

# **PROYECTO FINAL DE INGENIERÍA**

**Lenguaje de Programación Natural**

**Furriel, Mariano Rodrigo – LU1036959**

Ingeniería en Informática

**Rodriguez, Gabriel Pablo – LU1036569**

Ingeniería en Informática

Tutor:

**Godio, Claudio, UADE**

Co-Tutor:

**Ronconi, Francisco, UADE**

**Agosto 14, 2017**



**UNIVERSIDAD ARGENTINA DE LA EMPRESA**  
**FACULTAD DE INGENIERÍA Y CIENCIAS EXACTAS**

## **Agradecimientos**

A nuestras familias por el apoyo a lo largo de toda la carrera.

## Resumen

El presente Proyecto Final de Ingeniería propone el diseño e implementación de una herramienta de procesamiento capaz de realizar el pasaje de una porción de texto redactado en lenguaje natural, hacia su equivalente en un lenguaje de programación real ejecutable por una computadora. Esta herramienta se complementa con una plataforma gráfica interactiva que hace uso de la misma, y permitirá a usuarios con conocimientos técnicos mínimos tener un primer acercamiento al mundo de la programación.

El diseño de la plataforma fue llevado adelante teniendo en mente principal pero no únicamente a usuarios jóvenes y adolescentes, y contemplando la posibilidad de emplearlo en ámbitos académicos de nivel secundario, terciario o universitario. A partir de este potencial surgió también la idea y posterior implementación de una mecánica de “aula virtual”, emulando la dinámica profesor-estudiante y permitiendo la generación de ejercicios a medida en forma de cursos o clases, en paralelo a un intercambio constante entre profesores y estudiantes en forma de correcciones y consultas.

Desde lo técnico, el proyecto se divide en dos partes conceptualmente distintas: Por un lado, las herramientas relacionadas al pasaje de lenguaje natural a lenguaje de programación; por otro, la implementación de la plataforma interactiva. El pasaje de lenguaje natural a lenguaje de programación fue resuelto con una mecánica de dos pasos, con un pasaje intermedio hacia un pseudocódigo de definición propia, y finalmente con su traducción hacia un lenguaje de programación. La plataforma interactiva fue desarrollada en forma de página web, con un back-end sencillo que implementa el modelo de aula virtual a la vez que hace uso de las herramientas de lenguaje natural para el procesamiento de los inputs del usuario.

Entre las conclusiones a las que arribamos tras el desarrollo destacamos la observación del gran potencial de aplicación real del proyecto, particularmente con la inclusión la plataforma interactiva, que añade un importante valor didáctico. Dicho valor es fundamental, considerando que el objetivo central del proyecto es, en definitiva, promover el aprendizaje.

**Abstract**

The current Engineering Final Project proposes the design and implementation of a processing tool capable of translating a portion of natural language text to its equivalent portion of computer-runnable code. Said tool will be complemented by a graphical interactive platform that will make use of it, and allow users with little or no technical knowledge to make a first approach to the world of programming.

The design of the platform was carried on keeping in mind mainly -but not exclusively- teenage and young users, and considering the possibility of it being used in an academic environment. From this possibility also came the idea and later implementation of a 'virtual classroom' mechanic, emulating professor-student dynamics and giving place to creating custom exercises as part of courses or classes, parallel with a constant exchange between professors and students in the shape of questions and observations.

On a technical aspect, the project was conceptually divided into two different parts: On one side, the tools related to translating natural language to a programming language; on the other, the implementation of the interactive platform. The passage from natural language to programming language was solved with a two-step mechanism, with an intermediate translation to a pseudocode specified later on, and finally its' translation to a specific programming language. The interactive platform was developed as a web page, with a simple back-end that implements the virtual classroom model and makes use of the natural language tools to process user inputs.

Amongst the conclusions obtained after the development of the project, its potential for real world application is the most relevant, especially including the interactive platform, which adds a significant didactic value. Said value is fundamental, considering the central goal of the project is, at its core, to promote learning.

## Contenidos

<b>Agradecimientos</b>	<b>2</b>
<b>Resumen</b>	<b>3</b>
<b>Abstract</b>	<b>4</b>
<b>Contenidos</b>	<b>5</b>
<b>Introducción</b>	<b>7</b>
<b>Antecedentes</b>	<b>9</b>
Logo	10
Scratch	10
Codecademy	11
<b>Descripción</b>	<b>12</b>
Conceptos iniciales	12
Lenguaje de programación	14
Lenguaje natural	14
Pseudocódigo	15
Lenguaje de programación práctico	17
Plataforma interactiva	18
Conceptos	20
Ejercicios	20
Aula virtual	20
Tutorial	22
Vistas	22
<b>Metodología de Desarrollo</b>	<b>29</b>
Conceptos iniciales	29
Lenguaje de programación	31
Componentes	31
Lenguaje natural	31
Clasificación	33
Limpieza	35
Stemming	36
Traducción y formato	37
Pseudocódigo	38
Analizador Léxico (Lexer)	39

---

Árbol de sintaxis abstracta (AST)	43
Tecnologías	45
Plataforma interactiva	46
Cliente	46
Tecnologías	46
Diseño	47
Servidor	48
Modelo	49
API	50
Persistencia	53
<b>Pruebas Realizadas</b>	<b>57</b>
Lenguaje Natural	57
<b>Discusión</b>	<b>61</b>
<b>Conclusiones</b>	<b>63</b>
<b>Anexos</b>	<b>65</b>
Especificación del Pseudocódigo	65
Variables	65
Condicionales	65
Sentencias	66
Bloques	66
Operadores	66
Natural Language Processing	66
Clasificación de Textos	66
<b>Bibliografía</b>	<b>68</b>

## Introducción

Durante las últimas dos décadas, las computadoras remodelaron todos los aspectos de la vida en sociedad. Desde el trabajo hasta la vida doméstica, desde una oficina hasta el medio del campo, máquinas extremadamente potentes que caben en la palma de la mano intervienen en prácticamente todas nuestras actividades diarias en el presente

Con un hardware cada vez más potente y más accesible, está cobrando más importancia que nunca el *software*, y en consecuencia, el *desarrollo de software*. En un mercado que previsiblemente seguirá creciendo de manera exponencial, la oferta de empleos sigue siendo mucho mayor que la oferta de mano de obra capacitada, ante un área profesional que sigue siendo un misterio para la mayoría de la gente: La informática.

La programación es la piedra angular del desarrollo de software, y es quizás, a ojos de inexpertos, un arte oscura que se encuentra en algún lugar entre la matemática y la criptografía. Lejos de ello, la programación es en realidad una disciplina que pone en práctica la lógica, y requiere de un análisis metódico de los problemas para alcanzar una solución. Visto así, es comprensible que muchos expertos en educación consideren que la programación debería formar parte de los contenidos básicos escolares:

*¿Es beneficioso aprender programación? Dado el poco progreso en las clases de programación típicas, puede parecer que dichas clases no ofrecen ningún beneficio a los estudiantes. Pero tal conclusión es prematura. Las clases ejemplares sí proveen beneficios considerables a los estudiantes en tanto que desarrollan habilidades de resolución de problemas complejos en una disciplina particular. En este sentido, los cursos de programación introducen a los estudiantes a una cadena de logros muy prometedora. (Linn y Dalbey, 1989).*

A pesar de la existencia de abundante material gratuito en internet (en forma de manuales, tutoriales y videos), el primer contacto con la programación es casi siempre un tópico “tabú”, y buena parte del contenido accesible para los interesados en aprender sobre el tema asume un nivel importante de conocimientos técnicos previos.

El objetivo del presente proyecto es diseñar e implementar una solución para posibilitar el primer contacto de un usuario con la programación, sin requerir de ningún tipo

de conocimiento específico previo. A partir de esta solución, la propuesta es demostrar mediante ejemplos y prácticos sencillos que un lenguaje de programación es, a fin de cuentas, un lenguaje, y más aún: un lenguaje razonablemente similar al nuestro. Si bien la propuesta contempla como usuarios finales a jóvenes y adolescentes, la solución final no discriminará de manera alguna el posible uso por parte de adultos mayores.

El proyecto que se describirá a lo largo de este informe se divide conceptualmente en dos partes: la primera es el diseño e implementación de un conjunto de herramientas que permiten interpretar un texto escrito en lenguaje natural, y “traducir” el mismo hacia un lenguaje de programación. La segunda parte consta de una plataforma interactiva que, haciendo uso de las herramientas antes descritas, propone una serie de ejemplos y ejercicios guía para que el usuario interactúe de manera autodidáctica.

En el informe trataremos, primeramente, los antecedentes de este proyecto, centrados en desarrollos anteriores accesibles al público, como lenguajes de programación diseñados para niños, plataformas web para el aprendizaje de lenguajes de programación populares, plataformas de programación mediante gráficos y cuadros conceptuales, etcétera.

Posteriormente, describiremos el desarrollo de la solución según fue registrado en cada una de sus etapas, abarcando primero las justificaciones teóricas de las decisiones tomadas en el diseño de la solución, y luego detallando las metodologías y herramientas empleadas en la implementación.



## Antecedentes

Antes de proseguir, consideramos pertinente hacer una aclaración: En el mundo de la programación, desde su nacimiento, se ha tendido naturalmente al diseño e implementación de lenguajes de programación de cada vez más “alto nivel”; esto es, cada vez menos similares al código binario o a las instrucciones en Assembler, y más similares al lenguaje empleado entre personas. Como es de esperarse, han existido varios proyectos, en general sin mayor éxito, de desarrollar un lenguaje de programación de uso práctico lo más similar posible al lenguaje natural (aquí por “uso práctico” entendemos que debería poder ser empleado en el desarrollo de software profesional, como una herramienta de trabajo). Sin embargo, hasta el día de hoy, parece existir una suerte de límite máximo en el “nivel” del lenguaje, el cual una vez cruzado comienza a disminuir progresivamente el valor práctico de dicho lenguaje como herramienta.

Desde el punto de vista de una persona con conocimientos previos de programación, es fácil describir este problema: Una función o serie de comandos mínima que realiza una determinada actividad puede ser escrita en unas pocas líneas de la mayoría de los lenguajes de programación comúnmente usados, incluso en un set relativamente pequeño de instrucciones en Assembler. Ahora bien, si esa misma función tuviera que ser descrita simplemente en forma de prosa, probablemente multiplicaría varias veces su extensión, y además, por las características propias del lenguaje empleado en las comunicaciones interpersonales, sumaría ambigüedad, lo cual es un problema mayor para trabajar con máquinas determinísticas como lo son las computadoras.

Contemplando lo anterior, realizamos la siguiente aclaración: Nuestro propósito en el diseño e implementación del *lenguaje de programación natural* propuesto no es el de crear un nuevo lenguaje de programación que reemplace a los comúnmente usados hoy en el desarrollo de software, sino crear un lenguaje didáctico que permita hacer una primera aproximación conceptual a la programación y permita, si así lo desea el interesado, progresar luego a lenguajes de programación de uso práctico.

La primera instancia de la fase de investigación a la hora de preparar el proyecto fue, lógicamente, comprobar que el desarrollo propuesto en un principio no fuera una reinención

---

la rueda. A lo largo de las semanas iniciales del proyecto, pero también durante etapas posteriores, encontramos un número relevante de proyectos que implementaron hasta cierto punto características individuales de las propuestas en la solución trabajada en este informe.

Estos proyectos fueron o son en su mayoría llevados adelante por individuos o agrupaciones de manera abierta y gratuita (*open source*), y son en casi todos los casos esfuerzos de investigación/experimentación que tienen un uso muy reducido en la práctica. Otros, por ser llevados por organizaciones con recursos disponibles, han tenido un éxito relativo en cuanto a la puesta en uso en ámbitos académicos y cursos online.

A continuación, se enumeran y describen brevemente los proyectos más destacables e interesantes que se intersectan en algún punto con la solución tratada más adelante en este informe.

## Logo

*Logo* es un lenguaje de programación diseñado en 1967 con el objetivo principal de ser empleado como lenguaje didáctico. Posee un conjunto relativamente reducido de comandos y operaciones predeterminadas que permiten generar un output gráfico sencillo para plasmar los resultados. Es el lenguaje pionero en el campo de lenguajes y plataformas de programación didácticos, aunque, dada su antigüedad, su uso práctico actual y la comunidad que lo rodea son muy pequeños.

Por ser un lenguaje basado en *Lisp*, pese a tener un número de comandos reducidos y tendientes a simplificar la gramática, no es necesariamente muy intuitivo ni fácil de poner en uso inicialmente, por lo que no encontramos sentido en realizar un análisis comparativo más detallado con la solución propuesta en este desarrollo. Sin embargo, para hacer justicia, cabe recordar que Logo fue diseñado hace ya medio siglo.

## Scratch

*Scratch* es un lenguaje de programación visual desarrollado por el *Massachusetts Institute of Technology*. Este lenguaje no se presenta en forma de texto, sino en forma gráfica a través de “bloques” que se ensamblan, representando comandos y sus agrupaciones, a través

de una interfaz gráfica web que produce un output también gráfico (representando el movimiento de personajes, texto, y demás).

Desde nuestro punto de vista, existen dos grandes factores limitantes en el abordaje que *Scratch* hace a la programación:

Primero, su forma de programación “visual”, que no se condice con la mayoría de lenguajes de programación de uso práctico que consisten en texto plano; si bien puede ser atractivo (particularmente para el público predominantemente infantil al que la plataforma apunta), también puede suponer un salto conceptual importante a la hora de pasar a otro lenguaje de programación de formato textual.

Segundo, su presentación como plataforma: Por el diseño de la plataforma, así como por la dificultad y variedad de ejercicios propuestos, queda en claro que *Scratch* apunta básicamente a estudiantes de edad escolar primaria. Si bien no establece ningún tipo de limitaciones o impedimentos al uso de estudiantes mayores y adultos, ciertamente tampoco propone ningún atractivo, especialmente desde su diseño lógicamente comparable al de los típicos manuales escolares.

## Codecademy

*Codecademy* es una plataforma web interactiva que permite al usuario seguir “cursos” en forma de ejercicios de dificultad incremental, a través de los cuales se lo guía en el uso de un lenguaje en particular.

Si bien la plataforma ofrece cursos en variados lenguajes de programación y dificultades, la mayor parte de dichos cursos requieren de una suscripción paga. Pese a esto, es una alternativa razonablemente atractiva a la hora de iniciarse en la programación, esencialmente por su interactividad y la capacidad de recibir un feedback instantáneo sobre el código escrito

Sin embargo en muchas ocasiones, dada la falta de un trasfondo más conceptual, el usuario puede encontrarse resolviendo ejercicios básicamente calcando las respuestas que dan las guías de la propia plataforma, o simplemente reescribiendo código hasta acertar, sin aplicar el análisis lógico de los problemas en el que se basa realmente la programación.

## Descripción

### Conceptos iniciales

En esta sección procederemos a presentar y describir la solución propuesta en base al objetivo inicial, considerando los antecedentes observados en la sección anterior. Dicho objetivo, como ya fue mencionado, consiste en el desarrollo conjunto de un lenguaje de programación didáctico y de una plataforma interactiva para hacer uso de dicho lenguaje de manera simple e intuitiva, todo esto a modo de facilitar el primer acercamiento conceptual hacia la disciplina de la programación. Abordaremos los componentes de la solución propuesta dividiendo esta sección en dos sub-secciones: La primera, enfocada en el lenguaje de programación; la segunda, enfocada en la plataforma interactiva.

### Lenguaje de programación natural

Como fue descrito en el objetivo, la meta principal es la implementación de una herramienta capaz de hacer una “traducción” de un texto en forma de lenguaje natural hacia su equivalente en código.

En primer lugar, el tratamiento de lo que se conoce como lenguaje natural -es decir, el lenguaje empleado comúnmente para la comunicación interpersonal- se vincula a una rama de la informática conocida como *Natural Language Processing* o *Procesamiento de Lenguaje Natural* (de aquí en más, *NLP*). Las tareas asociadas a *NLP* están, en general, íntimamente relacionadas con implementaciones de inteligencia artificial, lo cual ya da una medida del grado de complejidad que puede ser asociado a soluciones que incorporen subrutinas de este tipo.

Como primera aproximación, se entiende que el proceso a seguir, desde la mayor abstracción, es:

***Lenguaje natural* → *Lenguaje de máquina*,**

entendiéndose por lenguaje de máquina al *Assembler* o *lenguaje ensamblador*, interpretado a bajo nivel por la computadora. Dado que la interpretación de un lenguaje de mayor nivel a

lenguaje ensamblador supone una tarea importante, y que queda claramente fuera del alcance de este proyecto, la alternativa más próxima que discernimos fue la siguiente:

***Lenguaje natural* → *Lenguaje de programación práctico*,**

entendiendo por *lenguaje de programación práctico* a cualquier lenguaje preexistente ya aceptado y especificado, y de uso común.

La “traducción” hacia un lenguaje de programación predeterminado presenta, en este caso, dos ventajas para el proyecto: Primero, resuelve el gran problema que de otro modo hubiera representado la traducción a un lenguaje de bajo nivel como lo es *Assembler*; el lenguaje de programación ya tendrá su propio intérprete o compilador que se encargue de llevarlo hacia un lenguaje a nivel de máquina. Segundo, abre la posibilidad de generar una experiencia más didáctica, ya que el código escrito en *lenguaje natural* por el usuario podrá tener una representación real y directa en un lenguaje de programación de público conocimiento, lo cual permite realizar una comparación directa entre el “lenguaje humano” y su traducción hacia un “lenguaje de computadora”.

Sin embargo, un proceso de dos pasos (*lenguaje natural* → *lenguaje de programación práctico*) continúa siendo una sobre-simplificación de un conjunto complejo de subprocesos, particularmente los relacionados al tratamiento del lenguaje natural. Además de esto, las características propias de todos los lenguajes naturales presentan serios inconvenientes a la hora de interpretarlo y traducirlo a un lenguaje de programación; entre estas características “perjudiciales” la más destacable es, quizás, la ambigüedad.

En el “lenguaje de máquina” no se da lugar a la ambigüedad. Todo comando debe poder ser interpretado unívocamente para que un programa elabore resultados consistentes. Esto representa una clara dificultad a la hora de hacer el salto de un lenguaje natural, fuertemente dependiente del contexto y librado a ambigüedades e interpretaciones subjetivas, hacia un lenguaje mucho más determinístico. Para tratar esto, proponemos un tercer elemento intermedio en el proceso de traducción:

***Lenguaje natural* → *Pseudocódigo* → *Lenguaje de programación práctico*.**

Como última observación, cabe mencionar que este proceso fue ideado también teniendo en cuenta la extensibilidad del proyecto a futuro: Si bien, para el desarrollo tratado en el presente informe, el paso final del proceso consiste en la traducción hacia un lenguaje de

programación específico, que indicaremos más adelante, la existencia anterior de un pseudocódigo definido abre la posibilidad de generar traducciones hacia distintos lenguajes de programación prácticos. Para esto únicamente habría que desarrollar un nuevo módulo que implemente el último paso del proyecto (*Pseudocódigo* → *Lenguaje de programación práctico*) hacia el lenguaje deseado.

### **Lenguaje natural**

El trabajo realizado sobre el lenguaje natural es, quizás, el aspecto central del proyecto. Por lenguaje natural entendemos al idioma, hablado o escrito, empleado entre humanos para la comunicación. Para el alcance de este proyecto, reducimos esa definición, más estrictamente, al idioma *español y escrito*.

El *Procesamiento del Lenguaje Natural* (*PLN* o *NLP*, por *Natural Language Processing*) es un campo de las ciencias de la computación que estudia y trabaja sobre las interacciones entre el lenguaje natural y las computadoras. En particular, respecto al alcance de este proyecto, nos interesan las herramientas de *NLP* que trabajan sobre el reconocimiento de significado sobre el lenguaje natural, permitiendo a humanos comunicarse con una computadora o darle comandos a través de, justamente, lenguaje natural.

Abordando esto más específicamente, el desarrollo necesario, según lo introducido en la sección anterior, es el de un módulo que a partir de *lenguaje natural* (o una expresión textual lo más similar posible al lenguaje natural español escrito) realice una traducción hacia el pseudocódigo especificado; este desarrollo, previsiblemente, será el más complejo del proyecto, implicando la implementación de técnicas y métodos propios de la Inteligencia Artificial. Dentro del campo de *NLP*, el problema está vinculado de manera más enfocada con lo que se conoce como *Comprensión del Lenguaje Natural* (*CLP* o *NLU*, por *Natural Language Understanding*), que trata el problema de interpretar una instrucción o una serie de instrucciones para la computadora a partir de un texto natural, lógicamente de manera automatizada y resuelta por la propia computadora. Por su complejidad, *NLU* es considerado un problema *IA-completo* o *IA-duro*:

*(...) estas amplias áreas pueden ser consideradas IA-completas, en el sentido de que la solución del problema del área es equivalente a la solución completa del problema de la IA: producir un programa computacional inteligente en términos generales. (Shapiro, 1992).*

### **Pseudocódigo**

Lo que comúnmente se conoce como *pseudocódigo* es un lenguaje de programación definido informalmente, para uso didáctico en el aprendizaje de algoritmos y conceptos básicos de programación. El pseudocódigo, por no ser un lenguaje estandarizado ni formalmente definido, tiene ciertas libertades que le permiten asemejarse más al lenguaje natural, actuando como híbrido entre éste y los lenguajes de programación reales. Esto es así con el objetivo de que dicho pseudocódigo sea comprensible para el observador sin que éste requiera conocer previamente ninguna sintaxis demasiado específica, como sí debe hacerlo con los lenguajes de programación reales.

Si bien no existe un estándar ni una definición “de facto” de pseudocódigo comúnmente usada, sí existen documentos redactados al respecto por autoridades e instituciones académicas respetables, las cuales tuvimos en cuenta a la hora de discernir entre el uso de un pseudocódigo previamente definido o la definición de uno propio.

Tras un análisis, concluimos que la mejor alternativa para el proyecto sería el desarrollo de un pseudocódigo propio. Estos fueron los factores determinantes para la toma de la decisión:

- 1) El idioma: Al igual que la mayoría de los lenguajes de programación prácticos, las definiciones de pseudocódigo realizadas por autoridades e instituciones académicas hacen uso del idioma inglés. Así como el informe, el desarrollo del proyecto se dispuso a ser realizado en español, por lo cual sería necesaria, al menos, la traducción de una definición de pseudocódigo a palabras clave en idioma español.
- 2) La sintaxis específica: Si bien el propósito fundamental del pseudocódigo es el de ser comprensible y no estar atado sintácticamente a ningún lenguaje de programación práctico en particular, buena parte de las definiciones de pseudocódigo relevadas emplean sintaxis relativamente más similares a un

lenguaje de programación que a un lenguaje natural, siendo este segundo escenario justamente lo que buscamos en el pseudocódigo a emplear en este proyecto.

- 3) La implementación del traductor *pseudocódigo* → *lenguaje de programación práctico*: Cualquiera fuera la decisión tomada al respecto del pseudocódigo a emplear (sea que se decidiera emplear una definición ya realizada, o la definición propia de un pseudocódigo), para el resto del proyecto es necesario el desarrollo de un módulo traductor de pseudocódigo al lenguaje de programación que definiremos más adelante. El diseño de un pseudocódigo propio nos daría la libertad de contemplar esto en la propia especificación de la sintaxis, facilitando el posterior desarrollo del traductor.

La implementación para el desarrollo de este proyecto comprenderá un módulo que efectúe la traducción desde un conjunto de comandos redactados según la sintaxis del pseudocódigo hacia los comandos equivalentes expresados en un lenguaje de programación práctico que, como se justificará más adelante en este informe, se determinó sea *JavaScript*.

El propio módulo, al igual que el traductor de lenguaje natural, fue también implementado en *JavaScript*, y, también como dicho traductor, en forma de módulo de *Node.js*. Dicho módulo, publicado bajo el nombre *pseudo-js*, presenta en su interfaz dos métodos: *compileToSyntaxTree* y *compileToJS*. Ambos métodos reciben como parámetro una cadena de caracteres. Como se puede deducir por el nombre de cada método, *compileToSyntaxTree* recibe una porción de pseudocódigo y devuelve (en forma de *JSON*) el *Abstract Syntax Tree* de *JavaScript* correspondiente a la interpretación de dicho pseudocódigo; por su parte, *compileToJS* devuelve en forma de cadena de caracteres el código JavaScript resultante de la interpretación.

La definición y documentación completa del pseudocódigo queda adjunta como anexo

<sup>1</sup>.

---

<sup>1</sup> Ver Anexos - Especificación del Pseudocódigo.



### Lenguaje de programación práctico

A la hora de escoger un lenguaje de programación real para la implementación del traductor *pseudocódigo* → *lenguaje de programación* contemplado en el alcance del proyecto, se plantearon varios requerimientos:

- 1) El lenguaje debería ser relativamente flexible en cuanto a su sintaxis y a las buenas prácticas asociadas. Los lenguajes de paradigma *orientado a objetos* son particularmente difíciles de inferir desde una descripción textual, por ser también los que más abstracción conceptual aplican en su desarrollo; pensando de manera inversa (es decir, considerando el texto natural a partir del cual se llegará al código final), es muy difícil plasmar textualmente conceptos fundamentales de la *POO* como las clases, los objetos, las herencias, interfaces, y demás. Si bien sería posible, es innecesariamente complicado y se aleja del foco de desarrollar un conjunto de herramientas sencillas e intuitivas para realizar un primer acercamiento a la programación. Por este motivo, lenguajes con una sintaxis fuertemente vinculada a conceptos de *POO*, como Java, quedan descartados para el alcance de este proyecto.
- 2) El lenguaje no debería ser fuertemente tipado, y las formas de declaración de variables y funciones deberían ser reducidas o unívocas.
- 3) En lo posible, el lenguaje debería ser popular y de uso masivo, para asegurar la existencia de mayor contenido documental, tutoriales, manuales y comunidades relacionadas, en caso de que el usuario de la plataforma buscase luego interiorizarse en el mismo.
- 4) Los participantes del desarrollo del proyecto deberían tener conocimiento previo del lenguaje, para evitar añadir dificultad al desarrollo.

Cumpliendo en la mayor medida posible los requerimientos anteriores, el lenguaje definido para el desarrollo del proyecto es *JavaScript*.

*JavaScript* es un lenguaje débilmente tipado, con variables que no requieren definición previa del tipo, y que puede ser definido y modificado durante la ejecución del programa. Tiene una sintaxis relativamente sencilla y con prácticas similares a la de varios otros

lenguajes de programación (como la separación de líneas mediante el carácter “;”, la irrelevancia del espaciado más que como separador de palabras, la agrupación de código en bloques mediante caracteres reservados -los corchetes “{” y “}”-, etc.), y, además, se lo considera un lenguaje *multi-paradigmas*, ya que posee características propias de paradigmas como el *orientado a objetos* y el *funcional*, pero ninguna de ellas es imprescindible para la redacción de código, particularmente para casos de uso sencillos<sup>2</sup>. Un conjunto reducido de líneas de Javascript es “auto-contenido”, en términos de que pueden, por ejemplo, ser ejecutadas de manera directa por cualquier intérprete de Javascript sin necesitar estar englobadas en una función o en una clase (como contraejemplo a esto, podemos pensar en Java: mínimamente es necesario declarar una clase, dentro de ella un método específico que contenga los comandos, y luego especificar a la VM de *Java* qué método de qué clase ejecutar al iniciar el programa).

## Plataforma interactiva

Al igual que las herramientas para la traducción de lenguaje natural a pseudocódigo y, luego, a un lenguaje de programación práctico, el diseño de una plataforma interactiva para que el usuario haga uso de ellas es central en el desarrollo del proyecto. Por tratarse de una solución didáctica, enfocada principal pero no únicamente en usuarios jóvenes, la plataforma debería ser sencilla, intuitiva, y con un diseño gráfico y experiencia de usuario atractivos y que permitan hacer un uso auto-guiado de la misma.

En primera instancia, fue necesaria la definición del formato de dicha plataforma. Dando por descontado que la plataforma debería ser ejecutable en una computadora, y descartando en el alcance el desarrollo de aplicaciones para celulares, tablets y otros dispositivos, surge la disyuntiva entre la implementación de una aplicación nativa o la implementación de una aplicación web.

Entre estas alternativas, la de una aplicación nativa / *desktop* y la de una aplicación web, varios factores inclinan la balanza en favor de la alternativa web:

---

<sup>2</sup> Ver Anexos - *Javascript*

- 1) Mayor experiencia en desarrollo web en contraposición a desarrollo de aplicaciones nativas por parte de quienes realizamos el proyecto.
- 2) Desarrollo unificado multi-sistemas: Una de las mayores ventajas del desarrollo web es la abstracción respecto del navegador, sistema operativo y hardware sobre los cuales se ejecuta la aplicación. Dada la estandarización del formato de los sitios web, una misma web debería funcionar en distintos navegadores sobre distintos entornos, sin necesidad de contemplar mayores factores específicos a cada entorno en el desarrollo.
- 3) Estándar *de facto* de acceso a contenido en la actualidad: Internet y, con él, la web, se han convertido progresivamente desde finales del siglo pasado en el principal medio de acceso a todo tipo de contenidos desde computadoras y, más recientemente, dispositivos móviles como los smartphones. A fines de marzo de 2017, se estima que, a nivel global, 49,6% de la población tiene acceso estable a una conexión de internet. Más específicamente, en Argentina, la cifra asciende a 79,4% (Internet World Stats, 2017). En nuestro país, a modo de ejemplo, casi 30 millones de personas están registradas en Facebook (esto es un 66,2% de la población). Recordamos aquí a modo anecdótico pero con mucha importancia que esta explosión en el uso de internet se ha dado principalmente apenas en las últimas dos décadas; y más aún: Facebook sólo existe desde 2004. Por la experiencia propia notada a diario como usuarios y desarrolladores de software, y por datos estadísticos como los aquí presentados, se puede asegurar que el uso masivo de aplicaciones vinculadas a internet y la web es más que una tendencia pasajera, y varios desarrollos en el área por parte de las principales empresas de desarrollo en el mundo lo confirman<sup>3</sup>.

Considerando los puntos enumerados, optamos por desarrollar una plataforma interactiva en forma de aplicación web que permita a los usuarios hacer uso de las demás herramientas componentes del paquete final de la solución propuesta.

---

<sup>3</sup> Ver Anexos - *El futuro de la web*.

Habiendo determinado que la plataforma para el usuario será desarrollada como una web, queda determinar el formato de dicha web, tanto conceptual como gráficamente.

### **Conceptos**

Se realiza a continuación una breve descripción de los conceptos básicos asociados al modelo representado en la plataforma interactiva.

#### Ejercicios

Un concepto fundamental que se eligió como base de la plataforma interactiva es el de los ejercicios. Un ejercicio se compone por tres elementos: Una guía, una resolución y una respuesta. La guía del ejercicio es esencialmente un cuerpo de texto que explica los pasos del ejercicio y puede dar indicios o pistas sobre su resolución. La resolución del ejercicio es la que elabora el usuario, en forma de texto natural, dentro del campo habilitado en la web para tal propósito. La respuesta del ejercicio es prefijada, y se empleará para evaluar contra la resolución del usuario y determinar si la misma es válida o inválida. Un ejercicio es aprobado si la resolución se condice con la respuesta prefijada.

La idea de implementar ejercicios dentro de la plataforma es la de guiar al usuario a través de una serie de ejemplos de creciente complejidad, que lo iniciarán primero en los conceptos básicos del lenguaje natural, a la vez que introducirán los conceptos fundacionales de la programación, todo de una manera “auto-guiada” por el propio usuario, con la asistencia de los elementos de guía de cada ejercicio.

#### Aula virtual

Si bien por una parte, la idea de la parte web del proyecto es la de presentar una plataforma auto-didáctica, se decidió también ampliar esto con la implementación conceptual de un “aula virtual”. El usuario de la web podrá llevar adelante una registración con alguna información básica para identificarse, y luego participar de un sistema de “aulas interactivas”. Este sistema se compone por tres elementos básicos: Estudiantes, profesores y clases.

Estudiantes y profesores son, ambos, tipos de usuarios con roles distintos. Los profesores podrán crear clases, y dentro de cada clase crear ejercicios asociados

(especificando, según corresponda, una guía para cada ejercicio, así como una respuesta). Los estudiantes, por su parte, podrán “inscribirse” a las clases, y dentro de cada clase irán completando de manera lineal los ejercicios propuestos (es decir, deberán resolver cada ejercicio antes de poder pasar al siguiente).

Las clases actúan esencialmente como agrupaciones conceptuales de ejercicios, y su disposición queda a criterio de cada profesor.

A estos tres elementos básicos, se añade un sistema de revisión de ejercicios que suma al intercambio estudiante-profesor, emulando los intercambios que se producirían en un aula real. El método de avance original a través de los ejercicios pasa a tener dos pasos: Al terminar de redactar la resolución de un ejercicio, el estudiante realizará la “presentación” del mismo (esencialmente confirmando la entrega mediante algún elemento de la interfaz, como puede ser un botón). Inmediatamente al momento de presentar la resolución, la plataforma realizará un control automático preliminar contra la respuesta predefinida del ejercicio; suponiendo que se espera que el ejercicio, por ejemplo, resulte en un valor numérico de 5, la plataforma evaluará el código (en lenguaje natural) ingresado por el usuario y comprobará que efectivamente su ejecución retorne el valor 5. En caso de no conformar, el usuario recibirá alguna indicación gráfica de que la resolución es incorrecta. En caso de sí pasar este control, se informará al usuario que la resolución es, preliminarmente, correcta. Luego de este chequeo preliminar, la resolución del estudiante es enviada al profesor. Desde la interfaz, el profesor podrá revisar la resolución entregada por el estudiante de manera textual (visualizará el texto natural que hubiera redactado, con el formato en que lo hubiera hecho). En este paso de revisión, el profesor podrá hacer observaciones o comentarios sobre el código entregado, y finalmente podrá aprobar o rechazar la entrega. En caso de ser aprobada, el estudiante podrá proseguir de la misma manera con el siguiente ejercicio de la clase, en caso de haberlo. En caso de ser rechazada, el estudiante deberá revisar la resolución antes de poder volver a presentarla; si el profesor hubiera realizado comentarios u observaciones sobre la resolución, estos se mostrarán también en la interfaz del estudiante a modo de feedback (a modo aclarativo, cabe pensar en estas observaciones como en las correcciones realizadas por un profesor en cualquier entrega en formato físico).

---

## Tutorial

En paralelo a la mecánica antes mencionada, se implementará una serie de ejercicios predefinidos de desarrollo independiente a las clases de aula virtual. Cualquier usuario, sin necesidad de hacer un registro, podrá desarrollar los ejercicios del tutorial.

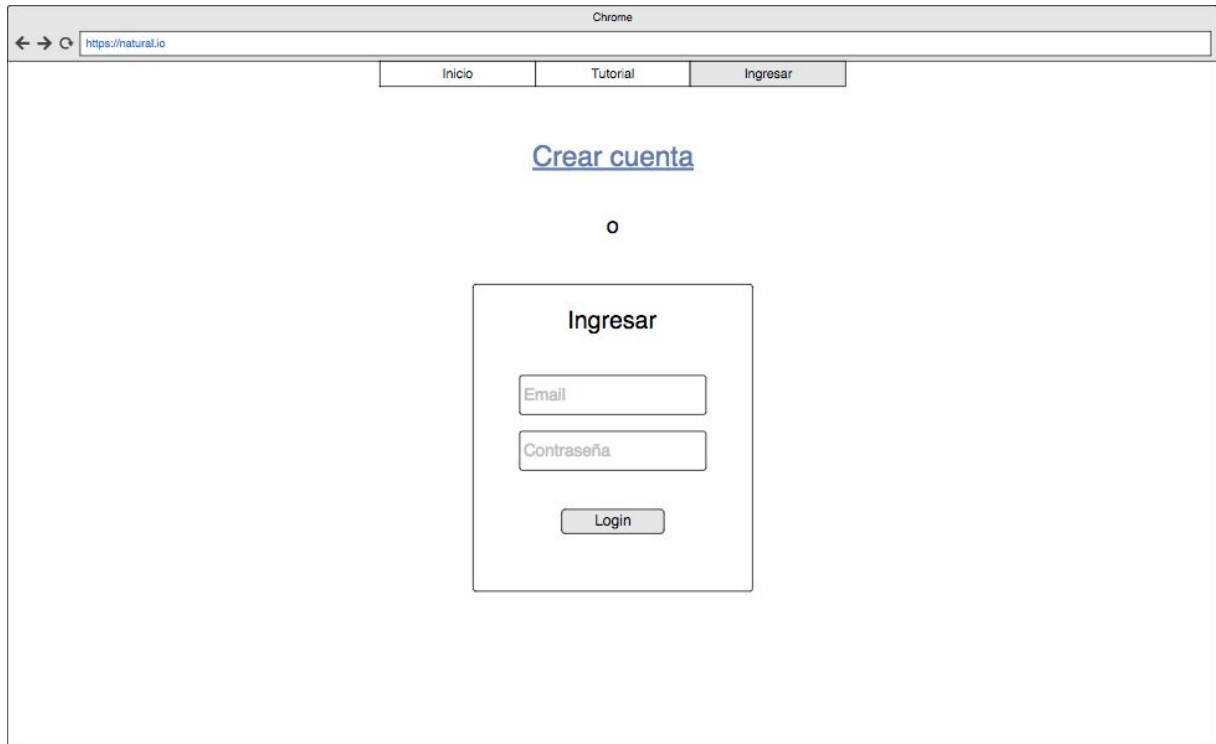
El tutorial, en esencia, funcionará con la misma mecánica que una clase como contenedor de ejercicios sucesivos. Estos ejercicios habrán sido definidos estáticamente para actuar como una introducción explicativa del manejo y uso de la plataforma interactiva y del lenguaje de programación natural.

## Vistas

A continuación se describe el esquema básico de las vistas que componen la plataforma web.

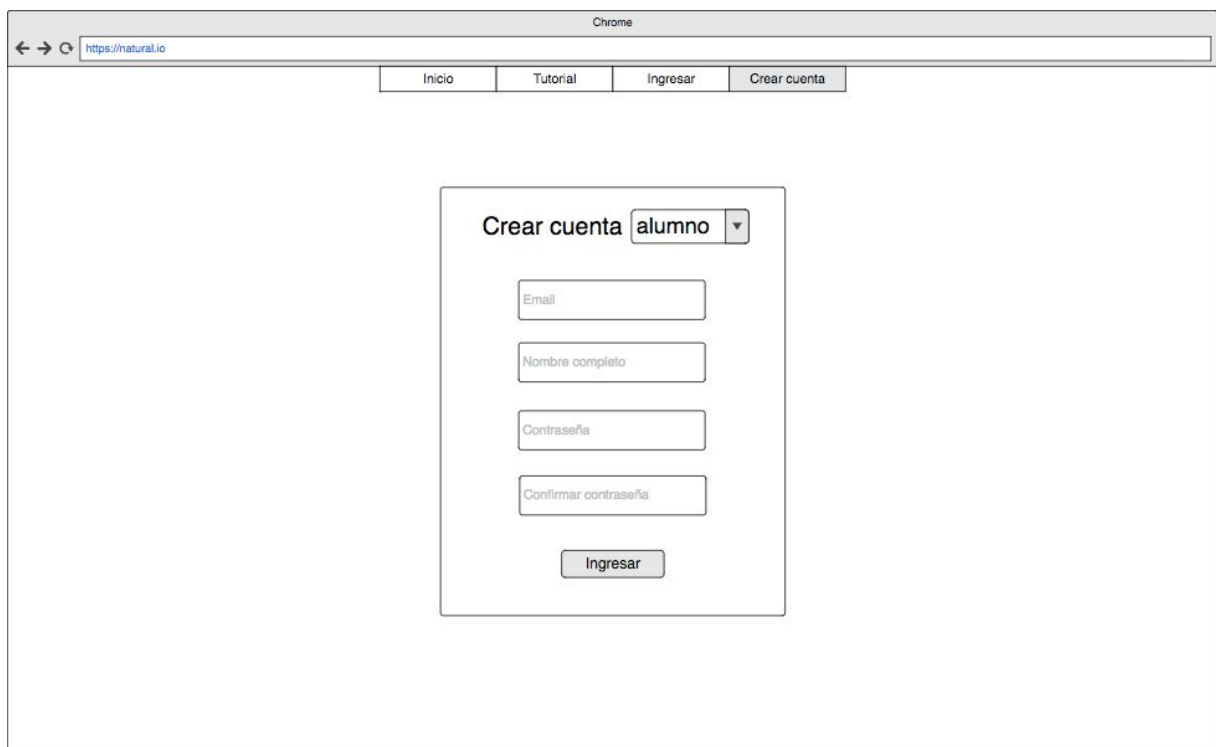


*Figura 1: Vista inicial o landing page, con imágenes y una breve descripción de la plataforma y su propósito.*



The screenshot shows a web browser window with the URL <https://natural.io>. The navigation menu includes 'Inicio', 'Tutorial', and 'Ingresar'. The main content area features a blue link for 'Crear cuenta' and a small 'o' separator. Below this is a centered box titled 'Ingresar' containing an 'Email' input field, a 'Contraseña' input field, and a 'Login' button.

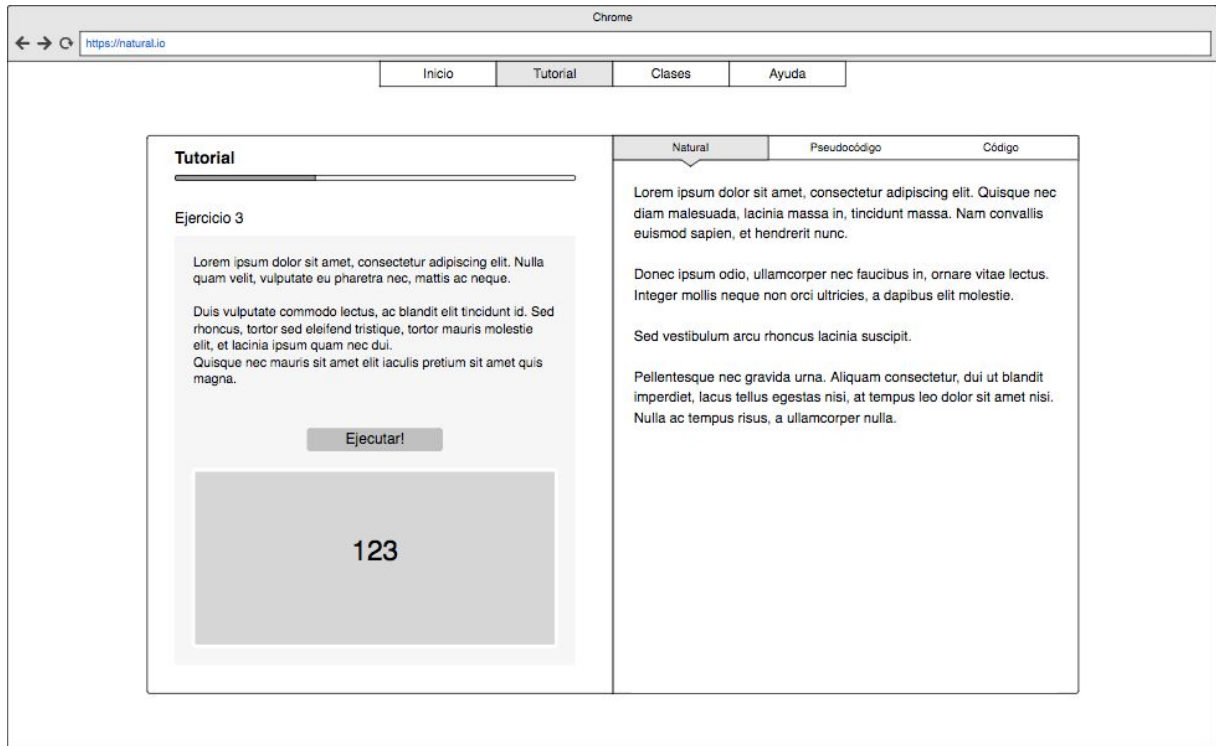
Figura 2: Vista de acceso para usuarios registrados.



The screenshot shows a web browser window with the URL <https://natural.io>. The navigation menu includes 'Inicio', 'Tutorial', 'Ingresar', and 'Crear cuenta'. The main content area features a 'Crear cuenta' label with a dropdown menu set to 'alumno'. Below this are four input fields: 'Email', 'Nombre completo', 'Contraseña', and 'Confirmar contraseña', followed by an 'Ingresar' button.

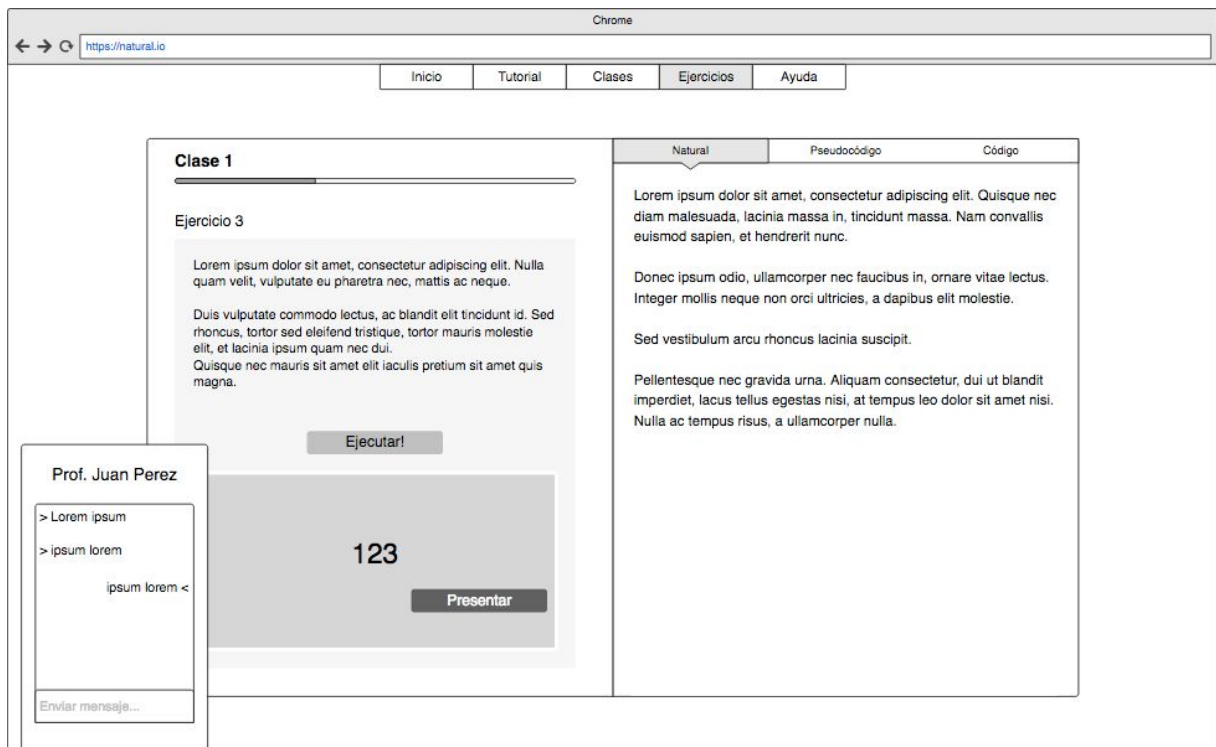
Figura 3: Vista de creación de cuenta. Al momento de realizar el registro, el usuario

*especifica su rol (profesor o estudiante), además de ingresar otros datos identificatorios: dirección de email, nombre completo y contraseña.*



*Figura 4: Vista de tutorial. De modo similar a la vista de ejercicios del estudiante, la vista de tutorial mostrará una sucesión de ejercicios introductorios prefijados para demostrar los conceptos básicos de la plataforma y del lenguaje. No es necesario iniciar sesión para hacer uso de esta vista.*





*Figura 5: Vista de ejercicios para el estudiante. Aquí, tras seleccionar la clase, verá el ejercicio en progreso de la misma (es decir, el último que aún no haya resuelto). En la sección izquierda se muestran los detalles del ejercicio, incluyendo el nombre y la descripción. Debajo de esto, un botón permite “ejecutar” el código redactado, y debajo del mismo se muestra el resultado de la ejecución. En caso de que el resultado de lo ejecutado sea correcto de acuerdo al resultado preestablecido para el ejercicio, se habilita un botón que permite al estudiante “presentar” su resolución al profesor, que luego la aprobará o rechazará con observaciones.*

*A la derecha se muestra el editor de texto, dentro del cual podrá haber un texto inicial en caso de que el ejercicio lo contenga, y sobre el cual el usuario realizará la redacción de la resolución. Cambiando hacia las pestañas “código” o “pseudocódigo”, el alumno podrá ver las traducciones automáticas del texto natural hacia el correspondiente estado. En esta vista, se habilita un chat en vivo con el profesor de la clase, en el que el estudiante podrá hacer las consultas que necesitare.*

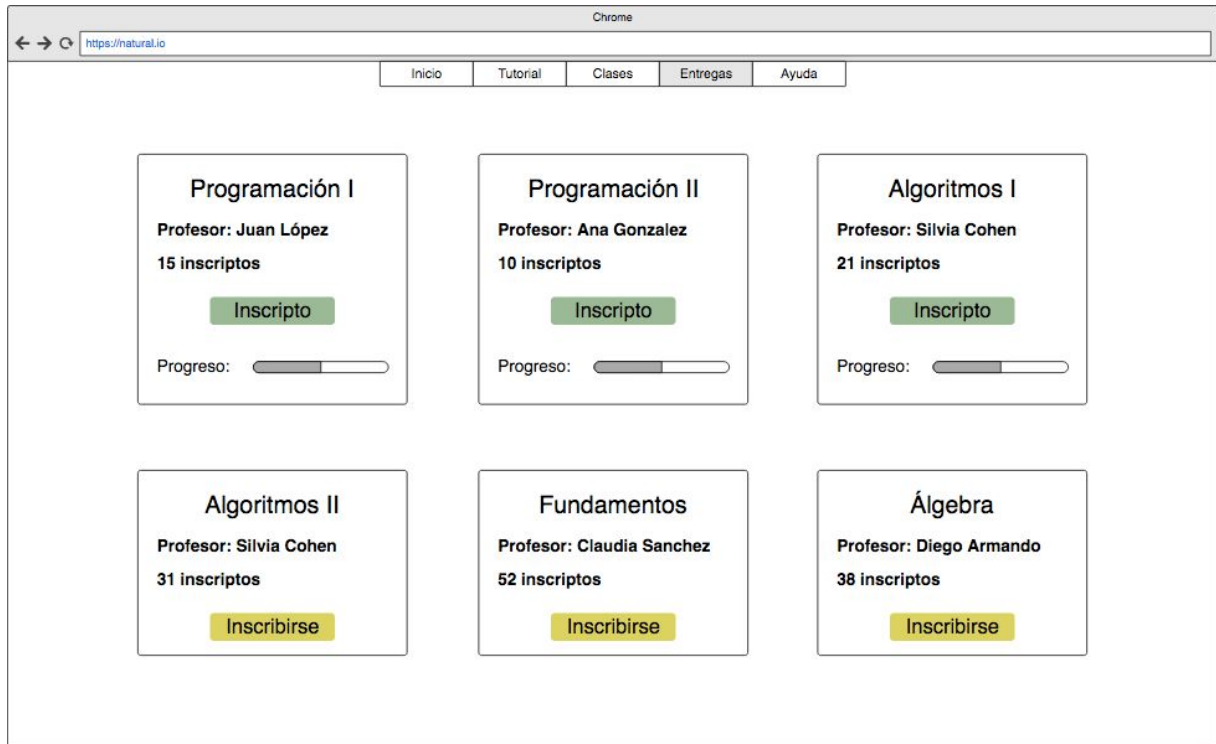


Figura 6: Vista de clases del estudiante. Aquí el usuario puede ver la lista de clases disponibles, inscribirse a ellas y visualizar su progreso en las que ya estuviera inscripto.

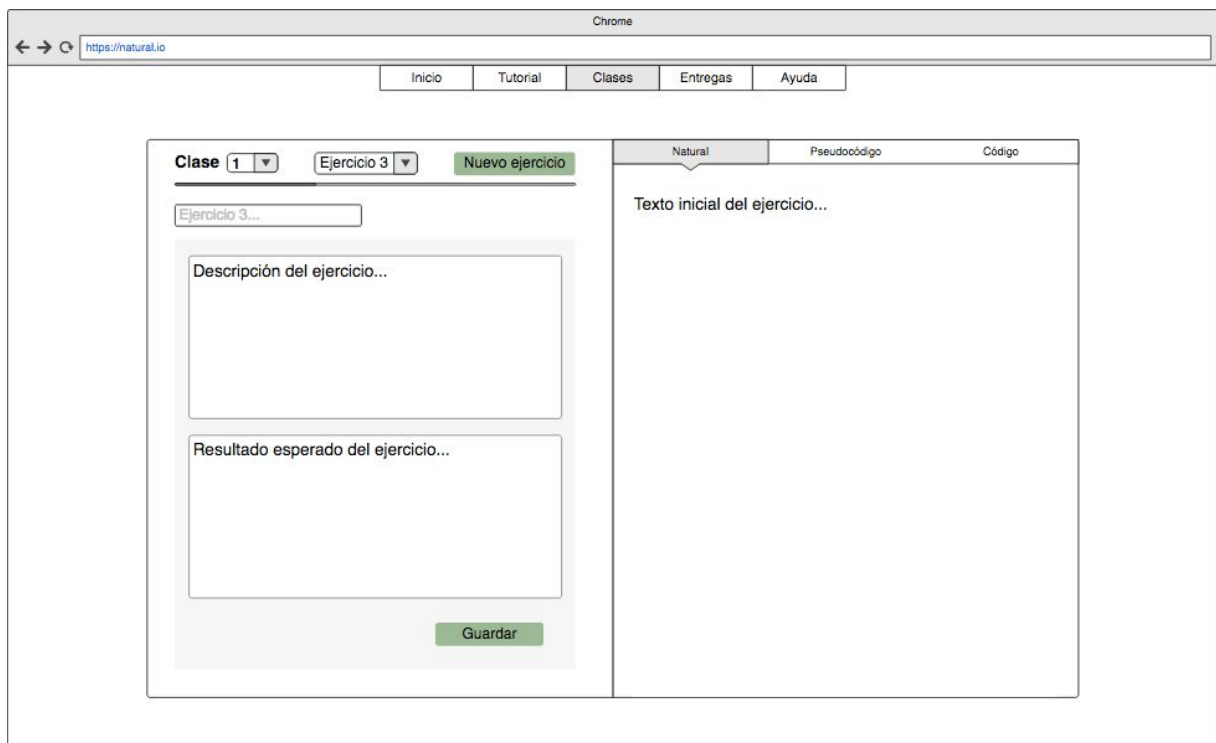
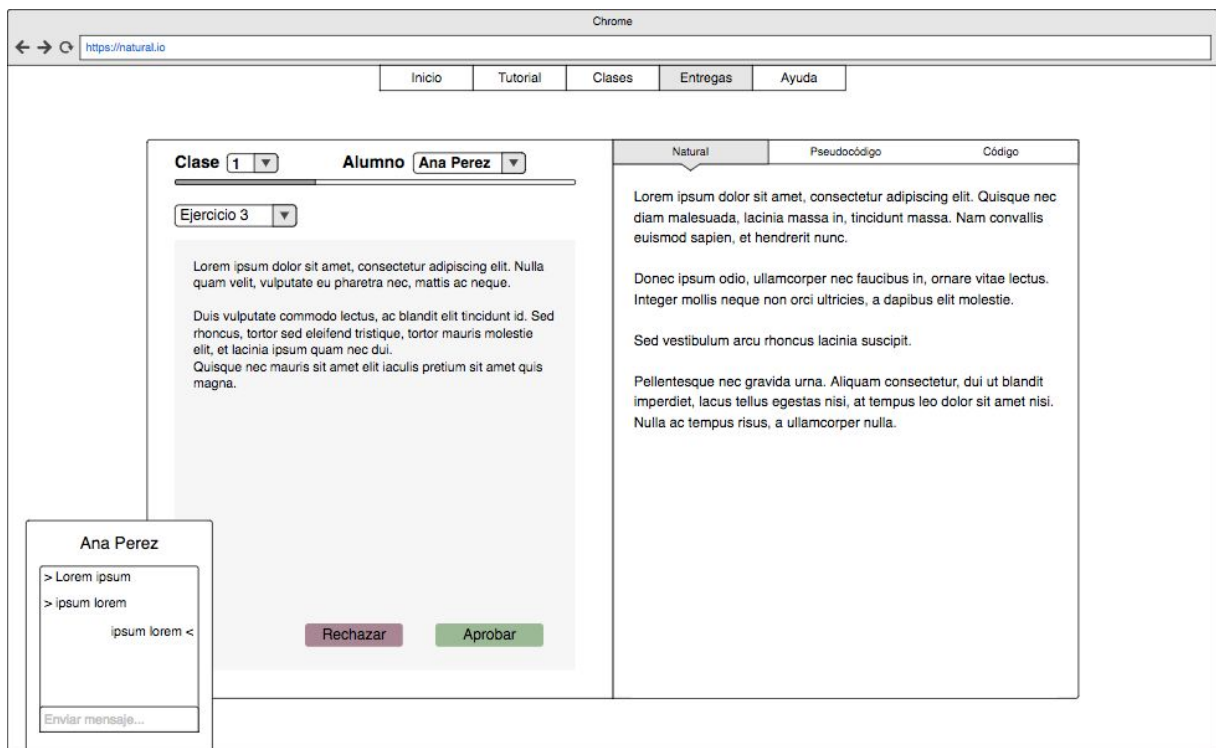
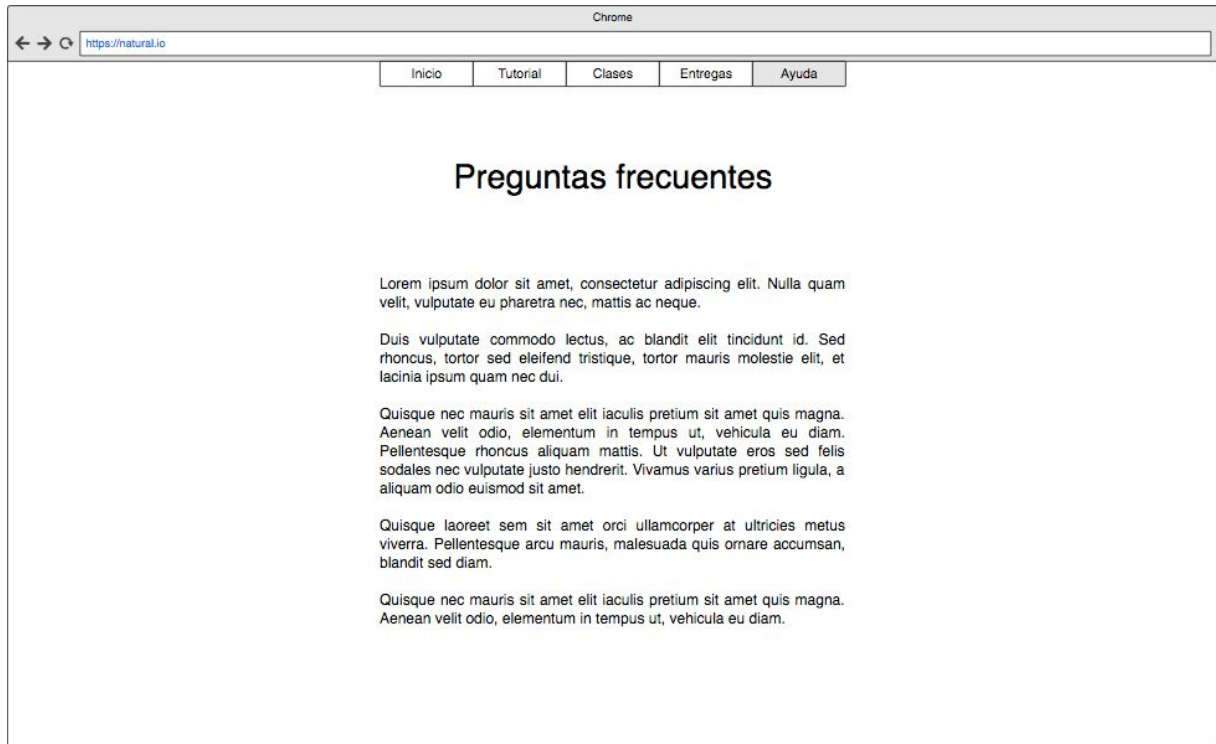


Figura 7: Vista de clases del profesor. En esta vista el profesor puede crear nuevos ejercicios

*y asignarlos a la clase correspondiente. De manera similar a la vista de ejercicios final del estudiante, en la sección izquierda se definen el nombre, la descripción y el resultado esperado del ejercicio. En la sección derecha, puede introducir el código inicial a mostrar al iniciar la resolución del ejercicio.*



*Figura 8: Vista de entregas del profesor. En esta vista, tras seleccionar clase y estudiante, el profesor puede revisar las entregas de ejercicios que dicho estudiante haya realizado, aprobar o rechazar la última. Además de visualizar el código a la derecha, puede editarlo y agregarle comentarios a modo de devolución para el estudiante, más allá de que el mismo estuviera o no aprobado. También se muestra aquí una ventana de chat con el estudiante en cuestión.*



*Figura 9: Vista de ayuda, con respuestas a posibles preguntas frecuentes y otras aclaraciones complementarias al tutorial.*

## Metodología de Desarrollo

### Conceptos iniciales

Previamente a entrar en detalle al respecto de las metodologías y herramientas empleadas en el desarrollo de cada uno de los componentes de la solución final propuesta en el presente proyecto, haremos una salvedad respecto a la decisión de una herramienta en común al desarrollo de todos los módulos.

Teniendo en cuenta la selección anterior de Javascript como lenguaje de programación final del proceso de traducción *lenguaje natural* → *pseudocódigo* → *lenguaje de programación práctico*, y considerando que quienes llevamos adelante este proyecto tenemos suficiente experiencia con dicho lenguaje, tanto por trabajos académicos anteriores como por experiencia laboral, concluimos que sería conveniente la implementación de la totalidad del proyecto en dicho lenguaje.

En los últimos años, la comunidad de desarrolladores y herramientas alrededor del *JavaScript* ha crecido de manera muy importante. Sólo en los repositorios públicos alojados en *GitHub*, por ejemplo, *JavaScript* es el lenguaje más utilizado en la implementación de proyectos de todo tipo, dejando segundo al anterior lenguaje más popular, Java, con menos de la mitad de proyectos<sup>4</sup>. Con este gran *ecosistema* alrededor del lenguaje, existe una gran cantidad de librerías, herramientas y soluciones de código abierto a disposición de cualquier programador para simplificar la implementación de una aplicación.

El desarrollo de la plataforma interactiva de este proyecto se decidió llevar adelante en forma de web, y el lenguaje de programación *de facto* en web es *JavaScript* hace décadas. Pero, sumado a esto, más recientemente han surgido tecnologías como *Node.js*, que permiten implementar soluciones en *JavaScript* más allá de la web. Con herramientas de este estilo, una aplicación web puede hoy desarrollarse enteramente (tanto *backend* como *frontend*) en el mismo lenguaje de programación.

---

<sup>4</sup> Estadísticas publicadas por GitHub en la recopilación *GitHub Octoverse 2016*.

Tras todas las observaciones mencionadas, concluimos que la mejor alternativa entre lenguajes de programación con los cuales implementar el proyecto propuesto sería *JavaScript*, lo cual simplificará enormemente el *stack* de tecnologías empleadas en total.

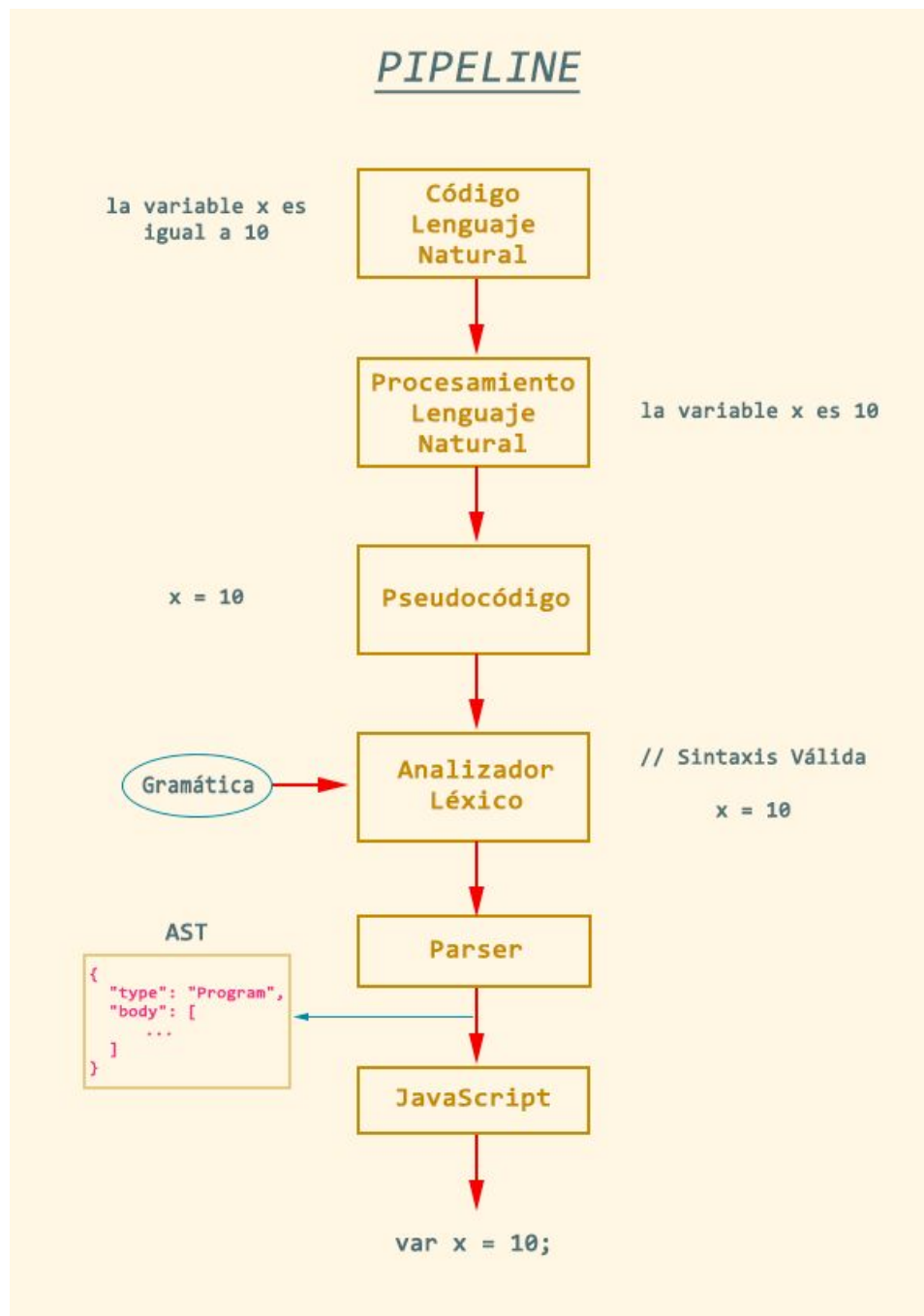


Figura 10: Representación visual del proceso de traducción de lenguaje natural a código.

## Lenguaje de programación

A modo de demostrar de una manera más práctica los procesos realizados para transformar el lenguaje natural en código, se propone a continuación un ejemplo cuyo procesamiento se irá siguiendo en el desarrollo de esta sección.

*Inicialmente se sabe que lista de alumnos contiene a "Juan", "Florencia", "Juan" y "Ana". Además, la variable `juan` es igual a 0, y también la variable `otros` es igual a 0.*

*Luego, por cada nombre en la lista de alumnos: si el nombre es igual a "Juan", `juan` es igual a `juan` más 1. Sino, `otros` es igual a `otros` más 1.*

*Finalmente, si `juan` es mayor a `otros`, entonces la repetición será verdadera. Sino, la repetición será falsa.*

En esta pieza de texto natural, se describe de manera comprensible un proceso sencillo que, a partir de una lista de nombres, realiza un conteo. Por cada nombre en dicha lista, si el nombre es “Juan”, suma uno a la variable `juan`. De lo contrario, suma 1 a la variable `otros`. Se verá cómo este texto sufre modificaciones progresivas en cada paso hasta el resultado final en forma de código.

## Componentes

### Lenguaje natural

La implementación del módulo traductor de lenguaje natural a pseudocódigo fue en forma de módulo o paquete para *Node.js*, publicado bajo el nombre de *prose-js* (“*prose*” en referencia a “*prosa*”, *forma de expresión habitual, oral o escrita, no sujeta a las reglas del verso* (Real Academia Española, 2014).

El módulo *prose-js*, desarrollado enteramente en *JavaScript*, publica en su interfaz un único método llamado *compileToPseudocode*. Este método recibe como input en forma de cadena de caracteres el texto natural, sobre el cual realizará los procesos pertinentes para devolver, nuevamente como cadena de caracteres, el pseudocódigo final interpretado a partir

de dicho texto; esto es así siempre y cuando no se presente algún fallo o excepción que impida completar el proceso de traducción.

El procesamiento realizado sobre la entrada del usuario al sistema, en forma de lenguaje natural, agrupa varias tareas asociadas a *NLU* que permiten alcanzar como salida del componente a una porción de pseudocódigo representativa de las instrucciones interpretadas en el texto original. Sin entrar en detalle sobre el orden exacto de la ejecución de las subtareas, dado que se ejecutan de manera entrelazada, el proceso sobre el texto puede ser descrito en los siguientes pasos:

- 1) Clasificación.
- 2) Limpieza.
- 3) Stemming.
- 4) Traducción y formato.

Para el desarrollo de los procedimientos mencionados, se empleó la asistencia de dos librerías de código abierto para *JavaScript*. Por un lado, *NaturalNode*, una librería que recopila herramientas de uso común en tareas de *NLP*, y que implementa algoritmos básicos como *tokenizers*, empleados para dividir un texto en palabras o conjuntos de palabras, y un *stemmer*. Por otro lado, se hizo uso de un componente de la librería *Salient* que facilita el *Part-of-Speech tagging (POS tagging)*, que consiste en clasificar cada palabra según su tipo (sustantivo, adjetivo, verbo, adverbio, conjunción, etc.); el *POS tagger* es esencial en el trabajo realizado sobre el texto en los pasos de clasificación y limpieza.

Cabe destacar que, fuera de la implementación básica de la lógica de los algoritmos, en varios casos debimos realizar adaptaciones al código abierto de las librerías empleadas debido a que, en su mayoría, las librerías de este estilo están diseñadas para trabajar principal o únicamente con el idioma inglés. Si bien una de ellas (*Salient*) ofrece soporte oficial para español, su rendimiento en las funcionalidades que empleamos estaba lejos de ser óptimo y era muy inferior al desempeño de las mismas funcionalidades sobre un texto en idioma inglés. Esto se debe a la fuerte relación que existe entre la implementación de soluciones de *NLP* y el idioma sobre el cual se trabajará (tanto es así que, a día de hoy, existen soluciones de este tipo que están en un nivel de desarrollo muy inferior para ciertos idiomas como el chino o algunos



dialectos del indio y el árabe, en comparación a sus contrapartes para idiomas como el español o el inglés).

El desarrollo del resto de los procedimientos y la articulación de los mismos fue enteramente propio.

A continuación, se describen los procesos realizados dentro de cada uno de los pasos de procesamiento del lenguaje natural.

#### Clasificación

Un *clasificador lineal* es un componente asociado a *machine learning* cuyo objetivo es identificar un objeto para determinar a qué clase pertenece. Más específicamente en lo asociado a *NLP*, los clasificadores son empleados para reconocer el significado o la intención en frases de uso común cuya interpretación se conoce previamente (un ejemplo propio de la solución implementada en este proyecto: la frase “*x es mayor o igual a 2*” o similares debería ser interpretada y traducida a pseudocódigo como “ $x \geq 2$ ”).

Existen varios tipos de clasificadores que aplican técnicas de machine learning, entre los cuales los más comúnmente usados en clasificación de textos son los que aplican *Naive Bayes* y *Regresión Logística*<sup>5</sup>, los cuales toman un set de datos de entrenamiento sobre el cual luego se basan para realizar la clasificación de las piezas de texto que se les pase. De entre estos dos, el más comúnmente usado en *NLP* es *Naive Bayes*, y comprobamos por experimentación que, con el mismo set de datos de entrenamiento, era el que mejores resultados daba en el proceso de clasificación.

El paso de clasificación es el primero del proceso dado que se busca clasificar expresiones más extensas (usualmente varias palabras u oraciones completas) y con una forma más específica, para reducirlas al equivalente deseado en pseudocódigo; como se verá más adelante, en los siguientes pasos del proceso se realiza limpieza y reducción de palabras que distorsionan el texto original, y dificultarían la clasificación de las expresiones deseadas.

Traduciendo esto al ejemplo propuesto:

---

<sup>5</sup> Ver Anexos - *Natural Language Processing*: Clasificación de textos

*Inicialmente se sabe que lista de alumnos contiene a "Juan", "Florencia", "Juan" y "Ana". Además, la variable `juan` es igual a 0, y también la variable `resto` es igual a 0.*

*Luego, por cada nombre en la lista de alumnos: si el nombre es igual a "Juan", `juan` es igual a `juan` más 1. Sino, `resto` es igual a `resto` más 1.*

*Finalmente, si `juan` es mayor a `resto`, entonces la repetición será verdadera. Sino, la repetición será falsa.*

Inicialmente, se buscan dentro del texto estructuras similares a las que el clasificador fue entrenado para reconocer. Con asistencia del *tagger POS*, que identifica el tipo de cada tipo de palabra, se identifican piezas de texto con una cantidad y tipos de palabras similares a los de las estructuras predefinidas en el clasificador.

Las piezas de texto identificadas son luego evaluadas por el clasificador, que fue inicialmente entrenado con un set de datos predefinidos. El clasificador retorna entonces una lista que contiene todas las estructuras que fue entrenado para reconocer, junto con un valor numérico entre 0 y 1. Este valor representa cuán “similar” es la pieza de texto introducida a la estructura predefinida, siendo 0 totalmente distinto y 1 exactamente igual. En caso de que esta probabilidad sea mayor a un valor definido arbitrariamente que actúa como margen de aceptación, se “acepta” a la pieza de texto en cuestión como un equivalente de la estructura correspondiente, y se pasa a reemplazar dicha pieza en el texto por dicha estructura.

En el ejemplo, “*es mayor a*” es reemplazado por la estructura equivalente “*>*”.

*Inicialmente se sabe que lista de alumnos contiene a "Juan", "Florencia", "Juan" y "Ana". Además, la variable `juan` `=` 0, y también la variable `resto` `=` 0.*

*Luego, por cada nombre en la lista de alumnos: si el nombre `==` "Juan", `juan` `+` `juan` más 1. Sino, `resto` `+` `resto` más 1.*

*Finalmente, si `juan` `>` `resto`, entonces la repetición será verdadera. Sino, la repetición será falsa.*

Se observa también aquí el porqué del uso de un clasificador en el reconocimiento de estructuras de este estilo: La misma expresión “*>*” podría ser representada de maneras

distintas en el texto natural (“*es mayor a*”, “*es mayor que*”, “*es más grande que*”, etc.). Realizar una identificación determinística para todas las posibles variantes representativas de dicha estructura, simplemente realizando una comparación directa de las piezas de texto, resultaría en un código mucho más extenso, menos prolijo y con peores resultados, especialmente considerando que habría que tener en cuenta cientos o miles de variantes posibles para identificar todas las estructuras relevantes.

#### Limpieza

Teniendo en cuenta el alcance de la solución a implementar, ciertas palabras y tipos de palabras empleados en el lenguaje natural no tienen relevancia en el proceso de “traducción” llevado adelante. Por esto, palabras como artículos, ciertas preposiciones y adverbios son removidos del texto tras el paso de *clasificación*. También se remueven ciertos signos de puntuación que no añaden significado.

Aplicando esto en el ejemplo:

*Inicialmente se sabe que la lista de alumnos contiene a "Juan", "Florencia", "Juan" y "Ana". Además, la variable  $juan = 0$ , y también la variable  $resto = 0$ .*

*Luego, por cada nombre en la lista de alumnos: si  $el\ nombre == "Juan"$ ,  $juan = juan + 1$ . Sino,  $resto = resto + 1$ .*

*Finalmente, si  $juan > resto$ , entonces la repetición será verdadera. Sino, la repetición será falsa.*

Aquí se observa la remoción de palabras y estructuras consideradas irrelevantes. En un comienzo, se eliminan adverbios como “*inicialmente*” o “*finalmente*”, junto con artículos, por no añadir un valor relevante a lo que se expresa en la oración. Algunas estructuras se identifican como irrelevantes por su posición dentro de la oración, como “*se sabe que*”, reconocida en primer lugar por la conjunción “*que*” con la que se une al resto de la oración, que sí tiene un significado relevante. Finalmente se eliminan signos de puntuación que hayan quedado erróneamente posicionados tras la eliminación de palabras; se suele dar el caso, tras

la eliminación de palabras, de que puntos y comas queden juntos por ejemplo, de manera incorrecta.

*lista de alumnos contiene a "Juan", "Florencia", "Juan" y "Ana". juan = 0, y resto = 0.*

*por cada nombre en lista de alumnos: si nombre == "Juan", juan = juan más 1.*

*Sino, resto = resto más 1.*

*si juan > resto, repetición será verdadera. Sino, repetición será falsa.*

### Stemming

El *stemming* es un método para reducir una palabra a su raíz. En la solución, esto es particularmente importante para trabajar con verbos; en el texto natural, los mismos pueden tomar distintas conjugaciones que dificultan la interpretación para generar el pseudocódigo, por lo que es importante que primero tomen su forma en infinitivo. El algoritmo más comúnmente usado y de mejores resultados para realizar stemming es el *algoritmo de Porter*.

*lista de alumnos contiene a "Juan", "Florencia", "Juan" y "Ana". juan = 0, y resto = 0.*

*por cada nombre en lista de alumnos: si nombre == "Juan", juan = juan más 1.*

*Sino, resto = resto más 1.*

*si juan > resto, repetición será verdadera. Sino, repetición será falsa.*

Nuevamente con asistencia del *POS tagger*, para identificar los verbos, se realiza el stemming de los mismos, reemplazándolos por el verbo raíz en modo infinitivo. En este caso, el verbo relevante “*será*” se intercambia por su modo infinitivo “*ser*”. A efectos prácticos y por la definición del pseudocódigo, todas las instancias del verbo “*ser*” son luego reemplazadas por “*es*”.

*lista de alumnos contiene a "Juan", "Florencia", "Juan" y "Ana". juan = 0, y resto = 0.*

*por cada nombre en lista de alumnos: si nombre == "Juan", juan = juan más 1.*

*Sino, resto = resto más 1.*

*si juan > resto, repetición es verdadera. Sino, repetición es falsa.*

#### Traducción y formato

Como último paso, se traducen determinadas palabras clave a su equivalente en pseudocódigo, y se determina el inicio y final de cada comando y cada bloque de comandos. También se identifican estructuras comunes como bloques condicionales y loops, y se los reestructura correspondientemente. Estas estructuras, por ser predeterminadas y responder a un patrón reconocible, son identificadas mediante el uso de expresiones regulares.

Adicionalmente, como paso final, se da formato al pseudocódigo final con espaciados, saltos de línea y tabulaciones para facilitar su lectura.

A continuación se muestra el pseudocódigo resultante al concluir el procesamiento del texto en lenguaje natural.

```
listaDeAlumnos = ["Juan", "Florencia", "Juan", "Ana"]
juan = 0
resto = 0
por cada nombre en listaDeAlumnos {
  si nombre == "Juan" {
    juan = juan + 1
  }
  Sino {
    resto = resto + 1
  }
}
si juan > resto {
  repetición es verdadero
}
Sino {
  repetición es falso
}
```

---

}

### Pseudocódigo

El componente de la solución relacionado al pseudocódigo consta, esencialmente, de dos partes: La primera parte cumple la función de *parser*; como tal, debe analizar una pieza de texto con una sintaxis específica (en este caso, la sintaxis del pseudocódigo definido, adjunta en el anexo), y descomponerlo en un *Árbol de Sintaxis Abstracto* o *Abstract Syntax Tree (AST)*. La segunda parte, necesaria no para la ejecución del código sino para sumar valor didáctico al proyecto, debe recomponer el *AST* generado por el módulo anterior, y retornar su representación en forma de código *JavaScript*.

Para el desarrollo de parsers de *lenguajes de dominio específico* o *domain-specific languages (DSLs)* es común el empleo de herramientas denominadas *parser generators*. En lugar de desarrollar particularmente el código de un parser completo, un *parser generator* o *generador de parser* permite definir, mediante una sintaxis específica de la herramienta, la sintaxis válida del lenguaje en cuestión y el valor que representa cada expresión, para generar luego el *AST* deseado. De entre las herramientas de este estilo disponibles para entornos de Javascript, encontramos que *PEG.js* es, además de la más popular, la más sencilla de usar y que disponía de las capacidades requeridas para el presente proyecto.

Mediante una sintaxis propia, *PEG.js* permite generar un parser a partir de la especificación del lenguaje de interés, y también da lugar a especificar el *AST* resultante de dicho *parser*. De esto, surgió la posibilidad de “saltar” el desarrollo de la segunda etapa del componente asociado a pseudocódigo: El uso de *PEG.js* dio lugar a especificar la generación de un parser para el pseudocódigo de modo tal que el mismo retornara un *AST* compatible con el estándar recibido por los principales motores intérpretes de *JavaScript*, lo cual permitiría, por un lado, la ejecución directa del código tras su parsing a un *AST* compatible, y por otro lado, la regeneración de código Javascript a partir de dicho árbol.

A través de esta forma de generar el parser para el pseudocódigo, el único añadido necesario fue el de una librería que, a partir de un *AST* compatible, genere código *JavaScript*.

A continuación, describimos los componentes que intervienen en el proceso de parsing del pseudocódigo.

#### Analizador Léxico (Lexer)

En los lenguajes de programación, las palabras son tratadas como objetos que describen nombre de variables, numeros, palabras claves, entre otras. Este tipo de palabras se conocen como *tokens*.

Un analizador léxico se puede definir como un programa que recibe una entrada en forma de frases o sentencias. Estas sentencias se dividen en palabras o símbolos individuales para formar lo que se conocen como *tokens*. Un *token* representa todo aquello que consideramos relevante para la sintaxis que permita identificar si pertenece o no a un lenguaje de programación. Un *token* también permitirá diferenciar espacios en blanco, comentarios, nueva línea, etc.

La principal función de un analizador léxico es identificar y validar si un conjunto de sentencias pertenecen a la sintaxis definida para un lenguaje. Existen razones suficientes para mantener una fase de análisis léxico entre el código fuente y el parser para generar un árbol de sintaxis abstracta (AST). Las más importantes son:

- **Eficiencia:** permite validar de manera rápida si un código fuente pertenece a la sintaxis definida por un lenguaje de programación, lo que a su vez permite identificar qué parte del código está fallando, antes de llegar a la fase del análisis sintáctico.
- **Modularidad:** describe de manera modular cada aspecto que conforma un lenguaje de programación, partiendo de la definición más simple como lo es definir desde qué es un número, un espacio en blanco o una letra, a una definición más compleja como, por ejemplo, cómo se conforma una declaración de variable.

Como se mencionó anteriormente, un analizador léxico parte desde la unidad o definición más simple hasta llegar a definir estructuras mucho más complejas como condicionales, declaración de variables, funciones, expresiones, etc. Para ello, se necesita una

estrategia que permita identificar, dado un código fuente, qué es un nombre de variable, un número, una letra y para eso utilizamos expresiones regulares.

El conjunto de números válidos o el conjunto de nombre de variables compuestas por una combinación de letras son obtenidos de un alfabeto definido previamente. Este grupo de alfabetos se conoce como un “lenguaje”. Por ejemplo, el alfabeto de los números consiste en dígitos que van del 0 al 9, y, para los nombres de variables, el alfabeto contiene tanto letras como números. Entonces, describiremos la combinación de los distintos símbolos del alfabeto por medio de expresiones regulares. Una expresión regular es un notación (representable por medio de un autómata finito) utilizada para generar de una manera simplificada posibles combinaciones de símbolos pertenecientes a un alfabeto para formar cadenas más complejas.

Por el momento conocemos cuál es el uso de un analizador léxico, su estructura y la estrategia para generar identificar cadena de símbolos; ahora, continuamos introduciendo las gramáticas. Una gramática léxica es una definición formal de un conjunto de *tokens* que identifican la sintaxis de un lenguaje, desde cómo se define una variable hasta qué es una función. A continuación se describen sólo algunos los elementos más básicos utilizados por la gramática definida para el nuestro pseudocódigo, a modo demostrativo:

- Números
    - *DecimalDigit*. Dígitos numéricos entre los valores cero y nueve,
    - *NonZeroDigit*. Dígitos numéricos entre los valores cero y nueve sin incluir el cero.
    - *DecimalIntegerLiteral*. Regla en la cuál un número puede ser:
      - Solamente un cero o un dígito distinto del cero (*NonZeroDigit*) más cualquier combinación de números (*DecimalDigit*).
      - Esta regla permite evitar, por ejemplo, que el número 0123 que comienza con cero sea una combinación válida para el lenguaje.
  - Literales
    - *BooleanLiteral*. Describe la regla que acepta los valores booleanos
    - *NumericLiteral*. Describe la regla que acepta los valores numéricos
    - *StringLiteral*. Describe la regla que acepta las cadenas de caracteres
  - Keywords
-



- *IsToken, IfToken, ElseToken, EqualsToken*. Define los tokens de la gramática que permite identificar palabras claves o reservadas dentro de la sintaxis del lenguaje.
- Operadores Multiplicativo y de Igualdad
  - *MultiplicativeOperator*. Regla utilizada para encontrar operaciones aritméticas dentro de una sentencia dada.
  - *EqualityOperator*: Regla utilizada para encontrar condiciones de igualdad dentro de estructuras condicionales.
- Booleanos
  - *BooleanLiteral*. Definición los valores “verdadero” y “falso”, devolviendo un nodo cuando encuentra los token *TrueToken* o *FalseToken*.

## GRAMATICA

```
// Números
```

```
DecimalDigit = [0-9]  
NonZeroDigit = [1-9]  
DecimalIntegerLiteral = "0" / NonZeroDigit DecimalDigit*
```

```
// Literales
```

```
Literal = BooleanLiteral / NumericLiteral / StringLiteral
```

```
// Keywords
```

```
Keyword = IsToken / IfToken / ElseToken / EqualsToken
```

```
// Tokens
```

```
IsToken = "is" / "en" / "="  
IfToken = "if" / "si"  
ElseToken = "else" / "sino"  
EqualsToken = "equals" / "igual" / "=="
```

```
// Operadores Multiplicativos
```

```
MultiplicativeOperator  
= $("*" !=") / $("/" !=") / $("%" !=")
```

```
// Operador de Igualdad
```

```
EqualityOperator  
= "==" / "!=" / EqualsToken {return "=="}  
/ NotEqualsToken {return "!="}
```

```
// Booleanos
```

```
BooleanLiteral  
= TrueToken { return { type: "Literal", value: true }; }  
/ FalseToken { return { type: "Literal", value: false }; }
```

Figura 11: Parte de la gramática definida para el pseudocódigo.

Cabe aclarar que la gramática completa definida para el pseudocódigo es mucho más extensa, con cientos de declaraciones como las antes indicadas.

#### Árbol de sintaxis abstracta (AST)

La sintaxis de un lenguaje de programación define cómo las distintas partes del código fuente (expresiones, instrucciones, declaraciones) deben estar combinadas para formar un programa ejecutable. Para ello, se debe expresar de una manera que sea comprensible tanto para personas como para una máquina.

El concepto de sintaxis involucra distintos conceptos: uno de ellos es el *Árbol de Sintaxis Abstracta* o *Abstract Syntax Tree*. Un *AST* es una estructura que representa cómo están compuestas las distintas sentencias de un programa, en el cual cada nodo es un operador que combina un conjunto de sentencias para formar otras sentencias. La idea de un árbol abstracto es la de capturar la estructura jerárquica de la sintaxis de un programa, evitando representar símbolos que son redundantes para un compilador o un generador de código. Por lo tanto, un *AST* es un árbol ordenado en donde sus hojas son variables y sus nodos interiores son operadores, cuyos hijos son argumentos.

La idea principal del uso de los árboles de sintaxis abstracta es reducir la cantidad de información que se genera a partir del analizador sintáctico, haciéndolo lo más agnóstico posible en cuanto a su implementación para que sea reconocido por la mayor cantidad de compiladores o generadores de código. En el siguiente gráfico se ilustra cómo a partir de una sentencia de un lenguaje de programación, en este caso pseudocódigo, se genera el *AST*:

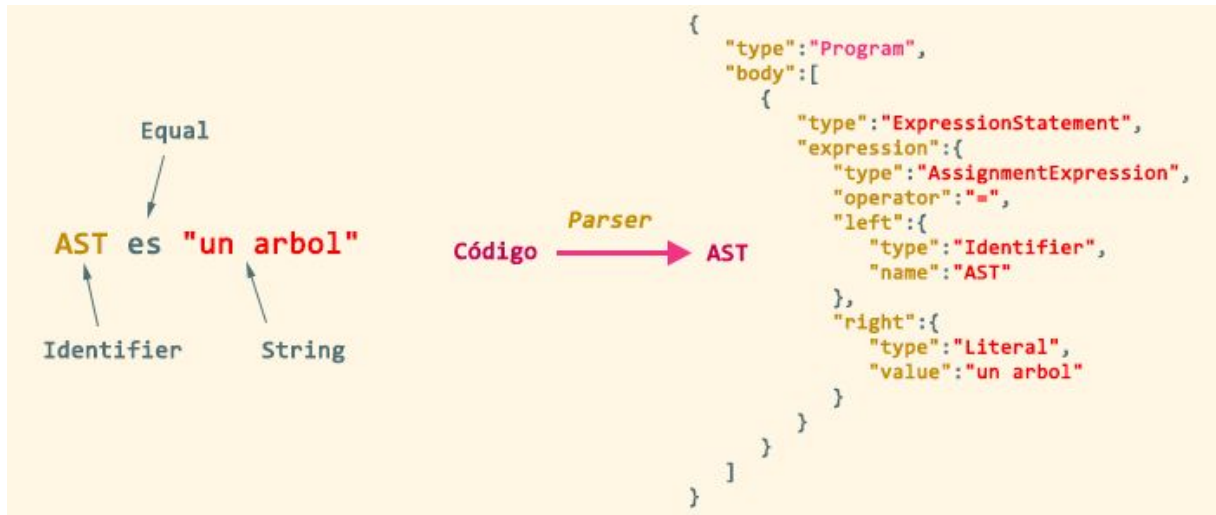


Figura 12: Ejemplo de AST a partir de una sentencia en pseudocódigo.

A continuación se procede a describir brevemente los campos que forman el *AST*:

- *type*: indica el tipo de expresión a la que corresponde dicho nodo.
- *body*: define el comienzo de las instrucciones de un programa.
- *expression*: nodo que describe cómo está compuesta una expresión.  
Conformada por el tipo, si contiene un operador de asignación y descripción del lado izquierdo y derecho de la expresión.
- *operator*: símbolo lógico que indica la operación a realizar.
- *left*: lado izquierdo de la expresión, en este caso, es el nodo que define la sentencia de declaración de la variable “AST”.
- *right*: lado derecho de la expresión, en este caso, es el nodo que define la sentencia de asignación de la variable.
- *name*: indica el nombre de la variable.
- *value*: indica el valor asignado a la variable creada anteriormente.

Para concluir con el seguimiento del ejemplo a lo largo de esta sección, a continuación se muestra el código final en *JavaScript* resultante de todo el proceso:

```
listaDeAlumnos = [  
    'Juan',  
    'Florencia',  
    'Juan',  
    'Ana'  
];  
juan = 0;  
resto = 0;  
for (nombre in listaDeAlumnos) {  
    if (nombre == 'Juan') {  
        juan = juan + 1;  
    } else {  
        resto = resto + 1;  
    }  
}  
if (juan > resto) {  
    repetición = true;  
} else {  
    repetición = false;  
}
```

## Tecnologías

Teniendo en cuenta la decisión de llevar adelante el desarrollo de la solución en *JavaScript*, optamos por desarrollar los componentes asociados a *lenguaje natural* y *pseudocódigo*, descritos más adelante, como dos elementos funcionalmente separados en forma de librerías. Más específicamente, en forma de librerías que implementan la especificación *AMD (Asynchronous Module Definition)*, lo cual permitirá importar de manera

estándar dichos componentes directamente a cualquier aplicación ejecutada sobre el entorno *Node.js* o (con asistencia de librerías como *require.js*) en navegadores.

*AMD* o *Asynchronous Module Definition* es una *API* que especifica un mecanismo para definir módulos de código (JavaScript) que puedan ser cargados o importados de manera asincrónica, lo cual es particularmente beneficioso para su uso en un navegador por el cambio en la performance. *AMD*, sin ser un estándar ni ser la única especificación de este tipo, se convirtió en el modo más popular para definir módulos de código en aplicaciones realizadas en *JavaScript*, tanto para el frontend (navegadores) como para aplicaciones backend. Añadido a esto, cumplir con la especificación abre la posibilidad de publicar las librerías como módulos a *Node Package Manager* o *npm*, el principal repositorio público online de librerías abiertas para *Node.js*.

## Plataforma interactiva

Como se mencionó en la descripción, la decisión respecto de la implementación de la plataforma interactiva fue de hacerla en forma de aplicación web. Para esto, nos apoyamos en una arquitectura tradicional *cliente - servidor*, que describiremos a continuación.

### Cliente

#### Tecnologías

Considerando que el cliente de la aplicación será un navegador web, las decisiones técnicas de bajo nivel para construir la plataforma quedan bastante reducidas al *stack* estándar de facto de la web: *HTML*, *CSS* y *Javascript*. Sin embargo, a día de hoy existe una amplia variedad de herramientas y frameworks de desarrollo muy potentes que simplifican el trabajo con estos 3 lenguajes y abstraen mucho del proceso de implementación.

De entre estos frameworks de desarrollo web, optamos por el relativamente novedoso *React*. *React* es una librería de *JavaScript* para el desarrollo de interfaces de usuario, enfocada más específicamente, pero no únicamente, a interfaces web. Esta librería permite el desarrollo de webs dinámicas con una importante lógica embebida (en contraposición a las ya deprecadas webs total o casi totalmente estáticas predominantes hasta hace algunos años),

todo ello con un gran nivel de abstracción y compatibilidad, considerando que simplifica mucho la portabilidad a los principales navegadores web, incluidos navegadores en dispositivos móviles. La particularidad de *React* como librería generadora de interfaces es que su ejecución se lleva a cabo directamente en el cliente (lo que se conoce como *client-side rendering*), en contraposición a otras tecnologías de *templating* más tradicionales como *JSP* y *ASP*; librerías como *React* facilitan la inclusión (necesaria en este proyecto) de una lógica compleja mediante *JavaScript* en la interfaz web, que deberá responder de distintas formas y de manera instantánea a las acciones relevantes del usuario.

El principal motivo para optar por *React*, además de sus funcionalidades, es que es una librería ya conocida y manejada por quienes desarrollamos el presente proyecto.

Adicionalmente a lo hasta ahora descrito, cabe destacar que todo el proyecto de la plataforma interactiva, es decir, tanto el código ejecutable sobre el cliente como el código del backend detallado más adelante, fue programado en *JavaScript* según la especificación *ES6*, también conocida como *ECMAScript6* o *ECMAScript2015*. *ECMAScript* es el nombre que se da a la especificación de un lenguaje de programación definido por la *European Computer Manufacturers Association*, cuya implementación más popular es *JavaScript*. En esta versión más reciente de la especificación se introducen una variedad de conceptos interesantes al lenguaje, de entre los cuales se destaca la implementación nativa de clases y herencias. Por esta y otras características consideradas de utilidad es que se tomó la decisión de emplear esta especificación en contraposición a *ES5*, que fue hasta recientemente la especificación estándar.

Debido a que *Node.js* aún no ha implementado por completo las características de *ES6*, el código *JavaScript* escrito bajo esta especificación no puede ser ejecutado directamente bajo dicha plataforma. Para esto, previo a la ejecución en sí, se aplica un paso intermedio que hace uso de *Babel*, un módulo de *Node* que compila *JavaScript* redactado bajo *ES6* hacia su equivalente en *ES5*, permitiendo su ejecución sobre *Node* o cualquier otro entorno o navegador compatible. Por esta naturaleza, no se puede afirmar que haya ventajas en performance en la ejecución comparando el uso de *ES6* frente a *ES5*, pero sí permite a los desarrolladores aprovechar las características de la especificación más reciente.

## Diseño

Por la naturaleza de la solución propuesta, el diseño de la interfaz de usuario juega un papel central. Esto es especialmente así por estar dirigida principalmente a un público joven, por lo cual la web debe ser visualmente atractiva y sobre todo llamativa, debe generar en el usuario un interés real que lo motive a hacer uso de las herramientas didácticas que son, en definitiva, el núcleo de la plataforma.

El diseño de la web se centra entonces en dos conceptos:

- Primero, se propone un acercamiento al uso de las herramientas en forma de “guía de ejercicios”. El usuario irá siendo guiado con textos, imágenes y ejemplos a través de una serie de ejercicios de dificultad incremental que le permitirán descubrir las funcionalidades propias de la herramienta, y, finalmente, los conceptos básicos de la programación. Dada la propuesta de realizarlo en forma de ejercicios, la plataforma contendrá las funcionalidades necesarias para corroborar efectivamente que la solución propuesta por el usuario sea correcta para cada situación planteada.
- En segundo lugar, tanto el planteo como la solución de los ejercicios, y la interfaz con la que el usuario interactúa para la resolución, deberían ser intuitivos y tener algún factor que “llame la atención”. Para esto, la propuesta consiste en mostrar en tiempo real, a medida que el usuario completa el ejercicio, el avance progresivo de la solución, como indicador también del acercamiento (o del alejamiento) a la respuesta correcta al problema planteado.

## Servidor

Considerando la decisión de implementar en *JavaScript* el stack entero de la solución propuesta, es decir, tanto la plataforma web como la implementación del lenguaje, surge la decisión de desarrollar el *backend* de dicha web sobre *Node.js*. *Node.js* es un runtime nativo de *JavaScript*, que permite ejecutar aplicaciones desarrolladas en este lenguaje fuera del navegador. Contiene además, en sus releases oficiales, herramientas suficientes para desarrollar un servidor web sencillo, que será precisamente el uso que se le dará en esta parte de la solución.



En conjunto con *Node.js* se empleó *Express*, uno de los módulos más populares para esta plataforma que permite la publicación de una *API* web, que describiremos más adelante. A través de esta *API*, el cliente web se comunicará con el servidor para realizar las correspondientes lecturas, modificaciones y creaciones de los recursos pertinentes. Los recursos manejados por el sistema serán en parte estáticos y en parte dinámicos. Entre los recursos estáticos se encuentran los archivos propios de la aplicación, cómo el código de frontend, las imágenes empleadas, las hojas de estilo y demás. Los recursos dinámicos abarcan esencialmente los asociados al modelo de backend del *aula virtual*, descrito en detalle más adelante.

La persistencia en el backend de los recursos dinámicos antes mencionados se realiza sobre una base de datos relacional, más específicamente *MySQL*, con un esquema a medida correspondiente con el modelo del aula virtual.

Respecto al uso de *JavaScript*, en el desarrollo del backend de esta plataforma aplican las mismas observaciones hechas en la descripción de la parte del cliente, vinculadas a *ES6* y *Babel*.

## Modelo

A continuación se describe el modelo de negocio implementado en el backend de la plataforma interactiva, vinculado a la mecánica de aula virtual descrita anteriormente.

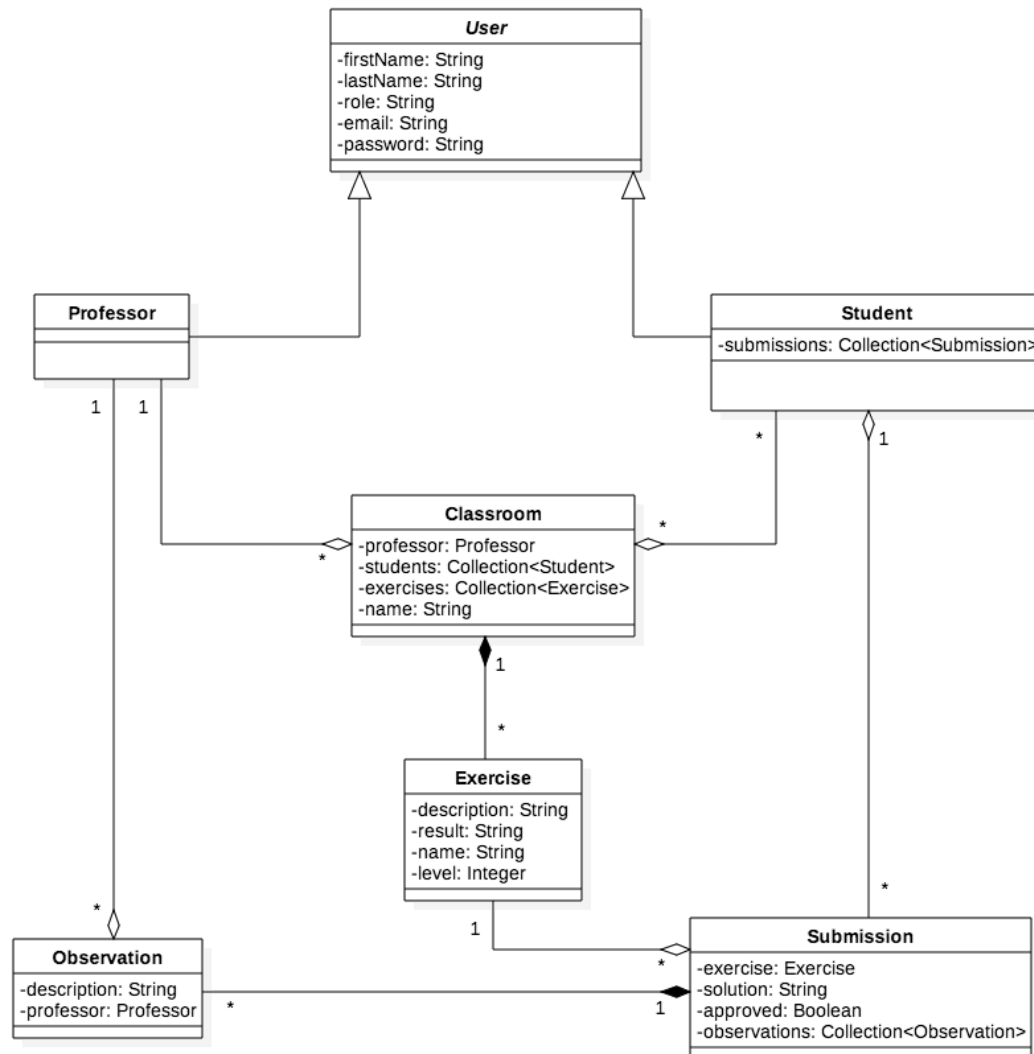


Figura 13: Modelo del aula virtual.

- *Usuario (User)*: Clase abstracta para representar a cualquier usuario creado mediante un alta y capaz de iniciar sesión en la plataforma. Sus campos más relevantes son un email, que debe ser único, y una contraseña.
- *Estudiante (Student)*: Clase que hereda características de *Usuario* y representa a los estudiantes del modelo. Contiene una lista de *Entregas*.
- *Profesor (Professor)*: Clase que hereda características de *Usuario* para representar a los profesores en el modelo.

- *Ejercicio (Exercise)*: Clase que representa un ejercicio individual del modelo de aula virtual, con un nombre, una descripción y el resultado esperado. Un *Ejercicio* es creado por un *Profesor*.
- *Clase (Classroom)*: Representa a una clase o curso del modelo de aula virtual, conteniendo una lista de ejercicios que los *Estudiantes* resolverán de forma sucesiva (un *Ejercicio* debe ser aprobado antes de poder continuar con el siguiente, si lo hay). Posee también una lista de los *Estudiantes* miembros que se hayan suscrito a la *Clase*. Una *Clase* es creada por un *Profesor*.
- *Entrega (Submission)*: Representa la entrega o presentación de la resolución de un determinado *Ejercicio* por parte de un *Estudiante*. Cada *Entrega* puede ser aprobada o rechazada, por lo cual un mismo *Estudiante* puede llegar a realizar varias sobre un mismo *Ejercicio*. Una *Entrega* puede también tener una o más *Observaciones* realizadas por el *Profesor* a cargo de la *Clase* que contiene al *Ejercicio*.
- *Observación (Observation)*: Representa una corrección o comentario que un *Profesor* puede realizar sobre una *Entrega* realizada por un *Estudiante* sobre uno de los *Ejercicios* de sus *Clases*.

## API

La comunicación entre el cliente web y el servidor se produce a través de una *API* (*Application Programming Interface*). Más específicamente, la *API* del proyecto fue diseñada siguiendo los lineamientos de las *APIs REST* (*Representational State Transfer*).

Una *API REST* es un tipo de interfaz web que permite al cliente, ya sea manejado por un usuario o una aplicación, interactuar con el servidor y acceder a las funciones o datos que el mismo publique. Estas interacciones se producen mediante pedidos o *requests HTTP* (*Hypertext Transfer Protocol*), empleando los métodos especificados por dicho protocolo para realizar distintas operaciones sobre los recursos publicados.

Todos los datos enviados y recibidos a través de la *API* implementada siguen el formato *JSON* (*JavaScript Object Notation*), a modo de unificar el formato de acceso y envío de datos para los distintos recursos publicados en la interfaz.

A continuación se describen brevemente las operaciones más relevantes publicadas por la *API* implementada, esencialmente las relacionadas al modelo de aula virtual:

- */login*
  - GET: Vista de inicio de sesión.
  - POST: Método estándar de inicio de sesión para los usuarios de la plataforma mediante una combinación de dirección de email y contraseña.
- */api/classrooms*
  - GET: Retorna un array de objetos JSON compuesto por las *clases* a las que el usuario en sesión tenga acceso.
  - */:id* GET: Retorna un único objeto JSON que representa la *clase* con el *id* indicado en la URL.
  - POST: Si el usuario en sesión es un *profesor*, crea una *clase* con los campos enviados.
  - */subscription* POST: Si el usuario en sesión es un *estudiante*, se suscribe a la *clase*.
- */api/exercises*
  - GET: Retorna un array de objetos JSON compuesto por los *ejercicios* a los que el usuario en sesión tenga acceso.
  - */:id* GET: Retorna un único objeto JSON que representa el *ejercicio* con el *id* indicado en la URL.
  - POST: Si el usuario en sesión es un *profesor*, crea una *clase* con los campos enviados.
- */api/observations*
  - GET: Retorna un array de objetos JSON compuesto por las *observaciones* a las que el usuario en sesión tenga acceso.
  - */:id* GET: Retorna un único objeto JSON que representa la *observación* con el *id* indicado en la URL.
  - POST: Si el usuario en sesión es un *profesor*, crea una *observación* con los campos enviados.

- */api/submissions*
  - GET: Retorna un array de objetos JSON compuesto por las *entregas* a las que el usuario en sesión tenga acceso.
  - */:id* GET: Retorna un único objeto JSON que representa la *entrega* con el *id* indicado en la URL.
  - POST: Si el usuario en sesión es un *estudiante*, crea una *entrega* con los campos enviados.
- */api/users*
  - */students*
    - GET: Retorna un array de objetos JSON compuesto por los *estudiantes* a los que el usuario en sesión pueda visualizar.
    - */:id* GET: Retorna un único objeto JSON que representa al *estudiante* con el *id* indicado en la URL.
    - POST: Crea un nuevo usuario de tipo *estudiante* con los datos enviados, siempre y cuando no existiera previamente un usuario con la misma dirección de email.
  - */professors*
    - GET: Retorna un array de objetos JSON compuesto por los *profesores* a los que el usuario en sesión pueda visualizar.
    - */:id* GET: Retorna un único objeto JSON que representa al *profesor* con el *id* indicado en la URL.
    - POST: Crea un nuevo usuario de tipo *profesor* con los datos enviados, siempre y cuando no existiera previamente un usuario con la misma dirección de email.
- */translate*
  - POST: A partir del input de texto enviado (lenguaje natural), retorna sus traducciones a pseudocódigo y *JavaScript*.

## Persistencia

El modelo de aula virtual antes descrito es persistido en el backend en una base de datos de tipo relacional. Específicamente, el motor de administración de bases de datos empleado es la “versión comunitaria” de *MySQL*, que es de código abierto y gratuito.

Como capa intermedia entre la lógica del modelo y la base de datos, dentro de la aplicación ejecutada en *Node* se implementa un mapeo asistido por la librería *Bookshelf*, un *ORM* (*Object-Relational Mapping tool*) nativo para la plataforma, que implementa las características típicas comunes a la mayor parte de las herramientas de este tipo, como el mapeo de relaciones entre modelos/clases, las herencias y demás abstracciones.

El esquema de la base de datos correspondiente a cada elemento del modelo es creado a partir de los siguientes scripts *SQL*:

```
CREATE TABLE students (  
  id bigint(19) not null auto_increment,  
  first_name varchar(500) not null,  
  last_name varchar(500) not null,  
  email varchar(500) not null,  
  password varchar(500),  
  primary key(id)  
);
```

Figura 14: Tabla *students*, vinculada a la clase *Estudiante* del modelo.

```
CREATE TABLE professors (  
  id bigint(19) not null auto_increment,  
  first_name varchar(500) not null,  
  last_name varchar(500) not null,  
  email varchar(500) not null,  
  password varchar(500),  
  primary key(id)  
);
```

Figura 15: Tabla *professors*, vinculada a la clase *Profesor* del modelo.

```
CREATE TABLE exercises (  
  id bigint(19) not null auto_increment,  
  id_classroom bigint(19) not null,  
  name varchar(500) not null,  
  description text not null,  
  result text not null,  
  level smallint(4) not null,  
  primary key(id),  
  foreign key (id_classroom) references classrooms(id)  
);
```

Figura 16: Tabla *exercises*, vinculada a la clase *Ejercicio* del modelo.

```
CREATE TABLE classrooms (  
  id bigint(19) not null auto_increment,  
  id_professor bigint(19) not null,  
  name varchar(500) not null,  
  primary key(id),  
  foreign key (id_professor) references professors(id)  
);
```

Figura 17: Tabla *classrooms*, vinculada a la *Clase* del modelo.

```
CREATE TABLE submissions (  
  id bigint(19) not null auto_increment,  
  id_exercise bigint(19) not null,  
  id_student bigint(19) not null,  
  solution text not null,  
  approved tinyint(1) not null default false,  
  primary key(id),  
  foreign key (id_exercise) references exercises(id),  
  foreign key (id_student) references students(id)  
);
```

Figura 18: Tabla *submissions*, vinculada a la clase *Entrega* del modelo.

```
CREATE TABLE observations (  
  id bigint(19) not null auto_increment,  
  id_submission bigint(19) not null,  
  id_professor bigint(19) not null,  
  description text not null,  
  primary key(id),  
  foreign key (id_submission) references submissions(id),  
  foreign key (id_professor) references professors(id)  
);
```

Figura 19: Tabla **observations**, vinculada a la clase **Observación** del modelo.

```
CREATE TABLE students_classrooms (  
  id_student bigint(19) not null,  
  id_classroom bigint(19) not null,  
  description text not null,  
  primary key(id_student, id_classroom),  
  foreign key (id_student) references students(id),  
  foreign key (id_classroom) references classrooms(id)  
);
```

Figura 20: Tabla auxiliar **students\_classrooms**, empleada para persistir la relación entre **Estudiante** y **Clase**, dada su naturaleza **many-to-many** (una clase puede tener muchos estudiantes asociados, y un estudiante puede estar asociado a su vez a varias clases).



## Pruebas Realizadas

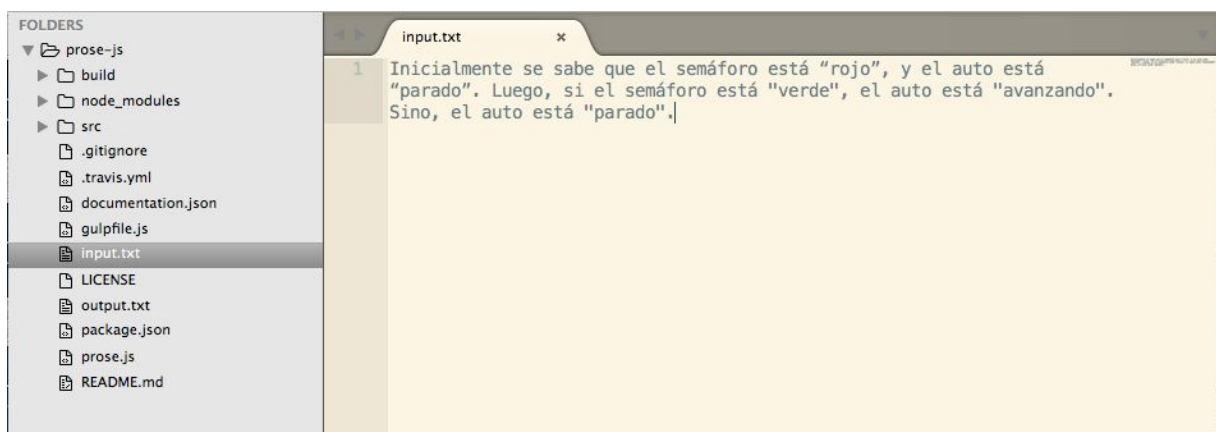
Llevamos adelante algunas pruebas de las herramientas desarrolladas para el pasaje de texto natural a código *JavaScript*.

Como fue mencionado anteriormente, estas herramientas fueron desarrolladas enteramente en *JavaScript* y en forma de módulos o paquetes de *Node*. Como tales, su funcionalidad final se expone en forma de métodos ejecutables en otras piezas de código *JavaScript*. Sin embargo, a fines de evaluación, realizamos algunos ajustes para que cada módulo pueda ser ejecutado independientemente como una pequeña aplicación por medio de línea de comando.

Mediante el comando `npm start`, se inicia la ejecución del programa, que lee el texto de un archivo de texto `input.txt` ubicado en la raíz del proyecto, y redacta el resultado final en un archivo `output.txt` en el mismo directorio. Este comportamiento es igual para ambos módulos, *prose-js* (traductor de lenguaje natural a pseudocódigo) y *pseudo-js* (traductor de pseudocódigo a *JavaScript*), cada uno, lógicamente, con el tipo de entrada correspondiente esperada, y con su correspondiente salida.

Planteamos, como caso de prueba, el siguiente ejemplo:

*Inicialmente se sabe que el semáforo está "rojo", y el auto está "parado". Luego, si el semáforo está "verde", el auto está "avanzando". Sino, el auto está "parado".*



```
1 Inicialmente se sabe que el semáforo está "rojo", y el auto está
   "parado". Luego, si el semáforo está "verde", el auto está "avanzando".
   Sino, el auto está "parado".
```

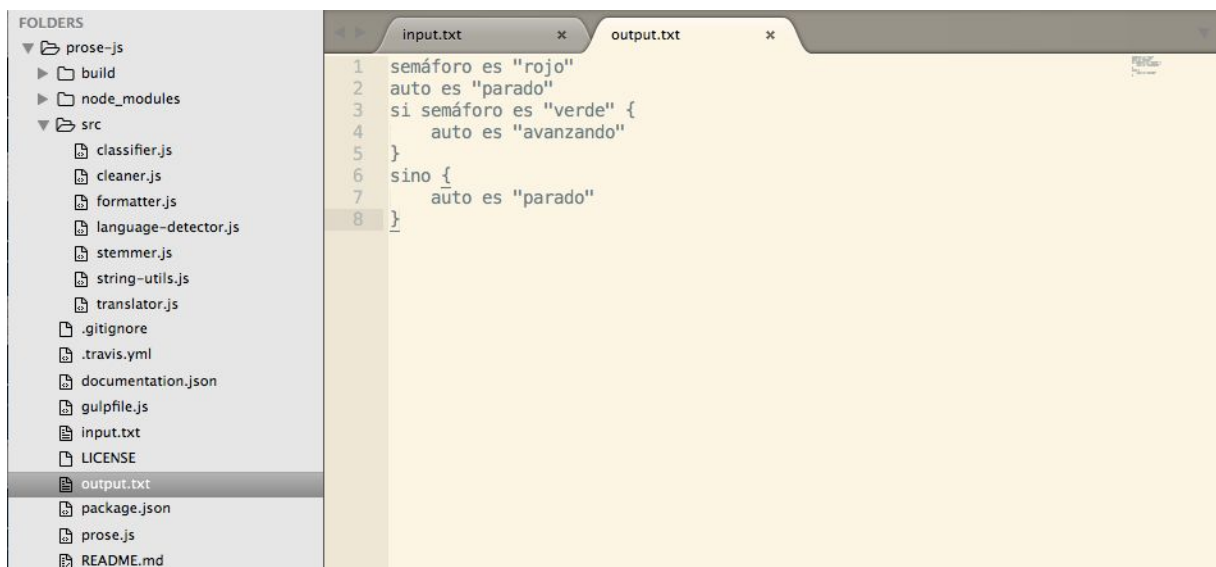
Figura 21: Se ingresa el ejemplo en el archivo *input.txt*, ubicado en la raíz del proyecto *prose-js*.

```
rodriguez@MacBook-Air-de-Gabriel:~/Desktop/PFI/prose-js$ npm start
> prose-js@0.0.12 start /Users/rodriguez/Desktop/PFI/prose-js
> NODE_ENV=development node prose.js

Leyendo el texto natural desde input.txt...
...
...
Procesando el texto natural...
...
...

Pseudocódigo
=====
semáforo es "rojo"
auto es "parado"
si semáforo es "verde" {
  auto es "avanzando"
}
Sino {
  auto es "parado"
}
Proceso completo! Los resultados se guardaron en output.txt
rodriguez@MacBook-Air-de-Gabriel:~/Desktop/PFI/prose-js$
```

Figura 22: Se ejecuta el programa desde la línea de comando.



```
FOLDERS
└─ prose-js
  └─ build
  └─ node_modules
  └─ src
    ├── classifier.js
    ├── cleaner.js
    ├── formatter.js
    ├── language-detector.js
    ├── stemmer.js
    ├── string-utils.js
    └── translator.js
  ├── .gitignore
  ├── .travis.yml
  ├── documentation.json
  ├── gulpfile.js
  ├── input.txt
  ├── LICENSE
  └── output.txt
  ├── package.json
  ├── prose.js
  └── README.md

input.txt x output.txt x
1 semáforo es "rojo"
2 auto es "parado"
3 si semáforo es "verde" {
4   auto es "avanzando"
5 }
6 sino {
7   auto es "parado"
8 }
```

Figura 23: Resultado final en *output.txt*.

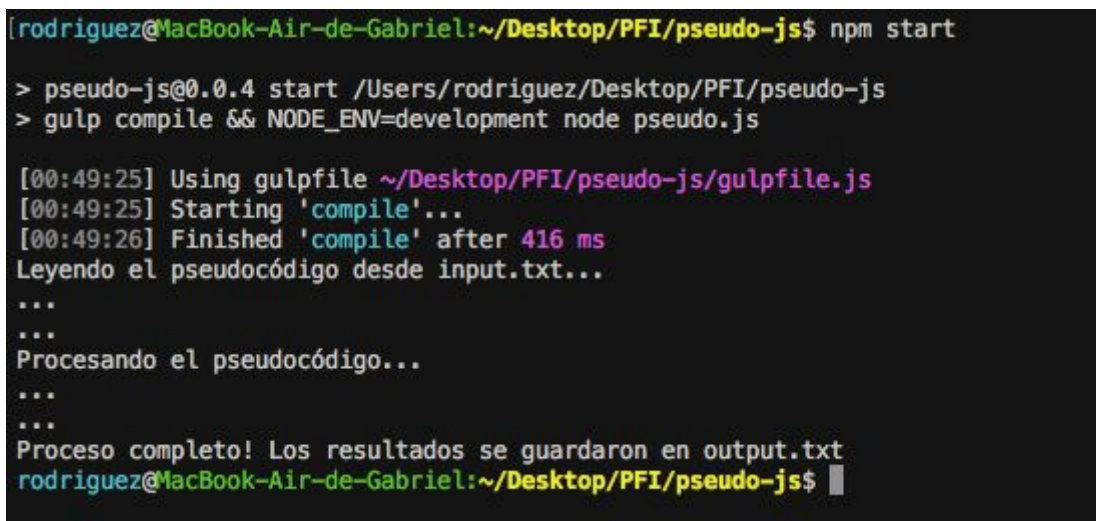
Concluido el procesamiento de *prose-js*, cambiamos de contexto hacia el proyecto *pseudo-js*. El programa de prueba leerá, nuevamente desde un archivo *input.txt* en la raíz del

proyecto, una porción de pseudocódigo cuya traducción, una vez concluída, se escribirá en *output.txt*.



```
1 semáforo es "rojo"
2 auto es "parado"
3 si semáforo es "verde" {
4     auto es "avanzando"
5 }
6 sino {
7     auto es "parado"
8 }
```

Figura 24: Se ingresa el resultado del proceso anterior, en forma de pseudocódigo, en el archivo *input.txt* ubicado en la raíz del proyecto *pseudo-js*.



```
[rodriguez@MacBook-Air-de-Gabriel:~/Desktop/PFI/pseudo-js$ npm start
> pseudo-js@0.0.4 start /Users/rodriguez/Desktop/PFI/pseudo-js
> gulp compile && NODE_ENV=development node pseudo.js

[00:49:25] Using gulpfile ~/Desktop/PFI/pseudo-js/gulpfile.js
[00:49:25] Starting 'compile'...
[00:49:26] Finished 'compile' after 416 ms
Leyendo el pseudocódigo desde input.txt...
...
...
Procesando el pseudocódigo...
...
...
Proceso completo! Los resultados se guardaron en output.txt
rodriguez@MacBook-Air-de-Gabriel:~/Desktop/PFI/pseudo-js$
```

Figura 25: Se realiza la ejecución del programa mediante línea de comandos.



```
1 semáforo = 'rojo';
2 auto = 'parado';
3 if (semáforo = 'verde') {
4     auto = 'avanzando';
5 } else {
6     auto = 'parado';
7 }
```

*Figura 26: Se observa en output.txt el resultado final del proceso, en forma de código JavaScript compatible.*

## Discusión

En la sección anterior, plasmamos pruebas realizadas con las herramientas de procesamiento sobre una porción de código en lenguaje natural. La prueba consiste en introducir una porción de texto escrito en español e ir mostrando el proceso de transformación que sufre esta entrada hasta producir una salida, es decir, otra porción de código que pueda ser ejecutado por el motor de algún lenguaje de programación. La prueba que realizamos produce la traducción de un texto natural al código equivalente en el lenguaje *JavaScript*, que se ejecuta sobre el motor conocido como *V8*.

Como se describió anteriormente, la herramienta de procesamiento de lenguaje natural efectúa la “traducción” del texto natural a pseudocódigo. Esto quiere decir, que todas las palabras y estructuras que no son relevantes o que agregan redundancia al código son eliminadas durante la primera parte del proceso. Luego, una vez obtenido el pseudocódigo, es necesario traducirlo a *JavaScript* para que sea ejecutable por algún motor.

La ventaja de utilizar estos pasos durante el proceso es la de proveer extensibilidad al traductor. Por lo tanto, si otra persona deseara utilizar estas herramientas para otro lenguaje de programación, por ejemplo *Java*, solo necesitará implementar el módulo traductor de pseudocódigo a un código equivalente *Java*, o generar un código en formato de bytecode para que pueda ser interpretado y ejecutado por la *JVM*. Manteniendo siempre fija la implementación del módulo traductor de texto natural, el proceso completo de traducción podría ser concluído con una variedad de módulos traductores de pseudocódigo a código desarrollados libremente, que sólo deberían seguir la especificación del pseudocódigo para efectuar su correspondiente proceso y generar el código final. Es de esta mecánica que surge un gran valor añadido de extensibilidad al proyecto.

Si bien concluimos el desarrollo de este proyecto conformes con el trabajo realizado, no podemos dejar de mencionar que no faltaron complicaciones a lo largo de los meses en que lo llevamos adelante. Trabajar con la interpretación de texto natural es una de los problemas más “icónicos” y relevantes dentro del desarrollo de software, particularmente en los últimos años, pero no por eso ha dejado de ser un problema de dificultad considerable ni se puede considerar que está cerca de estar resuelto. Por esto mismo no podemos negar que el

desarrollo que realizamos en este aspecto es claramente acotado y no está libre de fallas, teniendo muchos aspectos a mejorar y añadir, pudiendo bien ser un desarrollo de años.

Tampoco nos encontramos con un problema sencillo a la hora de resolver la interacción entre estas herramientas y el usuario final. Aquí tuvimos que tener en consideración uno de los aspectos quizás más difíciles de resolver para los desarrolladores (particularmente para quienes, como nosotros, no tienen un gran conocimiento específico sobre diseño): la experiencia de usuario. Si bien este trabajo también es mejorable, quedamos conformes con el desarrollo realizado sobre la plataforma web en cuanto a diseño y, particularmente, con la introducción de las mecánicas de aula virtual.

## Conclusiones

Es de importancia destacar que, a lo largo del proyecto, consideramos haber logrado cumplir los objetivos establecidos manteniendo la misma motivación que nos impulsó desde un comienzo.

La causa que nos llevó inicialmente al desarrollo de este proyecto es la de proveer una herramienta para aquellas personas, sean jóvenes o adultas, que desean comenzar a transitar sus primeros pasos en el mundo de la programación. La principal característica de la solución que ideamos fue desarrollar un lenguaje de programación con una sintaxis muy parecida al lenguaje natural, es decir, la forma en la que coloquialmente escribimos. Esto, en conjunto con la plataforma web interactiva, permite obtener una curva de aprendizaje muy leve para los nuevos usuarios, que desde el primer momento pueden experimentar e interiorizar conceptos de programación desde la plataforma.

Con esto hemos podido desarrollar una herramienta lo suficientemente capaz para hacer una traducción de texto español a código, reduciendo potencialmente en gran medida las barreras de entrada al campo de la programación; además, creemos que la inclusión de una plataforma didáctica para los usuarios suma más aún en este sentido, pues brinda la posibilidad de empezar a aprender sobre programación sin la necesidad de instalar ningún tipo de software.

Durante el desarrollo de este proyecto, surgió la idea que los usuarios puedan crear sus propios ejercicios. Esa idea evolucionó progresivamente hasta convertirse en la mecánica de aula virtual que incorporamos a la plataforma, y que consideramos que agrega un gran valor al producto final.

Como conclusión, pese a las limitaciones mencionadas en el apartado anterior, seguimos convencidos que el desarrollo realizado presenta un gran potencial real de aplicación. Además, teniendo en mente que en el último tiempo, los colegios y escuelas secundarias de Argentina están incluyendo paulatinamente la enseñanza de esta disciplina dentro de sus curriculas, anhelamos que nuestro desarrollo pueda ser utilizado en un futuro por la comunidad educativa como un medio práctico para la formación de estudiantes.

Este es un proyecto que definitivamente continuaremos desarrollando más allá del alcance de este informe.



## Anexos

### Especificación del Pseudocódigo

Adjuntamos a continuación la especificación inicial de la sintaxis definida para el pseudocódigo empleado en el proyecto. A modo aclaratorio, cabe destacar que es una gramática simplificada en comparación a las posibilidades y estructuras implementadas por la mayoría de los códigos de programación de uso práctico, teniendo en cuenta que el empleo de este pseudocódigo se realizará para ejemplos básicos y simples.

### Variables

Las variables son débilmente tipadas o *duck-typed*, lo cual implica que el control del tipo de valor se realiza en tiempo de ejecución, y no se les debe asignar preliminarmente un tipo. Para su declaración, es opcional el uso de la palabra reservada *var*.

La asignación de valores a una variable se realiza a través de los operadores *es* o *=*, que actúan como sinónimos, con la variable a asignar al lado izquierdo del operador, y el valor al lado derecho. Ejemplo: *valor es 2*.

En esta primera implementación, los valores posibles de las variables son de tipo numérico, alfanumérico, booleano y array.

### Condicionales

Los condicionales definen ciclos y condiciones que controlan el flujo de ejecución de acuerdo a la evaluación de valores booleanos.

Existen dos tipos de ciclos: *por cada X en Y...*, y *mientras CONDICION ...*.

Además, existen dos tipos de estructuras condicionales simples: *si CONDICION ...*, y *si CONDICION ... sino ...*.

## Sentencias

Una sentencia puede ser una asignación de variable (*valor es 2*), una operación básica ( $a + b$ ) o una combinación de ambas. Las sentencias se separan entre sí mediante saltos de línea.

## Bloques

Los bloques son listas de sentencias, que pueden incluir condicionales, a ser ejecutadas en orden sucesivo. Cada bloque (a excepción del bloque “principal” o externo) se encierra entre llaves (“{” y “}”). Por ejemplo: *si CONDICION { ... }*.

## Operadores

Además de los operadores de asignación “*es*” y “=”, se emplean otros operadores comunes a varios lenguajes de programación. Entre ellos, operadores de tipo aritmético (+, -, \*, / y &), de tipo comparativo (==, >, <, >=, <=, y !=) y de tipo booleano (y o &&, o u ||, y !).

## Natural Language Processing

### Clasificación de Textos

En primer lugar, cabe explicar qué significa “clasificación” en este contexto. En un problema de *clasificación*, dado un determinado conjunto de *clases*, se busca determinar a qué clase o clases pertenece cierto objeto. Más específicamente, los problemas de clasificación de textos toman como objetos a palabras individuales, oraciones o incluso textos completos, a los cuales se denominan *documentos*; las clases en las que ubicarlo abarcan un amplio rango, pudiéndose tratar, por ejemplo, de idiomas, temáticas, géneros literarios, y hasta sentimientos (un tipo específico de clasificación llamado *análisis de sentimientos*).

La automatización de los problemas de clasificación de textos involucra el uso de técnicas de machine learning. Estas soluciones aplican un análisis estadístico sobre un set de datos de entrenamiento, para luego realizar las clasificaciones. Dichas implementaciones

suelen ser de *aprendizaje supervisado*, es decir que el set de datos de entrenamiento y las clases a identificar son predeterminadas por un supervisor humano.

Uno de los principales métodos de resolución de este tipo de problemas es el de *Naive Bayes*. El principal motivo por el que este es el más comúnmente usado en la implementación de soluciones de este estilo, y también el empleado en el desarrollo descrito en este informe, es que alcanza un nivel de precisión adecuado más rápidamente, entendiendo por “más rápidamente” a un set de datos de entrenamiento más reducido en comparación a, por ejemplo, los requeridos para métodos de *regresión logística*. Esta cualidad encaja con el caso de uso de la solución propuesta, ya que el empleo del clasificador se reduce a la interpretación de ciertas expresiones comparativas booleanas y matemáticas que son relevantes para la traducción *lenguaje natural*  $\rightarrow$  *pseudocódigo*, pero que contienen un conjunto de datos de entrenamiento relativamente pequeño por la no tan amplia variabilidad de las expresiones equivalentes aceptables.

## Bibliografía

- Linn, Marcia y Dalbey, John. Cognitive consequences of programming instruction. En: Studying the novice programmer. 1ª. ed. New Jersey. Lawrence Erlbaum Associates, 1989. pp. 57–81 [fecha de consulta 11 de febrero de 2017]. Disponible en: <[https://www.researchgate.net/publication/238319032\\_Cognitive\\_Consequences\\_of\\_Programming\\_Instruction\\_Instruction\\_Access\\_and\\_Ability](https://www.researchgate.net/publication/238319032_Cognitive_Consequences_of_Programming_Instruction_Instruction_Access_and_Ability)>
- GitHub Octoverse 2016. GitHub, Inc. [fecha de consulta 21 de enero de 2017]. Disponible en: <<https://octoverse.github.com/>>.
- South America Internet and Facebook Users - Population 2016, Argentina [en línea]. [Estados Unidos]: Internet World Stats, [fecha de consulta 2 de febrero de 2017]. Disponible en: <<http://www.internetworldstats.com/south.htm#ar>>
- Manning, Christopher D., Raghavan, Prabhakar y Schütze, Hinrich. An Introduction to Information Retrieval [en línea]. [Cambridge, Reino Unido]: Cambridge University Press, abril 2009 [fecha de consulta 5 de febrero de 2017]. Disponible en: <[http://www.psi.uba.ar/investigaciones/revistas/normas/como\\_citar\\_doc\\_electronicos.pdf](http://www.psi.uba.ar/investigaciones/revistas/normas/como_citar_doc_electronicos.pdf)>
- Masse, Mark. REST API Design Rulebook. 1ª. ed. [Massachusetts, Estados Unidos]: O'Reilly Media, 2011.
- Ng, Andrew Y., Jordan, Michael. On Discriminative vs. Generative classifiers: A comparison of logistic regression and naive Bayes [en línea]. [Berkeley, Estados Unidos]: sin especificar, 2001 [fecha de consulta 5 de febrero de 2017]. Disponible en: <<https://ai.stanford.edu/~ang/papers/nips01-discriminativegenerative.pdf>>
- Shapiro, Stuart C. Artificial Intelligence. En: Encyclopedia of Artificial Intelligence. 2ª. ed. New York. John Wiley & Sons, 1992. pp. 54-57 [fecha de consulta 21 de abril de 2017]. Disponible en: <<http://www.cse.buffalo.edu/~shapiro/Papers/ai.pdf>>

- Real Academia Española. Diccionario de la lengua española (23ª ed.) [en línea]. [Madrid, España]: Real Academia Española, octubre 2014 [fecha de consulta 13 de junio de 2017]. Disponible en: <<http://dle.rae.es/>>