Generador de retroalimentación detallada para DOMjudge

Detailed feedback generator for DOMjudge

Trabajo Fin de Grado
Curso 2021-2022

Autores
Ricardo Enrique Freire Sacco
Adrián González Cabanillas
Erik Karlgren Domercq
Félix Redondo Manzanares

Directores
Enrique Martín Martín
Manuel Montenegro Montes

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Resumen

DOMfeed es una aplicación web que ofrece retroalimentación a los participantes de un concurso de programación que se lleve a cabo mediante el sistema DOMjudge. DOMjudge no ofrece la posibilidad de mostrar qué casos de prueba de un problema han fallado, ya que únicamente informa al usuario de si una solución es válida o no. En DOMfeed, sin embargo, para cada entrega inválida por parte de un usuario se muestra el caso de prueba erróneo. De esta forma, el usuario podrá agilizar el proceso de depuración de su solución. No obstante, el administrador del concurso deberá describir el formato de los ficheros de entrada y salida para que se almacenen en la base de datos de DOMfeed. Para ello definimos un lenguaje de especificaciones que permite, dado un problema, determinar cómo separar los casos de prueba en la entrada y la salida del mismo.

Palabras Clave

DOMjudge, retroalimentación, juez de programación, React, Node.js, aplicación web, DOMfeed, web scraping, Express

Abstract

DOMfeed is a web application that provides feedback to the participants of a programming contest conducted using the DOMjudge system. DOMjudge does not offer the possibility to show which test cases of a problem have failed, as it only informs the user whether a solution is valid or not. In DOMfeed, however, for each invalid submission by a user, the wrong test case is displayed. In this way, the user can speed up the debugging process of their solution. However, the contest administrator must describe the format of the input and output files to be stored in the DOMfeed database. For this purpose, we define a specification language that allows, given a problem, to determine how to separate the test cases in the input and output of the problem.

Keywords

DOMjudge, feedback, programming judge, React, Node.js, web application, DOMfeed, web scraping, Express

Índice general

1. Introducción **13**
1.1. Motivación 13
1.2. Objetivos 14
1.3. Plan de trabajo 15
1.4. Organización de la memoria 17

2. Introduction **19**
2.1. Motivation 19
2.2. Goals 20
2.3. Work plan 21
2.4. Report organisation 22

3. Preliminares **25**
3.1. Aprendizaje de la programación y el uso de problemas 25
3.2. Jueces de programación y su importancia en el aprendizaje 26
3.3. Limitaciones de los jueces de programación 27
3.3.1. COMPILER-ERROR 27
3.3.2. WRONG-ANSWER 27
3.4. DOMjudge como juez de programación y su API 27
3.5. Gramáticas formales 28
3.5.1. Definición 29
3.5.2. Ejemplos 29
3.6. Tecnologías usadas 30
3.6.1. JavaScript 30
3.6.2. Node.js 30
3.6.3. Express.js 32
3.6.4. React 32
3.6.5. React Router 32
3.6.6. Bootstrap 34
3.6.7. React Bootstrap 34
3.6.8. MySQL 34
3.6.9. PEG.js 36
3.6.10. Jest 37

4. Organización del sistema **39**
4.1. DOMjudge . . . 39
4.2. Aplicación web o *frontend* . . . 40
4.2.1. Vistas del administrador . . . 40
4.2.2. Vistas del usuario . . . 40
4.2.3. Máximo de correcciones . . . 42
4.3. Servidor o *backend* . . . 42
4.3.1. npm . . . 42
4.3.2. Peticiones del usuario . . . 43
4.3.3. Datos de DOMjudge . . . 43
4.3.4. Datos de la base de datos . . . 44
4.3.5. Procesamiento de los datos . . . 44
4.4. Base de datos . . . 44
4.5. Analizador de casos de prueba . . . 45
4.5.1. Comprobación de la validez de una especificación . . . 45
4.5.2. Separación de casos de prueba . . . 46

5. Analizador de casos de prueba **47**
5.1. Lenguaje de especificación . . . 47
5.1.1. Requisitos del lenguaje . . . 47
5.1.2. Descripción del lenguaje . . . 49
5.1.3. Ejemplos de especificaciones . . . 52
5.1.3.1. Ejemplo 1 . . . 52
5.1.3.2. Ejemplo 2 . . . 52
5.1.3.3. Ejemplo 3 . . . 52
5.1.3.4. Ejemplos problemáticos . . . 53
5.1.4. API de comprobación de la validez de una especificación: `tryToParseGrammar()` . . . 54
5.2. Separación de los casos de prueba . . . 54
5.2.1. Implementación de `separateCases()` . . . 55
5.2.1.1. API de separación de casos: `separateCases()` . . . 55
5.2.1.2. Función auxiliar: `_separateCases()` . . . 56
5.2.2. TokenStream . . . 59
5.2.3. Reglas de procesamiento . . . 60
5.2.3.1. Especificaciones aplicables a cadenas vacías . . . 60
5.2.3.2. Evaluación de una expresión aritmética . . . 61
5.2.3.3. Definiciones de las reglas . . . 62
5.3. Verificación del correcto funcionamiento del analizador . . . 65

6. Servidor principal: *frontend* y *backend* **67**
6.1. *Frontend* . . . 67
6.1.1. Inicio de sesión . . . 68
6.1.2. Cabecera . . . 68
6.1.3. Páginas del administrador . . . 69
6.1.3.1. Vista de problemas . . . 69

6.1.3.2. Vista de especificaciones 71
6.1.3.3. Vista de entregas 72
6.1.4. Páginas del usuario 73
6.1.4.1. Vista de entregas 73
6.1.5. Vista de correcciones 75
6.1.6. Código de interés 77
6.1.6.1. Mantener la sesión 77
6.1.6.2. Páginas protegidas 78
6.1.6.3. Notificaciones 81
6.2. *Backend* 83
6.2.1. Base de datos 84
6.2.1.1. Estructura de la base de datos 84
enabled_submissions 84
problem_specifications 85
max_contest_attempts 86
max_problem_attempts 86
6.2.1.2. Acceso a la base de datos 86
6.2.2. API de DOMfeed 87
6.2.2.1. *Midlewares* 87
6.2.2.2. Obtención de datos de DOMjudge 88
Concursos por equipos 89
Problemas por concurso 89
Entregas por equipos 89
Entregas por problemas 89
6.2.2.3. Gestión del número de correcciones de problemas y concursos 89
Número de correcciones máximo de un concurso . . . 89
Número de correcciones máximo de un problema . . . 90
Número de correcciones disponibles de un equipo para un problema 90
6.2.2.4. Gestión de especificaciones de un problema 90
Análisis de especificaciones 90
6.2.2.5. Separación de casos de prueba 91
Obtención de casos de prueba 91
Separación de los ficheros 92
6.2.2.6. Gestión de correcciones 92
Habilitación de entregas 92
Corrección de una entrega 92
6.2.3. *Web scraping* 92

7. Conclusiones y trabajo futuro **95**
7.1. Objetivos alcanzados 95
7.2. Dificultades encontradas 96
7.2.1. Grupo 1 96
7.2.2. Grupo 2 96

7.3. Trabajo futuro . . . 97
7.3.1. Lenguaje de especificación . . . 97
7.3.2. Implementación de la separación de casos de prueba . . . 97
7.3.3. *Frontend* . . . 98
7.3.4. *Backend* . . . 98

8. Conclusions and Future Work 99
8.1. Objectives achieved . . . 99
8.2. Difficulties encountered . . . 100
8.2.1. Group 1 . . . 100
8.2.2. Group 2 . . . 100
8.3. Future work . . . 101
8.3.1. Specification language . . . 101
8.3.2. Implementation of test case separation . . . 101
8.3.3. Frontend . . . 101
8.3.4. *Backend* . . . 102

9. Contribuciones personales 103
9.1. Félix Redondo Manzanares . . . 103
9.1.1. *Scraping* . . . 103
9.1.2. Algoritmo de búsqueda de caso erróneo . . . 103
9.1.3. API de DOMfeed . . . 104
9.1.4. Páginas de la interfaz web . . . 104
9.1.5. Sistema de notificaciones . . . 104
9.1.6. Páginas privadas . . . 105
9.1.7. Mantenimiento de la sesión del usuario . . . 105
9.1.8. Segurización de la API . . . 105
9.1.9. Llamada a la API de DOMjudge . . . 105
9.2. Ricardo Enrique Freire Sacco . . . 105
9.2.1. API de DOMfeed . . . 106
9.2.2. Base de datos . . . 106
9.2.3. Gestión del número de correcciones . . . 106
9.2.4. Gestión de especificaciones . . . 107
9.2.5. Interfaz web . . . 107
9.2.6. Integración del separador de casos en la aplicación web . . . 107
9.3. Adrián González Cabanillas . . . 107
9.3.1. Gramática del lenguaje de especificación . . . 107
9.3.2. Implementación de reglas . . . 108
9.3.3. Variables y estados . . . 108
9.3.4. Especificaciones que enlazan la entrada con la salida . . . 108
9.3.5. Elaboración de tests . . . 109
9.3.6. Agrupación de casos de prueba separados . . . 109
9.3.7. Manual de especificaciones . . . 109
9.4. Erik Karlgren Domercq . . . 109
9.4.1. TokenStream . . . 109

9.4.2. Gramática del lenguaje de especificación 110
9.4.3. Reglas de procesamiento 110
9.4.4. Separación de casos de la entrada 110
9.4.5. Elaboración de tests 111
9.4.6. Documentos 111

A. Ejemplos de especificaciones **I**

A.1. Obtener el mínimo de un vector I
A.1.1. Enunciado I
A.1.2. Especificaciones II
A.2. El Grande III
A.2.1. Enunciado IV
A.2.2. Especificaciones V

Bibliografía **IX**

Índice de figuras

1.1. Diagrama de Gantt de las tareas de cada grupo a lo largo del curso 2021/2022 16

2.1. Gantt chart of the tasks of each group during the academic year 2021/2022 21

3.1. Ejemplo de un problema de programación 25
3.2. Implementación en Python de un algoritmo de análisis descendente para nuestro lenguaje natural simple 31
3.3. Ejemplo de servidor en Express.js 33
3.4. Comparativa de Bootstrap con React-Bootstrap 35
3.5. Gramática para expresiones aritméticas para PEG.js 36
3.6. Ejecución de Jest para una implementación no válida de una función de suma. El comando `npm test` llama internamente a Jest en este proyecto. 38

4.1. Diagrama explicativo con los diferentes componentes del sistema . . 39
4.2. Vista de administrador 41
4.3. Vista de usuario 41

5.1. Esquema de un fichero de casos de prueba de entrada 48
5.2. Esquema de un fichero de casos de prueba de entrada con separadores 48
5.3. Gramática ambigua del lenguaje de especificación 50
5.4. Gramática desambiguada del lenguaje de especificación 51
5.5. `tryToParseGrammar()`: Función para comprobar la validez de una especificación 54
5.6. Implementación de la función `separateCases()` 56
5.7. Código de la función `_separateCases()`, que describe cómo se procesan los casos de prueba de entrada y salida 58
5.8. Resultado de ejecutar los tests del analizador de casos de prueba . . 66

6.1. Página de inicio de sesión 68
6.2. Cabecera con el concurso *demo* seleccionado 69
6.3. Página de problemas 71
6.4. Página de especificaciones 72
6.5. Página de entregas para administradores 73
6.6. Ventana emergente para confirmar la corrección 74

6.7. Página de entregas para usuarios normales 75
6.8. Página de correcciones 76
6.9. Esquema de caso de prueba 77
6.10. Contexto de la aplicación 78
6.11. Código del contexto y hook useAuth 79
6.12. Componente Layout general 80
6.13. Componente vista protegida 80
6.14. Ejemplo de enrutado 81
6.15. Contexto de las notificaciones 82
6.16. Ejemplo de uso de las notificaciones 83
6.17. Hook useNotification 83
6.18. Esquema de la base de datos 85
6.19. Creación del *pool* de conexiones 87
6.20. Ejemplo de consulta SQL utilizando el pool de conexiones 87
6.21. Estructura de los *endpoints* de la API de DOMfeed 88
6.22. Middleware para extraer el token 88

A.1. Mínimo de un vector: casos de prueba sin separar II
A.2. Mínimo de un vector: casos de prueba separados III
A.3. El Grande: casos de prueba sin separar VI
A.4. El Grande: casos de prueba separados VII

Capítulo 1

Introducción

En esta memoria describimos el proceso para diseñar y crear un generador de retroalimentación detallada para DOMjudge al que hemos decidido llamar **DOMfeed**. Empezaremos esta sección mencionando qué nos motivó a crear DOMfeed y cuáles eran nuestros objetivos con este proyecto. Finalmente, detallaremos tanto nuestro plan de trabajo como la estructura de la memoria.

1.1. Motivación

DOMjudge es un juez de programación destinado principalmente a competiciones de programación, pero también se usa frecuentemente en ciertas asignaturas que se cursan en la Facultad de Informática de la Universidad Complutense de Madrid. La mayoría de estas asignaturas están orientadas a la enseñanza de algoritmos y estructuras de datos, para lo cual es fundamental que el alumno practique con problemas de programación que pongan a prueba sus conocimientos.

Cuando un usuario envía una solución de un determinado problema a DOMjudge, este sistema compila y ejecuta la solución, comprobando su validez mediante una serie de casos de prueba almacenados en el servidor. DOMjudge captura la salida emitida por la solución, y comprueba que coincide con la salida esperada para ese problema. En caso afirmativo, el juez responde al usuario con un veredicto `CORRECT`. Por el contrario, si hay alguna discrepancia entre la salida obtenida y la esperada, se responde con un veredicto `WRONG-ANSWER`.

No obstante, ese último veredicto se limita a informar al alumno de que su solución es incorrecta, con lo que el alumno no recibe ninguna pista que le pueda ayudar a arreglar su solución. Esto le obliga, o bien, a pensar qué puede haber hecho mal, o bien a preguntarle a su profesor qué problema presenta su solución o en qué caso de prueba falla.

Aunque consideramos que un alumno necesita dedicar tiempo a pensar qué puede estar fallando con su solución, también creemos que sería conveniente que hubiera alguna forma automática de darle una pista. Además, a pesar de que hemos encontrado algunos jueces de programación que muestran una retroalimentación más detallada que DOMjudge (por ejemplo, Jutge.org [10, 17]), no hemos encontrado ninguna versión modificada de DOMjudge o aplicación que interactúe

con este que muestre más información para las soluciones erróneas. Es por ello que decidimos crear DOMfeed, una aplicación web cuyo papel es ofrecer al alumno una retroalimentación más detallada que la de DOMjudge para soluciones erróneas.

1.2. Objetivos

A la hora de desarrollar DOMfeed, como comentamos anteriormente, queríamos ofrecer una mejor retroalimentación al alumno cuando una solución realizada por él o su equipo no generara la salida esperada. Para ello consideramos conveniente mostrarle los casos de prueba para los que ha fallado la solución, lo cual nos obligó a:

1. Encontrar una manera de separar los casos de prueba.
2. Dados los casos de prueba separados, comprobar en qué casos de prueba falla el alumno.

El segundo paso era el más sencillo, pues una vez tenemos todos los casos de prueba separados, es relativamente trivial comparar la salida esperada y la del alumno caso por caso hasta encontrar uno en el que difieran. No obstante, separar los casos de prueba de un fichero es un problema significativamente más complejo, pues no todos los problemas de programación representan los casos de prueba de la misma forma. Por ende, decidimos crear un lenguaje para especificar casos de prueba y es tarea del profesor el definir una especificación correcta para la entrada y la salida de un problema. Dicho lenguaje necesitará una forma de delimitar dónde empieza y dónde acaba cada caso de prueba.

Por otro lado, decidimos que DOMfeed sería una aplicación web al igual que DOMjudge. Por tanto, necesitamos crear una página web con la que interactuaría tanto el alumno como el profesor (el *frontend*), y un *backend* para interactuar con DOMjudge junto con una base de datos propia. La interacción con DOMjudge es necesaria para poder consultar las entregas y problemas con sus respectivos ficheros con los casos de prueba de entrada y de salida, evitando de esta manera que el profesor tenga que subir dichos ficheros por su cuenta a DOMfeed. De hecho, también nos permite controlar que solo puedan acceder a DOMfeed alumnos y profesores que estén registrados en DOMjudge usando las mismas credenciales. La base de datos, en cambio, la necesitamos para guardar información que no podemos obtener de DOMjudge, como las especificaciones de cada problema suministradas por los profesores.

En esta aplicación web, el alumno podría ver las entregas fallidas de DOMjudge y descubrir los casos de prueba en los que haya fallado cada una. No obstante, para evitar que el alumno abuse de este sistema, decidimos que el profesor pudiera limitar el número de veces que el alumno podría ver dicha información tanto por problema como por concurso. Asimismo, los profesores usarían la aplicación web para suministrar las especificaciones de cada problema, como hemos mencionado.

En conclusión, hemos definido los siguientes objetivos principales:

- La definición de un lenguaje de especificación de casos de prueba y la implementación de un mecanismo que permita separar los casos de prueba de un problema a partir de una especificación.

- El desarrollo de una aplicación web que permita al alumno ver en qué caso de prueba ha fallado la entrega realizada por este y al administrador registrar las especificaciones necesarias para que se puedan separar los casos de prueba del problema.

1.3. Plan de trabajo

Al principio del curso académico 2021/2022, nos dividimos en dos grupos, de forma que Erik Karlgren Domercq y Adrián González Cabanillas (*Grupo 1*) tendrían el objetivo de conseguir la separación de casos de prueba, mientras que Félix Redondo Manzanares y Ricardo Enrique Freire Sacco se encargarían de la aplicación web (*Grupo 2*). Para ambas partes, establecimos metas de trabajo cada dos semanas, momento en el que nos reuníamos con los tutores del proyecto (Enrique Martín Martín y Manuel Montenegro Montes) para mostrar el trabajo que habíamos avanzado. También comentábamos los problemas que nos habían surgido y cómo los habíamos solucionado. Para finalizar estas reuniones se concretaban nuevos objetivos que teníamos que realizar para las siguientes dos semanas en el caso de haber completado las tareas pendientes.

A modo de síntesis, se ha dividido el proyecto en varias fases, indicando qué objetivos se han logrado en cada una y en qué capítulos de la memoria se encuentran explicadas. Los hitos alcanzados se observan en la figura 2.1 y se describen a continuación:

- **Análisis del proyecto** (*20 de septiembre de 2021 - 7 de octubre de 2021*) Establecimos los objetivos y el alcance del proyecto. Comenzamos investigando las tecnologías que usaríamos en el proyecto. Nuestras conclusiones se detallan en la sección 3.6. El grupo 1 se encargó de realizar pruebas de concepto con la librería PEG.js mientras que el grupo 2 se dedicó a familiarizarse con la interfaz de DOMjudge así como con su API.

- **Desarrollo durante el primer cuatrimestre** (*8 de octubre de 2021 - 31 de enero de 2022*) Durante esta etapa, el grupo 1 estuvo implementando las características más básicas del lenguaje de especificación y de la separación de casos de prueba, añadiendo cada dos semanas nuevas funcionalidades. Por otro lado, el grupo 2 estuvo trabajando en una primera versión de la aplicación web de DOMfeed, donde se desarrolló la primera iteración del algoritmo de búsqueda del caso de prueba erróneo, los primeros *endpoints* de la API de DOMfeed, y el desarrollo de las páginas de visualización de problemas, entregas y correcciones. También se realizó todo el *web scraping* necesario para permitir la descarga de la salida obtenida del usuario.

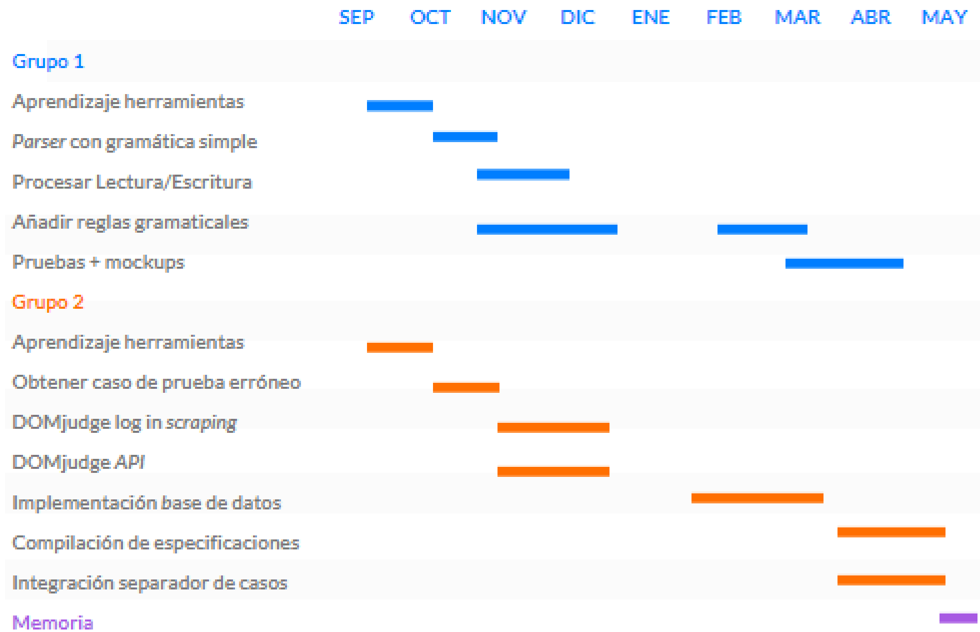


Figura 1.1: Diagrama de Gantt de las tareas de cada grupo a lo largo del curso 2021/2022

- **Desarrollo durante el segundo cuatrimestre** (*1 de febrero de 2022 - 31 de marzo de 2022*) El grupo 1 completó el desarrollo del lenguaje de especificación y la separación de casos, cuyos resultados pueden consultarse en el capítulo 5. Para probar que la implementación de ambos era la esperada, se crearon múltiples tests usando Jest (ver sección 5.3). El grupo 2 se centró en perfeccionar todo el trabajo realizado en el primer cuatrimestre y en crear tanto las pantallas de gestión del administrador, como la base de datos necesaria para su funcionamiento. También se modificó el algoritmo de búsqueda del caso erróneo, pues se introdujeron nuevos cambios en la especificación. El resultado final del trabajo del grupo 2 se puede consultar en el capítulo 6.

- **Integración del trabajo de ambos grupos** (*1 de abril de 2022 - 30 de abril de 2022*) Tras el desarrollo en paralelo de ambos grupos, se lleva a cabo la integración del analizador de casos de prueba con la aplicación web. El resultado de dicha integración se explica en el capítulo 4. El objetivo principal de esta fase es comprobar que la especificación introducida es válida, así como realizar la separación de los casos de prueba utilizando el trabajo realizado por el grupo 1. Además, el grupo 2 hace nuevas tareas de *web scraping* para obtener los ficheros de prueba del problema a tratar, pues hasta ese momento se estaban utilizando ficheros guardados de forma local.

- **Elaboración de la memoria** (*1 de mayo de 2022 - 30 de mayo de 2022*) Elaboramos conjuntamente este documento con ayuda de nuestros tutores, que se encargaron de revisarlo en varias ocasiones.

1.4. Organización de la memoria

La estructura de la memoria es la siguiente:

1. **Preliminares:** En el capítulo 3 explicamos todos los conceptos necesarios para entender el resto de esta memoria y del proyecto.

2. **Organización del sistema:** Explicamos cómo están interconectados los componentes de DOMfeed y qué tecnologías utilizan en el capítulo 4.

3. **Analizador de casos de prueba:** Definimos el lenguaje de especificación y describimos la separación de los casos de prueba en el capítulo 5.

4. **Backend y frontend:** Explicamos cómo funciona el servidor. Esto incluye la página web, la base de datos, cómo se utiliza el analizador de casos y cómo se actualizan los casos de prueba con separadores, así como la petición de datos a DOMjudge y su posterior tratamiento mediante la utilización de su API y el uso de *web scraping* para los casos donde la API de DOMjudge no proporciona los datos que necesitamos. Ver capítulo 6.

5. **Conclusiones y trabajo futuro:** Recapitulamos los objetivos conseguidos, discutimos hasta qué punto se han logrado y comentamos qué más cosas querríamos añadir a DOMfeed en el capítulo 7(la sección *Conclusions and future work* es idéntica pero en inglés, capítulo 8).

6. **Contribuciones personales:** Mencionamos las tareas que ha llevado a cabo cada uno de nosotros en el capítulo 9.

7. **(Apéndice) Ejemplos de especificación:** Por último, presentamos ejemplos de especificaciones para problemas reales de programación en el apéndice A.

Capítulo 2

Introduction

In this report we describe the process of designing and creating a detailed feedback generator for DOMjudge which we have decided to call **DOMfeed**. We will start this section by mentioning what motivated us to create DOMfeed and what our goals were with this project. Finally, we will detail both our work plan and report structure.

2.1. Motivation

DOMjudge is a programming judge mainly intended for programming competitions, but it is also frequently used in certain subjects taken in the Faculty of Computer Science at the Complutense University of Madrid. Most of these subjects are oriented towards the teaching of algorithms and data structures, for which it is essential that students practice with programming problems that test their knowledge.

When a user submits a solution to a given problem to DOMjudge, DOMjudge compiles and executes the solution, checking its validity through a series of test cases stored on the server. DOMjudge captures the output emitted by the solution, and checks that it matches the expected output for that problem. If it does, the judge responds to the user with a `CORRECT` veredict. Conversely, if there is any discrepancy between the output obtained and the expected output, a `WRONG-ANSWER` verdict is returned.

However, the latter verdict merely informs the student that his solution is incorrect, so the student is not given any clues that might help him to fix his solution. This forces them either to think about what they may have done wrong, or to ask their teacher what problem their solution presents or in which test case it fails.

While we believe that a student needs to spend time thinking about what might be going wrong with their solution, we also believe that some automatic way of giving them a hint would be desirable. Furthermore, although we have found some programming judges that show more detailed feedback than DOMjudge (e.g. Jutge.org [10, 17]), we have not found a modified version of DOMjudge or an application that interacts with DOMjudge that shows more information for wrong

solutions. This is why we decided to create DOMfeed, a web application whose role is to provide students with more detailed feedback than DOMjudge for wrong solutions.

2.2. Goals

When developing DOMfeed, as mentioned above, we wanted to provide better feedback to the student when a solution developed by him or his team did not generate the expected output. To do so, we considered it convenient to show him the test cases for which the solution failed, which forced us to:

1. Find a way to separate test cases.

2. Given separate test cases, check which test cases the student fails.

The second step was the simplest, because once we have all the separate test cases, it is relatively trivial to compare the expected output and the student's output on a case-by-case basis until we find one where they differ. However, separating test cases from a file is a significantly more complex problem, as not all programming problems represent test cases in the same way. Therefore, we decided to create a language for specifying test cases, and it is the task of the teacher to define a correct specification for the input and output of a problem. Such a language will need a way to delimit where each test case starts and where it ends.

On the other hand, we decided that DOMfeed would be a web application like DOMjudge. Therefore, we need to create a web page with which both the student and the teacher would interact (the *frontend*), and a *backend* to interact with DOMjudge together with its own database. The interaction with DOMjudge is necessary to be able to consult the deliveries and problems with their respective files with the input and output test cases, thus avoiding the teacher having to upload these files to DOMfeed on his own. In fact, it also allows us to control that only students and teachers who are registered in DOMjudge using the same credentials can access DOMfeed. The database, on the other hand, is needed to store information that we cannot get from DOMjudge, such as the specifications of each problem provided by the teachers.

In this web application, the student would be able to see the failed DOMjudge submissions and discover the test cases in which each failed. However, to prevent the student from abusing this system, we decided that the teacher could limit the number of times the student could view this information per problem and per quiz. Also, teachers would use the web application to provide the specifications for each problem, as mentioned above.

In conclusion, we have defined the following main goals:

- The definition of a test case specification language and the implementation of a mechanism to separate the test cases of a problem from a specification.

- The development of a web application that allows the student to see in which test case the student's delivery has failed and the administrator to record the necessary specifications so that the test cases can be separated from the problem.

2.3. Work plan

At the beginning of the 2021/2022 academic year, we divided into two groups, so that Erik Karlgren Domercq and Adrián González Cabanillas (*Group 1*) would have the objective of achieving the separation of test cases, while Félix Redondo Manzanares and Ricardo Enrique Freire Sacco would be in charge of the web application (*Group 2*). For both parts, we established work goals every two weeks, at which time we met with the project tutors (Enrique Martín Martín and Manuel Montenegro Montes) to show the work we had progressed. We also discussed the problems that had arisen and how we had solved them. At the end of these meetings, new objectives were set for the following two weeks if we had completed the pending tasks.

Summarising, the project has been divided into several phases, indicating which objectives have been achieved in each phase and in which chapters of the report they are explained. The milestones achieved are shown in figure 2.1 and are described below:

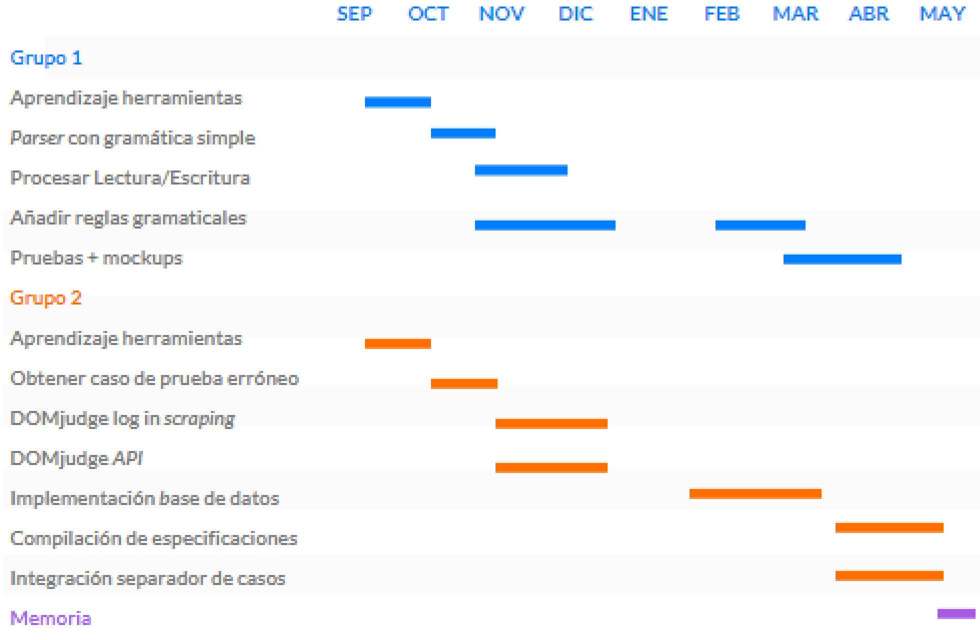


Figura 2.1: Gantt chart of the tasks of each group during the academic year 2021/2022

- **Project analysis** (*20 September 2021 - 7 October 2021*) We established the goals and scope of the project. We began by investigating the technologies

we would use in the project. Our conclusions are detailed in the section 3.6. Group 1 was in charge of performing proofs of concept with the PEG.js library while group 2 spent time familiarising themselves with the DOMjudge interface as well as the DOMjudge API.

- **Development during the first four-month period** (*8 October 2021 - 31 January 2022*)During this stage, group 1 was implementing the most basic features of the specification language and test case separation, adding new functionalities every two weeks. On the other hand, group 2 was working on a first version of the DOMfeed web application, where the first iteration of the failed test case search algorithm, the first DOMfeed API endpoints, and the development of the issue display, delivery and fix pages were developed. All the necessary web scraping was also performed to enable the download of the output obtained from the user.

- **Development during the second four-month period** (*1 February 2022 - 31 March 2022*) Group 1 completed the development of the specification language and the separation of cases, the results of which can be found in chapter 5. To test that the implementation of both was as expected, multiple tests were created using Jest (see section 5.3). Group 2 focused on refining all the work done in the first term and creating both the administrator management screens and the database necessary for their operation. The search algorithm for the erroneous case was also modified, as new changes were made to the specification. The final result of group 2's work can be found in chapter 6.

- **Integration of the work of both groups** (*1 April 2022 - 30 April 2022*) After the parallel development of both groups, the integration of the test case analyser with the web application is carried out. The result of this integration is explained in the chapter 4. The main goal of this phase is to check that the specification introduced is valid, as well as to carry out the separation of the test cases using the work done by group 1. In addition, group 2 performs new web scraping tasks to obtain the test files for the problem to be dealt with, as up to this point, locally stored files were being used.

- **Report** (*1 May 2022 - 30 May 2022*) We jointly developed this document with the help of our tutors, who were responsible for revising it on several occasions.

2.4. Report organisation

The structure of the report is as follows:

1. **Preliminaries:** In chapter 3 we explain all the concepts necessary to understand the rest of this report and the project.

2. **System organisation:** We explain how DOMfeed's components are interconnected and which technologies they use in chapter 4.

3. **Test case analyser:** We define the specification language and describe the separation of the test cases in chapter 5.

4. **Backend y frontend:** We explain how the server works. This includes the web page, the database, how the case parser is used and how test cases are updated with separators, as well as requesting data from DOMjudge and then processing it using its API and using web scraping for cases where the DOMjudge API does not provide the data we need. See chapter 6.

5. **Conclusions and future work:** We recap the objectives achieved, discuss to what extent they have been achieved and discuss what else we would like to add to DOMfeed in chapter 7(section *Conclusions and future work* is identical but in English, chapter 8).

6. **Personal contributions:** We mention the tasks carried out by each of us in chapter 9.

7. **(Appendix) Specification examples:** Finally, we present examples of specifications for real programming problems in the appendix A.

Capítulo 3

Preliminares

En esta sección se listan y se explican los conceptos necesarios para entender el proyecto, al igual que se presentan las tecnologías usadas para el desarrollo del mismo.

3.1. Aprendizaje de la programación y el uso de problemas

A la hora de aprender a programar es fundamental la resolución de problemas para poner a prueba los conocimientos adquiridos. En programación existen muchos tipos de problemas, pero para todos ellos hay tres conceptos principales:

- El problema a resolver en sí, que determina lo que se quiere calcular, y para el cual se necesita definir un algoritmo.
- Las entradas del problema, que son los parámetros u opciones necesarios para que el problema se pueda resolver.
- Y las salidas, que corresponden al resultado del problema una vez que se ha ejecutado teniendo en cuenta las entradas.

Supongamos el siguiente problema: dados dos números enteros, tenemos que calcular su suma. Para ello definimos un programa o algoritmo (por ejemplo, en JavaScript) que recibe dos parámetros, `entrada1` y `entrada2`, y devuelve el resultado de sumar esos dos valores, `resultado`, como podemos ver en la figura 3.1:

```
1 function sumar (var entrada1 , var entrada2) {
2     var resultado = entrada1 + entrada2;
3     return resultado;
4 }
```

Figura 3.1: Ejemplo de un problema de programación

Supongamos ahora que tenemos una entrada, 2 y 3, y queremos ejecutar el problema con esa entrada. El resultado o salida esperado sería 5. En la tabla 3.1 se proporcionan más ejemplos de entradas y salidas para dicho problema.

Entrada	Salida
2, 3	5
6, 1	7
4, 5	9
2, 2	4

Tabla 3.1: Ejemplo de entradas y salidas para el problema 3.1

A partir de ahora, cuando mencionemos **entrada**, corresponderá a los valores que se pasan al problema. Y cuando hablemos de **salida**, será el resultado o resultados esperados. Los **casos de prueba** son una colección de entradas con sus respectivas salidas.

3.2. Jueces de programación y su importancia en el aprendizaje

A menudo se utilizan algunas aplicaciones web donde un usuario sube una solución a un problema de programación previamente definido, y posteriormente recibe un veredicto que resulta de compilar y ejecutar el código para una gran colección de entradas y salidas de valores. Estas aplicaciones se llaman **jueces de programación**. Entre ellas cabría destacar DOMJudge [6], Mooshak [11] y Jutge.org [10].

Inicialmente, estos jueces fueron creados para realizar concursos de programación. Esto se hace así para que todo código que compile y se pueda ejecutar, esté siempre dentro de un mismo entorno con las mismas condiciones de software y hardware. De esta manera, todos los participantes tienen las mismas condiciones. Sin embargo, estos jueces son utilizados también en las universidades como *herramienta de aprendizaje*.

Los profesores o jueces pueden crear un concurso o clase, y dentro pueden añadir problemas con sus entradas y salidas esperadas. Posteriormente, los alumnos pueden ir enviando sus soluciones a los diferentes problemas e irán obteniendo los veredictos.

Esto tiene una gran utilidad, pues para un mismo problema puede haber decenas de soluciones distintas y el trabajo que conlleva para el profesor a la hora de corregirlo es mayúsculo. Sin embargo, con los jueces la corrección se hace en cuestión de segundos.

3.3. Limitaciones de los jueces de programación

Los jueces de programación tienen ciertas limitaciones. **La retroalimentación** al usuario, una vez que ha compilado y ejecutado la solución, es muy escueta y muchas veces insuficiente. Esto es un gran problema, sobre todo cuando se utiliza el juez como herramienta de aprendizaje, ya que el usuario no sabe realmente el motivo por el cual el juez rechaza su solución. La retroalimentación que se recibe solo indica si se ha ejecutado correctamente la solución al problema y genera los resultados esperados o si algo ha salido mal, en cuyo caso indica en qué momento ha ocurrido el error, es decir, si ocurre en tiempo de compilación, en tiempo de ejecución, si el resultado obtenido difiere del esperado, etc.

La mayoría de los veredictos no requieren de información extra, pues la propia salida ya es suficientemente descriptiva. Pero existen algunos veredictos que sí requieren de algo más de información. Los posibles veredictos se muestran en la siguiente tabla 3.2.

3.3.1. COMPILER-ERROR

Indica que ha ocurrido un error en el momento de la compilación del código, o si la compilación ha tardado más de 30 segundos de duración. Este error, en concreto, es el único del que se puede extraer directamente información de lo que está sucediendo desde la página de entrega del problema, ya que los jueces permiten mostrar el mensaje de error emitido por el compilador.

3.3.2. WRONG-ANSWER

Indica que la salida obtenida del problema tras ejecutar los diferentes casos de prueba no es la esperada. El juez no indica en ningún momento qué caso es el que está fallando ni ninguna otra información. Esto es una gran limitación para la persona que está realizando el problema. La única forma de saber qué está ocurriendo, es preguntar a un administrador o juez qué caso o casos de prueba no producen la salida esperada. Obviamente, esto es una tarea manual por parte de ambas partes, lo que provoca un gran cuello de botella en el proceso.

3.4. DOMjudge como juez de programación y su API

Este proyecto se centra en DOMjudge [6] como juez de programación concreto. Utilizando las herramientas y posibilidades que proporciona su API [5], se intenta dar soporte y una retroalimentación clara de los errores ocurridos cuando la salida del problema es WRONG-ANSWER.

Una **API**, en inglés *Application Programming Interface*, es un conjunto de definiciones y protocolos que sirven para integrar y desarrollar el software de las aplicaciones. La finalidad es poder unir aplicaciones o servicios sin necesidad de

tener conocimiento de cómo están implementados. Esto ayuda a ahorrar tiempo, dinero y esfuerzo en el desarrollo de aplicaciones. Mediante el uso de la API de DOMjudge, se pueden hacer tanto consultas como inserciones a los problemas, entregas, concursos, etc.

Resultados	Significado
CORRECT	La entrega es correcta. Todos los test han sido pasados sin errores
COMPILER-ERROR	Ha ocurrido un error en tiempo compilación
TIMELIMIT	Se ha excedido el tiempo máximo permitido para la ejecución del problema
RUN-ERROR	Ha ocurrido un error en tiempo de ejecución
NO-OUTPUT	El problema no ha generado una respuesta por la salida estándar
OUTPUT-LIMIT	Se ha excedido el número de salidas permitidas
WRONG-ANSWER	Los test se han pasado con errores. La solución del problema no es correcta
TOO-LATE	La entrega del problema se ha realizado fuera de plazo

Fuente: https://www.domjudge.org/docs/manual/8.0/team.html?#possible-results

Tabla 3.2: Posibles veredictos

DOMjudge [6] requiere de los siguientes componentes y servicios para poder funcionar:

- DOMjudge server (DOMserver), es la unidad central que se encarga de servir la interfaz web y la API.
- Uno o más *judgehosts*. Es el servidor encargado de compilar y evaluar las soluciones a los problemas entregados.
- Una base de datos MySQL o MariaDB, a la que se conecta DOMserver y que sirve para almacenar todos los datos necesarios.
- Estaciones de trabajo. Cada equipo y administrador requiere de una máquina desde la que puedan, mediante un navegador web, conectarse a DOMjudge.

3.5. Gramáticas formales

Las gramáticas formales son mecanismos con los que se pueden definir lenguajes de relativa complejidad como lenguajes de programación (C, C++, Java, JavaScript, ...), lenguajes de marcado (HTML, XML, ...) y lenguajes de especificación (expresiones regulares, por ejemplo). En el desarrollo de DOMfeed acabamos usando un lenguaje de especificación para definir la entrada y la salida de cada caso de prueba, y para poder trabajar con él, hemos definido una gramática formal para el mismo.

En esta sección se presenta una definición de las gramáticas formales, así como un ejemplo de cómo usar una para comprobar si una cadena de caracteres pertenece o no a un lenguaje.

3.5.1. Definición

Antes de definir lo que es una **gramática formal**, conviene explicar otros conceptos previos. Un **alfabeto** no es más que un conjunto finito de caracteres o símbolos, como es el caso del alfabeto español. A partir de un alfabeto, podemos definir un **lenguaje formal**, que es un conjunto de cadenas de caracteres tomados del alfabeto o *strings* [3]. Una **gramática formal** define un conjunto de reglas para formar las cadenas de un lenguaje formal (también llamadas *reglas de producción*). En concreto, las **gramáticas incontextuales** se caracterizan porque sus reglas son independientes del contexto, es decir, que no importa qué símbolos haya a la hora de aplicar una regla (en caso contrario sería una gramática contextual). La mayoría de lenguajes de programación se pueden describir con gramáticas incontextuales.

Las reglas de producción de una gramática incontextual son del tipo:

$$A ::= \alpha$$

Donde A es un símbolo no terminal, y α es una secuencia de símbolos terminales y no terminales (puede ser una secuencia vacía). Todo símbolo no terminal A tiene al menos una regla asociada como la anterior, y al aparecer en el lado derecho de una regla (es decir, en α) se expanden sus reglas asociadas recursivamente. Por otro lado, los símbolos terminales t 'terminan' la producción y se forman usando el alfabeto del lenguaje asociado a la gramática.

3.5.2. Ejemplos

Veamos un ejemplo en el que intentamos definir un lenguaje natural simple basado en el español, donde las oraciones se forman solo con un sujeto seguido de un verbo en infinitivo. Tendríamos entonces una gramática como la siguiente:

```
Oración  ::=  Sujeto ‘ ’ Verbo
Sujeto   ::=  ‘yo’
          |   ‘tú’
          |   ‘él’
Verbo    ::=  ‘correr’
          |   ‘suspender’
          |   ‘nacionalizar’
```

Nótese que la notación `A ::= a | b` es equivalente a tener dos reglas `A ::= a` y `A ::= b`. Teniendo esto en cuenta, el proceso para comprobar, por ejemplo, si la cadena $s =$ "*yo correr*" pertenece al lenguaje usando la gramática incontextual anterior sería el siguiente:

1. Empezamos con la primera regla `Oración ::= Sujeto ' ' Verbo`. Vemos que una oración empieza con un sujeto, así que consultamos las reglas o producciones del tipo `Sujeto ::= ...`

2. Las reglas `Sujeto ::= ‘yo’ | ‘tú’ | ‘él’` definen que la siguiente cadena a leer de *s* tiene ser o bien ’yo’, o bien ’tú’, o bien ’él’. Como *s* empieza por ’yo’, se acepta la cadena para la regla `Sujeto ::= ‘yo’`.

3. Volviendo a la regla original `Oración ::= Sujeto ‘ ’ Verbo`, ahora tendríamos que comprobar si, tras leer el sujeto, ahora en *s* nos encontramos un espacio en blanco ‘ ’. Esto es así y, por tanto, seguimos al siguiente paso.

4. Por último, en la regla `Oración ::= Sujeto ‘ ’ Verbo` nos falta comprobar si *s* acaba con un verbo o no. Por tanto, consultamos las reglas asociadas a `Verbo`.

5. Las reglas `Verbo ::= ‘correr’ | ‘suspender’ | ‘nacionalizar’`, al igual que pasó con `Sujeto`, nos dicen qué cadenas se pueden usar como verbo. En el caso de *s*, ese verbo es ‘correr’, así que se acepta la cadena.

6. Como la cadena *s* ha encajado con toda la parte derecha de la regla `Oración ::= Sujeto ‘ ’ Verbo`, podemos concluir que *s* es una oración y que, por ende, es parte de nuestro lenguaje.

El proceso anterior es el de un algoritmo de *análisis descendente* o *top-down analysis* (ver figura 3.2 para una implementación en Python). No obstante, en nuestro proyecto usamos una librería llamada PEG.js que no usa algoritmos de análisis descendente sino *gramáticas de expresión de análisis sintáctico* o *Parsing Expression Grammars* (ver sección 3.6.9).

3.6. Tecnologías usadas

En este apartado se detallan las tecnologías usadas en la elaboración de DOMfeed, así como las razones que nos llevaron a incluirlas.

3.6.1. JavaScript

JavaScript es un lenguaje de programación interpretado [8], es decir, no necesita de compilación. Es débilmente tipado, dinámico y basado en prototipos. Se ejecuta en el lado del cliente, es decir, en el navegador. Ofrece la posibilidad de interactuar con la aplicación permitiendo grandes mejoras en la interfaz de usuario y sirve para crear páginas web dinámicas.

JavaScript es la base de muchas de las tecnologías que se utilizan en este proyecto y que se detallan en los siguientes puntos de este subapartado.

3.6.2. Node.js

Node.js [12] [22] es un entorno de ejecución de JavaScript enfocado para la programación de servidores web y que, como su nombre indica, está basado en JavaScript. Esto permite ejecutar código JavaScript en el lado del servidor, lo cual

```
'''En todas las funciones aqui definidas salvo
esta_en_el_lenguaje(), se consume una cadena con cada
llamada, o se lanza una excepcion si no se puede consumir.'''

def esta_en_el_lenguaje(cadena):
    try:
        '''Comprobamos que, tras consumir toda la cadena,
        nos quedemos sin nada, pues si no la cadena no
        perteneceria al lenguaje'''
        return Oracion(cadena) == ""
    except ValueError:
        return False

def Oracion(cadena):
    sin_sujeto = Sujeto(cadena)
    sin_espacio = consume(sin_sujeto, " ")
    sin_verbo = Verbo(sin_espacio)
    return sin_verbo

def Sujeto(cadena):
    for suj in ["yo", "tu", "el"]:
        try:
            return consume(cadena, suj)
        except ValueError:
            continue
    else:
        raise ValueError()

def Verbo(cadena):
    for vb in ["comer" , "suspender", "nacionalizar"]:
        try:
            return consume(cadena, vb)
        except ValueError:
            continue
    else:
        raise ValueError()


def consume(cadena, prefijo):
    '''Si 'cadena' empieza por 'prefijo', devuelve 'cadena'
    quitando 'prefijo' del principio. Si no, lanza una
    excepcion.'''
    if cadena.startswith(prefijo):
        return cadena[len(prefijo):]
    else:
        raise ValueError()
```

Figura 3.2: Implementación en Python de un algoritmo de análisis descendente para nuestro lenguaje natural simple

es una gran ventaja, pues ofrece la posibilidad de utilizar la misma tecnología tanto en el front-end como en el back-end.

Node.js ofrece una gran cantidad de librerías que facilitan el desarrollo de aplicaciones, las cuales son fácilmente gestionables mediante el gestor de paquetes de node: *npm*.

Además, Node.js es un lenguaje asíncrono basado en *callbacks* o promesas, permitiendo así atender múltiples conexiones o llamadas simultáneamente. Esto también permite que tareas que puedan tomar cierto tiempo pasen a segundo plano, como el caso de llamadas a la API o a la base de datos y que así no se detenga el flujo de la aplicación.

3.6.3. Express.js

Express.js [7] es un framework para Node.js que permite estructurar y enrutar aplicaciones web y crear APIs. Además también ofrece otras funciones como la gestión de sesiones y cookies. Debido a su sencillez y a que está programado en Node, se adapta perfectamente a todos los requisitos ofreciendo gran flexibilidad y rendimiento.

Además Express.js ofrece gran sencillez a la hora de desarrollar cualquier tipo de APIs, pues con una simple línea de código, se puede poner en funcionamiento un servidor para que reciba peticiones de clientes. Así mismo, ofrece multitud de métodos para crear y manejar correctamente las rutas y peticiones al servidor. En la figura 3.3 se puede ver un claro ejemplo de la simplicidad de Express.js, donde se está creando un servidor en pocas líneas de código.

Para el tratamiento de las sesiones, cookies, permisos, y en definitiva, el análisis de las peticiones y la gestión de errores, se usan los *middlewares*. Un **middleware** es una función que se sitúa entre la solicitud inicial y la ruta final a la que se pretende llegar y que puede ejecutar cualquier código, cambiar la petición inicial, modificar la respuesta o terminar la petición impidiendo que llegue a su ruta final.

3.6.4. React

React[20] es una biblioteca de JavaScript que se utiliza para facilitar la creación de interfaces de usuario en el *front-end* (o navegador web). Es un lenguaje declarativo basado en componentes reutilizables que se renderizan y actualizan automáticamente cuando ocurre un cambio en la información que muestra la aplicación.

Además de resultar cómodo para el desarrollo de la aplicación, React es una tecnología ampliamente utilizada en la actualidad, lo cual ayuda a maximizar la mantenibilidad del sistema.

3.6.5. React Router

React Router [19] es una librería de enrutamiento en JavaScript basada en componentes para React. Esta librería permite crear un enrutamiento dinámico en

```
1  var express = require('express')
2  var app = express()
3
4  // middleware que muestra un mensaje por pantalla cada vez que
      llega una peticion al servidor enviando una respuesta y
      permitiendo que la llamada continue
5  app.use(function (req, res, next) {
6    console.log('Ha llegado una peticion')
7    next()
8  })
9
10 // Se crea un routa /hello que devuelve 'hello world' al cliente
11 app.get('/hello', function (req, res) {
12   res.send('hello world')
13 })
14
15 // middleware que maneja los posibles errores que se hayan
      producido y devuelve un mensaje de error
16 app.use(function (err, req, res, next) {
17   console.error(err.stack)
18   res.status(500).send('Algo ha salido mal')
19 })
20
21 // Se pone en marcha un servidor que escucha por el puerto 3000
22 app.listen(3000)
```

Figura 3.3: Ejemplo de servidor en Express.js

la parte del servidor, realizar redirecciones, pasar parámetros en las rutas, tener acceso al historial de navegación y manejar las páginas de errores.

En esencia, esta tecnología permite cargar unos componentes u otros dinámicamente dependiendo de la ruta de la aplicación en la que estemos.

3.6.6. Bootstrap

Bootstrap [2] es un framework CSS que ofrece diversas funcionalidades y plantillas para el diseño del front-end de aplicaciones web. El framework combina tanto CSS como JavaScript para dar estilo a la web.

El principal objetivo de Bootstrap es crear una interfaz web adaptable (*responsive*), es decir, que la web se adapte dependiendo del dispositivo y tamaño de pantalla desde donde se esté accediendo. Así mismo, también contiene una gran cantidad de componentes ya creados como alertas, barras de progreso y menús de navegación, entre muchos otros. También añade una gran biblioteca de cientos de iconos para poder utilizar en la web muy fácilmente.

La decisión de utilizar Bootstrap para el diseño del proyecto se debe a la intención de replicar el diseño de la aplicación de DOMjudge, la cual está desarrollada con Bootstrap.

3.6.7. React Bootstrap

React Bootstrap [18] reemplaza completamente la parte de JavaScript de Bootstrap eliminando JQuery, creando componentes totalmente optimizados para React y mucho mas fiables. En la figura 3.4 se puede ver un ejemplo.

Como se puede ver en la figura, al utilizar React Bootstrap en vez de Bootstrap, se obtiene un código mucho más declarativo y fácil de escribir y leer. En vez de tener que utilizar clases para definir el estilo de un componente '*div*' en Bootstrap, en React Bootstrap se inserta un componente, el cual internamente ya tiene definido todo lo necesario para su correcto funcionamiento: clases, estilos, código JavaScript, lógica funcional, etc. Lo único que hay que definir, en este caso, al usar React Bootstrap, es el componente que se va a usar, *Alert*, que es de tipo *danger* y que además puede cerrarse añadiendo el atributo *dismissible*. También hay que especificar que se quiere una cabecera, *Alert.Heading*, y el cuerpo de la alerta, *p*.

React Bootstrap proporciona un código mucho más limpio y legible frente a Bootstrap. Además, si nos fijamos, nos estamos ahorrando la creación del botón porque el atributo *dismissible* lo inserta internamente, sin contar todas las clases que no se añaden porque ya se incluyen por defecto al crear el componente *Alert*.

3.6.8. MySQL

MySQL es un sistema utilizado para la administración y gestión de bases de datos relacionales.

Un sistema gestor de bases de datos relacional es una aplicación que se encarga de almacenar y ofrecer fácil y rápido acceso a un conjunto de datos de

```
1
2  // BOOTSTRAP
3
4  import React from 'react';
5
6  function Example() {
7     return (
8         <div class="alert alert-danger alert-dismissible
9                    fade show" role="alert">
10            <strong>Oh snap! You got an error!</strong>
11            <p>
12                Change this and that and try again.
13            </p>
14            <button type="button" class="close"
15                    data-dismiss="alert" aria-label="Close">
16                <span aria-hidden="true">&times;</span>
17            </button>
18        </div>
19    )
20 }
21
22 // REACT-BOOTSTRAP
23
24 import React, { Component } from 'react';
25 import Alert from 'react-bootstrap/Alert';
26
27 function Example() {
28    return (
29        <Alert dismissible variant="danger">
30            <Alert.Heading>
31                Oh snap! You got an error!
32            </Alert.Heading>
33            <p>
34                Change this and that and try again.
35            </p>
36        </Alert>
37    )
38 }
```

Figura 3.4: Comparativa de Bootstrap con React-Bootstrap

```
start
  = additive

additive
  = left:multiplicative "+" right:additive { return left + right; }
  / multiplicative

multiplicative
  = left:primary "*" right:multiplicative { return left * right; }
  / primary

primary
  = integer
  / "(" additive:additive ")" { return additive; }

integer "integer"
  = digits:[0-9]+ { return parseInt(digits.join(""), 10); }
```

Figura 3.5: Gramática para expresiones aritméticas para PEG.js

manera estructurada y ordenada, agrupándolos mediante tablas y relacionándolos entre sí.

Para nuestro proyecto esta tecnología resulta de gran utilidad a la hora de almacenar información que no podemos obtener a partir de la API de DOMjudge, como las especificaciones de los problemas o el número de revisiones asociado a cada problema.

3.6.9. PEG.js

PEG.js es una librería desarrollada en JavaScript para generar analizadores sintácticos o *parsers* [16]. Es especialmente adecuada para procesar lenguajes de programación e incluso acabar construyendo intérpretes o compiladores. Además, está basada en gramáticas de expresión de análisis sintáctico (en inglés *Parsing Expression Grammar*, y de ahí las siglas PEG) en vez de gramáticas $LL(k)$ o $LR(k)$.

Al descargarnos la librería podemos instalar también un comando `pegjs` con el que generar analizadores sintácticos o *parsers* a partir de ficheros como el de la figura 3.5, el cual es un ejemplo proveniente de la documentación de PEG.js [15]. En este fichero se describe una gramática para reconocer expresiones aritméticas de números enteros que soporta el reconocimiento de sumas, multiplicaciones y paréntesis.

Como podemos observar, la estructura de estos ficheros sigue una estructura muy similar a la usada normalmente para definir gramáticas, lo cual facilita enormemente el uso de la librería. No obstante, PEG.js también nos permite realizar cálculos o procesamientos con las cadenas de caracteres que reconoce cada regla usando JavaScript. De hecho, en el fichero de la figura 3.5, la regla `integer` hace lo siguiente:

1. Reconoce una cadena de caracteres con solo dígitos (`[0-9]+`).
2. Asigna esa cadena a una variable `digits`.
3. Usa esa variable en un fragmento de código en JavaScript para convertir la cadena a un número entero. Nótese que la variable `digits` contiene un array de caracteres en vez de un `string` de JavaScript, y por eso necesitamos usar el método `join()` para convertirlo a un `string`.

El resto de reglas funcionan de forma parecida, pero se usan para calcular el valor de la expresión aritmética partiendo de los enteros reconocidos por la regla `integer`. Por ello, al terminar de procesar una expresión aritmética reconocible por la gramática que define este fichero, el analizador sintáctico devuelve el valor de dicha expresión. Para más detalles, véase la documentación [15].

3.6.10. Jest

Jest [9] es una librería implementada en JavaScript para ejecutar tests que funciona con Node.js, entre otras tecnologías. El principal atractivo de esta librería es que funciona sin tener que configurar nada y que incluye muchas funcionalidades en un solo paquete. Entre ellas se encuentran: tests, *mocking* de objetos, análisis de cobertura del código (*code coverage*), y mensajes muy informativos en caso de que un test falle (véase la figura 3.6, donde al intentar sumar 1 y 2 se espera un 3 pero se obtiene un 4).

Al ser una librería tan completa hemos decidido usarla con algunas partes de nuestro proyecto.

```
❯ npm test

> node@1.0.0 test
> jest

 FAIL  ./sum.test.js
  × adds 1 + 2 to equal 3 (4 ms)

  ● adds 1 + 2 to equal 3

    expect(received).toBe(expected) // Object.is equality

    Expected: 3
    Received: 4

      2 |
      3 | test('adds 1 + 2 to equal 3', () => {
    > 4 |   expect(sum(1, 2)).toBe(3);
        |                     ^
      5 | });
      6 |

      at Object.toBe (sum.test.js:4:21)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.347 s, estimated 1 s
Ran all test suites.
```

Figura 3.6: Ejecución de Jest para una implementación no válida de una función de suma. El comando `npm test` llama internamente a Jest en este proyecto.

Capítulo 4

Organización del sistema

El sistema está dividido en dos partes. Por un lado, se encuentra la aplicación web desarrollada *DOMfeed* y, por otro lado, el juez de programación *DOMjudge*. DOMfeed a su vez está compuesto por una aplicación web con la que interactúa el usuario, un servidor, una base de datos relacional y un analizador (o *parser*) de casos de prueba. En la figura 4.1 se muestra un diagrama explicativo de cómo el sistema está estructurado. A continuación, se detallan cada una de las partes y sus funciones principales de forma específica.

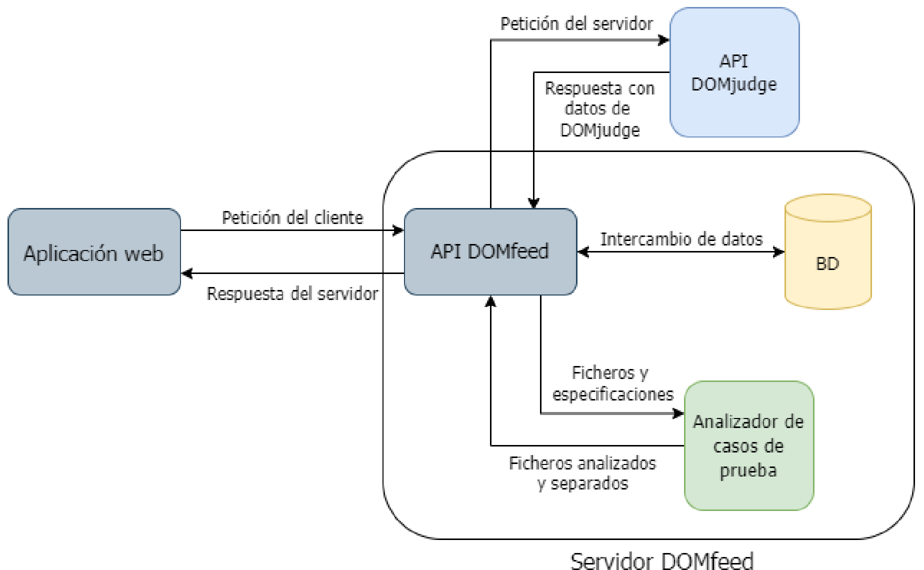


Figura 4.1: Diagrama explicativo con los diferentes componentes del sistema

4.1. DOMjudge

DOMjudge es la base principal de DOMfeed. Todas las funcionalidades desarrolladas en DOMfeed dependen directamente de llamadas a la API de DOMjudge [5]. La API de DOMjudge es limitada, pues no provee de todas las llamadas que

se necesitan para que funcione DOMfeed. Por ello, hemos tenido que recurrir al *scraping* a la hora de descargar los ficheros de prueba y salidas del usuario

El *scraping* es una técnica que sirve para extraer información de páginas web de manera automatizada. Existen ciertas herramientas que facilitan el uso del *scraping*, pero en este caso, ya que su uso es puntual, no se ha utilizado ninguna. Más adelante, se detalla su uso en el proyecto (ver sección 6.2.3).

4.2. Aplicación web o *frontend*

El *frontend* es la parte de la aplicación con la que interactúa el usuario. Es la encargada de mostrar la información que se solicite. La aplicación está dividida en dos partes independientes según el rol del usuario:

- Una parte de administración, a la que accede el profesor (de ahora en adelante el 'administrador'). Este tiene acceso a la configuración de todos los concursos.
- Otra parte para el alumno (de ahora en adelante el 'usuario'), donde este puede consultar las entregas de los problemas del equipo al que pertenezca.

El *frontend* está programado en su totalidad en React [20]. Debido a que la aplicación requiere de diferentes páginas para los distintos componentes o funcionalidades, hemos utilizado React Router [19] para facilitar el enrutado.

La aplicación web requiere de múltiples peticiones al servidor para obtener todos los datos necesarios. Para ello, decidimos utilizar la librería Axios [1], la cual provee de grandes ayudas a la hora de hacer peticiones al servidor.

En cuanto a la maquetación y CSS, hemos utilizado React Bootstrap [18], el cual reemplaza a Bootstrap [2] como framework. A diferencia de Bootstrap, React Bootstrap está completamente optimizado para React, permitiendo así mejoras en rendimiento y en velocidad de desarrollo.

4.2.1. Vistas del administrador

Las pantallas del administrador se basan en la gestión y control de DOMfeed. El administrador tiene privilegios totales de visualización, pudiendo acceder a todas las entregas o *submissions* de cualquier equipo.

La tarea principal del administrador es la de suministrar, para cada uno de los problemas que hay registrados en DOMjudge, una especificación válida y un número máximo de veces que un equipo puede ver la solución a una entrega para un problema determinado y concursos. En la sección 4.2.3 se explica detenidamente en qué consiste esta parte.

4.2.2. Vistas del usuario

El rol principal del usuario en DOMfeed es visualizar, para cualquier problema donde DOMjudge haya mostrado un veredicto de WRONG-ANSWER, la salida

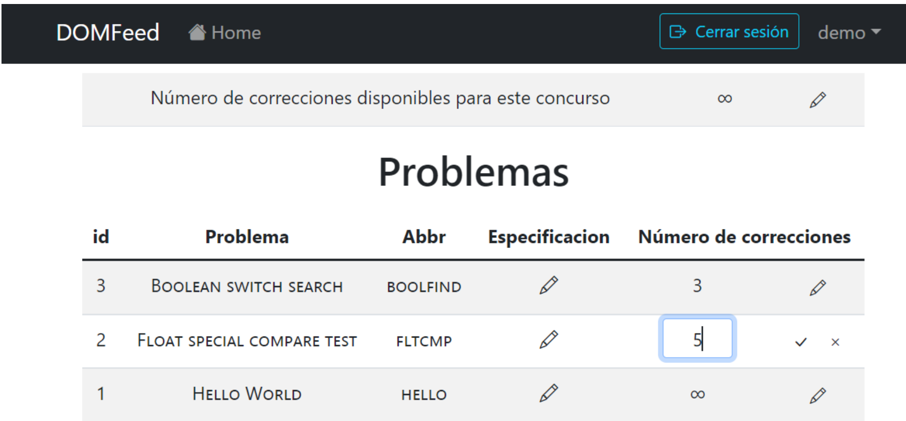


Figura 4.2: Vista de administrador

esperada frente a la salida obtenida de ejecutar la solución proporcionada por el usuario. Para ello hay que solicitar al servidor esos datos.

De esta forma, una vez hecha la petición, se mostrará una pantalla explicativa donde se pueden ver los casos de entrada, la salida esperada y la salida obtenida por la solución del usuario. El caso de prueba en el que difiere la salida esperada aparece de color rojo, para indicar al usuario donde está el error en su solución del problema.

Un usuario solamente puede ver los concursos a los que pertenezca, así como las entregas que haya hecho él mismo o su equipo. Además, solo podrá ver las soluciones en el caso de que tenga correcciones restantes o cuando ya ha solicitado la ayuda para dicha entrega previamente. Este punto se explicará con más detalle en la sección 6.1.4.1.

DOMFeed Home Cerrar sesión demo

Entregas

id	Hora	Problema	Lenguaje	Resultado
6	02-05 / 21:49	BOOLFIND	CPP	WA
5	14-03 / 19:36	FLTCMP	CPP	WA
4	14-03 / 19:36	BOOLFIND	CPP	WA
3	14-03 / 19:36	HELLO	CPP	WA
2	20-02 / 23:29	HELLO	CPP	WA

Figura 4.3: Vista de usuario

4.2.3. Máximo de correcciones

Hagamos un breve inciso para explicar en qué consiste la parte de los intentos o correcciones para un determinado problema o concurso. Debido a que un usuario puede solicitar la corrección para un problema, donde se mostrará el primer error obtenido en su salida de ejecución con respecto a la salida esperada, es importante que un profesor pueda limitar el número de peticiones que se puedan realizar por parte del usuario. Si un usuario pudiese solicitar casos de prueba de modo ilimitado, en algún momento podría conseguir todo el conjunto de casos de prueba para dicho problema. Al limitar la cantidad de peticiones que se puede hacer, eliminamos esta posibilidad.

La manera en la que funciona la aplicación web es la siguiente. Cuando un usuario solicita retroalimentación detallada para un envío fallido, esta solicitud queda registrada en la base de datos. Al solicitar la corrección, se aumenta en uno la cantidad de correcciones gastadas que el usuario o grupo ha utilizado en dicho problema y también se aumenta el contador de correcciones usadas en el concurso correspondiente. En el caso de que el grupo haya gastado todas las correcciones en un problema determinado, no se le dejará volver a ver ninguna nueva corrección para dicho problema, a no ser que la solicitud corresponda a una entrega cuya solicitud se haya solicitado previamente. En ese caso, siempre se podrá volver a visualizar la corrección sin necesidad de incrementar el número de solicitudes realizadas. Lo mismo pasa cuando un usuario llega al número máximo de correcciones para problemas que pertenecen a un concurso. En este caso, el usuario no podrá ver ninguna de las correcciones correspondientes a los problemas que pertenecen a dicho concurso, a no ser que la entrega ya haya sido canjeada anteriormente.

4.3. Servidor o *backend*

El servidor es la parte de la aplicación que se encarga de recibir las peticiones del usuario, procesar los datos necesarios obtenidos tanto de la base de datos como de la API de DOMjudge y, posteriormente, servirlos al *frontend* para su visualización. El *backend* se compone de cuatro módulos diferenciados:

1. Gestión de las peticiones del usuario y comprobación de permisos.
2. Obtención de datos de la API de DOMjudge.
3. Obtención de datos de la base de datos de DOMfeed.
4. Procesamiento, transformación y formateado de los datos obtenidos para enviarlos al *frontend* en el formato correcto.

4.3.1. npm

Todo el servidor está construido sobre Node.js [12]. Node provee de un gestor de paquetes de código libre llamado *npm* [13]. La configuración tanto de Node.js

como de npm se puede encontrar en un fichero localizado en la raíz del proyecto que se llama **Package.json** [14].

Existen muchas opciones de configuración del fichero *Package.json*, todas ellas importantes, tanto para describir el proyecto, como para indicar qué personas han participado en él, definir carpetas a usar, etc. En el siguiente listado, vamos a explicar cuáles son las propiedades más importantes de este archivo de configuración.

- *name*, indica el nombre del proyecto.
- *version*, indica la versión del proyecto.
- *license*, sirve para especificar la licencia de la que dispone el proyecto.
- *scripts*, es un diccionario utilizado para ejecutar scripts predefinidos.
- *config*, sirve para indicar parámetros de configuración específicos.
- *dependencies*, es el lugar donde se indican las dependencias del proyecto.
- *devDependencies*, es donde se indican las dependencias de desarrollo del proyecto.

npm se utiliza para instalar paquetes o dependencias en un proyecto de Node.js. El comando `npm install nombre-de-la-dependencia` proporciona esta funcionalidad. El comando, aparte de instalar el paquete correspondiente, agrega el nombre de la dependencia en la sección *dependencies* del fichero *Package.json*.

4.3.2. Peticiones del usuario

La principal tarea de un servidor es recibir peticiones de un usuario para posteriormente entregarle una respuesta. Para ello, hemos utilizado Express.js [7] para poder crear una API, y así las rutas necesarias y las comprobaciones de seguridad oportunas.

Lo más destacable de esta parte del *backend*, es el uso de un *middleware* situado antes de todas las rutas. Este *middleware* se encarga de extraer un token en formato JWT, *JSON Web Token*, y añadirlo al objeto generado en la petición. Posteriormente, este token se lee mediante otro *middleware* antes de cada petición, siempre que sea necesario, para comprobar datos acerca de la petición: quién la ha realizado y si tiene permisos para obtener los datos solicitados.

4.3.3. Datos de DOMjudge

Para realizar las peticiones a la API de DOMjudge [5], estamos utilizando la librería Axios [1]. Mediante una petición GET a la ruta correspondiente de la API de DOMjudge, obtenemos los datos requeridos. Estos datos, posteriormente los tratamos (ver sección 4.3.5) y los enviamos al cliente. Si bien es cierto que esta parte es trivial, hay una situación donde la API de DOMjudge no es suficiente y hemos tenido que recurrir al *scraping*.

La manera en la que un usuario inicia sesión en DOMfeed es igual a la que haría en DOMjudge. La idea que siempre hemos tenido a la hora de desarrollar DOMfeed es que se pareciera en todo lo posible a DOMjudge. Es por eso, que todo usuario de DOMjudge debe poder iniciar sesión en DOMfeed con los mismos credenciales.

El momento en el ciclo de DOMfeed en el que es necesario hacer scraping es cuando necesitamos obtener tanto los ficheros de entrada y salida esperados, es decir, los casos de prueba, como los resultados obtenidos al ejecutar la solución del usuario a un problema. La API de DOMjudge no proporciona esta información.

4.3.4. Datos de la base de datos

Además de los datos obtenidos mediante la API de DOMjudge, hay más información necesaria para el funcionamiento de la aplicación que no se puede obtener mediante esta. Esto se debe a que dicha información no es propia de DOMjudge, sino de nuestra aplicación, por ejemplo, las especificaciones de los problemas. Es por ello que generamos y almacenamos esa información en una base de datos SQL.

En función de la petición del cliente y de los permisos del mismo, se genera la solicitud a la base de datos, insertando, actualizando u obteniendo los datos solicitados y tratándolos en función del tipo de solicitud.

4.3.5. Procesamiento de los datos

Todos los datos que se obtienen en la aplicación pasan por un proceso de tratamiento. Mediante este proceso, los datos de filtran, se transforman y se formatean para adaptarse a los requisitos del usuario.

Supongamos un ejemplo: al obtener los datos de DOMjudge obtenemos un listado con todas las entregas pertenecientes a todos los grupos, pero el usuario que realiza la petición pertenece al grupo 4, por lo que solo puede ver las entregas pertenecientes a dicho grupo. Además, el listado no contiene el nombre del problema al que pertenece una entrega concreta, si no que aparece el identificador en su lugar. El procesamiento de datos en este caso concreto consiste en filtrar las entregas, quedándonos solamente con las que pertenezcan al equipo 4 y también reemplazar el identificador del problema por el nombre del problema que corresponda. Una vez hecho todo esto, la petición del usuario estaría disponible para poder enviarla al cliente.

4.4. Base de datos

Como ya se ha mencionado en puntos anteriores, DOMfeed almacena información en una base de datos relacional, la cual está alojada en un servidor *SQL* cuya estructura se explicará detalladamente en la sección 6.2.1.

Si bien es cierto que la mayor parte de la información de la aplicación se obtiene de DOMjudge, la necesidad de una base de datos propia surge al querer imple-

mentar ciertas funcionalidades, como la administración del número de correcciones para un problema o la especificación de los casos de prueba.

El acceso a la base de datos se hace mediante la librería *mysql2* de Node. Mediante esta librería se crea un *pool* de conexiones mediante el cual se puede hacer peticiones de forma segura a la base de datos. Un ***pool* de conexiones** es un conjunto de conexiones limitado que son gestionadas por un servidor, el cual asigna las conexiones de acuerdo a las peticiones de los clientes, de modo que se puedan realizar varias conexiones de manera simultánea. De este modo, se mantiene siempre un número fijo de conexiones abiertas y no se tiene que abrir una nueva conexión por cada petición de los usuarios, lo cual es costoso.

4.5. Analizador de casos de prueba

Como mencionamos en la introducción, para que el alumno pueda saber en qué casos de prueba ha fallado su solución a un problema hipotético de programación, antes necesitamos una manera de separar los casos de prueba. Por tanto, hemos creado un analizador de casos para dicho fin.

El analizador de casos de prueba es una biblioteca escrita en su mayoría en JavaScript, aunque también cuenta con ficheros para crear analizadores sintácticos usando el lenguaje de definición de gramáticas de PEG.js [16]. Por otro lado, tiene un *script* desarrollado en Python para crear ficheros de casos de prueba de tamaños arbitrarios a modo de pruebas, para poner a prueba el consumo de memoria y el tiempo de ejecución de la librería.

El analizador de casos de prueba tiene dos funciones principales:

1. Comprobar la validez de una especificación que describe cómo separar los casos de prueba de un fichero.

2. Separar tanto los casos de prueba de un fichero de entrada como los de un fichero de salida usando una especificación para la entrada y otra para la salida.

4.5.1. Comprobación de la validez de una especificación

Las especificaciones que describen un fichero de casos de prueba usan un lenguaje propio inspirado en las expresiones regulares. Este se detalla más adelante en el capítulo 5.

Para comprobar la validez de una especificación dada por un profesor, nuestra biblioteca dispone de una función para no solo comprobar si una especificación es válida o no, sino para también comunicar qué errores hay en caso de que no lo sea. Estos errores luego se muestran en la aplicación web para proporcionar *feedback* al usuario.

Este componente de la librería está desarrollado mayoritariamente en JavaScript. No obstante, para comprobar la validez de una especificación se usa una gramática escrita en el lenguaje que describe la documentación de PEG.js [15] para generar un analizador sintáctico.

4.5.2. Separación de casos de prueba

La biblioteca del analizador de casos tiene también una función para separar los casos de prueba de un fichero de entrada y otro de salida usando especificaciones para ambos ficheros. Estas especificaciones se escriben con el lenguaje que se describirá en la sección 5.1.

El profesor es el encargado de proporcionar dichas especificaciones mediante la pantalla correspondiente en la página web de DOMfeed (ver figura 6.4). Una vez introducidas las especificaciones y tras haberlas validado, se generan asíncronamente los nuevos ficheros de entrada y salida correspondientes al problema en cuestión con los casos de prueba separados. Estos ficheros se guardarán en una subcarpeta que se encuentra en la misma ruta que los ficheros originales para poder comparar posteriormente con la salida de la solución proporcionada por el alumno.

Adicionalmente, hay otra función similar a la anterior encargada únicamente de separar los casos de prueba de entrada. El objetivo principal de la misma es ofrecer en la vista del administrador una previsualización de los casos de prueba de entrada en tiempo real mientras no se haya introducido ninguna especificación de salida.

Capítulo 5

Analizador de casos de prueba

Como comentamos en el anterior capítulo, el analizador de casos de prueba es una biblioteca que cuenta con tres funciones:

- Comprobar la validez de una especificación.
- Separar los casos de prueba de entrada y salida a partir de una especificación para la entrada y otra para la salida.
- Separar solo los casos de prueba de entrada dada una especificación.

En este capítulo profundizaremos en estas funcionalidades. Empezaremos describiendo el lenguaje usado para las especificaciones y cómo se verifica que una especificación dada es válida y, finalmente, describiremos el proceso de separación de los casos de prueba. Además, también explicaremos parte del código del analizador de casos.

5.1. Lenguaje de especificación

Para poder separar los ficheros con los casos de prueba de un problema necesitamos una forma de describir esos ficheros y, especialmente, determinar dónde empiezan y acaban cada uno de esos casos. Por ello hemos desarrollado un lenguaje propio para ese mismo fin que, al poder ser usado para describir ficheros, también servirá para realizar una validación básica de los mismos.

En esta sección empezaremos mencionando los requisitos que tenía nuestro lenguaje de especificación y las razones por las que incluimos ciertas características. Posteriormente, describiremos dicho lenguaje y su gramática correspondiente y aportaremos algunos ejemplos. Por último, mencionaremos cómo puede un usuario de nuestra biblioteca verificar la validez de una especificación.

5.1.1. Requisitos del lenguaje

La razón fundamental por la que necesitamos el lenguaje de especificación es para separar los casos de prueba de un fichero. Es por ello que debe tener una manera de especificar dónde empieza y donde acaba cada uno de ellos.

Información previa
Caso de prueba 1
Caso de prueba 2
...
Caso de prueba n
Fin de fichero

Figura 5.1: Esquema de un fichero de casos de prueba de entrada

Información previa
<*separador*>
Caso de prueba 1
<*separador*>
Caso de prueba 2
<*separador*>
...
<*separador*>
Caso de prueba n
<*separador*>
Fin de fichero

Figura 5.2: Esquema de un fichero de casos de prueba de entrada con separadores

Además, debemos tener en cuenta que los ficheros de muchos problemas tienen algo de información adicional al principio o al final del propio fichero que no pertenece a ningún caso de prueba, como muestra el esquema de la figura 5.1. La información presente al principio del fichero suele indicar el número de casos de prueba que contiene, mientras que la parte final se suele usar para indicar que no hay más casos de prueba.

Por consiguiente, hemos considerado que un caso de prueba debe tener siempre un separador antes y otro después, de modo que la información del principio del fichero solo tendrá un separador después, y la del fin del fichero tendrá el separador antes de la misma (véase la figura 5.2). De esta forma podemos siempre distinguir qué parte de un fichero es un caso de prueba o no.

Hasta el momento hemos visto qué criterio aplicar a la hora de poner separadores. No obstante, necesitamos también especificar qué estructura tienen los casos de prueba y la información del principio y del final del fichero.

Para ello primero necesitamos introducir el concepto de *token* o *componente/token léxico*. Un **token** es una cadena de caracteres al que se le asigna un significado. Por ejemplo, la cadena '1234' se puede tratar como un número entero y podríamos llamarlo un *token entero*. Por otro lado, la cadena 'x' podría ser tratada como el nombre de una variable o de una función en un lenguaje de programación y podríamos denominarlo como un *token identificador*. El concepto de token es fundamental para poder describir la estructura de los casos de prueba de un fichero, por lo cual decidimos incorporarlo a nuestro lenguaje.

De hecho, muchos problemas de programación trabajan con entradas formadas

únicamente por números enteros. A raíz de esto, era indispensable incluir un token de tipo número entero a nuestro lenguaje de especificación. No obstante, los problemas de programación también pueden esperar cadenas de caracteres o *strings*, por lo que decidimos también incluir un token *string*. Este token serviría para representar cadenas de caracteres delimitadas por caracteres de espacio, tabuladores o fin de línea. Además, en algunos problemas parte de esas cadenas son siempre las mismas, por lo que convenía que nuestro lenguaje de especificación soportase *literales*. Un **literal** es un valor constante, como puede ser `'1'`, `'abc'` o `'hello world'` (nótese que el último ejemplo incluye un espacio, por lo que equivaldría a dos tokens *string*). Por último, decidimos también incluir un token para representar líneas de texto y otro para representar el fin de fichero al que llamaremos *EOF* (del inglés *End Of File*).

Nuestro lenguaje además necesitaba poder reconocer expresiones formadas por secuencias de tokens separadas por espacios, pero también disyunciones de estas mismas expresiones. Por ejemplo, en algunos problemas el alumno tiene que escribir en la salida o bien un *string* o bien otro (por ejemplo, 'SI' o 'NO'), así que decidimos incluir disyunciones en el lenguaje.

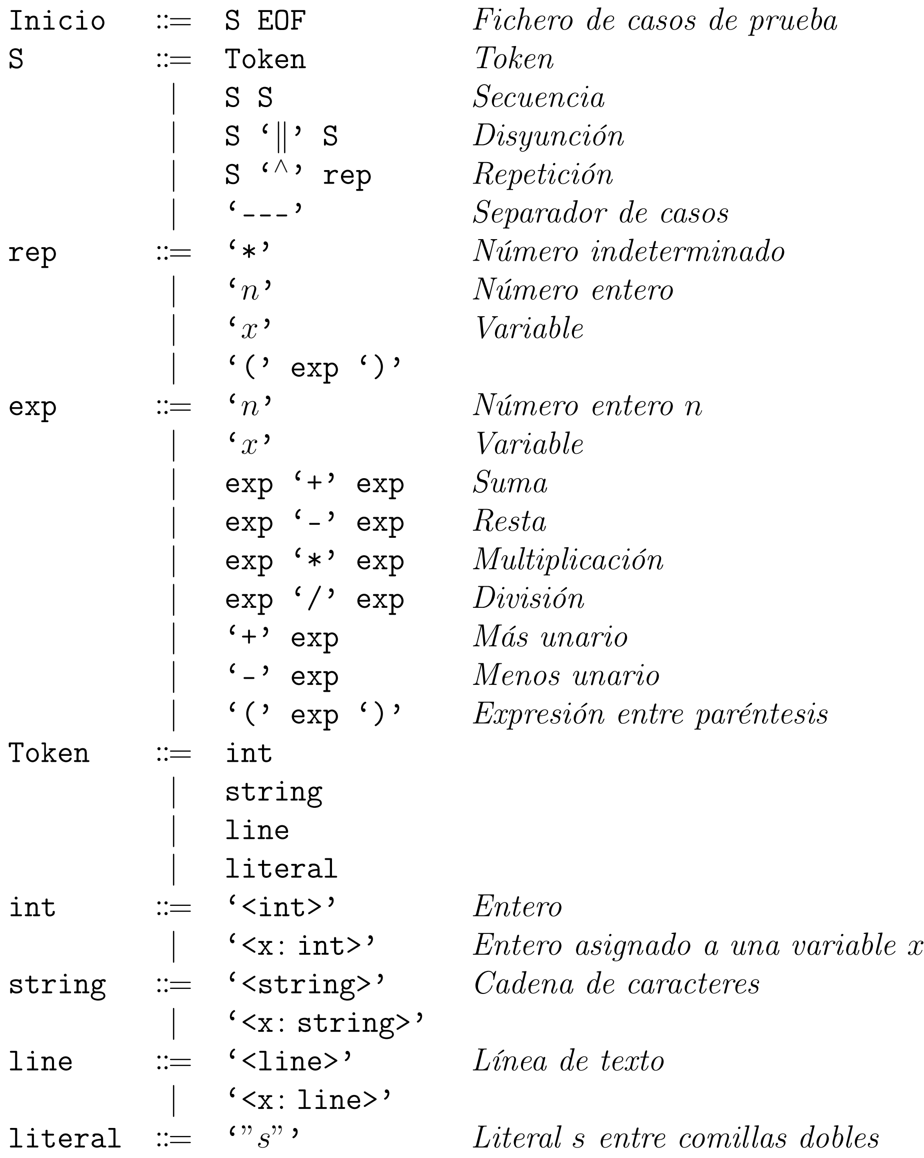
Por último, incluimos la posibilidad de que una expresión se repita, o bien un número indeterminado de veces, o bien una cantidad constante. De hecho, como en la información del principio de un fichero está presente en muchas ocasiones el número de casos de prueba, decidimos añadir la posibilidad de que el número de repeticiones dependa de un token entero anterior, de modo que una especificación pueda indicar que el primer token es un entero n, y que a continuación hay n casos de prueba.

5.1.2. Descripción del lenguaje

La gramática de la figura 5.3 puede usarse como documentación del lenguaje de especificación e incluye todas las características que comentamos en la sección anterior. No obstante, es **ambigua**, es decir, se pueden usar sus reglas de varias formas distintas para describir una misma especificación, lo cual complica enormemente su uso a la hora de implementar un analizador o *parser* del lenguaje. Por tanto, hemos definido la gramática de la figura 5.4, la cual no es ambigua y describe la prioridad a la hora de aplicar unas reglas sobre otras, algo de lo que carecía la gramática de la figura 5.3.

Por ejemplo, veamos la especificación <int>^(2 * 4 + 3). La gramática ambigua (figura 5.3) no sería capaz de discernir si la expresión `2 * 4 + 3` se procesaría como `(2 * 4) + 3` o `2 * (4 + 3)`, es decir, no sabe qué operación es más prioritaria entre la multiplicación y la suma. Esto es así porque las reglas `expr ::= expr + expr` y `expr ::= expr * expr` pueden usarse en cualquier orden. No obstante, la gramática no ambigua (figura 5.4) no presenta tal problema, pues la regla `A1 ::= A2 * A1` siempre se tiene que aplicar después de la regla `A0 ::= A1 + A0`.

Sin embargo, la gramática no ambigua es significativamente más sencilla, por lo que sigue siendo útil para documentar las características del lenguaje.



```
Inicio   ::= S EOF             Fichero de casos de prueba
S        ::= Token             Token
           | S S               Secuencia
           | S '‖' S           Disyunción
           | S '^' rep         Repetición
           | '---'             Separador de casos
rep      ::= '*'               Número indeterminado
           | 'n'               Número entero
           | 'x'               Variable
           | '(' exp ')'
exp      ::= 'n'               Número entero n
           | 'x'               Variable
           | exp '+' exp       Suma
           | exp '-' exp       Resta
           | exp '*' exp       Multiplicación
           | exp '/' exp       División
           | '+' exp           Más unario
           | '-' exp           Menos unario
           | '(' exp ')'       Expresión entre paréntesis
Token    ::= int
           | string
           | line
           | literal
int      ::= '<int>'           Entero
           | '<x: int>'        Entero asignado a una variable x
string   ::= '<string>'        Cadena de caracteres
           | '<x: string>'
line     ::= '<line>'          Línea de texto
           | '<x: line>'
literal  ::= '"s"'             Literal s entre comillas dobles
```

Figura 5.3: Gramática ambigua del lenguaje de especificación

Inicio	::=	Disyunción EOF	*Fichero de casos de prueba*
Disyunción	::=	Secuencia ‘‖’ Disyunción	
	|	Secuencia	
Secuencia	::=	Repetición Secuencia	
	|	Repetición	
Repetición	::=	S ‘^’ rep	
	|	S	
S	::=	Token	
	|	‘(’ Disyunción ‘)’	
	|	‘---’	*Separador de casos*
rep	::=	‘*’	*Número indeterminado*
	|	exp	
exp	::=	A3	*Expresión aritmética*
A0	::=	A1 ‘+’ A0	*Suma*
	|	A1 ‘-’ A0	*Resta*
	|	A1	
A1	::=	A2 ‘*’ A1	*Multiplicación*
	|	A2 ‘/’ A1	*División*
	|	A2	
A2	::=	‘+’ A3	*Más unario*
	|	‘-’ A3	*Menos unario*
	|	A3	
A3	::=	‘n’	*Número entero* n
	|	‘x’	*Variable*
	|	‘(’ A0 ‘)’	*Expresión entre paréntesis*
Token	::=	int	
	|	string	
	|	line	
	|	literal	
int	::=	‘<int>’	*Entero*
	|	‘<x: int>’	*Entero asignado a una variable* x
string	::=	‘<string>’	*Cadena de caracteres*
	|	‘<x: string>’	
line	::=	‘<line>’	*Línea de texto*
	|	‘<x: line>’	
literal	::=	‘"s"’	*Literal* s *entre comillas dobles*

Figura 5.4: Gramática desambiguada del lenguaje de especificación

5.1.3. Ejemplos de especificaciones

Para que el lector se familiarice con el lenguaje de especificaciones descrito en las figuras 5.3 y 5.4, conviene que presentemos algunos ejemplos sencillos. Para ejemplos más avanzados, consulte el apéndice A, el cual contiene especificaciones para problemas reales de programación.

5.1.3.1. Ejemplo 1

Empecemos por un problema que siempre tiene un único caso de prueba. Si dicho caso de prueba consiste de tres números enteros, podríamos usar la siguiente especificación:

```
--- <int> <int> <int> ---
```

La cual se puede abreviar de la siguiente manera:

```
--- <int>^3 ---
```

Aquí `<int>` representa un entero cualquiera, y `---` representa el separador de casos de prueba (lo que en las figuras 5.1 y 5.2 se denomina *<separador>*). Como mencionamos en la sección 5.1.1, un caso de prueba debe tener un separador antes y otro después.

5.1.3.2. Ejemplo 2

Veamos un uso típico de variables. Tenemos un problema donde la entrada empieza con un número n seguido de n casos de prueba, de modo que cada uno consiste en otros dos números enteros. Una especificación válida podría ser la siguiente:

```
<n:int> --- (<int> <int> ---)^n
```

Con el token `<n:int>`, el valor del primer entero de la entrada se guarda para poder referenciarse posteriormente con la variable n. De hecho, se usa este valor n para describir cuántos casos de prueba tiene la entrada, usando `^n`. Además, al igual que en el ejemplo anterior, cada caso de prueba queda delimitado por dos separadores, mientras que `<n:int>` tiene solo uno después al tratarse de información previa a los casos de prueba.

5.1.3.3. Ejemplo 3

Veamos esta vez un problema donde cada caso de prueba de la entrada es, o bien una cadena `"comprar"` seguida de un entero, o bien una cadena `"vender"` seguida también de un entero. Además, supongamos que el número de casos es indeterminado. Entonces podríamos usar la siguiente especificación:

```
--- (("comprar" || "vender") <int> ---)^*
```

`("comprar" || "vender")` indica que puede aparecer, al igual que dice el enunciado, tanto la cadena `'comprar'` como la cadena `'vender'`. Por otro lado, `^*` indica que puede haber 0 o más casos de prueba.

5.1.3.4. Ejemplos problemáticos

Sea un problema de programación donde tenemos un número indeterminado de casos de prueba, de modo que cada uno consiste de dos cadenas de caracteres sin espacios, y delimitamos el fin del fichero con la cadena '-fin fichero-' (es un literal). Una posible especificación sería la siguiente:

```
--- (<string> <string> ---)^* "-fin fichero-"
```

Esta especificación presenta un problema de indeterminismo debido a los requisitos del enunciado. A menos que ningún caso de prueba pudiera tener las cadenas '-fin' y 'fichero-', si al ir leyendo la entrada encontramos el literal '-fin fichero-', no podemos asegurar que hayamos llegado al final del fichero si no seguimos leyéndolo. Esto supone un problema a la hora de procesar un fichero de entrada junto a esta especificación, pues nos fuerza a comprobar si el literal '-fin fichero-' son dos cadenas de un mismo caso de prueba, y en caso contrario tenemos que volver atrás y procesarlo como un literal.

Este problema de indeterminismo no solo existe entre tokens *string* y literales, pues un número como el 0 puede ser aceptado tanto por "0" como <int>, como ocurre en las siguientes especificaciones:

```
--- (<int> ---)^* "0"

--- ("0" ---)^* "0"
```

Además, las siguientes especificaciones siempre presentan los problemas anteriores, independientemente de que los casos de prueba tengan siempre un contenido distinto al del fin de fichero:

```
--- (<string> ---)^* <string>

--- (<int> ---)^* <int>
```

De hecho, podemos generalizar todos los casos anteriores. Dadas las especificaciones R, S y T, la especificación `R S^* T` necesitará el mecanismo de vuelta atrás mencionado anteriormente si existe una cadena x que sea aceptada tanto por la especificación S como por T. Eso sí, en el caso de una especificación `--- (S' ---)^* T`, donde S' es la especificación de un caso de prueba, dejará de suponer un problema si dicha cadena x no aparece nunca como caso de prueba.

Esta vuelta atrás complica enormemente el procesamiento, y como veremos en la sección 5.2, hemos decidido optar por implementar un mecanismo alternativo para resolver el indeterminismo. Si tenemos dos especificaciones S_1 y S_2, S_1 acepta la cadena vacía, y una cadena x es aceptada por ambas especificaciones, a la hora de procesar la especificación S_1S_2 optaremos por descartar la especificación S_1 y proseguir con S_2.

Este mecanismo de procesamiento, no obstante, tiene a su vez el problema de que algunas especificaciones dejan de ser aplicables a algunos ficheros. Por ejemplo, volvamos a considerar la siguiente especificación:

```
--- (<int> ---)^* <int>
```

Debido al mecanismo anterior, esta especificación solo podría describir ficheros con un único número entero, el cual serviría como fin de fichero. Por tanto, la especificación anterior equivaldría a `--- <int>`.

Por otro lado, volvamos a considerar también la siguiente especificación:

```
--- (<int> ---)^* "0"
```

Esta especificación serviría para describir cualquier fichero que consista solo de números enteros siempre y cuando el 0 solo aparezca al final. En cambio, para un fichero cuyo contenido sea `12 0 3 1 0`, el primer 0 se trata como fin de fichero y se ignoran el resto de enteros, lo cual no ocurriría si hubiéramos implementado la vuelta atrás.

5.1.4. API de comprobación de la validez de una especificación: `tryToParseGrammar()`

Para comprobar la validez de una especificación dada por un profesor, nuestra biblioteca dispone de la función `tryToParseGrammar()` (ver figura 5.5), la cual recibe una especificación y devuelve no solo si es válida de acuerdo a la gramática de la figura 5.4, sino también qué errores hay en caso de que no lo sea.

```
1 function tryToParseGrammar(spec) {
2   try {
3     parseSpecification(spec);
4     return { valid: true, errorMessage: undefined };
5   } catch (error) {
6     return { valid: false, errorMessage: error.message };
7   }
8 }
```

Figura 5.5: `tryToParseGrammar()`: Función para comprobar la validez de una especificación

5.2. Separación de los casos de prueba

En esta sección se explica cómo funciona la separación de casos de prueba a alto nivel. Nos centraremos primero en la implementación de la función `separateCases()`, que se encarga de ir separando los casos de prueba de entrada y de salida. Posteriormente, veremos un tipo abstracto de datos llamado **TokenStream** que hemos creado para el tratamiento de los tokens de un fichero, y finalmente describiremos unas reglas usadas para procesar la separación de casos.

No explicaremos la función `separateInputCases()`, pues funciona igual que `separateCases()`, pero separando solo los casos de prueba de entrada.

5.2.1. Implementación de `separateCases()`

En esta sección presentaremos dos funciones:

- `separateCases()`, que es la API que usa el servidor para separar los casos de prueba de entrada y de salida de sus respectivos ficheros.
- `_separateCases()`, que tiene la implementación real de la separación de los casos de prueba y es llamada internamente por `separateCases()`.

5.2.1.1. API de separación de casos: `separateCases()`

Para empezar, los parámetros de entrada de la función `separateCases()` son:

- `inputFilename`: El nombre del fichero con los casos de prueba de entrada.
- `inputSpec`: La especificación que describe cómo separar los casos de prueba de entrada.
- `outputFilename`: El nombre del fichero con los casos de prueba de salida.
- `outputSpec`: La especificación que describe cómo separar los casos de prueba de salida.
- `separator`: Una cadena de caracteres que se vaya a usar para separar los casos de prueba. Es un argumento opcional, por lo que si no se especifica se usa por defecto la cadena `'<==========>'`.

A continuación explicamos el código en sí de `separateCases()` (ver figura 5.6). En primer lugar, al tratarse de una API, se comprueba que todos los argumentos anteriormente descritos sean válidos, y que en el caso de no elegir un separador para los casos de prueba se use el separador por defecto.

Después, en la función auxiliar `createCasesFile()`, que es llamada tanto para la entrada como para la salida, creamos una carpeta nueva donde se guardarán los ficheros con los casos de prueba separados, verificamos la especificación y construimos un **TokenStream** para *tokenizar* o separar en tokens el contenido de un fichero (ver sección 5.2.2 para más información).

Si no ha ocurrido ningún error durante la llamada a `createCasesFile()` (por una especificación inválida, por ejemplo), se crea un diccionario para guardar el estado de las variables definidas en las especificaciones. A este diccionario nos referiremos de ahora en adelante como **estado**. En informática un diccionario es una colección de pares (clave, valor) cuyas claves no se repiten. La clave puede ser de cualquier tipo de datos. En nuestro caso, es de tipo 'string' porque corresponde al nombre de una variable. Por otro lado, los enteros y literales son los únicos tipos de datos soportados en nuestro diccionario. La necesidad de tener un diccionario es debido a que se pueden referenciar variables en las especificaciones, y un diccionario es una manera muy efectiva de guardar y consultar posteriormente su valor.

Por último, tras haber realizado la separación de casos, se comprueba si ha habido algún error en el proceso en la función `checkForErrors()`. Finalmente, se

devuelve la ruta de los ficheros de entrada y salida separados junto con el separador usado.

```
async function separateCases(inputFilename, inputSpec,
    outputFilename, outputSpec, separator = undefined) {
  checkInputs(inputFilename, inputSpec, outputFilename, outputSpec
    , separator);

  if (!separator) {
    const config = await getConfiguration();
    separator = config.default_separator;
  }

  const input = await createCasesFile(inputFilename, inputSpec,
    separator);
  const output = await createCasesFile(outputFilename, outputSpec,
    separator);
  let currentState = new Map();

    // Separamos los casos
  _separateCases(input, output, currentState);
  checkForErrors(input, output, currentState);

  return {
    inputSeparatedCasesFilename: input.separatedCasesFilename,
    outputSeparatedCasesFilename: output.separatedCasesFilename,
    separator
  };
}
```

Figura 5.6: Implementación de la función `separateCases()`

5.2.1.2. Función auxiliar: `_separateCases()`

En esta función (véase el código en la figura 5.7), el TokenStream junto con las especificaciones se irán consumiendo basándose en unas **reglas de procesamiento** en la función `step()`. Estas reglas son descritas más adelante en la sección 5.2.3, pero, a modo de introducción, debemos mencionar que dichas reglas controlan no solo cómo se deben consumir una especificación y el TokenStream al que describe, sino también la inserción de separadores.

Así pues, comenzamos con el grueso de la separación de los casos de prueba, que se trata de un bucle que termina cuando tanto la especificación de entrada como la de salida se consuman o si ha ocurrido un error inesperado. Mientras no se cumplan las condiciones previas, procesamos primero la entrada paso a paso con llamadas a `step()` hasta que:

- Se encuentre un separador. Esto indica el fin del caso de prueba actual y el comienzo del siguiente.

- Se encuentre que la especificación no es válida y, por tanto, no se puede aplicar al fichero de entrada representado por su respectivo TokenStream.
- Se consuma completamente la especificación, es decir, se quede vacía.

Después procesaríamos la salida hasta que se diera cualquiera de las condiciones anteriores. Es decir, hasta que no se termina de separar el primer caso de prueba de salida, no se comienza con el segundo caso de prueba de entrada, y lo mismo ocurre con los siguientes casos de prueba. Las llamadas a `step()` se encargan de:

- Devolver el nuevo estado de las variables en caso de ser modificado.
- Devolver la especificación actualizada tras aplicarle las reglas de procesamiento.
- Actualizar el TokenStream, consumiendo tokens según indiquen las reglas de procesamiento.
- Devolver si sigue siendo válido aplicar la especificación al TokenStream.
- Y, finalmente, devolver si se ha encontrado el separador que indica el final del caso actual.

Después de procesar la entrada, el nuevo estado actual σ es usado para procesar la salida. No obstante, este segundo procesamiento crea un nuevo estado σ' que hemos decidido ignorar y, por tanto, para el siguiente caso de entrada usamos el estado σ. Este detalle es fundamental para el uso de variables comunes entre la especificación de entrada y la de salida. Implica que cualquier variable definida en la especificación de entrada puede usarse tanto en la especificación de entrada como en la de salida, pero las variables definidas en la especificación de salida solo pueden usarse en dicha especificación.

Por ejemplo, veamos la siguiente especificación de entrada:

```
<n:int> --- (<string> <int> ---)^n
```

En esta especificación hemos definido una variable n para definir el número de casos de prueba de entrada. Podemos volver a utilizarla en una especificación de salida como la siguiente:

```
--- (<int> ---)^n
```

Nótese que en esta especificación no hemos definido la variable n en ningún sitio. De hecho, podemos intentar redefinir una variable en la especificación de salida previamente definida en la de entrada, pero esto originará un comportamiento a primera vista inesperado. Lo que ocurrirá es que cualquier redefinición de dicha variable solo afectará al caso de prueba actual, y fuera de este volverá a tener el valor anterior. De hecho, a continuación se muestran varias especificaciones de salida que intentan redefinir n y a qué especificación equivale cada una teniendo en cuenta el comportamiento que hemos descrito anteriormente:

```
function _separateCases(input, output, currentState) {
  while (input.parsedSpec.type !== 'empty' && output.parsedSpec.
      type !== 'empty' && input.foundSeparator && output.
      foundSeparator) {
    input.foundSeparator = false;
    output.foundSeparator = false;

    while (input.validSpec && input.parsedSpec.type !== 'empty' &&
        !input.foundSeparator) {
      ({
        state: currentState,
        spec: input.parsedSpec,
        valid: input.validSpec,
        separator: input.foundSeparator
      } = step(input.parsedSpec, input.tokenStream, currentState))
        ;
    }

    const tempState = currentState;
    while (output.validSpec && output.parsedSpec.type !== 'empty'
        && !output.foundSeparator) {
      ({
        // sin campo state porque ignoramos el nuevo estado
        spec: output.parsedSpec,
        valid: output.validSpec,
        separator: output.foundSeparator
      } = step(output.parsedSpec, output.tokenStream, tempState));
    }
  }
}
```

Figura 5.7: Código de la función `_separateCases()`, que describe cómo se procesan los casos de prueba de entrada y salida

```
<n:int> --- (<int> ---)^n ≡ <m:int> --- (<int> ---)^n

--- (<int> ---)^n <n:int> ≡ --- (<int> ---)^n <m:int>

--- (<n:int> ---)^n <string>^n ≡ --- (<m:int> ---)^n <string>^n

--- (<n:int> <int>^n ---)^n ≡ --- (<m:int> <int>^m ---)^n
```

Por último, debemos mencionar otro detalle sobre la definición de variables. Recordemos las figuras 5.1 y 5.2. En la especificación de entrada podemos declarar una variable en la sección de 'Información previa' (que contenga el número de casos de prueba, por ejemplo), y esta podrá ser usada en cualquier sitio de la especificación de salida. No obstante, también podemos declarar variables que sean locales a un único caso de prueba. Pongamos el caso de la siguiente especificación de entrada:

```
<n:int> --- (<m:int> "abc"^m ---)^n
```

Recordemos que, como llegamos a mencionar durante la sección 5.1.3.1, la cadena `'---'` indica dónde queremos poner un separador. Además, en la especificación anterior tenemos dos variables:

- n, que aparece en la sección de 'Información previa' del fichero, e indica el número de casos de prueba.
- m, que aparece dentro de cada caso de prueba, e indica el número de veces que aparece el literal 'abc'.

La variable m se redefine para cada caso de prueba de entrada. No solo eso, sino que cada valor de m se puede usar en su respectivo caso de prueba de salida. Esto nos permite definir una especificación de salida como la siguiente:

```
--- ("cba"^m ---)^n
```

Lo que significa que la salida consistirá de n casos de prueba, y en cada uno de ellos aparecerá m veces el literal 'cba', dependiendo el valor de m del respectivo caso de prueba de entrada. Esto nos permite un control más fino para especificar la salida en función de la entrada, y en principio podría usarse también para validar los ficheros con los casos de prueba.

5.2.2. TokenStream

Para manejar los tokens de un fichero y escribir los casos de prueba separados con facilidad, decidimos crear un tipo abstracto de datos al que hemos llamado **TokenStream**. Este representa un flujo o *stream* de tokens y permite explorar el tipo del siguiente token del fichero sin consumirlo. Para crear un TokenStream, se necesita una cadena de texto para tokenizar que se haya leído de un fichero, un

`stream.Writable` [21] donde escribir los casos de prueba separados y, opcionalmente, una cadena de texto que sirva como separador.

Esta clase cuenta con funciones para conocer de qué tipo es el siguiente token (`hasLiteral(In,lit)`, `hasInteger(In)`, ...), leer y consumir el siguiente token (`nextInteger(In)`, `nextString(In)`, ...), para escribir un separador en el flujo de salida (`writeSeparator(In)`), y para cerrar el TokenStream con `close(In)` cuando se espera el fin del fichero. A continuación, veremos qué reglas existen y cómo hacen uso de las funciones definidas en la clase TokenStream.

5.2.3. Reglas de procesamiento

Como hemos visto anteriormente, el procesamiento de un TokenStream y una especificación se hace paso a paso. En cada paso se consultan una serie de reglas que, según unas precondiciones, describen qué operaciones hay que realizar sobre el TokenStream y la especificación.

No obstante, para definir las reglas, debemos considerar que:

- Hay especificaciones que se pueden aplicar a cadenas vacías ('' o ε). Por ello definiremos una función booleana $nilP(S, \sigma)$, que indica si la cadena vacía es aceptada por una especificación S dado el estado σ.

- Para las repeticiones de una misma especificación, tenemos que calcular el número de veces que esta se repite, por lo que definiremos una función $evaluate(exp, \sigma)$, que, dada una expresión exp y un estado σ (asignación de variables a valores), devuelve el valor representado por dicha expresión.

Finalmente, presentaremos las definiciones de las reglas de procesamiento usando notación matemática.

5.2.3.1. Especificaciones aplicables a cadenas vacías

Algunas especificaciones pueden aplicarse a una cadena vacía ('' o ε), lo que implica que, dado un TokenStream In y una especificación S, se puede aplicar la especificación S sin consumir nada de In.

Sea, por tanto, la función $nilP(S, \sigma)$, donde S es una especificación y σ el estado de todas las variables definidas hasta ahora. Esta función indica si una

especificación acepta o no una cadena vacía, y se define a continuación:

$$
\begin{aligned}
nilP(\varepsilon, \sigma) &= true \\
nilP(\text{`}\texttt{<x:int>}\text{'}, \sigma) &= \mathit{false} \\
nilP(\text{`}\texttt{<x:string>}\text{'}, \sigma) &= \mathit{false} \\
nilP(\text{`}\texttt{<x:line>}\text{'}, \sigma) &= \mathit{false} \\
nilP(\text{``}\texttt{literal}\text{''}, \sigma) &= \mathit{false} \\
nilP(EOF, \sigma) &= \mathit{false} \\
nilP(\text{`}\texttt{---}\text{'}, \sigma) &= \mathit{false} \\
nilP(S_1 \parallel S_2, \sigma) &= nilP(S_1, \sigma) \vee nilP(S_2, \sigma) \\
nilP(S_1\, S_2, \sigma) &= nilP(S_1, \sigma) \wedge nilP(S_2, \sigma) \\
nilP(S^\wedge *, \sigma) &= true \\
nilP(S^\wedge exp, \sigma) &= \begin{cases} true & \text{si } m = 0 \\ \mathit{false} & \text{si } m > 0 \\ \text{error} & \text{si } m < 0 \end{cases} \\
\text{donde } m &= evaluate(exp, \sigma)
\end{aligned}
$$

5.2.3.2. Evaluación de una expresión aritmética

En las reglas de procesamiento que trabajan con un número determinado de repeticiones de especificaciones (por ejemplo, `<int>^2` o `<line>^(2*x)`), necesitamos calcular dicho número para poder procesar un TokenStream *In*.

Por tanto, necesitamos una función $evaluate(exp, \sigma)$ para calcular el número de repeticiones de una especificación, donde:

- *exp* es una expresión aritmética. Esta tiene que ser aceptada por la gramática de la figura 5.4 (ver reglas para el no terminal `exp`).
- σ es el estado actual de las variables previamente definidas al procesar una especificación S. La notación $\sigma[x]$ se usa para referirse al valor de una variable x.
- $evaluate(exp, \sigma) \in \mathbb{N}$ es el valor numérico de la expresión aritmética *exp*. En caso de encontrar una variable x en *exp*, se consulta su valor en el estado σ.

La función $evaluate(exp, \sigma)$ se define de la siguiente manera:

$$
\begin{aligned}
evaluate(n, \sigma) &= n,\ n \in \mathbb{N} \\
evaluate(x, \sigma) &= \sigma[x],\ x \in \sigma \\
evaluate(e_1 + e_2, \sigma) &= evaluate(e_1, \sigma) + evaluate(e_2, \sigma) \\
evaluate(e_1 - e_2, \sigma) &= evaluate(e_1, \sigma) - evaluate(e_2, \sigma) \\
evaluate(e_1 * e_2, \sigma) &= evaluate(e_1, \sigma) \times evaluate(e_2, \sigma) \\
evaluate(e_1\ /\ e_2, \sigma) &= \lfloor evaluate(e_1, \sigma) \div evaluate(e_2, \sigma) \rfloor \\
evaluate(+e, \sigma) &= e \\
evaluate(-e, \sigma) &= -e
\end{aligned}
$$

5.2.3.3. Definiciones de las reglas

Aquí describimos las reglas que hay que aplicar para procesar un TokenStream In y una especificación S. Usaremos la siguiente notación para representar un paso de procesamiento realizado con éxito:

$$S, In, \sigma \longrightarrow S', In', \sigma'$$

Donde S es la especificación actual, In es el TokenStream actual y σ es el estado actual. S', In' y σ' representan respectivamente los mismos elementos pero tras un paso de procesamiento.

Además, usaremos la siguiente notación cuando no se pueda aplicar ninguna regla a una especificación S, un TokenStream In y a un estado σ:

$$S, In, \sigma \not\longrightarrow$$

Por otro lado, al procesar un token entero $t =$ '`<x:int>`', siendo n el valor del token t, la notación $\sigma[x \to n]$ es el estado σ, pero modificado de modo que $\sigma[x] = n$, lo cual a su vez significa que la variable x tiene el valor n para el nuevo estado. Esta notación se extiende también a tokens *string*, tokens de línea, y tokens literales.

Usaremos también la notación ε para referirnos a una especificación vacía o *consumida*, la cual solo se puede aplicar a una cadena vacía.

Por último, las reglas de procesamiento están escritas con la siguiente notación:

$$\frac{Precondiciones}{Procesamiento}$$

A continuación se muestran las reglas de procesamiento:

1. Si el siguiente token del TokenStream *In* es un entero de valor n y la especificación es `‘<x:int>’` (un token entero cuya variable es x), se consume tanto el token anterior como la especificación, y se actualiza el estado σ a un nuevo estado $\sigma[x \to n]$, de modo que se asigna el valor n a la variable x.

$$\frac{\mathit{hasInteger?}(\mathit{In}) \quad \mathit{nextInteger}(\mathit{In}) = (\mathit{In}', n)}{\texttt{`<x:int>'}, \mathit{In}, \sigma \longrightarrow \varepsilon, \mathit{In}', \sigma[x \to n]}$$

2. Si el siguiente token del TokenStream *In* es una cadena de caracteres de valor *str* y la especificación es `‘<x:string>’` (un token string cuya variable es x), se consume tanto el token anterior como la especificación, y se actualiza el estado σ a un nuevo estado $\sigma[x \to n]$, de modo que se asigna el valor *str* a la variable x.

$$\frac{\mathit{hasString?}(\mathit{In}) \quad \mathit{nextString}(\mathit{In}) = (\mathit{In}', \mathit{str})}{\texttt{`<x:string>'}, \mathit{In}, \sigma \longrightarrow \varepsilon, \mathit{In}', \sigma[x \to \mathit{str}]}$$

3. Si el siguiente token del TokenStream *In* es una línea de valor *ln* y la especificación es `‘<x:line>’` (un token de línea cuya variable es x), se consume tanto el token anterior como la especificación, y se actualiza el estado σ a un nuevo estado $\sigma[x \to n]$, de modo que se asigna el valor *ln* a la variable x.

$$\frac{\mathit{hasLine?}(\mathit{In}) \quad \mathit{nextLine}(\mathit{In}) = (\mathit{In}', \mathit{ln})}{\texttt{`<x:line>'}, \mathit{In}, \sigma \longrightarrow \varepsilon, \mathit{In}', \sigma[x \to \mathit{ln}]}$$

4. Si el siguiente token del TokenStream *In* es un literal de valor *lit* y la especificación es `‘‘lit’’`, se consume tanto el token anterior como la especificación. No se actualiza en este caso el estado σ.

$$\frac{\mathit{hasLiteral?}(\mathit{In}, \mathit{lit}) \quad \mathit{nextLiteral}(\mathit{In}, \mathit{lit}) = (\mathit{In}', \mathit{lit})}{\texttt{``lit''}, \mathit{In}, \sigma \longrightarrow \varepsilon, \mathit{In}', \sigma}$$

5. Si el TokenStream *In* se puede cerrar porque ya no quede ningún otro token por procesar, y la especificación es el token de fin de fichero, se consume la especificación y se cierra *In*.

$$\frac{\mathit{In}' = \mathit{close}(\mathit{In})}{\mathit{EOF}, \mathit{In}, \sigma \longrightarrow \varepsilon, \mathit{In}', \sigma}$$

6. Si la especificación es una disyunción $S_1 \parallel S_2$ y se puede dar un paso de procesamiento $S_1, \mathit{In}, \sigma \longrightarrow S_1', \mathit{In}', \sigma'$, descartamos S_2 y seguimos el procesamiento con S_1', In' y σ'.

$$\frac{S_1, \mathit{In}, \sigma \longrightarrow S_1', \mathit{In}', \sigma'}{S_1 \parallel S_2, \mathit{In}, \sigma \longrightarrow S_1', \mathit{In}', \sigma'}$$

7. Si la especificación es una disyunción $S_1 \parallel S_2$, no podemos dar un paso de procesamiento solo con S_1 (es decir, $S_1, In, \sigma \not\longrightarrow$) y se puede dar un paso de procesamiento $S_2, In, \sigma \longrightarrow S_2', In', \sigma'$, descartamos S_1 y seguimos el procesamiento con S_2', In' y σ'.

$$\frac{S_2, In, \sigma \longrightarrow S_2', In', \sigma' \quad S_1, In, \sigma \not\longrightarrow}{S_1 \parallel S_2, In, \sigma \longrightarrow S_2', In', \sigma'}$$

8. Si la especificación es una secuencia S_1S_2, la especificación S_1 no acepta la cadena vacía y se puede dar el paso de procesamiento $S_1, In, \sigma \longrightarrow S_1', In', \sigma'$, entonces seguimos el procesamiento con $S_1'S_2$, In' y σ'.

$$\frac{\neg nilP(S_1, \sigma) \quad S_1, In, \sigma \longrightarrow S_1', In', \sigma'}{S_1S_2, In, \sigma \longrightarrow S_1'S_2, In', \sigma'}$$

9. Si la especificación es una secuencia S_1S_2, la especificación S_1 acepta la cadena vacía y se puede dar el paso de procesamiento $S_2, In, \sigma \longrightarrow S_2', In', \sigma'$, entonces descartamos la especificación S_1 y seguimos el procesamiento con S_2', In' y σ'. Puede ocurrir que se hubiera podido dar el paso de procesamiento $S_1, In, \sigma \longrightarrow S_1', In'', \sigma''$, y que por no haberlo hecho la especificación S_1S_2 deje de ser aplicable a In tras haber dado varios pasos con la especificación S_2'. No obstante, esto complicaría sustancialmente el procesamiento, por lo que hemos decidido mantener esta limitación. Véase la sección 5.1.3.4 para ver ejemplos de este tipo de especificaciones.

$$\frac{nilP(S_1, \sigma) \quad S_2, In, \sigma \longrightarrow S_2', In', \sigma'}{S_1S_2, In, \sigma \longrightarrow S_2', In', \sigma'}$$

10. Si la especificación es una secuencia S_1S_2, la especificación S_1 acepta la cadena vacía, y no se puede dar el paso de procesamiento solo con S_2 (es decir, $S_2, In, \sigma \not\longrightarrow$), pero sí se puede dar el paso $S_1, In, \sigma \longrightarrow S_1', In', \sigma'$, entonces seguimos el procesamiento con $S_1'S_2$, In' y σ'.

$$\frac{nilP(S_1, \sigma) \quad S_1, In, \sigma \longrightarrow S_1', In', \sigma' \quad S_2, In, \sigma \not\longrightarrow}{S_1S_2, In, \sigma \longrightarrow S_1'S_2, In', \sigma'}$$

11. Si la especificación es $S^\wedge*$ (es decir, la especificación S aparece cero o más veces seguidas), probamos si se puede dar el paso de procesamiento $SS^\wedge*, In, \sigma \longrightarrow S', In', \sigma'$. Si es así, entonces seguimos el procesamiento con S', In' y σ'. Usamos la especificación $SS^\wedge*$ porque así comprobamos si se puede aplicar S al TokenStream In, y este proceso se repetirá todas las veces que podamos seguir aplicando S.

$$\frac{SS^\wedge*, In, \sigma \longrightarrow S', In', \sigma'}{S^\wedge*, In, \sigma \longrightarrow S', In', \sigma'}$$

12. Si la especificación es $S\hat{}exp$, el valor de exp dado el estado σ es n y n es estrictamente mayor que 0, probamos si se puede dar el paso de procesamiento $SS\hat{}(n-1), In, \sigma \longrightarrow S', In', \sigma'$. Si es así, entonces seguimos el procesamiento con S', In' y σ'. Usamos la especificación $SS\hat{}(n-1)$ porque así comprobamos si se puede aplicar una vez S al TokenStream In y este proceso se repetirá otras $n-1$ veces, cuando tengamos la especificación $S\hat{}0$ (ver siguiente regla).

$$\frac{n = evaluate(exp, \sigma) \quad n > 0 \quad SS\hat{}(n-1), In, \sigma \longrightarrow S', In', \sigma'}{S\hat{}exp, In, \sigma \longrightarrow S', In', \sigma'}$$

13. Si la especificación es $S\hat{}exp$ y el valor de exp dado el estado σ es 0, se consume la especificación sin alterar ni el TokenStream In ni el estado σ.

$$\frac{evaluate(exp, \sigma) = 0}{S\hat{}exp, In, \sigma \longrightarrow \varepsilon, In, \sigma}$$

14. Si la especificación es `---` (un separador), entonces consumimos la especificación y escribimos un separador en el TokenStream In'. No alteramos el estado σ.

$$\frac{In' = writeSeparator(In)}{\texttt{---}, In, \sigma \longrightarrow \varepsilon, In', \sigma}$$

5.3. Verificación del correcto funcionamiento del analizador

Por último, tras la implementación de la separación de casos y del lenguaje de especificaciones generado, decidimos incluir la biblioteca Jest [9] para generar conjuntos de *tests*. De esta forma podemos asegurar que ambas funcionalidades se ejecutan sin errores. En la figura 5.8 podemos observar que tenemos 11 *suites* incluyendo un total de 129 *tests* que se han pasado con éxito. La mayor parte de estas pruebas se centran en el procesamiento de tokens, más en concreto, en la clase **TokenStream**, debido a su complejidad. En menor medida, se han comprobado tanto las reglas utilizadas para el procesamiento de los ficheros como la gramática utilizada para especificarlos.

```
❯ npm test

> test-cases-separator@1.0.0 test
> jest

 PASS  tests/TokenStream/read_line_tokens.test.js
 PASS  tests/CasesSeparator/test1.test.js
 PASS  tests/TokenStream/literals.test.js
 PASS  tests/TokenStream/integers.test.js
 PASS  tests/TokenStream/two_strings.test.js
 PASS  tests/TokenStream/closing_tokenstream.test.js
 PASS  tests/TokenStream/only_whitespace_input.test.js
 PASS  tests/TokenStream/writing_separators.test.js
 PASS  tests/TokenStream/empty_input.test.js
 PASS  tests/TokenStream/read_strings_on_several_lines.test.js
 PASS  tests/grammar.test.js

Test Suites: 11 passed, 11 total
Tests:       129 passed, 129 total
Snapshots:   0 total
Time:        3.836 s
Ran all test suites.
```

Figura 5.8: Resultado de ejecutar los tests del analizador de casos de prueba

Capítulo 6

Servidor principal: *frontend* y *backend*

En este capítulo presentaremos el funcionamiento del ***frontend***. Explicaremos una a una todas las páginas de las que se compone la aplicación y cómo se usan, separándolas en páginas del administrador y páginas del usuario. Veremos que algunas de las vistas son comunes para ambos, eliminando ciertas restricciones para los administradores. Al final de la sección explicaremos el *frontend*, donde se explicarán algunos aspectos del código que puedan ser de interés.

También veremos detenidamente la parte del ***backend***. Explicaremos cómo está desarrollado, los algoritmos utilizados, las llamadas a la API de DOMjudge, cómo se transforman y se tratan los datos de las diferentes llamadas y cómo se está haciendo el *scraping* necesario. De la misma manera, también se explicará nuestra API a la que se accede desde el *frontend* y los requisitos o permisos necesarios para la obtención de los diferentes datos.

La explicación de las diferentes páginas junto con su código será de una manera progresiva y continua, es decir, iremos siguiendo los diferentes pasos que un usuario debería ir haciendo desde el primer momento en que accede a la aplicación hasta que se desconecta de ella.

6.1. *Frontend*

Como ya explicamos en la sección 4.2), el *frontend* es la parte visual de DOMfeed con la que el usuario interactúa. Se compone de dos partes: la parte del administrador, orientada a la gestión y control, y la parte del usuario, orientada a la consulta de las entregas de los problemas con resultado erróneo.

Como se explicó anteriormente, el *frontend* está programado con React como *framework* de JavaScript, React-Bootstrap para encapsular Bootstrap en componentes nativos de React para optimizar y simplificar el código, y posteriormente bibliotecas como Axios para hacer las correspondientes llamadas a la API de DOMfeed.

6.1.1. Inicio de sesión

La página del inicio de sesión es la primera que aparecerá al acceder a DOMfeed, a no ser que haya iniciado sesión previamente en la aplicación. Es una página común tanto para el administrador como para el usuario, y cuya única función es permitir que un usuario pueda acceder DOMfeed.

La página está compuesta de un formulario sencillo con dos campos: el primero para introducir un usuario y el segundo una contraseña. Se puede observar que no existe ninguna opción de registro, pues todo usuario que tiene una cuenta en DOMjudge, también la tiene en DOMfeed automáticamente. Es el profesor correspondiente quién debe dar de alta al usuario en DOMjudge. Si ocurre algún error realizando el inicio de sesión, aparecerá una notificación con el mensaje de error correspondiente. En la figura 6.1 se muestra un ejemplo. La parte de la izquierda representa la página de inicio de sesión y la parte derecha se muestra el mensaje de error al introducir incorrectamente las credenciales.

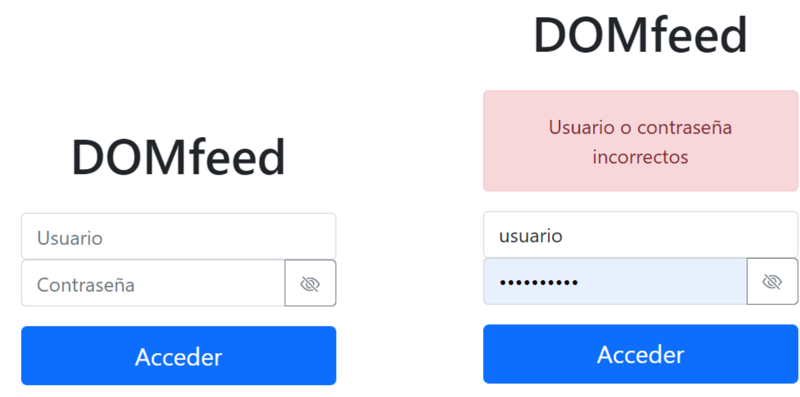


Figura 6.1: Página de inicio de sesión

6.1.2. Cabecera

Al iniciar la sesión, el usuario accederá a la página principal. Para los usuarios administradores será la página de problemas (ver sección 6.1.3.1), mientras que para el resto de usuarios será la página de entregas (sección 6.1.3.3), y estará seleccionado el primer concurso al que pertenezca el usuario.

De esta manera, el usuario podrá acceder a la cabecera de la aplicación, la cual es común a todos los usuarios y a todas las páginas. Mediante la cabecera, el usuario podrá realizar 3 acciones:

- **Ir a la página principal**, la cual depende del tipo de usuario, como se ha explicado anteriormente.
- **Cerrar la sesión**, y volver a la página de inicio de sesión (sección 6.1.1).

- **Seleccionar un concurso**, para así poder acceder a toda su información relevante (problemas, entregas, correcciones). Toda esa información va ligada al concurso en el que se encuentre el usuario. Un mismo problema puede estar en varios concursos, y un mismo equipo puede estar también en varios concursos, por lo que para un mismo problema se van a mostrar entregas diferentes para un mismo equipo, dependiendo del concurso previamente seleccionado.

Para ir a la página principal se dispone de un botón *Home* a la izquierda del todo, para cerrar la sesión se dispone de otro botón más a la derecha con el texto *Cerrar sesión*, y finalmente, para seleccionar el concurso, a la derecha del todo existe un menú desplegable que contiene todos los concursos donde el usuario que ha iniciado sesión tiene permisos para acceder.

Para seleccionar un concurso simplemente hay que desplegar el control de los concursos y hacer clic sobre el concurso deseado. En la figura 6.2 se muestra una captura de la cabecera de DOMfeed.

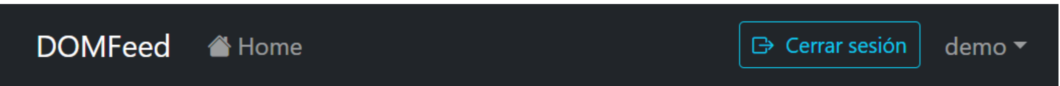


Figura 6.2: Cabecera con el concurso *demo* seleccionado

6.1.3. Páginas del administrador

Las diferentes páginas del administrador están orientadas a la gestión de DOMfeed. Desde ellas, el administrador podrá configurar los problemas, añadiendo las especificaciones correspondientes y el número de correcciones de la solución que el usuario tendrá disponibles para dicho problema. El administrador también podrá ver todas las entregas de todos los grupos pertenecientes al concurso previamente seleccionado, así como sus correcciones o casos fallidos.

6.1.3.1. Vista de problemas

Esta página es exclusiva de los administradores y se accede a ella tras seleccionar el concurso deseado en el desplegable destinado a ello. Tras elegir el concurso se mostrarán todos los problemas correspondientes. Es una página tanto informativa como de gestión, pues es el lugar donde el administrador especifica el número de correcciones máximas para un problema y donde también se accede a la página de especificación.

La página está compuesta de una tabla donde se muestran todos los problemas relativos al concurso seleccionado. Esta tabla contiene una breve información de cada problema que explicamos a continuación:

- **id**, es el identificador del problema. Es único para cada problema. Se crea automáticamente en DOMjudge al registrarse un problema nuevo.

- **Problema**, corresponde al nombre del problema. Suele ser un nombre descriptivo que indique la temática del problema.

- **Abbr**, una abreviación para el nombre del problema.

- **Especificación**, al hacer clic en la celda correspondiente al problema, se accederá a la página de definición de las especificaciones. Se explica detenidamente en la siguiente sección (6.1.3.2).

- **Número de correcciones**, corresponde al número de correcciones disponibles que tiene el usuario o equipo para dicho problema.

Aparte de la tabla de los problemas, en la parte superior de la página se puede ver otro campo con el texto *Número de correcciones disponibles para este concurso*, que indica el número de correcciones máximas que un usuario o equipo tiene para ese concurso. Como ya se ha comentado en la sección 4.2.3, este límite sirve para impedir que el usuario solicite infinitas correcciones.

Como ya se ha mencionado anteriormente, en esta página el administrador puede modificar el número máximo de correcciones disponibles que tendrá un usuario para cierto problema o concurso. Para ello, el administrador tiene que hacer clic sobre el icono correspondiente, en forma de lápiz, junto al número de correcciones de dicho problema. De esta forma, se habilitará la edición en la celda de las correcciones del problema en cuestión. Cuando el administrador desee guardar el valor nuevo, debe hacer clic al icono de *check* que ha aparecido donde antes estaba el lápiz de edición. Si desea cancelar la edición del campo, simplemente tendrá que hacer clic sobre el icono de cruz ubicado a la izquierda del *check*. De manera análoga, para editar el número máximo de correcciones en el concurso, hay que hacer clic en el icono situado a la derecha del valor actual.

En el caso que un problema o concurso no tenga límite de correcciones para el usuario, aparecerá indicado con el símbolo del infinito como valor actual. Para indicar que un problema o concurso no tiene límite de correcciones, simplemente hay que dejar el campo vacío a la hora de editar.

También se puede acceder a la página de las especificaciones que se explica en la sección 6.1.3.2 para crear la especificación del problema seleccionado. La forma de acceder es mediante el icono correspondiente situado en la columna 'especificación' de la tabla de los problemas.

También al hacer clic en una fila de la tabla de los problemas, el profesor puede acceder a una nueva página donde se mostrarán todas las entregas que han realizado todos los grupos del concurso. Esta nueva página se explica en detalle en la sección 6.1.3.3.

Además, al hacer una edición, ya sea del número de correcciones de un problema o del número de correcciones para el concurso, se mostrará una notificación indicando si dicha operación ha tenido éxito o si, de lo contrario, ha ocurrido algún error. Todas estas funcionalidades se pueden ver reflejadas en la figura 6.3.

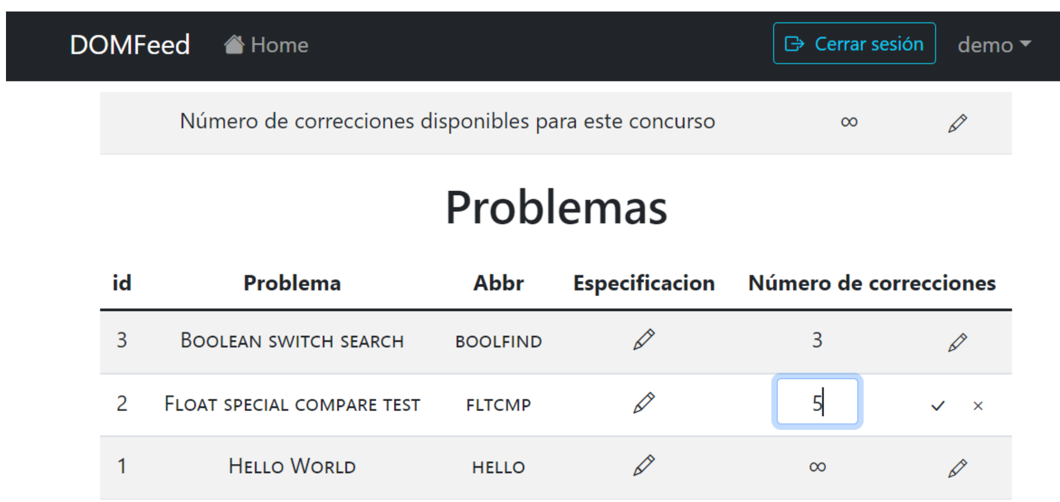


Figura 6.3: Página de problemas

6.1.3.2. Vista de especificaciones

Para cada problema existe una página de especificaciones, cuya función es añadir las dos especificaciones de los casos de prueba, una para el fichero de entrada y otra para el fichero de salida. Estas especificaciones son necesarias para poder separar los casos de prueba y así poder obtener una corrección para las entregas de ese mismo problema.

Al escribir una especificación, esta se analiza continuamente, obteniéndose así una salida en tiempo real de la compilación de la especificación introducida.

En el caso de que la especificación introducida sea incorrecta, la compilación muestra si existe un error en la sintaxis de la especificación introducida o si, en su defecto, la sintaxis es correcta, pero ocurrió un error a la hora de usar la especificación para generar el fichero de salida tras dividir los casos de prueba.

Por otra parte, en el caso de que la especificación introducida sea correcta, se mostrará un ejemplo de cómo queda un fichero de entrada o salida separado utilizando la especificación introducida.

Esta página dispone de 4 componentes principales: dos campos de texto editables para escribir las especificaciones de los problemas y dos campos de texto no editables donde se muestra la salida de la compilación de los mismos.

Cada vez que se edita una de las especificaciones se muestra el resultado de la compilación de dicha especificación en el campo de texto correspondiente a dicha especificación. Este campo se coloreará de verde o de rojo en función de si la especificación introducida es correcta o no. Esto se puede ver claramente en la figura 6.4.

Además, al final de la página se dispone de dos botones:

- Uno para **volver** a la página de problemas.

- Otro para **guardar y aplicar la especificación**, el cual solo tendrá efecto si ambas especificaciones son correctas.

Finalmente, al guardar las especificaciones, aparecerá en la parte superior de la página una notificación que indica si la operación se ha realizado con éxito.

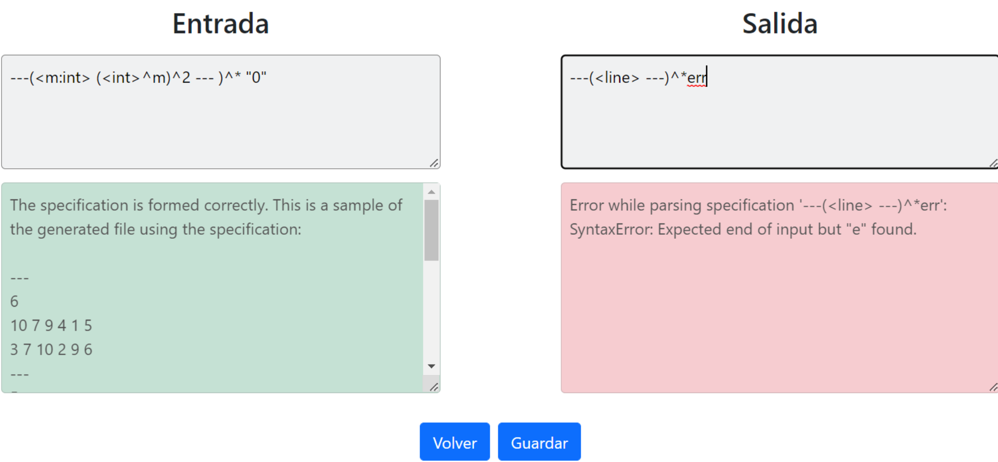


Figura 6.4: Página de especificaciones

6.1.3.3. Vista de entregas

En esta página el administrador puede visualizar las entregas que han realizado todos los grupos para un problema y concurso concreto. Es una vista meramente informativa donde la única interacción que existe en la página es hacer clic en una entrega para acceder a la siguiente página de correcciones.

La página consta de una tabla central, similar a la tabla de los problemas, que contiene todas las entregas de todos los grupos o usuarios para el problema seleccionado. La información de la tabla es la siguiente:

- **id**, identificador de la entrega. Es único para cada entrega. Se crea automáticamente en DOMjudge en el momento que un usuario realiza una entrega a un problema.

- **Hora**, muestra a qué hora el usuario ha realizado la entrega del problema en DOMjudge.

- **Problema**, corresponde al nombre del problema.

- **Equipo**, indica qué equipo ha realizado la entrega. Un equipo puede tener muchos usuarios.

- **Lenguaje**, en qué lenguaje de programación se ha programado el código.

- **Resultado**, muestra el estado de error resultado de ejecutar el problema en DOMjudge con los casos de prueba correspondientes.

En este proyecto, solo estamos contemplando las entregas que DOMjudge haya valorado con *WRONG-ANSWER*. El resto de valoraciones se están ignorando, pues no entran dentro del alcance de DOMfeed, ya que en esos casos no se podrá generar una retroalimentación detallada del caso de prueba que ha fallado. Esto significa que solamente se está mostrando las entregas donde la salida de DOMjudge sea *WA*, es decir, *WRONG-ANSWER*. En la figura 6.5 podemos ver un ejemplo.

La forma en la que se accede a la corrección de la entrega es la que explicamos a continuación. El administrador debe hacer clic sobre la fila de la tabla correspondiente a la entrega que desee ver. Automáticamente se mostrará la siguiente página, que explicamos en la sección 6.1.5

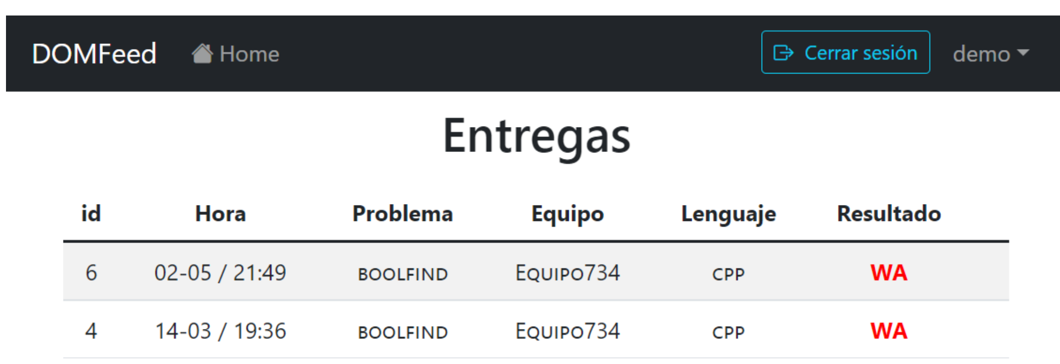


Figura 6.5: Página de entregas para administradores

6.1.4. Páginas del usuario

Las páginas del usuario están destinadas a mostrar información sobre las entregas realizadas con resultado WRONG-ANSWER. Es por ello que el usuario no tiene acceso a la página de problemas como tiene el administrador, ya que esta es una página dedicada a la gestión. Las únicas funcionalidades que tiene el usuario en DOMfeed es ver el listado de entregas realizado y solicitar las correcciones sobre dichas entregas, siempre que se puedan solicitar.

6.1.4.1. Vista de entregas

La vista de entregas del usuario es prácticamente igual a la explicada anteriormente en la sección del administrador. Las diferencias que encontramos son descriptivas, tal y como explicamos a continuación.

La página está compuesta de una tabla central donde se muestran todas las entregas que ha realizado el alumno o grupo para cualquier problema. Esto ya es

una diferencia con respecto al administrador, el cual ve todas las entregas de todos los grupos para un problema determinado.

Otra diferencia se localiza en los componentes de la tabla de las entregas. En el caso de los usuarios, la tabla no contiene la columna 'Equipo', como se mostraba en la vista de los administradores, ya que el equipo siempre es el mismo y sería información redundante. El resto de columnas de la tabla son exactamente las mismas (ver sección 6.1.3.3 del administrador).

La diferencia principal con respecto al administrador se produce en el momento en el que un usuario hace clic sobre una fila de la tabla para ver la corrección de la entrega correspondiente. Al hacer clic, aparece una ventana emergente, la cual muestra la cantidad de correcciones disponibles que quedan y solicita al usuario la confirmación de que realmente quiere canjear una corrección. Cabe recordar que las correcciones van ligadas al grupo o equipo y no a un usuario en concreto, por lo que si un grupo lo componen dos usuarios, y uno de ellos gasta la última corrección, el otro usuario tampoco podrá canjear correcciones nuevas para ese problema. En la parte inferior de la ventana hay dos botones, uno de aceptar y otro de cerrar. Esta ventana se puede ver en la figura 6.6.

Una vez que el usuario ha aceptado la petición de confirmación, internamente se agregará a la base de datos un registro indicando que el usuario ha solicitado y aceptado la corrección para la entrega confirmada. Esto servirá posteriormente para calcular los intentos o comprobaciones restantes que le quedan al grupo para dicho problema y concurso. Cuando se explique el código relativo a esta parte, se explicará con detalle qué datos se guardan en la base de datos y cómo se usan posteriormente para calcular el número de correcciones utilizadas por el grupo. (secciones 6.2.1.1 6.2.2.6).

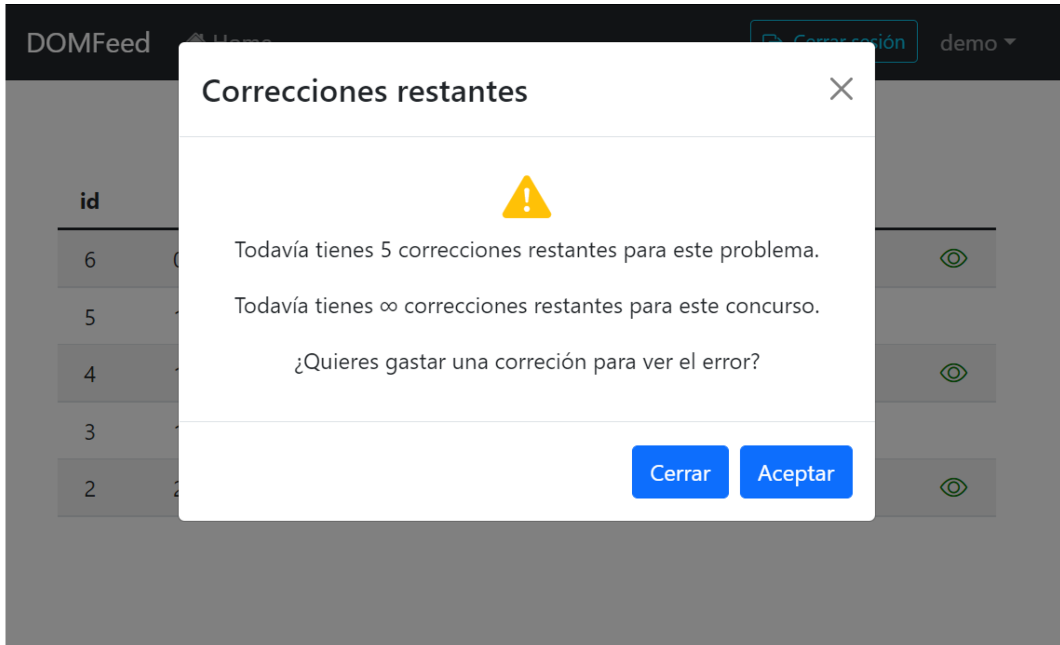


Figura 6.6: Ventana emergente para confirmar la corrección

Si el usuario desea cancelar la petición, debe de hacer clic en el botón de cerrar. De esta forma no quedará registrado y no se consumirán correcciones para dicho problema.

En el caso de que el usuario quiera canjear una corrección y no le queden ya correcciones disponibles, la ventana emergente mostrará un mensaje informativo indicando que no quedan disponibles más correcciones para ese problema. En ese caso, solo aparecerá el botón de cerrar.

Las entregas que hayan sido canjeadas o corregidas, aparecerán representadas en la tabla con un icono de ojo en la última columna, como se puede observar en la figura 6.7. Para estas entregas, si el usuario quiere volver a ver la corrección, no quedará registrada la petición, por lo que no consumirá correcciones restantes. Las entregas anteriormente corregidas podrán volverse a consultar siempre que el usuario quiera, se tengan correcciones disponibles o no.

Figura 6.7: Página de entregas para usuarios normales

6.1.5. Vista de correcciones

La página de las correcciones es meramente informativa y carente de cualquier tipo de interactividad. Es exactamente igual tanto para el administrador como para el usuario. Aquí es donde se muestra la primera discrepancia detectada en la salida de la solución que el usuario ha entregado y la salida esperada. El error o la discrepancia en el resultado obtenido se muestra en rojo, junto al resultado esperado y el caso de prueba que lo ha provocado. Esto se puede comprobar en la figura 6.8. En el caso de que no exista una especificación para el problema referente a la entrega, no se podrá mostrar ninguna corrección y aparecerá un texto en pantalla indicándolo.

La página está dividida en tres secciones debidamente señalizadas y marcadas:

- **Entrada**, se localiza en la parte izquierda de la página y muestra el caso de prueba fallido.
- **Salida esperada**, es la parte central de la página y representa la salida esperada para el caso de prueba fallido.
- **Salida obtenida**, localizada a la derecha de la página y muestra la salida obtenida del usuario al ejecutar el caso de prueba fallido.

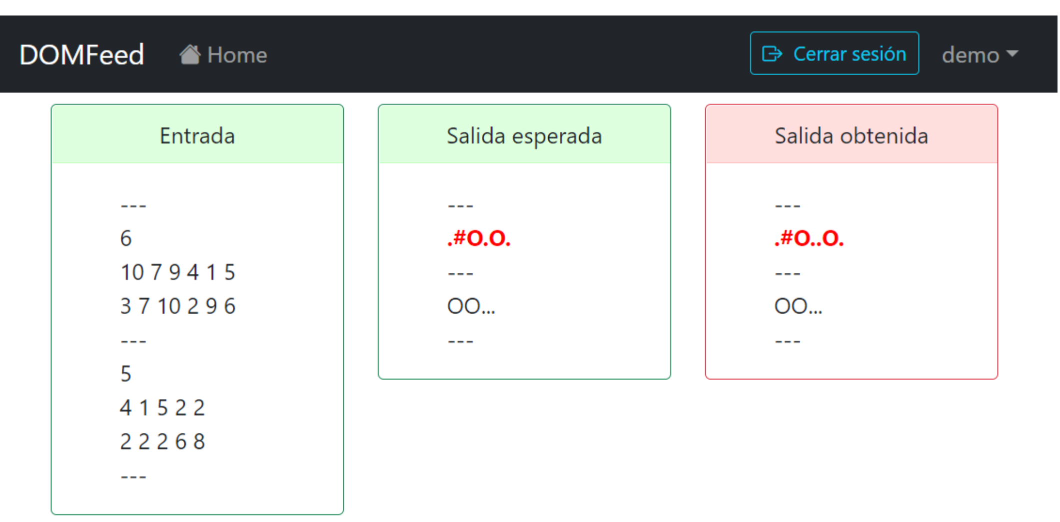


Figura 6.8: Página de correcciones

Hay que tener en cuenta un par de consideraciones con respecto a los datos mostrados. Por lo general, se mostrará solamente un caso de prueba erróneo, junto a su salida esperada y obtenida. Pero existen algunos casos en los que se puede mostrar dos casos de prueba consecutivos. Esto ocurre cuando la salida obtenida que difiere de la esperada corresponde a la frontera con el siguiente caso de prueba.

Supongamos el siguiente ejemplo. Existe un problema que tiene los casos de prueba mostrados en la figura 6.9. Este problema tiene dos casos de prueba y dentro de cada caso hay tres líneas, 1.1, 1.2, 1.3 para el caso 1, 2.1, 2.2, 2.3 para el caso 2. Si el error del usuario se produce al ejecutar la línea 1.2 o 2.2, por ejemplo, se mostraría solamente el caso de prueba, pues no es una sección o caso frontera con el siguiente. Sin embargo, si ocurre en la línea 1.3, por ejemplo, debido a la forma en la que está desarrollado el algoritmo de comparación, no podemos determinar si la discrepancia se debe a que la solución del usuario ha producido la salida incorrecta para la línea 1.3, o si se debe a que el usuario no ha producido ninguna salida para esa línea y, por tanto, correspondería con el principio del siguiente caso de prueba. Es por eso que para que el caso erróneo se muestre siempre al usuario, mostramos los dos casos, tanto el 1 como el 2.

A la hora de mostrar los casos siempre se muestran completos. Es decir, si el error ha ocurrido en el caso 2.2, mostramos el caso 2 entero.

Información previa
<*separador*>
Caso de prueba 1.1
Caso de prueba 1.2
Caso de prueba 1.3
<*separador*>
Caso de prueba 2.1
Caso de prueba 2.2
Caso de prueba 2.3
<*separador*>
Fin de fichero

Figura 6.9: Esquema de caso de prueba

6.1.6. Código de interés

En esta sección explicaremos partes del código que puedan ser de interés o no sean completamente triviales. La mayoría de las pantallas son meramente informativas o de gestión, con formularios simples que no requieren de mayor explicación. Existen algunas partes del código que están más elaboradas y son en las que nos centraremos en esta sección.

6.1.6.1. Mantener la sesión

Una de las partes más laboriosas de DOMfeed ha sido mantener la sesión iniciada. Para ello se ha utilizado un *token* de sesión, el cual guarda diferentes datos del usuario, almacenándolo en el *localStorage* del navegador web. Puesto que los datos guardados en *localStorage* no tienen fecha de caducidad, uno de los valores almacenados en el token es precisamente eso, una fecha de caducidad para que el usuario logueado deba iniciar de nuevo sesión. Otro valor de gran importancia que almacenamos es si el usuario es juez o administrador o, por el contrario, es un usuario normal.

Ahora bien, el proceso por el cual un usuario se mantiene con la sesión iniciada durante todo su recorrido por DOMfeed es el siguiente. Toda la aplicación está encapsulada por un contexto de aplicación llamado `ProvideAuth`, como se puede ver en la figura 6.10. Un contexto de aplicación en esencia es un estado global para todo el árbol de componentes de la aplicación, de manera que sea accesible un valor desde cualquier punto.

La forma en la que hemos creado el contexto utilizado para mantener la sesión es la siguiente. Primero se crea el contexto de la aplicación con la función de React llamada `createContext`. En nuestro caso, hemos guardado el contexto con el nombre `AuthContext`. Posteriormente, se crea el componente `ProvideAuth` que envolverá a toda la aplicación como vimos en la figura 6.10. El componente contiene la variable u objeto que queremos que se mantenga y comparta a lo largo de toda la aplicación, que en este caso sería `auth`, que es un objeto cuyos atributos exportan tres funcionalidades que explicamos a continuación. Dicho objeto se pasa

```
import React from 'react'
import { createRoot } from 'react-dom/client'
import { BrowserRouter } from 'react-router-dom'
import { ProvideAuth } from '@/hooks/useAuth'
import App from '@/App'

const container = document.getElementById('root')
const root = createRoot(container)

root.render(
  <BrowserRouter>
    <ProvideAuth>
      <App />
    </ProvideAuth>
  </BrowserRouter>
)
```

Figura 6.10: Contexto de la aplicación

al componente `AuthContext.Provider` mediante su atributo `value`. De esta manera, todo componente que sea hijo del componente `ProviderAuth`, es decir, esté dentro del componente o debajo en el árbol jerárquico de la aplicación, compartirá el objeto `auth`, en este caso un objeto que exporta lo siguiente:

- **signin**, una función asíncrona cuya función es comprobar si las credenciales proporcionadas por el usuario son correctas haciendo una petición al *backend*, en cuyo caso, los datos del usuario junto, con el token recibido de la respuesta será almacenado en *localStorage*.

- **signout**, es una función que cierra la sesión del usuario. Para ello, elimina los datos de *localStorage*, del estado del contexto y redirige al usuario a la vista de inicio de sesión.

- **user**, es el valor del estado global de la aplicación. Contiene la información relativa al usuario que actualmente ha iniciado sesión.

Además del componente que crea el contexto de la aplicación, también se exporta el *hook* `useAuth`. Este *hook* sirve para obtener el contexto desde cualquier parte de la aplicación para poder acceder a todas las funciones del contexto de la aplicación previamente descritas. En la figura 6.11 se puede ver todo el código referente lo anteriormente explicado.

6.1.6.2. Páginas protegidas

A excepción de la vista de inicio de sesión, todo el resto de páginas de DOM-feed están protegidas, es decir, requieren que el usuario haya iniciado sesión para poder visualizarlas. En el caso de que un usuario no este logueado, se le redirigirá automáticamente desde cualquier página a la vista de inicio de sesión. Una vez que

```
// imports necesarios para el correcto funcionamiento

const AuthContext = createContext()

function ProvideAuth ({ children }) {
  const auth = useProvideAuth()
  return <AuthContext.Provider value={auth}>{children}</
      AuthContext.Provider>
}

const useAuth = () => {
  return useContext(AuthContext)
}

function useProvideAuth () {
  const [user, setUser] = useState(null)
  const navigate = useNavigate()
  const location = useLocation()

  const from = location.pathname !== '/login' ? location.pathname
      : '/login'

  useEffect(() => {
    const subscribe = () => {
      const user = JSON.parse(localStorage.getItem('user'))
      if (user) {
        setUser(user)
        serviceSetToken(user.token)
      }
    }
    if (!user) {
      subscribe()
      navigate(from, { replace: true })
    }
    return () => subscribe()
  }, [])

  const signin = async (username, password) => {
    ...
    // Se hace llamada a la API para confirmar credenciales
    localStorage.setItem('user', JSON.stringify(data))
    setUser(data)
    // Se guarda en token del usuario en los diferentes servicios
        de API
  }
  const signout = () => { ... // Funcion que cierra la sesion y
      redirige al usuario a la pantalla del login }

  return {user, signin, signout}
}
export { ProvideAuth, useAuth }
```

Figura 6.11: Código del contexto y hook useAuth

```
1  import Container from 'react-bootstrap/Container'
2  import VistaProtegida from '@/components/VistaProtegida'
3
4  export default function Layout () {
5
6    return (
7      <VistaProtegida>
8        <Container fluid>
9        ...
10       </Container>
11     </VistaProtegida>
12   )
13 }
```

Figura 6.12: Componente Layout general

```
1  import { Navigate, useLocation } from 'react-router-dom'
2  import { useAuth } from '@/hooks/useAuth'
3
4  const VistaProtegida = ({ children }) => {
5    const auth = useAuth()
6    const location = useLocation()
7    const milliseconds = new Date().getTime()
8
9      // Si el usuario no ha iniciado sesion o si esta ha caducado
10   if (!auth.user || auth.user.hasta < milliseconds) {
11     return <Navigate to='/login' state={{ from: location }} />
12   }
13
14   return children
15 }
16
17 export default VistaProtegida
```

Figura 6.13: Componente vista protegida

el usuario inicie sesión correctamente, se le volverá de redirigir automáticamente a la página a la que quería acceder inicialmente.

Para implementar páginas de acceso restringido, hemos creado el componente `VistaProtegida`. Este componente envuelve todas las rutas de la aplicación, realizando una comprobación de autenticación cada vez que un usuario cambia de ruta en DOMfeed. Para ello, el componente `VistaProtegida` hace uso del *hook* `useAuth` que hemos explicado en la sección anterior. El componente es muy sencillo, simplemente recupera el objeto *auth* del contexto de la aplicación mediante el *hook* `useAuth`. Ese objeto contiene los datos referentes al usuario que ha iniciado sesión. En el caso de que no exista el objeto del usuario o si el tiempo que el usuario lleva logueado ha excedido el tiempo máximo permitido, entonces se le redirige a la vista de inicio de sesión. Se puede ver el código referente a este apartado en las figuras 6.12 y 6.13.

```
<Route path='/problems' element={<Layout />}>
    <Route index element={<Problems />} />
    <Route path=':contestId/' element={<Problems />} />
    ...
</Route>
```

Figura 6.14: Ejemplo de enrutado

6.1.6.3. Notificaciones

En DOMfeed hay algunas páginas en las cuales se muestran notificaciones tras realizar una acción. Son algunas de las pantallas de gestión del administrador: pantalla de problemas y pantalla de especificaciones. Con la idea de que las notificaciones sean un componente general y único para toda la aplicación con el objetivo de que el código no se repita en cada pantalla donde se necesite una notificación, hemos creado un *hook* para mostrar las notificaciones de una forma sencilla en todas las páginas en las que se necesite.

La forma que decidimos para realizar los componentes de notificación es crear un contexto a nivel de enrutado. En la figura 6.14 podemos ver una parte del enrutado de DOMfeed. Como se puede observar, existe una ruta padre `/problems` que renderiza del componente `<Layout/>`. Dentro de dicha ruta tenemos las rutas hijas, que serían, por ejemplo de la siguiente forma `/problems/:contestId/` donde *contestId* corresponde a un número entero positivo. Esta ruta concreta renderiza el componente `<Problems/>`.

El enrutado descrito funciona de la siguiente manera. Cuando un usuario accede a la ruta `/problems/3` para continuar con el ejemplo, se renderiza el componente `<Layout/>`. Dicho componente renderiza a su vez otro componente que se llama `<Outlet/>`. En esencia el componente `<Outlet/>` sería sustituido por el componente que renderice la ruta hija del padre, en este caso `<Problems/>`.

El componente `<Outlet/>` acepta un parámetro *context* el cual se utiliza para crear un contexto a nivel de *Outlet* tal y como habíamos mencionado anteriormente. La creación de un contexto es lo que permite, tal y como vimos en la sección anterior, poder utilizar un objeto en cualquiera de los componentes hijos. En este caso, al ser un contexto que proporciona el propio componente `<Outlet/>` no hay que crearlo manualmente como hicimos con el contexto del usuario (ver sección 6.1.6.1). Simplemente se debe pasar el objeto a compartir como valor del parámeto *context* en el componente `<Outlet/>` y posteriormente utilizarlo en el componente que renderice *Outlet* independientemente de cual sea. En la figura 6.15 se muestra un ejemplo de uso, donde se crea un estado que sirve mostrar las notificaciones de la aplicación. A nosotros lo que nos interesa es que se pueda modificar el valor a mostrar en la notificación en cualquier componente que se renderice. Es por eso que pasamos el método `setNotification` como valor con contexto.

La forma de utilizar las notificaciones en un componente que las necesite sería la mostrada en la figura 6.16. Para ello es necesario importar el *custom hook* `useNotification`. El código de dicho *hook* esta mostrado en la figura 6.17. Es un *hook* muy simple, el cual devuelve el objeto *useOutletContext* que devuelve el

```
import { useState } from 'react'
import Container from 'react-bootstrap/Container'
import Header from '@/components/header/Header'
import Notificacion from '@/components/Notificacion'
import { Outlet } from 'react-router-dom'
import VistaProtegida from '@/components/VistaProtegida'

export default function Layout () {
  const [alert, setAlert] = useState({ type: '', message: '' })

  const setNotification = notification => {
    setAlert(notification)
    setTimeout(() => {
      setAlert({ type: '', message: '' })
    }, 3000)
  }

  return (
    <VistaProtegida>
      <Container fluid>
        <Header />
        <Notificacion alert={alert} />
        <Outlet context={{ setNotification }} />
      </Container>
    </VistaProtegida>
  )
}
```

Figura 6.15: Contexto de las notificaciones

```
1
2
3 export default function Problem ({contestId, id, newValue}) {
4   const { setNotification } = useNotification()
5
6   problemServices.setProblemMaxAttempts(contestId, id, newValue)
7       .then(() => {
8         setNotification({ type: 'success', message: 'Se ha
              actualizado el numero de correcciones' })
9       })
10      .catch(err => {
11        console.error(err.response.data)
12        setNotification({ type: 'danger', message: 'No se ha
              podido actualizar el numero de correcciones' })
13      })
14  }
15
16  return (
17    <tr role='button' className='table-row' {...props}>
18      <td className='fw-normal' onClick={() => onClick(id)}>{id}</
          td>
19      ...
20    </tr>
21  )
22 }
```

Figura 6.16: Ejemplo de uso de las notificaciones

```
1 import { useOutletContext } from 'react-router-dom'
2
3 export function useNotification () {
4   return useOutletContext()
5 }
```

Figura 6.17: Hook useNotification

contexto del *Outlet*, en este caso el método *setNotification* para poder editar la notificación.

6.2. *Backend*

El servidor o *backend* es la parte de DOMfeed que se encarga de procesar las peticiones y los datos obtenidos de DOMjudge, tal y como explicamos anteriormente en la sección 4.3. En esta sección explicaremos las partes más importantes del *backend* y mostraremos ejemplos de cómo se han programado las partes más relevantes en caso que fuera necesario. Además, explicaremos todas las funciones que ofrece la API de DOMfeed indicando los parámetros que espera y los datos que devuelve al cliente. En caso de que fuera necesario, se explicará cómo se ha desarrollado el código para el cálculo de los datos solicitados. Es importante men-

cionar que todas las peticiones a la API de DOMjudge se realizan con un usuario y contraseña administrador codificado en base64 y pasado en la petición mediante la cabecera `Authorization`.

Recordemos que el servidor está desarrollado en su totalidad en Node.js [12], utilizando Express [7] para crear la API necesaria para la solicitud de datos al servidor por parte del *frontend*, y *mysql2* como biblioteca necesaria para conectar e interactuar con la base de datos.

6.2.1. Base de datos

Como ya se ha mencionado en apartados anteriores, DOMfeed tiene una base de datos propia, donde se almacena la información que no se puede obtener mediante la API de DOMjudge, pero es necesaria para las funcionalidades de la aplicación.

Dado que la mayor parte de la información se encuentra en DOMjudge, la base de datos por sí sola posee muy poca información, y las relaciones entre las tablas están limitadas.

La información almacenada es la siguiente:

- El número de correcciones disponibles para un concurso.
- El número de correcciones disponibles para un problema.
- Las especificaciones de los problemas.
- Las entregas cuya retroalimentación se ha solicitado.

6.2.1.1. Estructura de la base de datos

La base de datos está compuesta por cuatro tablas, las cuales se explican detalladamente a continuación. En la figura 6.18 se representa un esquema de cómo están relacionadas las diferentes tablas.

enabled_submissions

En esta tabla se almacenan todas aquellas entregas que han sido habilitadas por un usuario para ser visualizadas mediante el canje de una corrección. Todas las entregas que se encuentran en esta tabla se consideran habilitadas y podrán ser visualizadas por el usuario que la haya canjeado. Por otra parte, aquellas entregas que no se encuentren en esta tabla solo podrán ser visibles por usuarios administradores.

- **contest_id**: Un número entero que simboliza el identificador del concurso al cual pertenece la entrega que se habilita. Este valor se corresponde con el identificador que tiene el concurso en DOMjudge.
- **submission_id**: Un número entero que simboliza el identificador de la entrega que se habilita. Este valor se corresponde con el identificador que tiene la entrega en DOMjudge.

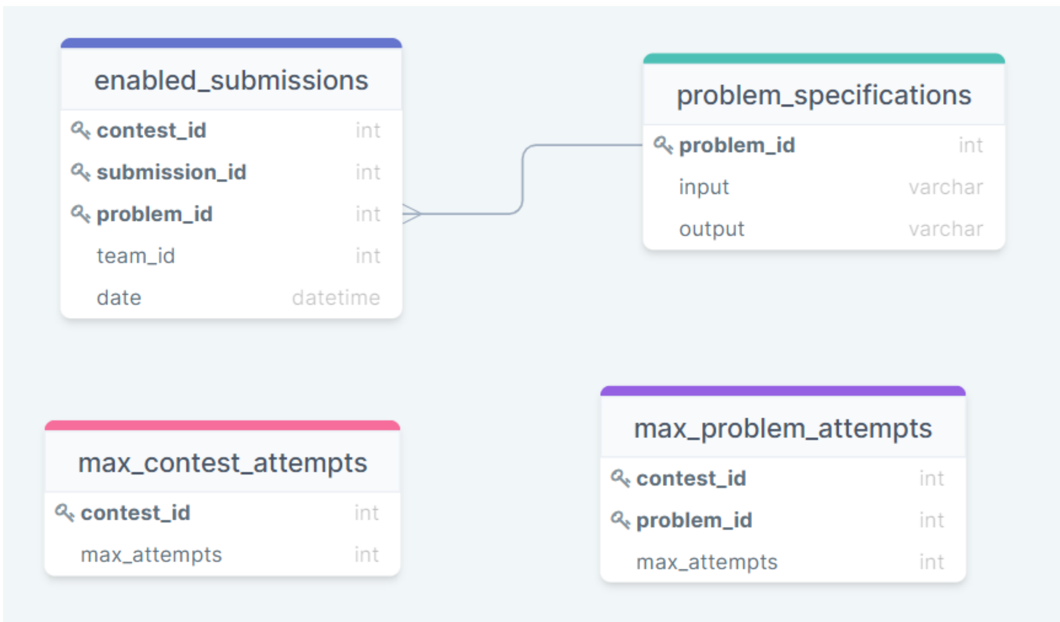


Figura 6.18: Esquema de la base de datos

- **problem_id**: Un número entero que simboliza el identificador del problema al cual pertenece la entrega que se habilita. Este valor se corresponde con el identificador que tiene el problema en DOMjudge. Además, tiene una relación con el campo *problem_id* de la tabla *problem_specifications*, dado que cada problema habilitado debe estar asociado a una especificación.
- **team_id**: Un número entero que simboliza el identificador del equipo que ha habilitado la entrega. Este valor se corresponde con el identificador que tiene el equipo en DOMjudge.
- **date**: Fecha que indica el momento en que se habilitó la entrega.

Es importante señalar que solo con los campos *problem_id* y *submission_id* ya se puede identificar la entrega, pero se añade a la clave el *contest_id*, dado que un mismo problema puede estar presente en varios concursos, pero una corrección podría estar habilitada para únicamente uno de ellos.

problem_specifications

En esta tabla se almacenan las especificaciones de los problemas introducidas por los profesores mediante los siguientes campos:

- **problem_id**: Un número entero que simboliza el identificador del problema cuya especificación se almacena. Este valor se corresponde con el identificador que tiene el problema en DOMjudge.
- **input**: Una cadena variable de 500 caracteres que simboliza la especificación para los ficheros de entrada correspondientes al *problem_id*

- **output**: Una cadena variable de 500 caracteres que simboliza la especificación para los ficheros de salida correspondientes al *problem_id*

max_contest_attempts

En esta tabla se almacena el número de correcciones disponibles para cada concurso. Se considera que cualquier concurso que no se encuentre en esta tabla no tiene un número definido de correcciones, por lo que se considerará que este número es infinito.

- **contest_id**: Un número entero que simboliza el concurso. Este valor se corresponde con el identificador que tiene el concurso en DOMjudge.
- **max_attempts**: Un número entero que se corresponde con el número de correcciones asignado al concurso. Este valor puede tomar el valor *NULL*, en caso de que el concurso tenga una cantidad infinita de intentos.

max_problem_attempts

En esta tabla se almacena el número de correcciones disponibles para cada problema. Se considera que cualquier problema que no se encuentre en esta tabla no tiene un número definido de correcciones, por lo que se considerará que este número es infinito.

- **contest_id**: Un número entero que simboliza el concurso al que pertenece el problema. Este valor se corresponde con el identificador que tiene el concurso en DOMjudge.
- **problem_id**: Un número entero que simboliza el el problema. Este valor se corresponde con el identificador que tiene el problema en DOMjudge.
- **max_attempts**: Un número entero que se corresponde con el número de correcciones asignado al problema. Este valor puede tomar el valor *NULL*, en caso de que el problema tenga una cantidad infinita de intentos.

Al igual que con los identificadores de entregas, el campo *problem_id* ya es suficiente para identificar a un problema, pero se añade a la clave el campo *contest_id*, debido a que un mismo problema puede estar presente en varios concursos, y se puede llegar a especificar un número de intentos para un problema en función del concurso en el que se encuentre.

6.2.1.2. Acceso a la base de datos

Para el acceso a la base de datos se ha utilizado la librería ***mysql2*** de *Node.js*. Mediante esta librería se crea un *pool* de conexiones, el cual se encarga de suministrar conexiones a la base de datos para las peticiones de los usuarios. En la figura 6.19 se muestra un ejemplo de creación de un *pool* de conexiones utilizando la librería antes mencionada.

```
import mysql from 'mysql2'

const pool = mysql.createPool({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'domfeed'
})

const promisePool = pool.promise()

export { promisePool }
```

Figura 6.19: Creación del *pool* de conexiones

Para hacer las peticiones a la base de datos se escribe una consulta SQL parametrizada, donde cada uno de los parámetros correspondientes será analizado internamente y posteriormente sustituido en el lugar que corresponda para así sanear y evitar *ataques de inyección SQL*. Cada una de las consultas a la base de datos está separada en un método individual que sigue el esquema mostrado en la figura 6.20.

```
import { promisePool } from '../../utils/db_connection.js'

const setContestMaxAttempts = (contestId, attempts) => {
  return promisePool.query(`
    INSERT INTO max_contest_attempts (contest_id, max_attempts)
    VALUES ( ? , ? )
    ON DUPLICATE KEY UPDATE max_attempts= ?`, [contestId, attempts
       , attempts])
}
```

Figura 6.20: Ejemplo de consulta SQL utilizando el pool de conexiones

6.2.2. API de DOMfeed

Como ya mencionamos en la sección 4.3.2, DOMfeed dispone de una API realizada mediante la biblioteca Express.js [7]. La API es la base de DOMfeed pues todas las peticiones del usuario, tanto para solicitar datos como para modificarlos se realizan a través de esta API. Todos los *endpoints* de la API siguen la estructura que podemos ver en la figura 6.21.

6.2.2.1. *Midlewares*

Una parte muy importante de las peticiones al servidor son los *middlewares*. Los *middlewares* son funciones intermedias que interceptan las peticiones del usuario antes de llegar a su *endpoint* destino para realizar algún trabajo. Posteriormente

```
router.post('/:contestId/maxAttempts', express.json(), (req, res,
   next) => {
  const { contestId } = req.params
  const { maxAttempts } = req.body

  contestHelper.setMaxAttempts(contestId, maxAttempts)
    .then(() => res.status(200).end())
    .catch(err => next(err))
})
```

Figura 6.21: Estructura de los *endpoints* de la API de DOMfeed

pueden hacer que la petición siga su curso o devolver una respuesta al usuario. Mediante estas funciones se pueden modificar los parámetros de la petición, añadir o borrar datos del cuerpo, etc.

En DOMfeed estamos utilizando un middleware que se ejecuta en cada una de las solicitudes del usuario para extraer la información de un *token* de JWT que se envía en la cabecera de las peticiones en el campo `Authorization`. Ese *token* se crea y se envía al *frontend* en el momento en el que inicia sesión un usuario. Una vez extraído el *token* se guarda como objeto en el *request* de la petición. El código se puede ver en la siguiente figura 6.22.

```
const extractToken = (request, response, next) => {
  const token = request.get('authorization')
  const extractToken = !token || !token.toLowerCase().startsWith('
     bearer ') ? null : token.substring(7)
  request.token = extractToken

  next()
}
```

Figura 6.22: Middleware para extraer el token

El *token* JWT contiene información acerca del usuario que ha realizado la petición: los *roles* de los que forma parte, identificador, *username* y expiración del *token*. Al enviarse de vuelta ese token tras cada llamada a la API de DOMfeed, se comprueba que los parámetros de la petición correspondan a los permisos que tiene el usuario que vienen introducidos en dicho *token*. Imaginemos que un usuario solicita las entregas realizadas para el concurso con identificador 5, pero ese usuario en la información recibida del *token* dice que pertenece al concurso 3. Entonces se producirá un error de permisos bloqueando la solicitud. Esta funcionalidad, está realizada mediante otro *middleware* llamado `verifyTokenAndRequestPermissions` que se ejecuta en todas las peticiones.

6.2.2.2. Obtención de datos de DOMjudge

Para obtener los datos de DOMjudge se ha utilizado la librería Axios [1] la cual nos ayuda a realizar peticiones a servidores de una manera sencilla.

Concursos por equipos

La llamada a `GET /api/team/:teamId/contests` sirve para obtener todos los concursos a los que pertenece un equipo. Sin embargo, la API de DOMjudge no contempla esta petición, si no que, por el contrario, solamente se pueden obtener los equipos a los que pertenece un concurso. Es por ello que a partir de los datos que obtenemos de DOMjudge, obtenemos los datos que necesita DOMfeed.

Problemas por concurso

Mediante la llamada `GET /api/problems/:contestId` se obtienen los problemas pertenecientes al concurso *contestId*. En este caso, la API de DOMjudge sí que dispone de un *endpoint* específico, por lo que no hay que hacer tratamiento especial para devolver dichos datos.

Entregas por equipos

Al invocar al *endpoint* `GET /api/submissions/:contestId/:teamId` de DOMfeed, se obtienen las entregas que ha realizado el equipo con identificador *teamId* para el concurso *contestId* junto con el resultado obtenido por DOMjudge. Obtener estos valores de DOMjudge no es trivial pues no existe una llamada que los proporcione directamente. Para obtenerlos hemos tenido que hacer uso de varias llamadas a la API de DOMjudge. Primero debemos realizar una llamada para obtener todos las entregas para el concurso con identificador *contestId*. DOMjudge no proporciona las entregas filtradas por equipos, por lo que hemos tenido que hacerlo nosotros una vez recibimos los datos. Además, se ha tenido que hacer para cada entrega ya filtrada otra llamada a DOMjudge para obtener el resultado de su ejecución. Aparte, se devuelve también si esa entrega ya ha sido solicitada por el equipo, por lo que hay que hacer otra llamada a nuestra base de datos.

Entregas por problemas

La llamada `GET /api/submissions/:contestId/problem/:problemId` devuelve las entregas pertenecientes al problema con identificador *problemId* para el concurso *contestId*. Los problemas y soluciones presentadas en el cálculo de este *endpoint* son los mismos que los explicados en el caso anterior. El código también es prácticamente idéntico.

6.2.2.3. Gestión del número de correcciones de problemas y concursos

Número de correcciones máximo de un concurso

Cómo se ha explicado previamente, los concursos pueden tener asignados un número máximo de correcciones, el cual se almacena en la base de datos.

Mediante la llamada `GET /api/contests/:contestId/maxAttempts` se puede obtener el número máximo de correcciones del concurso con identificador *contestId*.

En caso de que no esté definido dicho valor en la base de datos, se obtiene el valor predeterminado para el número de intentos de un concurso: `infinito`.

Mediante el *endpoint* `POST /api/contests/:contestId/maxAttempts`, que recibe un número de correcciones máximo en el cuerpo de la petición, se puede establecer el número máximo de correcciones del concurso con identificador *contestId*.

Número de correcciones máximo de un problema

Al igual que los concursos, los problemas pueden tener asignados un número máximo de correcciones en función del concurso al que pertenezcan, el cual se almacena en la base de datos.

Mediante la llamada `GET /api/problems/:contestId/:problemId/maxAttempts` se puede obtener el número máximo de correcciones del problema con identificador *problemId*, perteneciente al concurso *contestId*. En caso de que no esté definido dicho valor en la base de datos, se obtiene el valor predeterminado para el número de intentos de un problema: `infinito`.

Mediante la llamada `POST /api/problems/:contestId/:problemId/maxAttempts`, que recibe un número de correcciones máximo en el cuerpo de la petición, se puede establecer el número máximo de correcciones del problema con identificador *problemId* perteneciente al concurso *contestId*.

Número de correcciones disponibles de un equipo para un problema

Para poder obtener el número de correcciones que le quedan disponibles a un equipo para un problema dentro de un concurso, se utiliza la llamada `GET /api/problems/:contestId/:problemId/userAttempts/:userId`.

Para obtener este valor se calcula el número de correcciones que el usuario aún tiene disponibles para el concurso y para el problema. Para ello se tiene en cuenta el número de correcciones máximo establecido y el número de entregas que se han habilitado para el concurso y el problema.

6.2.2.4. Gestión de especificaciones de un problema

Las especificaciones de un problema son almacenadas en la base de datos para su posterior uso en la generación de los ficheros de casos de prueba separados.

El proceso de obtención de las especificaciones de un problema se hace mediante la llamada a la API de DOMfeed `GET /api/problems/:problemId/specifications` mientras que su modificación se hace llamando a `POST /api/problems/:problemId/specifications`.

Solo se almacenarán aquellas especificaciones que sean válidas para el problema. Dicha comprobación de validez se hace mediante el proceso explicado a continuación.

Análisis de especificaciones

Como ya se ha mencionado anteriormente, en la vista de especificaciones 6.1.3.2 se hace uso de un compilador que muestra si la especificación introducida es co-

rrecta o incorrecta, mostrando también un fragmento del resultado de generar los casos de prueba separados utilizando la especificación introducida.

Para realizar esta operación se utiliza el *endpoint* `POST /api/separator/parse`, la cual recibe en el cuerpo de la petición la especificación a tratar, el identificador del problema, el tipo de especificación (de entrada o de salida), así como la segunda especificación (en el caso de querer tratar la especificación de entrada, este parámetro sería la especificación de salida, y viceversa).

Primero se comprueba si la sintaxis de la especificación introducida es correcta. En caso afirmativo, se descargan los ficheros 6.2.2.5 de los casos de prueba del problema en caso de no encontrarse en el sistema de ficheros de DOMfeed. Una vez hecho esto, en función del tipo de especificación recibida, se intenta separar el primer fichero de casos de prueba para comprobar si la especificación introducida concuerda con el contenido del fichero. Finalmente, si no ha ocurrido ningún error, se separa un caso de prueba y se obtiene el resultado como muestra.

Es importante aclarar que para generar el fichero correspondiente a la especificación de salida, se necesita también la especificación de entrada, de ahí la necesidad de recibir la segunda especificación como parámetro, como se ha explicado previamente.

6.2.2.5. Separación de casos de prueba

Al guardar una especificación en la base de datos mediante la llamada a la API `POST /api/problems/:problemId/specifications`, también se realiza el proceso de separación de los casos de prueba.

Obtención de casos de prueba

Para la obtención de los casos de prueba, debido a que no los proporciona de forma nativa la API de DOMjudge, hay que recurrir al *web scraping*. Mediante el uso del *scraping* en DOMjudge, se inicia sesión en DOMjudge para obtener un token de autenticación que necesitamos usar en las correspondientes llamadas necesarias para la descarga de los ficheros de entrada y salida. En la sección 6.2.3 se explica con mayor detalle.

Una vez autenticados, se procede a la descarga de los ficheros. Dado que no hay una llamada la API que permita obtener estos ficheros, los obtenemos descargándolos a través de la URL `/jury/problems/:problemId/testcases/:i/fetch/input` en el caso de los ficheros de entrada y a la URL `/jury/problems/:problemId/testcases/:i/fetch/output` en el caso de los ficheros de salida, donde i es el número del caso de prueba (por ejemplo, para $i = 1$ se consigue el primer caso de prueba, para $i = 2$ el segundo y así sucesivamente).

Para poder obtener todos los casos de prueba, se accede a ambas URL aumentando el valor de i iterativamente, hasta que se obtenga un error 404 como respuesta, es decir, que no se encuentre la página, indicando así que no hay más casos de prueba disponibles.

Separación de los ficheros

Una vez obtenidos los ficheros con los casos de prueba, se itera sobre cada par entrada - salida esperada y se generan los ficheros con los separadores mediante la función *separateCases* que explicamos en la sección 5.6.

6.2.2.6. Gestión de correcciones

Habilitación de entregas

Un usuario puede habilitar una entrega para obtener su corrección través de la vista de entregas 6.1.3.3. Las entregas que se habilitan se almacenan en la base de datos, para así poder llevar un registro de cuantas correcciones ha utilizado un usuario para un problema o concurso, así como para que este usuario pueda volver a acceder a la retroalimentación de su entrega sin necesidad de consumir otra corrección.

El proceso de habilitación de una entrega, así como el utilizado para obtener si una entrega se ha habilitado previamente se hace mediante las llamadas a la API `POST /api/submissions/:contestId/:submissionId/enabled` y `GET /api/submissions/:contestId/:submissionId/enabled`, respectivamente.

Corrección de una entrega

Una vez que un usuario ha habilitado una entrega para la cual existe una especificación válida, se puede proceder a su corrección, mediante la llamada a la API `GET /api/submissions/:contestId/:submissionId/resultados`.

Una vez que se ha comprobado mediante la API de DOMjudge que el resultado obtenido de la evaluación del problema es `WRONG ANSWER`, se obtiene el fichero de salida generado por el código del usuario a través de la URL `/jury/submissions /:submissionId/runs/:contestId/:runId/team-output`.

Tras la obtención de la salida generada por el usuario junto con la entrada y la salida esperada del problema, obtenidas previamente, los casos de prueba del problema, se hace uso de la función `getCasoErroneo` la cual devuelve el caso de prueba que ha fallado junto con la salida del usuario errónea. Para ello hace uso del la función `comparar` que a partir de la salida esperada y la salida obtenida devuelve el caso erróneo del usuario junto con el número de línea que difiere de la salida esperada.

6.2.3. *Web scraping*

Debido a las limitaciones de la API de DOMjudge nos hemos visto obligados a utilizar *web scraping* para obtener datos de DOMjudge que de otra manera habría sido imposible conseguir de forma automática. Durante todo el desarrollo de la aplicación se ha necesitado hacer *scraping* para descargar tanto los datos de prueba de un problema como la salida obtenida tras ejecutar la solución al problema de un alumno en DOMjudge. El único fichero que se puede obtener mediante la API de DOMjudge es el código que ha entregado como solución al problema.

La forma que encontramos de poder obtener esos ficheros es realizando una llamada a ciertas URL que se explican en las secciones 6.2.2.5 y 6.2.2.6. El problema se presenta cuando descubrimos que para obtenerlos había que haber iniciado sesión en DOMjudge previamente con un usuario administrador. Es por eso que tenemos que usar *scraping* para iniciar sesión en DOMjudge de una manera programática, capturar la *cookie* de inicio de sesión que se genera y posteriormente utilizarla en las peticiones a las rutas necesarias para descargar los archivos. De esta manera podemos descargar tanto los casos de prueba como la salida obtenida por el usuario haciendo creer a DOMjudge que estamos logueados. Para ello son necesarios dos pasos: obtener el *CSRF token* del formulario de inicio de sesión junto con el PHPSESSID, y posteriormente utilizar esos valores para hacer una petición de inicio se sesión a DOMjudge, colocándolos en su lugar correspondiente en la llamada, con un usuario y contraseña administrador.

Capítulo 7

Conclusiones y trabajo futuro

En esta sección explicamos hasta qué grado se han cumplido los objetivos principales, las dificultades encontradas durante el desarrollo del proyecto y qué mejoras se pueden añadir en un futuro.

7.1. Objetivos alcanzados

Este proyecto tenía como objetivos principales:

- La definición de un lenguaje de especificación de casos de prueba y la implementación de un mecanismo que permita separar los casos de prueba de un problema a partir de una especificación.
- El desarrollo de una aplicación web que permita al alumno ver en qué caso de prueba ha fallado la entrega realizada por este y al administrador registrar las especificaciones necesarias para que se puedan separar los casos de prueba del problema.

Podemos decir que ambos objetivos se han cumplido.

En primer lugar, el lenguaje de especificación desarrollado es intuitivo y no hemos encontrado ningún problema para el que no pueda describir la entrada y salida. Además, hemos redactado una guía de uso para que los profesores puedan familiarizarse con dicho lenguaje, la cual, no obstante, no hemos podido añadir a la página web al final. Por otra parte, hemos desarrollado más de 100 pruebas o *tests* para garantizar el correcto funcionamiento tanto de la gramática que reconoce el lenguaje de especificación como de la separación de casos de prueba, y el analizador ha superado todos (ver sección 5.3).

Con respecto al segundo objetivo, la aplicación web es bastante similar a DOMjudge, ya que así logramos mantener consistencia externa y facilitar a los nuevos usuarios el proceso de familiarización con el entorno. Las credenciales de acceso a la aplicación son las mismas que para DOMjudge por lo que no hay proceso de registro. Para cada problema, el usuario tendrá un número de correcciones fijado por el administrador del concurso, tanto para el concurso como el problema. En la vista de la corrección se muestra el primer caso de prueba de la solución del

usuario que difiere de la solución esperada. Al introducir la especificación de un problema, se lleva a cabo en tiempo real la comprobación de la sintaxis en la vista del administrador, así como la generación de un caso de prueba con separadores para verificar su correcto funcionamiento.

7.2. Dificultades encontradas

Durante el desarrollo del proyecto nos surgieron varias dificultades, a veces derivadas del uso de tecnologías que no conocíamos previamente, y otras veces por problemas que no fuimos capaces de prever. Otro inconveniente que afectó a todo el equipo fue que casi todos tuvimos menos tiempo el primer cuatrimestre que el segundo, por lo que el desarrollo del proyecto fue más lento al principio. Además, a continuación se exponen las dificultades y problemas que afectaron a cada grupo durante el desarrollo del proyecto.

7.2.1. Grupo 1

- El grupo 1 tuvo que aprender a manejar el lenguaje de gramáticas de PEG.js. Por tanto, antes de crear el lenguaje de especificación de DOMfeed, sus miembros estuvieron experimentando con gramáticas más sencillas y leyendo la documentación [15].

- Al principio la función separadora de casos de prueba funcionaba de forma síncrona, pues el grupo 1 tenía un conocimiento limitado de JavaScript, y además nunca había trabajado con Node.js. Por tanto, no fue hasta el segundo cuatrimestre que consiguieron un separador de casos asíncrono. Las dificultades surgieron principalmente por no estar familiarizados con la programación dirigida por eventos característica de Node.js.

7.2.2. Grupo 2

- Al iniciar el proyecto, el grupo 2 tenía muy poca experiencia trabajando con React y Node.js, por lo que se tuvo que familiarizar y aprender a utilizar dichas herramientas para la elaboración de la aplicación web.

- Durante el mes de diciembre, cuando el proyecto ya tenía una base sólida, DOMjudge lanzó una nueva versión de su API. Sin embargo, esta nueva actualización eliminaba el dato *team_id* en la respuesta de la llamada a la API con los datos del usuario. Sin este dato no se podía obtener la información de concursos y problemas del usuario, por lo que la aplicación no funcionaba correctamente. Para solucionar este problema se contactó con los desarrolladores de DOMjudge, quienes confirmaron que, efectivamente, se trataba de un error y lo solucionaron.

- Las limitaciones de la API de DOMjudge, al no ofrecer los casos de prueba así como la salida de un usuario a un problema determinado, nos provocaron la búsqueda de alternativas. Llegamos a la conclusión de que la única forma de poder obtener los ficheros de una manera automatizada era mediante *scraping*. Hubo que aprender algunos conceptos del *scraping* para poder utilizarlo correctamente.

- La API de DOMjudge no siempre proporciona los datos de la forma en la que los necesitamos, por lo que hemos tenido que procesar muchos de los datos obtenidos. En algunos casos se han tenido que hacer hasta tres llamadas a la API de DOMjudge para obtener todos los datos necesarios para construir el objeto que necesitamos.

7.3. Trabajo futuro

Los objetivos iniciales han sido alcanzados. No obstante, creemos que el proyecto tiene capacidad de mejora, por lo que a continuación se exponen algunas ideas para ello:

7.3.1. Lenguaje de especificación

- **Introducir un token de fin de línea:** En el enunciado de muchos problemas de programación se indica, por ejemplo, que un caso de prueba tiene dos líneas, donde la primera tiene un entero y la segunda tiene varios. Ahora mismo, nuestro lenguaje de especificación tiene dos formas de describir casos de prueba como el anterior: usando tokens `<int>` o tokens `<line>`. Sin embargo, no hay ninguna manera de especificar que una línea tiene un solo entero. Es por ello que consideramos útil que se pueda especificar dónde acaba y, por tanto, empieza la siguiente línea usando un nuevo token.

- **Introducir un token para números reales:** Consideramos que por completitud debería añadirse un token para números reales, pues ya hay uno para números enteros. Ahora mismo una especificación tendría que usar un token *string* para reconocer un número real.

7.3.2. Implementación de la separación de casos de prueba

- **Reducir el consumo de memoria del TokenStream:** Para crear un TokenStream, es necesario haber leído todo el fichero con los casos de prueba y pasarle un `string` a la constructora. Esto puede llegar a incurrir en un gran consumo de memoria para ficheros inusualmente grandes, y además supone que no podemos tokenizar el fichero hasta haberlo leído completamente. Se sugiere buscar una manera sencilla de ir leyendo el fichero por partes sin complicar excesivamente la lógica de las funciones que interactúan directamente con la implementación de un TokenStream.

- **Escribir más pruebas o *tests*:** Al tener una especificación que no se pueda aplicar a un fichero de casos de prueba, se imprime un mensaje de error. Este mensaje debería imprimir la fila y la columna correcta del fichero donde se encontró el error, pero no se está verificando que este comportamiento funcione todavía. Además, convendría realizar más pruebas que cubran las reglas de procesamiento y la separación de casos, pues la cobertura de las pruebas existentes no supera más del 60 % del código en ambos casos.

7.3.3. *Frontend*

- Se podría valorar la posibilidad de proporcionar el código del usuario que ha propiciado la salida errónea. Existe un *endpoint* en la API de DOMjudge que devuelve el código entregado como solución a un problema para cierta entrega.

- Las tablas de la aplicación actualmente no tienen ningún tipo de paginación, por lo que en el momento en el que un concurso tenga muchos problemas o un problema tenga muchas entregas, habrá un problema grave de rendimiento. Sería interesante añadir una paginación a las tablas de veinte filas, por ejemplo.

- Actualmente, el *frontend* no contiene ningún test para comprobar funcionalidades y evitar que estas se rompan al añadir nuevas o modificar existentes. Se podría utilizar la biblioteca Cypress [4] para realizar este trabajo.

- Existen ciertas páginas en DOMfeed que tienen una apariencia demasiado simple como es la página 404. Podría definirse una hoja de estilos para estas páginas.

- Se podría crear un icono para la aplicación, el cual se pueda colocar en la página de inicio de sesión, en la cabecera y en el favicon de la página web.

- Se podría añadir el manual de las especificaciones a la aplicación web, para que así los profesores tengan un fácil y rápido acceso a la hora de redactar especificaciones.

7.3.4. *Backend*

- Implementar un *log* de errores para la aplicación, donde se lleve registro de todos los imprevistos ocurridos en el servidor. Actualmente, solo se devuelven los errores al cliente y se muestran o por consola o como notificación.

- Se podría mejorar la gestión del sistema de ficheros. Actualmente, se descargan las salidas obtenidas por las ejecuciones de los usuarios cada vez que se pide una corrección y nunca se borran. Solo se sustituyen cuando se vuelve a pedir la corrección. Se podría evitar volver a descargar un fichero ya existente, o borrar periódicamente aquellos ficheros que ya no sean necesarios o que no se utilicen.

Capítulo 8

Conclusions and Future Work

In this section we explain to what extent the main objectives have been met, the difficulties encountered during the development of the project and what improvements can be added in the future.

8.1. Objectives achieved

The main objectives of this project were:

- The definition of a test case specification language and the implementation of a mechanism to separate the test cases of a problem with a specification.
- The development of a web application that allows the student to see in which test case the student's submission has failed and the administrator to record the necessary specifications so that the test cases can be separated from the problem.

We can say that both goals have been met.

First, the specification language developed is intuitive, and we have not found any problem for which there is no specification to describe the input and output. In addition, we have written a user's guide for teachers to familiarize themselves with this language, which, however, we have not been able to add to the website in the end. On the other hand, we have developed more than 100 tests to ensure the correct functioning of both the grammar that recognizes the specification language and the separation of test cases, and the parser has passed all of them (see section 5.3).

Regarding the second objective, the web application is quite similar to DOMjudge, as this way we manage to maintain external consistency and make it easier for new users to familiarize themselves with the environment. The access credentials to the application are the same as for DOMjudge, so there is no registration process. For each problem, the user will have a number of corrections set by the contest administrator, both for the contest and the problem. In the correction view, the first test case of the user's solution that differs from the expected solution is displayed. When entering a problem specification, syntax checking is performed in

real time in the administrator's view, as well as the generation of a test case with separators to verify correct operation.

8.2. Difficulties encountered

During the development of the project we encountered several difficulties, sometimes derived from the use of technologies that we were not previously familiar with, and other times due to problems that we were not able to foresee. Another inconvenience that affected the whole team was that almost all of us had less time in the first quarter than in the second, so the development of the project was slower at the beginning. In addition, the following are the difficulties and problems that affected each group during the development of the project.

8.2.1. Group 1

- Group 1 had to learn how to handle the PEG.js grammar language. Therefore, before creating the DOMfeed specification language, its members were experimenting with simpler grammars and reading the documentation. [15].

- At first the test case separator function worked synchronously, as group 1 had limited knowledge of JavaScript, and also had never worked with Node.js. Therefore, it was not until the second quarter that they got an asynchronous case separator. The difficulties arose mainly because they were not familiar with the event-driven programming characteristic of Node.js.

8.2.2. Group 2

- At the beginning of the project, group 2 had very little experience working with React and Node.js, so they had to familiarize themselves with and learn how to use these tools for the development of the web application.

- During the month of December, when the project already had a solid base, DOMjudge released a new version of its API. However, this new update eliminated the `team_id` data in the API call response with the user's data. Without this data, it was not possible to obtain the information of contests and problems from the user, so the application did not work correctly. To solve this problem, we contacted DOMjudge developers, who confirmed that it was indeed a bug and fixed it.

- The limitations of the DOMjudge API, by not offering the test cases as well as the output of a user to a given problem, caused us to search for alternatives. We came to the conclusion that the only way we could get the files in an automated way was by scraping. We had to learn some concepts of scraping to be able to use it correctly.

- The DOMjudge API does not always provide the data the way we need it, so we have had to process a lot of the data obtained. In some cases, we have had to make up to three DOMjudge API calls to get all the data needed to build the object we need.

8.3. Future work

The initial objectives have been achieved. However, we believe that the project has room for improvement, so here are some ideas for it:

8.3.1. Specification language

- **Creating an end-of-line token:** In the statement of many programming problems it is stated, for example, that a test case has two lines, where the first has one integer and the second has several. Right now, our specification language has two ways to describe test cases like the above: using tokens `<int>` or tokens `<line>`. However, there is no way to specify that a line has a single integer. This is why we find it useful to be able to specify where the next line ends and thus begins using a new token.

- **Creating a token for real numbers:** We believe that for completeness a token for real numbers should be added, as there is already one for integers. Right now, a specification would have to use a string token to recognize a real number.

8.3.2. Implementation of test case separation

- **Reducing TokenStream memory consumption:** In order to create a TokenStream, it is necessary to have read the whole file with the test cases and pass a `string` to the constructor. This can be very memory intensive for unusually large files, and also means that we cannot tokenize the file until it has been read completely. It is suggested to find a simple way to read the file in parts without overcomplicating the logic of the functions that interact directly with the implementation of a TokenStream.

- **Writing more tests:** When a specification cannot be applied to a test case file, an error message is printed. This message should print the correct row and column of the file where the error was encountered, but this behavior is not being verified to work yet. In addition, more tests covering processing rules and case separation would be desirable, as the existing test coverage does not exceed more than 60 % of the code in both cases.

8.3.3. Frontend

- It could be considered to provide the user code that caused the erroneous output. There is an endpoint in the DOMjudge API that returns the code

delivered as a solution to a problem for a certain submission.

- The application tables currently do not have any pagination, so the moment a contest has many problems or a problem has many submissions, there will be a serious performance problem. It would be interesting to add pagination to tables of twenty rows, for example.

- Currently, the frontend does not contain any tests to check functionality and prevent it from breaking when adding new or modifying existing functionality. The Cypress [4] library could be used to do this job.

- There are certain pages in DOMfeed that have a too simple appearance, such as the 404 page. A stylesheet could be defined for these pages.

- An icon could be created for the application, which can be placed on the login page, in the header and in the favicon of the web page.

- The specifications manual could be added to the web application, so teachers can have an easy and quick access when writing specifications.

8.3.4. *Backend*

- Implement an error log for the application, where all unforeseen events occurring on the server are logged. Currently, only errors are returned to the client and are displayed either by console or as a notification.

- The management of the file system could be improved. Currently, the outputs obtained by user runs are downloaded each time a correction is requested and are never deleted. They are only replaced when the correction is requested again. It would be possible to avoid re-downloading an existing file, or periodically deleting files that are no longer needed or used.

Capítulo 9

Contribuciones personales

En esta sección se describen las contribuciones de cada componente del grupo. Es importante recordar que el equipo se dividió en dos subgrupos, de modo que el grupo conformado por Erik Karlgren Domercq y Adrián González Cabanillas se encargó de la separación de los casos de prueba, mientras que el grupo formado por Félix Redondo Manzanares y Ricardo Enrique Freire Sacco se encargó de la aplicación web.

9.1. Félix Redondo Manzanares

Félix ya tenía conocimientos iniciales tanto en `Javascript` como en Node.js y React, por lo que tuvo un inicio más rápido en el desarrollo de DOMfeed. Los primeros días estuvo familiarizándose con DOMjudge y todo lo relativo a su API. Durante el primer cuatrimestre, Félix asumió todo el trabajo referente a la parte correspondiente al *frontend* y *backend* ya que Ricardo no tenía disponibilidad. En el segundo cuatrimestre fue Ricardo quien realizó casi todo el trabajo restante.

9.1.1. *Scraping*

Una de las primeras tareas fue poder descargar la salida producida por el alumno para cierto problema. Félix se dio cuenta de que no se podía obtener dicha salida por medio de la API, así que tuvo que buscar una alternativa. La única forma que encontró fue mediante una llamada a DOMfeed desde una de las pantallas de gestión del administrador, por lo que era necesario poder iniciar sesión como administrador y descargar el archivo. Para ello tuvo que recurrir al *scraping*, por el cual consiguió iniciar sesión en DOMjudge y extraer el *token* de inicio de sesión necesario para poder descargar la salida del usuario. En la sección 6.2.3 se explica detalladamente el proceso seguido.

9.1.2. Algoritmo de búsqueda de caso erróneo

Una vez obtenido la salida del alumno había que compararla con la salida esperada. Inicialmente, los casos de prueba del problema (casos de entrada y salida

esperada para dichos casos), estaban almacenados en su local. Félix realizó una primera iteración del algoritmo que posteriormente tuvo que modificarlo dos veces más a lo largo del desarrollo a causa de variaciones en los requerimientos.

9.1.3. API de DOMfeed

Félix desarrolló los siguientes *endpoints* de la API:

- `POST /api/login`: para el inicio de sesión del usuario.
- `GET /api/team/:teamId/contests`: para obtener los concursos de un usuario o equipo.
- `GET /api/problems/:contestId`: que devuelve todos los problemas de un concurso.
- `GET /api/submissions/:contestId/:teamId`: retorna las entregas de un usuario o equipo para un concurso.
- `GET /api/submissions/:contestId/problem/:problemId`: entrega todas las entregas de todos los equipos para un problema de un concurso.
- `GET /api/submissions/:contestId/:submissionId/resultados`: devuelve los casos de prueba erróneos.

9.1.4. Páginas de la interfaz web

La mayor parte de la interfaz web fue desarrollada por Félix, pues tenía más experiencia. Las páginas junto con la lógica de estas que desarrolló son las siguientes:

- **Inicio de sesión**: vista para el inicio de sesión del usuario.
- **Problemas**: página que muestra los problemas por concursos.
- **Entregas**: vista que muestra las entregas de un equipo para un concurso.
- **Correcciones**: página donde se muestran el caso erróneo de la entrega.

9.1.5. Sistema de notificaciones

Félix desarrolló un componente de React para mostrar notificaciones al usuario como *feedback* tras realizar ciertas acciones. El componente se puede utilizar desde cualquier página de la web, pues está colocado en un nivel inicial del árbol de componentes que componen la aplicación, y cuya función o mecanismo para mostrar y cambiar el texto de la notificación está guardado en el contexto de React, por lo que cualquier componente hijo puede acceder a él.

9.1.6. Páginas privadas

La única página de DOMfeed donde un usuario puede acceder sin iniciar sesión es la vista de *login*. El resto de páginas requieren de una autenticación previa del usuario. Para ello, Félix desarrolló un *hook* que comprueba en cada pantalla que el usuario está *logueado*. El *hook* se almacena a nivel de contexto de aplicación, por lo que se puede acceder desde cualquier vista de la web.

9.1.7. Mantenimiento de la sesión del usuario

Para evitar que el usuario perdiera la sesión tras refrescar el navegador y que tuviera que volver a iniciar sesión en DOMfeed, Félix desarrolló un método donde se guardaba un *token* de sesión en el *localStorage* del navegador. En esencia, cada vez que se refresca el navegador, se comprueba que existe ese *token* y se agrega al contexto de la aplicación para indicar la existencia de un usuario *logueado*.

9.1.8. Segurización de la API

Para asegurar que todas las peticiones a la API de DOMfeed fueran de usuarios autenticados, Félix tuvo que desarrollar un sistema de seguridad. Este, en esencia, consiste en un *token* con información relativa al usuario que ha iniciado sesión que manda al *frontend* al inicio de esta, y posteriormente se devuelve en cada llamada a la API. Básicamente, la seguridad se consigue comprobando la existencia del *token* en cada llamada a la API por medio de un *middleware* y, en el caso de que exista y no esté alterado ni corrupto, hacer otra serie de comprobaciones para validar que el usuario que realiza la petición tiene permisos para obtener los datos solicitados.

9.1.9. Llamada a la API de DOMjudge

Tanto las llamadas a la API como el posterior tratamiento de los datos obtenidos fueron desarrolladas por Félix, puesto que esta parte fue trabajo del primer cuatrimestre. Tras cada llamada corresponde un tratamiento de los datos para adaptarlos a lo solicitado por el usuario a través de nuestra API de DOMfeed. En algunas ocasiones tuvo que hacer varias llamadas a DOMjudge y cruzar los datos obtenidos para poder conseguir los datos necesarios.

9.2. Ricardo Enrique Freire Sacco

Al iniciar el proyecto, Ricardo no tenía ninguna experiencia trabajando con tecnologías como React o Node.js, por lo que los primeros meses se centró principalmente en aprender los conceptos básicos de dichas tecnologías. Además, por temas de disponibilidad no pudo aportar al proyecto durante el primer cuatrimestre, por lo que el segundo cuatrimestre asumió la mayor parte de la carga de trabajo.

9.2.1. API de DOMfeed

Ricardo desarrolló los siguientes *endpoints* de la API:

- `GET /api/contests/:contestId/maxAttempts`: para obtener el número máximo de correcciones de un concurso.
- `POST /api/contests/:contestId/maxAttempts`: para establecer el número máximo de correcciones de un concurso.
- `GET /api/problems/:contestId/:problemId/userAttemps/:userId`: para obtener el número de correcciones que un usuario tiene para un problema en un concurso.
- `GET /api/problems/:contestId/:problemId/maxAttempts`: para obtener el número máximo de correcciones de un problema.
- `POST /api/problems/:contestId/:problemId/maxAttempts`: para establecer el número máximo de correciones de un problema.
- `GET /api/problems/:problemId/specifications`: para obtener las especificaciones de un problema.
- `POST /api/problems/:problemId/specifications`: para establecer las especificaciones de un problema y separar los casos de prueba.
- `POST /api/separator/parse`: para hacer la comprobación de las especificaciones y separar un caso de prueba para obtenerlo como muestra.
- `GET /api/submissions/:contestId/:submissionId/enabled`: para obtener si una entrega ha sido habilitada para su corrección previamente.
- `POST /api/submissions/:contestId/:submissionId/enabled`: para habilitar una entrega para su corrección.

9.2.2. Base de datos

El diseño, la creación y la integración de la base de datos en la aplicación fue realizado por Ricardo. Esta sería utilizada para las funciones de gestión del número de correcciones, así como para la gestión de especificaciones.

9.2.3. Gestión del número de correcciones

Ricardo se encargó de todo lo relacionado con limitar el número de correcciones de un problema o concurso. Se encargó de crear las tablas en la base de datos necesarias para esta funcionalidad, así como de las llamadas de la API necesarias y de las operaciones correspondientes para el tratamiento de los datos.

9.2.4. Gestión de especificaciones

De manera similar a la gestión del número de correcciones, Ricardo se encargó de todo lo relacionado con la gestión de las especificaciones de un problema (acceso a la base de datos, llamadas a la API y tratamiento de datos), así como de su posterior uso para la separación de los casos de prueba.

9.2.5. Interfaz web

Como se ha mencionado previamente, la mayor parte de la interfaz web fue desarrollada por Félix, por lo que la mayor parte del trabajo de Ricardo en la interfaz web consistió en establecer el intercambio de información entre la interfaz y la API de DOMfeed (envío y recibimiento de información, así como de mensajes de error).

Además, también llevo a cabo el desarrollo de la vista de especificaciones, en la cual se introducen las especificaciones de los problemas y se obtiene el resultado de su análisis.

9.2.6. Integración del separador de casos en la aplicación web

Una vez que el separador de casos estaba finalizado, Ricardo se encargó de su integración en la aplicación web.

Para ello se tuvo que encargar de las siguientes tareas:

- Comprobar que las especificaciones introducidas por el usuario son válidas, tanto en sintaxis como a la hora de separar un fichero con casos de prueba.
- Obtener los ficheros con los casos de prueba de los problemas de la API de DOMjudge.
- Utilizando los ficheros con los casos de prueba y las especificaciones, obtener todos los casos de prueba separados para un problema.

9.3. Adrián González Cabanillas

Durante las primeras semanas del proyecto realizó pruebas con gramáticas simples para familiarizarse con la librería `PEG.js` así como refrescar sus conocimientos de `Javascript` y aprender Node.js. En el primer cuatrimestre no pudo seguir aportando al proyecto por problemas de disponibilidad. No obstante, en el segundo cuatrimestre llevó a cabo las siguientes tareas:

9.3.1. Gramática del lenguaje de especificación

Adrián incorporó las siguientes funcionalidades a la gramática:

- Variables anónimas
- Variables asociadas a un número o literal
- Secuencia de disyunciones
- Repetición de secuencias

9.3.2. Implementación de reglas

Estas reglas son usadas para interpretar la gramática definida en la sección superior.

- **Repetición**: Añadió al lenguaje de especificaciones la posibilidad de que una expresión se repita, o bien un número indeterminado de veces, o bien una cantidad constante. Esta característica corresponde a la función `stepForRep()` dentro de `step()`
- **Disyunciones**: La disyunción es un operador lógico que permite procesar una especificación u otra dependiendo de su validez. La implementación de esta funcionalidad hace que problemas complejos en los que la entrada pueda estar definida de varias formas posibles resulte sencilla de especificar. La función implementada es `stepForDisjunctive()`.

9.3.3. Variables y estados

Implementó las variables dentro de la gramática para así poder, por ejemplo, repetir una expresión un número constante de veces. Esta funcionalidad da lugar a más posibilidades a la hora de especificar un fichero, ya que el estado de una variable puede ser modificado más de una vez dentro de una especificación. Si un token no es asignado a una variable, se interpreta que este tiene una variable anónima asociada.

9.3.4. Especificaciones que enlazan la entrada con la salida

Por otro lado, consiguió que la especificación de salida dependa de la especificación de entrada. Es decir, si una variable está definida en la especificación de entrada, al ser referenciada en la especificación de salida, esta tendrá el valor previamente establecido en la entrada. Cabe mencionar que el procesamiento de los casos se hace de forma individual, primero consumiendo la especificación de entrada y posteriormente la de salida. Es decir, hasta que el primer caso no termina de procesarse, el segundo caso no comienza.

9.3.5. Elaboración de tests

La creación de tests fue realizada en su mayoría por Erik, sin embargo, Adrián también participó realizando tests para probar la implementación de las disyunciones en el lenguaje de especificación. Para ello realizó *tests* con expresiones disyuntivas simples, complejas y anidadas.

9.3.6. Agrupación de casos de prueba separados

Adrián implementó una función para crear una carpeta por problema en el sistema de ficheros. De esta forma, al terminar la separación de casos de prueba, los ficheros se almacenan en dicho directorio.

9.3.7. Manual de especificaciones

Tanto Erik como Adrián decidieron que sería bastante provechoso la elaboración de un manual para que cualquier administrador de concursos pueda especificar cualquier problema. Por ello, Adrián comenzó describiendo el tipo de elementos de la gramática, las posibles combinaciones entre especificaciones y aspectos generales a tener en cuenta.

9.4. Erik Karlgren Domercq

Al principio del curso, Erik no tenía experiencia alguna con `JavaScript`, al contrario que sus compañeros. Por tanto, dedicó las primeras semanas a aprender `JavaScript` y `Node.js`, pero también a usar `PEG.js` con gramáticas sencillas. Tuvo que asumir casi todo el trabajo de la parte del analizador de casos de prueba durante el primer cuatrimestre debido a que Adrián no podía dedicar apenas tiempo al proyecto. En cambio, en el segundo cuatrimestre, el reparto de tareas fue significativamente más equitativo.

A continuación se describen las tareas que llevó Erik a cabo durante todo el proyecto:

9.4.1. TokenStream

Erik desarrolló la primera versión del TokenStream y se encargó de añadirle nuevas funcionalidades a lo largo del transcurso del proyecto. Esto incluye las operaciones para:

- Consultar el tipo del siguiente token: `hasInteger()`, `hasString()`, `hasLine()` y, además, `hasLiteral()`.
- Leer el siguiente token: `nextInteger()`, `nextString()`, `nextLine()` y, también, `nextLiteral()`.
- Escribir el separador de casos de prueba: `writeSeparator()`.

Posteriormente, en el segundo cuatrimestre, refactorizó el código del TokenStream para facilitar su comprensión, y escribió decenas de pruebas o *tests* para comprobar que el TokenStream funcionaba correctamente. Veáse la sección 5.3 para más detalles.

9.4.2. Gramática del lenguaje de especificación

Erik desarrolló casi todo el lenguaje de especificación. Las siguientes características fueron desarrolladas por él:

- Inclusión de tokens para números enteros (`<int>`), cadenas de caracteres separadas por espacios (`<string>`), líneas de texto (`<line>`), literales (`‘‘...’’`), el separador (`---`) y el fin de fichero.

- Secuencias de especificaciones.

- Repeticiones de especificaciones, empezando por un número indeterminado de veces (`^*`) y un número fijo (`^n`). Finalmente, implementó las repeticiones que usan expresiones aritméticas con números enteros y variables (ejemplo: `^(2*x)`).

El resto de características del lenguaje fueron desarrolladas por Adrián.

9.4.3. Reglas de procesamiento

Erik programó las reglas de procesamiento relativas a las características del lenguaje de especificación ya mencionadas. Además, creó las siguientes funciones auxiliares:

- $\mathit{nilP}(S, \sigma)$ para comprobar si la especificación S acepta una cadena vacía dado el estado σ.

- $\mathit{evaluate}(\mathit{exp}, \sigma)$ para calcular el valor de la expresión exp dado el estado σ.

9.4.4. Separación de casos de la entrada

Hasta el segundo cuatrimestre solo trabajábamos con especificaciones de entrada. Por tanto, la separación de casos al principio solo era para la entrada, y esta la implementó Erik.

Originalmente, la separación se hacía de forma síncrona, con lo que había que esperar a que se terminaran de separar los casos de prueba antes de poder realizar otras acciones. No obstante, Erik acabó implementando una versión asíncrona tras estudiar cómo funcionaban los *callbacks* y las funciones *async* en `Node.js`.

9.4.5. Elaboración de tests

Como ya hemos comentado, Erik realizó varios *tests* para el TokenStream. Sin embargo, también creó algunos tests para verificar que:

- La gramática de `PEG.js` leía correctamente las especificaciones y rechazaba las incorrectas.
- Se separaban los casos de prueba de varios ficheros de ejemplo como se esperaba.

No obstante, se deberían hacer más pruebas para comprobar que realmente funcionan correctamente tanto la separación de casos de prueba como el procesamiento de especificaciones.

9.4.6. Documentos

Erik completó el manual de especificaciones que empezó Adrián con más ejemplos y explicaciones. Está pensado para que los profesores que usen DOMfeed puedan aprender a especificar cualquier problema de programación usando el lenguaje de especificaciones.

Apéndice A

Ejemplos de especificaciones

En este apéndice se presentan ejemplos de especificaciones de entrada y de salida para problemas de programación reales y se muestra cómo quedarían separados los ficheros de casos de prueba. Estos problemas han sido usados en exámenes o prácticas del grado de Ingeniería Informática de la Universidad Complutense de Madrid.

A.1. Obtener el mínimo de un vector

Este problema apareció en el examen final de septiembre de 2014 de la asignatura de Fundamentos de Algoritmia.

A.1.1. Enunciado

Consideramos un vector $V[N]$ de números enteros, cuyos valores se han obtenido aplicando una rotación sobre un vector ordenado en orden estrictamente decreciente. Implementa un algoritmo que calcule el mínimo del vector con una complejidad $O(log(n))$. El número de elementos sobre los que se aplica la rotación para obtener el vector de entrada es un valor entre 0 y N y no se conoce.

Por ejemplo, un posible vector de entrada sería el vector `70 55 13 4 100 80`, obtenido desplazando los dos primeros elementos del vector `100 80 70 55 13 4` al final del mismo.

Requisitos de implementación.

Se debe implementar una función recursiva (resolver) que dado el vector, con los datos de entrada ya leídos, obtenga el mínimo en tiempo logarítmico respecto al número de elementos del vector. Se pueden utilizar más parámetros si se considera necesario.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de dos líneas. En la primera se indica el número de elementos del vector y en la

```
1  Entrada                          Salida
2
3  4                                2
4  8 6 4 2                          2
5  4                                2
6  2 8 6 4                          2
7  4                                1
8  4 2 8 6                          1
9  4                                4
10 6 4 2 8
11 5
12 8 5 3 1 10
13 5
14 5 3 1 10 8
15 6
16 70 55 13 4 100 80
```

Figura A.1: Mínimo de un vector: casos de prueba sin separar

segunda los valores del vector.

Salida

Para cada caso de prueba se escribe en una línea diferente el mínimo del vector.

Véase la figura A.1 para ver el ejemplo de entrada y de salida del enunciado de este problema.

A.1.2. Especificaciones

Para definir las especificaciones solo necesitamos prestar atención a las secciones de *Entrada* y de *Salida*. Como bien dice el enunciado, para cada caso de prueba de entrada tenemos dos líneas, teniendo la primera un entero con el número de elementos de la siguiente línea, que a su vez son también enteros. No se menciona en ningún sitio que haya algo de información previa a todos los casos de prueba, como la cantidad de los mismos, ni de que se marque el fin de la entrada de alguna forma. Por tanto, tenemos dos posibles **especificaciones para la entrada**:

```
--- (<n:int> <int>^n ---)^*
```

```
--- (<line>^2 ---)^*
```

No obstante, hay que tener en cuenta que en la primera especificación no importa que cada caso de prueba esté en dos líneas, en una, o si todos los casos de prueba se encuentran en la misma línea. Por otro lado, la segunda especificación no nos serviría en el caso de que los datos de cada caso de prueba de la entrada no siguieran de manera estricta las condiciones del enunciado y los datos de un mismo caso estuvieran en una o más de dos líneas.

Por último, hay también dos posibles **especificaciones para la salida**:

```
1  Entrada                          Salida
2
3  ---                              ---
4  4                                2
5  8 6 4 2                          ---
6  ---                              2
7  4                                ---
8  2 8 6 4                          2
9  ---                              ---
10 4                                2
11 4 2 8 6                          ---
12 ---                              1
13 4                                ---
14 6 4 2 8                          1
15 ---                              ---
16 5                                4
17 8 5 3 1 10                       ---
18 ---
19 5
20 5 3 1 10 8
21 ---
22 6
23 70 55 13 4 100 80
24 ---
```

Figura A.2: Mínimo de un vector: casos de prueba separados

```
--- (<int> ---)^*
```

```
--- (<line> ---)^*
```

Ambas especificaciones tienen las mismas peculiaridades que sus respectivas especificaciones para la entrada, pero al igual que estas, si las condiciones sobre la entrada y la salida se siguen de manera estricta, servirían para separar los casos de prueba apropiadamente. De hecho, los casos de prueba que aparecen en el enunciado (figura A.1) se separan correctamente usando cualquiera de las especificaciones de entrada y de salida anteriores, como puede observarse en la figura A.2.

A.2. El Grande

Este problema apareció en una práctica de la asignatura de Estructuras de Datos. Fue creado por Manuel Montenegro, uno de los tutores del TFG.

En este problema se menciona que hay que implementar un *TAD*, que son las siglas de *Tipo Abstracto de Datos*.

A.2.1. Enunciado

El Grande es un juego de mesa ambientado en la España medieval. En este juego, los jugadores tienen que colocar estratégicamente a sus caballeros por las distintas regiones del país. Decimos que un jugador predomina en una región cuando tiene estrictamente más caballeros en dicha región que cualquiera de los restantes jugadores. Por otro lado, decimos que una región está en disputa si está ocupada por al menos un caballero, pero ningún jugador predomina en la misma. En particular, las regiones vacías (esto es, sin caballeros) no están en disputa.

En este ejercicio trataremos de implementar un TAD que almacene la información correspondiente a una partida de este juego. Las operaciones a implementar son:

- anyadir_jugador(jugador). Añade un jugador a la partida. Si el jugador (de tipo string) ya estaba inscrito en ella, se lanza una excepción domain_error con el mensaje Jugador existente.

- colocar_caballero(jugador, region). Indica que el jugador (que es de tipo string) coloca un caballero en la region (de tipo string) indicada. Si el jugador no se encuentra inscrito en la partida, se lanza la excepción domain_error con el mensaje Jugador no existente. Si la region no existe, se dará de alta.

- puntuacion(jugador). Devuelve el número de regiones en las que predomina el jugador pasado como parámetro. Si el jugador no se encuentra inscrito en la partida, se lanza la excepción domain_error con el mensaje Jugador no existente.

- regiones_en_disputa(). Devuelve un vector<string> con la lista de regiones que están en disputa. La lista ha de estar ordenada ascendentemente, en orden alfabético según el nombre de la región.

- expulsar_caballeros(region). Elimina a todos los caballeros de la region pasada como parámetro. Si la región no existe o no tiene ningún caballero, se lanza la excepción domain_error con el mensaje Region vacia.

En este ejercicio se pide:

1. **Implementar** las operaciones descritas en el TAD. Ninguna de ellas debe realizar operaciones de E/S. El manejo de E/S debe hacerse en la función tratar_caso().

2. Indicar el **coste** de cada operación. Las operaciones deben implementarse de la manera más eficiente desde el punto de vista del coste asintótico en tiempo.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso está formado por una serie de líneas en las que se muestran las operaciones a llevar a cabo, una por cada línea: el nombre de la operación seguido de sus argumentos. La palabra FIN en una línea indica el final de cada caso.

Los nombres de jugadores y regiones son cadenas de caracteres sin espacios en blanco.

Salida

Las operaciones que generan salida son:

- puntuacion J, que debe escribir una línea con el mensaje Puntuacion de J: X, donde J es el jugador del cual quiere obtenerse la puntuación, y X es la puntuación del mismo.

- regiones_en_disputa, que debe escribir una línea con el mensaje Regiones en disputa:, seguida de los nombres de las regiones que están en disputa, uno por línea.

Si una operación produce un error, entonces se escribirá una línea con el mensaje ERROR:, seguido del mensaje de la excepción que lanza la operación, y no se escribirá nada más para esa operación.

Cada caso termina con una línea con tres guiones (---)

Véase la figura A.3 para ver el ejemplo de entrada y de salida del enunciado de este problema.

A.2.2. Especificaciones

La salida y la entrada de este problema es significativamente más compleja que la del problema anterior. Aun así, podemos usar especificaciones bastante sencillas para describirlas. Veamos las siguientes:

- Entrada: --- (<line>^* "FIN" ---)^*

- Salida: --- (<line>^* "---" ---)^*

Aquí estamos aprovechando que el analizador de casos de prueba, dada una especificación S^* T y una cadena x, intentará primero comprobar si la cadena x es aceptada por la especificación T, y solo si no es el caso, comprobará si es aceptada por la especificación S. Por eso, en el caso de la especificación de entrada, si se lee una cadena 'FIN' no se trata como un token de línea, sino como el literal ‘‘FIN’’. Nótese además que ''---'' no se trata como un separador en la especificación al tratarse de un literal. Véase la figura A.4 para ver cómo quedan los casos de prueba separados.

```
1  Entrada                                 Salida
2
3  anyadir_jugador jug1                    Puntuacion de jug1: 1
4  anyadir_jugador jug2                    Puntuacion de jug2: 0
5  colocar_caballero jug1 Granada          Puntuacion de jug1: 0
6  colocar_caballero jug2 Granada          Puntuacion de jug2: 0
7  colocar_caballero jug1 Granada          Regiones en disputa:
8  puntuacion jug1                         Aragon
9  puntuacion jug2                         Granada
10 colocar_caballero jug2 Granada          ---
11 puntuacion jug1                         ERROR: Jugador no existente
12 puntuacion jug2                         Puntuacion de jug1: 2
13 colocar_caballero jug2 Aragon           Puntuacion de jug1: 1
14 colocar_caballero jug1 Aragon           ERROR: Region vacia
15 regiones_en_disputa                     ---
16 FIN
17 colocar_caballero jug1 Sevilla
18 anyadir_jugador jug1
19 colocar_caballero jug1 Valencia
20 colocar_caballero jug1 Galicia
21 puntuacion jug1
22 expulsar_caballeros Valencia
23 puntuacion jug1
24 expulsar_caballeros Valencia
25 FIN
```

Figura A.3: El Grande: casos de prueba sin separar

```
1  Entrada                                    Salida
2
3  <====================>                     <====================>
4  anyadir_jugador jug1                       Puntuacion de jug1: 1
5  anyadir_jugador jug2                       Puntuacion de jug2: 0
6  colocar_caballero jug1 Granada             Puntuacion de jug1: 0
7  colocar_caballero jug2 Granada             Puntuacion de jug2: 0
8  colocar_caballero jug1 Granada             Regiones en disputa:
9  puntuacion jug1                            Aragon
10 puntuacion jug2                            Granada
11 colocar_caballero jug2 Granada             ---
12 puntuacion jug1                            <====================>
13 puntuacion jug2                            ERROR: Jugador no existente
14 colocar_caballero jug2 Aragon              Puntuacion de jug1: 2
15 colocar_caballero jug1 Aragon              Puntuacion de jug1: 1
16 regiones_en_disputa                        ERROR: Region vacia
17 FIN                                        ---
18 <====================>                     <====================>
19 colocar_caballero jug1 Sevilla
20 anyadir_jugador jug1
21 colocar_caballero jug1 Valencia
22 colocar_caballero jug1 Galicia
23 puntuacion jug1
24 expulsar_caballeros Valencia
25 puntuacion jug1
26 expulsar_caballeros Valencia
27 FIN
28 <====================>
```

Figura A.4: El Grande: casos de prueba separados

Bibliografía

[1] *Axios, Promise based HTTP client.* URL: https://axios-http.com/.

[2] *Bootstrap, the world's most popular framework for building responsive sites.* URL: https://getbootstrap.com/.

[3] *Classical Logic.* URL: https://plato.stanford.edu/archives/win2009/entries/logic-classical/.

[4] *Cypress, frontend testing.* URL: https://www.cypress.io/.

[5] *DOMjudge API.* URL: https://www.domjudge.org/demoweb/api/doc.

[6] *DOMjudge documentation.* URL: https://www.domjudge.org/docs/manual/8.0/index.html.

[7] *Express.js, A API framework for Node.js.* URL: https://expressjs.com/es/.

[8] Russ Ferguson. *Beginning Javascript: The Ultimate Guide to Modern JavaScript Development.* Apress, 2019. ISBN: 1484243943.

[9] *Jest - Delightful JavaScript Testing.* URL: https://jestjs.io/.

[10] *Jutge, The Virtual Learning Environment for Computer Programming.* URL: https://jutge.org/.

[11] *Mooshak, automatic judge.* URL: https://mooshak.dcc.fc.up.pt/.

[12] *Node.js, A JavaScript runtime built.* URL: https://nodejs.org/es/.

[13] *npm, The world's largest software registry.* URL: https://www.npmjs.com/.

[14] *Package.json, Node and npm config file.* URL: https://docs.npmjs.com/cli/v7/configuring-npm/package-json.

[15] *PEG.js - Documentation.* URL: https://pegjs.org/documentation.

[16] *PEG.js - Parser Generator for JavaScript.* URL: https://pegjs.org/.

[17] J. Petit y col. «Jutge.org: Characteristics and Experiences». En: *IEEE Transactions on Learning Technologies* 11.3 (2017), págs. 321-333. DOI: https://doi.org/10.1109/TLT.2017.2723389.

[18] *React Bootstrap, the world's most popular framework rebuilt for React.* URL: https://react-bootstrap.github.io/.

[19] *React Router, A routing library for React.* URL: https://reactrouter.com/.

[20] *React, A JavaScript library for building user interfaces*. URL: https://es.reactjs.org/.

[21] *stream.Writable | Node.js v18.2.0 Documentation*. URL: https://nodejs.org/api/stream.html#class-streamwritable.

[22] Basarat Ali Syed. *Beginning Node.js*. Apress, 2014. ISBN: 9781484201886.