transformarse en «muy pequeños» o «pequeños y traviesos», que sí necesitarían un sintagma adjetival para su representación.

`ProtoSentence` definido por defecto. La definición del ejercicio específico solo tendrá que modificar los campos relevantes para el ítem en cuestión[15]. Esta representación, además, garantiza que al manipular un `ProtoSentence` no sea necesario comprobar si un campo está presente o no.

```
data ProtoSentence = Sentence {
      norCons :: Constituent,
      attribute :: Constituent,
      noriCons :: Constituent,
      norkCons :: Constituent,
      mainVerb :: Verb
      sentType :: SentenceType,
      negated :: Bool,
      tenseS :: Tense,
      aspectS :: Aspect
    }
```

Código 15: Definición del tipo ProtoSentence

A partir de un `ProtoSentence`, las funciones del módulo `Syntax.SentenceModels` que se encargan de:

- implementar la concordancia entre distintos constituyentes (sujeto-atributo, sujeto-objecto-verbo);
- generar `Constituents` correctamente ordenados, según el tipo de oración que se quiera generar, y
- introducir algunas etiquetas (*tags*) con información sobre el constituyente; esta información será utilizada en el proceso de introducir huecos para generar ejercicios.

El flujo del programa es similar al introducido en la presentación de inflexión morfológica. A partir de una función de entrada (`createSentence`), se comprueba el tipo de verbo, ya que este es el criterio que más afecta a la estructura de la oración. A partir de aquí, se llama a la función correspondiente, que se encargará de implementar la concordancia y el orden según el tipo de oración (si es interrogativa o no, si es negativa o no).

Como se puede observar en el Código 16, en el estado actual el programa solo genera oraciones con verbos copulativos (el verbo «ser») y oraciones con verbos de tipo `NorNork`. Estas últimas son equivalentes a las oraciones transitivas del castellano, oraciones que tienen un sujeto y un objeto (alguien hace una cosa, alguien compra una cosa, alguien lleva una cosa, etc.), con una diferencia importante: en euskera, el verbo concuerda tanto con el sujeto como con el objeto.

15 Estrictamente hablando, no modificamos ningún objeto. Creamos un objeto nuevo a partir del anterio.

```
createSentence :: ProtoSentence -> Sentence
createSentence s = case mainVerb s of
       Nor {} -> error "Unimplemented"
       Copula {} -> copula s
       NorNork {} -> norNork s
       NorNori {} -> error "Unimplemented"
       NorNoriNork {} -> error "Unimplemented"
```

Código 16: Definición de createSentence

4.2.1 Extender ProtoSentence

Entre los constituyentes listados en los campos de `ProtoSentence` no encontramos unidades no argumentales, como los que aparecen subrayados en los siguientes ejemplos.

(1) *Ayer María comió fideos con palillos.*

(2) *Juan ha limpiado la cocina cuidadosamente.*

(3) *Hemos comprado el jarrón en el rastro.*

No se había definido en el alcance inicial del proyecto generar oraciones con este tipo de constituyentes, pero sí se espera hacerlo en el futuro. Esto requerirá aumentar todavía más los campos de `ProtoSentence`. A diferencia de los argumentos verbales, estos constituyentes no son obligatorios, por lo que será necesario cerciorarse de si uno de estos constituyentes está presente o no para incluirlo en la estructura sintáctica. Para implementar esta funcionalidad, todos los constituyentes no argumentales representados en `ProtoSentence` deberían ser del tipo opcional `Maybe Constituent`. En la sección 5.3 se habla en profundidad de estos tipos y de cómo se usan en la actualidad en el programa.

5 El módulo semántico

semántica

f. Ling. Disciplina que estudia el significado de las unidades lingüísticas y sus combinaciones.

semántica léxica

f. Rama de la semántica que estudia el significado de las palabras, así como las diversas relaciones de sentido que se establecen entre ellas.

En el Capítulo 3 hemos explicado cómo generar palabras bien formadas a partir de lexemas o raíces de palabras, pero ¿de dónde obtenemos esos lexemas? En este módulo tratamos este aspecto de la generación de oraciones.

La motivación principal del componente semántico no es únicamente almacenar las palabras que utilizará el programa, sino representar las relaciones entre las distintas categorías semánticas de una forma que permita generar oraciones que tengan sentido con la menor intervención posible por parte del programador.

5.1 La base de datos

5.1.1 Elección de una base de datos

Para generar oraciones congruentes, es necesario almacenar el vocabulario de tal forma que podamos recuperar elementos compatibles entre sí. Ilustraremos esta problemática con dos ejemplos:

1. Verbos y argumentos relacionados

Queremos generar oraciones con verbos de tipo `NorNork`. Estos verbos tienen dos argumentos: un agente que realiza la acción, en el caso ergativo (*nork*), y un objeto sobre el que esta se realiza, en el caso absolutivo (*nor*). Por simplicidad, el agente siempre será un pronombre. Podríamos almacenar en la base de datos todos los verbos de tipo `NorNork` («comer», «leer», «comprar», etc.) por un lado, y objetos de distintas clases (comidas, publicaciones, ropa, etc.) por otro. Si lo hiciéramos así, ¿qué nos impediría generar oraciones como las siguientes?

(1) Yo he comido una camiseta.

(2) María ha leído esa película.

(3) Nosotros hemos comprado las matemáticas.

Para evitar estas oraciones, la base de datos debe modelar las relaciones que se dan en

el mundo real hasta donde sea relevante para el programa.

2. Todas las palabras que completan una estructura

Deseamos recuperar todos los lexemas que pueden completar alguna de las siguientes oraciones:

(4) Yo soy…

(5) Yo estoy…

En euskera, (4) puede completarse con distintas categorías: sustantivos que indican las profesión de una persona, nombres propios, o adjetivos que indican características físicas o de personalidad. Por el contrario, (5) puede completarse con adverbios o complementos verbales de lugar. ¿Cómo organizamos la base de datos para recuperar todas las categorías que pueden completar (4) o (5)?

Para solucionar los dos casos de uso expuestos, necesitamos una base de datos que nos permita expresar múltiples relaciones entre palabras de forma intuitiva. Hemos barajado dos opciones para representar estas relaciones: el uso de bases de datos de grafos o el uso de ontologías. En concreto, hemos considerado Neo4J como base de datos y una ontología OWL, puesto que son las soluciones que exploramos en la asignatura Representación del Conocimiento.

Enumeramos a continuación algunas diferencias entre estos dos tipos de soluciones, tomadas de [42].

- La razón de ser de una ontología es especificar y compartir conocimiento. Una ontología funciona como una descripción de un área de conocimiento, y sus usos son muy variados: comunicación de campos de conocimiento, interoperatividad entre distintos sistemas, o búsqueda, por mencionar algunos. Por el contrario, el uso de una base de datos es muy concreto: estructurar un conjunto de datos de forma que el almacenamiento y la recuperación de estos sea lo más eficiente posible.
- Los sistemas de ontología utilizan *triplestores* o almacenes de RDF, bases de datos especialmente diseñadas para ontologías que se especializan en consultas semánticas. Aunque son muy flexibles en cuanto a las búsquedas que se pueden efectuar sobre ellas, su rendimiento es inferior al de las bases de datos. En contraste, las bases de datos se diseñan con la velocidad y la escalabilidad en mente desde el primer momento.

En resumen, una ontología es una instrumento de gran flexibilidad, equipada con herramientas muy potentes para modelar y razonar sobre campos de conocimiento de forma general y exhaustiva. Pero esta flexibilidad tiene un coste, la eficiencia. Nuestra aplicación no aspira a modelar el lenguaje de forma completa, ni deseamos llevar a cabo sobre el modelo razonamientos completos. Pero si la aplicación llegara a desplegarse, por ejemplo, como un servicio RESTful para un *frontend* web, la eficiencia

de las consultas sería un aspecto crucial; por ello, hemos optado por utilizar Neo4J como base de datos.

5.1.2 Introducción a Neo4J

Neo4J es una base de datos de grafos que utiliza grafos etiquetados (*property graphs*), diseñada para optimizar el almacenamiento y el recorrido de estos [43].

Un grafo es una estructura compuesta por nodos y aristas; las aristas conectan los nodos dos a dos. La direccionalidad de la relación puede representarse (grafos dirigidos) o no; en Neo4J, las relaciones siempre son dirigidas[16].

En Neo4J, los nodos y las aristas se definen mediante un nombre (etiqueta o *label*) y un conjunto de propiedades. Los nodos pueden tener tantos *labels* como se desee (incluso ninguno), pero las aristas deben tener una única etiqueta. Las propiedades son siempre opcionales y consisten en parejas clave-valor, donde los valores pueden ser de tipos heterogéneos (números, *strings*, booleanos, listas...).

Los nodos representan las entidades del modelo, y las etiquetas permiten clasificar estas entidades. Las aristas representan relaciones entre las entidades.

Para representar nodos y relaciones hemos seguido la convenciones recomendadas en el manual de Neo4J :

- Etiquetas de nodos: *camel case* con mayúscula inicial: `Category, Noun, Verb…`
- Etiquetas de aristas: mayúsculas con barras bajas: `IDENTITY, PROPERTY_ASCRIPTION...`
- Propiedades: camel case con minúscula inicial: `root, defaultNumber…`

Para insertar datos y consultar la base de datos se utiliza el lenguaje Cypher [44], un lenguaje declarativo similar a SQL.

Para conectar el programa con la base de datos hemos utilizado la librería Hasbolt [45], que utiliza el protocolo Bolt para comunicarse con el servidor de Neo4J. Es una librería no oficial de código libre; no es habitual encontrar soluciones oficiales para un lenguaje minoritario como Haskell.

5.2 Representación de las relaciones

Nuestro objetivo es organizar la base de datos de tal forma que podamos recuperar todos los lexemas necesarios para la generación de un tipo de ítem con el menor número de consultas a la base de datos, idealmente solo una.

16 Pero la direccionalidad puede obviarse en las búsquedas.

5.2.1 Todos los lexemas que completan una estructura

En los ejemplos (4) y (5) hemos presentado estructuras para las cuales necesitamos recuperar lexemas heterogéneos tanto en cuanto a categoría gramatical como en cuanto a contenido semántico. Reiterando un ejemplo, la oración *«Yo soy...»* puede completarse con nombres propios, adjetivos de personalidad, adjetivos físicos, profesiones, etcétera.

La cuestión es cómo organizar todos estos elementos en la base de datos para poder recuperarlos con una única *query*, al tiempo que mantenemos su independencia para poder utilizarlos también en otros casos; por ejemplo, también querremos recuperar únicamente: (a) adjetivos, para usarlos en sintagmas nominales (*«la chica alegre»*), y (b) nombres propios, que pueden funcionar como sujetos de oraciones (*«Ane está en el bar»)*.

El *Generalized Upper Model* (GUM, [27], [28]) ha sido de gran utilidad para resolver este problema. GUM es una ontología cuyo objetivo es proporcionar un modelo semántico para las expresiones del lenguaje natural. Es un descendiente del *Penman Upper Model* [46], una ontología desarrollada para un influyente generador de lenguaje natural en inglés, el Penman [47], [48].

Una de las categorías del GUM es *BeingAndHaving* (existencia y posesión). Dentro de esta, nos interesa especialmente la subclase *Intensive*[17], que representa todos aquellas propiedades que pueden definir a un objeto. En la Figura 10 podemos observar jerarquía completa.

La clave es que todas las relaciones de tipo *Intensive* puede expresarse con el verbo «ser»[18].

Si utilizamos estas categorías como nombres de las relaciones en la base de datos e implementamos también su organización jerárquica, será posible recuperar los nombres propios, adjetivos y oficios buscando aquellas entidades relacionadas con `Person` mediante la relación `INTENSIVE`.

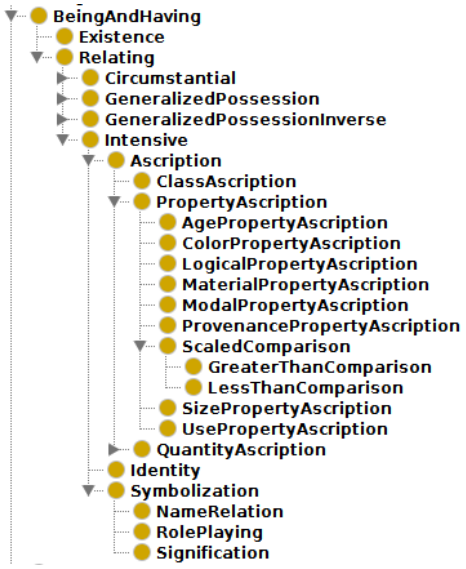

Figura 10: La relación Intensive *en GUM*

El problema es que no existe ninguna forma de expresar una jerarquía de relaciones en Neo4J. Lo que sí nos permite la base de datos es asociar el mismo par de nodos con más de una arista. Por ello, la solución que

17 Intensión. Ling. Conjunto de rasgos semánticos de una unidad léxica. (DLE)

18 El GUM está basado en el inglés. Algunas de estas relaciones no son equivalentes al castellano o al euskera; por ejemplo, la edad (*AgePropertyAscription*) se expresa en euskera con el verbo «eduki» 'tener'.

hemos encontrado para representar la jerarquía de GUM ha sido crear tantas aristas como relaciones nos encontramos en el camino entre `INTENSIVE` y la relación final que queremos representar.

Así, para recuperar todos los lexemas que funcionarían en «Yo soy ...» podemos ejecutar esta consulta:

```
match (:Person)-[:INTENSIVE]-(n) return n
```

Código 17: Consulta Cypher para recuperar todos los nodos relación con Person mediante la relación INTENSIVE

Y para consultar solo los nombres propios, podemos ejecutar esta otra:

```
match (:Person)-[:NAME_RELATION]-(n) return n
```

Código 18: Consulta Cypher para recuperar todos los nombres propios de persona

5.2.2 Lexemas interdependientes

Para la representación de los argumentos verbales, la jerarquía de GUM no nos ha servido de ayuda. El GUM representa los tipos de acciones según sus propiedades semánticas más generales, como se puede ver en la Figura 11; por ejemplo, si la acción tiene un argumento afectado («carta» en «enviar una carta» o «jabalí» en «cazar un jabalí») o no lo tiene («correr», «florecer»), si se trata de una acción de comunicativa, etc.

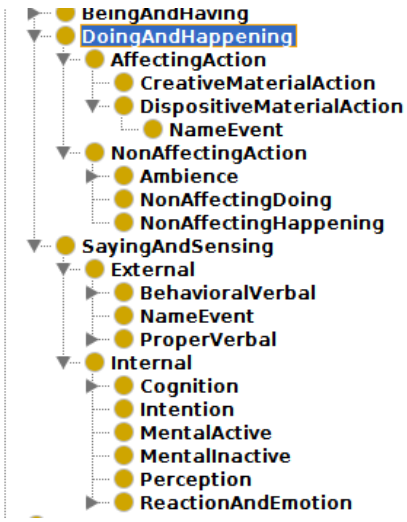

Figura 11: La relación DoingAndHappening *en GUM*

Este tipo de clasificación más general sí nos interesa porque, en los ejercicios, desearemos obtener verbos de un solo tipo[19]. Para reflejarla en la base de datos, simplemente añadiremos una etiqueta a cada verbo que indique su categoría: `Nor`, `NorNork`, etc.

Pero, adicionalmente, como hemos visto en los ejemplos (1), (2) y (3), lo que necesitamos es asociar los verbos a objetos compatibles: el verbo «comer» necesita objetos comestibles, y el verbo «estudiar», objetos estudiables.

No hay una solución automática para este problema[20]: debemos clasificar el vocabulario manualmente en categorías relevantes para el programa.

19 Clases de verbos en Figura 5, pág. 14.

20 Una solución automática sería utilizar técnicas de aprendizaje automático. El inconveniente de esta solución es que, como hemos comentado en la introducción, no nos permite controlar el nivel de dificultad del vocabulario.

La organización actual se refleja en la Figura 12. Un verbo (verde) tiene artistas salientes para los argumentos que toma; la etiqueta de la arista describe el caso que lleva ese argumento: NOR, NORK, NORI, etc. La arista no conecta directamente con un lexema, sino con una categoría semántica (naranja). Por ejemplo, el verbo «erosi» 'comprar', está asociado a múltiples categorías: ropa, comida, instrumentos musicales, publicaciones... Sin embargo, el verbo «irakurri» 'leer', está asociado solo a la categoría publicaciones.

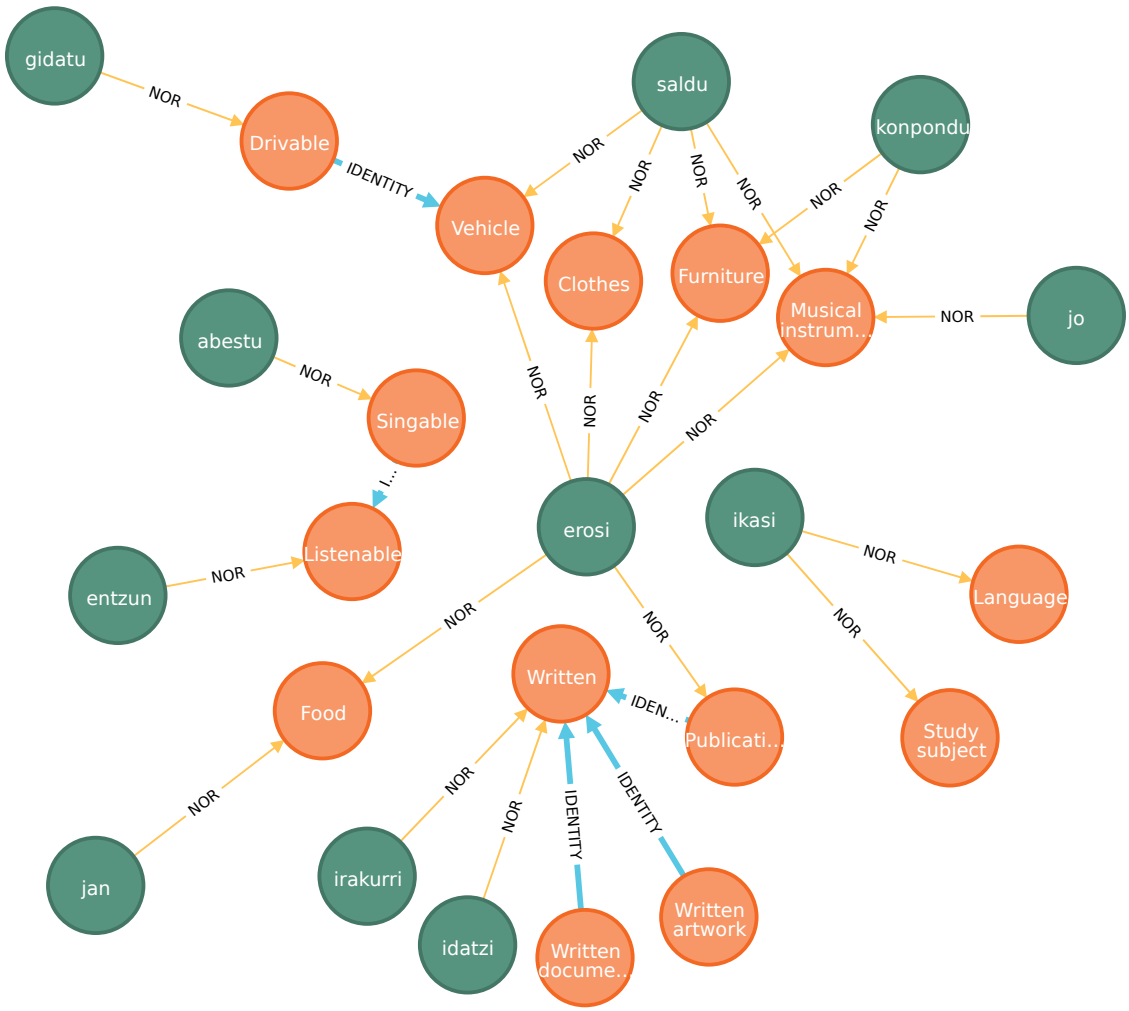

Figura 12: Verbos (verde) y categorías semánticas (naranja) a las que están asociados

Las categorías semánticas se organizan jerárquicamente mediante la relación `IDENTITY`. Veamos algunos ejemplos:

- la categoría `Vehicle` tiene la subcategoría `Drivable`. En el mundo real, todos los vehículos se pueden conducir, pero lo cierto es que oraciones como «Ha conducido una bicicleta» o «He conducido ese helicóptero» son extrañas. Por lo tanto, el verbo «gidatu» 'conducir' solo está asociado a `Drivable`, mientras que los verbos «erosi» 'comprar' y «saldu» 'vender' están asociados a `Vehicle`.

- `Singable` es una subcategorías de `Listenable`. Así, el verbo «entzun» puede asociarse a todos los objetos que se pueden escuchar (incluidos aquellos que se pueden cantar), mientras que «abestu» 'cantar' solo se asocia a estos últimos.
- Hay muchos tipos de objetos que se pueden leer y escribir (`Writable`). Pero de estos, únicamente los que pertenecen a la categoría `Publication` se pueden comprar («saldu»).

La Figura 12 hace patente que algunos verbos naturalmente se pueden combinar con muchas categorías (como «erosi» 'comprar' o «saldu» 'vender'), mientras que otros tienen campos semánticos más limitados.

Con Cypher podemos recuperar toda la jerarquía indicando que queremos acceder a los nodos de tipo `Category` relacionados entre sí mediante 0 o más relaciones de tipo `IDENTITY`:

```
match (v:Verb)-[:NOR]-(c1:Category)-[:IDENTITY*0..]-(c2:Category)
return v, c1, c2
```

Código 19: Consulta Cypher para recuperar las categorías semánticas asociadas a verbos mediante la relación NOR

Otra ventaja de esta organización, no explotada en este momento, es que permitiría generar ejercicios de un único campo semántico.

La mayor desventaja de este sistema, sin embargo, es que es muy laborioso introducir los datos. Además, si en el futuro resultara que esta organización no es la idónea, habría que reorganizar prácticamente todo. Hasta ahora yo he creado la base de datos de forma automática, utilizando *scripts* de Haskell que generan *scripts* Cypher a partir de listas de palabras. Así, si añado o elimino palabras a alguna lista, solo tengo que regenerar los *scripts* de Cypher y repoblar la base de datos. Sin embargo, sería deseable implementar un sistema que permita automatizar este proceso todavía más.

5.2.3 Tripletes

Aunque en el desarrollo actual todavía no se ha dado el caso, en el futuro es muy probable que sea necesario recuperar de la base de datos tuplas de lexemas asociados que tengan una aridad mayor que 2. Por ejemplo, podríamos tener tríos de verbo-objeto-lugar, para representar acciones que se realizan prototípicamente en un lugar:

1. comprar-libro-tienda de libros,
2. comprar-comida-supermercado,
3. sacar-libro-biblioteca,
4. comer-comida-restaurante, etc.

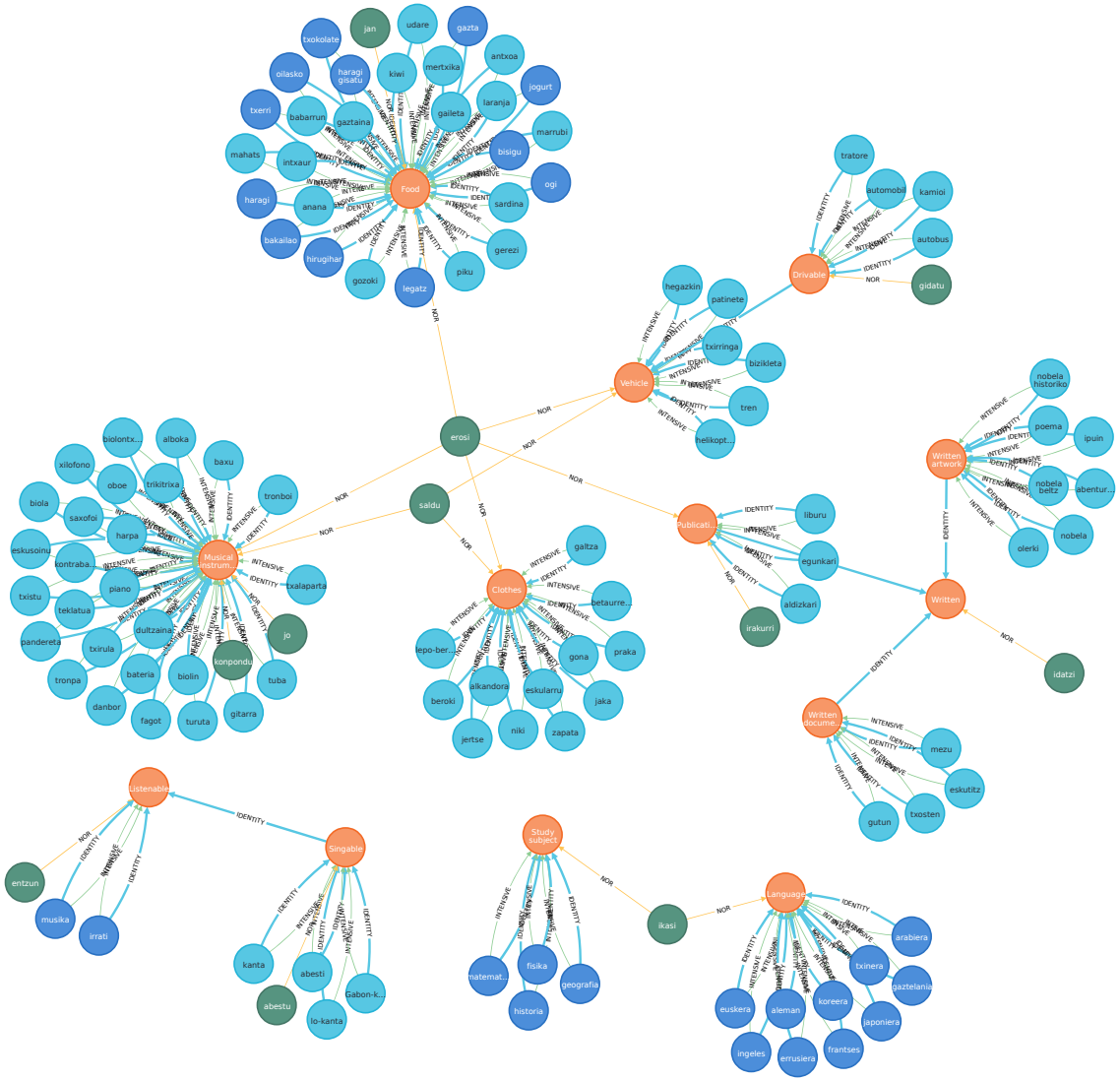

Figura 13: Verbos (verde) asociados a sustantivos (azul: contable; turquesa: no contable) a través de categorías (naranja)

Estas asociaciones solo tienen sentido en grupos de tres. La estrategia descrita en el apartado anterior no funciona en estos casos, ya que todas relaciones OBJETO y LUGAR asociadas a un mismo verbo no son todas compatibles entre sí; por ejemplo, «comida» no es compatible con «tienda de libros».

De esto se desprende que un único nodo del verbo «comprar» no sería suficiente para representar tanto la relación (1) como la (2). Habría que crear tantos nodos del mismo verbo como asociaciones distintas queramos representar, es decir, utilizar tripletes.

5.3 Conversión de nodos a Lexis y gestión de errores

El módulo `Neo4J.Conversion` se encarga de transformar nodos en `Lexis`. El punto de entrada al sistema es la función `nodeToLexis`, que accede a las etiquetas (*labels*) del nodo para comprobar a qué clase de palabra pertenece. Siguiendo el mismo patrón

que en otras ocasiones, tenemos funciones específicas para la creación de las distintas clases de `Lexis` a partir de las distintas propiedades de los nodos (Código 20).

```
nodeToLexis :: Node -> Either EuskeraAppError Lexis
nodeToLexis node
       | "ANTHROPONYM" `elem` wordclass = createAnthroponym node
       | "COMMON" `elem` wordclass = createCommonNoun node
       | "MASS" `elem` wordclass = createMassNoun node
       | "ADJECTIVE" `elem` wordclass = createQualitativeAdjective node
       | "NOR_NORK" `elem` wordclass = createNorNorkVerb node
       | otherwise = Left Unimplemented
       where wordclass = T.unpack <$> labels node
```

Código 20: Función nodeToLexis, interfaz entre la base de datos y el programa

Es posible que un nodo esté corrupto, y que propiedades necesarias para transformarlo en un tipo del programa no estén presentes. Esto haría imposible la creación de una instancia de `Lexis`. No deseamos que estos elementos corruptos afecten de ningún modo a la generación de oraciones y ejercicios, pero sí nos gustaría poder detectarlos.

Cuando construimos un programa sobre funciones matemáticas, el tipo que resulta de evaluar una expresión siempre debe ser el mismo; no podemos devolver un valor de tipo `a` en unas ocasiones, y un valor de tipo `b` en otras (por ejemplo, un valor nulo). Pero esta situación es tremendamente común, y es a la que nos enfrentamos en la conversión de nodos a entidades del programa. La solución, en Haskell y otros lenguajes funcionales, son los tipos opcionales.

Un tipo opcional es un tipo que representa la presencia o ausencia de un valor. En Haskell, el tipo más básico de esta clase es `Maybe`, y esta es su definición:

```
data Maybe a = Nothing
             | Just a
```

Código 21: El tipo opcional Maybe

El constructor `Nothing` no acepta parámetros, y representa el caso del valor ausente. `Just` acepta un parámetro, un valor de tipo `a`. `Maybe` es un tipo paramétrico, y `a` es su variable de tipo. Esto significa que el tipo `Maybe` es un tipo que funciona como un envoltura para otros tipos.

Otro tipo opcional es `Either`, muy similar a `Maybe`, pero que adicionalmente permite retornar un valor en caso de error. Esta es su definición:

```
data Either a b = Left a
                | Right b
```

Código 22: El tipo opcional Either

Como vemos, `Either` tiene dos parámetros de tipo, uno para cada constructor. En la práctica, así conseguimos retornar dos tipos distintos utilizando un único tipo. En la

signatura de la función `nodeToLexis` (Código 20) observamos que devuelve el tipo `Either EuskeraAppError Lexis`. Por convención, el constructor `Left` se utiliza para el caso de error, y el constructor `Right` para el caso de éxito. Cuando resulta imposible convertir un nodo a `Lexis`, devolvemos un error de conversión que nos indica el identificador del nodo corrupto (Código 23).

```
conversionError :: Node -> Either EuskeraError Lexis
conversionError node = Left $ ConversionError
                                "Node id: " ++ show (nodeIdentity node)
```

Código 23: Error de conversión de la base de datos

Así, el resultado de una petición a la base de datos es una lista de tipo `Either EuskeraError Lexis` (o una lista de tuplas de este tipo). Las funciones `rights` y `lefts` que implementa el paquete `Data.Either`, nos permiten o bien extraer de esta lista únicamente los nodos que se han procesado correctamente, o bien recuperar únicamente los casos de error.

6 Generación de ejercicios

Ahora que hemos presentado todos los componentes principales del programa, podemos exponer el proceso de generación de ejercicios.

Un ejercicio contiene en una colección de ítems del mismo tipo. Cada ítem consiste en (a) una oración correcta, y (b) la misma oración con algunos constituyentes sustituidos por huecos. Para generar un nuevo ejercicio debemos llevar a cabo dos acciones principales:

1. Crear las instrucciones para generar un nueva `ProtoSentence`.
2. Indicar la etiqueta que se quiere convertir en *gap*.

El paso 1, a su vez, requiere:

1. Definir la consulta a la base de datos para recuperar el vocabulario necesario.
2. Establecer los rasgos gramaticales configurables del ejercicio.
3. Generar el resto de rasgos gramaticales de forma aleatoria

Cada ejercicio se crea dentro de una función que lleva a cabo estos pasos y toma como argumentos: (a) el número de oraciones a generar, (b) una conexión a la base de datos[21] y (c) los rasgos gramaticales configurables de este ejercicio. Por ejemplo, si el mismo ejercicio sirve para practicar verbos en presente y en pasado, `Tense` y `Aspect` serán dos argumentos que reciba la función.

6.1 Consulta del vocabulario

El primer paso de la generación de un ejercicio es recuperar el vocabulario necesario para ello.

Las consultas están definidas en el módulo `Neo4J.Queries`. Cuando ejecutamos la consulta en la base de datos, esta nos devuelve una lista de `Node` (tipo de la librería Hasbolt que representa un nodo de la base de datos). Procesamos esta lista con las funciones del módulo `Neo4J.Conversion`, para obtener una lista de `Lexis` o de tuplas de `Lexis`.

Este proceso es agnóstico al número de oraciones que será necesario generar. Por si este fuera mayor que el número de `Lexis` disponibles, a partir de la lista original creamos un lista infinita utilizando la función `cycle` de `Data.List`.

21 Podríamos utilizar una mónada (ver la sección 6.2.1) para no tener que pasar este argumento manualmente. No se ha implementado por falta de tiempo.

6.2 Aleatoriedad

Hemos introducido aleatoriedad en dos lugares del proceso generativo para evitar que el programa genere siempre las mismas oraciones:

- Las consultas a la base de datos devuelven los resultados ordenados de forma aleatoria; así aseguramos la novedad del vocabulario de cada ítem. Para conseguir esta funcionalidad ha sido necesario utilizar la librería APOC [49] de Neo4J.
- En cada ítem, algunos rasgos gramaticales son definitorios y otros pueden aleatorizarse. Por ejemplo, si queremos practicar el verbo *izan* 'ser' en presente, los rasgos de tiempo y aspecto son definitorios del ejercicio. Pero para cada ítem, los rasgos de persona y número del sujeto se generan aleatoriamente, para garantizar la variedad del ejercicio.

6.2.1 ¿Funciones puras y aleatorización?

En un lenguaje funcional puro como Haskell, las funciones —como en matemáticas— siempre generan el mismo resultado al ser llamadas con los mismos argumentos. Es decir, ninguna función puede depender del estado del programa, solo de los argumentos que recibe. Esto supone un problema en muchas ocasiones, entre ellas aquellas que necesitan de aleatoriedad.

Para imitar secuencias de números aleatorios se utilizan generadores de números pseudoaleatorios, algoritmos que parten de una semilla (el valor inicial que recibe el algoritmo) y generan series de números. Estos algoritmos son deterministas: a partir de la misma semilla, siempre obtendremos la misma serie de números; la aleatoriedad radica en que, dado un número de la serie, no es posible predecir cuál será el siguiente. Los generadores de números pseudoaleatorios mantienen un estado, cuya función es determinar el siguiente número en la serie; sin embargo, en los lenguajes de programación no puros a los que estamos acostumbrados, este estado, que es parte del programa, es invisible para el programador.

La única solución que puede dar la programación funcional pura a este problema es propagar el estado manualmente a través del programa. La forma más intuitiva para conseguir esto es convertir el estado tanto en un argumento de la función, como en parte del valor que resulta de evaluar la función.

Por ejemplo, la hipotética función `nextNumber` recibe un generador como argumento. Al evaluarla, obtenemos una tupla que contiene el siguiente número aleatorio, y un nuevo generador.

```
nextNumber :: randomNumberGenerator -> (Int, randomNumberGenerator)
```

Código 24: Signatura de una hipotética función de generación de números aleatorios

Si usamos `nextNumber` en nuestro código, todas las funciones que necesitan de aleatoriedad directa o indirectamente deben recibir y devolver el generador aleatorio. El código rápidamente se convierte en un galimatías, plagado de argumentos aparentemente inútiles.

Afortunadamente, existe otra forma de propagar este estado: el uso de mónadas [50]–[52]. Las mónadas son una estructura tomada de la Teoría de Categorías [53] que se ha convertido en una herramienta básica del repertorio del programador funcional; Haskell cuenta con una clase de tipo para representarlas.

Podemos definir una mónada como un triplete (`M, unit, bind`), donde `M` es un constructor, y `unit` y `bind`[22] son funciones polimórficas con la siguiente signatura:

```
unit :: a -> M a
bind :: M a -> (a -> M b) -> M b
```

Código 25: Signatura de las funciones necesarias para definir una mónada

La función `unit` sirve para introducir cualquier valor en el contexto de la mónada (las mónadas son tipos envolventes, como `Maybe`); `bind` toma una mónada de tipo `a`, y una función de `a` a mónada de `b`, y retorna una mónada de tipo `b`.

¿Por qué esta última función recibe `a` y no `M a`? Aquí, precisamente, reside la magia de las mónadas que nos permite propagar el estado implícitamente. Veamos un ejemplo sencillo de una mónada[23]:

```
type Debuggable a = (a, String)
```

El tipo `Debuggable` es una tupla que contiene el valor que nos interesa, y adicionalmente un `String`. Los valores adicionales (`String`) son el contexto, y queremos añadir información a este contexto a medida que se ejecuta el programa. Se suele afirmar que, si consideramos que el tipo a representa un valor, la mónada `M a` (`Debuggable a`, o cualquier otra) representa una computación [52].

El objetivo de `unit` es convertir un valor en una computación, o introducir el valor en el contexto de la mónada. Por lo tanto:

```
unit x = (x, “”)
```

22 En Haskell unit es return, y bind el operador de tipo infijo (>>=).
23 Este ejemplo procede de [54], la mejor introducción a las mónadas que he encontrado (y he leído muchas).

El objetivo de `bind` es evaluar una computación y obtener un valor. Por lo tanto:

```
bind (Debuggable (x, s)) f =
      let (x', s') = f x in Debuggable (x', s++s')
```

¿Qué se consigue con `bind`? Escribir funciones que no necesitan recibir la información adicional, pero sí la devuelven; de esta forma, podemos concatenar operaciones monádicas, sin preocuparnos por propagar el estado. Además, en Haskell, las mónadas cuentan con una sintaxis especial, muy similar a la programación imperativa. Veremos un ejemplo de ello más adelante (Código 26).

El generador de ejercicios utiliza la mónada `State` (Estado). La mónada `State` es similar a `Debuggable`, pero con un añadido: permite utilizar el valor del estado en la computación que se lleva a acabo.

Hemos definido un tipo `SentenceState` para propagar el estado durante proceso de transformar un `ProtoSentence`; se trata de una tupla `(StdGen`[24]`, ProtoSentence)`, y todas las funciones que hemos declarado para modificar una `ProtoSentence` son monádicas (compatibles con `bind`): toman la `ProtoSentence` que se quiere modificar como argumento, pero retornan un `SentenceState`.

No todas las modificaciones que llevamos a cabo sobre un `ProtoSentence` necesitan aleatoriedad, pero todas las funciones deben ser monádicas, para poder concatenarlas con las que sí requieren aleatoriedad y propagar así el estado del generador de números aleatorios.

Las funciones que sí introducen aleatoriedad se encuentran en el módulo `Exercises.RandomizeSentence` y comienzan por la palabra clave *random* (`randomPersonNumber`, `randomDeterminer`) o por el verbo *randomize* (`randomizeSentenceType`). Puesto que las mónadas son tipos paramétricos, el valor que devuelven puede ser de cualquier tipo, no tiene por qué ser un `ProtoSentence`. Las funciones que simplemente actualizan el estado de la oración viven en el módulo `Exercises.UpdateSentence` y comienzan por el verbo *update* (`updateVerb`, `updateNorWithNP`) o *initialize* (`initializeCopula`, `initializeNorNork`). Esta distinción en la nomenclatura nos ayuda a entender qué manipulaciones se llevan a cabo al crear un ítem.

En el Código 26 vemos un ejemplo. La función `copulaItem` crea un ítem con una oración copulativa[25]. Vemos que recibe los rasgos de tiempo y aspecto y una pareja de `Lexis`. El código es muy claro en cuánto a qué se hace en cada paso.

24 StdGen es el tipo que representa al generador de números pseudoaleatorios de la librería Random de Haskell.

25 Una oración copulativa utiliza el verbo «ser» como verbo principal, y tiene la particularidad de que el sujeto y el atributo concuerdan en género y número.

```haskell
copulaItem ::
      Tense ->
      Aspect ->
      (Lexis, Lexis) ->
      State SentenceState ProtoSentence
copulaItem t a nouns =
      do
       initializeCopula
       randomizeSentenceType
       updateTenseAndAspect t a
       nf <- randomPersonNumber
       updatePronominalSubjectAndAttribute nf nouns
```

Código 26: Función copulaItem para generar ítems con el verbo "izan" ('ser')

En la función `CopulaItemsPresent`, que genera un ejercicio de ítems copulativos en tiempo verbal presente, encontramos esta línea:

```haskell
let (sents, newState) = runState (mapM (copulaItem Present Punctual)
namePairs) s
```

Código 27: Generación de oraciones dentro de una mónada

Donde `namePairs` es una lista de `(Lexis,Lexis)` obtenida de la base de datos, y `s` un estado `State SentenceState` inicial. La función `mapM` es la versión monádica de `map`[26]. Con una única línea de código generamos tantas oraciones como elementos contiene la lista. En ningún lugar del Código 26 o del Código 27 se encuentra una mención al estado, excepto al inicio absoluto del proceso y en el resultado obtenido (`s` y `newState` son estados que contienen un generador).

Aunque requiere más trabajo que simplemente llamar a una función que genera números aleatorios, los defensores de este paradigma de programación afirman que separar claramente las computaciones puras de las que producen efectos ofrece claras ventajas, sobre todo porque ayudan a razonar sobre los programas.

> Haskell's method of isolating side effects into I/O actions provides a clear boundary. You can always know which parts of the system may alter state and which won't. You can always be sure that the pure parts of your program aren't having unanticipated results. This helps you to think about the program. It also helps the compiler to think about it. Recent versions of ghc, for instance, can provide a level of automatic parallelism for the pure parts of your code — something of a holy grail for computing [55]

Comprender bien el funcionamiento de las mónadas requiere bastante esfuerzo. Para mí ha sido muy interesante aprender a definirlas y utilizarlas, y creo que todavía me queda mucho por comprender. Una vez definidas las funciones monádicas, programar con ellas es una experiencia muy ágil y agradable. Como punto negativo, creo que hay que mencionar que convertir un código no monádico en monádico puede requerir una refactorización de dimensiones considerables. Por lo tanto, habría que intentar detectar las situaciones que requieren de esta estructura lo antes posible en el proceso

26 Map aplica una función a todos los elementos de una lista y devuelve el resultado de esta aplicación.

de desarrollo. Este es un problema que parece común en que la programación funcional pura:

> But, sometimes, a seemingly small change may require a program in a pure language to be extensively restructured, when judicious use of an impure feature may obtain the same efect by altering a mere handful of lines. [52]

6.3 Inserción de huecos

Como hemos visto en el Capítulo 4, todos los constituyentes contienen una lista de etiquetas. Para generar ejercicios, simplemente debemos elegir las etiquetas que nos interesa convertir en *gaps*; por ejemplo, «mainVerb» o «genitive» en el caso de los ejercicios disponibles actualmente en el programa. La función `addSimpleGaps` hace un recorrido por árbol de constituyentes, generando un nuevo árbol que contiene constituyentes de tipo `FillGap` en lugar de los originales en los lugares donde queremos insertar un hueco.

Para que este sistema funcione, es necesario incluir en la definición original de Sentence un nuevo constituyente: `FillGap` (ver Código 28).

```
data Constituent = S [Tag] SF [Constituent]
                 |  VP [Tag] VF [Constituent]
                 | NP [Tag] NF [Constituent]
                 | L [Tag] Lexis
                 | FillGap
```

Código 28: Definición final del tipo Constituent

Este método de insertar huecos proporciona tres grandes ventajas. En primer lugar, la misma representación sintáctica nos permite generar ejercicios distintos, simplemente seleccionando etiquetas distintas. En segundo lugar, el sistema se adapta de forma natural a casos en que los constituyentes que hay que eliminar para generar la tarea son discontinuos. Por ejemplo, en la oración *Nik **ez ditut** sagarrak **jan*** (‘Yo no me he comido las manzanas’), hay que eliminar tres constituyentes: la negación (*ez*), el verbo auxiliar (*ditut*) y el verbo principal (*jan*). El ejercicio se genera correctamente si todos estos elementos contienen una etiqueta igual. Por último, este sistema permite insertar huecos en cualquier nivel del árbol sintáctico: en constituyentes grandes o en palabras individuales.

7 Conclusiones

7.1 Grado de consecución de los objetivos

El objetivo de este trabajo era sentar las bases de un generador automático de ejercicios de euskera para los niveles de iniciación, A1 y A2 del Marco Común Europeo de Referencia [2]. Las características principales que el sistema debía mostrar eran:

1. Generar oraciones coherentes y sin errores gramaticales
2. A partir de estas oraciones, introducir huecos o *gaps* en las posiciones adecuadas para generar un ejercicio.
3. Permitir al usuario escoger entre ejercicios de distintos fenómenos sintácticos o morfológicos.
4. Permitir al usuario escoger un área semántica de los ejercicios.

```
----------------------
1  Ni altua _______.
Ni altua naiz.

  Oso ondo! :D
----------------------


----------------------
2  Gu idazleak _______.
Gu idazleak gaude.

  La respuesta correcta era: Gu idazleak gara.
----------------------


----------------------
3  Hura _______ _______ zoroa.
```

Figura 14: Ejemplo de la aplicación en acción

Hemos probado el sistema con tres hablantes nativos de euskera[27] para comprobar si se había cumplido el objetivo 1, y podemos afirmar que así es. Cada individuo ha probado todos los ejercicios que expone ahora mismo la interfaz (Figura 3): oraciones copulativas en presente y pasado, oraciones transitivas en presente perfecto y en futuro, y declinación del caso genitivo. Ninguno de los *testers* ha hallado oraciones incorrectas o carentes de sentido, aunque se les indicó que prestaran especial atención a esos aspectos al leer los ítems.

Sí comentaron que las oraciones no sonaban «del todo naturales». Sin embargo, al mostrarles ejercicios de un libro de texto de euskera y del curso en línea *Ikasten* [3], afirmaron que todas ellas les parecían igual de artificiales.

Lo cierto es que los ejercicios de nivel inicial en cualquier idioma suenan acartonados a los hablantes nativos. Esto puede deberse a que son oraciones totalmente

27 Naturales de Guipúzcoa.

descontextualizadas, o a que las estructuras sintácticas que presentan son extremadamente sencillas; también a que utilizan oraciones completas, cuando en la lengua oral normalmente se eliden ciertos constituyentes (en castellano y en euskera, por ejemplo, el sujeto suele elidirse porque la declinación verbal ya da información sobre él). Adicionalmente, los ejercicios de euskera utilizan la variedad estándar de la lengua, el *batua*, mientras que los hablantes nativos se comunican en las variedades dialectales su región.

El proceso de conseguir que las oraciones fuesen coherentes no ha sido fácil y, a medida que crezca el sistema, probablemente aumente la complejidad del mapa semántico. Aunque antes de comenzar el proyecto parecía que la mayor fuente dificultades en este sentido sería la organización del vocabulario, lo cierto es que lo más difícil ha sido predecir qué rasgos semánticos serán incompatibles con qué rasgos sintácticos, y adaptar el sistema a ello.

Por ejemplo, debemos introducir reglas especiales para que los sustantivos no contables (como «mantequilla» o «aceite») nunca aparezcan en plural, lo cual implica hacer comprobaciones durante el proceso de asignación aleatoria del rasgo gramatical `Number`.

Otro caso peliagudo, cuya implementación no hemos abordado todavía, es la de la compatibilidad entre tipos de verbos y tiempos verbales. Los verbos se pueden clasificar según el tipo de evento que designan; en Lingüística, esta característica de los verbos se denomina «aspecto léxico». Por ejemplo, «amar» o «estar enfermo» son estados, y no suelen aparecer en perífrasis progresivas como «estoy amando a esa persona» o «estoy estando enfermo». Para lidiar con estas incompatibilidades habrá que añadir una propiedad a los nodos verbales que indique su aspecto léxico.

El objetivo número dos, la generación de huecos, se ha conseguido con facilidad mediante el etiquetado de constituyentes. En este ámbito cabe mencionar que lo que ha supuesto el mayor desafío ha sido un punto que ni siquiera se había considerado al inicio: la necesidad de dar pistas sobre qué elemento léxico hay que completar —qué verbo hay que conjugar, qué sustantivo hay que declinar—.

El tercer objetivo, escoger ejercicios de distintas estructuras gramaticales, también se ha completado con éxito. En este momento se pueden hacer ejercicios de tres tipos diferentes:

- Practicar la conjugación del verbo «izan» en oraciones copulativas, tanto en presente como en pasado.
- Practicar la conjugación de distintos verbos de tipo nor-nork —aquellos que tienen agente y un objeto afectado— en pretérito perfecto y en futuro. Fácilmente se pueden añadir otras conjugaciones verbales.
- Practicar la declinación nominal en caso genitivo, para nombres propios y

pronombres.

Con la arquitectura actual, añadir ejercicios sobre el vocabulario existente es relativamente sencillo. De hecho, implementar la actividad de declinación nominal no llevó más de un par de horas. Sin embargo, añadir vocabulario nuevo es costoso.

El cuarto objetivo, por el contrario, solo se ha cumplido parcialmente. La organización del vocabulario en la base de datos sí permitiría escoger el campo semántico sobre el que generar los ejercicios; simplemente habría que restringir las categorías con las que se quiere trabajar. Sin embargo, el vocabulario actual es bastante limitado (tenemos alrededor de 300 nodos, incluyendo nodos de vocabulario y de categorías), y no permite generar un ejercicio temático con suficiente variación.

7.2 Líneas de trabajo futuro

El objetivo del proyecto era reflexionar sobre cómo podría ser una arquitectura apropiada para una aplicación de generación de ejercicios de euskera en un lenguaje funcional, e implementarla parcialmente. Aunque se ha conseguido implementar una prueba de concepto con funcionalidad limitada, se podría afirmar que la mayor parte del trabajo está por hacer.

A pesar de esto, mi prioridad principal en este momento es exponer la funcionalidad existente a partir de una API REST, y crear una página web para hacer pública la aplicación.

Una vez conseguido esto, hay dos vertientes en las que seguir trabajando. Por una parte, hay que ampliar la funcionalidad añadiendo vocabulario y ejercicios; todas las tareas planificadas como secundarias y no implementadas serían el lugar idóneo por donde empezar. Por otra, quedan muchos aspectos de la aplicación que cabe mejorar; los enumeramos aquí por orden de prioridad.

- Implementar tests. Uno de los objetivos mencionados en la introducción del trabajo era la implementación de tests. Dadas las restricciones temporales y la complejidad del proyecto, los tests pasaron a ocupar un segundo plano. Aunque se implementaron algunos durante el desarrollo, la mayoría se han eliminado del entregable final porque no eran exhaustivos. Sin embargo, sería imprudente seguir ampliando la aplicación sin desarrollar un set completo de tests.
- Mejorar y automatizar hasta donde sea posible la introducción de vocabulario en la base de datos.
- Extender el uso mónadas en otros contextos. Por ejemplo, en la gestión de la conexión a la base de datos.
- Estudiar la eficiencia del código actual.

7.3 Seguimiento de la planificación

La planificación inicial mostrada en la sección 1.4 *Planificación del Trabajo* es la planificación final, adaptada durante el periodo de desarrollo del trabajo. Mostramos un diagrama de Gantt de la planificación inicial en la Figura 15.

La planificación inicial sufrió algunos cambios debido, sobre todo, a que no se había asignado tiempo suficiente para cada tarea. Esto era un riesgo que habíamos asumido desde el principio, ya que pensábamos que la planificación inicial era algo optimista.

Sin embargo, la organización del trabajo en ciclos resultó ser un buen aliado. Puesto que al finalizar cada ciclo se había planificado entregar una aplicación completamente funcional, se pudo eliminar un ciclo completo sin que el producto final quedase inacabado. Asimismo, tener las tareas clasificadas en principales y secundarias ayudó enormemente durante el proceso de desarrollo, porque me permitió concentrarme en todo momento en las funcionalidades esenciales.

En la Tabla 1 se muestra una comparativa de las principales tareas planificadas y las tareas realizadas finalmente.

Tabla 1: Comparación entre tareas planificadas y realizadas

Se planifican...	**Se realizan...**
Cuatro ciclos de desarrollo	Tres ciclos de desarrollo, más un ciclo de revisión y mejora del código
Tareas primarias y secundarias	Tareas primarias y solo algunas secundarias
Tests unitarios	--
Patrones verbos copulativos, nor-nork, y nor-nori-nork	Patrones verbos copulativos y nor-nork
Verbos de movimiento y caso locativo	Verbo «izan» y caso genitivo[28].

28 Se sustituyó esta funcionalidad porque en la aplicación de referencia, ArikIturri [6], los ejercicios con el caso genitivo son unos de los que no se pudieron generar con éxito.

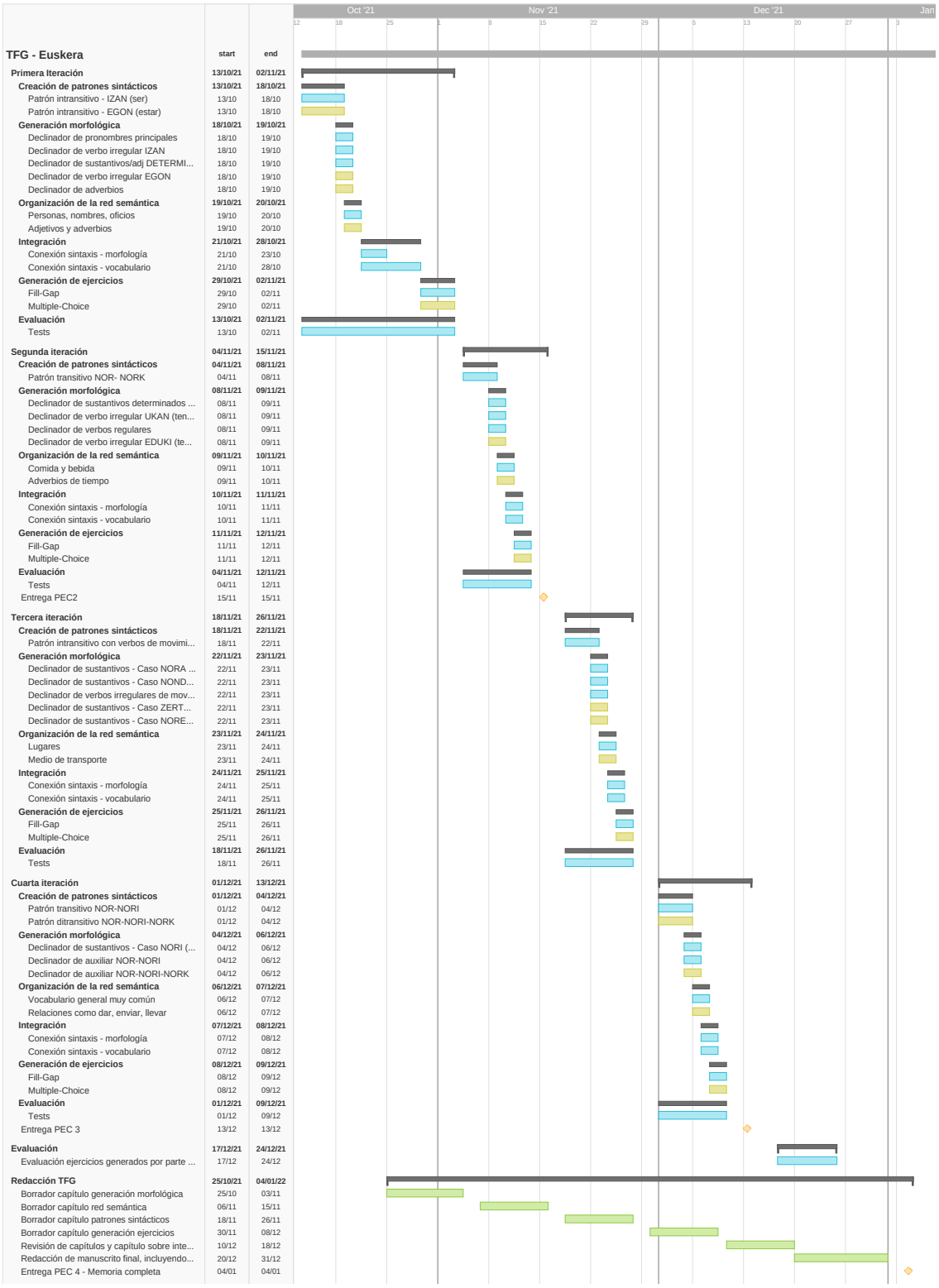

Figura 15: Diagrama de Gantt de la planificación inicial del proyecto

7.4 Seguimiento de la metodología

En la sección 1.3 se sentaron una líneas generales que se seguirían para el desarrollo de la aplicación, en concreto:

1. Usar Haskell como lenguaje de programación.
2. Modelar la estructura gramatical mediante patrones, más que mediante reglas.
3. Utilizar el GUM [27], [46] como punto de partida para el diseño de la base de datos.
4. Inspirarse en los sistemas existentes de generación de ejercicios para convertir oraciones en ítems con huecos.

Hemos seguido todas estas declaraciones de intenciones en mayor o en menor medida, excepto el punto 4:

1. La totalidad del programa se ha programado utilizando Haskell.
2. Las estructuras oracionales se han definido como patrones en lugar de utilizar reglas generativas. Esto queda patente, por ejemplo, en la definición de estructuras negativas e interrogativas. No utilizamos reglas transformacionales para pasar de una estructura afirmativa a una de otro tipo, si no que cada estructura tiene su propia definición. Creo que esta metodología ha sido un acierto que ha agilizado bastante el proceso de programación, y que hace que el programa sea más fácilmente extensible. Alterar patrones es tan fácil como cambiar el orden de una lista, mientras que alterar reglas transformacionales es mucho más complejo.
3. Nos hemos inspirado en la estructura jerárquica de entidades del GUM siempre que ha sido útil. En concreto, hemos trasladado algunas entidades del GUM a las relaciones entre lexemas y categorías del grafo que constituye nuestra base de datos de vocabulario.
4. Hemos usado una estrategia novedosa para generar los huecos de los ejercicios: etiquetar los constituyentes. Puesto que los sistemas de referencia se basaban en técnicas de aprendizaje automático, no era posible trasladar sus soluciones a nuestro programa.

7.5 Reflexión final y lecciones aprendidas

La realización de este trabajo ha sido tremendamente enriquecedora tanto a nivel personal como a nivel académico. A nivel personal, porque ha sido un proyecto motivado por mis intereses y necesidades. Por una parte, llevo dos años aprendiendo euskera, y la idea de crear estas aplicación surgió precisamente mientras estudiaba. Por otra parte, mis estudios previos son en Filología y Lingüística, pero nunca había abordado la formalización de los conceptos teóricos al nivel que exige un programa de generación del lenguaje natural. Mi base previa en gramática ha sido muy útil en dos aspectos principales:

- para organizar las funcionalidades del programa en módulos cohesionados e independientes, que a su vez me ayudaron a afrontar cada problema de forma aislada.
- para entender rápidamente a qué se debían los problemas que iban surgiendo en la generación de oraciones. Por ejemplo, cuando se generó la oración «He comprado algunas mantequillas»[29] entendí rápidamente que debía añadir la subclase de nombres no contables (`Mass`) al tipo `Noun`.

Paralelamente, me he dado cuenta de que el nivel de detalle en el que suelen trabajar los lingüistas se queda corto cuando nos enfrentamos a la problemática real de diseñar algoritmos de generación de oraciones, ya que las irregularidades son mucho más abundantes de lo que uno espera, sobre todo en el ámbito semántico. No he sido la primera en tener esta experiencia:

> Implementers often look at non–computational linguistics and find it "fuzzy", and lacking in detail. Implementation requires a completeness and explicitness, which a theoretician would find tedious. Much of theoretical linguistics has not yet reached a level of understanding at which it becomes implementable. [56]
>
> It appears that there is a natural tendency to write the grammar with excessive homogeneity, not allowing for possible exception cases. [57]

Pero el reto no ha sido únicamente la formalización de las reglas lingüísticas, sino también el lenguaje de programación escogido para hacerlo. La programación funcional en general, y en concreto la programación funcional pura que implementa Haskell, requiere un cambio completo de mentalidad. En ocasiones (en muchas ocasiones, a decir verdad) el ejercicio de programar ha sido enormemente incómodo: no dominaba la sintaxis, no dominaba las librerías básicas, no dominaba ni los patrones de programación básicos ni los más avanzados. He pasado muchas horas simplemente estudiando estructuras como las mónadas, para comprenderlas y utilizarlas eficazmente.

Sin embargo, también ha habido momentos en los que he podido valorar la capacidad de los lenguajes funcionales para expresar y hacer mucho con pocas líneas de código.

29 Traducida del euskera

He entendido de veras qué significa evaluar expresiones en lugar de ejecutar funciones. Me he maravillado al comprender las mónadas, los funtores y los aplicativos. El proceso de detección y solución de *bugs,* por ejemplo, ha sido muy fácil, y esto me ha sorprendido gratamente. Pensar en las funciones desde el punto de vista de los tipos y escribir sus signaturas antes de implementarlas se ha convertido en un hábito.

Aunque el desarrollo del programa ha sido un reto en todos los aspectos técnicos, creo que esta inmersión intensiva en el mundo de la programación funcional me ha convertido en una programadora un poco mejor, y ha conseguido ahondar más mi interés por aprender, comprender y conseguir dominar las técnicas y las bases teóricas de la programación funcional.

8 Glosario

ADT	*«Algebraic data type» o tipo de dato algebraico. En programación funcional, es un tipo formado por la composición de otros tipos.*
Argumento (verbal)	*Los argumentos de un verbo son aquellos constituyentes que el verbo necesita, por oposición a aquellos que son opcionales. El concepto es similar al argumento de una función.*
Caso gramatical	*El caso gramatical designa la función sintáctica que realiza un constituyente nominal en la oración y, al mismo tiempo, la forma flexiva que los sustantivos, adjetivos, determinantes, etc. toman cuando realizan dicha función.*
Constituyente	*Un constituyente sintáctico es un conjunto de palabras que funciona como una unidad dentro de la estructura sintáctica de la oración.*
Fill-gap	*Tipo de ejercicio gramatical que consiste en presentar una oración con algunas palabras eliminadas; el objetivo del ejercicio es que el alumno rellene los huecos.*
Flexión	*Proceso de alteración que sufren las palabras para expresar los significados gramaticales que requiere su función en la oración.*
Lexema	*El lexema de una palabra es su raíz: la parte que permanece invariante en los procesos de flexión.*
Morfema	*Por oposición a lexema, llamamos morfema a las partes de la palabra que varían en los procesos de flexión.*
Mónada	*En programación funcional, una mónada es un tipo que envuelve a otro tipo (un valor). Este envoltorio proporciona un contexto computacional al valor.*
NLG	*«Natural language generation» o generación del lenguaje natural.*
NLP	*«Natural language processing» o procesamiento del lenguaje natural.*
***Type class* (clase de tipo)**	*En Haskell, las clases de tipo son el mecanismo que se utiliza para permitir el polimorfismo* ad-hoc. *Son similares a las interfaces de Java, en cuanto que describen el comportamiento que debe implementar el tipo que los implementa.*

9 Bibliografía

[1] Eustat, «Nivel del alumnado de los euskaltegis, por territorio y sexo. 2019/20», *Eustat.eus*, mar. 17, 2021. http://www.eustat.eus/elementos/tbl0015053_c.html (accedido sep. 22, 2021).

[2] Council of Europe, Council for Cultural Co-operation, Education Committee, y Modern Languages Division, *Common European framework of reference for languages: learning, teaching, assessment*. Cambridge: Cambridge University Press, 2001.

[3] «Ikasten. Cursos de euskara online.» https://www.hiru.eus/es/e-ikasi/idiomas/ikasten (accedido oct. 09, 2021).

[4] J. Metsämuuronen, «Effect of Repeated Testing on the Development of Secondary Language Proficiency», *J. Educ. Dev. Psychol.*, vol. 3, n.º 1, Art. n.º 1, ene. 2013, doi: 10.5539/jedp.v3n1p10.

[5] P. Smolen, Y. Zhang, y J. H. Byrne, «The right time to learn: mechanisms and optimization of spaced learning», *Nat. Rev. Neurosci.*, vol. 17, n.º 2, pp. 77-88, feb. 2016, doi: 10.1038/nrn.2015.18.

[6] I. Aldabe, M. L. de Lacalle, M. Maritxalar, E. Martinez, y L. Uria, «ArikIturri: An Automatic Question Generator Based on Corpora and NLP Techniques», en *Intelligent Tutoring Systems*, Berlin, Heidelberg, 2006, pp. 584-594. doi: 10.1007/11774303_58.

[7] L. Schwartz, T. Aikawa, y M. Pahud, «Dynamic language learning tools», 2004.

[8] R. Mitkov, «Computer-aided generation of multiple-choice tests», en *Proceedings of the HLT-NAACL 03 Workshop on Building Educational Applications Using Natural Language Processing*, 2003, pp. 17-22.

[9] O. Kraif, G. Antoniadis, S. Echinard, M. Loiseau, T. Lebarbé, y C. Ponton, «NLP tools for CALL: the simpler, the better», presentado en InSTIL/ICALL Symposium, 2004.

[10] A. Hoshino y H. Nakagawa, «A Real-Time Multiple-Choice Question Generation For Language Testing: A Preliminary Study», en *Proceedings of the Second Workshop on Building Educational Applications Using NLP*, Ann Arbor, Michigan, jun. 2005, pp. 17-20. Accedido: sep. 27, 2021. [En línea]. Disponible en: https://aclanthology.org/W05-0203

[11] C.-L. Liu, C.-H. Wang, Z.-M. Gao, y S.-M. Huang, «Applications of Lexical Information for Algorithmically Composing Multiple-Choice Cloze Items», en *Proceedings of the Second Workshop on Building Educational Applications Using NLP*, Ann Arbor, Michigan, jun. 2005, pp. 1-8. Accedido: sep. 18, 2021. [En línea]. Disponible en: https://aclanthology.org/W05-0201

[12] L. Perez-Beltrachini, C. Gardent, y G. Kruszewski, «Generating Grammar Exercises», en *The 7th Workshop on Innovative Use of NLP for Building Educational Applications, NAACL-HLT Worskhop 2012*, Montreal, Canada, jun. 2012, pp. 147-157. Accedido: sep. 18, 2021. [En línea]. Disponible en: https://hal.archives-ouvertes.fr/hal-00768610

[13] I. Dennis, S. Handley, P. Bradon, J. Evans, y S. Newstead, «Approaches to Modeling Item-Generative Tests», en *Item Generation for Test Development*, Routledge, 2002.

[14] M. K. Singley y R. E. Bennett, «Item generation and beyond: Applications of schema theory to mathematics assessment», en *Item generation for test development*, Mahwah, NJ, US: Lawrence Erlbaum Associates Publishers, 2002, pp. 361-384.

[15] P. Deane y K. Sheehan, *Automatic Item Generation via Frame Semantics: Natural Language Generation of Math Word Problems*. 2003. Accedido: sep. 18, 2021. [En línea]. Disponible en: https://eric.ed.gov/?id=ED480135
[16] C. J. Fillmore, «Frame semantics and the nature of language», en *Annals of the New York Academy of Sciences: Conference on the origin and development of language and speech*, 1976, vol. 280, n.º 1, pp. 20-32.
[17] «About FrameNet». https://framenet.icsi.berkeley.edu/fndrupal/about (accedido sep. 27, 2021).
[18] E. Reiter y R. Dale, «Building applied natural language generation systems», *Nat Lang Eng*, 1997, doi: 10.1017/S1351324997001502.
[19] R. T. Kasper, «A Flexible Interface for Linking Applications to Penman's Sentence Generator», presentado en HLT 1989, 1989. Accedido: sep. 18, 2021. [En línea]. Disponible en: https://aclanthology.org/H89-1022
[20] M. Halliday y C. Matthiessen, *An Introduction to Functional Grammar*. Routledge, 2014.
[21] J. A. Bateman, «Sentence generation and systemic grammar: an introduction», *Iwanami Lect. Ser. Lang. Sci. Iwanami Shoten Publ. Tokyo*, 1997.
[22] C. Matthiessen y J. A. Bateman, *Text Generation and Systemic-Functional Linguistics: Experiences from English and Japanese*. London: Pinter, 1991.
[23] W. C. Mann, «An Overview of the Nigel Text Generation Grammar», en *21st Annual Meeting of the Association for Computational Linguistics*, Cambridge, Massachusetts, USA, jun. 1983, pp. 79-84. doi: 10.3115/981311.981326.
[24] T. Becker, «Fully Lexicalized Head-Driven Syntactic Generation», Niagara-on-the-Lake, Ontario, Canada, ago. 1998. Accedido: sep. 27, 2021. [En línea]. Disponible en: https://aclanthology.org/W98-1422
[25] G. Wilcock, «Head-driven generation with HPSG», en *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 2*, 1998, pp. 1393-1397.
[26] S. Müller, «HPSG and construction grammar», en *Head-Driven Phrase Structure Grammar: The handbook*, 2019, pp. 941-992.
[27] J. A. Bateman, R. Henschel, y F. Rinaldi, «The Generalized Upper Model 2.0», presentado en Proceedings of the ECAI94 Workshop: Comparison of Implemented Ontologies, 1995.
[28] J. A. Bateman, J. Hois, R. Ross, y T. Tenbrink, «A linguistic ontology of space for natural language processing», *Artif. Intell.*, vol. 174, n.º 14, pp. 1027-1071, sep. 2010, doi: 10.1016/j.artint.2010.05.008.
[29] S. Marlow y otros, «Haskell 2010 Language Report», 2010. Accedido: nov. 11, 2021. [En línea]. Disponible en: https://www.haskell.org/onlinereport/haskell2010
[30] A. C. Schalley, «Ontologies and ontological methods in linguistics», *Lang. Linguist. Compass*, vol. 13, n.º 11, p. e12356, 2019.
[31] S. Farrar, W. D. Lewis, y D. T. Langendoen, «A common ontology for linguistic concepts», en *Proceedings of the Knowledge Technologies Conference*, 2002, pp. 10-13.
[32] «The Haskell Tool Stack». https://docs.haskellstack.org/en/stable/README/ (accedido ene. 04, 2022).
[33] «Haddock 1.0 documentation». https://haskell-haddock.readthedocs.io/en/latest/# (accedido ene. 04, 2022).
[34] S. F. Chen, L. Mangu, B. Ramabhadran, R. Sarikaya, y A. Sethy, «Scaling shrinkage-based language models», en *2009 IEEE Workshop on Automatic Speech Recognition & Understanding*, 2009, pp. 299-304.
[35] A. Celikyilmaz, R. Sarikaya, M. Jeong, y A. Deoras, «An empirical investigation of word class-based features for natural language understanding», *IEEEACM Trans. Audio Speech Lang. Process.*, vol. 24, n.º 6, pp. 994-1005, 2015.

[36] R. M. Burstall, D. B. MacQueen, y D. T. Sannella, «HOPE: An experimental applicative language», en *Proceedings of the 1980 ACM conference on LISP and functional programming*, 1980, pp. 136-143.
[37] D. Piponi, «Haskell Monoids and their Uses». http://blog.sigfpe.com/2009/01/haskell-monoids-and-their-uses.html (accedido dic. 18, 2021).
[38] N. Chomsky, *Aspects of the Theory of Syntax*, vol. 11. MIT press, 2014.
[39] I. A. Mel'cuk, *Dependency syntax: theory and practice*. Albany: State University of New York Press, 1988.
[40] L. Meertens, «First steps towards the theory of rose trees», *IFIP Work. Group*, n.º Working paper 592, 1988.
[41] R. Bird, *Introduction to functional programming using Haskell*, 2nd ed. London [etc.]: Prentice Hall, 1998.
[42] M. Uschold, «Ontology and database schema: What's the difference?», *Appl. Ontol.*, vol. 10, n.º 3-4, pp. 243-258, 2015.
[43] «Neo4j documentation», *Neo4j Graph Database Platform*. https://neo4j.com/docs/docs/ (accedido dic. 26, 2021).
[44] «The Neo4j Cypher Manual v4.4», *Neo4j Graph Database Platform*. https://neo4j.com/docs/cypher-manual/4.4/ (accedido dic. 26, 2021).
[45] P. Yakovlev, *Hasbolt*. [En línea]. Disponible en: https://github.com/zmactep/Hasbolt
[46] J. A. Bateman, «Upper modeling: a general organization of knowledge for natural language processing», presentado en Proceedings of the 5th International Workshop on Natural Language Generation, 1990.
[47] E. Hovy, «The Penman natural language project», en *Proceedings of the workshop on Speech and Natural Language*, USA, jun. 1990, p. 430. doi: 10.3115/116580.1138614.
[48] E. H. Hovy, «The current status of the Penman language generation system», en *Proceedings of the workshop on Speech and Natural Language*, USA, oct. 1989, p. 449. doi: 10.3115/1075434.1075512.
[49] «Neo4j APOC Library - Developer Guides», *Neo4j Graph Database Platform*. https://neo4j.com/developer/neo4j-apoc/ (accedido ene. 03, 2022).
[50] E. Moggi, *Computational lambda-calculus and monads*. University of Edinburgh, Department of Computer Science, 1988.
[51] P. Wadler, «Comprehending monads», en *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 1990, pp. 61-78.
[52] P. Wadler, «The Essence of Functional Programming», 1992, pp. 1-14.
[53] S. Mac Lane, *Categories for the working mathematician*, vol. 5. Springer Science & Business Media, 2013.
[54] D. Piponi, «You Could Have Invented Monads! (And Maybe You Already Have.)», *A Neighborhood of Infinity*. http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html (accedido dic. 28, 2021).
[55] O'Sullivan, Bryan, D. Stewart, y J. Goerzen, *Real World Haskell*, 1st ed. Sebastopol, CA: O'Reilly Media, Inc., 2008.
[56] M. O'Donnell, «Sentence Analysis and Generation–a Systemic Perspective», PhD Thesis, University of Sydney, 1994.
[57] W. C. Mann y C. M. Matthiessen, «Nigel: A Systemic Grammar for Text Generation», University of Southern California. Marina del Rey Information Sciences Institute., 1983.