Aspectos de adquisición de lenguaje en la enseñanza de programación

por

Marcos J. Gómez

Presentado ante la Facultad de Matemática, Astronomía, Física y Computación para obtener el grado de Doctor en Ciencias de la Computación de la
Universidad Nacional de Córdoba

Julio, 2020

FAMAF - UNC

Directora: Dra. Luciana Benotti

CÓRDOBA ARGENTINA

Agradecimientos

En este mundo tan desigual e injusto, no puedo dejar de lado los privilegios que he tenido y las diferentes oportunidades que se han presentado durante todo este proceso. En ese sentido, poder agradecer a cada una de las personas que hicieron realidad este doctorado:

A Luciana Benotti, directora de doctorado y amiga. Por acompañarme, ayudarme, aconsejarme y estar siempre a disposición. Además, de compartir más de 7 años de trabajo e investigación en conjunto.

A los miembros del jurado, Ana Casali, Laura Brandan y Francisco Tamarit, por sus dedicadas lecturas, comentarios de la tesis y comprensión constante ante cada cambio de fecha en este año tan particular.

A Cecilia Martinez, formadora en lo pedagógico, consejera en lo didáctico, para poder pensar la enseñanza de la programación en contextos educativos reales.

A la Escuela Nueva Juan Mantovani, por confiar en el proyecto, y permitir llevar la enseñanza de la programación a la escuela. A Rubén Ulloque, ex director del nivel primario y Janet Saltanovich, vice rectora de la institución.

A Fernando Schapachnik por su confianza y apoyo constante. A esa enorme comunidad que es la Fundación Sadosky.

A cada una y uno de los miembros de UNC++, con quienes compartimos este hermoso objetivo de la democratización de las Ciencias de la Computación y la programación. A Ma. Emilia Echeveste, Joshep Cortez, Eduardo Rodriguez. A Marco Moresi, con quien compartimos investigación, trabajo y una gran amistad en la última etapa del doctorado.

A mi querida FaMAF, personal no docente y docente que siempre han estado a disposición. A Nicolas Wolovick, Pedro D' Argenio, Araceli Acosta, Javier Blanco, Daniel Fridlender por su apoyó en diferentes situaciones que sucedieron en el proceso del doctorado.

Al programa ProA, por la confianza en cuanto a propuestas para poder llevar la enseñanza de la programación. A Gabriel Scarano por su apoyo.

A los chicos y chicas de Mumuki. Por el trabajo en conjunto, por nuevas experiencias de trabajo. A Franco Bulgarelli por la confianza.

A mi familia. A mi mamá, a mi papá. Gracias a ellos, que me dieron la posibilidad y privilegio de poder primero, sólo estudiar y ocuparme de la carrera de grado, y brindarme su apoyo total en la nueva aventura del doctorado. A Ange, Fede y Viri. A mis abuelas y a mi familia.

A mis queridos amigos y amigas de toda la vida. A mis queridos amigos de la facu. A mis queridas amigas del manto.

A cada uno y una de los y las estudiantes que participaron de las diferentes iniciativas, experiencias, desafíos de enseñanza de programación. Gracias por su paciencia, y poder hacer realidad la enseñanza de programación en la escuela.

A mi compañera de aventuras, alegrías, tristezas. A mi compañera de vida. Gracias por el apoyo, el aguante, la paciencia. Gracias de corazón, por ser ese pilar tan importante que permitió hacer realidad este hermoso doctorado. Vamos por más. Gracias Anita.

A cada una de aquellas personas con las que he tenido el enorme gusto de trabajar y poder construir, debatir sobre la enseñanza de la programación.

Clasificación (ACM CCS 2012):

- Applied computing ∼ Education ∼ Interactive learning environments
- Social and professional topics ∼ Professional topics ∼ Computing education ∼ Computing education programs ∼ Computer science education
- Computing methodologies ∼ Artificial intelligence
- Computing methodologies ∼ Machine learning

Palabras Claves:

Machine Learning, Adaptive learning path, Artificial Intelligence, Learning Environments, Computer Science Education, Language Learning, Language Aqcuisition, Interactive Feedback, Language Expressiveness, Fluency

Resumen

La enseñanza de la programación se ha transformado en un tema popular en los últimos años. Gobiernos y grandes empresas invierten en comprar o desarrollar hardware, software y materiales didácticos para la enseñanza de la programación. Sin embargo, la programación es una disciplina tan reciente que poco se sabe de su didáctica. Se dedica mucho esfuerzo en diseñar herramientas pero mucho menos en evaluarlas de forma rigurosa. El objetivo general de esta tesis es estudiar cómo diversos factores del área de adquisición de lenguaje influencian la enseñanza de la programación. Nos enfocamos en tres aspectos. Primero, estudiamos la enseñanza de programación desde el punto de vista de la *expresividad del lenguaje*. Diseñamos un entorno de enseñanza de programación que incluye lenguajes con diferente nivel de expresividad y realizamos un estudio en nivel inicial y primario. Segundo, exploramos el impacto de distintos tipos de *interactividad formativa* automática asociada a distintos tipos de errores que ocurren durante el proceso de enseñanza de un lenguaje de programación. Al igual que los lenguajes naturales, los lenguajes de programación no se aprenden de modo espontáneo sino que se adquieren y evolucionan a través de la interacción. Tercero, propusimos y comparamos diferentes métodos basados en técnicas de procesamiento de lenguaje natural y de aprendizaje automático para evaluar el nivel de *fluidez* de un estudiante durante el proceso de adquirir un lenguaje de programación. Estos métodos permiten clasificar la gran variabilidad y creatividad de errores producidos por principiantes de una manera que no es posible con las técnicas formales de detección de errores en programación.

Abstract

Teaching programming has become popular in recent years. Governments and companies invest in purchasing or developing hardware, software, and teaching materials. However, programming is a recent discipline and we do not know enough about how to teach it. Much effort is spent on designing new tools for teaching programming, but much less on rigorous evaluation. The general objective of this thesis is to study how different factors in the area of language acquisition influence the teaching of programming. We focus on three aspects. First, we study programming language acquisition from the point of view of the expressiveness of the language. We design a programming teaching environment that includes languages with different levels of expressiveness and we develop an study at preschool and primary level. Second, we explore the effect of different types of automatic formative interactivity associated with different types of errors during the process of teaching a programming language. Like natural languages, programming languages cannot be learned instantly, but are acquired and evolve through interaction. Third, we propose and compare different natural language processing and machine learning techniques to assess a student level of fluency during the process of acquiring a programming language. These methods select for the great variability and creativity of early errors in a way that is not possible with formal programming error detection techniques.

Índice general

Capítulo 1

Introducción

La enseñanza de la programación se ha transformado en un tema popular en los últimos años. Gobiernos y grandes empresas invierten en comprar o desarrollar hardware, software y materiales didácticos para la enseñanza de la programación. Sin embargo, la programación es una disciplina tan reciente que poco se sabe de su didáctica. Se dedica mucho esfuerzo en diseñar herramientas pero mucho menos en evaluarlas de forma rigurosa.[1] El objetivo general de esta tesis es estudiar cómo diversos factores conocidos del área de adquisición de lenguaje influencian la enseñanza de la programación.

Este capítulo introductorio se organiza de la siguiente manera. En la Sección 1.1 presentamos diferentes razones que hacen de la enseñanza de programación un tema importante en la actualidad y a la vez difícil de implementar en el sistema educativo.[2] Luego, en la Sección 1.2 introducimos tres aspectos de la adquisición de lenguaje relevantes para la enseñanza de programación: expresividad, interactividad formativa y fluidez.[3] En la Sección 1.3 presentamos los vínculos que motivaron y se generaron a medida que desarrollamos el trabajo descripto en la tesis. En la Sección 1.4 enumeramos las publicaciones logradas en el proceso del doctorado. Finalmente, en la Sección 1.5 presentamos un resumen de cada uno de los capítulos de esta tesis.

1.1. Motivación para la enseñanza de programación

Poder llevar la enseñanza de la programación a todos los niveles educativos es un desafío que están atravesando muchos países del mundo. Reúne intereses de universidades, empresas, organizaciones y países como Reino Unido, Nueva Zelanda, Estados Unidos y Alemania [117, 47, 85, 38]. Gran parte del interés se basa en poder comenzar a formar posibles futuros empleados, teniendo en cuenta el auge actual y futuro de las profesiones relacionadas con la programación. Se dice que las profesiones relacionadas a la computación y programación son fundamentales para resolver los problemas y desafíos del presente y el futuro. Incluso quienes dudan de esta postura reconocen el crecimiento sostenido de programadores en el país y en el mundo.

En la Figura 1.1 podemos observar el crecimiento de la demanda laboral en empleos relacionadas al desarrollo de software en Argentina desde el año 2009 al 2018. El empleo, la variable de mayor sustento para entender el crecimiento del sector y su potencialidad, ha aumentado un 47,8 % entre 2009 y 2018, a una tasa anual acumulativa del 4,4 %. A modo de comparación, el empleo registrado de todo el sector privado entre ambos años creció un 11,4 % a una tasa anual

[1] En el Capítulo 3 realizamos una revisión de trabajo previo.
[2] Estas razones se profundizan en el Capítulo 2 a través de un estudio empírico.
[3] Estos aspectos se desarrollan en los Capítulos 4, 5 y 6 respectivamente.

acumulativa del 1,2 % [87]. A pesar de la crisis de 2008-2009 y las devaluaciones de principios de 2014, fines de 2015 y 2018, las ventas del sector medidas en dólares aumentaron en los últimos diez años un 2,9 % acumulativo anual. Este área en crecimiento demanda cada vez más profesionales formados.

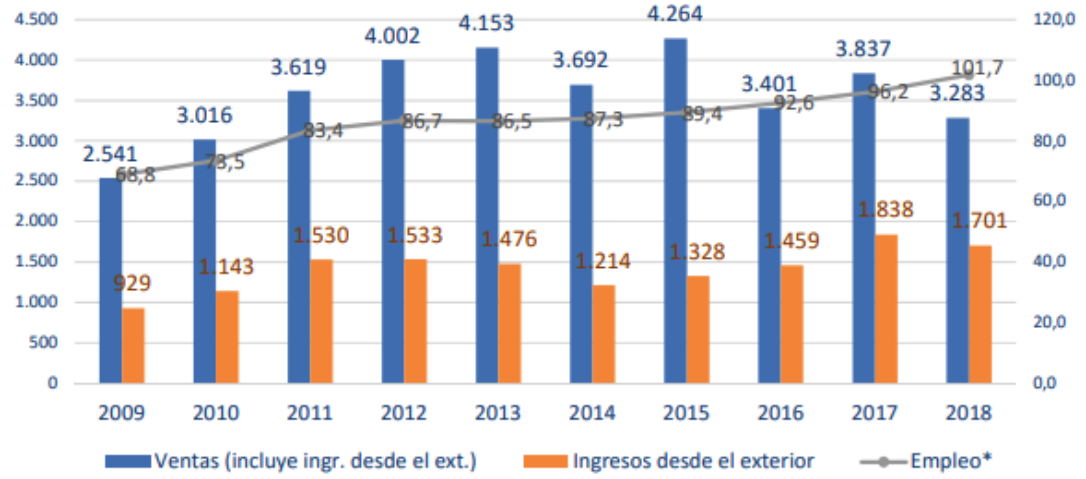


Figura 1.1: Evolución anual de ventas totales (en millones de USD), ingresos del exterior (en millones de USD) y empleo (en miles de empleados registrados) del sector de software en el período 2009 al 2018. Fuentes: Ventas: relevamiento OPSSI – Ingresos desde el Exterior: Balanza de Pagos INDEC – Empleo: Observatorio de Empleo y Dinámica Empresarial (Ministerio de Producción y Trabajo).

Más allá de la formación de programadores profesionales, en la actualidad hay muchos que defienden la importancia de *democratizar* la computación y la programación enseñándola en el sistema educativo obligatorio [18]. El mundo del cual formamos parte está digitalizado. Las computadoras nos atraviesan en nuestra cotidianeidad. Somos grandes consumidores de aplicaciones, videojuegos, celulares pero el usuario promedio desconoce su funcionamiento y gran parte de sus riesgos. La mayor parte de las actividades que realizamos en el día son acompañadas por una computadora. Nos despertamos con la alarma del celular, nos acostamos revisando cómo estará el clima, confirmando nuestras actividades en el calendario de google o resolviendo alguna situación de último momento con Whatsapp. Somos usuarios intensivos de programas y aplicaciones. En esta visión, enseñar a programar es tan importante como enseñar geografía en la escuela.

Esta es la primer generación de niños y adolescentes para la que, las computadoras se encuentran tan presentes en su hogar. Es común escuchar en reuniones familiares o con amigos un comentario similar a “Viste lo inteligente que son los chicos. Tienen esa facilidad para usar el celular. A mi me cuesta un montón”. Muchas veces nos sorprendemos con la facilidad en que los niños utilizan los programas. Desde los medios escuchamos, leemos que se refieren a los niños y adolescentes como “nativos digitales”. Para poner en tensión esta idea de “nativos digitales” me gustaría reflexionar a partir de las siguientes preguntas:

¿Tienen estos niños en la escuela la oportunidad de reflexionar acerca de quién, cómo y por qué son creados estos programas? Y, ¿si en realidad les es fácil utilizar estas aplicaciones porque fueron diseñadas y pensadas para ellos? Y al no entender cómo realmente funcionan estos programas y aplicaciones ¿los hace ciudadanos que pueden elegir libremente los programas o aplicaciones que utilizan? Y por lo tanto, ¿qué significa ser nativo digital? ¿Significa no necesitar

aprender computación en la escuela como algunos implican? ¿O significa ser el consumidor digital para el cual las empresas desarrollan los programas?

Este trabajo también desarrolla un deseo personal con respecto a la enseñanza de programación en la escuela primaria. Los niños se encuentran en un etapa de desarrollo de su aprendizaje. Son curiosos y en particular, es una etapa en la cual se interesan por todo. Al mismo tiempo, es parte de la población sobre la cual más influencia tienen las computadoras. Entonces, en plena etapa de aprendizaje e interés, son consumidores de aplicaciones, juegos y programas que influencian su toma de decisiones. Como docente de una escuela primaria en el espacio curricular de informática soy total partícipe de esta realidad. Y en ese sentido, creo fundamental comenzar la democratización de la programación en contextos reales y formales educativos, en particular de escuela primaria. Poder entender cómo funcionan los programas que utilizan permitirá a los niños ser usuarios críticos de la tecnología para no ser únicamente consumidores digitales.

En ese sentido, tanto modelar la fluidez de los estudiantes para predecir si el estudiante puede resolver un ejercicio por su cuenta como poder brindar un diseño curricular de programación para nivel primario comparten un objetivo: la enseñanza de programación de forma masiva.

1.2. Aspectos de adquisición del lenguaje

Un lenguaje de programación es un lenguaje formal. En esta tesis estudiamos aspectos de la adquisición de lenguajes formales y naturales aplicados a la adquisición de un lenguaje formal como es un lenguaje de programación. En particular, nos enfocamos en tres aspectos de la adquisición de lenguajes en la enseñanza de programación: expresividad, interactividad formativa y fluidez.

1.2.1. Expresividad

Los lenguajes de programación existentes, pueden clasificarse en varias categorías dependiendo de su nivel de expresividad. La expresividad del lenguaje es un tema central en la enseñanza de lenguajes formales como lenguajes lógicos, sin embargo no es una característica de los lenguajes de programación que se haga explícita al diseñar un lenguaje para enseñar a programar. El nivel de expresividad del lenguaje define el nivel de profundidad o control que podemos tener al momento de crear un nuevo programa. En particular nos enfocamos en lenguajes basados en bloques, lenguajes basados en texto, lenguajes doble modalidad y lenguajes que traducen de bloques a texto. Analizamos el impacto de lenguajes con diferentes niveles de expresividad al momento de enseñar a programar a principiantes. Los lenguajes de mayor expresividad nos permiten tener un mayor control de los programas en bajo nivel, pero para crear primeros programas simples al aprender a programar pueden ser complejos. Los lenguajes de menor expresividad permiten crear programas simples fácilmente pero el programador se ve restringido al control acotado del lenguaje.

1.2.2. Interactividad

Al igual que los lenguajes naturales, los lenguajes de programación no se aprenden de modo espontáneo sino que se adquieren y evolucionan a través de la interacción. Llamamos interactividad formativa a los refuerzos negativos y positivos que se reciben durante el aprendizaje. Este tipo de interactividad formativa normalmente es provista por un docente pero ante la ausencia de este, puede ser generada automáticamente por un sistema. Considerando el tipo de interactividad formativa automatizada que permite un entorno de enseñanza de programación, los

entornos se pueden clasificar en cerrados o abiertos. En los entornos cerrados los ejercicios de programación están predefinidos y por lo tanto el docente puede prever algunos de los tipos de errores de los estudiantes y es posible automatizar cierta interactividad formativa. Los entornos abiertos permiten implementar didácticas exploratorias como la didáctica basada en problemas pero no permiten al docente prever el tipo de errores que aparecerán y necesitan un seguimiento más personalizado de los estudiantes por parte del docente. Como parte de nuestra investigación desarrollamos experiencias en contextos educativos reales para poder evaluar el impacto al adquirir un lenguaje de programación en base a la interactividad formativa del entorno.

En lingüística, las correcciones que hacen explícito un error en la adquisición de un lenguaje natural se llaman solicitudes de reparación [100]. En educación se conocen como *feedback formativo*. Las solicitudes de reparación en las conversaciones entre adultos y niños guían la adquisición del lenguaje por parte de los niños. Los niños responden a las solicitudes de reparación, ya sean abiertas (¿Hm?, ¿Qué?) o restringidas (¿ocultaste qué?). Los niños responden a las solicitudes de reparación con auto correcciones en el próximo turno y hacen uso de los comentarios ofrecidos [27]. Durante la adquisición de lenguaje natural las solicitudes de reparación son más efectivas si aparecen de forma sincrónica con la producción del lenguaje. Es decir, la adquisición es más efectiva si las correcciones se producen en tiempo real [53], llevando a lo que en esta tesis llamamos una *interactividad formativa*.

1.2.3. Fluidez

La fluidez es el término utilizado por la lingüística para caracterizar y medir las habilidades de una persona con respecto a un lenguaje natural. De manera similar a la adquisición del lenguaje de programación, hay muchas aplicaciones educativas para el aprendizaje de un segundo idioma que han aumentado en popularidad en los últimos años. Estas aplicaciones generan grandes cantidades de datos de aprendizaje de los estudiantes que pueden aprovecharse para impulsar la instrucción personalizada. En este contexto, existe trabajo previo que propone metodologías para evaluar automáticamente la fluidez del aprendizaje de un lenguaje natural [60, 109]. Para evaluar automáticamente la fluidez en un lenguaje de programación, inspiramos nuestro trabajo en el área de adquisición de un segundo lenguaje natural. La fluidez se evalúa usando solo un fragmento de texto producido por el estudiante e intenta predecir si su dominio del lenguaje es suficiente o no para resolver un ejercicio dado sin ayuda humana [132]. En nuestro caso aplicamos el concepto de fluidez para lenguajes de programación. En esta tesis proponemos mecanismos para evaluar de forma automática el nivel de fluidez del estudiante durante el proceso de adquisición de un lenguaje de programación. En base a un programa creado por un estudiante, proponemos evaluar el nivel de fluidez de un estudiante para intentar predecir si podrá resolver un determinado ejercicio de programación sólo con la ayuda de feedback formativo automatizado o si necesita intervención de un docente.

1.3. Impacto provincial, nacional e internacional

Investigar la enseñanza de programación me permitió poder establecer vínculos y compartir experiencias con diferentes personas, empresas y organizaciones provinciales, nacionales e internacionales. Formar parte del grupo de extensión e investigación UNC++[4] de la Universidad Nacional de Córdoba, dirigido por Luciana Benotti y Cecilia Martínez me dio la oportunidad de acercarme a la docencia, a las escuelas y conocer de cerca el tema. UNC++ es un grupo

[4] http://masmas.unc.edu.ar/

interdisciplinario compuesto por personas de ciencias de la educación y ciencias de la computación cuyo objetivo principal es cerrar brechas digitales a partir de la enseñanza de las Ciencias de la Computación en las escuelas. Como miembro de UNC++ tuve la posibilidad de llevar la enseñanza de programación a diferentes escuelas secundarias de Córdoba y poder dictar cursos de formación docente. Con UNC++ llegamos a 500 docentes y más de 10000 estudiantes en toda la provincia de Córdoba. Fui convocado por la Unión de Educadores de la Provincia de Córdoba (UEPC) para diseñar y dictar cursos de formación docente de enseñanza de programación. También trabajamos en conjunto con el equipo de la Municipalidad de Córdoba para desarrollar secuencias didácticas para implementar la enseñanza de programación en todas las escuelas municipales.

Me desempeño como docente de nivel primario en la Escuela Nueva Juan Mantovani, donde con el apoyo de los equipos de gestión de la institución, desarrollamos algunos de los diferentes experimentos de enseñanza de programación descriptos en esta tesis. A partir de los mismos también pudimos desarrollar la primer propuesta curricular para la enseñanza de programación en la escuela primaria que está actualmente siendo evaluada por el Ministerio de Educación de la Provincia. Coordiné la orientación en desarrollo de software del Programa Avanzado de Educación con énfasis en TIC, conocidas como las escuelas ProA de la Provincia. Gabriel Scarano y Gabriela Peretti (coordinadores del proyecto ProA) me brindaron su total apoyo para revisar el diseño curricular existente y realizar una nueva propuesta curricular centrada en la enseñanza de programación.

Como parte del trabajo de esta tesis desarrollé el entorno de enseñanza de programación UNCDuino. UNCDuino fue adaptado por la empresa de robótica educativa RobotGroup como entorno alternativo para programar sus kits de robótica. Actualmente es utilizado por el Ministerio de Ciencia y Tecnología de la provincia de Córdoba en el programa de alfabetización tecnológica. El entorno UNCDuino y las diferentes secuencias didácticas que implementamos, fueron utilizados y modificados por la Fundación Sadosky para poder implementar cursos de formación profesional para docentes en enseñanza de programación y experiencias en diferentes instituciones educativas, trabajando con Fernando Schapanick y Belén Bonello. Fui único autor de uno de los capítulos del manual "Ciencias de la Computación para el aula – 2° ciclo de Primaria", proyecto generado por la Fundación Sadosky.

A partir de una experiencia que implementamos utilizando el entorno Mumuki[5] en escuela primaria, la empresa Ikumi, creadora de Mumuki decidió implementar dentro de su entorno un conjunto de guías desarrolladas exclusivamente para escuela primaria, incluyendo también lenguajes basados en bloques. Esta herramienta llegó a decenas de miles de chicos en San Luis. Antes de la experiencia que implementamos para escuela primaria, Mumuki estaba pensando para nivel secundario y universitario. Poder contar con el apoyo y contacto de Franco Bulgarelli, Agustina Pina y Nadia Finzi de Ikumi, nos permitió poder profundizar en evaluar este tipo de entornos y su interactividad formativa al momento de adquirir un nuevo lenguaje al enseñar a programar y poder escribir publicaciones científicas junto a emprendedores del país. Nos brindaron el conjunto de datos sobre el cual entrenamos nuestros modelos automáticos para evaluar el nivel de fluidez de los estudiantes. En el proceso de diseño de los diferentes modelos visité y dicté un seminario en el Departamento de Ciencias en la Computación de la Universidad de Rutgers. Allí conocí a Georgiana Haldeman [51] quien es estudiante de doctorado dirigida por Thu Nguyen. En este proceso también participó Marco Moresi como parte de su trabajo final de su Licenciatura en Ciencias de la Computación que dirigimos junto a Luciana Benotti.

Formé parte de los proyectos de Google Traiblazer, Google Rise y Google CS4HS. Presen-

[5]https://mumuki.io/home/

té trabajos en la conferencias internacionales “Innovation and Technology in Computer Science Education (ITiCSE)”, “Special Interest Group on Computer Science Education (SIGCSE)” y “Robotics in Education (RIE)”. En las mismas pude compartir trabajos, experiencias con investigadores como Tim Bell, Moti Ben-Ari y formar parte de Working Groups donde participé de la escritura de una publicación junto a investigadores de todo el mundo.

1.4. Publicaciones

En está sección presentamos las publicaciones internacionales y nacionales publicadas durante el desarrollo de este trabajo. En el Apéndice C están recopiladas la primera página de cada una de las publicaciones internacionales.

1.4.1. Internacionales

- María Cecilia Martínez, Marcos J. Gómez, Luciana Benotti. *A Comparison of Preschool and Elementary School Children Learning Computer Science Concepts through a Multilanguage Robot Programming Platform*. Proceedings of the 20th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITICSE 2015). **Número de citas: 36**.

- M. Cecilia Martinez, Marcos J. Gómez, Marco Moresi y Luciana Benotti. *Lessons Learned on Computer Science Teachers Professional Developmen*t. Proceedings of the 21th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITICSE 2016). **Número de citas: 15**.

- Luciana Benotti, Marcos J. Gómez, and M. Cecilia Martinez. *UNC++Duino: A kit for learning to program robots in Python and C++ starting from blocks*. Robotics in Education (pp. 181-192). **Número de citas: 5**.

- Allan Fowler, Johanna Pirker, Ian Pollock, Bruno Campagnola de Paula, Maria Emilia Echeveste, and Marcos J. Gómez. 2016. *Understanding the benefits of game jams: Exploring the potential for engaging young learners in STEM*. In Proceedings of the 2016 ITiCSE Working Group Reports (ITiCSE ’16). **Número de citas: 13**.

- Luciana Benotti, Federico Aloi, Franco Bulgarelli, and Marcos J. Gomez. *The Effect of a Web-based Coding Tool with Automatic Feedback on Students’ Performance and Perceptions*. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE ’18). **Número de citas: 14**.

- Marcos J. Gómez, Marco Moresi y Luciana Benotti. *Text-based programming in elementary school: A comparative study of programming abilities in children with and without block-based experience*. Proceedings of the 24th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITICSE 2019). Association for Computing Machinery. **Número de citas: 0**.

1.4.2. Nacionales

- Marcos J. Gómez. *Ciencias de la Computación para el aula: 2do ciclo de Primaria (1 edición)*. Autor Capítulo 4: Procedimientos. Fundacion Sadosky: Program.ar, Ciudad Autónoma de Buenos Aires. Agosto 2018. ISBN 978-987-27416-5-5.

- Hector García, Marcos J. Gómez, Gabriel Scarano, Natalia Zalazar. *Programar en la Escuela Pública y en la Educación Formal, el diseño curricular del Programa ProA*. Poster presentado en las 1eras Jornadas Argentinas de Didáctica de la Programación (JADiPRO 2018).

- Marco Moresi y Marcos J. Gómez. *Generación Automática de Feedback Formativo en Enseñanza de la Programación Aplicado a Mumuki*. Poster presentado en las 1eras Jornadas Argentinas de Didáctica de la Programación (JADiPRO 2018).

- María Cecilia Martinez y Marcos Javier Gomez. *Programar computadoras en Educación Infantil*. Edutec. Revista Electrónica de Tecnología Educativa 65 (2018): 40-53.

- Marcos J. Gómez. *Entornos digitales inteligentes para la enseñanza de la programación en la escuela primaria*. 2das Jornadas Argentinas de Didáctica de la Programación (JADiPRO 2019).

1.5. Mapa de la tesis

Esta tesis está compuesta por seis capítulos además de la introducción. A continuación presentamos cada uno de estos capítulos, y su rol en el desarrollo de los objetivos generales de este trabajo.

1.5.1. Cap. 1: Introducción

En este capítulo presentamos las motivaciones que nos llevaron a pensar en la enseñanza de la programación y la adquisición de un nuevo lenguaje de programación como ejes del desarrollo de este trabajo. En particular, la importancia de aprender a programar, entender las computadoras para vivir en el mundo de hoy. Introducimos los tres aspectos de adquisición de lenguaje que estudiamos en relación a la enseñanza de programación: expresividad, interactividad y fluidez. También presentamos una lista de publicaciones realizadas durante el desarrollo de esta tesis y describimos cómo el entorno provincial, nacional e internacional ha colaborado a lo que aquí se presenta.

1.5.2. Cap. 2: Formación docente para la enseñanza de programación

Para poder enseñar programación es importante contar con docentes que puedan enseñar a programar. Ante la falta de docentes formados para llevar la enseñanza de programación a todos los niveles educativos, diseñamos e implementamos un curso de formación docente. Evaluamos el impacto del curso en los docentes teniendo en cuenta dos ejes centrales: la incorporación conceptual de los conceptos fundamentales de programación y el efecto del curso en las prácticas de enseñanza en los docentes.

1.5.3. Cap. 3: Trabajo previo

En este capítulo describimos trabajo previo relacionado a los tres aspectos que estudiamos: la expresividad, la interactividad y la fluidez (y su relación con la evaluación automática). Para poder describir cada uno de estos aspectos, ejemplificamos utilizando entornos de enseñanza de programación existentes. Los entornos de enseñanza de programación son herramientas muy

utilizadas al momento de llevar la enseñanza de programación en diferentes contextos educativos. También realizamos una breve introducción al surgimiento de lenguajes de programación específicamente diseñados para la enseñanza.

1.5.4. Cap 4: Expresividad del lenguaje de programación

En este capítulo describimos el impacto del nivel de expresividad de un lenguaje al momento de adquirir un nuevo lenguaje. Presentamos en este capítulo el entorno multi lenguaje UNCDuino desarrollado durante esta tesis. UNCDuino, en base a la expresividad de los lenguajes de programación con los que cuenta, se clasifica como un lenguaje que traduce automáticamente de lenguajes de menor a mayor expresividad. UNCDuino permite programar kits de robótica Arduino en cuatro lenguajes de programación: Icony, Blockly, Python, C++. Los lenguajes están organizados de menor a mayor nivel de expresividad. Icony es un lenguaje icónico que diseñamos junto a docentes de nivel inicial para que estudiantes que no saben leer ni escribir puedan aprender a programar. Además describimos y analizamos diferentes experiencias implementadas para enseñar a programar utilizando UNCDuino en contextos educativos reales.

1.5.5. Cap. 5: Interactividad formativa

En este capítulo analizamos el impacto de la interactividad formativa automatizada sobre distintos tipos de errores para la adquisición de un nuevo lenguaje. Utilizamos el entorno cerrado Mumuki para implementar dos experiencias de enseñanza de programación en dos niveles educativos diferentes. Una de las experiencias se implementó en dos cursos universitarios. La otra experiencia se implementó en dos escuelas primarias. En el caso de la experiencia implementada en el nivel universitario, se evaluó el efecto del entorno Mumuki en cuanto a las métricas cuantitativas y cualitativas al adquirir el lenguaje de programación Haskell. En el nivel primario, analizamos y evaluamos el impacto de utilizar un entorno cerrado como Mumuki para adquirir un lenguaje basado en texto. En ambas experiencias comparamos grupos con y sin conocimiento previo en programación.

1.5.6. Cap. 6: Evaluación automática de la fluidez

En este capítulo describimos y analizamos diferentes métodos implementados utilizando técnicas de procesamiento de lenguaje natural y de aprendizaje automático para evaluar el nivel de fluidez de los estudiantes durante el proceso de adquirir un nuevo lenguaje de programación. En base a un programa creado por un estudiante que está aprendiendo a programar, comparamos distintos métodos que predicen si el estudiante podrá resolver con éxito el ejercicio de programación usando sólo feedback formativo automatizado y sin ayuda docente personalizada. Comparamos el desempeño de modelos creados usando ingeniería de características basados en teorías pedagógicas, con modelos modelos de redes neuronales recurrentes y representaciones basadas en teorías lingüísticas de semántica distribucional diseñadas para el procesamiento de lenguaje natural.

1.5.7. Cap. 7: Conclusiones y trabajo futuro

En este capítulo presentamos un resumen de las experiencias implementadas y las conclusiones de cada uno de los experimentos realizados en cuanto a la adquisición de un lenguaje al enseñar a programar. La falta de conocimiento sobre una didáctica específica para el área y la escasez de docentes formados son problemas compartidos por los países que quieren implementar

la enseñanza de la programación en las escuelas. A lo largo de esta tesis vamos experimentando y diseñando enfoques para motivados por estos problemas desde la perspectiva del área de adquisición de lenguaje. Presentamos conclusiones obtenidas de implementar experiencias de enseñanza de programación en contextos educativos reales en distintos niveles: desde el nivel inicial hasta el universitario.

1.5.8. Apéndices

La tesis cuenta con tres apéndices. En el primero de ellos describimos la interfaz del entorno de enseñanza de programación Mumuki para dos tipos de usuario: estudiante y docente. En el segundo apéndice describimos y ejemplificamos características definidas para capturar diferentes aspectos de los estudiantes y ejercicios. El objetivo de estos dos primeros apéndices es permitir la reproducibilidad de este trabajo. En el tercer apéndice compartimos la primer hoja de cada uno de las publicaciones internacionales presentadas.

Capítulo 2

Formación docente para la enseñanza de programación

Como describimos en el Capítulo 1 existen numerosas iniciativas y un importante interés por parte de un gran número de países en introducir las Ciencias de la Computación y la programación como parte de la currícula obligatoria de los sistemas educativos. En Argentina, desde 2010 los miembros de la Secretaría Nacional de Ciencia e Innovación Productiva junto con las universidades nacionales han estado promoviendo la enseñanza de programación en la escuela. En Agosto del 2015, el Consejo Federal de Educación (CFE), a través de la Resolución Nº 263/15, declaró de importancia estratégica a la enseñanza y el aprendizaje de la Programación en todas las escuelas, durante la escolaridad obligatoria. A partir de la decisión política e interés de poder llevar la enseñanza de la programación en todos los niveles educativos, se observa una problemática central al momento de querer implementar en contextos reales: actualmente no hay suficientes docentes formados para enseñar a programar en las escuelas. Además de ser necesarios un diseño curricular y la intención política de querer llevar la enseñanza de la programación a las escuelas, es imprescindible contar con docentes formados y calificados para llevar a cabo la tarea. Uno de los principales desafíos para la enseñanza de programación es, entonces, poder formar docentes.

Con el propósito de investigar cómo los docentes incorporan conceptos de programación y analizar si experiencias de formación profesional (FP) existentes son suficientes para capacitar a los docentes, diseñamos un estudio exploratorio que presentamos en este capítulo y cuyos resultados principales descriptos fueron publicados en [74].

En la Sección 2.1 introducimos brevemente experiencias previas desarrolladas a nivel mundial relacionadas a la implementación de cursos de FP para docentes. Luego, en la Sección 2.2 describimos una propuesta particular de FP implementada a gran escala en Argentina. En la Sección 2.3 presentamos las herramientas de recolección de datos que definimos para poder evaluar el impacto de la propuesta de FP. En la Sección 2.4, en base a los datos recolectados analizamos el impacto del curso de FP en cuanto a la incorporación de los conceptos y en las prácticas de enseñanza en los docentes. Finalmente, en la Sección 2.5 describimos conclusiones preliminares del estudio implementado que motivan la necesidad de contar con entornos de soporte para enseñanza de programación como los que se presentan en el Capítulo 3.

2.1. Experiencias previas

A medida que muchos países avanzan en sus esfuerzos por introducir la enseñanza de la programación en la currícula escolar obligatoria, un tema de debate entre académicos, responsables políticos y toda la comunidad educativa es quién enseñará a programar en las escuelas y cómo serán formados los docentes. Actualmente, uno de los principales desafíos para la enseñanza de la programación es la falta de docentes con conocimientos en el área. Una realidad que comparte nuestro país con muchos otros es que la cantidad de estudiantes que eligen carreras universitarias relacionadas a la programación no logra satisfacer la oferta laboral. Por lo tanto, si no es posible satisfacer la oferta laboral en la industria del software, mucho más complejo es conseguir docentes de programación.

Si bien la mayoría de los investigadores e impulsores de políticas públicas coinciden en que las carreras terciarias o universitarias son la mejor opción para formar docentes calificados [101], ante la falta de docentes para enseñar programación, muchos países capacitan docentes que ya están en actividad (como Reino Unido, Nueva Zelanda, Estados Unidos y Alemania) [117, 47, 85, 38]. La falta de programas de formación docente en programación coherentes, docentes de programación en actividad con formación heterogénea y la falta de profesionales de la industria del software interesados en la enseñanza, han motivado a muchos países a formar docentes a partir de cursos de corto plazo.

Investigaciones previas relacionadas a programas de FP docente [19], sugieren que a pesar de los esfuerzos de ofrecer cursos para docentes, la mayoría de estos programas presentan contenidos sin articulación entre sí, superficiales y no tienen en cuenta la amplia literatura pedagógica en FP. Sin embargo, hay vasto conocimiento pedagógico en qué características de los programas de FP son beneficiosos para promover el aprendizaje de los docentes. Los investigadores han documentado por ejemplo que los programas de FP, que incluyen un enfoque explícito en el tema a desarrollar y analizan el pensamiento de los estudiantes, son beneficiosos para promover el aprendizaje de los docentes [43]. En el resto de este capítulo describimos un enfoque explícito de FP que tiene en cuenta la amplia literatura pedagógica en el área, y evaluamos su impacto en diversas dimensiones.

Un punto crucial a tener en cuenta para la evaluación de la propuesta de FP presentada en este capítulo es que, los docentes de programación que forman parte del sistema educativo actual, no tienen necesariamente conocimientos previos en la disciplina, siendo muchos de ellos provenientes de áreas relacionadas a las Tecnologías de la Información y la Comunicación (TIC), por lo que hacen foco en enseñar a utilizar programas en vez de centrarse en enseñar a programar [117, 47, 38]. En su gran mayoría, los docentes que se encargan de enseñar a programar, no tienen formación formal en programación [47, 85, 86]. En la próxima sección describimos en detalle el grupo de docentes que formó parte de la FP que evaluamos.

Si bien la mayoría de las investigaciones realizadas [38, 117] parecen mostrar que los docentes se sienten más seguros con respecto a la enseñanza de la programación luego de participar de cursos de FP, no es claro cómo los docentes incorporan en sus clases los contenidos aprendidos en los cursos y qué relación existe entre las actividades propuestas en los cursos FP y las prácticas de enseñanza que luego los docentes llevan a sus aulas. En base a esta situación, diseñamos una propuesta de Formación Profesional en programación para poder evaluar el impacto en los docentes, tanto en la incorporación de conceptos de programación como en la implementación de la enseñanza de la programación en sus aulas.

2.2. Propuesta de formación profesional

A partir del año 2013 organizamos y desarrollamos cursos de FP de programación para docentes de nivel primario y secundario en la Universidad Nacional de Córdoba. Los cursos tuvieron una duración total de 50 horas presenciales anuales. Las clases se distribuyeron durante un semestre escolar. 40 de las horas del curso fueron implementadas en la universidad en formato de taller. Cada taller tenía una duración de entre 4 y 6 horas diarias, y los mismos se realizaban cada tres semanas. Para completar las 50 horas de cursado, los docentes tenían que desarrollar 10 horas de práctica docente, implementando la enseñanza de programación en sus escuelas. En la escuela, los docentes eran acompañados por tutores de nuestro equipo, quienes eran estudiantes universitarios avanzados en carreras de computación. Cada docente contaba con un tutor, a quién podía pedir ayuda para implementar y planificar las clases. Los tutores ayudaban a los docentes en la elección del entorno de programación a utilizar durante la práctica docente y proponían diferentes estrategias para que las clases se enfoquen en los conceptos de programación y no en el entorno elegido. Los tutores tenían una disposición de 4 horas por docente. Las 4 horas eran extras a las 50 horas presenciales obligatorias del curso.

Durante el curso de formación docente, se usaron tres herramientas: programación de animaciones, chatbots y robots. Los conceptos de programación introducidos fueron: secuencia, evento, condicional, ciclo, variable, uso y definición de procedimientos, parámetros, entre otros. Para cada una de las herramientas se utilizó un entorno de programación distinto. Para programar animaciones utilizamos Alice [30]. Para programar chatbots y robots utilizamos entornos desarrollados en la Universidad Nacional de Córdoba: Chatbot [14] y UNCDuino [72, 13]. En el Capítulo 3 compararemos distintos tipos de entornos existentes para la enseñanza de la programación. En el Capítulo 4 describiremos el entorno UNCDuino que fue desarrollado en marco de esta tesis con el objetivo de satisfacer la demanda de docentes de nivel inicial cuyos estudiantes aún no saben leer y escribir en ningún idioma.

A pesar de utilizar 3 herramientas distintas en el curso de formación docente, siempre trabajamos los mismos conceptos de programación, con el objetivo de que los docentes no sean dependientes de un entorno en particular ni de un lenguaje, ni tampoco de un tipo de aplicación como son la robótica, los chatbots o las animaciones. Cada uno de los talleres desarrollados en la universidad siguieron un enfoque basado en exploración. En los primeros 10 minutos del taller, introducimos el entorno de programación seleccionado. Antes de presentar un nuevo concepto, proponemos un desafío a los docentes que necesita el nuevo concepto para ser resuelto. Por ejemplo, programar al robot utilizando el entorno UNCDuino para que sea capaz de esquivar objetos con el objetivo de aprender el concepto de condicional. Los docentes resolvían cada uno de los desafíos planteados en grupos de 4 o 5 participantes. Al momento de resolver los desafíos, los tutores estaban a disposición de los docentes para poder prevenir frustración al resolver los ejercicios planteados. Intencionalmente, los tutores fueron capacitados para que guíen a los docentes con nuevas preguntas, y no con la solución al problema. Una vez que la necesidad del concepto de programación era expresada por los docentes, se presentaba el concepto. Esto se conoce como experiencia de aprendizaje significativa. Se eligió esta estrategia didáctica para intentar que los docentes la reproduzcan en sus aulas. En base a la literatura existente, sabemos que la mayoría de los docentes tienden a enseñar con las mismas estrategias de enseñanza con las que aprendieron, y que el aprendizaje de teorías pedagógicas no puede competir con lo que los docentes experimentaron como estudiantes [23]. También sabíamos que la mayoría de los docentes que participaban del curso no tenían experiencia previa en el aprendizaje de programación, por lo tanto, desarrollamos una experiencia de aprendizaje de programación significativa para tener impacto [129]. En la próxima sección describimos el diseño del estudio para poder evaluar el

impacto de la propuesta de formación descripta en la incorporación conceptual por parte de los docentes con diferentes conocimientos previos.

2.3. Diseño del estudio

Para poder conocer el impacto de nuestra propuesta de formación en programación en los docentes en base los cursos ofrecidos en 2014 y 2015, realizamos estudios exploratorios. Específicamente, queríamos documentar qué conceptos de programación y estrategias de enseñanza utilizan los docentes para enseñar sus clases, y cómo estas estrategias se relacionan con las actividades que ofrecemos en los cursos de FP. La sección está organizada de la siguiente manera. Primero, describimos la muestra de docentes participantes de los cursos de formación. Luego, presentamos los métodos de recolección de datos utilizados para poder evaluar el impacto de las experiencias desarrolladas.

2.3.1. Descripción de los docentes participantes

La inscripción a los cursos de formación fue abierta para todos aquellos docentes de nivel primario y secundario que estuviesen dispuestos a llevar la enseñanza de programación a sus aulas. Publicitamos los cursos por redes sociales, la página web de la Universidad Nacional de Córdoba y mediante programas televisivos. Por lo tanto, nuestra muestra está conformada por docentes que eligieron formar parte de la experiencia con acceso a los canales de comunicación anteriormente mencionados. El curso fue gratuito para los docentes. En el año 2014 el curso de FP fue financiado por el programa Google for Education, mientras que en 2015 la financiación estuvo a cargo de la Fundación Sadosky.

Todos los requisitos para completar el curso fueron informados por adelantado. 46 docentes finalizaron el curso en 2014, mientras que 60 pudieron completarlo en 2015. Por diferentes razones (personales, institucionales y cognitivas), alrededor del 10 % de los docentes registrados no pudieron cumplir con todos los requisitos para completar el curso.

La formación y el perfil de los docentes participantes fue muy diversa. En la Tabla 2.1 podemos observar el perfil de los docentes que se registraron en el curso de FP en los años 2014 y 2015. 75 % de los docentes que se registraron se desempeñaban en escuelas públicas, el resto formaba parte de escuelas privadas que cuentan con subsidios estatales. 20 % de los docentes registrados enseñaban en escuelas primarias, el 60 % en escuelas secundarios y el 20 % restante en nivel terciario.

Los maestros de escuela primaria en Argentina tienen una preparación general en pedagogía y formación en contenido específico de la escuela primaria. La mayoría de los programas de formación de maestros de primaria incluyen un curso llamado “tecnología” y otro curso llamado “medios audiovisuales”. Pero ninguno incluye contenido formal de programación. Entre los docentes de las escuelas secundarias, el 22 % de ellos tenían un título en un campo relacionado con programación, como técnico de programación, técnico en informática, analista en sistemas o similares. El 20 % de los docentes tenía un título terciario en tecnología educativa. La mayoría de los programas de tecnología educativa forman a los docentes para integrar las TIC en las escuelas, pero no abordan la programación. El resto de los docentes tenían títulos en matemática, lengua, arte y química.

En la Tabla 2.1 podemos ver que el 58 % de los docentes participantes no tenían formación en programación ni tecnología educativa. Solo el 20 % de los docentes tenía formación en programación. Muchos de los docentes que contaban con un título en tecnología educativa enseñaban

materias relacionadas computación o informática. Martinez y Echeveste en un estudio descripto en [73] documentaron que en los espacios curriculares relacionados a computación no se ofrecían contenidos relacionados a computación o programación, probablemente debido a la falta de conocimientos en programación por parte de los docentes a cargo.

Perfil general de los docentes	**%**
Docentes de escuela públicas	75 %
Docentes de escuela privada	25 %
Docentes de nivel primario	20 %
Docentes de nivel secundario	60 %
Docentes de terciario	20 %
Docentes con título relacionado a programación	22 %
Docentes con título en tecnología educativa	20 %
Docentes sin formación en programación o tecnología	58 %
Docentes que dictan espacios curriculares de tecnología	13 %
Docentes que dictan espacios curriculares programación	25 %
Docentes que dictan matemática	20 %
Docentes que dictan otros espacios curriculares	42 %

Tabla 2.1: Distribución de la matrícula de docentes en los cursos según el sector, público o privado, según el nivel de enseñanza, primario, secundario y terciario, según formación y espacios curriculares que dictan (n=106).

2.3.2. Recolección y análisis de datos

Como parte del estudio exploratorio, los tutores participaron de las clases de los docentes para completar con los requisitos del curso. Los tutores escribieron observaciones semiestructuradas después de participar de cada una de las clases. Las observaciones incluyeron elementos tales como datos demográficos sobre los estudiantes y la escuela, los conceptos y contenidos de programación que el docente introducía en la clase, descripción de la experticia que tenía el docente al momento de presentar los conceptos elegidos, descripción de las estrategias de enseñanza utilizadas, el mejor momento de la clase y las partes de la clase que podrían haberse hecho de manera diferente. En total, reunimos 73 observaciones de tutores entre las ediciones de 2014 y 2015. Debido a que los docentes se registraron en parejas, algunos de ellos también decidieron dar clases en pareja por lo que los tutores sólo escribieron una observación para ambos docentes. Fueron pocos los casos (en total 7) donde los docentes no quisieron recibir ayuda ni visita de los tutores.

A partir de las descripciones obtenidas de las observaciones de clases realizadas por los tutores, identificamos ciertos temas emergentes, y construimos categorías para poder analizar las observaciones realizadas por los tutores. 9 personas, incluyendo tutores e investigadores, se encargaron de analizar y categorizar cada una de las observaciones. Para garantizar la confiabilidad, cada observación fue categorizada por más de un responsable. En la próxima sección describimos los resultados obtenidos a partir del análisis de los datos recolectados.

2.4. Resultados

En esta sección, describimos nuestros hallazgos con respecto al aprendizaje de programación por parte de los docentes durante el curso de FP y cómo pudieron aplicar los conceptos incorporados en sus aulas.

En base al análisis de las observaciones realizadas por los tutores en las clases a cargo de los docentes, identificamos que en el 75 % de las clases, los docentes intentaron incorporar y explicar diferentes conceptos fundamentales de programación trabajados en el curso de FP. A partir del análisis sobre cada observación identificamos diferentes niveles de experticia al momento de explicar los conceptos. En la Tabla 2.2 observamos la distribución de las 73 observaciones de clases en base a la experticia al momento de introducir un concepto de programación por parte del docente.

El docente no pudo explicar el concepto.	5 %
El docente explicó el concepto de memoria.	3 %
El docente explicó el concepto con errores.	17 %
El docente explicó el concepto correctamente pero superficialmente.	35 %
El docente explicó el concepto correctamente utilizando analogías, ejemplos y situaciones prácticas.	23 %
Otros	17 %

Tabla 2.2: Explicación de los docentes de conceptos de programación

El 58 % de los docentes pudo explicar los conceptos de programación elegidos de forma correcta, mientras que solo un 23 % mostró realmente un nivel de experticia y confianza considerable. A continuación podemos observar una reflexión realizada por uno de los tutores en una de las observaciones.

"Es la primera vez que observo a un docente explicando los conceptos de método y parámetro. Y lo hizo perfectamente. Para ello, basó su clase en los diferentes video tutoriales presentados durante el curso, pero sin utilizar el video en clase, explicó a los estudiantes como crear un programa para que un personaje en Alice pueda caminar. Para ello utilizó el método "posar". Primero, explicó como crear un nuevo método y preguntó a los estudiantes como pensaban que debían hacer para que el personaje pudiera caminar cerca de un objeto..."

Los conceptos de programación más frecuentes que los docentes explicaron fueron estructura condicional, ciclo, variable, secuencia, métodos, números aleatorios, objetos y eventos. Los docentes enseñaron al menos tres de estos conceptos en aproximadamente el 70 % de las clases observadas. Parámetro, números binarios, constantes, redes y otros temas fueron menos frecuentes y se presentaron en el 30 % de las lecciones observadas. El enfoque basado en la exploración permitió integrar más de un concepto en una clase.

Casi el 20 % de los docentes confundieron conceptos de programación con elementos propios de los entornos de enseñanza de programación que utilizaron. Por ejemplo, uno de los docentes que decidió trabajar con el entorno Alice, consideró la selección de personajes y escenas como un concepto de programación.

Los docentes que no pudieron enseñar conceptos de programación de forma confiable no tenían formación previa en programación. También observamos que los docentes con formación en tecnología educativa tuvieron dificultades, como mencionamos anteriormente [73]. Esta certificación prepara a los docentes para usar software educativo en las escuelas, pero no aborda los conceptos relacionados a la programación. Este hallazgo sugiere que los maestros sin experiencia

previa en programación necesitan una formación más extensa que un curso de 50 horas.

2.5. Conclusiones del capítulo

Una limitación en el estudio desarrollado, es que sólo observamos aquellas clases que los docentes debían de dictar como parte del curso de FP. Además, quienes observaron las clases, fueron los tutores que dieron soporte a los docentes, por lo que pueden haber influenciado en la práctica docente. Por lo tanto, no sabemos sobre los efectos a largo plazo de nuestro curso. Sin embargo, la evidencia presentada aquí muestra que un curso de FP en programación que incluye experiencias de programación de primera mano, debates curriculares entre docentes y práctica en el aula con el apoyo de tutores, contribuyó a aumentar los conocimientos de los docentes en programación y la implementación de clases de programación basadas en la exploración.

Implementar cursos de FP docente durante dos años nos permitió comenzar a entender de qué forma los docentes aprenden programación e implementan la enseñanza de la programación en las escuelas. La mayoría de los docentes que participaron de los cursos, tenían una concepción errónea sobre la programación, muchos entendían que enseñar programación es equivalente a enseñar ofimática y uso de las TICs. Algunos docentes tenían conocimientos en programación, pero en base a los diseños curriculares creían no necesario enseñar a programar. En muchos casos, debido a que los docentes tuvieron experiencias difíciles al aprender a programar y desconocían la existencia de recursos didácticos y entornos de enseñanza de programación recientemente desarrollados, no podían imaginar cómo enseñar a programar a estudiantes de primaria y secundaria. Comprender las creencias y necesidades de los docentes es necesario para diseñar cursos de FP efectivos. La reflexión y los debates sobre las prácticas de los docentes contribuyeron a cambiar las percepciones de los docentes sobre la enseñanza de la programación.

Además, los docentes pudieron incorporar nuevas estrategias pedagógicas basadas en exploración y resolución de problemas con experiencias de primera mano. Aprender sobre pedagogía es tan importante como incorporar los conceptos fundamentales de programación, ya que para generar impacto real en la enseñanza, es necesario tener herramientas para poder implementarla. La resolución de los desafíos durante el curso de FP, contar con tutores para poder diseñar e implementar sus prácticas, proporcionó a los docentes recursos pedagógicos para llevar la programación a sus aulas. Requerir las prácticas de los docentes en el aula como parte de las horas del curso de FP fue esencial para promover la implementación en el aula.

Si bien nuestros cursos fueron implementados con un diseño pedagógico sólido, nuestra conclusión principal es que esta capacitación fue insuficiente para preparar a los docentes sin formación previa en programación. Los docentes sin formación previa abordaron sin profundidad y con errores importantes los conceptos de programación en sus aulas. Esta tesis estudia distintas variables involucradas en la complejidad de enseñar programación considerando aspectos fundamentales en la adquisición de un lenguaje. Desde la perspectiva de los docentes, nuestro estudio de aspectos de adquisición del lenguaje puede contribuir a reflexionar sobre las preguntas recurrentes: Qué lenguaje, qué entorno y cómo evaluar la enseñanza de programación.

Capítulo 3

Trabajo Previo

Como describimos en el Capítulo 1, hay un enorme interés por parte países, empresas e investigadores de poder llevar la enseñanza de la programación a las escuelas. Pero como observamos en el Capítulo 2 la falta de docentes formados en programación y el poco impacto de los cursos de formación de docentes en actividad en la incorporación de los conceptos fundamentales de programación dificultan la implementación de la enseñanza de programación. Teniendo en cuenta estas complejidades, en este capítulo definimos variables que impactan en la enseñanza de programación a principiantes. Ilustramos la discusión describiendo trabajo previo.

Si bien este capítulo describe ciertos lenguajes y entornos de enseñanza de programación que han sido utilizados para que principiantes aprendan a programar, no pretende presentar un listado completo de tales entornos sino dar definiciones básicas que serán útiles en esta tesis e ilustrarlas con trabajo previo. Para leer una clasificación y descripción de los numerosos lenguajes y entornos existentes de enseñanza de programación más utilizados recomendamos [22].

Este capítulo está organizado de la siguiente manera. Primero, en la Sección 3.1 introducimos brevemente la creación de lenguajes y entornos específicos para la enseñanza de programación y sus entornos de aplicación. Luego, en la Sección 3.2 comparamos lenguajes usados para la enseñanza de programación en base a su nivel de expresividad. En la Sección 3.3 definimos y ejemplificamos los entornos de enseñanza de programación teniendo en cuenta el tipo de interactividad que permiten durante el proceso de adquisición del lenguaje de programación. Finalmente, en la Sección 3.4 describimos cómo las variables anteriores impactan métodos existentes para la evaluación formativa y discutimos su escalabilidad.

3.1. Lenguajes para enseñar a programar y sus aplicaciones

Poder llevar la enseñanza de la programación a todos los niveles educativos no es una idea que ha surgido en los últimos años. En la década del 70 podemos encontrar las primeras experiencias de enseñanza de programación para niños de escuela primaria, por ejemplo. En 1976, Solomon y Papert presentaron en [114] una experiencia donde una niña de de 7 años aprendía a programar utilizando un lenguaje de programación pensado y diseñado para enseñar a programar.

El trabajo realizado por Papert y el lenguaje de enseñanza de programación Logo [90, 104] ha influido en el desarrollo de los lenguajes de enseñanza de programación y de los entornos existentes. Logo fue diseñado para poder enseñar a programar a principiantes, haciendo foco en estudiantes de escuela primaria. Es conocido como el "lenguaje de la tortuga" ya que permite programar a una tortuga robótica para que se mueva y dibuje utilizando instrucciones sencillas. En la Figura 3.1 podemos ver a Papert con la tortuga robótica. Logo está compuesto por ins-

trucciones simples como *forward, left, right* que permiten crear programas simples que hacen que la tortuga robótica se mueva. A estos programas simples, Papert et al. le llaman el "piso bajo" de Logo [90], implicando que los estudiantes pueden ver un efecto concreto fácilmente apenas empiezan a aprender. El dibujo del pez que se observa en la parte inferior de la figura fue dibujado por la tortuga robótica siguiendo instrucciones del lenguaje Logo. La *aplicación* de Logo es mover la tortuga y dibujar, pero su objetivo es enseñar conceptos fundamentales de la programación. Por ejemplo, Logo [90, 104] permite enseñar recursión y luego *aplicar* este conocimiento para hacer dibujos complejos. La habilidad del lenguaje de permitir la enseñanza de conceptos avanzados Papert et al. [90] la describen como "techo alto". Papert et al. dirían que diseñaron un lenguaje con instrucciones simples para enseñar a programar pensado para tener un "piso bajo pero un techo alto".

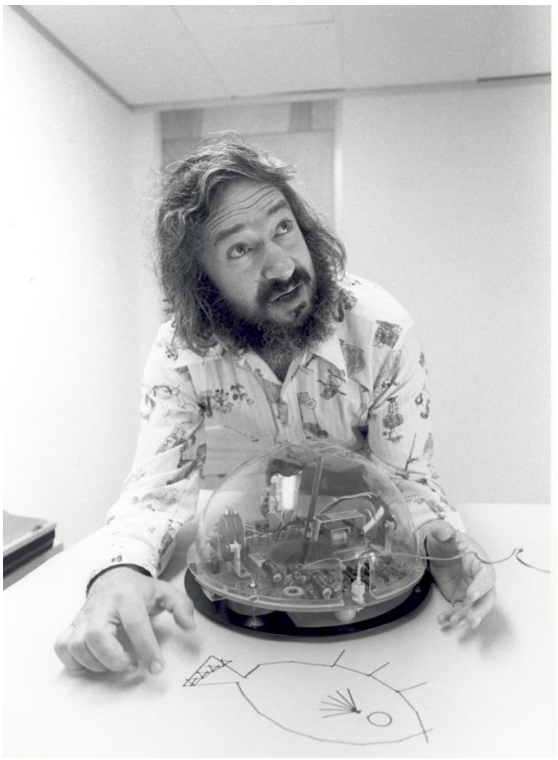


Figura 3.1: Papert con la tortuga robótica programable con el lenguaje Logo. El dibujo del pez que se observa en la parte inferior fue dibujado por la tortuga robótica siguiendo instrucciones del lenguaje Logo.

En base a lo descripto anteriormente, podemos ver dos ejes centrales sobre los cuáles Papert hizo foco y generó impacto en el área de enseñanza de la programación:

1. La idea de contar con lenguajes de programación específicos para enseñar a programar a principiantes. Estos entornos y lenguajes pensados con piso bajo y con techo alto.

2. A diferencia de los lenguajes de programación conocidos hasta el momento, que eran de propósito general, estos nuevos lenguajes de programación para enseñar a programar se pensaron de propósito específico, con un área de *aplicación* concreta.

Logo usa la robótica educativa y el dibujo como un motivador para la enseñanza de la programación. Así la *aplicación* del lenguaje toma un rol central, a veces perdiéndose de vista que el objetivo es la enseñanza de la programación y no la aplicación. El objetivo no es mover

la tortuga y dibujar, el objetivo es aprender a programar. Armoni et al. en [3], luego de revisar diferentes experiencias documentadas, clasifican las experiencias de enseñanza de programación para niños en tres categorías teniendo en cuenta la *aplicación* como variable.

Programación visual: Lenguajes que permiten a los estudiantes crear sus propias animaciones, videojuegos, simulaciones, etc. Alice [30], Scratch [103, 70], MicroMundos [37], son entornos que ejemplifican esta categoría. Las producciones creadas por los estudiantes tienen efecto sobre el mundo digital.

Actividades kinestésicas: Actividades que no utilizan computadoras para la enseñanza de programación. El estudiante usa su propio cuerpo, su movimiento y juegos grupales para "actuar" conceptos de algoritmos, programación y otras áreas de ciencias de la computación. Ejemplos de este tipo de actividades se pueden encontrar en [11, 8, 20].

Robótica educativa: actividades relacionadas a la programación de kits de robótica son muy utilizadas para llevar la enseñanza de la programación a principiantes [33]. Al igual que las actividades kinestésicas, el efecto de los mismos se observa en el mundo real.

Por lo tanto, podemos clasificar las experiencias de enseñanza de programación en dos categorías más generales: aquellas que tienen efecto sobre el mundo real y aquellas que tienen efecto sobre el mundo digital. Los entornos de enseñanza de programación relacionados a la robótica son los que se vinculan de forma directa con el efecto sobre el mundo real pero también lo son las que involucran actividades kinestésicas. Para poder observar el efecto de los programas creados es necesario contar con kits de robótica específicos, y ver el efecto en el mundo real. Lego Mindstorms [121, 106], BlocklyDuino [66], Kibo [116] o Thymio [82, 133] son ejemplos de este tipo de entornos.

La transición de Logo a entornos visuales como MicroMundos, fue el principio de los entornos con efecto sobre el mundo digital. De necesitar la tortuga robótica para ver el efecto de los programas creados en Logo, a ver directamente el efecto del mismo en la pantalla. Entornos como Alice o Scratch permiten la creación de videojuegos y animaciones, con un efecto claro sobre el mundo digital.

La *aplicación de* los lenguajes de enseñanza de programación toma un rol demasiado relevante en detrimento de otras variables importantes, no tan obvias a primera vista, que afectan la enseñanza de la programación. En las próximas 3 secciones describimos 3 de estas variables: la expresividad del lenguaje, la interactividad formativa que permite el entorno y la evaluación de la fluidez.

3.2. Expresividad del lenguaje de programación

Los lenguajes de programación existentes, pueden clasificarse en varias categorías dependiendo de su nivel de expresividad. Teniendo en cuenta su expresividad, una de las clasificaciones más obvias que surgen es la de lenguajes basados en bloques y lenguajes basados en texto. El primer lenguaje de bloques fue Logo Blocks [7]. El mismo fue desarrollado en el año 1996 por el MIT Media Lab. A partir de este surgimiento, los lenguajes de enseñanza se empezaron a clasificar en basados en bloques o basados en texto.

Los entornos de enseñanza de programación basados en texto son aquellos que utilizan lenguajes de programación donde al aprendiz puede escribir texto libremente. Las primeras versiones de Logo [90, 104], Greenfoot [54, 63], Chatbot [14, 15], Gobstones [68] son ejemplos de este tipo

de entornos basados en texto diseñados especialmente para enseñar a programar. En este tipo de entorno, los errores de sintaxis son posibles y frecuentes para los programadores principiantes.

Actualmente, es tema de debate si es mejor enseñar a programar usando lenguajes basados en texto o basados en bloques. Weintrop y Wilensky en [126] se preguntan: "to block or not to block". Uno de los argumentos de los defensores de los entornos basados en bloques es que los frecuentes errores de sintaxis que sufren los programadores principiantes los desmotivan y generan abandono. Uno de los argumentos de los defensores de comenzar directamente con texto es que los lenguajes de bloques sólo retrasan el problema de los errores de sintaxis pero no lo solucionan porque eventualmente, para crear programas interesantes, es necesario aprender a programar con lenguajes basados en texto.

La creciente popularidad de los lenguajes basados en bloques como Scratch [103, 70] y Alice [30] es evidente. Los lenguajes basados en bloques son utilizados para principiantes de todas las edades, principalmente en escuela primaria y secundaria, pero también en nivel universitario [127]. Duncan et al. en [34] realizaron una revisión de los lenguajes de enseñanza de programación usados en secundaria en Estados Unidos. En total analizaron 47 entornos. 31 de los 47 entornos fueron lanzados después del 2010. 28 de los 47 entornos analizados son basados en bloques. Cuándo hacemos referencia a lenguajes basados en bloques nos referimos a lenguajes de programación donde las instrucciones se representan usando un conjunto de bloques predefinido y finito, posiblemente parametrizable como podemos ver en la Figura 3.2. En la figura podemos observar un programa creado con los bloques y la interfaz de Scratch. Scratch es un entorno de enseñanza de programación basado en bloques diseñado por el MIT para ayudar a aprender a programar a principiantes mientras crean sus videojuegos o animaciones. Scratch es utilizado en todo el mundo, y cuenta con una gran comunidad activa.[1]. Los programadores, en vez de escribir código, arrastran y unen los bloques como si fuera un rompecabezas. Sólo las secuencias de bloques sintácticamente permitidas pueden ser unidas. La figura ilustra un bloque de tipo booleano que está por ser ubicado en un espacio hexagonal que acepta bloques de tipo booleano. Hay bloques que no pueden unirse cuando dicha secuencia no corresponde a un programa sintácticamente correcto. Una característica principal de dichos lenguajes es que, todo programa escrito en ellos compila correctamente. Es decir, todos los programas en estos lenguajes son sintácticamente correctos por construcción. Scratch [103, 70], Alice [30] y MIT App Inventor [130, 131] son tres de los entornos basados en bloques más utilizados como se argumenta en [3].

A partir de la figura podemos identificar ciertas ventajas y desventajas de los lenguajes basados en bloques para enseñar a programar a principiantes. Empecemos por las ventajas. En la figura podemos ver que los estudiantes cuentan con categorías de bloques que representan diferentes conceptos. Además tienen todos los bloques a disposición, por lo que no tienen que memorizar instrucciones. Cada bloque se puede comparar con una pieza de rompecabezas. Los bloques proporcionan pistas visuales al usuario sobre cómo y dónde pueden ser usados. La forma de los bloques y los colores facilitan al usuario la tarea de programar. En la figura podemos observar que los bloques se agrupan por categoría, y cada categoría se identifica con un color. Todos los bloques azules se relacionan con instrucciones que permiten mover a los personajes de Scratch. Los bloques naranjas hacen referencia a las estructuras de control. De esta manera los estudiantes pueden ir relacionando conceptos con colores, facilitando la búsqueda de los bloques necesarios. Las formas también caracterizan los conceptos que representa cada bloque. Las bloques que representan los booleanos tienen forma hexagonal, y pueden ser utilizados como condiciones de estructuras condicionales. Para crear sus propios programas, los estudiantes deben de arrastrar bloques y unir los mismos como piezas de rompecabezas. En la figura podemos

[1] Cuenta con 50 millones de usuarios registrados https://scratch.mit.edu/

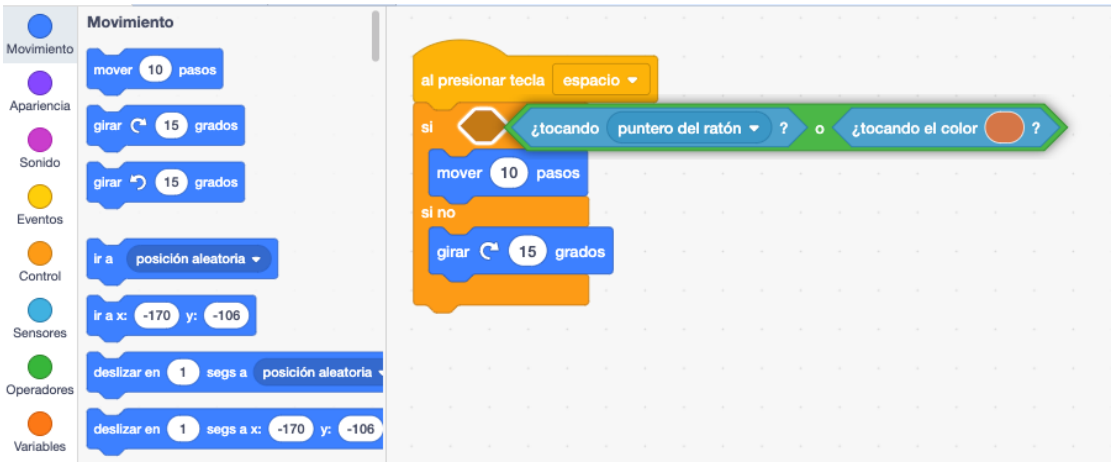


Figura 3.2: Interfaz del entorno de enseñanza de programación Scratch. Cada categoría de bloques se representa con un color específico. Al seleccionar una categoría específica se presenta la lista de bloques correspondiente a la misma. En este caso, el usuario seleccionó la categoría Movimiento, por lo que se observan los bloques de color azul. En el editor podemos observar el programa creado por el usuario. Cada bloque tiene el color de la categoría a la cual pertenece. Por ejemplo los bloques de las estructuras de control tiene color naranja.

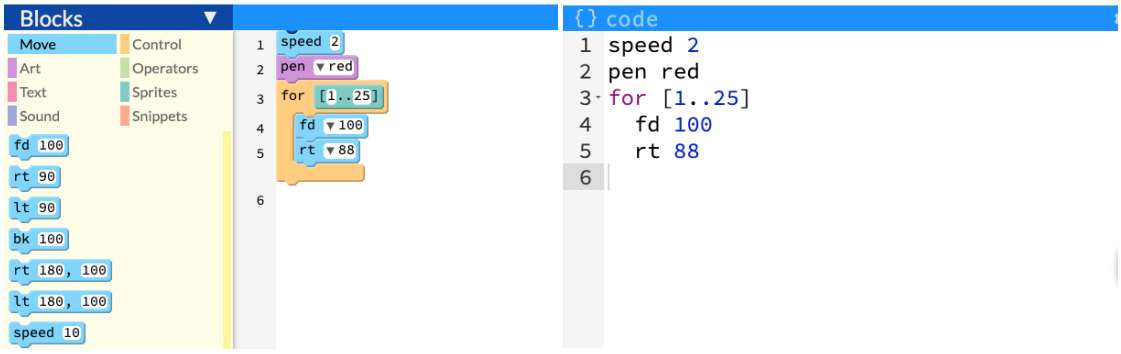
observar cómo el estudiante está arrastrando el operador booleano **o** en la condición de la estructura condicional. El bloque del operador lógico tiene forma de hexágono como la condición del condicional. Si no se pueden unir los bloques, el entorno no permite que esto ocurra, evitando así los errores de sintaxis, permitiendo a los estudiantes enfocarse en los conceptos, argumentan los defensores de este tipo de entornos [40].

La falta actual de docentes formados en programación y el poco impacto de los cursos cortos de formación, pueden ser la causa de que los entornos basados en bloques como Scratch hayan ganado una gran popularidad, argumentan los detractores de este tipo de lenguajes. Los docentes pueden aprender y enseñar los conceptos fundamentales de programación de forma más simple utilizando este tipo de entornos, sin tener que enfrentarse ellos tampoco a los errores de sintaxis y a memorizar todas las instrucciones.

Pero, los entornos basados en bloques tienen sus desventajas. Y la más cuestionada se relaciona justamente con una de las ventajas con las que cuentan los entornos basados en bloques. La falta de errores de sintaxis de los entornos basados en bloques, puede pensarse como posponer un problema que tarde o temprano surgirá al querer cambiar a un lenguaje de texto. Termina siendo un tema central el pensar la transición de entornos basados en bloques a texto para pensar la enseñanza de la programación. Weintrop y Wilensky en [127] identifican dos enfoques principales para intentar lograr esta transición. Un enfoque es el pedagógico, dejando la responsabilidad de la transición a cargo del docente. En el Capítulo 5 recuperamos y describimos estudios donde se implementa el enfoque pedagógico para la transición de bloques a texto. La falta de docentes formados complican la factibilidad de poder implementar este enfoque de una forma escalable. El otro enfoque propuesto es diseñar entornos de enseñanza de programación que propicien y faciliten la transición de un lenguaje basado en bloques a un lenguaje de texto. A partir de este enfoque, la clasificación de los entornos de enseñanza de programación deja de ser binaria en cuanto a si son basados en bloques o texto. Estos nuevos entornos diseñados para facilitar la transición de bloques a texto, se pueden clasificar en 2 categorías [127] que se describen a continuación.

Entornos de doble modalidad

Los entornos de doble modalidad, permiten al estudiante elegir si desea programar en bloques o en texto y traduce automáticamente los cambios de un lenguaje a otro. Es decir, permiten realizar una traducción bidireccional, de bloque a texto y de texto a bloque. Esto es posible porque los lenguajes de texto y de bloques tienen el mismo nivel de expresividad, a diferencia de los entornos que traducen de bloques a texto que describimos a continuación. Icaro [5], Pencil Code [6], Tild Grace [55] entre otros son ejemplos de entornos de doble modalidad. En la Figura 3.3 podemos observar la interfaz del entorno Pencil Code, además de una secuencia de bloques y una secuencia de texto equivalentes. El lenguaje basado en bloques y el basado en texto tienen el mismo nivel de expresividad, por lo que, el estudiante puede modificar con cualquiera de los dos lenguajes.



(a) (b)

Figura 3.3: Interfaz del entorno Pencil Code. En la Figura 3.3a observamos un programa creado utilizando el lenguaje basados en bloques. El programa dibuja un cuadrado de color rojo. El código se traduce al otro lenguaje al presionar el botón como se observa en la Figura 3.3b.

Entornos que traducen de bloque a texto

EduBlocks, App Inventor Java Bridge [123], BlocklyDuino [66], Blockly [42] son ejemplos de entornos que traducen el código en bloques a texto. Blockly [42] permite traducir los bloques a diferentes lenguajes de texto como JavaScript, Python, C++ entre otros. Pero no es posible de traducir texto a bloques, sólo traduce en una dirección. Esto se debe a que, en este tipo de entornos, los lenguajes basados en bloques tienen un menor nivel de expresividad que los lenguajes basados en texto. En la mayoría de estos entornos, los lenguajes basados en bloques permiten a los estudiantes crear programas simples, con menor expresividad que los lenguajes de texto. Por lo que al momento de querer agregar funcionalidades más complejas o tener mayor control sobre el programa, es necesario utilizar los lenguajes basados en texto. En la Figura 3.4 podemos ver la interfaz del entorno de programación de kits de robótica BlocklyDuino. El estudiante únicamente puede programar utilizando bloques los cuales se pueden traducir a C++ al elegir la pestaña Arduino. El programa que se observa en la figura permite que un motor gire a la izquierda hasta que la temperatura sensada por el sensor sea mayor que 35. El lenguaje de bloques no permite especificar a que velocidad girará el motor mientras que C++ sí.

La necesidad práctica de traducir entre lenguajes de texto y bloques pone en evidencia una característica que diferencia a distintos lenguajes de enseñanza de programación que no es evidente a simple vista: el nivel de expresividad del lenguaje. Los lenguajes basados en texto permiten al estudiante tipear libremente y por lo tanto dan una ilusión de mayor expresividad,

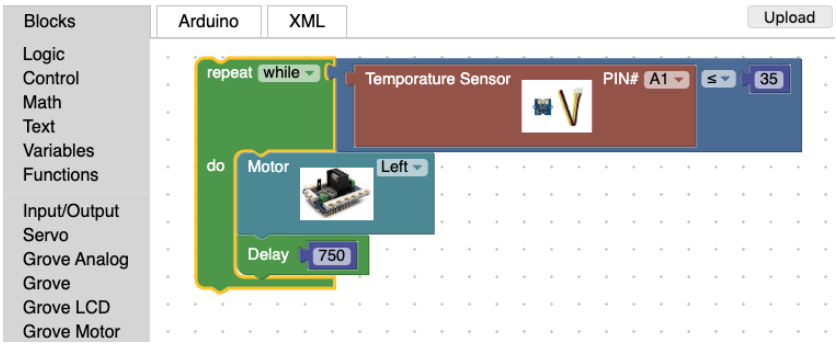


Figura 3.4: Interfaz del entorno BlocklyDuino [66]. En la primer figura observamos un programa creado utilizando el lenguaje basados en bloques. El programa que se observa en la figura permite que un motor gire a la izquierda hasta que la temperatura sensada por el sensor sea mayor que 35. Al hacer clic en la pestaña Arduino, se puede ver la traducción automática a C++. No es posible modificar el código en C++ y volver a traducir al lenguaje de bloques por el mayor nivel de expresividad que tiene C++.

sin embargo, el léxico permitido en dichos lenguajes está restringido por la gramática formal del lenguaje de programación. Generalmente los lenguajes basados en bloques son menos expresivos que los basados en texto pero esto no es necesariamente así. Es una coincidencia causada porque los lenguajes de bloques son generalmente diseñados para aprendices más jóvenes y los lenguajes con menor nivel de expresividad también. En el Capítulo 4 veremos dos lenguajes, ambos basados en bloques que tienen distinto niveles de expresividad: Icony y Blockly. Icony fue diseñado durante esta tesis como un lenguaje de programación con un nivel de expresividad restringido con el objetivo de enfocarnos en la enseñanza de fundamentos de la programación en escuela primaria y nivel inicial.

3.3. Interactividad formativa del entorno

Al igual que los lenguajes naturales, los lenguajes de programación no se aprenden de modo espontáneo sino que se adquieren y evolucionan a través de la interacción. El aprendizaje del lenguaje se ve afectado por distintas condiciones de interactividad. En la sección anterior hablamos de distintos niveles de expresividad de los lenguajes. En esta sección nos enfocaremos en otro tipo de clasificación. Considerando el tipo de interactividad formativa que permite el entorno de aprendizaje, los entornos se pueden clasificar en abiertos o cerrados. Una diferencia entre los entornos abiertos y cerrados es que en los entornos cerrados los ejercicios de programación están predefinidos dentro del entorno mientras que en los entornos abiertos es el estudiante o el docente quien diseña (o saca de un libro) el ejercicio a programar.

Los entornos abiertos están diseñados para que los usuarios puedan diseñar y programar distintos tipos de aplicaciones, como animaciones, videojuegos, apps, chatbots, robots, etc. Scratch [103, 70], Alice [30], Greenfoot [54, 63], Pilas-Engine[2] son ejemplos de entornos abiertos utilizados para la creación de videojuegos o animaciones. En [41] realizamos una revisión de diferentes entornos abiertos existentes para la creación de videojuegos utilizados en Game Jams. MIT App Inventor [130, 131] es un entorno abierto para la creación de aplicaciones para teléfonos móviles. Chatbot [14, 15] es un entorno que se puede utilizar en modo abierto o cerrado diseñado para enseñar a programar creando chatbots que pueden contestar a conversaciones en

[2]https://pilas-engine.com.ar/

las redes sociales. Lego Mindstorms [121, 106] y UNCDuino [72, 13] son ejemplos de entornos abiertos para enseñar a programar con robots educativos. A pesar de que entornos abiertos como Scratch o Alice hayan sido diseñados específicamente para que principiantes aprendan a programar, la corrección manual de los proyectos puede ser problemática si los docentes no cuentan con una sólida formación previa o si tienen aulas con numerosos estudiantes. Mumuki.io [12], Code.org [59], Pilas Bloques [108] y Codecademy [22] son ejemplos de otros entornos cerrados con evaluación automática.

Otro ejemplo de entorno cerrado con evaluación automática es la versión Coartada de Chatbot [14, 15]. Los estudiantes pueden participar de una historia de misterio usando Chatbot. Para ello eligen un sospechoso de un crimen para defender y programan un chatbot para que responda un conjunto de preguntas de forma correcta. El feedback formativo de los programas creados por los estudiantes se basa en preguntas específicas que el programa de los estudiantes debe responder en base a respuestas esperadas. En Coartada, denominan a este conjunto de preguntas como interrogatorio del detective. Una vez que el cuestionario del detective es cargado en Chatbot, el entorno simula una conversación entre el chatbot creado por el estudiante para representar al sospechoso y el chatbot del detective como observamos en la Figura 3.5. En el panel izquierdo de la figura observamos los cuestionarios del detective que han sido cargados y el porcentaje de correctitud (cobertura en la figura) en base al chatbot programado por el estudiante. En este caso hay dos interrogatorios. Uno de ellos cubierto en 50 % y el otro en 0 %.

En el panel derecho, observamos los diferentes tipos de feedback formativo generados por Coartada. Primero, observamos el conjunto de preguntas que forman parte del interrogatorio a responder. Cada pregunta y respuesta esperada forman parte del interrogatorio. En el caso de la conversación de la figura, la respuesta que se muestra se basa en el chatbot programado por el estudiante. En la figura observamos que cada pregunta con su respectiva respuesta puede ser de color negro, violeta o rojo. En el caso en que el chatbot programado por el estudiante sea capaz de reconocer la pregunta y además la respuesta que de a la misma sea la esperada, es etiquetada como correcta utilizando el color negro. En el caso en que el chatbot creado por el estudiante, sea capaz de identificar la pregunta pero la respuesta no es la esperada, la pregunta se etiqueta como incorrecta utilizando el color violeta. Finalmente, si la pregunta no es capturada por el chatbot programado por el estudiante la pregunta se etiqueta de color rojo, y el chatbot del estudiante se confiesa como culpable de forma automática. Los estudiantes deben crear un chatbot que no se declare como culpable pero también evite etiquetar preguntas como incorrectas. En base a las respuestas del chatbot, se define un puntaje general. En el caso del ejemplo el puntaje global es del 25 %. Este puntaje se obtiene de haber obtenido un 50 % del primer cuestionario y un 0 % del segundo. Para poder conseguir un puntaje global del 100 % todas las preguntas de todos los cuestionarios tienen que ser respondidas de forma correcta.

El estudiante puede utilizar el botón “Pista” para poder inspeccionar cualquier pregunta etiquetada de color rojo o violeta. Al crear el cuestionario del detective, el docente a cargo puede relacionar pistas a diferentes preguntas. Las pistas se relacionan con conceptos de programación necesarios para resolver el problema que se presenta. Las pistas se brindan al estudiante un link a vídeos o material de lectura donde se introduce el concepto.

Los tipos de feedback formativo que utiliza Chatbot pueden clasificarse según Shute [112] de la siguiente manera. Primero, el porcentaje de correctitud del chatbot creado por los estudiantes con respecto a los cuestionarios predefinidos puede ser considerado como feedback de *verificación* al evaluar la salida del programa. Segundo, Chatbot clasifica los errores como “No responde la pregunta” o “Responde la pregunta de forma incorrecta”. Este tipo de feedback no puede ser relacionado de forma directa con los tipos de feedback recopilados en [112]. Es bastante común

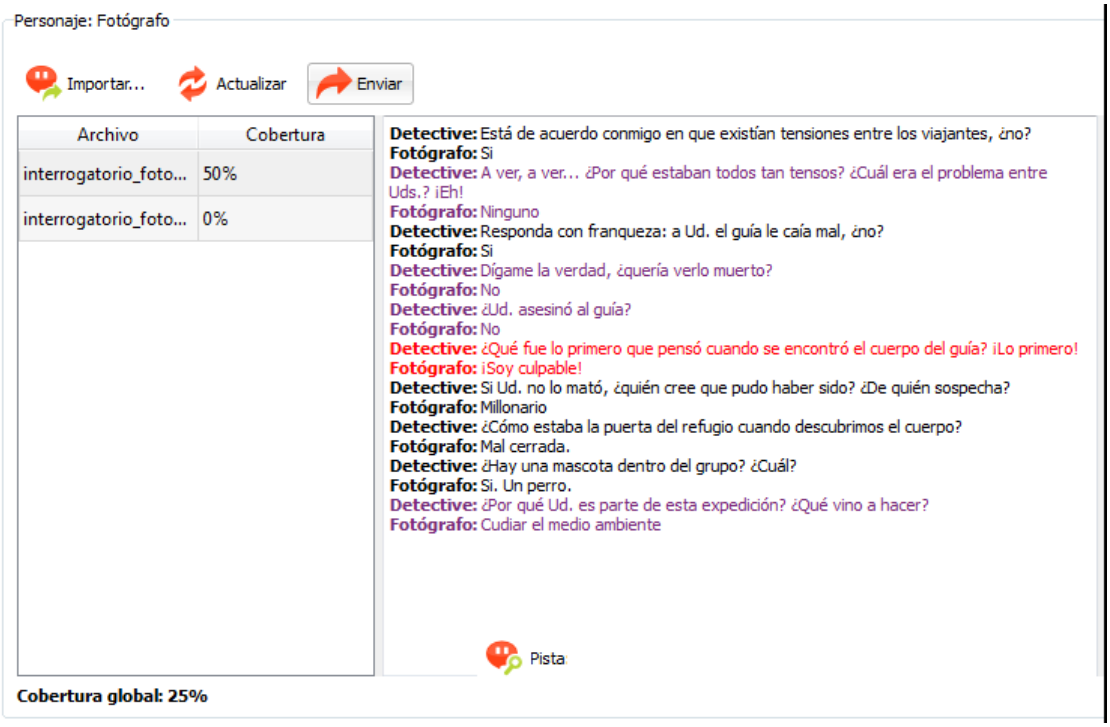


Figura 3.5: Interfaz del entorno Coartada. En el panel de la izquierda observamos los interrogatorios utilizados y el nivel de correctitud del chatbot programado por el estudiante. En el panel de la derecha observamos las preguntas correspondientes al interrogatorio y las respuestas generadas por el chatbot del estudiante.

que los docentes de programación clasifiquen el tipo de errores que cometen los estudiantes al aprender a programar. Como tercer tipo de feedback, Chatbot le brinda al estudiante un link directo a la parte del programa que debe revisar. Este tipo de feedback según Shute corresponde a *error flagging*. Finalmente, Chatbot puede brindar pistas sobre los conceptos necesarios para poder resolver un ejercicio (por ejemplo, compartir un video o material de lectura al estudiante). Este tipo de feedback puede ayudar al estudiante a identificar como resolver un error dentro de su programa. En base al trabajo de Shute podemos clasificar este tipo feedback como *hint*. En resumen, Chatbot cuenta con verificación, error flagging y hint relacionados al ejercicio a resolver. La versión de Coartada de Chatbot cuenta también con estos tipos de feedback, pero al ser un entorno cerrado permite ocuparse de ciertas dificultades de usar el entorno abierto Chatbot. En [14, 2] los autores muestran que Coratada, la versión cerrado de Chatbot, disminuye la deserción en un curso online abierto y masivo.

Todos estos elementos hacen de Coartada un entorno cerrado, el cual permite definir ejercicios específicos y generar feedback formativo automático para cada uno de ellos. De esta manera, los docentes encargados de crear los cuestionarios pueden decidir en qué conceptos de programación hacer foco.

Los entornos cerrados, a diferencia de los entornos abiertos, cuentan con un conjunto de ejercicios creados con una secuenciación predefinida en los contenidos y conceptos. Los entornos cerrados suelen incluir herramientas de evaluación automática, las cuales permiten generar una interacción formativa con los estudiantes. Esto es posible ya que los programas corresponden a un problema predefinido. Este tipo de entornos pueden ser efectivos para cursos con muchos estudiantes o para docentes que no tenga formación sólida en programación. Ciertos entornos

cerrados son capaces de almacenar las soluciones enviadas, la evaluación de las mismas y el feedback formativo por cada estudiante. De esta manera permite realizar un seguimiento personalizado para cada estudiante. En el Capítulo 5 describiremos el impacto del uso de un entorno cerrado para la enseñanza de programación en dos contextos educativos diferentes.

3.4. Evaluación automática y fluidez

La *fluidez* es el término utilizado por la lingüística para caracterizar y medir las habilidades de una persona con respecto a un lenguaje natural. En esta tesis usamos el concepto de fluidez para caracterizar las habilidades de un estudiante al escribir en un lenguaje de programación. En particular, en esta tesis analizamos la posibilidad de evaluar la fluidez de forma automática, para asistir a los docentes durante la enseñanza de programación.

Las evaluación automática forma parte de las herramientas que son utilizadas al momento de enseñar a programar. Cuando los cursos son multitudinarios o los docentes no tienen formación previa en programación pueden ser herramientas útiles para ayudar al docente a enseñar los contenidos de manera más efectiva [95]. Pero no pensadas para que se encarguen de la evaluación total del estudiante, sino para que faciliten y sinteticen información para el docente al momento de realizar la evaluación de cada uno de sus estudiantes.

Como describimos en la sección anterior los entornos cerrados tienen como característica contar con herramientas de evaluación automática. Pilas Bloques [108], Mumuki.io [12], Code.org [59], Codecademy [22], cuentan con estas herramientas. En el caso de los entornos cerrados, al ser ejercicios concretos, con un problema definido, es posible diseñar distintas herramientas de evaluación automática. A continuación describimos algunas de ellas.

Code.org [128] organiza a nivel mundial la campaña “La hora de código”, donde millones de personas han escrito sus primeros programas. Code.org cuenta con un entorno cerrado donde los estudiantes aprenden programando al personaje principal para que pueda resolver laberintos. Una vez que los estudiantes envían una solución, Code.org evalúa de forma automática la misma y brinda un feedback formativo específico al estudiante en forma de ayuda. En la Figura 3.6 podemos observar la interfaz de Code.org y el feedback formativo que genera en el caso de enviar una solución incorrecta como la que se observa en la figura. En el feedback formativo podemos ver que directamente le avisa al estudiante que bloque debe utilizar para resolver el ejercicio.

Codecademy [22] es un entorno cerrado que contiene tutoriales con ejercicios para aprender a programar en lenguajes como Python, Ruby, PHP, Javascript y otros. Los ejercicios se completan siguiendo un conjunto instrucciones que brinda el entorno. El usuario escribe el código requerido en el editor de texto. Codecademy es muy utilizado, pero, como se describe en [111], una de sus principales limitaciones es que no puede identificar cuándo los estudiantes se han desmotivado o están teniendo dificultades con un ejercicio específico. Después de una cantidad fija de soluciones incorrectas, Codecademy le brinda la solución correcta al estudiante.

Para los entornos abiertos existen herramientas que pueden realizar evaluaciones automáticas de las habilidades en programadores principiantes que son independientes del ejercicio particular. Las evaluaciones automáticas más frecuentes son la detección código duplicado y código muerto (es decir, que nunca se ejecuta). Dr. Scratch [83] es una herramienta de evaluación automática para programas creados en Scratch. A cada proyecto lo evalúa con un puntaje entre 0 y 21. Cada proyecto es evaluado por Dr. Scratch en base a siete habilidades sobre el lenguaje. Cada habilidad es evaluada con un puntaje entre 0 y 3. En la Tabla 3.1 podemos ver las dimensiones y los niveles de competencia presentada por Troiano et al. en [118].

Habilidades desarrolladas	Nivel de Competencia		
	Básico (1)	**En desarrollo (2)**	**Competente (3)**
Abstracción	Programar más de un script en dos objetos o más	Definir bloques propios	Utilizar clones
Representación de la información	Modificar propiedades de un objeto	Operaciones con variables	Operaciones con listas
Control de flujo del programa	Secuencia de bloques	Uso de bloques de repetición: repetir o por siempre	Uso de bloque: repetir hasta
Lógica	Uso de bloque condicional: si	Uso de bloque condicional: si, sino	Uso de operadores lógicos
Paralelismo	Dos scripts basados en el evento al presionar la bandera verde	Dos scripts basados en el evento al presionar tecla o dos scripts en el mismo objeto basados en el evento al hacer click el este objeto	Dos scripts que utilicen el bloque al recibir mensaje, crear clones, dos scripts que utilicen el bloque Cuando volumen del sonido/cronómetro >, dos scripts que utilicen el bloque cuando el fondo cambie a
Sincronización	Uso del bloque esperar	Uso de mensajes: bloque para enviar mensaje, recibir mensaje, detener todos los scripts, detener los scripts del objeto, detener el script	Uso de los bloques: esperar hasta que, enviar mensaje y esperar, cuando el fondo cambie a
Interacción usuario	Uso del bloque al presionar la bandera verde	Uso de los bloques: al presionar tecla, preguntar al usuario y esperar, al hacer click en este objeto	Uso de bloque: Cuando volumen del sonido/cronómetro >, uso de bloques de video, uso de bloques de sonidos

Tabla 3.1: Métricas definidas por Dr. Scracth para determinar el nivel de competencia que tiene el estudiante para cada una de las habilidades evaluadas.

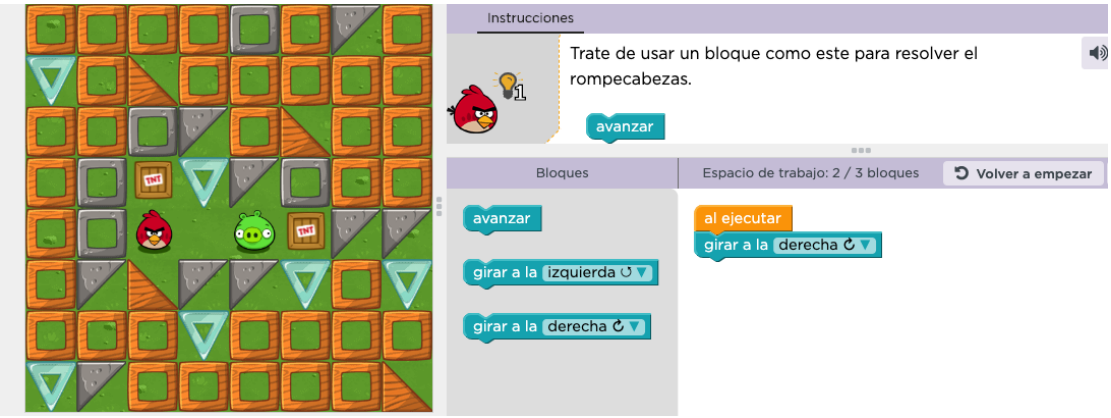


Figura 3.6: Interfaz del entorno cerrado de enseñanza de programación Code.org.

Al analizar las habilidades, y sobre todo, los criterios para poder definir el nivel de competencia de cada habilidad, podemos encontrar muchas debilidades en la herramienta. La evaluación automática que brinda Dr. Scratch, se enfoca en contar ciertas estructuras y bloques en los programas creados por los estudiantes. Por lo tanto un estudiante puede crear un programa que no resuelva los problemas planteados por el docente, sin sentido y que no sea funcional obteniendo el puntaje más alto en Dr. Scratch. En la Figura 3.7 podemos ver un ejemplo de un programa que recibe la puntuación más alta por Dr. Scratch en la habilidad de pensamiento lógico, a pesar de contar un error que hace que el condicional nunca sea ejecutado.

Figura 3.7: Programa creado en Scratch con error lógico en la condición.

La condición siempre será falsa por lo que el cuerpo no se ejecuta nunca. Entonces, un programa que no se ejecuta correctamente por un error lógico, recibe el máximo puntaje solo por tener en el código las estructuras que se definen en la Tabla 3.1. Si tenemos en cuenta la falta de docentes formados para poder enseñar a programar, este tipo de herramientas podrían generar problemas, sobre todo si son utilizadas con fines evaluativos. Los estudiantes pueden identificar las estructuras deben utilizar para que el puntaje siempre sea el más alto. No es una herramienta para que el docente dependa. En el caso de ser un docente con formación y experiencia, puede utilizarla para obtener información sintetizada de los programas creados por sus estudiantes.

A pesar de que la evaluación automática en este tipo de entornos sea una herramienta interesante para los docentes y estudiantes, Wang et al. en [124] plantean un tema preocupante. La habilidad de los docentes de poder observar el proceso de sus estudiantes parece perderse en la evaluación automática. Poder entender el proceso y progreso de un estudiante es invalorable. Si un docente puede tener conocimiento de cuando un estudiante está teniendo problemas para

resolver un ejercicio específico podría rápidamente ayudar al estudiante a superar la situación. Codecademy al brindar la solución luego de una serie de intentos y Code.org con su feedback tan dirigido también pueden ser engañosos al momento de guiarnos únicamente con la evaluación automática. Dr. Scracth evalúa con el máximo puntaje en la categoría de pensamiento lógico, en un programa con un error lógico. Entonces, la evaluación automática por si misma no genera una evaluación real del nivel de fluidez de los estudiantes al aprender a programar.

3.5. Conclusiones del capítulo

En este capítulo describimos tres aspectos que influyen al momento de adquirir un lenguaje al enseñar a programar. En la Sección 3.2 introducimos los tipos expresividad de los lenguajes de programación, centrándonos en los lenguajes basados en bloques y los lenguajes basados en texto. En el Capítulo 4 analizamos el impacto de la expresividad del lenguaje al momento de enseñar a programar. Describimos el entorno de enseñanza de programación con múltiples expresividades UNCDuino [72, 13] y el lenguaje icónico Icony, los cuales desarrollamos para enseñar a programar en nivel inicial y primario.

En la Sección 3.3 describimos el impacto de la interactividad formativa del entorno de enseñanza de programación. En particular, clasificamos los entornos como abiertos y cerrados, ejemplificando para cada caso. En la Capítulo 5 describimos el impacto de la interactividad del entorno cerrado Mumuki.io [12] al enseñar a programar con lenguajes basados en texto a estudiantes principiantes en nivel primario y universitario.

Finalmente, en la Sección 3.4 describimos las diferentes herramientas de evaluación automática para diferentes tipos de entornos. En el Capítulo 6 compararemos distintos métodos automáticos que intentan detectar la fluidez de un estudiante en base al análisis de sus programas y su trayectoria. De esta manera, poder predecir si el estudiante será capaz de resolver un ejercicio dado sin ayuda.

Capítulo 4

Expresividad del lenguaje de programación

En este capítulo describimos experiencias de aprendizaje de fundamentos de la programación realizadas en nivel inicial y primario usando el entorno web de enseñanza UNCDuino. UNCDuino fue desarrollado para esta tesis e incluye lenguajes de programación de distinto nivel de expresividad. UNCDuino permite programar en cuatro lenguajes de programación. En particular, incluye al lenguaje Icony, que diseñamos para enseñar a programar en nivel inicial. Icony es un lenguaje icónico, utilizable por niños que aún no saben leer y escribir. El capítulo está organizado en cuatro secciones. En la Sección 4.1 describimos el entorno UNCDuino. En la Sección 4.2 presentamos el lenguaje icónico Icony, que diseñamos específicamente para nivel inicial. En la Sección 4.3 presentamos y analizamos experiencias de enseñanza de programación en nivel inicial y primario utilizando Icony. Finalmente en la Sección 4.4 discutimos las conclusiones de este capítulo y lo relacionamos con capítulos de esta tesis.

Los resultados principales descriptos en este capítulo fueron publicados en [72], [75] y [13]. En particular, [72] describe el uso del lenguaje Icony en nivel inicial y primario. Es una de las primeras experiencias sobre cómo estudiantes de nivel inicial y primario adquieren nociones fundamentales de programación. El trabajo [72] ha sido citado por reconocidos investigadores del área y por el estándar de enseñanza de las Ciencias de la Computación de la Association for Computing Machinery [29].

4.1. Un entorno de enseñanza con múltiples expresividades

UNCDuino es una entorno educativo open source, pensado y diseñado para llevar la enseñanza de la programación a los niveles inicial y primario de la escuela. Sin embargo, también fue utilizado en nivel secundario y en cursos de formación docente [74]. El mismo fue diseñado para poder introducir los conceptos fundamentales de programación, a partir de la programación de kits de robótica basados en Arduino [35]. La elección de la robótica para llevar la enseñanza de programación no fue casual. En el Capítulo 3 describimos los tres enfoques principales más utilizados para enseñar a programar en el nivel primario. Uno de ellos es la robótica.

Existen varios kits de robótica utilizados para enseñar a programar que se describen en el Capítulo 3. Decidimos basar UNCDuino en Arduino porque UNCDuino es un proyecto de software libre y Arduino es un proyecto de hardware libre, basado en una placa con un microcontrolador, diseñada para facilitar el uso de la electrónica en proyectos multidisciplinares.

UNCDuino es una extensión del entorno BlocklyDuino [66]. BlocklyDuino es un entorno que utiliza el lenguaje basados en bloques Blockly [42] para programar placas Arduino. El entorno permite crear un programa utilizando los bloques, los cuales son traducidos a C++. Otros entornos web de enseñanza de programación usan Blockly. Ejemplos de ellos son el MIT App Inventor [98, 50] y Code.org [59, 93] como describimos en el capítulo anterior.

UNCDuino permite la programación de placas Arduino en 4 lenguajes de programación. Cada uno de estos lenguajes tiene diferentes niveles de expresividad y de control del hardware. Estos lenguajes listados en nivel creciente de expresividad son: Icony, Blockly, Python y C++. C++ y Python son lenguajes industriales ampliamente difundidos. Blockly [42] es un lenguaje basado en bloques desarrollado por Google con fines educativos. Icony es un lenguaje icónico, el cual diseñamos para poder introducir a la programación a quienes no sepan leer ni escribir, como ocurre en el nivel inicial escolar. El entorno permite traducir automáticamente de lenguajes de menor expresividad a lenguajes de mayor expresividad. UNCDuino, teniendo en cuenta la clasificación de los lenguajes realizado en el capítulo anterior se clasifica como un entorno abierto y como un entorno que traduce de bloque a texto.

La interfaz de UNCDuino es simple. Podemos observar en la Figura 4.1 cómo la misma está organizada. En la parte superior se encuentran cuatro pestañas: dos en color rojo y dos en color verde. Las pestañas son rojas para los lenguajes basados en bloques y verdes para los lenguajes basados en texto.

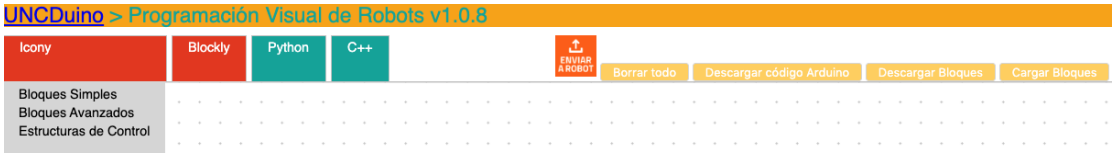


Figura 4.1: Captura de pantalla de la interfaz del entorno para robótica educativa UNCDuino.

Cada una de estas pestañas hace referencia al lenguaje de programación que se podrá utilizar para programar el kit de robótica. Al seleccionar alguna de las pestañas rojas, se nos habilitarán los bloques y el editor para poder programar con lenguajes basados en bloques. En el caso de seleccionar alguna de las pestañas verdes, se habilitará el editor para poder programar con lenguajes basados en texto. El orden de aparición de los lenguajes en la interfaz es de menor a mayor nivel de expresividad. En primer lugar se encuentra el lenguaje icónico Icony. Este lenguaje está restringido en el control que el programador puede tener sobre el hardware como explicaremos en la Sección 4.2.

En el caso en que el usuario de la plataforma haya comenzando con Icony, pero precise de más control sobre el hardware puede cambiar a cualquiera de los otros lenguajes. Al hacer click en la pestaña deseada, el código creado en Icony se traduce automáticamente a Blockly, Python o C++. Siempre es posible traducir de un lenguaje menor nivel de expresividad a uno de mayor nivel de expresividad. Pero esto no es bidireccional como dijimos en el Capítulo 3. Es decir, que el entorno no traduce automáticamente de un lenguaje más expresivo a uno menos expresivo. Esto no es posible ya que por ejemplo Icony no es tan expresivo como Blockly. En la Sección 4.2 presentamos ejemplos de estas diferencias de expresividad de los lenguajes usados.

4.2. Un lenguaje de programación para nivel inicial

En esta sección describimos el lenguaje de programación icónico Icony, su sintaxis y su semántica. La sección está organizada en dos subsecciones. En la Sección 4.2.1 describimos la

sintaxis y la semántica intuitiva de Icony y los diferentes bloques creados para poder programar el kit de robótica a utilizar en las experiencias en nivel inicial y primario, que describiremos en la Sección 4.3. En la Sección 4.2.2 describimos el método formal de dar semántica al lenguaje Icony que usamos en UNCDuino, traduciéndolo a los otros tres lenguajes.

4.2.1. Descripción de la sintaxis

Icony es un lenguaje de programación icónico, pensado y diseñado para poder enseñar a programar sin tener que saber escribir o leer. El foco de Icony está puesto en la enseñanza de programación en nivel inicial y para los primeros grados de nivel primario. Icony nace como necesidad de un grupo de docentes de nivel inicial que formaron parte de una de las capacitaciones descriptas en el Capítulo 2. Como parte del trabajo final del curso de formación, los docentes debían de poder implementar una experiencia de enseñanza de programación en sus aulas con alguno de los entornos utilizados durante la experiencia o alguno propuesto por ellos. Los entornos utilizados por el momento habían sido Alice, Chatbot y UNCDuino para la programación de kits de robótica basados en Arduino. Hasta ese momento, UNCDuino no contaba con el lenguaje Icony. Era posible programar el kit de robótica utilizando Blockly, Python o C++. Las docentes de nivel inicial notaron que sería imposible poder implementar con sus estudiantes ya que la mayoría no sabía escribir o leer, pero que sería una experiencia motivadora para los estudiantes poder tener una primer experiencia programando robots. En conjunto con las docentes de nivel inicial diseñamos una secuencia de actividades a implementar y las características de un nuevo lenguaje. Así surgió Icony.

El lenguaje Icony está compuesto por 8 bloques. Cada bloque está vinculado con algún concepto puntual de programación o con los actuadores y sensores del kit de robótica. En la Figura 4.2, se pueden observar 4 bloques verdes, los cuales son las estructuras de control que forman parte de Icony.

Figura 4.2: Estructuras de control pertenecientes al lenguaje Icony.

En la Figura 4.2 podemos ver el bloque que utilizamos para presentar el concepto de condicional. Está compuesto por dos imágenes, las cuales representan dos condiciones. La primera figura representa la condición “el robot tiene un objeto en frente”, mientras que la segunda figura representa “el robot no tiene ningún objeto en frente”. Estas condiciones usan un sensor ultrasó-

nico que detecta si hay algún objeto que bloquee el camino del robot. De esta manera niños que no sepan leer y escribir pueden crear un programa que les permita utilizar la estructura condicional, sin tener que definir condiciones propias. El bloque con la figura de botón rojo también es utilizado para introducir el condicional. Como se puede deducir de la figura, la condición que representa el mismo es "presionó el botón de encendido".

Los bloques que tienen como figura la flecha circular representan la estructura de control ciclo dentro de Icony. Icony cuenta con un ciclo finito y otro infinito. El bloque de mas a la derecha representa un ciclo infinito. Este bloque es útil para modelar el comportamiento continuo del robot hasta que se presione el botón para apagar el robot. El bloque de su izquierda mientras que el bloque es el ciclo finito, en el cual se puede modificar la cantidad de iteraciones a ejecutar.

La Figura 4.3 incluye el resto de los bloques pertenecientes a Icony. Contamos con 3 bloques que representan funciones simples y uno de ellos a una función parametrizable. Cada bloque representado con una flecha, permite programar el movimiento del robot. La flecha que apunta hacia abajo permite a los usuarios programar al robot para que se mueva hacia adelante poniendo en movimiento los motores. El programador no puede elegir la velocidad de movimiento ni la distancia a recorrer. La expresividad del lenguaje no lo permite. La semántica de este bloque está fija. La misma se define arbitrariamente para que el robot se mueva hacia adelante 20 centímetros, y luego espera 1 segundo antes de poder ejecutar la instrucción siguiente. Los bloques con las flechas curvas son para programar al robot para que gire a la izquierda o derecha 90 grados. La velocidad, tiempo y ángulo de giro no pueden ser modificados por el programador. Una vez finalizado el giro, hay una espera de 1 segundo para ejecutar la próxima instrucción. El bloque que se observa en la Figura 4.3, con el símbolo musical, permite programar al kit de robótica para que ejecute una melodía monofónica. El programador puede elegir entre varias opciones de melodías monofónicas para que el kit de robótica reproduzca en el momento deseado parametrizando así este bloque.

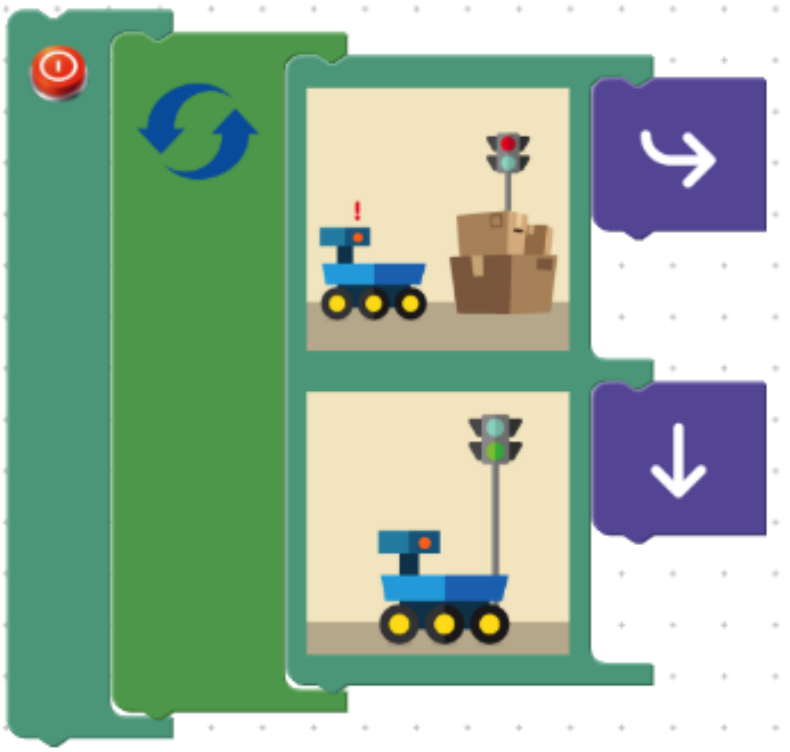
Figura 4.3: Bloques pertenecientes al lenguaje Icony.

4.2.2. Traducción a otros lenguajes

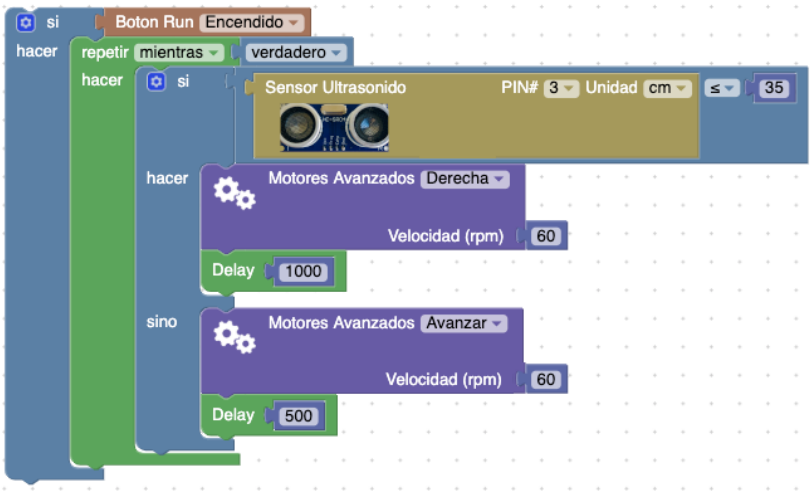
Cada bloque presentado en la sección anterior es traducido a cualquiera de los lenguajes pertenecientes a UNCDuino: Blockly, Python o C++. Lo que no es posible, es traducir de un lenguaje de mayor expresividad a uno de menor expresividad. Es decir, que no es posible traducir de Blockly, Python o C++ a Icony.

En la Figura 4.4, podemos observar un programa creado en Icony y su traducción a Blockly. El programa creado permite al robot esquivar objetos a medida que va avanzando. Los dos programas que se observan en la figura realizan la misma tarea. Es mucho más simple escribir un programa en Icony, pero tiene sus limitaciones. La velocidad de los motores y el tiempo durante el cuál se ejecutarán estos movimientos están definidos como se puede ver en la traducción a Blockly. Por ejemplo, los motores girarán a la derecha a 60 revoluciones por minuto (rpm) durante 1000 milisegundos como indica el bloque Delay . Icony tampoco permite definir qué distancia se considera como cerca o lejos. En el programa de Blockly se puede ver que cerca se

define como menor a 35cm. Las limitaciones en cuanto a la expresividad del lenguaje icónico están pensadas como estrategias didácticas. En el caso de querer tener más control sobre el kit de robótica podrá elegir Blockly, Python o C++.



(a)



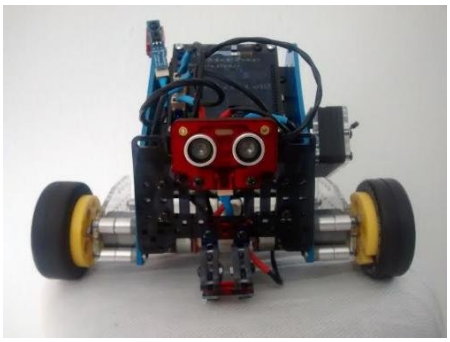
(b)

Figura 4.4: En la Figura 4.4a podemos ver un programa creado en Icony para que el robot esquive objetos. En la Figura 4.4b observamos la traducción del programa en Icony a Blockly.

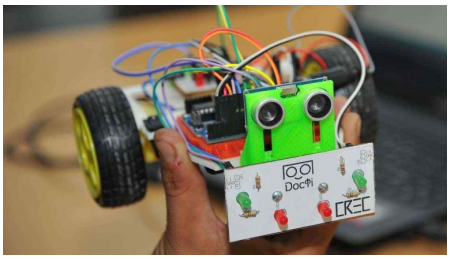
.

4.2.3. Hardware

El lenguaje Icony está pensado para kits de robótica basados en Arduino con los cuales se puedan construir robots que al menos contengan dos motores, dos ruedas, un sensor ultrasónico y un sintetizador de sonido. Para nuestros experimentos usamos el kit de robótica N6 desarrollado por la empresa Argentina RobotGroup[1]. Con el kit de robótica N6 es posible crear un robot de 3 ruedas, como el se puede observar en la Figura 4.5a.



(a) Kit de Robótica N6.



(b) Kit de Robótica Docti.

El kit está compuesto por una placa Arduino, dos motores(con dos ruedas), un sensor ultrasónico y dos sensores infrarrojos. El kit también está compuesto un puerto USB 2.0 que permite cargar al robot el programa escrito en UNCDuino y 6 conectores para sensores analógicos de 10 bits que sirven por ejemplo para agregar otro tipo de sensores al kit. También incluye un conjunto de leds. Finalmente, el kit también incluye un sintetizador de sonido. Utilizamos este hardware debido a las siguientes ventajas. Para empezar, su precio era 4 veces más barato que kits de robótica Lego Mindstorms. Al estar basado en Arduino, cualquier otro kit de robótica que contenga alguna de los componentes enumerados anteriormente puede ser programado utilizando UNCDuino y Icony. Este es el caso del kit de robótica cordobés Docti, el cual podemos ver en la Figura 4.5b.

4.3. Experiencias de enseñanza de programación en nivel inicial y primario

Con el objetivo de investigar cómo los niños aprenden conceptos fundamentales de programación utilizando la programación de robots en la escuela y poder contribuír al desarrollo de un diseño curricular para la enseñanza de la programación, diseñamos un estudio exploratorio para comparar cómo los niños de nivel inicial (de 3 y 5 años) y de primaria (de 8 a 11 años) incorporan conceptos fundamentales de programación. Para ello implementamos clases en un entorno educativo real enfocándonos en conceptos como ciclo, variable, condicional, secuencia y parámetro.

En esta sección describimos y analizamos los resultados del estudio exploratorio realizado en nivel inicial y primario de enseñanza de programación utilizando el entorno UNCDuino. En 4.3.1 presentamos la institución participante de la experiencia, describimos características generales de los grupos de estudiantes y definimos la metodología y actividades implementadas. En 4.3.2 describimos las herramientas de recolección de datos utilizadas durante el desarrollo de la experiencia en los diferentes niveles educativos. En 4.3.3 analizamos los datos obtenidos

[1] RobotGroup `https://robotgroup.com.ar` es una empresa Argentina que desarrolla kits de robótica educativa.

teniendo en cuenta la edad y género de los estudiantes. Finalmente en 4.3.4 realizamos un análisis detallado para cada uno de los conceptos fundamentales desarrollados durante las experiencias.

4.3.1. Diseño del estudio

El estudio se desarrolló en horario escolar en la institución pública de gestión privada Escuela Nueva Juan Mantovani (ENJM), en la ciudad de Córdoba. No hay estudiantes bajo la línea de la pobreza, siendo la mayoría perteneciente a la clase media y con padres profesionales. La currícula de la escuela se basa en el aprendizaje basado en problemas y exploración a partir de proyectos. Las experiencias se desarrollaron en nivel inicial y nivel primario. Tanto en nivel inicial como en nivel primario los estudiantes finalizaron sus experiencias utilizando el entorno UNCDuino con el lenguaje Icony. Pero, las experiencias previas fueron diferentes en cada nivel. A continuación describimos las mismas.

En nivel inicial, la experiencia fue completamente pensada para la programación de los robots. En conjunto con las profesoras de nivel inicial diseñamos una secuencia didáctica compuesta por tres etapas. Durante la primer etapa, los estudiantes actuaban como robots quienes seguían las instrucciones que sus compañeros decían. El equipo docente diseñó un juego el cual consta de una grilla de 5 x 5 cuadrados en el piso. En los cuadrados se pueden encontrar diferentes obstáculos para evitar u objetos que conseguir. Utilizando un lenguaje de flechas físicas similares a las de Icony, los estudiantes elegían la secuencia de instrucciones para que su compañero (quien hacía de robot) llegue hasta el objetivo. Inicialmente se definieron solo tres flechas: avanzar, girar a la derecha y girar a la izquierda. En la segunda etapa los estudiantes transformaron el juego de la primer etapa, en un juego de mesa. Cada estudiante creó su propio robot de juguete, y usando una tablero de mesa, replicaron la experiencia. Es el primer paso para poder separar los movimiento del robot del cuerpo de los estudiantes. Finalmente los estudiantes programaron el robot utilizando el lenguaje Icony. Los estudiantes programaron el robot aplicando los conceptos de secuencia, ciclo y uso de parámetros para salir de un laberinto en formato de grilla (similar a la usada en la primer fase). Luego programaron el robot para que evite objetos moviéndose libremente sobre el mundo real utilizando el concepto de condicional.

En el nivel primario la experiencia constó de 3 etapas. En la primera de ellas los estudiantes comenzaron a utilizar los conceptos de secuencia, ciclo y condicional utilizando las actividades propuestas por el entorno Code.org descripto en el Capítulo 3. En la segunda etapa, trabajaron los conceptos fundamentales creando animaciones con el entorno Alice [30]. Finalmente transfirieron los conceptos aprendidos programando los robots con Icony. En particular se introdujeron los conceptos de secuencia, condicional, ciclo y uso de parámetros.

4.3.2. Recolección de datos

Para poder medir el impacto de las experiencias y la incorporación conceptual de los conceptos de programación por parte de los estudiantes, diseñamos un examen de múltiple opción que debían resolver una vez finalizada la experiencia. El examen está compuesto por siete preguntas. Las preguntas se clasifican en simples o compuestas. Una pregunta se considera *simple* cuando es necesario utilizar un solo concepto para resolver la actividad, mientras que se clasifica como *compuesta*, si es necesario combinar conceptos para resolver el desafío planteado. Los conceptos evaluados fueron los trabajados durante la experiencia: secuencia, parámetro, ciclo y condicional. En la Figura 4.6, se observa uno de los ejercicios propuestos en el examen. En este ejercicio, los estudiantes debían elegir cuál de los tres programas en Icony le permitía al robot llegar hasta el objetivo (la cara feliz). Notar que cuando las profesoras de nivel inicial desarrollaron la secuencia

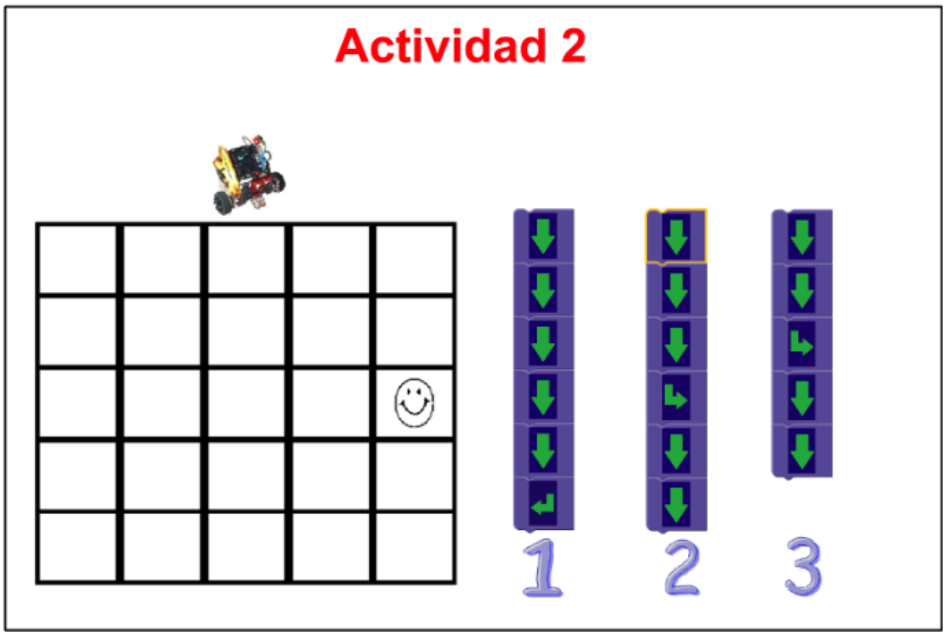


Figura 4.6: Actividad perteneciente al examen múltiple opción que completaron los estudiantes.

didáctica decidieron que el robot debía posicionarse fuera de la grilla. Para poder ingresar a la misma es necesario utilizar una flecha para avanzar. Por eso en la Figura 4.6 el robot se encuentra fuera de la cuadrícula. Teniendo en cuenta estas consideraciones, la respuesta correcta es la número 2. El robot se mueve tres veces hacia adelante, para posicionarse en la fila donde se encuentra el objetivo. Luego gira 90 grados sobre su eje para cambiar su dirección. Finalmente se mueve dos veces hacia adelante para poder posicionarse sobre el objetivo.

Todos los estudiantes programaron el kit de robótica utilizando las conceptos de secuencia, condicional, ciclo y uso de parámetros en diferentes tareas. La tarea podía de necesitar un solo concepto para ser resuelta, una combinación de conceptos o podía requerir resolver dos grillas distintas con el mismo programa.

Además de los exámenes de múltiple opción se realizaron observaciones durante las clases de programación de robots. Las observaciones fueron realizados por investigadoras de ciencias de la educación. A partir de las mismas pudimos documentar comentarios o situaciones que sucedieron durante la experiencia que se reportan en las Secciones 4.3.3 y 4.3.4.

Al final de toda la intervención se realizó un grupo focal con tres salas diferentes de nivel inicial. Solo una de las salas no había recibido la intervención y fue usada a modo de grupo de comparación, puesto que los estudiantes compartían las mismas características demográficas que las otras salas. El grupo focal indagó sobre los tipos de trabajos que diferentes profesiones realizan, incluyendo profesiones que se relacionan con disciplinas obligatorias en el currículum de educación infantil tales como música, teatro y matemáticas. También preguntamos por oficios de otras disciplinas no obligatorias en el currículum como antropología y computación. Buscábamos entender si el acceso a una disciplina les ofrecía una visión sobre el oficio en esa área. Durante el grupo focal mostramos una aspiradora robot que se desplaza con ruedas y aspira mientras se traslada. Pedimos a los niños que describieran qué era el objeto (la aspiradora) y preguntamos además por qué pensaban que se movía y doblaba al encontrarse con un obstáculo.

En total participaron 190 estudiantes de la experiencia. 55 estudiantes de nivel inicial y 135 de nivel primario. En nivel inicial trabajamos con estudiantes de sala de 3 (3 - 4 años) y sala de 5 (5 - 6 años). En nivel primario participaron de la experiencia estudiantes de 4, 5 y 6to grado.

Nivel Educativo	Nivel Inicial	Nivel Inicial	Nivel Primario	Nivel Primario
#Participantes	25	30	70	65
#Evaluados	17	26	42	43
Edad	3-4	5-6	8-9	10-11

Tabla 4.1

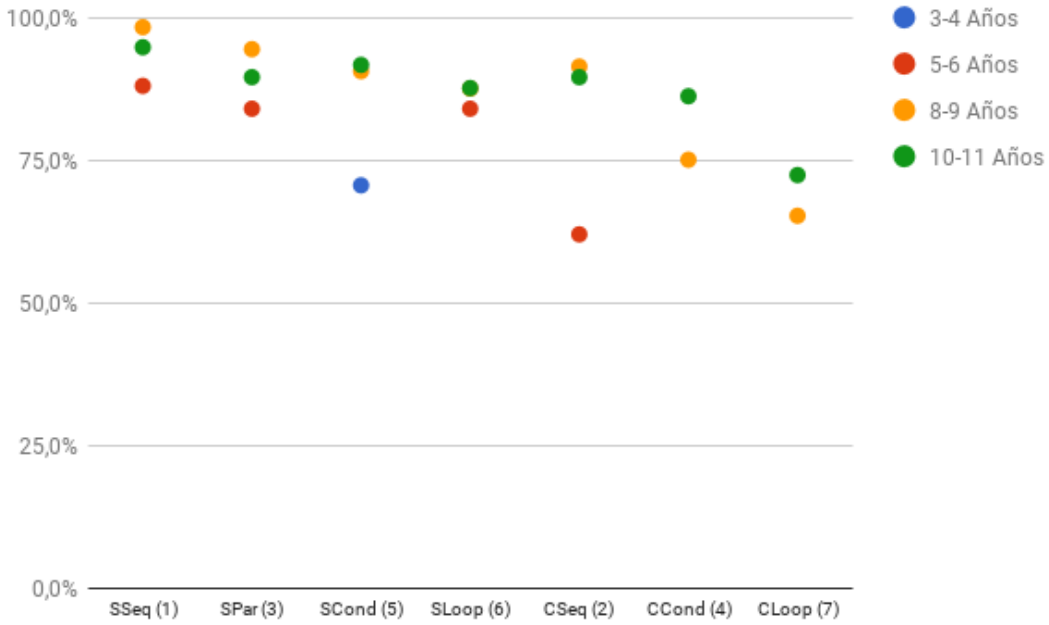


Figura 4.7: Desempeño por edad y concepto de programación en base al examen múltiple opción. Los ejercicios que comienzan con la letra S refieren a los ejercicios simples, mientras que los ejercicios que comienzan con la letra C corresponden a los ejercicios compuestos

No todos los estudiantes tomaron el examen. En la Tabla 4.1, se describen datos generales de la población participante de la experiencia. Además se enumeran la cantidad de estudiantes que participaron por nivel educativo y por edad.

A partir de los datos obtenidos de los exámenes múltiple opción, realizamos un análisis el cual describimos en la próxima sección.

4.3.3. Resultados

En esta sección presentamos los resultados obtenidos a partir de los datos de los exámenes de múltiple opción y las observaciones realizadas en el aula. Realizamos comparaciones por edad y género para poder medir la incorporación conceptual. No todos los grupos pudieron ser evaluados en todos los conceptos y no todos los estudiantes que participaron de la experiencia pudieron tomar el examen.

En la Figura 4.7 se puede observar el porcentaje de estudiantes evaluados que resolvió de forma correcta cada ejercicio del examen por edad. Cada una de las preguntas del examen estaba relacionada con los conceptos trabajados durante toda la experiencia por los estudiantes. Como definimos anteriormente, hay dos categorías de ejercicios: los simples, donde se evalúa un concepto puntual, y los compuestos, donde se evalúa la combinación de diferentes conceptos.

En la Figura 4.7 se observa que únicamente los estudiantes del nivel primario resolvieron todos los ejercicios propuestos en el examen. Los estudiantes de nivel inicial pertenecientes a sala de 4 (entre 3-4 años) solo resolvieron la actividad relacionada con el uso del bloque condicional. Los estudiantes de nivel inicial pertenecientes a la sala de 5 (entre 5-6 años) resolvieron los ejercicios basados en secuencia y ciclo.

Los estudiantes de mayor edad pudieron resolver de mejor manera los ejercicios donde se combinaban conceptos, los cuál es esperable teniendo en cuenta las diferentes etapas madurativas en las cuales los estudiantes se encuentran.

4.3.4. Discusión

Las observaciones se realizaron en nivel inicial y primario. Para cada uno de los conceptos trabajados (secuencia, condicionales, ciclos y uso de parámetros) se registraron situaciones particulares. A continuación realizamos un análisis por cada concepto introducido durante la experiencia en base a los resultados de los preguntas del examen de múltiple opción y las observaciones realizadas. Finalmente realizamos un análisis del impacto de la experiencia teniendo en cuenta el género.

Secuencia

De 26 estudiantes evaluados en sala de 5, 23 resolvieron correctamente el ejercicio en el cual había que aplicar el concepto de secuencia. Las observaciones de las clases indicaron que la mayoría de los niños en edad preescolar podrían proporcionar una serie de instrucciones secuenciales con flechas de avanzar o girar de forma intercalada. Tanto en el juego de mesa como cuando programaron un robot, pudieron identificar claramente la cantidad y flechas necesarias para que el robot alcance el objetivo. El pensamiento divergente también estuvo presente cuando los niños sugirieron diferentes secuencias de instrucciones para que el robot pueda llegar al objetivo.

Sin embargo, los niños de 3 a 4 años no pudieron en general, relacionar cada instrucción icónica con su significado. Por ejemplo, para lograr que un compañero que tomaba el rol de robot gire y luego avance, en vez de utilizar la secuencia de instrucciones girar y avanzar, posicionaron la instrucción de avanzar orientada hacia la dirección deseada, suponiendo que el robot giraría y luego avanzaría. Este último grupo de estudiantes también tuvo dificultades al darse cuenta de que los robots ejecutan el código secuencial que programaban en las computadoras. Vincular el mundo virtual de la pantalla con los movimientos espaciales concretos de los robots no fue un problema para los niños de sala de 5.

Como se puede observar en la Figura 4.7, en el nivel primario, la mayoría de los estudiantes pudo resolver los ejercicios del examen relacionados con el concepto de secuencia. A partir de las observaciones realizadas, se observó que resuelven con facilidad y en poco tiempo los desafíos secuenciales. Solo dos grupos no resolvieron los desafíos planteados en clase. Pero no porque no hubiesen incorporado los conceptos, sino porque querían resolver otro tipo de problemas o desafíos, ya que las actividades propuestas les parecieron demasiado sencillas. Programar robots generó en los estudiantes un motivante para diseñar sus propias actividades . Por ejemplo en cuarto grado (niños entre 8 y 9 años) un observador de la clase registró la siguiente situación.

Ya que los desafíos les parecieron bastante sencillos, los estudiantes decidieron crear sus propios desafíos. Los estudiantes más inquietos fueron los más activos para poder resolver uno de los desafíos más complejos propuestos, lograr que el robot pudiese pasar bajo un puente armado con las sillas de la escuela. Para ello, previo a programar el robot, recorrieron el circuito para

poder decidir cada instrucción a elegir. Solo 2 de 30 estudiantes no pudieron resolver el desafío. Un grupo de estudiantes quería cambiar el movimiento de robot para lograr que se moviera en círculos mientras que otro grupo quería que el robot esquive objetos. Los estudiantes comenzaron a preguntar sobre los sensores y cómo lograr que el robot esquive objetos. Uno de los estudiantes consultó cómo era posible cambiar la distancia del sensor de proximidad. Querían saber también cómo poder controlar la velocidad de los motores. De manera espontánea comenzaron a observar el código generado por los bloques icónicos en C++ y comenzaron a modificar el código de C++ sin conocer su sintaxis.

Condicional

Para introducir en nivel inicial el concepto de condicional se presentó el desafío de lograr que el robot esquive objetos. Como primer parte de la actividad, preguntamos a los estudiantes: "¿Qué queremos que suceda si en el camino el robot se encuentra con una caja?" Su respuesta fue: "El robot tiene que girar". Luego presentamos a los estudiantes el bloque de Icony que permite programar al robot para detectar si hay un objeto cerca. Los estudiantes trabajaron en grupos de 5 o 6 estudiantes para poder programar el robot utilizando el concepto de condicional. Los estudiantes de 5-6 años automáticamente detectaron el funcionamiento del bloque, identificando cada una de las condiciones y qué significaban las mismas. Al momento de identificar el funcionamiento de las condiciones generadas por las imágenes no tuvieron problemas. Programaron el cuerpo del condicional para que el robot se moviera hacia adelante si no había un objeto en frente, y girara en el caso de encontrarse con un objeto en su camino. Mientras que los estudiantes de 5-6 años podían leer el código icónico y predecir las acciones del robot al momento de ser ejecutadas, los niños de 3-4 años no pudieron darse cuenta de que el robot ejecutaría las instrucciones que programaron en la computadora.

12 de los 17 estudiantes de 3-4 años resolvieron de forma correcta la pregunta del examen relacionada al uso de condicional. En la Figura 4.8, se puede observar el ejercicio. En el mismo se presenta como estaba programado el robot y la situación en la cual se encuentra. Los estudiantes tenían que elegir la decisión que tomaría el robot teniendo en cuenta el programa que tenía cargado. 5 estudiantes eligieron la opción 1, en la cual el robot choca contra la caja. Esto nos llamó la atención ya que al momento de programar el robot para que evitará los objetos pudieron hacerlo sin problemas. Al preguntarles personalmente, nos dijeron: "elegimos esa foto porque queremos que el robot se choque". A partir de las respuestas inferimos que las respuestas erróneas no están relacionadas a la no comprensión del concepto condicional, sino a la decisión de no cumplir con el enunciado.

Al igual que los estudiantes de nivel inicial, los estudiantes de primaria incorporaron el concepto de condicional a través del desafío de programar el robot para que evite objetos. Las notas tomadas de las observaciones nos permitieron detectar que un grupo de 10-11 años llamaron espontáneamente "decisión" al concepto de condicional. Los estudiantes pudieron trasladar el concepto de condicional que habían adquirido utilizando Alice [30] o Code.org [59] sin problemas a UNCDuino. Mientras que los estudiantes entre 8 y 11 años tuvieron un desempeño similar en la resolución del ejercicio del examen múltiple opción de condicional simple, los estudiantes de entre 10 y 11 años tuvieron un mejor desempeño al momento de resolver el ejercicio donde se combinaban varios conceptos con el condicional. Creemos que los desafíos que proponen combinación conceptual demandan un mayor entendimiento y nivel de abstracción.

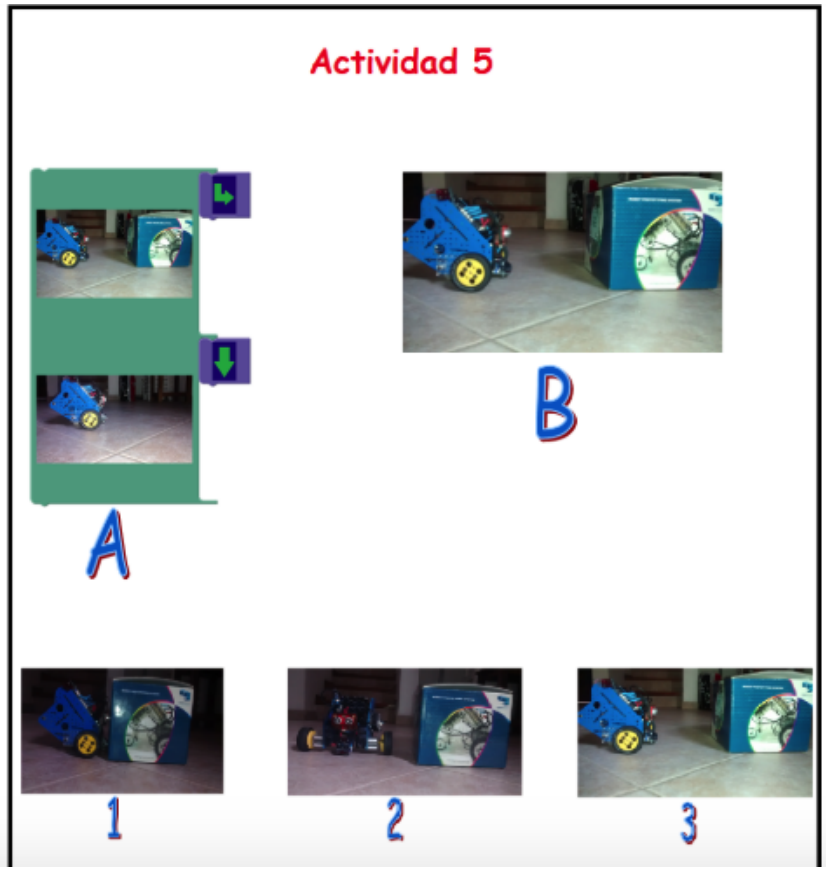


Figura 4.8: Actividad perteneciente al examen múltiple opción para evaluar la incorporación del concepto de condicional.

Ciclos

En nivel inicial el concepto de ciclo sólo fue presentado a los estudiantes de 5-6 años, ya que es necesario una mínima comprensión de conteo y multiplicación para poder desarrollarlo. Simplemente les dijimos a los niños que existía con un nuevo bloque que les permitía repetir la cantidad de veces el bloque que ellos quisieran. Para poder confirmar si habían comprendido la función del bloque les preguntamos: "Si tenemos este programa 4.9, ¿cuántas veces se moverá hacia adelante el robot?"

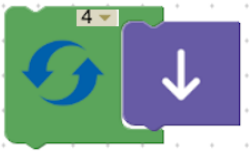


Figura 4.9: Ejemplo de un programa utilizando la estructura de control repetir con Icony.

Los estudiantes respondieron: "el robot avanzará 4 veces. No nos vamos a cansar por tener que escribir muchas flechas (recuperado de los registros de las observaciones de clase)". Al programar utilizando el nuevo bloque, no tuvieron inconvenientes. Las observaciones que realizamos a medida que programaban con el bloque de ciclo, nos hacían inferir que había cierta comprensión del concepto de ciclo como algo que les permitía repetir acciones ahorrando la cantidad de

instrucciones necesarias. En un momento de la clase planteamos el desafío de lograr que el robot se moviese 10 veces hacia adelante. Los estudiantes pudieron resolverlo sin problema. Pero un un grupo en particular nos llamó la atención. Habían programado la solución del desafío de la siguiente manera: repetir 5 veces, y dentro del bloque 2 flechas de avanzar. Al preguntar al grupo de sala de 5 como funcionaba el programa nos respondieron: "El robot va a avanzar 10 veces, repitiendo 5 veces moverse 2 veces hacia adelante".

En nivel primario las notas tomadas a partir observaciones nos mostraron que los estudiantes automáticamente preguntaron al docente si había un bloque repetir cuando tuvieron que crear una secuencia simple de muchas instrucciones. Correctamente asumieron que podrían programar el robot utilizando la estructura ciclo, a partir de la transferencia realizada de haber utilizado los entornos Alice o code.org. Los resultados del examen de múltiple opción nos muestran el mismo patrón de desempeño que observamos tanto para secuencia como para condicional. Al momento de resolver los ejercicios simples, todos los estudiantes, sin importar edad o género, aplican correctamente el concepto de ciclo. Sin embargo, cuando el desafío propone combinación de conceptos, los estudiantes mayores tienen un mejor desempeño.

Parámetros

Para poder introducir el concepto de parámetro en nivel inicial, pedimos a los estudiantes que una vez que el robot llegase al objetivo debía cantar una canción. Utilizando el bloque musical, los estudiantes podían programar el robot para que reproduzca una canción. El bloque cuenta con 3 opciones: dos canciones conocidas o hacer silencio. Todos los estudiantes pudieron agregar el bloque de sonido con la canción apropiada, una vez finalizada la secuencia de instrucciones que permitía llegar al objetivo. Contar con un bloque que reproduzca sonidos los motivo. El 85 % de los estudiantes resolvió correctamente el ejercicio de parámetro simple del examen múltiple opción.

Los estudiantes de nivel primario pudieron transferir con facilidad la noción de parámetro trabajada previamente en Alice. También fue una motivación para los estudiantes de primaria contar con un bloque que les permita programar el robot para que reproduzca canciones. Cada vez que programaban el robot, agregaban el bloque de reproducción de canciones. Pero a diferencia de los niños en edad pre escolar, debido a que UNCDuino permite ver el código tanto en bloque como en C++ o Python , los niños cambiaron espontáneamente a la interfaz de C++ o Python, sin tener experiencia previa en ellos. Al programar el robot, los estudiantes querían cambiar la velocidad del robot y evitar la pausa entre cada movimiento. Decidieron leer el código, sin ningún conocimiento de C++ y buscar cómo modificar el mismo para cambiar la velocidad. Primero, observaron que dentro de la función *loop*, se encontraba *avanzar()*. Tras este descubrimiento, su primera intervención fue agregar un argumento numérico para modificar la velocidad. Al compilar el código, observaron un error, por lo que comenzaron a leer el código completo y descubrieron la definición del procedimiento *avanzar()*, sin saber qué es un procedimiento. Observaron que dentro del cuerpo del procedimiento avanzar había muchas instrucciones como *motor0.setSpeed(60)*, *motor1.setSpeed(60)* y *delay(1000)*. Se dieron cuenta de que si cambiaban el argumento del método *setSpeed()* de ambos motores, podían cambiar la velocidad del robot. Pero el robot todavía se movía 1 segundo, y luego se frenaba. Así que volvieron al código en C++ y notaron que la instrucción *delay(1000)* era responsable del tiempo de movimiento del motor. Borraron la instrucción delay y las instrucciones que ponían la velocidad de los motores a 0, haciendo que el robot avance rápidamente y durante más tiempo. Continuaron jugando con los argumentos de las instrucciones, haciendo que el robot retrocediera o se moviera en círculos, acciones que no serían posibles con la versión icónica. La expresividad restringida de Icony motivó la exploración

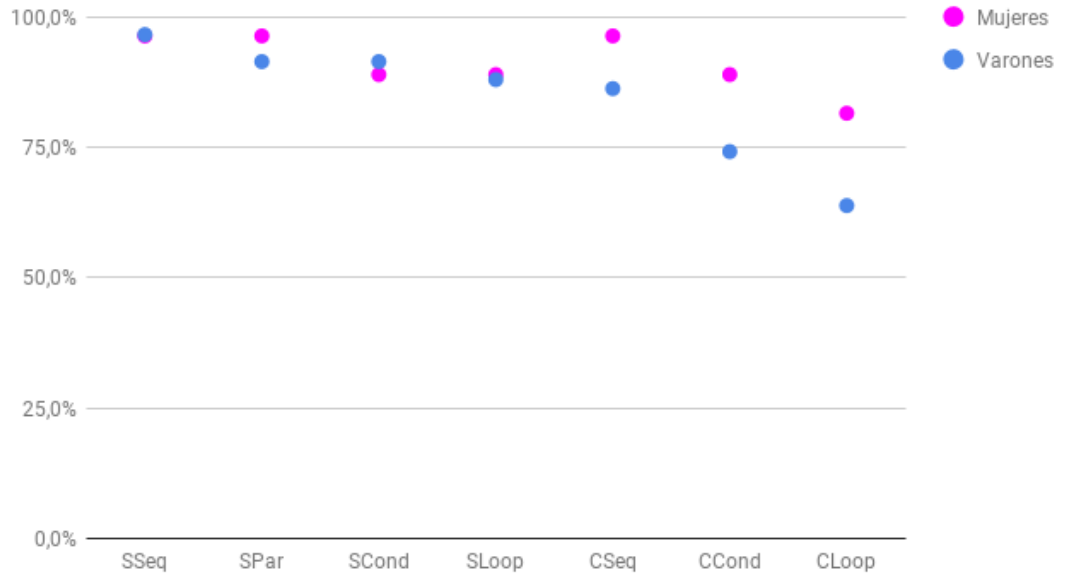


Figura 4.10: Desempeño por género y concepto de programación en base al examen múltiple opción. Reportamos únicamente los resultados de los estudiantes de escuela primaria.

de lenguajes de mayor expresividad.

4.3.5. Brecha de género en programación

En base a los resultados del examen, realizamos un análisis por género, basado en la correctitud de los ejercicios del examen. Como los estudiantes de nivel inicial resolvieron pocos ejercicios, realizamos los análisis de género solo en el nivel primario. En la Figura 4.10 se observa el desempeño en el examen por género de los estudiantes en nivel primario.

Este primer análisis nos permite observar que no hay diferencia en el desempeño entre mujeres y varones para los ejercicios simples. Sin embargo, las mujeres tienen mejor desempeño en los ejercicios compuestos, donde debían de combinar conceptos para resolver los mismos. Al comenzar la experiencia de programación de robots, las mujeres no estaban muy entusiasmadas con la actividad, ya que relacionaban al robot con un auto de juguete. Sin embargo, cuando comenzaron a programar el robot y lograr que el mismo hiciese lo que ellas querían, comenzaron a interesarse. El entusiasmo de los varones se mantuvo constante. Trabajo previo muestra que las mujeres tienen calificaciones más bajas que los varones en materias relacionadas a computación en el primer año universitario. Redmond et al. en [102] argumentan que las calificaciones bajas están relacionadas con el hecho de que las chicas han sido menos expuestas a las computadoras, y por lo tanto su confianza para trabajar con las mismas es menor. Basados en los resultados obtenidos, sospechamos que la brecha de género empieza más tarde.

4.4. Conclusiones del capítulo

A partir de las experiencias de enseñanza de programación utilizando UNCDuino descriptas en este capítulo, descubrimos que los estudiantes pudieron incorporar conceptos fundamentales de programación como secuencia, ciclos, condicional y uso de parámetros. Como era de esperar los estudiantes de mayor edad pudieron combinar diferentes conceptos para poder programar

comportamientos compuestos en los robots. Por más evidentes que puedan parecer estos resultados, antes de este estudio no existía evidencia empírica sobre los conceptos de programación que los diferentes grupos de edad pueden aprender. Es necesario contar con esta evidencia para poder definir un diseño curricular en los distintos niveles educativos. También identificamos que los estudiantes de 3 a 4 años no podían relacionar el programa escrito en la computadora con las acciones que el robot ejecutaría. Las niñas tuvieron un leve mejor desempeño en los ejercicios del examen múltiple opción donde había que combinar conceptos, sugiriendo que la brecha de género se inicia más tarde. Los estudiantes mostraron un alto compromiso con los robots y, nuestra plataforma multilenguaje motivó la exploración de lenguajes con distinta expresividad y permitió a los niños cambiar fácilmente de lenguajes de menor a mayor expresividad, centrándose en conceptos en lugar de en la sintaxis. Hay varias implicaciones a partir de estos hallazgos. Primero, a partir de los resultados obtenidos sugerimos que es posible enseñar programación en nivel inicial y primario. Las estrategias de enseñanza basadas en la indagación resultaron valiosas porque permitieron a los alumnos construir de forma intuitiva los conceptos, aplicando los mismos en diferentes espacios, y finalmente aplicarlas en la programación de robots. Esta es una contribución al campo de la educación en programación ya que todavía estamos debatiendo sobre el enfoque pedagógico para introducir la programación en las escuelas. Nuestra plataforma multilenguaje diseñada para estudiantes de nivel inicial y primaria motivó la exploración y el aprendizaje de los niños sobre conceptos de programación. En general, los entornos existentes para enseñar a programar están diseñados para grupos de edades específicas, por lo que, los niños al crecer, no quieren utilizar las mismas . Nuestra plataforma alienta a los estudiantes a crecer con ella. Reconocemos que se necesita más investigación con entornos multilenguajes, pero esta podría ser una dirección motivar a los niños.

Capítulo 5

Interactividad formativa y su efecto en la adquisición del lenguaje

En este capítulo describimos experiencias de aprendizaje de programación realizadas en nivel universitario y nivel primario usando el entorno *cerrado* de enseñanza de programación Mumuki. Como argumentamos en el Capítulo 3, los entornos cerrados, a diferencia de los abiertos, pueden permitir aumentar la cantidad de estudiantes cuando la cantidad de docentes es limitada. Implementamos y analizamos experiencias de enseñanza de programación utilizando lenguajes de programación basados en texto en el entorno cerrado Mumuki.

Mumuki es un entorno web para la enseñanza de programación open source que brinda a los usuarios feedback formativo (del inglés *formative feedback*), tiene soporte para 17 lenguajes de programación y cuenta con guías de ejercicios diseñados para cada lenguaje. Este capítulo está organizado en tres secciones. Mumuki, inicialmente fue diseñado como un entorno de enseñanza de programación para nivel universitario. En la Sección 5.1 presentamos la versión de Mumuki que diseñamos para nivel primario y analizamos cómo estudiantes de primaria que vienen trabajando sobre entornos abiertos basados en bloques adquieren un lenguaje de programación textual usando Mumuki. Luego, en la Sección 5.2 presentamos las experiencias y los resultados de evaluar el impacto del entorno Mumuki en la adquisición de un lenguaje de programación en estudiantes de nivel universitario. Finalmente en la Sección 5.3 reflexionamos sobre las conclusiones generadas a partir de las experiencias.

Los resultados principales descriptos en este capítulo fueron publicados en [12] y [46].

5.1. Experiencias en la escuela primaria

Como describimos en el Capítulo 3, la mayor parte de la investigación realizada sobre la enseñanza de la programación se base en educación secundaria y superior, los estudios en niveles primario no son tan comunes [29]. Los estudios que existen para el nivel primario se limitan a describir experiencias de enseñanza con un solo lenguaje de programación (por ejemplo, [72]). Además, la mayoría de las experiencias documentadas sobre enseñanza de programación en escuela primaria se basan en entornos basados en lenguajes de bloques como el que presentamos en el capítulo anterior.

Se supone que los lenguajes basados en bloques permiten a los estudiantes enfocarse en los conceptos, dejando de lado la complejidad sintáctica presente cuando aprenden a programar usando un lenguaje basado en texto. Pero, ¿qué tan bien pueden los estudiantes de primaria

pasar de lenguajes basados en bloques a lenguajes basados en texto? En otras palabras, nuestra pregunta de investigación Q1: *Los niños con experiencia en programación con lenguajes de bloques ¿adquieren habilidades de programación que facilitan el aprendizaje de un lenguaje basado en texto?*

Para abordar la pregunta, planteamos dos hipótesis no direccionales. *Ha* está relacionada con errores sintácticos mientras que *Hb* está relacionada con errores semánticos. Decimos que un estudiante comete un error sintáctico cuando su programa no cumple con la sintaxis del lenguaje de programación. Decimos que un estudiante comete un error semántico cuando el comportamiento del programa no es el esperado.

Ha (null): No existe una diferencia significativa en la cantidad de errores sintácticos generados por los estudiantes con y sin experiencia previa en programación basada en bloques.

Hb (null): No existe una diferencia significativa en la cantidad de errores semánticos generados por los estudiantes con y sin experiencia previa en programación basada en bloques.

Con el objetivo de evaluar la pregunta de investigación y poder contribuir a un diseño curricular para el nivel primario, diseñamos un estudio observacional en la escuela primaria, donde niños de 10 a 11 años, con y sin experiencia previa basada en lenguajes de bloques, aprenden un lenguaje de programación basado en texto. Llevamos a cabo clases de programación en un entorno educativo real centrándonos en los conceptos de ciclo, condicional, secuencias y parámetros al igual que en el capítulo anterior.

Las contribuciones de esta sección son 3. Primero, presentamos Mumuki, un entorno online que provee interactividad formativa y registra cada interacción con el estudiante así como el feedback formativo (del inglés formative feedback) generado automáticamente por el sistema[1]. Segundo, analizamos comparativamente cómo estudiantes de 10-11 años con y sin experiencia previa con lenguajes de bloques cometen errores sintácticos y semánticos al adquirir un nuevo lenguaje de programación basado en texto. Finalmente comparamos a ambos grupos sobre errores conceptuales que sufren los aprendices de la programación.

La sección está compuesta por cuatro partes. En 5.1.1, recuperamos trabajo previo de enseñanza de programación en escuela primaria. Luego, en 5.1.2 describimos el entorno web online Mumuki, el lenguaje de programación basado en texto y cómo utilizamos los mismos para implementar las experiencias. En 5.1.3 introducimos las dos instituciones participantes de las experiencias, características generales de los grupos de estudiantes, herramientas de recolección de datos y describimos las dos intervenciones implementadas. Finalmente, en 5.1.4 presentamos y analizamos los resultados obtenidos a partir de las intervenciones.

5.1.1. Trabajo previo

Como describimos en el Capítulo 3 existen varios entornos y herramientas que permiten a niños crear, ejecutar y debuggear sus propios programas [28, 61, 40]. Trabajo previo [39, 61] muestra que incluso los niños que no saben leer y escribir se motivan al programar utilizando entornos tangibles, con bloques de madera o virtuales como vimos en el capítulo anterior. Algunas implicaciones de aprender a programar en edades tempranas han sido analizadas. Según el trabajo de Clements [28] niños que usan programas de computadora tienen la oportunidad de analizar una situación y reflexionar sobre las propiedades de los objetos que tienen que manipular. Clements

[1]Para esta tesis Mumuki fue adaptado a nivel primario, siendo inicialmente diseñado para nivel universitario. Actualmente la empresa desarrolladora de Mumuki tomó esta adaptación y está disponibñe en `https://mumuki.io/primaria/`.

concluyó que la programación permite vincular el conocimiento intuitivo de los niños (sobre movimientos y dibujos) con ideas formales más explícitas utilizando un lenguaje.

Otros estudios [39] sugieren que la programación desarrolla habilidades de pensamiento de orden superior, como competencias metacognitivas. Ejemplos de habilidades metacognitivas son proponer diferentes métodos para resolver un mismo problema y tener la capacidad de debuggear y hallar errores en el propio razonamiento. Los niños están reflexionando sobre su pensamiento cuando buscan errores en programas creados por ellos. Este es un fuerte indicador de metacognición.

La mayoría de los estudios en la enseñanza de programación en escuela primaria hacen foco en herramientas para enseñar pero no en qué están aprendiendo los estudiantes ni en los errores que cometen. Algunos de estos estudios encuentran que las principales dificultades están en las habilidades motoras inmaduras y la sintaxis de los lenguajes de programación [84, 40]. Por lo tanto se han desarrollado entornos específicos para llevar la enseñanza de programación a los niños más pequeños (Toon Talk [58], Scratch Jr [40], etc). Los lenguajes basados en bloques han sido muy utilizados para enseñar a programar en escuela primaria. Morgado et al. [84] proponen planificaciones de clases para estudiantes de nivel inicial y primario basados en sus experiencias en el aula. Pudieron documentar como estudiantes de nivel primario se motivaron al aprender conceptos como parámetros, paralelismo, concurrencia, procedimientos, entrada y salida utilizando el entorno Toon Talk. Como resultado de las experiencias que llevaron a cabo, propusieron ciertas estrategias para poder enseñar cada uno de estos conceptos. A pesar de que la mayoría de las intervenciones para enseñar programación a niños en edad preescolar y primaria logran una alta participación de los estudiantes, aún necesitamos entender más cómo el uso de estos entornos promueve el desarrollo de habilidades y el aprendizaje de conceptos fundamentales de programación y cuáles son los obstáculos que se presentan.

Son pocos los estudios existentes que exploran los conceptos de programación que los estudiantes pueden ir incorporando en diferentes edades, para de esta manera poder definir un diseño curricular. En definitiva no existen estudios profundos y sistemáticos sobre adquisición de lenguajes de programación en nivel primario. En particular, no existen estudios de una duración mayor a un año. La mayor parte de los estudios incluyen unas pocas clases.

Como describimos en el Capítulo 4 y en [72, 75] niños entre 4 y 10 años pudieron aprender y aplicar conceptos fundamentales como secuencia, ciclos y condicionales utilizando actividades Unplugged y entornos con lenguajes basados en bloques. Magnenat et al. [69] implementaron la enseñanza de programación con robots a estudiantes de distintas escuelas primaria utilizando un lenguaje basado en bloques para programar las acciones de los robots. Al comparar el desempeño de los grupos, encontraron que la mayoría de los estudiantes comprendieron y resolvieron tareas simples como lograr que el robot se mueva al presionar un botón.

Por otro lado Dagiene et al. [32] compararon como estudiantes de entre 7 y 12 años de Finlandia, Suecia y Lituania, resolvieron ejercicios relacionados a las Ciencias de la Computación. Utilizando preguntas de múltiple opción, evaluaron conceptos como análisis de grafos, algoritmos de búsqueda, estructuras de datos, pattern matching, pero no específicamente relacionados a la adquisición de lenguajes de programación ni a los errores asociados . No encontraron grandes diferencias entre los grupos de diferentes edades, pero sí en cuanto a los conocimientos previos de los estudiantes. Los autores sugieren que tanto el contexto educativo, la calidad académica y los conocimientos previos de los estudiantes están fuertemente relacionados con el aprendizaje de los conceptos relacionados al pensamiento computacional, como sucede con muchas áreas del aprendizaje.

Hay un interés importante a nivel mundial en introducir a los niños a la programación y se

han desarrollado muchas herramientas educativas [41]. Sin embargo, necesitamos poder evaluar sistemáticamente los errores, el impacto en el aprendizaje de los conceptos y la transferencia de un lenguaje de programación a otro. En las próximas secciones describiremos el impacto de la programación basada en bloques en los estudiantes de primaria al momento de pasar a un lenguaje de programación basado en texto.

5.1.2. Entorno de interactividad formativa

En esta sección introducimos el lenguaje de programación basado en texto (Gobstones [68]) y el entorno online con interactividad formativa de errores sintácticos y semánticos (Mumuki [15]), los cuales fueron utilizados para implementar los experimentos diseñados. En la Figura 5.1 podemos observar un ejercicio dentro del entorno Mumuki diseñado para primaria. El programa escrito en el editor de texto a la derecha de la figura está escrito en el lenguaje Gobstones basado en texto.

Figura 5.1: Interfaz del entorno de programación Mumuki. En el panel izquierdo observamos el enunciado del ejercicio. En el panel derecho encontramos el editor de texto con un programa escrito en el lenguaje de texto Gobstones.

Mumuki [12], como se puede ver en la figura, incluye una barra de progreso la cual nos indica que ejercicios han sido resueltos (en color verde), aquellos que han sido intentados de resolver pero el programa no es ejecutable ya que tienen errores de sintaxis (en rojo oscuro), aquellos que son semánticamente incorrectos (en rojo claro), y aquellos que el usuario todavía no ha intentado resolver (en gris). Los estudiantes pueden resolver los ejercicios en el orden que ellos prefieran, aunque Mumuki sugiere un orden con la barra de progreso. El ejercicio que está tratando de resolver se marca con un punto azul. En la figura se muestra el nombre y la descripción del ejercicio a resolver y también dos grillas: la grilla inicial y la grilla final esperada. Los estudiantes tienen que escribir un programa en Gobstones que transforme la grilla inicial en la grilla final esperada. A partir de este objetivo, definimos dos posibles tipos de errores.

Un estudiante comete un *error sintáctico* si el programa no compila por no cumplir con la sintaxis de Gobstones y es marcado como *rojo oscuro* por Mumuki. Un estudiante comete un *error semántico* cuando el programa creado por el estudiante llega a una grilla distinta a la esperada. Es decir que el comportamiento no es el esperado. Los errores semánticos son marcados en *rojo*

claro por Mumuki. Un programa etiquetado como rojo claro es sintácticamente correcto pero no cumple con el objetivo del ejercicio. Luego de describir el lenguaje Gobstones ejemplificamos ambos tipos de errores.

Gobstones [68] es un lenguaje de programación diseñado inicialmente para introducir los conceptos fundamentales de la programación para estudiantes universitarios que no tienen experiencia previa en programación. Gobstones es un lenguaje de programación imperativo cuya sintaxis es una versión simplificada de C, y cuyo output está restringido a mover bolitas de colores en un tablero o grilla. Martinez-Lopez et al. en [68], describen los tres elementos que son parte de la filosofía de Gobstones: un output gráfico, una sintaxis simple y un nivel de expresividad medio que permite hacer foco en conceptos fundamentales de programación. En la Figura 5.1 el output gráfico es un tablero rectangular cuyo tamaño se puede modificar, compuesto por celdas las cuales pueden contener bolitas de colores, como la azul que se ve en la figura. Las bolitas pueden ser manipuladas por un cabezal que solo puede estar sobre una celda por vez. La celda sobre la cual está el cabezal es aquella cuyo interior es de color amarillo y el borde de color rojo. El cabezal puede ejecutar las siguientes primitivas definidas en [68]. `Poner:` pone un bolita del color elegido (puede ser Rojo, Verde, Azul o Negro) en la celda donde se encuentra el cabezal. `Sacar`: saca un bolita del color elegido en la celda donde se encuentre el cabezal si la bolita se encuentra en el tablero. En caso contrario, el tablero se rompe (explota). `Mover`: mueve el cabezal en la dirección elegida (Norte, Sur, Este, Oeste) si es posible moverse en esa dirección. En caso contrario, el tablero explota.

Figura 5.2: Feedback generado por Mumuki para el programa en Gobstones que se observa en la Figura 5.1. El programa es sintácticamente correcto pero su ejecución es incorrecta.

El ejercicio de la Figura 5.1 le pide a los estudiantes que escriban un programa que ponga una bolita azul, cuatro celdas al este desde la posición inicial. El panel de la derecha incluye un editor donde el estudiante puede escribir la solución. En la figura, el programa enviado está marcado como rojo claro. El feedback formativo generado por Mumuki para este programa incorrecto se puede observar en la Figura 5.2. La respuesta de Mumuki muestra que el tablero obtenido a partir del programa, es diferente al tablero final esperado. En la Figura 5.3 observamos un programa correcto para el ejercicio. La diferencia centre el programa incorrecto (de la Figura 5.1) y el programa correcto (de la Figura 5.3) es que la primitiva `Poner(Azul)` se encuentra fuera del cuerpo del repetir en el programa correcto.

```
SOLUCIÓN: program{
            repeat(4){
                Mover(Este)
            }
            Poner(Azul)
          }
```

FEEDBACK:

Figura 5.3: Programa correcto para el ejercicio definido en la Figura 5.1 y el feedback formativo correspondiente que genera Mumuki.

En la Figura 5.4 el estudiante cometió un error de sintaxis. Se olvidó de cerrar un paréntesis. En este caso el ejercicio es evaluado como rojo oscuro porque el código no es sintácticamente correcto y, por lo tanto, no se puede ejecutar.

```
SOLUCIÓN:  program{
               repeat(4){
                   Mover(Este
               }
               Poner(Azul)
           }
```

FEEDBACK:

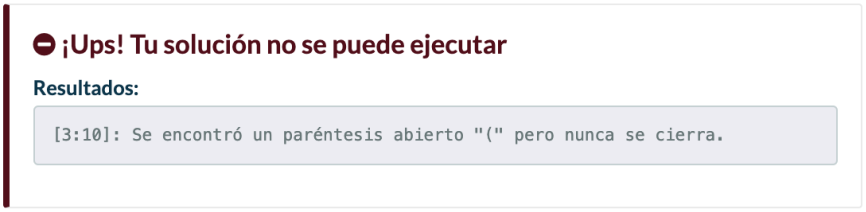


Figura 5.4: Programa con error de sintaxis para resolver el ejercicio de la Figura 5.1 y el feedback formativo correspondiente que genera Mumuki.

Mumuki almacena todos los programas enviados por un estudiante en su base de datos. La marca de tiempo, el código y los resultados de los casos de test también son almacenados, por lo que se puede reproducir la historia completa. En el Apéndice A enumeramos cada uno de los campos que almacena Mumuki. Esta información podría ser utilizada por un docente para mejorar los ejercicios, explorar errores comunes, etc. En esta sección utilizamos la información para analizar los tipos de errores cometidos por los estudiantes.

5.1.3. Diseño del estudio

Aquí describimos el estudio observacional diseñado para evaluar las hipótesis *Ha* y *Hb*. Primero presentamos las escuelas participantes, el diseño de las experiencias y la experiencias previas de los estudiantes en programación. Luego describimos las herramientas de recolección de datos utilizados, y la información obtenida a partir de los mismos para cada escuela.

Características de los grupos

Las intervenciones fueron realizadas en dos escuelas primarias públicas de gestión privada. No hay niños debajo de la línea de pobreza y la mayoría de ellos forma parte de familias pertenecientes a la clase media. Las escuelas se encuentran en el mismo barrio. En ambas experiencias enseñamos Gobstones utilizando Mumuki. Ambos experimentos se llevaron a cabo en las escuelas durante el horario escolar con estudiantes de 10 y 11 años de edad, que asistían al quinto grado de la escuela primaria. Una de estas escuelas se llama Escuela Nueva Juan Mantovani (ENJM), y la otra, Nuestra Señora de Valle (NSDV).

La experiencia completa duró 5 horas en cada escuela. Mientras usaban Mumuki, los estudiantes podían trabajar de forma individual o en grupos de dos o tres. El mismo docente estuvo a cargo de la experiencia en ambas escuelas. Se pidió a los estudiantes que completaran 23 ejercicios de programación organizados en 2 lecciones de Mumuki. A través de estos ejercicios, se introdujo a los estudiantes a conceptos de programación como secuencia, condicionales, ciclos y parámetros. Después de la experiencia, los estudiantes completaron un examen de múltiple opción.

Los estudiantes de ENJM tenían experiencia previa en programación con el lenguajes de bloques Scratch [103, 70] programando videojuegos y animaciones. Los estudiantes tuvieron 1 hora de programación por semana durante 3 semestres antes de participar de la intervención que presentamos aquí. La enseñanza de la programación se llevó a cabo en base a proyectos y utilizando la resolución de problemas y la exploración como ejes pedagógicos. Los proyectos y secuencias didácticas utilizadas pueden encontrarse en uno de los manuales para la enseñanza de la programación desarrollados en conjunto por la Fundación Sadosky y la Universidad Nacional de Córdoba [2].

Los estudiantes de la escuela NSDV, no tenían experiencia previa en programación. El equipo directivo y docente de la institución nos informaron que la programación no formaba parte de su plan de estudios. Los estudiantes confirmaron que no han habían participado de experiencias relacionadas a la programación fuera del contexto escolar.

Recolección de datos

Para poder medir el impacto de la intervención, analizamos los programas enviados por los estudiantes utilizando el entorno Mumuki, e implementamos un examen de múltiple opción.

En total 129 estudiantes participaron de las dos experiencias. 63 estudiantes de la ENJM y 66 de NSDV. Los estudiantes de la escuela Mantovani se organizaron en 37 grupos para utilizar Mumuki. Los grupos podían ser de 2 o 3 personas o individuales. En el caso de la escuela NSDV se organizaron en 31 grupos de 2 o 3 estudiantes. No hubo grupos individuales en NSDV. Cada una de las experiencia duró en total 5 horas. Debido a tiempos escolares particulares en la experiencia ENJM se llevaron a cabo 5 reuniones de una hora, mientras que en NSDV, la experiencia duró 2 reuniones de dos horas y media cada una.

Los grupos de estudiantes que participaron en las experiencias produjeron un total de 1866 programas con errores al intentar resolver una mediana de 17 ejercicios en cada escuela. La

Tabla 5.1 presenta el total de programas con errores por escuela. Los 31 grupos de estudiantes de NSDV cometieron 1025 errores, y los 37 grupos de ENJM cometieron 841 errores. Aunque había más grupos en ENJM, cometieron menos errores que los estudiantes de NSDV. La tabla también reporta el número de ejercicios intentados y el tiempo dedicado para resolver los ejercicios por escuela. El tiempo dedicado fue definido como la suma de minutos entre los programas enviados para todos los grupos estudiantes de una misma institución en un día determinado. Se puede observar que el tiempo total dedicado para ambos grupos es similar. Es equivalente a unas 50 horas en total. Esto da un promedio de aproximadamente 1 hora y 45 minutos para cada grupo. Esto es una estimación de lo que normalmente en pedagogía se conoce como tiempo concentrado en tarea.

Escuela	**ENJM**	**NSDV**
Cantidad de Grupos	37	31
Total de ejercicios intentados	607	611
Tiempo dedicado (en minutos)	3046	3072
Total de errores sintácticos	305	385
Total de errores semánticos	536	640
Total de errores	841	1025

Tabla 5.1: La parte superior compara métricas básicas en el uso del entorno en ambas escuelas. La parte inferior de la tabla compara los cantidad de tipos de errores por escuela.

Una vez finalizada la experiencias, los estudiantes de forma individual, resolvieron un examen de múltiple opción, el cual no era obligatorio. El examen estaba compuesto por 6 preguntas múltiple opción. Cada pregunta estaba compuesta por un programa escrito en Gobstones y un tablero inicial. Los estudiantes debían, en base el código y el tablero inicial, elegir uno de cuatro posibles tableros finales, que fuese producto de ejecutar el programa sobre el tablero inicial. Un ejemplo de una pregunta del examen puede observarse en la Figura 5.5. La respuestas correcta es la Opción 2.

Todas las preguntas tuvieron el mismo peso en la puntuación final. La escala de puntaje del examen fue de 0 a 10. En La tabla 5.2 se informa la cantidad de estudiantes por escuela y género que participaron en la experiencia y que decidieron resolver el examen de múltiple opción.

Escuela	**ENJM**			**NSDV**		
Género (F/M) y Total	**F**	**M**	**T**	**F**	**M**	**T**
Total de estudiantes participantes	30	33	63	37	29	66
Total de estudiantes evaluados	26	31	57	18	11	29

Tabla 5.2: Distribución de participantes (el examen no fue obligatorio).

5.1.4. Resultados y análisis

En esta sección presentamos los resultados obtenidos teniendo en cuenta las dos hipótesis formuladas en 5.1. Utilizando los datos guardados por Mumuki para cada solución enviada por los estudiantes, realizamos una comparación en la cantidad se programas con errores sintácticos y semánticos en las dos escuelas. Recordemos que cada programa fue enviado por grupos de 1, 2 o 3 estudiantes. Luego, presentamos los resultados correspondientes al examen de múltiple opción.

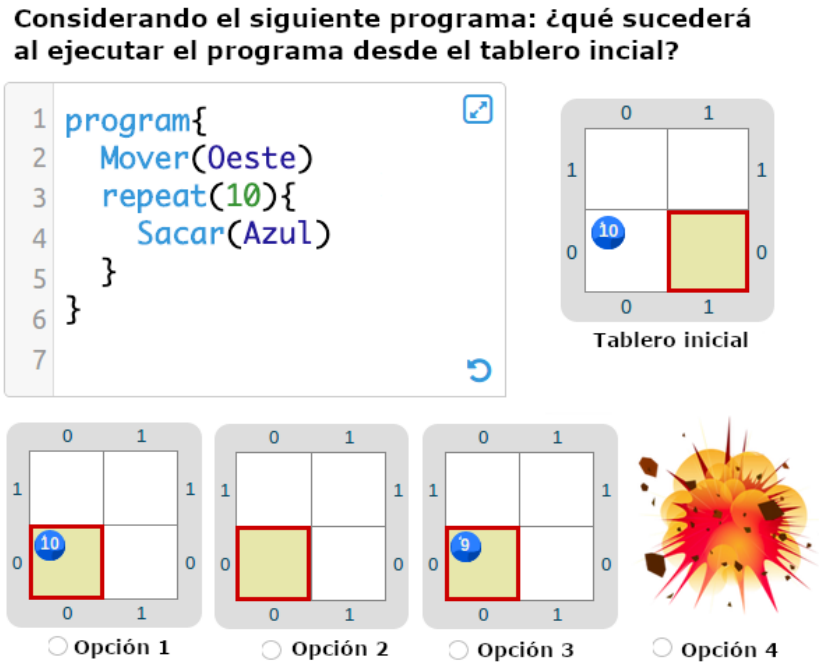


Figura 5.5: Ejemplo de ejercicio utilizado en el examen múltiple opción

El examen fue resuelto de forma individual. Finalmente reflexionamos sobre algunas limitaciones del experimento y su implementación.

Resultados en base a los programas enviados

En las Tablas 5.4 y 5.3 podemos observar la cantidad de errores sintácticos y errores semánticos generados por los grupos de estudiantes de las escuelas ENJM y NSDV. La cantidad de *ejercicios intentados* por un grupo es el número de ejercicios diferentes para los cuales el grupo envió al menos un programa. Los *programas con errores de sintaxis* son programas enviados dentro de Mumuki que tienen errores de sintaxis. Los *programas con errores semánticos* son los que contienen errores en los casos de test. Notar que puede haber más de un programa enviado (y frecuentemente hay) por ejercicio intentado. El total para las primeras 3 columnas indica la sumatoria. En cambio para las últimas 2 columnas se indica el promedio.

Para poder comparar los resultados de las escuelas al momento de resolver los diferentes ejercicios de programación en Mumuki definimos dos métricas. El *ratio semántico por grupo* es la cantidad de programas enviados con errores semánticos por un grupo estudiantes dividido la cantidad de ejercicios intentados por ese grupo de estudiantes. El *ratio sintáctico por grupo* es análogo al ratio semántico pero considerando los errores de sintaxis. En la Tabla 5.5 en la columna ratio, se reportan los promedios del ratio semántico y del ratio sintáctico para cada una de las escuelas. Además se reportan la cantidad total de programas con errores sintácticos y errores semánticos para cada institución.

Además en la tabla se presentan los resultados de aplicar unpaired t-tests entre los promedios de los ratios de ENJM Y NSDV para ambas métricas definidas. El promedio del ratio semántico en la escuela ENJM es de 1.01 mientras que en la escuela NSDV el promedio es de 0.74.

En otras palabras, después de superar los errores sintácticos, los grupos ENJM cometen errores semánticos en 7 de cada 10 de ejercicios intentados en promedio. Por otro lado, los

Grupo	Ejercicios intentados	Programas con errores de sintaxis	Programas con errores semánticos	Ratio sintáctico	Ratio semántico
1	24	8	19	0.33	0.79
2	17	6	29	0.35	1.71
3	16	8	3	0.5	0.19
4	29	5	25	0.17	0.86
5	16	20	24	1.25	1.5
6	16	4	27	0.25	1.69
7	17	22	30	1.29	1.76
8	37	48	53	1.3	1.43
9	18	10	28	0.56	1.56
10	40	48	71	1.2	1.78
11	17	11	2	0.65	0.12
12	10	16	5	1.6	0.5
13	27	29	29	1.07	1.07
14	29	16	41	0.55	1.41
15	13	5	7	0.38	0.54
16	10	12	8	1.2	0.8
17	14	7	11	0.5	0.79
18	17	6	25	0.35	1.47
19	12	13	18	1.08	1.5
20	27	3	10	0.11	0.37
21	11	4	11	0.36	1
22	17	10	6	0.59	0.35
23	27	17	26	0.63	0.96
24	16	8	26	0.5	1.63
25	30	11	16	0.37	0.53
26	17	10	19	0.59	1.12
27	15	4	10	0.27	0.67
28	16	3	11	0.19	0.69
29	17	3	18	0.18	1.06
30	21	10	21	0.48	1
31	18	8	11	0.44	0.61
Total	611	385	640	0.62	1.01

Tabla 5.3: En la tabla observamos el desempeño de los 31 grupos de estudiantes de la escuela NSDV en base a: la cantidad de ejercicios que intentaron resolver, la cantidad de programas con errores sintácticos y errores semánticos enviados, ratio sintáctico y ratio semántico.

Grupo	Ejercicios intentados	Programas con errores de sintaxis	Programas con errores semánticos	Ratio sintáctico	Ratio semántico
1	28	12	28	0.43	1
2	2	7	0	3.50	0
3	18	7	23	0.39	1.28
4	21	10	17	0.48	0.81
5	23	15	8	0.65	0.35
6	10	3	8	0.30	0.8
7	27	24	44	0.89	1.63
8	15	4	23	0.27	1.53
9	3	3	1	1.00	0.33
10	14	4	12	0.29	0.86
11	5	6	0	1.20	0
12	1	0	0	0.00	0
13	15	0	4	0.00	0.27
14	27	21	18	0.78	0.67
15	20	5	19	0.25	0.95
16	28	7	35	0.25	1.25
17	5	9	1	1.80	0.2
18	24	22	32	0.92	1.33
19	13	12	8	0.92	0.62
20	6	8	4	1.33	0.67
21	12	11	3	0.92	0.25
22	22	4	14	0.18	0.64
23	17	13	20	0.76	1.18
24	15	1	5	0.07	0.33
25	20	0	33	0.00	1.65
26	39	16	29	0.41	0.74
27	17	4	4	0.24	0.24
28	23	12	18	0.52	0.78
29	9	10	6	1.11	0.67
30	1	3	0	3.00	0
31	19	3	8	0.16	0.42
32	30	1	29	0.03	0.97
33	21	21	27	1.00	1.29
34	18	4	23	0.22	1.28
35	12	13	13	1.08	1.08
36	15	6	10	0.40	0.67
37	12	4	9	0.33	0.75
Total	607	305	536	0.70	0.74

Tabla 5.4: En la tabla observamos el desempeño de los 31 grupos de estudiantes de la escuela ENJM en base a: la cantidad de ejercicios que intentaron resolver, la cantidad de programas con errores sintácticos y errores semánticos enviados, ratio sintáctico y ratio semántico.

	ENJM		NSDV		ratio t-test
	#	ratio	#	ratio	p
Errores sintácticos	305	0.70	385	0.62	0.588
Errores semánticos	536	0.74	640	1.01*	0.024

Tabla 5.5: Total de errores sintácticos y semánticos por escuela. Promedio de los ratios sintácticos y semánticos por escuela. Las diferencias estadísticas significativas para los ratios en base a unpaired t-test se marcan con *.

grupos de estudiantes de la escuela NSDV cometen errores semánticos en 10 de cada 10 ejercicios intentados. La unpaired t-test nos indica que hay una diferencia estadísticamente significativa entre los grupos de ENJM y NSDV (p = 0.024) al momento de evaluar el promedio de los ratios semánticos.

Estos resultados nos dan evidencia para poder rechazar la hipótesis nula Hb (null) a favor de la alternativa: los grupos con experiencia previa en programación basada en bloques cometen un cantidad significativamente menor de errores semánticos que aquellos grupos que no tienen experiencia.

Con respecto al promedio del ratio sintáctico no hay diferencia significativa entre las escuelas en base al unpaired t-test, como se puede observar en la tabla (p =0.588). Por lo tanto, la hipótesis Ha (null) no puede ser rechazada. Este resultado es coherente con el hecho de que ninguno los grupos de estudiantes de ambas escuelas tienen experiencia con lenguajes de programación basados en texto, por lo que son esperables proporciones similares de errores sintácticos.

Resultados de los exámenes múltiple opción

En la Figura 5.6 podemos observar un diagrama de cajas donde se representan los notas de los exámenes para ambas escuelas (el máximo puntaje es 10).

La mediana para las notas en NSDV es 6.66 y para ENJM es 8.33 (la diferencia es 1.67 puntos). El examen fue diseñado para incluir respuestas que contengan errores conceptuales comunes de programación. Por ejemplo, en la Figura 5.5 las opciones incorrectas (1, 3 y 4) ilustran diferentes errores conceptuales comunes sobre el comportamiento de los ciclos que se reportan en [126]. La diferencia entre el promedio de las calificaciones de las escuelas es estadísticamente significativa en base a unpaired t-test (p = 0.0014), lo que proporciona más evidencia para rechazar la hipótesis nula Hb (null) a favor de la alternativa.

Limitaciones del estudio

En esta sección discutimos las posibles limitaciones del estudio. Primero, no hay una evaluación formal de la experiencia previa en programación de los estudiantes de la escuela NSDV. La experiencia previa es informada por docentes, directores y los propios estudiantes como se describe en la Sección 5.1.3. Cuando se les preguntó a los estudiantes sobre su experiencia previa en programación, todos afirmaron que no tenían.

En segundo lugar, las experiencias, a pesar de haber durado en total 5 horas, tuvieron diferentes implementaciones en cuanto a tiempo y cantidad de clases. Los estudiantes de ENJM participaron de 5 clases de una hora, mientras que los estudiantes de NSDV tuvieron 2 clases de 2.5 horas. Las clases fueron semanales para ambas escuelas. El docente a cargo notó que los estudiantes necesitaban repasar ciertos conceptos al comienzo de cada clase. Sin embargo, no pareció afectar el tiempo dedicado a la resolución de ejercicios como muestra la Tabla 5.1.

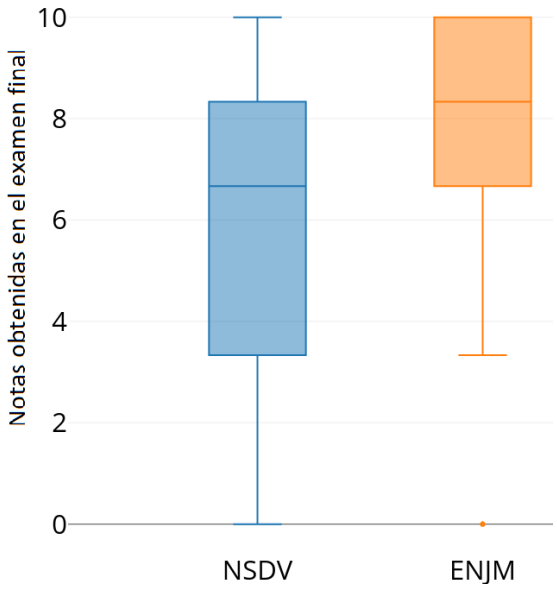


Figura 5.6: Diagrama de caja de las notas de examen para ambas escuelas.

Finalmente, sólo 29 de los 66 estudiantes que participaron en la experiencia en la escuela NSDV decidieron resolver el examen, como se informa en la Tabla 5.2. En la escuela ENJM, 57 de 63 decidieron realizar el examen. El examen no fue obligatorio en ninguna de las dos instituciones. Si bien los estudiantes pueden tener diferentes motivos para no realizar un examen, la confianza que tengan en sí mismos con respecto al desempeño en el examen puede ser una de ellas.

5.1.5. Conclusiones preliminares

El estudio que desarrollamos y describimos se realizó en dos escuelas primarias con estudiantes de 10-11 años. En una de las escuelas los estudiantes tenían 1 año y medio de experiencia aprendiendo programación con un lenguaje basado en bloques, mientras que en la otra institución los estudiantes no tenían experiencia previa en programación. En este contexto, exploramos la pregunta de investigación Q1: *Los niños con experiencia en programación con lenguajes de bloques ¿adquieren habilidades de programación que facilitan el aprendizaje de un lenguaje basado en texto?*

Como describimos anteriormente, realizamos un análisis comparativo entre las dos escuelas, en base a los programas enviados por los estudiantes al utilizar Mumuki y los resultados del examen múltiple opción. Para ello evaluamos las hipótesis no direccionales Ha y Hb. Ha está relacionada con la cantidad errores sintácticos y Hb con los errores semánticos que cometen los estudiantes al incorporar un nuevo lenguaje de programación basado en texto.

Al evaluar Ha, encontramos que los estudiantes de primaria cometen una cantidad similar de errores sintácticos independientemente de su experiencia previa con lenguajes de bloques. Este hallazgo es coherente con el trabajo previo [99, 92] que argumenta que los lenguajes basados en bloques no abordan problemas sintácticos sino que simplemente los dejan para más adelante. Dicho trabajo previo se realizó a nivel universitario mientras que el nuestro es en escuela primaria. Una razón para comenzar con los lenguajes basados en bloques podría ser que no todo se puede aprender al mismo tiempo. Weintrop y Wilensky [126] compararon grupos separados de estudiantes de secundaria aprendiendo programación. Un grupo utilizó un lenguaje basado en

bloques mientras que el otro aprendió un lenguaje basado en texto. Sus resultados sugieren que la comprensión de los conceptos fundamentales de programación (como condicionales y ciclos) en estudiantes principiantes es mejor en lenguajes basados en bloques. Sin embargo este estudio es criticado porque los exámenes de los 2 grupos no son iguales. Nuestro diseño experimental permite evaluar a los 2 grupos con las mismas herramientas.

Para evaluar la hipótesis Hb analizamos los programas enviados por los estudiantes al utilizar Mumuki y los exámenes múltiple opción. En las secuencias de programas encontramos que, luego de poder resolver los errores sintácticos, los estudiantes con experiencias en lenguajes basados en bloques, cometen significativamente menos errores semánticos que los estudiantes sin experiencia. En promedio, los grupos de estudiantes con experiencia cometen errores semánticos en 7 de cada 10 ejercicios intentados mientras que los grupos sin experiencia cometen errores semánticos en 10 de cada 10 ejercicios intentados. Los ejercicios de Gobstones son en general como el que describimos en la Figura 5.1, y permiten realizar pequeñas modificaciones en el código que puedan tener un gran impacto para llegar a una solución correcta. Por lo tanto, es posible que a través de prueba y error, los estudiantes puedan llegar a una solución correcta mientras puedan seguir existiendo una comprensión errónea de los conceptos de programación. Al considerar la cantidad de errores semánticos, observamos más prueba y error en la escuela sin experiencia previa en lenguajes basados en bloques.

Además, para poder verificar la hipótesis Hb analizamos los exámenes de múltiple opción. El examen fue diseñado para proporcionar dentro en las respuestas alternativas conceptos erróneos comunes sobre los conceptos de programación [126, 3, 122] asociados a construcciones de programación fundamentales como respuestas alternativas incorrectas. Encontramos que las calificaciones en el examen es significativamente mayor para los niños con experiencia en programación basada en bloques. Con base en los registros y los resultados del examen, concluimos que el grupo con experiencia previa basada en bloques cometió menos errores semánticos que aquellos sin experiencia.

En resumen, hemos presentado evidencia de que los estudiantes de 10-11 años con experiencia basada en bloques aprenden un nuevo lenguaje basado en texto más fácilmente que estudiantes similares sin experiencia basada en bloques.

Meerbaum-Salant et al. [77] argumentan que los lenguajes basados en bloques generan malas prácticas de programación cómo el abuso de la prueba y el error en los primeros meses de aprendizaje y se preguntaba si esta sería una práctica que superaría o no. Ellos plantean la pregunta: *¿debemos hacer las cosas "fáciles" para los estudiantes durante sus estudios iniciales o debemos enseñarles el "camino correcto" desde el principio?* Nuestra evidencia empírica respalda la afirmación de que los niños con experiencia basada en bloques desarrollan habilidades de programación que facilitan el aprendizaje de un lenguaje basado en texto. Al comenzar desde la escuela primaria, la forma "fácil" podría ser un paso hacia la "forma correcta".

5.2. Experiencias en la universidad

En esta sección describimos y analizamos experiencias de enseñanza de programación con el lenguaje de programación Haskell en nivel universitario utilizando el entorno web online de enseñanza de programación Mumuki. Comparamos la adquisición de lenguaje en dos grupos diferentes de estudiantes. Para uno de los grupos es su primer lenguaje de programación, mientras que el otro grupo ya sabe programar en otros lenguajes.

Cómo describimos en el Capítulo 3, a pesar de haber cada vez más entornos de programación desarrollados para dar soporte al aprendizaje y enseñanza de la programación, son pocos los

utilizados dentro del sistema formal educativo. Brusilovsky et al. [22] encontraron que una de las razones más comunes de la desconfianza que presentan los docentes a nuevos entornos es que su utilidad y beneficios no hayan sido demostrados por experiencias previas. En general, las evaluaciones de estos entornos utilizan métricas ad-hoc y se realizan a partir de experimentos de laboratorio aplicadas en un solo curso piloto. Una excepción a esta situación son las experiencias implementadas por [57], donde se argumenta la necesidad crítica de replicar los estudios en diferentes contextos y de evaluar varios factores. También es necesario utilizar métricas estándar para poder comparar los resultados con trabajo relacionado. En las experiencias realizadas analizamos el efecto de Mumuki en el proceso de aprendizaje considerando los altos porcentajes de deserción o abandono que tienen los cursos introductorios a la programación que fueron analizados en trabajo previo [125].

La sección está organizada de la siguiente manera. Primero revisamos trabajos previos sobre enseñanza de programación funcional con entornos de programación cerrados y trabajos previos sobre el efecto de los entornos cerrados sobre el rendimiento y percepción de los estudiantes. Luego, describimos el entorno web online Mumuki diseñado originalmente para nivel universitario. A continuación introducimos el diseño de las experiencias implementadas. Finalmente, presentamos y analizamos los resultados obtenidos a partir de las experiencias.

5.2.1. Trabajo previo

En los últimos años entornos web online que proveen ejercicios y feedback formativo automático han tomado importancia para la enseñanza de programación para principiantes. Entornos como Codingbat [91], CodeRunner [67], CodeLab [4], CloudCoder [115] son ejemplos de ello. Los resultados de experiencias de enseñanza de programación donde se utilizan este tipo de entornos, muestran efectos positivos en los estudiantes al tratar de incorporar la sintaxis de lenguajes de programación como Python o Java [57]. A pesar del interés que este tema ha recibido por parte de la comunidad científica, el uso de entornos web online para la enseñanza de programación es poco implementado en los sistemas educativos formales [22]. Experiencias como la descripta por Ihantola et al. en [57] reflexionan sobre la importancia de este tipo de entornos para poder identificar estudiantes que estén teniendo complicaciones en el aprendizaje y poder detectar estudiantes con riesgo de deserción o abandono en cursos donde el seguimiento por parte del docente no es posible.

Casi todos los entornos de enseñanza de programación se centran en lenguajes imperativos. En base a nuestro relevamiento nos encontramos con un único entorno web online para Haskell desarrollado con fines educativos: Ask-Elle. Ask-Elle [45, 44] es un tutor web que comparte algunas funcionalidades con Mumuki. Por ejemplo contiene ejercicios en Haskell y genera feedback formativo de forma automática sobre los programas creados por los estudiantes. Al igual que Mumuki, evalúa los programas de los estudiantes utilizando casos de prueba.

A diferencia de Mumuki, Ask-Elle fue creado como un tutor que proporciona pistas a los estudiantes en los pasos intermedios que va realizando el mismo para poder programar una solución que resuelva el ejercicio planteado. Ask-Elle intenta predecir qué pasos básicos deben tomarse para llegar a una solución. Gerdes et al. [44] denominan a este conjunto de pasos básicos una “estrategia de programación”. Para conseguir los mismos, los derivan de forma automática a partir de modelos de soluciones provistas por los docentes. Sin embargo, a partir de los experimentos que realizaron encuentran que es muy difícil poder predecir el tamaño del próximo paso que darán los estudiantes. Además, frecuentemente las soluciones escritas por un estudiante no se pueden comparar con una de las soluciones modelo proporcionadas por los docentes. Como resultado, no pueden proveer feedback formativo sobre casi el 40 % de las soluciones generadas.

Los autores señalan que los estudiantes suelen dar pasos entre solución y solución más grandes que los esperados por el sistema, y muchas veces presentan soluciones correctas en un solo paso. Mumuki espera una solución completa cada vez que un estudiante envía el programa. Por lo tanto, puede proporcionar feedback formativo para todas las soluciones recibidas.

Otra diferencia entre los dos entornos, es que Ask-Elle requiere que los docentes definan las soluciones modelo para cada ejercicio, mientras que para Mumuki no es necesario. Ask-Elle usa QuickCheck [26] que genera casos de test aleatorios. Los desarrolladores de QuickCheck reconocen que la principal limitación del software es que no se mide la cobertura de la prueba, ya que los casos de prueba se generan al azar. Mumuki requiere que los docentes definan manualmente los casos de prueba para que se pueda poner especial atención en los casos bordes y ramas problemáticas. A diferencia de Ask-Elle, Mumuki proporciona una consola de Haskell para cada ejercicio donde los estudiantes pueden probar sus propios casos de prueba.

Para evaluar Ask-Elle los autore desarrollaron una única experiencia en la Universidad de Utrecht. En la misma participaron 40 estudiantes, los cuales utilizaron el entorno durante 2 clases. En el estudio no se informa cuánto tiempo duraron las clases. Para poder evaluar el impacto de Ask-Elle utilizaron un cuestionario, el cual no reportan si está basado en alguna metodología estándar. Al momento de evaluar al entorno en base a la pregunta *"La pista que el tutor recomendaba se correspondía con la intuición del estudiante"*, tuvo un puntaje promedio de 2.82 en base a una escala Likert del 1 al 5. No se informa el efecto sobre los abandonos del curso y las proporciones de aprobación y no aprobación del mismo.

Kumar [64] argumenta que los entornos online pueden ser utilizados para aumentar la confianza de los estudiantes de programación. Esto es particularmente útil en los primeros cursos de carreras relacionadas a programación, cuando la confianza en sí mismo del estudiante es menor. Kumar también afirma que el impacto es más fuerte en las estudiantes. Sin embargo, no informan el efecto de estos entornos a partir de los resultados de los estudiantes en las evaluaciones correspondientes a la materia, como realizamos en nuestra investigación.

Hasta donde exploramos, solo hay un estudio [4] donde los autores implementaron con éxito, en dos instituciones diferentes, el uso de un entorno online para la enseñanza de la programación. Los resultados son alentadores. Sin embargo, la comparación es puramente cualitativa, y no se informa el impacto del entorno CodeLab sobre métricas cuantitativas, como las proporciones de aprobación o abandono. Para poder evaluar la interactividad formativa de Mumuki, evaluamos el impacto utilizando métricas de aprobación y deserción siguiendo el trabajo previo [115]. También para poder evaluar la percepción de los estudiantes sobre el entorno Mumuki, adaptamos Modelo de Aceptación de Tecnología (del inglés (Technology acceptance model (TAM)) [25].

5.2.2. Entorno de interactividad formativa

Mumuki es un entorno digital inteligente online para la enseñanza de programación que provee de manera automática feedback formativo. La misma fue desarrollada por un equipo de docentes de universidades públicas Argentinas. Es open source[2] bajo la licencia GPLv3 MIT. Mumuki brinda soporte para 17 lenguajes de programación, incluyendo Haskell, Prolog, Python, JavaScript, C, Ruby, Gobstones entre otros. Mumuki incluye secuencias didácticas que introducen diferentes conceptos de programación de forma incremental a partir de guías de ejercicios. Se pueden resolver los mismos ingresando a `http://www.mumuki.io`.

Mumuki es utilizado por más de 50 instituciones educativas[3]. Se destacan 8 universidades de Argentina y los niveles educativos primarios y secundarios de las provincias de Mendoza, San

[2]El código está disponible en `https://github.com/mumuki`.

[3]La lista completa de organizaciones que utilizan Mumuki pueden encontrarse en `http://www.mumuki.org`

Luis y Tierra del Fuego. Además se utiliza en el plan Ceibal que se lleva a cabo en Uruguay. En total cuenta con más de cientos de miles usuarios registrados y miles de usuarios activos todos los meses.

Figura 5.7: Captura de pantalla de la interfaz de Mumuki. El panel de la izquierda muestra el enunciado del ejercicio. El panel de la derecha muestra el editor donde un estudiante escribió un programa para resolver al ejercicio planteado. Finalmente, en la parte inferior de la figura se muestra el feedback formativo cuando el programa es correcto.

En la Figura 5.7 se puede observar una captura de pantalla de Mumuki. La interfaz incluye una barra de progreso que muestra qué ejercicios se han resuelto de forma correcta (en verde), cuáles se han intentado pero son incorrectos (en rojo o amarillo) y cuáles aún no se han intentado (en gris). El ejercicio actual está marcado con un punto azul debajo de la barra de progreso. Los estudiantes pueden resolver los ejercicios en el orden que deseen, pero Mumuki sugiere un orden con la barra de progreso. En el Apéndice A describimos en profundidad como es la interfaz de Mumuki para los estudiantes y para los docentes.

Desde el punto de vista del estudiante, cada ejercicio incluye la descripción del problema a resolver acompañado de un ejemplo y una lista de funciones que el estudiante puede o debe reutilizar para resolver el ejercicio (como se puede ver en el panel de la izquierda de la Figura 5.7). El panel de la derecha, incluye una pestaña con un editor para el programa creado por el estudiante y otra pestaña con una consola interactiva donde el programa y las funciones reutilizables pueden ser probadas. Vamos a definir la palabra "programa" como un fragmento de código que intenta resolver un ejercicio de programación y puede ser incorrecto o estar incompleto. El feedback automático es dado al estudiante luego que presiona el botón "Enviar". El programa es probado bajo ciertos casos de test y expectativas desarrolladas por el diseñador del ejercicio. Los casos de test tienen como objetivo chequear el correcto funcionamiento del programa mientras que las expectativas permiten requerir que el programa use algún concepto específico o que reutilice alguna función dada anteriormente. El programa que se encuentra en la Figura 5.7 pasa todos los casos de test y cumple con las expectativas definidas.

Desde el punto de vista del docente el ejercicio está compuesto por dos partes. Por un lado, incluye la descripción del problema que los estudiantes tienen que resolver. Por otro lado, el diseñador del ejercicio proporciona información para poder generar de forma automática el feedback para los programas generados por los estudiantes en dos niveles diferentes que llamaremos correctitud y calidad.

Para evaluar la correctitud del programa enviado por el estudiante, el docente proporciona casos de test representativos, los cuales son ejecutados con la solución del estudiante. El programa puede incluir funciones auxiliares programadas por el profesor. En cuanto a la calidad del programa del estudiante, se realiza un análisis del árbol de sintaxis abstracta. El docente puede utilizar una herramienta brindada por Mumuki donde puede elegir patrones predefinidos que deben estar presentes en el programa enviado llamadas *expectativas* en Mumuki. Las expectativas se definen de tal modo que deben estar presentes o no en el programa del estudiante, dependiendo del objetivo del ejercicio. Por ejemplo, si el ejercicio pretende evaluar recursividad, el docente puede definir en la expectativa que la función se debe implementar de forma recursiva. Cuando el estudiante envía un programa, cada expectativa definida es evaluada.

Mumuki evalúa cada uno de los programas enviados por los estudiantes con cuatro valores posibles: rojo oscuro, rojo claro, amarillo o verde. El rojo oscura nos indica que el programa no es sintácticamente correcto, por lo tanto no es posible su ejecución. El rojo claro indica que el programa es sintácticamente correcto pero algún caso de test no es correcto. Amarillo significa que todos los casos de test son correctos pero no se cumplen algunas expectativas. Un ejercicio se etiquete verde cuando todos los casos de test son correctos y se cumplen todas las expectativas. La Figura 5.7 muestra un ejercicio evaluado como verde. A continuación ejemplificamos los colores rojo claro y amarillo.

El ejercicio de la Figura 5.7 le pide al estudiante que defina una función llamada soloNumerosPares que retorna únicamente los números pares de una lista de números dada. Por ejemplo, al aplicar la función soloNumerosPares a la lista [8,7,6,5] se espera que retorne la lista [8,6]. Los siguientes casos de test son proporcionados por el docente que diseño el ejercicio y no se presentan al estudiante hasta que envíe un programa intentando resolver el ejercicio:

- soloNumerosPares [] = [].
- soloNumerosPares [1,2,3] = [2].
- soloNumerosPares [7,14,9,10] = [14,10].

El docente también puede definir expectativas para un ejercicio. En esta caso se definieron dos expectativas. Como se espera que los estudiantes resuelvan el ejercicio utilizando la función *filter*, el uso de recursión directa no está permitido para resolver este ejercicio. Esto fue especificado por el docente como una patrón que no debería ser encontrado dentro del programa del estudiante. Además, el docente quiere motivar la reutilización de funciones previamente definidas por lo que el programa debe utilizar la función *even* provista por la biblioteca estándar de Haskell como expectativa. Esto fue especificado por el docente como un patrón que debería ser encontrado dentro de la solución del estudiante.

En la Figura 5.8 el estudiante utiliza la función *even*, pero resuelve el problema utilizando recursión directa en vez de utilizar *filter*. Mumuki informa al estudiante, a través del feedback formativo que el programa enviado pasa todos las casos de test, pero que tiene que utilizar *filter* en vez de recursión directa. Por lo tanto, el programa enviado por el estudiante es evaluada en color amarillo por Mumuki. El programa pasa todos los casos de test pero una de las expectativas definidas no se cumple.

En la Figura 5.9 el estudiante no utiliza la función *even* pero además comete un error al definir su propia versión de esta función. En vez de verificar si el módulo es igual a cero, compara que el mismo sea igual a uno. En este caso, el ejercicio es evaluado rojo claro ya que algunos de los casos de test son incorrectos.

Cada uno de los programas enviados por un estudiante se almacena, por lo tanto, el docente a cargo no sólo puede observar el programa final sino también el conjunto de programas previos. La

SOLUCIÓN:
```
soloNumeroPares [] = []
soloNumerosPares (x:xs)
  | even x    = x:(soloNumerosPares xs)
  | otherwise = soloNumerosPares xs
```

FEEDBACK:

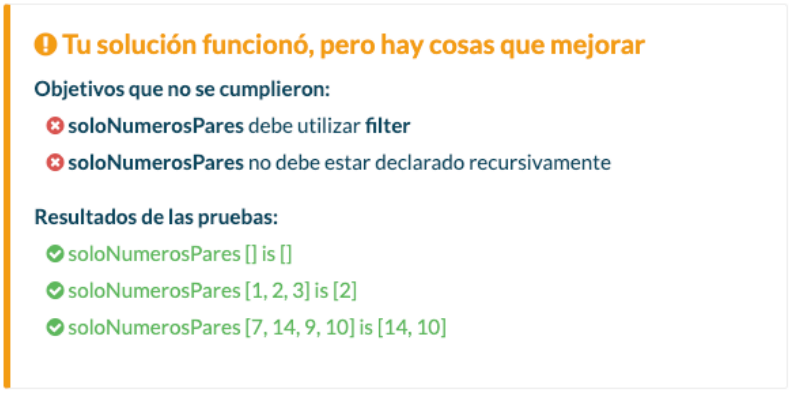


Figura 5.8: Ejemplo de una solución evaluada Amarilla por mumuki ya que no cumple con una de las expectativas definidas.

marca de tiempo, los resultados de los casos de test y de las expectativas también son guardados. De esta manera el docente puede recuperar la historia completa, mejorar los ejercicios, explorar errores comunes, etc. El profesor cuenta con una interfaz específica donde puede ver el historial de programas enviados del estudiante que se muestra en la Figura 5.10. En la figura, el estudiante ha enviado diez programas diferentes para resolver el ejercicio. El último programa está en verde ya que pasa todas las pruebas y cumple con todas las expectativas.

5.2.3. Diseño del estudio

En esta sección describimos las características de los dos cursos donde implementamos las experiencias y los estudios observacionales. Para ello, planteamos las siguientes hipótesis direccionales para ser probadas en cada uno de los cursos.

H_0 **(null)** : no hay diferencia significativa entre la proporción de abandono en un curso que utiliza Mumuki y uno que no lo hace.

H_1 **(alt)** : la proporción de abandono en un curso que utiliza Mumuki es significativamente menor que la de un curso que no lo hace.

Recolección y análisis de datos

Las experiencias fueron realizadas en dos universidades públicas de Argentina. En ambas universidades enseñamos Haskell utilizando Mumuki. Como discutimos en la sección anterior gran parte del trabajo previo existente sobre evaluación de entornos online utilizados para la enseñanza de programación se implementan en una sola institución y en un solo curso. Para la implementación de nuestras experiencias seleccionamos dos universidades públicas argentinas que introducen programación funcional con Haskell. Las universidades donde se llevaron a cabo los experimentos fueron la Facultad Regional Buenos Aires de la Universidad Tecnológica Nacional (UTN) y la Facultad de Matemática, Astronomía, Física y Computación en la Universidad Nacional de Córdoba (UNC). En la UTN introducen programación funcional en Haskell en la

```
SOLUCIÓN: soloNumerosPares ls =
            filter (\x -> x `mod` 2 == 1) ls
```

FEEDBACK:

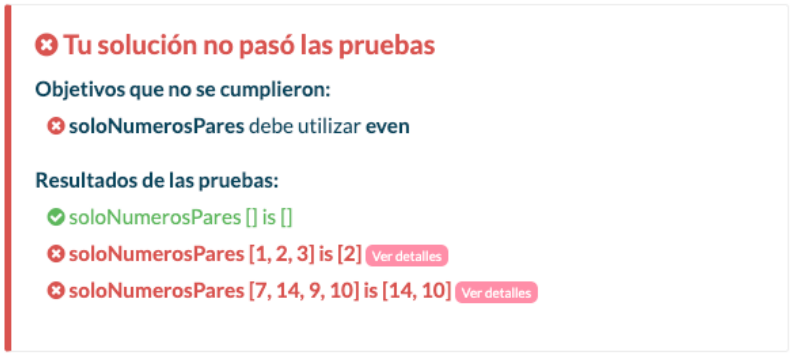


Figura 5.9: Ejemplo de un programa evaluado por Mumuki como rojo claro ya que algunos casos de test son incorrectos.

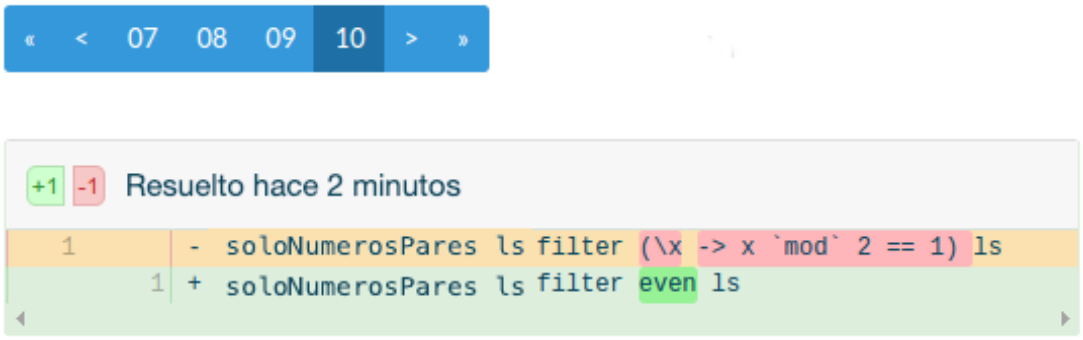


Figura 5.10: Dos programas creados para un mismo ejercicio. La diferencia entre ambas soluciones es resaltada.

materia *Paradigmas de la Programación* en segundo año de la carrera. Llamaremos a este grupo de estudiantes CS2 porque están en 2do año de la carrera y Haskell no es el primer lenguaje de programación que adquieren. En el caso de la UNC, el paradigma funcional con Haskell acompaña a los estudiantes a partir de *Introducción a los Algoritmos*, en el primer año de la carrera. Llamaremos a este grupo CS1 porque están en 1er año y Haskell es el primer lenguaje de programación que adquieren. Contar con grupos de estudiantes con diferentes características, nos permite replicar nuestro estudio observacional. En total, 114 estudiantes participaron en este estudio observacional. 59 estudiantes en el nivel CS2 y 55 estudiantes en el nivel CS1.

Los estudiantes tenían que resolver 82 ejercicios de programación de Haskell en Mumuki que abarcan conceptos introductorios de programación funcional. Para poder evaluar el aprendizaje y comprensión de los conceptos, utilizamos exámenes escritos en papel. En los exámenes los estudiantes no utilizaron Mumuki ni compiladores para verificar sus programas. El examen fue corregido manualmente por los profesores. Los estudiantes tuvieron dos oportunidades para aprobar el examen; hubo al menos un mes de distancia entre ellos.

Para evaluar la percepción de los estudiantes con el entorno Mumuki, los estudiantes completaron un cuestionario cuyo diseño esta basado en el Modelo de Aceptación de Tecnología (TAM) [25].

Experiencia CS1 en la Universidad Nacional de Córdoba

La experiencia fue implementada en la materia Introducción a los Algoritmos de la Licenciatura en Ciencias de la Computación en la Facultad de Matemática, Astronomía, Física y Computación. La materia pertenece al primer año de la carrera. Es la primer materia donde los estudiantes comienzan a programar, siendo la primer materia de programación de una carrera de 5 años. La carrera tiene una carga horaria de entre 24 y 36 horas semanales, por lo que son pocos los casos de estudiantes que trabajan y estudian.

La experiencia en la UNC se desarrolló desde agosto a noviembre de 2016. En el dictado de la materia participaron un profesor universitario, dos docentes asistentes y un ayudante alumno. Para nuestro estudio consideramos aquellos estudiantes que se matricularon y asistieron al curso por lo menos dos días. En total 55 estudiantes cumplían con estas condiciones. De los 55 estudiantes, solo 2 informaron experiencia previa en programación. La edad promedio de los estudiantes es de 21 años.

En total se llevaron a cabo 15 clases de programación funcional. Cada una de estas clases tuvo una duración de cuatro horas. El curso tuvo una carga semanal de 8 horas, la cual se dividía en dos días por semana de 9 a 13. Cada clase de 4 horas se dividió en dos partes de 2 horas.

Durante la primer parte de la clase, los estudiantes utilizaban Mumuki para programar funciones en Haskell en uno de los laboratorios de la facultad. Mumuki proporcionó interactividad formativa de forma automática. Además, el profesor y los docentes asistentes recorrían el laboratorio respondiendo preguntas de los estudiantes sobre los ejercicios.

La segunda parte de la clase se desarrollaba en otra aula donde el profesor explicaba los conceptos trabajados anteriormente con Mumuki. También presentaba ejemplos de ejercicios resueltos y discutía los errores más comunes que se observaban en la primer parte de la clase. En la segunda parte de la clase, los estudiantes no utilizaron la computadora, motivándolos a programar en Haskell utilizando papel y lápiz.

Los estudiantes utilizaron el entorno cerrado Mumuki en clase, durante 30 horas. También usaron Mumuki fuera de los horarios de clase. El 85 % de los programas enviados por los estudiantes fueron realizados durante las horas de clase.

Experiencia CS2 en Universidad Tecnológica Nacional

En el caso de la Universidad Tecnológica Nacional de Buenos Aires, la experiencia se implementó en la materia Paradigmas de la Programación, correspondiente al segundo año de la carrera de Ingeniería en Sistemas. Los estudiantes cursan alrededor de 20 horas semanales. Por lo tanto, tienen la posibilidad en cuanto a tiempos y horarios de estudiar y trabajar.

El dictado de la materia estuvo a cargo de un profesor universitario, dos docentes asistentes y seis ayudantes alumnos, quienes dictaron 9 clases de programación funcional desde Marzo a Mayo del 2016. Se dictaba una clase por semana, la cual tenía una duración de 4 horas, comenzando a las 9 y finalizando a las 13.

En las materias anteriores los estudiantes comienzan a programar utilizando lenguajes imperativos como Pascal y C. Por lo tanto, es la primer experiencia con el paradigma funcional dentro de la carrera pero no es la primera experiencia de programación para ningún estudiante. 59 estudiantes comenzaron y asistieron al curso durante al menos dos clases. En el semestre anterior, se utilizaron Pascal y C para introducir estructuras de control y estructuras de datos básicas como listas, pilas y colas. La edad promedio de los estudiantes es de 22 años.

Mumuki fue utilizado en las clases para presentar nuevos conceptos y para poner en práctica los conceptos que se desarrollaron anteriormente. Cuando era necesario, el equipo docente detenía

la práctica con Mumuki y utilizando el pizarrón o proyectando algún ejercicio de Mumuki, se presentaban conceptos, discutían errores y compartían diferentes programas. Las explicaciones no duraban más de 10 minutos en promedio y nunca superaron el 10 % del tiempo total de la clase. No se les pidió a los estudiantes programar en papel, utilizaron Mumuki durante toda la clase. Por lo tanto, los estudiantes usaron en un total de 36 horas de Mumuki durante las clases. Los estudiantes también utilizaron Mumuki fuera de las clases, pero el 70 % de las programas fueron enviados en horarios correspondientes a las clases.

5.2.4. Resultados y análisis

En esta sección presentamos los resultados obtenidos a partir de las dos experiencias implementadas. Primero describimos las métricas utilizadas para evaluar el desempeño de los estudiantes en los dos cursos: proporción de aprobación, desaprobación y abandono. Luego presentamos los resultados obtenidos a partir de las respuestas de los estudiantes en base al Modelo de aceptación de tecnología (TAM)

Métricas de abandono, aprobación y desaprobación

Los resultados con respecto al desempeño del año 2016 que se observan las tablas 5.6 y 5.7 hacen referencia al primer año en el cual se utilizó en el dictado de las materias el entorno Mumuki. Los resultados de desempeño en los años anteriores a 2016[4] también son reportados. Cuando Mumuki no fue utilizado en las materias, los estudiantes utilizaban directamente compiladores como Hugs o Ghci para resolver o probar programas para los ejercicios de programación. Las diferencias con respecto al año 2016 y los anteriores consisten en el grupo de estudiantes y el entorno de programación utilizado. El equipo docente, los ejercicios utilizados y los tipos de exámenes fueron los mismos. Para poder evaluar y comparar el rendimiento de los estudiantes definimos las siguientes métricas:

Examen Aprobado: Número y porcentaje de estudiantes que aprobaron el examen en su primer intento.

Recuperatorio Aprobado: Número y porcentaje de estudiantes que aprobaron el examen la segunda vez que intentaron. La distancia temporal entre el examen y el recuperatorio fue de un mes.

Abandono: Número y porcentaje de estudiantes que se inscribieron y asistieron al menos a dos clases. Estuvieron ausentes o desaprobaron el primer examen, y estuvieron ausentes durante el examen recuperatorio.

Desaprobados: Número y porcentaje de estudiantes que desaprobaron el examen y el recuperatorio.

En ambas tablas se observa una disminución estadísticamente significativa en el porcentaje de abandonos en el año que se utilizó Mumuki. Para realizar este análisis utilizamos unpaired t-tests entre los años 2015 y 2016 para todas las métricas definidas anteriormente. Los valores para p y t se reportan en ambas tablas.

El porcentaje de abandono en la UTN en el año 2016 es de 14 % mientras que la misma fue de 28 % en 2015. El porcentaje de abandono en la UNC fue de 35 % en 2016 mientras que fue de 59 % en 2015 y de 58 % en 2014. Estos resultados nos dan evidencia en contra de la hipótesis

[4]El año 2014 no es reportado para UTN ya que no se contaba con esa información.

	2015		2016		t-test	
	n	%	n	%	p	t
Examen aprobado	15	33	22	37	.62	.49
Recuperatorio aprobado	3	6	8	14	.25	1.17
Abandono	13	28	8	14*	.05	2.01
Desaprobado	15	33	21	35	.75	.32
Total	46	100	59	100	-	-

Tabla 5.6: Cantidad y porcentaje de abandono, aprobado y desaprobado en CS2 en UTN en 2015 y 2016. Las diferencias significativas entre 2015 y 2016 están marcadas con *, se informan valores de p y t para resultados de unpaired t-test..

	2014		2015		2016		t-test	
	n	%	n	%	n	%	p	t
Examen aprobado	21	32	16	31	21	38	.43	.79
Recuperatorio aprobado	3	4	4	5	12	22*	.02	2.3
Abandono	38	58	34	59	19	35*	.01	2.6
Desaprobado	4	6	4	5	3	5	.75	.31
Total	66	100	58	100	55	100	-	-

Tabla 5.7: Cantidad y porcentaje de abandono, aprobado y desaprobado en CS1 en UNC en 2014, 2015 y 2016. Las diferencias significativas entre 2015 y 2016 están marcadas con *, se informan valores de p y t para resultados de unpaired t-test.

nula H_0 (null), en favor de la alternativa H_1 (alt): el porcentaje de abandono en un curso que usa Mumuki es significativamente menor que la de un curso que no lo hace. Esta hipótesis fue probada en dos contextos que son bastante diferentes. Según los resultados, el efecto parece ser más fuerte en la muestra correspondiente a CS1, pero también es significativa en CS2.

La Tabla 5.7 muestra que la diferencia entre 2015 y 2016 también es significativa para el porcentaje de Recuperatorio aprobado en la UNC. Los docentes del curso reportan que los estudiantes utilizan Mumuki como una herramienta para prepararse para el examen recuperatorio. Entre el examen y el recuperatorio, la materia sigue avanzando con nuevos contenidos, por lo que los estudiantes tienen que prepararse para el recuperatorio por su propia cuenta. En ese sentido, los estudiantes dijeron que Mumuki les da la oportunidad de estudiar a su propio ritmo y bajo sus necesidades. Profundizaremos en este punto en la siguiente sección sobre el Modelo de aceptación de tecnología.

Si comparamos el porcentaje de abandonos de nuestros estudios experimentales con los resultados de estudios realizados en otros países reportados en la Sección 5.2.1, podemos observar que son más altos. Watson reporta un promedio de 20 % de abandono [125]. Tanto la UNC como la UTN son universidades públicas, la cual implica que son totalmente gratuitas. El trabajo en simultáneo con el estudio y la falta de conocimientos en matemáticas pueden ser posibles causas.

Modelo de Aceptación de Tecnología (TAM)

Finalmente analizamos las percepciones de los estudiantes con respecto a Mumuki con el Modelo de Aceptación de Tecnología (TAM) [25]. TAM es una metodología para evaluar cuánto los usuarios aceptan una tecnología innovadora e intentan predecir cuánto la usaran en el futuro. El modelo sugiere que cuando los usuarios se enfrentan con una tecnología nueva, existen un

Afirmaciones evaluadas	Dimensión de TAM	CS1	CS2
1. Mumuki ayuda a mejorar mis habilidades para programar en Haskell	Mejora	6.24	6.32
2. Mumuki me permite controlar cuánto quiero o puedo aprender por día	Productividad	5.76	5.55
3. Mumuki me ayuda a resolver los ejercicios más rápido	Eficiencia	5.56	5.74
4. Mumuki proporciona ayuda ante las dificultades	Cooperación	5.52	4.53
5. Mumuki me ayuda a resolver los ejercicios correctamente	Efectividad	5.36	5.21
6. En general, Mumuki es útil para aprender a programar en Haskell	Utilidad	6.16	5.82
7. Aprender a usar Mumuki es fácil para mi	Familiaridad	6.20	6.74
8. Es fácil que Mumuki haga lo que yo quiero	Manejabilidad	4.56	5.24
9. La interacción con Mumuki es clara y entendible	Claridad	5.40	6.08
10. La interacción con Mumuki es flexible (e.j. yo decido qué ejercicio resolver)	Flexibilidad	6.28	6.45
11. Fue fácil para mi manejar Mumuki hábilmente	Competencia	6.00	6.37
12. En general, pienso que Mumuki es fácil de usar	Usabilidad	6.44	6.61

Tabla 5.8: Resultados de la aplicación del Modelo de aceptación tecnológica (TAM) a los cursos CS1 y CS2. Las primeras seis frases evalúan la utilidad y las restantes están relacionadas con la facilidad de uso.

conjunto de factores que influyen en su decisión sobre cómo y cuándo lo utilizarán. Las dimensiones utilizadas para medir el impacto de la tecnología en los usuarios son las de utilidad y facilidad de uso. La encuesta no era obligatoria y en total fue completada por 87 estudiantes en 2016. Les pedimos a los estudiantes que indicaran su nivel de acuerdo con las frases enumeradas en la Tabla 5.8 utilizando una escala Likert de 7 puntos (del 1 al 7). También les pedimos a los estudiantes que proporcionaran comentarios libres sobre las mejores y peores características de Mumuki. De acuerdo a TAM, las primeras 6 preguntas (del 1 al 6) evalúan la utilidad de Mumuki y las últimas 6 preguntas (del 7 al 12) evalúan la facilidad de uso. Se puede observar en la tabla que las preguntas están orientadas a evaluar diferentes aspectos de la utilidad (por ejemplo, la productividad y la eficiencia) y de la facilidad de uso (por ejemplo, claridad y flexibilidad).

Preguntamos explícitamente si planeaban usar Mumuki para aprender otros lenguajes de programación; El 95 % de los estudiantes respondieron de forma afirmativa. Entre las ventajas que mencionan al usar Mumuki, las más frecuentes son las siguientes: “Te dice cuándo su solución es incorrecta y puedo intentarlo muchas veces hasta que lo haga bien”, “La mejor parte es que tenés feedback inmediato”, “Puedo usarlo en cualquier momento y desde cualquier lugar a mi propio ritmo”, “Es particularmente útil cuando te perdés alguna clase”.

En general, los estudiantes están de acuerdo con las declaraciones propuestas por el modelo TAM, por lo tanto el modelo predecería que a los estudiantes les gustaría continuar utilizando Mumuki una vez finalizado el curso. En particular, ambos grupos de estudiantes estuvieron fuertemente de acuerdo con las afirmaciones “Mumuki ayuda a mejorar mis habilidades para programar en Haskell” (6.24 en CS1 y 6.32 en CS2) y “En general, pienso que Mumuki es fácil de usar” (6.44 en CS1 y 6.61 en CS2).

En cuanto a la utilidad, el valor más bajo encontrado es el valor promedio de los estudiantes de CS2 para “Mumuki proporciona ayuda ante las dificultades” en 4.53. Los estudiantes explicaron

que “cuando tenían un error de sintaxis, a veces no entendían la explicación dada por Mumuki”. Mumuki informa errores de sintaxis mostrando el mensaje del compilador.

En cuanto a la facilidad de uso, el menor valor promedio de los estudiantes de CS2 fue para “Es fácil que Mumuki haga lo que yo quiero”, en 4.56. Este valor es ligeramente positivo, el valor medio de la escala usada es 4. Los estudiantes explicaron que “fue frustrante recibir el mismo mensaje de error para diferentes programas enviados”. Mumuki no tiene cambios en los programas enviados cuando genera el feedback formativo. Tener en cuenta estos cambios y detectar mejor cuando un estudiante necesita ayuda es el tema del próximo capítulo.

5.2.5. Conclusiones preliminares

A partir de las experiencias desarrolladas y los resultados obtenidos, encontramos que en ambas universidades, la proporción de abandono es significativamente menor al utilizar Mumuki, en base a los datos correspondientes a años anteriores cuando el entorno no era utilizado. Es importante destacar que el uso del entorno Mumuki no es la única variable de este estudio. También cambian los grupos de estudiantes. Por lo tanto, futuras reproducciones de estudios similares serían deseables.

El modelo de aceptación de tecnología (TAM) nos permitió analizar percepciones de los estudiantes sobre el entorno que nos invitan a pensar que seguirán utilizando Mumuki una vez finalizado el curso. Una de las características mejor valoradas por los estudiantes es la interactividad formativa automática que ayuda a los estudiantes a corregir errores por su cuenta.

A continuación presentamos conclusiones preliminares en los estudios realizados con respecto al uso de entornos que presentan interactividad formativa para estudiantes principiantes en programación.

Permite a los estudiantes desenvolverse en un contexto controlado. Los estudiantes cuentan con ejercicios con un problema concreto a resolver. Además de contar con ejemplos y feedback formativo sobre sus errores para poder comprender el problema. Estos contextos tan controlados pueden ser un buen primer paso para aquellos programadores principiantes como muestra TAM. Contar con un entorno que brinde feedback formativo a cada solución enviada, puede dar cierta confianza a los estudiantes de tratar de resolver los ejercicios en sus propios tiempos. Esto puede impactar positivamente sobre la reducción de abandono. La interactividad formativa de Mumuki se puede mejorar. Como describimos en la sección anterior, los estudiantes se frustraban al recibir el mismo feedback formativo por parte de Mumuki a pesar de realizar cambios en sus programas. Mumuki podría mejorar su interactividad formativa si pudiese considerar el historial de programas que envía el estudiante para resolver el ejercicio planteado.

5.3. Conclusiones del capítulo

Las experiencias descriptas anteriormente se desarrollan en dos niveles educativos distintos: nivel universitario y nivel primario. En ambos casos, utilizamos el entorno cerrado Mumuki, con diferentes lenguajes de programación: Haskell en el nivel universitario y Gobstones en el nivel primario. A partir de los interrogantes planteados en el Capítulo 4 sobre el uso de entornos abiertos para la enseñanza de programación, nos enfocamos en un entorno cerrado con interactividad formativa para evaluar su impacto, en distintas métricas relacionadas a la adquisición del lenguaje: errores sintácticos, semánticos, resultados de exámenes, deserción y autopercepción. En las Secciones 5.2.5 y 5.1.5 describimos algunas conclusiones sobre el impacto del entorno cerrado Mumuki en cada uno de los niveles educativos en relación a los objetivos planteados.

Algo en común entre ambas experiencias en los niveles educativos es que las clases de los estudios descriptos anteriormente fueron dictadas por equipos docentes con formación académica de grado, posgrado y muchos años de experiencia docente en enseñanza de la programación. Teniendo en cuenta a docentes que no tienen formación específica como los descriptos en el Capítulo 2, ¿es posible detectar a los estudiantes que se encuentran con inconvenientes de aprendizaje o en riesgo de desertar?

Codecademy [22] es un entorno cerrado similar a Mumuki que se usa intensivamente en muchos países para aprender a programar en lenguajes como Python, Ruby, Java, entre otros. Por ejemplo fue elegido por la ciudad de Buenos Aires como solución a la falta de docentes capacitados. Los ejercicios se completan siguiendo un conjunto instrucciones que brinda el entorno. El usuario escribe el código requerido en el editor de texto. Codecademy es muy utilizado, pero como se describe en [111], una de sus principales limitaciones es que no puede identificar cuándo los estudiantes se han desmotivado o están teniendo dificultades con un ejercicio específico. Después de una cantidad fija de soluciones incorrectas, Codecademy le brinda la solución correcta al estudiante. Este tipo de interactividad no es formativa.

Como lo es Codecademy para nivel secundario e introducción a la universidad, Code.org [128] es la herramientas de elección a nivel mundial para enseñar programación en nivel primario. Code.org provee interactividad formativa en forma de "hints" que dicen al estudiante exactamente que bloque borrar o agregar para corregir sus errores.

A pesar de que la evaluación automática en este tipo de entornos sea una herramienta interesante para los docentes y estudiantes, Wang et al. en [124] plantean un tema preocupante. La habilidad de los docentes de poder observar el proceso de sus estudiantes parece perderse en la evaluación automática. Poder entender el proceso y progreso de un estudiante es invalorable. Si un docente puede tener conocimiento de cuando un estudiante está teniendo problemas para resolver un ejercicio específico podría rápidamente ayudar al estudiante a superar la situación. Codecademy al brindar la solución luego de una serie de intentos y Code.org con su feedback tan dirigido también pueden ser engañosos al momento de guiarnos únicamente con la evaluación automática. Cada vez son más utilizados este tipo de entornos en contextos educativos formales. La falta de docentes y el aumento del interés en estudiar programación, hacen que estos entornos tomen un lugar central en la enseñanza de programación. Pero, ¿qué tipo de programadores estamos formando? Se acostumbran a resolver ejercicios, en un contexto controlado y dirigido que les da la solución al primer indicio de frustración. El interés puntual de los estudiantes pasa a ser poder "pasar todos los niveles" de estos entornos gamificados . Las formas, eficiencia, estilo de código parecen quedar de lado. ¿Cómo podemos entonces sacar un provecho más profundo de este tipo de entornos ampliamente utilizados a nivel mundial?

Gracias a entornos cerrados como Mumuki, Codecademy, Code.org y otros [22] es posible recopilar grandes bases de datos de programas generados por los estudiantes en el proceso de aprendizaje. Tales conjuntos de datos son valiosos para realizar minería de datos educativos y analítica de aprendizaje [57]. Un área prometedora de investigación es el rastreo del conocimiento. El rastreo del conocimiento es la tarea de modelar el conocimiento del estudiante para poder predecir con precisión cómo se desempeñará en futuras interacciones. En particular, en base a un programa creado por un estudiante al momento de aprender a programar, una tarea de rastreo de conocimiento sería poder predecir cuando el estudiante estará listo para poder resolver un ejercicio de programación dado sin ayuda. Podemos pensar esta tarea como el nivel de fluidez que tiene un estudiante con respecto al lenguaje de programación específico que está aprendiendo.

En el Capítulo 6 presentamos modelos de aprendizaje automático que intentan medir de forma automática la fluidez de los estudiantes durante la adquisición de un lenguaje. Evaluaremos estos

modelos en dos grupos de estudiantes, uno que cuenta con docentes formados que los acompañan y otros en un contexto de curso online abierto y masivo (del ingles Massive Online Open Course MOOC).

Capítulo 6

Evaluación automática de la fluidez en un lenguaje de programación

En este capítulo, proponemos evaluar automáticamente programas creados por estudiantes principiantes. En base a un programa creado por un estudiante que está aprendiendo a programar, comparamos distintos métodos que predicen si el estudiante podrá resolver con éxito el ejercicio de programación sin ayuda. Esta tarea es desafiante por al menos dos motivos. Por un lado, hay mucha variabilidad en el tipo de errores que comete un principiante comparado a alguien con experiencia en desarrollo de software por lo cual las técnicas de detección de errores estándar no son suficientes. Por otro lado, como vimos en el capítulo anterior los programadores principiantes cometen muchos errores de sintaxis por lo cual no es posible asumir que los programas están respetando una gramática formal. Debido a esta gran variabilidad en los programas, este capítulo se basa en la intuición de que el código creado por estudiantes principiantes se parece más a un lenguaje natural que a un lenguaje formal.

El capítulo está organizado de la siguiente manera. En la Sección 6.1 describimos trabajos previos existentes relacionados a nuestra tarea. En la Sección 6.2 describimos los conjuntos de datos sobre los que utilizamos los experimentos que presentamos en este capítulo. Luego, en la Sección 6.3 presentamos cómo medimos el rendimiento de los expertos, así como los dos tipos de modelos de aprendizaje automático. Uno de estos modelos se basa en técnicas de procesamiento de lenguaje natural, mientras que los otros se basan en trabajos previos del área de educación. En la Sección 6.4 analizamos los resultados obtenidos por los modelos y describimos un ejemplo detallado. Luego, en la Sección 6.5 describimos trabajo previo existente en adquisición de un segundo idioma y técnicas computacionales para aprender automáticamente el significado contextual de las palabras de un lenguaje natural. Finalmente en la Sección 6.6 presentamos las conclusiones del capítulo.

6.1. Introducción

En entornos cerrados como Mumuki [12], Codecademy [22] y otros [22] es posible recopilar grandes bases de datos de programas generados por los estudiantes en el proceso de aprendizaje. Tales conjuntos de datos son útiles para realizar minería de datos educativos y analítica de aprendizaje [57]. Llamamos al conocimiento actual alcanzado en un lenguaje: la *fluidez*. En particular, en base a un intento de escritura de un programa en el que podemos detectar la fluidez en el lenguaje, la tarea que abordamos es predecir si el estudiante está listo para poder

resolverlo correctamente sin ayuda. A continuación describimos trabajo previo que aborda esta tarea para lenguajes basados en bloques y discutimos la diferencia de la tarea como nosotros la desarrollamos en esta tesis: en un lenguaje de programación más expresivo y basado en texto.

6.1.1. Evaluación de fluidez en un lenguaje de bloques

Code.org [128] organiza a nivel mundial la campaña "La hora de código". Code.org cuenta con un entorno cerrado donde los estudiantes aprenden programando al personaje principal para que pueda resolver laberintos como los que se muestran en la Figura 6.1. El estudiante programa usando el lenguaje de programación basado en bloques Blockly [42] que presentamos en los Capítulos 3 y 4 .

Figura 6.1: Interfaz del entorno Code.org. A la izquierda observamos el laberinto que el estudiante debe resolver. A la derecha, los bloques con los que el estudiante cuenta, y el editor de código con el programa creado por el estudiante.

Wang et al. en [124] exploran el uso de redes neuronales recurrentes (RNN) sobre los programas de bloques creados por los estudiantes en Code.org para tratar intentar predecir si el estudiante completará el ejercicio. Para entrenar sus modelos, Wang et al. [124] cuentan con más de un millón de programas enviados por cientos de miles de estudiantes para dos ejercicios. En la Figura 6.3 observamos la arquitectura de la red implementada. Cada programa enviado por un estudiante es modelado utilizando *program embeddings*. Piech et al. [97] proponen utilizar redes neuronales recurrentes para crear program embeddings para los programas creados por los estudiantes . Estas RNN toman como entrada el estado del programa, es decir el laberinto que se encuentra en la parte izquierda de la Imagen 6.1 antes de ejecutar cada instrucción del programa, y el laberinto que se obtiene después de ejecutar instrucción por instrucción del programa. Cada input de la RNN va tomando el estado del programa hasta que termina su ejecución para poder entrenar los program embeddings. Cada programa es representado con su árbol de sintaxis abstracta (del inglés abstract syntax tree, AST). Como Code.org utiliza el lenguaje de bloques Blockly [42] todos los programas enviados son sintácticamente correctos por construcción como describimos en los Capítulos 3 y 4. Para entrenar los program embeddings no se utilizan los AST completos. Utilizan subárboles, en lugar de utilizar el AST completo. De esta forma los embeddings no sólo son entrenados en base a la comienzo del programa sino también en base a diferentes partes que conforman al mismo. Con esta implementación consiguen información de cómo el programa fue implementado y no sólo de su funcionamiento general. En la Figura 6.2 observamos la RNN para entrenar los program embeddings.

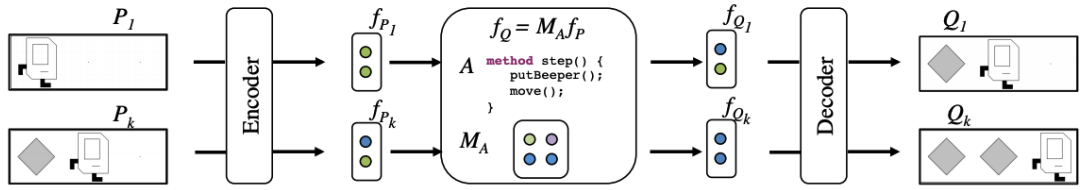


Figura 6.2: RNN para entrenar los program embeddings definido por Piech et al. Figura extraída de [97].

Por lo tanto para entrenar los program embeddings toman un subárbol del AST, información del estado en el cual se encuentra el subárbol antes de ser ejecutado (precondición) y observan el estado después de ejecutar el subárbol creado por el estudiante (postcondición). De esta manera pueden relacionar cada programa con una precondición y una postcondición. Como podemos ver en la Figura 6.1 el conjunto de estados (posiciones en las cuáles puede encontrarse el personaje) es finito, por lo que, a pesar de poder generar infinitos programas con el lenguaje basado en bloques, cada programa creado por el estudiante se representa con un conjunto finito de program embeddings. En el laberinto de la Figura 6.1 solo hay 7 posiciones posibles a las que puede llegar el personaje, es decir solo 7 configuraciones posibles para el laberinto.

Wang et al. utilizan una arquitectura Long Short Term Memory (LSTM). La red toma como entrada la trayectoria completa de un estudiante para resolver un ejercicio. La trayectoria de un estudiante es la secuencia de programas que el estudiante envió para resolver un ejercicio. Cada programa es representado con su árbol de sintaxis abstracta. Luego, cada AST se convierte en un program embedding como describimos anteriormente. La secuencia de program embeddings es la entrada de la RNN como observamos en la Figura 6.3. El modelo neuronal definido necesita de la trayectoria completa del estudiante para poder realizar la predicción de la tarea. Esta implementación no podría ser utilizada para el problema de *cold start*, el cual hace referencia a los estudiantes que están utilizando el entorno por primera vez y sobre el cual no hay información previa. En la mayoría de los contextos educativos (es decir, aulas), las tareas definidas no tienen suficientes datos históricos para el aprendizaje supervisado. No tenemos medio millón de envíos por ejercicio. ¿Qué pasa con los estudiantes que acaban de comenzar y no tenemos datos históricos sobre ellos? Podría decirse que estos son los estudiantes que necesitan mayor seguimiento. *Zero shot learning* tiene como objetivo identificar cuándo un estudiante tiene complicaciones para resolver un ejercicio mirando un sólo programa creado por el estudiante. Por lo tanto *Zero shot learning* tiene el objetivo de tener un rendimiento aceptable en el caso de *cold start*, es decir cuando no se cuenta con información del estudiante.

Wu et al. en [132] intentan brindar una solución para el problema de cold start implementando Zero Shot Learning [89, 113] para la adquisición de un lenguaje de programación, también basado en el lenguaje Blockly [42]. Definen como tarea generar feedback formativo de forma automática para los programas enviados por nuevos estudiantes. Para poder implementar su modelo también necesitan de programas que sean sintácticamente correctos. Al utilizar los programas enviados por estudiantes para resolver 8 ejercicios con lenguajes basados en bloque de Code.org, todos los programas son sintácticamente correctos por construcción. Cada programa es representado como una secuencia de tokens. Entrenan una gramática probabilística libre de contexto (PCFG siglas del inglés probabilistic context-free grammar), para poder resolver la tarea. A pesar de que con los lenguajes basados en bloques como el que utiliza Code.org se pueden generar infinitos programas, el conjunto de programas enviados por los estudiantes es mucho más pequeño que el generado por estudiantes que están utilizando lenguajes basados en texto. En la Figura 6.4

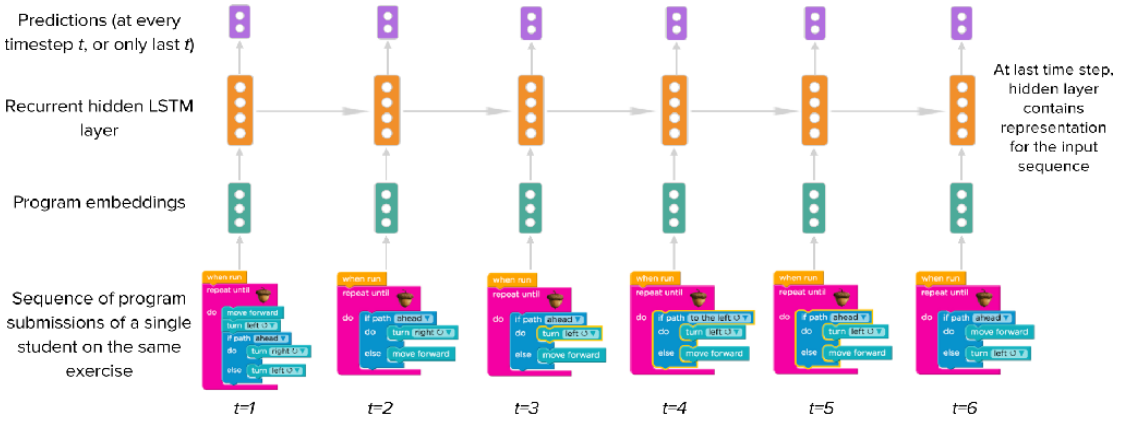


Figura 6.3: LSTM secuencial simplificada implementada por Wang et al. El modelo toma la trayectoria completa de programas enviados por un estudiante para intentar predecir si el estudiante tiene o no problemas para resolver el ejercicio. Figura extraída de [124]

podemos observar que el modelo definido por Wu et al. logra cubrir de buena forma el conjunto de datos generado por Code.org pero no sucede lo mismo con lenguajes de texto.

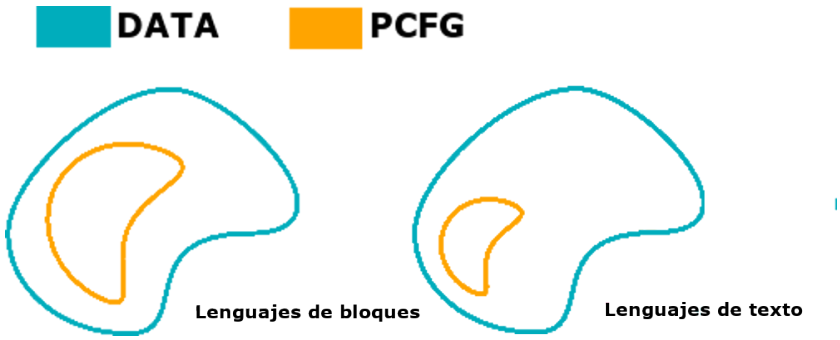


Figura 6.4: Conjuntos de programas basados en bloques y programas basados en texto generados por los estudiantes que el modelo PCFG puede cubrir. Figura extraída de [97]

6.1.2. Evaluación de fluidez en un lenguaje de texto

El objetivo de este capítulo es usar zero shot learning para predecir la fluidez en un lenguaje de programación basado en texto. El trabajo previo asume que los programas producidos por los estudiantes compilan. En nuestro caso, casi la mitad de los programas escritos por estudiantes no compilan, por lo que, no podemos partir de esta suposición. El lenguaje basado en texto sobre el que proponemos trabajar es Haskell, que es un lenguaje que incluye nombre de variables y funciones que el estudiante puede definir. Al no contar con la posibilidad de abstraernos de estos nombres usando árboles de sintaxis abstracta, se aumenta considerablemente la variabilidad de los datos sobre los que entrenaremos nuestros modelos.

Proponemos construir modelos cuya entrada sea directamente el texto tokenizado de un programa creado por un estudiante. El programa puede contar con todo tipo de errores incluyendo también errores de sintaxis. Los modelos son entrenados con programas creados por diferentes estudiantes para resolver el mismo conjunto de ejercicios. Los modelos toman como entrada el texto plano de un programa creado por un estudiante. Para construir nuestros modelos, comparamos técnicas de procesamiento del lenguaje natural (PLN) capaces de modelar dependencias

largas entre palabras conocidas en inglés como Long short term memory networks (LSTM) y word embeddings [110]. Comparamos dichos modelos con métodos más tradicionales en el área de minado de datos educativos basados en la ingeniería de características sobre las trayectorias de los estudiantes. Hay trabajo previo realizado utilizando técnicas de PLN para mejorar la eficiencia en de los entornos integrados de desarrollo usados por programadores profesionales [56], pero no así para procesar código de principiantes.

El conjunto de datos utilizados fue obtenido del entorno cerrado Mumuki que describimos en el Capítulo 5. Tenemos dos conjuntos de datos diferentes. Un conjunto de datos fue generado por 75 estudiantes que cursaron la materia Introducción a los Algoritmos en la Facultad de Matemática, Astronomía, Física y Computación de la Universidad Nacional de Córdoba. El otro fue generado por 4000 estudiantes que de forma autodidacta utilizaron el entorno Mumuki para aprender a programar.

Identificamos a la tarea de predecir si un estudiante va a abandonar un ejercicio de programación como una medida indirecta de la fluidez que tiene el mismo al momento de programar la solución del ejercicio.

6.2. Diseño de estudio

Para este estudio utilizamos dos conjuntos de datos diferentes recolectados dentro del entorno Mumuki: un conjunto de datos se obtuvo en un contexto de curso masivo online autodidacta (a distancia) al que llamaremos *mumuki.io*, mientras que el otro conjunto de datos se obtuvo a partir de un contexto presencial dentro de la Universidad de Córdoba (UNC) al que llamaremos *introalg*. Vamos a utilizar el término *programa* como un fragmento de código que intenta resolver un ejercicio de programación y puede ser incorrecto o estar incompleto. Cada vez que un estudiante presiona el botón enviar en Mumuki se registra un nuevo programa, por lo tanto, en nuestro conjunto de datos existen diversas versiones del mismo programa para el mismo estudiante y el mismo ejercicio. En la Figura 6.5 observamos la interfaz con la cual cuenta el docente para realizar un seguimiento de los programas enviados por un estudiante para resolver un ejercicio particular. En este caso, el estudiante envió 10 programas para resolver el ejercicio.

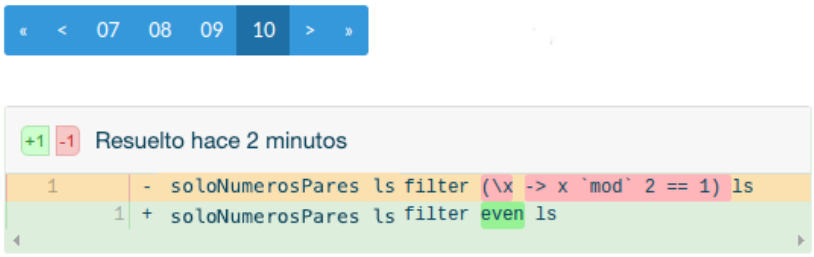


Figura 6.5: Herramienta de Mumuki con la que cuenta el docente para hacer un seguimiento de los diferentes programas enviados por un estudiante para resolver un ejercicio particular. La interfaz muestra al docente el programa actual y el anterior enviado por el estudiante.

Los programas del conjunto de datos *introalg* son enviados por estudiantes que cursaron la materia Introducción a los Algoritmos en el segundo semestre de 2018. Esta es la primera materia de programación que tienen los estudiantes en la Licenciatura en Ciencias de la Computación en la UNC. Los estudiantes cursaron 8 horas semanales, las cuales se dividieron en dos días: martes y jueves de 9 am a 1pm. Un docente adjunto, un docente asistente y un ayudante alumno estuvieron al frente del dictado de la materia. Los estudiantes del conjunto de datos *mumuki.io*

fueron personas que están aprendiendo a programar de forma autodidacta utilizando el entorno cerrado y las guías de ejercicios gratuitas de Mumuki. Los usuarios de mumuki.io no forman parte del sistema educativo formal. Estamos interesados en comparar nuestra propuesta sobre estos dos conjuntos de datos ya que se generaron en contextos educativos diferentes. El lenguaje de programación de los programas de ambos conjuntos de datos es Haskell y los ejercicios de programación que consideramos son aquellos que están presentes en ambos conjuntos de datos.

En la Tabla 6.1 presentamos la distribución de las programas enviados por los estudiantes en base a la evaluación automática formal que realiza Mumuki descripta en el Capítulo 5. Recordemos que el rojo oscuro indica que el programa no es sintácticamente correcto, por lo tanto no es posible su ejecución. El rojo claro indica que el programa es sintácticamente correcto pero algún caso de test no es correcto. Un programa se etiqueta como verde cuando todos los casos de test son correctos[1].

	introalg		**mumuki.io**	
Estado de la solución	#	%	#	%
Programas con errores de sintaxis	7457	38.5	69249	29.3
Programas con errores de caso de test	7855	40.5	86525	36.7
Programas correctos	4060	21.0	79927	34.0
Total de programas enviados	19372	100.0	235701	100.0
Total de estudiantes	75		3915	

Tabla 6.1: Distribución de la clasificación de los programas enviados por los estudiantes en base a la evaluación automática de Mumuki.

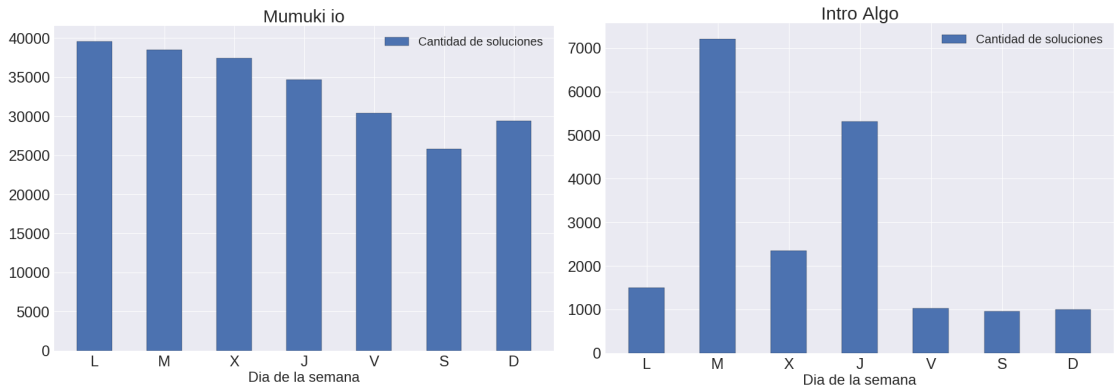
En la tabla podemos observar que el conjunto de datos introalg es diez veces más pequeño que el conjunto de datos mumuki.io. Además introalg tiene muchos menos estudiantes, pero en promedio envían más programas. Mientras que un estudiante de introalg envía en promedio 258 programas, un estudiante de mumuki.io envía 60.

En la Tabla 6.1 observamos que la proporción de programas evaluados con el color rojo oscuro en introalg es mayor que la proporción de programas evaluados como rojo oscuro en mumuki.io. El conjunto de datos introalg contiene una proporción mayor de errores de sintaxis, lo cual puede estar relacionado al hecho de que es el primer curso de programación de la carrera, por lo tanto, el primer lenguaje de programación que están aprendiendo los estudiantes. No tenemos acceso al perfil de los estudiantes que generaron el conjunto de datos mumuki.io, es posible que hayan tenido experiencia previa en Haskell o en otros lenguajes de programación.

Para conocer el comportamiento de los estudiantes que utilizan el entorno en dos contextos diferentes, analizamos la cantidad de programas enviados por día y el horario en el cual fueron enviados. En la Figura 6.6 observamos la distribución de programas enviados por día para ambos conjuntos de datos. En el conjunto de datos mumuki.io la distribución de programas enviados por día es pareja, a diferencia de introalg, donde la mayoría de las programas se envían los días que se cursa la materia (martes y jueves).

Al analizar la distribución por hora, observamos ver en la Figura 6.7b que en la franja horaria de 9 a 13, es la franja donde se concentra la mayor cantidad de programas enviados para introalg. Estos fueron los días y el horario en los cuales se dictaba la materia, podemos calcular que más del 40 % de los programas fueron enviados en horario de clase. En el caso de mumuki.io la utilización del sistema crece considerablemente luego de las 19 horas de acuerdo a la Figura 6.7a. Esto puede

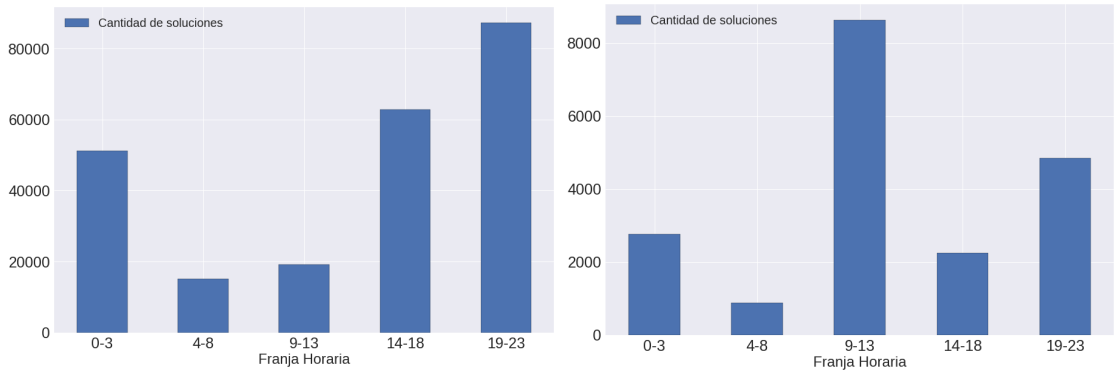
[1] Las programas enviados evaluados con el color amarillo que se describen en el Capítulo 5 fueron consideradas como verdes para este análisis ya que la cantidad con las que contamos no es significativa.



(a) (b)

Figura 6.6: Distribución de programas enviados por día para los conjuntos de datos. En la Figura 6.6a y 6.6b se informan la distribución para los conjuntos de datos mumuki.io e introalg respectivamente.

implicar que la mayor parte de los estudiantes de mumuki.io trabajan y usan Mumuki luego del horario laboral.



(a) (b)

Figura 6.7: Distribución de programas enviados por franja horaria para los conjuntos de datos. En la Figura 6.7a y 6.7b se informan la distribución para los conjuntos de datos mumuki.io e introalg respectivamente.

6.3. Metodología

En esta sección, primero definimos la tarea de evaluación automática de fluidez que abordamos como un problema de clasificación. Luego, en base a trabajo previo, realizamos ingeniería de características para la tarea definida. Además, presentamos la arquitectura de la LSTM que proponemos para la tarea definida. Finalmente, describimos cómo medimos el desempeño de docentes humanos en la tarea que proponemos automatizar.

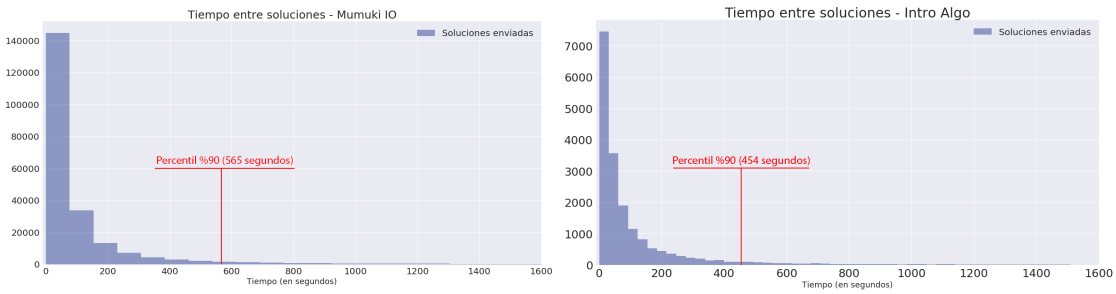
6.3.1. Definición de tareas y baselines

Formulamos nuestra tarea como una clasificación binaria. Específicamente, la tarea en la que nos enfocamos la definimos de la siguiente manera.

Tarea: Dado un programa incorrecto creado por un estudiante para resolver un ejercicio de programación específico, predecir si el estudiante podrá resolver el ejercicio por sí mismo en un futuro cercano o abandonará el ejercicio. Formulamos esta tarea como un problema de clasificación binaria entre las clases abandono y éxito.

A continuación definimos con mayor precisión el significado de "en un futuro cercano" introduciendo el concepto de sesión y "abandonar" en base a los errores previamente definidos.

Para un estudiante y un ejercicio concreto, definimos *sesión* a una secuencia de programas enviados dentro de un marco de tiempo en el que el tiempo de inactividad no supera un cierto umbral. Para definir el umbral por conjunto de datos, calculamos la distancia en segundos entre los programas consecutivos enviados por el estudiante. A medida que pasa el tiempo, la probabilidad de que un estudiante envíe un nuevo programa para el mismo ejercicio, disminuye. Definimos el umbral empíricamente como el tiempo transcurrido entre 2 programas enviados que cubre el 90 % de las programas de nuestros conjuntos de datos. Esto corresponde aproximadamente a 8 minutos para los dos conjuntos de datos. Es decir, después de más de 8 minutos de inactividad, consideramos que el envío de una nuevo programa corresponde a una nueva sesión. En la Figura 6.8b y la Figura 6.8a observamos la distribución del tiempo entre dos programas para introalg y mumuki.io respectivamente. Dado que la sesión se define con respecto a un ejercicio dado, consideramos que si el estudiante cambia a otro ejercicio, la sesión en el ejercicio anterior finaliza y comienza una nueva sesión.



(a) (b)

Figura 6.8: Distribución de la distancia en segundos entre dos programas enviados de forma consecutiva para los conjuntos de datos. En la Figura 6.6a y 6.6b se informan la distribución para los conjuntos de datos mumuki.io e introalg respectivamente.

Para un estudiante y un ejercicio concreto, un programa enviado se considera *abandono* si pertenece a una sesión cuya último programa enviado contiene errores de sintaxis o de casos de test, es decir cuando Mumuki lo evalúa el programa enviado como rojo claro o rojo oscuro. Intuitivamente, decimos que el estudiante abandona un ejercicio si abandona el mismo sin llegar a un programa correcto. Definimos un programa correcto como un programa que no tiene errores de sintaxis ni de casos de test. Por el contrario, un programa se clasifica como *éxito* si pertenece a una sesión cuyo último envío es un programa correcto. Usando estas definiciones anotamos de forma automática todas los programas enviados por los estudiantes, clasificando todos los programas pertenecientes a una sesión con el mismo valor que el último programa enviado en la sesión. Intuitivamente, anotamos automáticamente los programas mirando hacia al futuro en los datos de entrenamiento de nuestro modelo. En los conjuntos de datos todos los programas están etiquetados con el tiempo de envío lo que permite ordenarlos de forma temporal.

Utilizamos F_1 como métrica de evaluación. La métrica F1 es el promedio armónico de la precisión y recall, donde F1 alcanza su mejor valor en 1 y el peor en 0. Para asegurarnos de que

la performance del clasificador era independiente de la división realizada del conjunto de datos, utilizamos validación cruzada 5 veces para series temporales y series aleatorias [62].

Proponemos dos baselines sencillos para la tarea. Primero entrenamos un clasificador aleatorio estratificado. El otro baseline que construimos tiene como característica principal el contenido del programa enviado por el estudiante. Para poder utilizar el contenido de la solución que está escrita en Haskell, necesitamos tener una representación de la misma. Para ello representamos a los programos enviados mediante un Bag of Words (BoW). Cada programa enviado será transformado en un conjunto de tokens. La desventaja de este modelo, es que se pierde el orden original de los tokens. Implementamos estos clasificadores simples para nuestros baselines utilizando la regresión logística de scikit-learn [96], junto con funciones de pérdida ponderada para dar cuenta de las clases desbalanceadas dentro del conjunto de datos. En la Tabla 6.2 observamos el desempeño de los baselines definidos sobre series temporales aplicando validación cruzada 5 veces.

	F_1	
	introalg	**mumuki.io**
Muestreo aleatorio estratificado	0.50	0.57
Bag of Words	0.62	0.60

Tabla 6.2: Desempeño de los baselines propuestos utilizando validación cruzada 5 veces sobre series temporales.

En la próxima sección describimos cómo enriquecer el mejor baseline con diferentes características basadas en trabajo previo relacionado a la predicción de abandonos en lenguajes basados a texto.

6.3.2. Ingeniería de características sobre estudiantes y ejercicios

Luego de haber anotado ambos conjunto de datos, decidimos definir características que modelan el nivel de fluidez de un estudiante al aprender un lenguaje. Para ello consideraremos características de dos dimensiones: características del estudiante y características del ejercicio. En esta sección proponemos cómo representar ambas dimensiones extrayendo información automáticamente a partir de los conjuntos de datos introalg y mumuki.io. Describimos algunas de las características que implementamos. En el Apéndice B describimos y ejemplificamos cada una de las características que definimos.

Según Papert y Turkle [119] podemos clasificar a los estudiantes en dos tipos, *tinkerer* y *planner*. Esta clasificación depende del comportamiento de los estudiantes cuando intentan resolver ejercicios de programación. Los estudiantes clasificados como *tinkerers* realizan muchos cambios en los programas a través de prueba y error para crear una solución final válida. Los estudiantes clasificados como *planners* identifican un camino de acción y luego lo implementan con el objetivo de llegar a la solución que consideran válida.

Considerando la clasificación de Papert y Turkle, y siguiendo el trabajo realizado por Blikstein et al. en [16] definimos características que intentan modelar el tipo de estudiante y su comportamiento. Consideramos aspectos como el número de programas enviados por un estudiante para intentar resolver un ejercicio, la frecuencia con la que envía los programas, la cantidad de cambio en el código entre programas consecutivos, entre otros. Además de la dimensión de estudiante analizamos la dimensión de los ejercicios. Es decir, ver las particularidades de cada uno de ellos, por ejemplo el nivel de dificultad. A continuación se realiza una descripción de cada una de ambas dimensiones en profundidad.

Dimensión estudiante

Para poder capturar el comportamiento de los estudiantes, se tienen en cuenta 3 niveles que representan su comportamiento al momento de programar, nivel de experiencia, nivel de abandono y nivel de insistencia. El nivel de experiencia intenta capturar la experiencia en programación de un estudiante teniendo en cuenta los programas enviados para resolver los ejercicios de programación. Para ello definimos diferentes características que intentan capturar el nivel de experiencia del estudiante. Una de las características que definimos para intentar capturar el nivel de experiencia de los estudiantes es el promedio con aplazos, la cual definimos a continuación.

Promedio con aplazos (PCA), para representar el nivel de prueba y error de un estudiante, definimos la característica PCA. Definimos promedio con aplazos como el total de ejercicios que intento resolver un estudiante sobre la cantidad total de programas enviados para los mismos. Un ejercicio se define como intentado por el estudiante, si el estudiante envió al menos un programa para intentar resolver ese ejercicio.

El nivel de abandono tiene como objetivo representar si un estudiante abandona frecuentemente los ejercicios. Además intenta modelar bajo qué condiciones decide abandonar un ejercicio. Para ello definimos la característica proporción de abandonos.

Proporción de programas abandonados (PA), representamos el historial de abandono de un estudiante con la característica PA. PA es el total de los programas anotados como abandono para ese estudiante sobre el total de programas enviados por el estudiante.

A través del nivel de insistencia queremos capturar la intensidad del estudiante al momento de enviar programas al no poder resolver un ejercicio puntual. Describimos dos características para intentar capturar el nivel de insistencia de un estudiante: promedio de tiempo transcurrido entre soluciones consecutivas y promedio de distancia de Levenshtein [81].

Promedio de tiempo transcurrido (PTT), con el objetivo de medir con qué frecuencia el estudiante envía sus programas dentro de una sesión, calcularemos el promedio de tiempo transcurrido entre programas consecutivos (PTT), dentro de todas las sesiones llevadas a cabo por el estudiante, sin considerar el tiempo de inactividad, es decir el tiempo entre sesiones.

Promedio distancia de Levenshtein (PDL), intentando capturar que tanto el estudiante diseña el programa antes enviarlo, calculamos el promedio distancia de Levenshtein [81] entre el contenido de los programas sobre un mismo ejercicio, para todas las sesiones realizadas por el estudiante.

Dimensión ejercicio

En esta sección buscamos capturar características propias de los ejercicios que permitan generar información relevante para poder evaluar la dificultad del ejercicio. En este nivel se intenta representar que tan difícil es cada uno de los ejercicios a partir del desempeño de los estudiantes en los mismos. Para ello definimos la característica abandonos por estudiante.

Abandonos por estudiante (APE), una forma de ver la dificultad del ejercicio puede ser viendo cuántos programas envían los estudiantes antes de abandonarlo. Calculamos la proporción entre el número de estudiantes que abandonaron el ejercicio sobre la cantidad de programas enviados anotados como abandono. Esta característica intenta modelar cuantas soluciones en promedio envía un estudiante hasta que decide abandonarlo.

Definimos dos dimensiones que son de interés para explorar. Dentro de cada una de esas dimensiones definimos diferentes características las cuales pueden ayudar a modelar el trayecto de un estudiante. En la próxima sección describimos los modelos neuronales implementados para intentar resolver la tarea propuesta.

6.3.3. Técnicas secuenciales de procesamiento del lenguaje natural

Como argumentamos en la introducción, creemos que el procesamiento de programas usando word embeddings y técnicas secuenciales de procesamiento de lenguaje natural en lugar de modelos de bag of words es más adecuado para la tarea en cuestión. En la Sección 6.5 explicamos en profundidad las intuiciones sobre lo que son word embeddings y porque pensamos que funcionarían en este caso. Intuitivamente un bag of words pierde el orden de las palabras y una LSTM con word embeddings no lo pierde. También los word embeddings tratan de capturar la semántica, es decir el significado de una palabra dado su contexto.

Para probar esto, entrenamos una red neuronal recurrente simple (RNN) la cual podemos observar en la Figura 6.9 basada en redes de Memoria de Corto Plazo (LSTM). Usamos *fastText* [17] para construir word embeddings para todas las palabras del vocabulario. FastText calcula representaciones de vectores utilizando una red neuronal densa de dos capas que puede entrenarse sin supervisión en un corpus grande. En particular, usamos el modelo *skip-gram* [79] donde los embeddings de un token objetivo se utilizan para predecir embeddings de tokens contextualizados dentro de un tamaño de ventana fijo.

Cada uno de los programas será tokenizado con un vocabulario de 35000 tokens. Cada uno de los programas enviados por los estudiantes, luego de ser tokenizados, si tiene menos de 100 tokens, sus vectores son rellenados con 0 de modo que todos los vectores sean de dimensión 100 (padding). De esta manera cubrimos el 99,9 % percentil del tamaño del vocabulario y de la longitud de los programas. Exploramos diferentes tamaños de densidad para los vectores de embeddings para la semántica de palabras y encontramos que el vector de 256 dimensiones obtiene el mejor rendimiento para ambos conjuntos de datos. A través del ajuste estándar de hiperparámetros, encontramos que una LSTM con 100 unidades ocultas y una capa de abandono logran el mejor rendimiento en el conjunto de datos de desarrollo (dev set). El programa codificado de esta manera se alimenta a una capa densamente conectada con una función de activación sigmoidea que realiza la clasificación binaria.

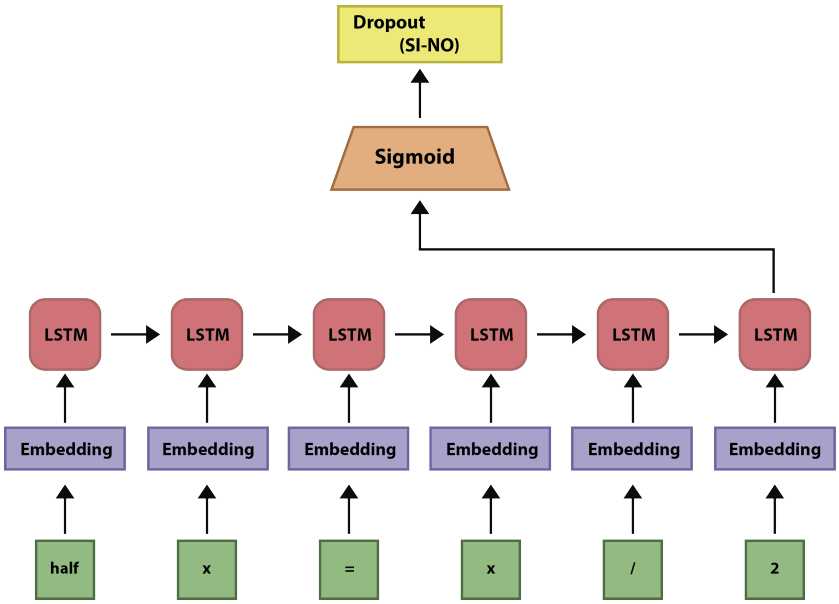


Figura 6.9: Diagrama simple de la LSTM implementada.

Entrenamos este modelo durante un máximo de 25 épocas, optimizando la pérdida de entropía cruzada binaria. Mantenemos el modelo que funciona mejor en el conjunto de desarrollo (dev

set). Logramos los mejores resultados después de 20 épocas. Utilizamos `Keras` [24] para ejecutar nuestros experimentos de aprendizaje profundo.

6.3.4. Rendimiento humano

Realizamos un experimento para medir el desempeño humano en la tarea definida. Esto ayuda a contextualizar el rendimiento de los modelos. Pedimos a dos docentes evaluar un total de 40 programas enviados. Elegimos 5 programas al azar para 8 ejercicios diferentes. Ambos evaluadores han sido docentes en el curso de Introducción a los Algoritmos durante varios años. Los docentes tienen que decidir si cada uno de los programas enviados se clasifica en abandono o éxito, basándose únicamente en el programa de la solución. Es decir, que los docentes no contaban con información sobre el estudiante ni su trayectoria al momento de realizar la clasificación. El rendimiento humano se evaluó reproduciendo un contexto de zero shot learning.

6.4. Resultados

En esta sección presentaremos los resultados de los modelos automáticos diseñados para la tarea de predicción propuesta para evaluar el nivel de fluidez de un estudiante al aprender a programar. Primero, en 6.4.1 presentamos y analizamos los resultados obtenidos por las modelos generados a partir de las características definidas y los modelos neuronales. Segundo, en 6.4.2 observamos el comportamiento de la red neuronal en cuanto a una secuencia de programas que intentan resolver un ejercicio puntual. En 6.4.3 reportamos los tiempos de entrenamiento y predicción para los modelos implementados. Luego en 6.4.4 discutimos sobre las limitaciones y fortalezas de los experimentos y modelos implementados. Finalmente en 6.4.5 describimos el impacto que pueden tener los modelos implementados en aulas masivas y heterogéneas.

6.4.1. Análisis cuantitativo

En la Tabla 6.3 reportamos los valores ponderados de F_1 en los conjuntos de prueba para los conjuntos de datos introalg y mumuki.io. Para cada conjunto de datos implementamos dos experimentos para entrenar nuestros modelos. En el primero de los experimentos tomamos con una distribución aleatoria el conjunto de entrenamiento para ambos conjuntos de datos. Los resultados reportados en la tabla fueron generados como promedio de realizar validación cruzada con diez iteraciones. Para el segundo de los experimentos, ordenamos los programas de ambos conjuntos de datos temporalmente. Los resultados reportados son el promedio de realizar validación cruzada con 5 iteraciones.

Las columnas están organizadas en dos secciones. Por un lado aplicando una metodología de partición de los datos usando una estrategia random para dividir los conjuntos de entrenamiento y test. La segunda metodología se basa en series temporales. Es importante notar que los resultados en los que se hace una división random son muchos más altos que cuando se hace la partición por series temporales. La partición en series temporales organiza el conjunto de datos de forma temporal. El conjunto de entrenamiento para series temporales se encuentra en el pasado, mientras que el conjunto de test en el futuro. El uso de una división random de datos para entrenamiento y test no es apropiado para el tipo de conjunto de datos con el que estamos trabajando, ya que es una tarea de edición y reenvío. Eso significa que los programas enviados del pasado y del futuro pueden tener muchos elementos en común. Si vemos los programas enviados del futuro en el conjunto de entrenamiento, le da una ventaja injusta en nuestro modo de

Modelo	F_1 distribución aleatoria		F_1 serie temporal	
	introalg	**mumuki.io**	**introalg**	**mumuki.io**
Baselines				
Muestreo aleatorio estratificado	0.51	0.57	0.50	0.57
Bag of Words (B)	0.59	0.69	0.62	0.60
Exploración de características utilizando RL				
B + PCA	0.62	0.68	0.63	0.65
B + PTT	0.60	0.64	0.64	0.60
B + PDL	0.61	0.63	0.65	0.60
B + APE	0.64	0.63	0.65	0.68
B + PA	0.65	0.76	0.63	0.61
B + APE + PA	0.67	0.76	0.63	0.68
Rendimiento humano				
Docente senior	–	–	0.64	–
Docente junior	–	–	0.58	–
Modelo RNN con Word Embeddings				
RNN con estudiante conocido	0.86	0.84	–	–
RNN con estudiante nuevo (cold start)	0.65	0.70	0.67	0.70

Tabla 6.3: Resultados comparando baselines, exploración de características usando Regresión logística, rendimiento humano y modelo RNN simple y desempeño de estudiantes con cold start que usan la RNN.

evaluación. Por lo tanto es indispensable usar series temporales para realmente predecir el comportamiento que tendría un sistema de este tipo si se pusiera en producción dentro del entorno Mumuki. Los resultados reales que se reportan son entonces los que pertenecen a la columna de series temporales. Hacemos la comparación con la división random porque hay bastante trabajo previo en el área que se evalúa usando esta técnica [132]. Sin embargo, como argumentamos anteriormente no es una técnica apropiada para este tipo de datos.

Organizamos las filas en 4 partes. En la primer parte de la tabla podemos observar los dos baselines utilizados. La primer fila reporta los resultados obtenidos por el clasificador dummy basado en un muestreo estratificado[2]. También mostramos el desempeño de un modelo de regresión logística entrenado con el contenido del programa escrito por el estudiante con un modelo simple de bag of words.

En la segunda parte de la tabla describimos la exploración de características utilizando regresión logística motivado por lo descripto en la Sección 6.3.2. La regresión logística se usa con frecuencia como baseline para comparar redes neuronales recurrentes profundas, ya que puede verse como una red neuronal poco profunda [48]. Como observamos en la tabla, para el conjunto de datos mumuki.io en el caso del conjunto de entrenamiento con distribución aleatoria, el desempeño del baseline bag of words es mejor que la mayoría de las combinaciones del mismo con características. La única característica que mejora el desempeño del baseline es proporción de abandonos. No sucede lo mismo para el caso de introalg para las características y el desempeño del baseline bag of words para la distribución aleatoria. Cada característica definida mejora el desempeño del baseline bag of words.

En la tercer parte de la tabla, informamos el desempeño de dos docentes experimentados que intentan predecir el abandono leyendo el programa enviado. La docente senior tiene 10 años de

[2]El muestreo estratificado es implementado utilizando la clase dummy de scikit-learn.

experiencia en la enseñanza de programación mientras que el docente tiene 2 años de experiencia. Los docentes no sabían quiénes eran los estudiantes. El acuerdo entre los docentes al predecir como abandono / éxito las 40 programas enviados por los estudiantes en base al Coeficiente kappa de Cohen fue del 50 %. El rendimiento humano supera al clasificador dummy, pero no es lo suficientemente bueno como algunas combinaciones de características o el modelo neuronal. La tarea de anotación tomó alrededor de 90 minutos por docente, mientras que los modelos lineales o la red neuronal completan la tarea en menos de milisegundos para cada ejemplo.

La cuarta parte de la tabla describe el desempeño del modelo neuronal para dos casos. En el primer caso, el modelo tiene acceso a los programas anteriores del estudiante. En el segundo caso, el modelo está haciendo predicciones para los programas de estudiantes (en el conjunto de test) que nunca ha visto (en el conjunto de entrenamiento). Todos los demás modelos de la tabla (excepto los docentes) tienen acceso a datos previos del mismo estudiante sobre el que están haciendo las predicciones para el caso de las series aleatorias. En el caso de las series temporales, al respetar el orden temporal de cada programa enviado, en el conjunto de test la red se encuentre con programas enviados en un futuro con respecto al conjunto de entrenamiento. Eso significa, que muchos de los programas enviados que se encuentran en el conjunto de test pertenecen a estudiantes cold start. Para el caso de la distribución aleatoria, tanto por el conjunto de datos introalg como para mumuki.io, la RNN baja su desempeño para el caso de estudiante cold start. El desempeño es similar al de la RNN entrenada en base series temporales.

6.4.2. Análisis cualitativo: ¿Qué aprende la red?

Considerando el desempeño obtenido en la tarea con la RNN, presentamos un análisis cualitativo del modelo. Para hacerlo, nos centramos en el ejercicio esBisiesto. En la Figura 6.10 observamos la interfaz de Mumuki y el enunciado del ejercicio que los estudiantes tienen que resolver.

Figura 6.10: Interfaz de Mumuki y el enunciado del ejercicio que los estudiantes tienen que resolver.

Para este ejercicio, los estudiantes, tienen que crear un programa en Haskell que permita identificar si un número es bisiesto o no. La Tabla 6.4 muestra la confianza de la red para las predicciones de programas enviados para el ejercicio esBisiesto. En la columna *Programa enviado* presentamos diferentes programas enviados por diferentes estudiantes para intentar resolver el ejercicio esBisiesto. En la columna *Conf.* reportamos la confianza de la RNN para predecir si

el programa enviado es clasificado como abandono. Mientras el valor sea más cercano a 1, la confianza de la RNN para predecir el programa enviado como abandono es alta. Los programas enviados están organizados en base al nivel de confianza de la red para predecir como abandono. Mientras que, cuando el valor es cercano a 0 la RNN esta segura de clasificar el programa como no abandono o éxito. En las columnas *Evaluación* y *Feedback* reportamos como evalúa y el tipo de feedback formativo que genera Mumuki para cada programa enviado. Finalmente, en la columna *Set* reportamos si la RNN clasificó como abandono (1) o no abandono (0) el programa enviado. La clasificación binaria se realiza simplemente cortando en 0,5.

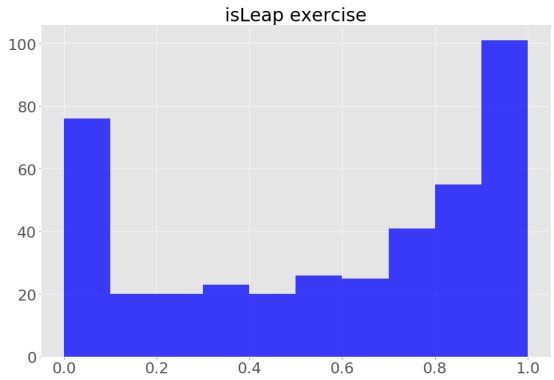


Figura 6.11: Confianza de predicción de la RNN para el ejercicio esBisiesto.

La tabla muestra que no es posible decidir si un estudiante abandona o no un ejercicio teniendo en cuenta únicamente errores sintácticos y semánticos. En la tabla observamos programas enviados con errores sintácticos, clasificados tanto como abandono o no abandono por la RNN. Lo mismo sucede con los programas enviados con errores semánticos.

Por ejemplo, en el programa enviado número 10 (*esBisiesto n = mod n 400 == 0 || (not (mod n 100 == 0) && (mod n 4 == 0)*) el estudiante cometió un error de sintaxis (se olvido de cerrar un paréntesis), pero al ser un error menor, la RNN se da cuenta de por más que tenga un error de sintaxis, ese error pueda ser superado por el estudiante y clasifica al programa como no abandono. No sucede lo mismo cuando la RNN predice sobre el programa enviado número 5 (*esBisiesto = mod n 400 4*), donde el error de sintaxis refleja diversos errores conceptuales. En este caso el estudiante no comprende la cantidad de argumentos que utiliza la función ni tampoco que las variables deben ser definidas antes de ser utilizadas. El modelo clasifica el programa enviado número 5 como abandono.

Esta misma situación se refleja también para programas con errores semánticos. Hay programas enviados con errores de casos de test que la RNN está segura de que el estudiante no abandonará el ejercicio, mientras que para otros programas con errores de caso de test, está segura que el estudiante abandonará. Por ejemplo, el programa enviado número (*esBisiesto x=not (esMultiploDe x 4)*), el cual es evaluado por Mumuki como Error de caso de test, la RNN esta segura de que el estudiante abandonará el ejercicio. Un número es bisiesto si es múltiplo de 400, o bien es múltiplo de 4 pero no de 100. El programa enviado, el estudiante define una única condición para intentar resolver el problema esBisiesto. Y la definición de la condición es incorrecta, ya que el estudiante niega la misma. No sucede lo mismo con el programa número 12 (*esBisiesto n = esMultiploDe n 400 && not (esMultiploDe n 100) && (esMultiploDe n 4)*), evaluado como Error de caso de test por Mumuki, pero la predicción de la RNN es no abandono. En este caos, el estudiante define correctamente las dos condiciones que hacen que un número sea considerado como bisiesto. El problema de su solución es que en vez de verificar si una de

las dos condiciones es correcta, verifica que las dos sean correctas en simultáneo. Es decir, que el número sea múltiplo de 4 y además que no sea múltiplo de 100 y sea múltiplo de 4. Debería cambiar la conjunción por una disyunción y el programa sería correcto.

La Figura 6.11 muestra que la polarización de la predicción es buena, lo que significa que la red puede clasificar los programas enviados como abandono y éxito con confianza. La figura muestra la predicción para los programas enviados para resolver el ejercicio esBisiesto del conjunto de datos introalg.

Id	Programa enviado	Conf.	Error	Feedback	Set
1	esBisiesto esMultiploDe x*4 = x*4	1.000	Sintaxis	Error en patrón	1
2	esBisiesto x=not (esMultiploDe x 4)	1.000	Caso de test	Tests incorrectos: 8, 800, 1004, 1996, 2000, 2014. 2021, 2022	1
3	esBisiesto 400=not esMultiploDe 400 100 A	0.999	Caso de test	Test incorrectos: todos	1
4	esBisiesto = mod 400 4 && not (mod 100)	0.985	Caso de test	Error de tipo	1
5	esBisiesto = mod n 400 4 && not (mod n 100)	0.918	Sintaxis	Variable no definida	1
6	esBisiesto n = mod n 400 4 && not (mod n 100)	0.867	Sintaxis	Cantidad de argumentos	1
7	esBisiesto n = mod n 400 && not (mod n 100)	0.790	Caso de test	Error de tipo	1
8	esBisiesto n = mod n 400 == 0 && not (mod n 100 == 0)	0.618	Caso de test	Tests incorrectos: 8, 800, 1004, 1996, 2000	1
9	esBisiesto n = mod n 400 == 0 && not (mod n 100 == 0) && (mod n 4 == 0)	0.310	Caso de test	Tests incorrectos: 100, 200, 2014, 2021, 2022	0
10	esBisiesto n = mod n 400 == 0 || (not (mod n 100 == 0) && (mod n 4 == 0)	0.222	Sintaxis	Paréntesis desbalanceados	0
11	esBisiesto n = esMultiploDe n 400 && not (esMultiploDe n 100)	0.238	Caso de test	Tests incorrectos: 100, 200, 2014, 2021, 2022	0
12	esBisiesto n = esMultiploDe n 400 && not (esMultiploDe n 100) && (esMultiploDe n 4)	0.011	Caso de test	Tests incorrectos: 100, 200, 2014, 2021, 2022	0
13	esBisiesto n = esMultiploDe n 400 || not (esMultiploDe n 100) && (esMultiploDe n 4))	0.006	Ninguno	Programa correcto	0
14	esBisiesto n = (esMultiploDe n 400) || ((esMultiploDe n 4) && not(esMultiploDe n 100))	0.001	Ninguno	Programa correcto	0

Tabla 6.4: Programas enviados por un estudiante para intentar resolver el ejercicio esBisiesto.

6.4.3. Análisis de tiempos

En esta sección reportamos los tiempos de entrenamiento y de predicción que demandan los modelos implementados. En particular, presentamos los tiempos correspondientes a los modelos entrenados con las series temporales para introalg y mumuki.io. Tanto los modelos neuronales como los modelos de regresión lineal fueron entrenados sobre una GPU Nvidia Tesla K40, brindada por la Facultad de Matemática, Astronomía, Física y Computación. En la Tabla 6.5 informamos el tiempo necesarios para entrenar del modelo de regresión logística con bag of words más las características abandonos por estudiante y proporción de programas abandonados (B + APE + PA) y para cada una de los modelos neuronales durante veinte épocas. Informamos los resultados tanto para introalg como mumuki.io. Los tiempos reportados son el resultado de promediar los resultados de realizar validación cruzada 5 veces para series temporales.

	introalg	**mumuki.io**
	Tiempo en segundos	
B + APE + PA	0.3	15.89
LSTM (Embeddings 256)	1804	17364

Tabla 6.5: Tiempos de entrenamientos en segundos para los modelos implementados.

El entrenamiento de la red se realiza generalmente offline, por lo que es importante conocer los tiempos de predicción de cada uno de los modelos pensando en un sistema en producción. Teniendo en cuenta que una de las particularidades de este trabajo es que el modelo propuesto puede ser integrado a modo de prueba en el sistema Mumuki, es importante que el tiempo de respuesta de la solución propuesta sea lo más rápido posible. A continuación en la Tabla 6.6 se muestra el tiempo que demora en clasificar un programa enviado cada uno de los modelos.

	introalg	**mumuki.io**
	Tiempo de predicción	
B + APE + PA	4 ms	16 ms
LSTM (Embeddings 256)	1.9 ms	1.8 ms

Tabla 6.6: Tiempos de predicción en milisegundos para cada una de los modelos con mejor desempeño.

6.4.4. Limitaciones y fortalezas del estudio

En esta sección describimos las limitaciones del estudio, poniendo en discusión la confiabilidad y validez de los métodos implementados.

Tanto las características definidas como el modelo neuronal tuvieron desempeños estables en base a los resultados obtenidos en ambos conjuntos de datos. En base a entrevistas realizadas a estudiantes que cursaron Introducción a los Algoritmos en 2018, encontramos consistencia en cuanto a nuestras definiciones de sesión y abandono. Los estudiantes respondieron en varias ocasiones que muchas veces cuando no podían resolver un ejercicio, decidían cambiar a otros ejercicios y resolverlos. Para poder seguir evaluando la confiabilidad de nuestros modelos es interesante utilizar conjuntos de datos generados con otros lenguajes de programación.

Con respecto a la validez de la metodología implementada, podemos reflexionar sobre cada una de los métodos propuestos. Con respecto a la anotación automática del conjunto de datos

propuesta, anotamos como abandono todos los programas de un estudiante perteneciente a una sesión tal que el último programa enviado no fuese correcto. Intuitivamente, podría ser mejor si de alguna manera consideramos qué tan lejos se encuentra un programa enviado de ser clasificado como abandono.

Para la ingeniería de características, es claro que existen más características de los que definimos y probamos. No estamos afirmando, en absoluto, que nuestras características sean las mejores para este problema. Sí podemos afirmar que la ingeniería de características para esta tarea es compleja, particularmente, para una tarea como ésta, en el que hay características en tantas dimensiones posibles (estudiante, ejercicio, programa, tiempo, etc.). En ese sentido, podemos decir que la ingeniería de características puede no ser necesaria ya que tenemos datos secuenciales tan ricos que pueden ser procesados por los modelos neuronales profundos actuales y lograr buenas predicciones para la tarea. Al evitar el uso de datos específicos del alumno, nuestros modelos no sufren del cold start para un nuevo estudiante.

La arquitectura de nuestro modelo neuronal es demasiado simple. Estamos ignorando los programas anteriores enviados por el mismo estudiante que ciertamente contienen información sobre lo que ese estudiante aprende. Nuestra contribución es mostrar que sin esa maquinaria pesada, el modelo puede ser lo suficientemente preciso como para predecir si un estudiante necesitará ayuda en un ejercicio en particular. Imaginamos que esto es similar a cuando un docente experimentado mira un programa enviado por un estudiante y al ver su código puede hacer una predicción de si al estudiante le irá bien en el examen. El docente puede hacer esto de manera simple si el programa enviado por el estudiante es muy bueno o muy malo, pero hay muchos casos difíciles en el que nuestro modelo ha demostrado tener un desempeño mejor que el docente cuando no conoce a los estudiantes.

Con respecto al desempeño humano al tratar de predecir la tarea propuesta, nuestra anotación no informa al docente sobre la identidad de los estudiantes que enviaron esos programas. Es probable que a los docentes tuvieran mejor desempeño si supieran quién es el estudiante que envió cada uno de los programas. Nuestro objetivo son los cursos superpoblados o los entornos de enseñanza masivos, donde es demasiado difícil o imposible poder hacer un seguimiento personalizado de cada estudiante. Por lo tanto intentamos reproducir esta configuración al hacer la evaluación humana. Por otro lado, los docentes hicieron 40 clasificaciones en dos horas, esta es una tarea cognitiva exigente y su rendimiento sería mejor haciendo las clasificaciones correspondientes con otra distribución de los tiempos.

6.4.5. Implicancias para el aula heterogénea

Cómo describimos anteriormente, hay un interés mundial en motivar a los jóvenes para que aprendan a programar y en llevar la enseñanza de la programación a todos los niveles educativos. Pero tenemos un problema, tenemos demasiados estudiantes para la cantidad de docentes formados con los que contamos. Para poder intentar resolver este problema se implementaron diferentes estrategias que describimos y analizamos en este trabajo como, el desarrollo de entornos de enseñanza de programación, desarrollo de material didáctico y recursos pedagógicos, organización de eventos como game jams o hackathons y cursos de formación docente.

Las diferentes experiencias enumeradas anteriormente se enfocan en las herramientas y recursos que pueden ayudar al docente en el aula. No significa que brinden apoyo en todos los aspectos relacionados con el aula. Una de ellas es la diversidad de estudiantes en el aula. Pero no nos referimos sólo al género o la diversidad cultural. Estamos interesados en los diferentes procesos y estilos de aprendizaje [94] de los estudiantes. Si no contamos con suficientes docentes y muchos de ellos son principiantes, agregar la tarea a los maestros de reconocer los diferentes

procesos en un aula heterogénea con muchos estudiantes no es una tarea sencilla de resolver. En la Sección 6.4 describimos cómo fue el desempeño de docentes experimentados que intentaban predecir en base a un programa enviado si el estudiante abandona o no el ejercicio. La tarea fue difícil para los docentes especializados y con experiencia en el área. Proporcionar herramientas que identifiquen automáticamente a los estudiantes que están en riesgo de abandonar un ejercicio propuesto por el docente, puede ser esencial para la enseñanza de la programación en aulas heterogéneas y en el contexto actual con respecto a los docentes formados. Predecir automáticamente qué estudiante abandonará un ejercicio podría ayudar a los maestros a reconocer fácilmente el nivel de fluidez de los estudiantes e identificar de manera más simple a los estudiantes con dificultades. Además, poder predecir abandono de forma temprana podría usarse como información relevante para definir diferentes trayectorias de ejercicios por estudiante.

6.5. Trabajo Previo

Hay bastante trabajo utilizando técnicas de PLN para dar soporte a las tareas de ingeniería de software [56] y como asistencia para el aprendizaje de un segundo idioma [109]. Sin embargo, el uso de PLN para apoyar el aprendizaje de lenguajes de programación es un área prometedora no muy explorada.

Intuitivamente, proponemos modelar el nivel de fluidez que tiene un programador principiante en un lenguaje de programación, de la misma manera como en trabajo previo se ha modelado el nivel de fluidez de los personas al momento de aprender un segundo idioma. A continuación, revisamos el trabajo previo sobre aprendizaje de un segundo idioma. También revisamos trabajo previo sobre word embeddings aplicados a programas.

6.5.1. Aprendizaje de segundo idioma

De la misma forma en la que existen diferentes tipos de entornos para aprender lenguajes de programación, existen muchas aplicaciones educativas para el aprendizaje de un segundo idioma. Las mismas han aumentado su popularidad en los últimos años. Estas aplicaciones generan grandes cantidades de datos sobre el aprendizaje de los estudiantes, los cuales pueden ser utilizados para generar atención personalizada. En este contexto, se propuso la tarea compartida de modelado de aprendizaje de segundo idioma (SLA, siglas del inglés de second language acquisition) [109]. Dado un historial de ejercicios que intentaron resolver estudiantes al aprender un segundo idioma, la tarea es poder predecir en qué momento futuro el estudiante cometerá un error.

Un hallazgo para la tarea SLA, para esta formulación particular del problema, es que la elección del algoritmo de aprendizaje parece ser más importante que implementar de forma inteligente ingeniería de características. En particular, los equipos más efectivos emplearon redes neuronales recurrentes que pueden capturar la naturaleza secuencial del lenguaje producido por los estudiantes. Además, el uso de un framework multitarea, un modelo unificado que aprovecha los datos de todos los diferentes tipos de ejercicios, sin importar quién sea el estudiante, puede proporcionar mejoras adicionales. Estos resultados sugieren dos ideas claves para el modelado de SLA. Primero, los algoritmos no lineales son particularmente deseables, y segundo, los enfoques de aprendizaje multitarea que comparten información entre ejercicios también son efectivos.

TMU [60] presenta un modelo que participa en la tarea compartida de aprendizaje de segundo idioma. Utilizan una RNN bidireccional implementada como una LSTM para predecir errores potenciales de un estudiante particular en ejercicio dado en base a una respuesta corta generada

por el estudiante. El modelo fue entrenado con respuestas anteriores generadas por diferentes estudiantes. Los autores no realizaron ingeniería de características, pero entrenaron un solo modelo para muchos ejercicios.

El trabajo descripto anteriormente [124] entrena su modelo neuronal utilizando un programa completo como un elemento del vocabulario de la LSTM. Si consideramos una analogía con las palabras, las oraciones son los programas sucesivos que los estudiantes presentan, es decir, sus trayectorias de aprendizaje. A diferencia de ellos y más similar a lo que realiza TMU hace para SLA, nuestro vocabulario son las palabras del programa. Por ejemplo, para nosotros "var1 = 3" contiene tres palabras de nuestro vocabulario. Como resultado, nuestro modelo puede capturar la similitud entre el ejemplo anterior y "var1 = 2" mientras que para [124] estos son dos programas diferentes. Decimos que nuestro modelo se entrena en *nivel de palabra* mientras que el modelo [124] se entrena en el *nivel de programa*. Usamos un word embedding para cada palabra del programa que estamos analizando mientras que Wang et al. en [124] usan un word embedding para cada programa completo de la trayectoria del estudiante. Como resultado, nuestro modelo puede aprender de diferentes ejercicios en un modelo unificado, mientras que [124] entrena un modelo diferente para cada ejercicio.

Proponemos comparar la ingeniería de características con los modelos de aprendizaje profundo (como los LSTM). En particular, optamos por modelos profundos que no requieren un historial de interacción con un SOLO estudiante, sino que aprenden de otros estudiantes. Lo hacemos para evitar un cold start para un nuevo estudiante. Esto es similar a lo que hizo [60] para SLA y diferente a [124]. La intuición es que los estudiantes que tienen niveles de fluidez similares en un ejercicio de programación y cometen errores similares, requerirán ayuda en puntos similares en su progreso.

6.5.2. RNN y word embeddings

Aunque son un desarrollo reciente en el área del procesamiento del lenguaje natural (PNL), los word embeddings son el método más avanzado para representaciones semánticas léxicas [80]. Los word embeddings representan relaciones semánticas entre palabras que se pueden aprender de grandes bases de datos de texto plano. Por ejemplo, puede aprender que la palabra de `if` de Haskell se puede usar en lugar de | si los datos de entrenamiento contienen suficientes ejemplos de los dos casos. La información semántica se representa en un espacio vectorial. La relación semántica entre `if` y | se representa en los word embeddings con dos vectores que son similares. Es decir, su resta es cercana a cero. También puede aprender relaciones entre más de dos palabras. Por ejemplo, podría aprender que 0 (el número cero) es a la suma como True (la constante booleana) a la conjunción. Cada una de las constantes corresponde al elemento neutro del operador.

Los word embeddings se usan generalmente para lenguajes naturales y no para lenguajes de programación. Sin embargo, hay algunos trabajos recientes que aplican word embeddings a programas para buscar funciones específicas en grandes conjuntos de datos de código [107]. Los word embeddings se aplican sobre programas que son (al menos) sintácticamente correctos. En este trabajo, proponemos entrenar nuestros propios word embeddings de forma similar a como hacen en [107] pero con programas creados por estudiantes principiantes que con frecuencia no son sintácticamente correctos. Además, en [107] los word embeddings se utilizan para encontrar ejemplos de código en bases de datos de código grandes. Para algunas preguntas, por ejemplo, "¿Cómo cerrar u ocultar mediante el teclado virtual de Android?", la información está fácilmente disponible en recursos populares como Stack Overflow. Pero las preguntas específicas del código propietario o las API (o el código escrito en lenguajes de programación menos comunes) necesitan

una solución diferente, ya que generalmente no se discuten en esos foros. Además, a diferencia de nuestro trabajo donde combinamos la semántica de diferentes palabras usando redes LSTM como se describe en la sección anterior, usan técnicas de recuperación de información (en particular TF – IDF) porque están trabajando con una tarea de búsqueda y no una clasificación binaria cómo en nuestro caso.

En la matriz de word embeddings, dos representaciones vectoriales son muy cercanas si las palabras correspondientes aparecen en contextos similares. Definimos la relación semántica de la siguiente manera: las palabras con vectores que están más cerca deberían tener significados relacionados. Esto se llama la hipótesis de distribución en la literatura de PLN [120], y creemos que el mismo concepto es válido para el código de los estudiantes.

6.6. Conclusiones del capítulo

Durante el desarrollo de los experimentos trabajamos sobre un conjunto de datos novedoso, sobre el cual todavía no se había trabajado. Propusimos una metodología para anotar de forma automática el conjunto de datos, formalizando la tarea de evaluar el nivel de fluidez de un estudiante al aprender a programar como: *dado un programa incorrecto creado por un estudiante para resolver un ejercicio de programación específico, predecir si el estudiante podrá resolver el ejercicio por sí mismo en un futuro cercano o abandonará el ejercicio*. Para la tarea propuesta implementamos diferentes modelos que intentasen predecir la misma. Como primer alternativa formalizamos dos dimensiones (estudiante y ejercicio) para construir características que permitan modelar la trayectoria del estudiante y así poder entrenar diversos modelos de aprendizaje automático y aprendizaje profundo que sean capaces de llevar adelante esa tarea de predicción. Para cada una de las dimensiones definimos características que intentaron capturar diferentes hipótesis basadas en teorías pedagógicas.

Construimos modelos de aprendizaje automático que realizan la tarea definida entre 1 y 16 milisegundos. Si la tarea de predecir abandonos fuera realizada por un humano sería prohibitivamente costosa, ya que el tiempo que demandaría y la cantidad de información que se debe procesar es muy grande. Esto nos permite realizar clasificación a gran escala, ayudando tanto a docentes especializados y con experiencia a realizar su tarea de forma más eficiente como a docentes con menos experiencia. Los experimentos con la red neuronal mejoraron el desempeño sobre ambos conjuntos de datos utilizando únicamente el programa enviado por el estudiante.

Nos llevo varios meses probar diferentes características. Las características reportadas en este capítulo son una parte del total de características definidas. Todas las características probadas están detalladas en el Apéndice B. Entrenar las RNN definidas es un trabajo que se completó en menos de una semana. Si bien el desempeño de ambos modelos es similar, el trabajo de ingeniería de características quizás se puede ver reemplazado por el tiempo de entrenamiento de la red.

En la Sección 6.4 presentamos los resultados para los diferentes modelos implementados en base a dos metodologías de distribución de los conjuntos de entrenamiento y test: distribución random y series temporales. El uso de división de los conjuntos de entrenamiento y test utilizando una distribución random es un error metodológico para este tipo de datos. Esto lo muestra la diferencia significativa entre los resultados obtenidos por las dos metodologías. El F_1 de 0.85 no es el desempeño que obtendríamos si pusiéramos el modelo en producción dentro de Mumuki, porque no es correcto entrenar en el futuro y predecir en el pasado. Si el sistema se pone en producción el desempeño esperado de F_1 es de 0.70, obtenido con series temporales.

Con respecto a los modelos de RNN entrenados, el modelo que contaba con información previa sobre el estudiante para realizar las predicciones no tiene un mejor desempeño para estudiantes

nuevos al compararlo con el modelo de RNN que en el conjunto de test encontraba únicamente programas enviados por nuevos estudiantes. Estos resultados, dan la pauta de que el modelo que cuenta con información previa de los estudiantes no esta pudiendo aprovechar la identidad del estudiante para poder generalizar características de los estudiantes, cómo la proporción de programas abandonados (PA). Por lo tanto, contar con información previa de estudiantes no está ayudando a la clasificación o no está pudiendo aprender.

Como conclusión, encontramos que la tarea de decidir cuándo un estudiante necesita ayuda personalizada porque está por abandonar un ejercicio no es una tarea fácil. Pero aún así decidimos abordarlo de dos formas distintas por un lado con un diseño minucioso de características pedagógicamente motivadas y por otro lado utilizando técnicas de procesamiento de lenguaje natural aplicado al texto plano generado por los estudiantes y usando esto como entrada para una red neuronal secuencial. El aprendizaje obtenido a partir del diseño de características es interesante ya que motiva parte del trabajo futuro y motiva también un trabajo extra que es clasificar el tipo de estudiante. Por último logramos construir modelos que están listos para ser puestos a prueba en el sistema Mumuki ya que el tiempo de clasificación de los modelos es realmente corto, lo cual permite que sea integrado en un sistema web sin agregar delay a la experiencia de usuario. Además los tiempos de clasificación conseguidos serían muy difícil de alcanzar si la tarea fuese realizada por un humano, pero aún es necesario seguir trabajando sobre la misma.

El desempeño utilizando ingeniería de características se basan wn información histórica de un estudiante. Los modelos presentados basados en PLN no necesitan información histórica del estudiante. Esta es una diferencia importante dado que uno podría pensar que los estudiantes que recién empiezan son los que necesitan más ayuda para saber si están yendo por el camino correcto.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo presentamos las conclusiones principales en base al trabajo realizado en esta tesis y direcciones posibles en las que continuar trabajando en el futuro.

Si tuviéramos que resumir las conclusiones en esta tesis en una lista de items serían los siguientes. Se agrega énfasis cuando se mencionan los aspectos de adquisición de lenguaje sobre los que se enfoca esta tesis.

- Es difícil formar docentes en actividad para que puedan enseñar a programar de forma efectiva.
- Es posible enseñar conceptos fundamentales de programación incluso a infantes que no saben leer ni escribir, con lenguajes de *expresividad* adaptada.
- La traducción automática a lenguajes de programación con mayor nivel de *expresividad* puede llevar a la exploración de lenguajes más expresivos.
- Ciertas habilidades adquiridas con un lenguaje de bloques son transferibles a un lenguaje basado en texto.
- La *interactividad formativa* automatizada de distintos tipos de errores de programación puede ser útil para disminuir la deserción en cursos numerosos.
- Es posible, hasta cierto punto, evaluar automáticamente la *fluidez* de un estudiante en un lenguaje y un problema dado. Esto puede usarse para predecir si podrá resolver el problema sin ayuda de un docente.

En la siguiente sección elaboramos sobre estos puntos. En la Sección 7.2 describimos diferentes líneas de trabajo futuro en base a los resultados obtenidos.

7.1. Conclusiones

Como describimos en el Capítulo 1 hay un interés mundial en promover y motivar la enseñanza de la programación. Es el caso de países como Estados Unidos [128], Nueva Zelanda [9, 10], Reino Unido [21] y Argentina. La falta de docentes formados en el área es un problema compartido por los países que quieren implementar la enseñanza de la programación en las escuelas. Además los cursos de formación docente en actividad no tienen el efecto esperado en la incorporación conceptual por parte de los docentes como analizamos en el Capítulo 2. Los cursos que describimos fueron insuficientes para preparar a los docentes sin experiencia previa en programación ya

que abordaron sin profundidad y con errores importantes los conceptos de programación en sus aulas. Teniendo en cuenta la dificultad de poder contar y formar docentes en programación, el desarrollo de entornos para la enseñanza de programación ha tomado un rol relevante para poder promover la enseñanza. Muchos entornos han sido desarrollados, pero pocos han sido evaluados en diferentes experiencias en contextos educativos reales como describimos en el Capítulo 3. Las experiencias documentadas y publicadas son de laboratorio, con pocos estudiantes y de unas pocas clases. En el Capítulo 3 ejemplificamos los entornos existentes teniendo en cuenta tres aspectos que impactan al momento de adquirir un nuevo lenguaje: expresividad, interactividad formativa y evaluación automática de la fluidez.

Para poder evaluar el impacto del nivel de expresividad del lenguaje al enseñar a programar, desarrollamos el entorno de enseñanza UNCDuino [72, 13] y el lenguaje Icony para programar kits de robótica basados en Arduino [35], los cuales describimos en el Capítulo 4. UNCDuino es un entorno multilenguaje que traduce de lenguajes de menor expresividad a lenguajes de mayor expresividad. Icony es un lenguaje de expresividad adaptada diseñado para enseñar los conceptos fundamentales de programación a infantes que no saben leer y escribir.

A partir de las experiencias descriptas y analizadas para nivel inicial en el Capítulo 4 observamos que es posible enseñar conceptos fundamentales de programación a niños de nivel inicial utilizando un lenguaje adaptado como Icony. En base a las experiencias en nivel primario que reportamos en el Capítulo 4, los estudiantes al poder contar con lenguajes de diferente expresividad, incorporaron los conceptos de programación en base a sus necesidades e intereses. Como describimos en las experiencias de escuela primaria utilizando el entorno UNCDuino, los estudiantes al contar con traducción automática a lenguajes de programación con mayor nivel de expresividad exploraron nuevos lenguajes de programación. Los entornos de enseñanza de programación con que cuenten con traducción automática a lenguajes de mayor expresividad como UNCDuino, pueden ser herramientas que permitan profundizar el aprendizaje de conceptos de programación y la adquisición de lenguajes de programación. La necesidad de contar con lenguajes de mayor expresividad para crear programas más complejos, motiva a los estudiantes a explorar y adquirir nuevos lenguajes de programación.

En el Capítulo 5 describimos experiencias de enseñanza de programación utilizando un entorno con *interactividad formativa* automatizada para introducir lenguajes de texto. Las experiencias de desarrollaron en nivel primario y nivel universitario. En ambas experiencias participaron dos grupos de estudiantes. En el caso de escuela primaria, uno de los grupos tenía experiencia en programación con lenguajes de bloques mientras que el otro grupo no tenía experiencia en programación. En el caso universitario, uno de los grupos tenían experiencia con lenguajes de texto imperativos, mientras que para el otro grupo era su primer experiencia con un lenguaje de programación. Para todos los grupos de estudiantes era la primer experiencia con el lenguaje de texto a utilizar. En el caso de escuela primaria utilizamos el lenguaje de texto Gobstones [68], mientras que para nivel universitario utilizamos el lenguaje funcional Haskell.

En el caso de la experiencia en primaria observamos indicios de los lenguajes de bloques utilizados por escuela primaria al momento de aprender a programar facilitan a posteriori el aprendizaje de lenguajes de texto. La *interactividad formativa* automatizada que provee el entorno es la distinción entre errores sintácticos y semánticos. Estos nos permitió observar que los estudiantes con experiencias previa con lenguajes de bloques tienen menos errores semánticos que aquellos estudiantes sin experiencia y necesitan de una menor cantidad de programas enviados para llegar a una solución correcta. Sin embargo, tienen una cantidad similar de errores sintácticos en comparación a aquellos estudiantes sin experiencia. La interactividad formativa nos permitió evaluar las hipótesis planteadas relacionadas a los errores sintácticos y semánticos

que comenten los estudiantes al incorporar un lenguaje de texto cuando tienen o no experiencia previa en lenguajes de bloques. Según Papert y Turkle [119] podemos clasificar a los estudiantes en dos tipos, *tinkerer* y *planner*. Esta clasificación depende del comportamiento de los estudiantes cuando intentan resolver ejercicios de programación. Los estudiantes clasificados como *tinkerers* realizan muchos cambios en los programas a través de prueba y error para crear una solución final válida. Los estudiantes clasificados como *planners* identifican un camino de acción y luego lo implementan con el objetivo de llegar a la solución que consideran válida. Los lenguajes basados en bloques son criticados en general por fomentar la práctica de la prueba y error frecuentemente. Pero, cuando comparamos el aprendizaje de un lenguaje basado en texto utilizando un entorno cerrado, entre estudiantes con experiencia en bloques y estudiantes sin experiencia en bloques, en base a la cantidad de programas enviados, los estudiantes con experiencia en bloques enviaron una menor cantidad de programas. Por lo tanto, los estudiantes con experiencias en lenguajes de bloques empiezan a adquirir nuevas habilidades relacionadas a la programación.

En base a las experiencias en la universidad, la interactividad formativa que provee el entorno que describimos en el Capítulo 5 , el cual distingue entre errores sintácticos y semánticos, nos permitió ver en dos cursos diferentes en dos universidades distintas que puede tener impacto sobre la deserción de los estudiantes en cursos numerosos. La percepción de los estudiantes sobre este tipo de entornos es positiva e indicaron que la seguirían utilizando en el futuro.

Teniendo en cuenta la falta de docentes formados, el interés de enseñar a programar en las escuelas y la cantidad de estudiantes en las aulas, la interactividad formativa en entornos cerrados puede ser una herramienta útil para este contexto como describimos en el Capítulo 5. La interactividad formativa es una forma de presentar al estudiante un error en su programa. Entornos que brinden feedback formativo a los estudiantes en forma verificación de programas, hints, clasifiquen errores o error flagging de forma automática permiten a los estudiantes tener una respuesta inmediata para desenvolverse de forma autónoma. En nuestro caso, la *interactividad formativa* automatizada que provee el entorno es la distinción entre errores sintácticos y semánticos, lo cual no es suficiente. Ante errores donde el feedback formativo no pueda ayudar a resolver el problema, se nos presenta un inconveniente en aulas con muchos estudiantes. El docente a cargo tiene que seguir haciendo un seguimiento personalizado de los estudiantes, y poder revisar cada uno de los ejercicios y programas resueltos por los estudiantes. Por lo que, es muy difícil que identifique qué estudiantes están teniendo inconvenientes.

Utilizamos la evaluación automática para poder conocer el nivel de fluidez del estudiante con el lenguaje de programación y predecir si el estudiante puede resolver un ejercicio por su propia cuenta. Dentro del área de detección de fluidez la tarea concreta que abordamos se define como, dado un fragmento de un programa creado por un estudiante para resolver un ejercicio, predecir si el estudiante podrá resolver el ejercicio sin ayuda. Encontramos que la tarea de decidir cuándo un estudiante necesita ayuda personalizada porque está por abandonar un ejercicio no es una tarea fácil. En el Capítulo 6 presentamos modelos neuronales y basados en características para intentar evaluar el nivel de fluidez de los estudiantes en base a la tarea definida. El desempeño humano de un docente con experiencia también es bajo y consume mucho tiempo, sobre todo considerando un curso con muchos estudiantes en los cuales el docente no tiene conocimiento personalizado de los mismos. El desempeño de los modelos neuronales y de las características definidas mejoran el desempeño de un docente con 10 años de experiencia al predecir, en base al programa creado por un estudiante, si podrá resolver o no ese ejercicio por su cuenta. A pesar de no ser una tarea fácil, decidimos abordarla de dos formas distintas. Por un lado con un diseño minucioso de características pedagógicamente motivadas y por otro, utilizando técnicas de procesamiento de lenguaje natural aplicado al texto plano generado por los estudiantes y usando

esto como entrada para una red neuronal secuencial. El aprendizaje obtenido a partir del diseño de características es interesante ya que motiva parte del trabajo futuro y motiva también un trabajo extra que es clasificar el tipo de estudiante. Por último logramos construir modelos que están listos para ser puestos a prueba en el sistema Mumuki ya que el tiempo de clasificación de los modelos es realmente corto, lo cual permite que sea integrado en un sistema web sin agregar delay a la experiencia de usuario. Además los tiempos de clasificación conseguidos serían muy difícil de alcanzar si la tarea fuese realizada por un humano, pero aún es necesario seguir trabajando sobre la misma.

Aspectos como expresividad, interactividad formativa y fluidez al adquirir un nuevo lenguaje al enseñar a programar permiten al docente pensar en un aula heterogénea. Contar con entornos que permitan programar con lenguajes con diferentes niveles de expresividad y con traducción automática de lenguajes de menor a mayor expresividad, permite a los estudiantes incorporar conceptos y lenguajes de programación respetando sus tiempos. La interactividad formativa automatizada permite a los estudiantes poder desenvolverse de forma autónoma, ya que ante cada programa enviado recibirán feedback formativo específico, pudiendo avanzar a su propio ritmo. De esta manera los estudiantes que no tengan complicaciones pueden avanzar, y los que tengan dificultades poder ser ayudados por el docente. Pero, en aulas heterogéneas y masivas los docentes no cuentan con el tiempo necesario para poder detectar a los estudiantes que tengan dificultades. Por lo tanto poder evaluar el nivel de fluidez de un estudiante en un lenguaje específico de forma automática y alertar al docente que estudiantes están teniendo inconvenientes permiten pensar en la enseñanza de programación en aulas heterogéneas y el contexto educativo en el cual estamos insertos.

En la próxima sección reflexionamos sobre distintas líneas de trabajo futuro a la que este trabajo podría lugar. También reflexionamos sobre distintas consideraciones y cuidados a tener en cuenta antes de utilizar estos sistemas automáticos en funcionamiento.

7.2. Trabajo futuro

En base a los resultados obtenidos por el impacto de los diferentes aspectos de adquisición del lenguaje para la enseñanza de programación, identificamos diferentes propuestas para profundizar como trabajo futuro. En este trabajo nos concentramos en el impacto de tres aspectos del lenguaje: la expresividad, la interactividad formativa y la fluidez. Sería interesante poder evaluar el impacto de otro aspectos de adquisición del lenguaje como el *bilingüismo* o la adquisición léxica contextualizada en un lenguaje natural.

En lenguaje natural se considera bilingüe a una persona que adquiere dos lenguajes maternos a la vez. Desde el nacimiento si está expuesto y aprende dos lenguajes naturales se considera como bilingüe. El caso en que una persona sea expuesta a un solo lenguaje al momento de nacer, y luego decide aprender un nuevo lenguaje no es considerado como bilingüe. Proponemos entonces implementar experiencias en contextos educativos reales y en diferentes niveles educativos para evaluar cómo impacta en un estudiante que está aprendiendo a programar, aprender dos lenguajes de programación en simultáneo.

La adquisición léxica contextualizada en un lenguaje natural, describe que para una persona es más sencillo incorporar nuevas palabras en base al contexto en las cuales son utilizadas y no por buscar su significado en el diccionario. Un estudiante incorpora nuevas palabras a medida que escucha las mismas en diferentes contextos. Podemos implementar experimentos donde comparar si la práctica en la adquisición léxica dentro de un lenguaje de programación puede llevar a una mejora en las habilidades de adquisición léxica contextualizada en un lenguaje natural o

viceversa. Evaluar cómo afecta la adquisición léxica contextualizada al momento de adquirir un lenguaje de programación.

Para el aspecto de expresividad del lenguaje, los resultados obtenidos son muy alentadores para aquellos entornos de enseñanza de programación que cuentan con lenguajes con diferente expresividad y traducción automática de lenguajes de menor a mayor nivel de expresividad. Realizar experimentos de largo plazo utilizando entornos con lenguajes de diferentes expresividad tanto en nivel primario como en otros niveles educativos.

Teniendo en cuenta que la tarea de evaluar el nivel de fluidez como predecir si un estudiante podrá resolver un ejercicio con o sin ayuda, no es una tarea sencilla, proponemos diferentes experimentos para tratar de mejorar el rendimiento de los modelos definidos en el capítulo anterior. En el Capítulo 5 describimos el entorno cerrado Mumuki el cual cuenta con interactividad formativa automatizada. Para poder evaluar la percepción de los estudiantes sobre el entorno Mumuki, adaptamos Modelo de Aceptación de Tecnología (del inglés Technology acceptance model (TAM)) [25]. "Es fácil que Mumuki haga lo que yo quiero" es una de las afirmaciones pertenecientes al TAM que los estudiantes tenían que puntuar. El valor superó por poco la media, y ante esto los estudiantes explicaron que "fue frustrante recibir el mismo mensaje de error para diferentes programas enviados". Mumuki no tiene cambios en los programas enviados cuando genera el feedback formativo. Por lo tanto, una nueva característica para detectar el nivel fluidez puede ser la distancia de Levenshtein entre el feedback formativo brindado por el entorno de programación utilizado.

Otra alternativa simple que podría mejor el desempeño de los modelos neuronales para evaluar la fluidez es probar nuevas estructuras para la red neuronal implementada. A pesar de haber conseguido un buen desempeño con la opción neuronal, se puede seguir mejorando. Una posibilidad sería probar el desempeño de una red neuronal LSTM pero bidireccional la cual permite que la capa de salida de la red obtenga información de estados pasados y futuros. Esto tiene sentido dado que no sólo las palabras siguientes sino también las anteriores dan significado a un token dentro de un programa.

Como describimos en el capítulo anterior logramos definir características que tienen un buen desempeño a la hora de clasificar los programas enviados con un modelo de regresión logística. Sería interesante construir un modelo neuronal que permita combinar las características construidas junto con una representación de la solución como dato de entrenamiento. Y analizar si se consiguen mejoras en el desempeño.

Además de poder evaluar el nivel de fluidez de los estudiantes podemos comenzar a clasificar a los estudiantes teniendo en cuenta los programas que envían. Haber construido características que modelan nivel de experiencia, insistencia, abandono entre otras, nos abre una puerta para identificar qué tipo de estudiante está trabajando en el sistema.

En base el nivel de fluidez de un estudiante podríamos proponer trayectorias de ejercicios específicas. Para ello se le podría sugerir retomar algún ejercicio previamente resuelto donde se trabaje el concepto que puede estar haciéndole falta o simplemente enviarlo a un ejercicio más fácil de modo que lo motive a seguir practicando.

En este trabajo propusimos anotar de forma automática el conjunto de datos considerando toda la sesión como abandono. Recordemos que una sesión estaba compuesta por un conjunto de programas que un estudiante enviaba para resolver un ejercicio. La sesión finaliza cuando el estudiante cambiaba de ejercicio o supera un umbral de tiempo de inactividad con respecto al último programa enviado. La alternativa que proponemos como trabajo futuro para anotar el conjunto de datos, es analizar el desempeño del estudiante a lo largo de la sesión, para poder identificar el momento donde el desempeño comienza a empeorar, y a partir de ese momento y

en adelante como abandono hasta que concluya la sesión.

Es interesante, poder pensar en como un entorno podría aprovechar la predicción de nivel de fluidez. Para ello debemos tener en cuenta los errores que pueden generarse al momento de intentar predecir el nivel de fluidez. Hay dos tipos de errores que pueden ocurrir en nuestros modeles. Primero, es no predecir como abandono a un un estudiante que va a abandonar. Segundo, predecir como abandono a un estudiante que no va a abandonar. En ambos casos hay que tener cuidado con la respuesta del sistema ante la información obtenida por la predicción. Como primer paso es recomendable que el sistema notifique al docente ante un caso de posible abandono. En el caso en el que el sistema predice como abandono, pero el estudiante puede resolver el ejercicio por su cuenta, no sería un inconveniente que el docente revise por las dudas. Por otro lado, no sería bueno que el sistema genere feedback automático en el cual le informe a un estudiante que tiene problemas para resolver el ejercicio, y posiblemente abandone el mismo, cuando en realidad el estudiante puede resolver el ejercicio por sus propios medios. Decidir que hacer con la información con respecto al nivel de fluidez generado por un sistema, teniendo en cuenta la tasa de error del sistema desarrollado y el impacto que tiene la información que se transmite al estudiante sobre su comportamiento futuro requiere un nuevo trabajo de investigación.

Otra dirección de trabajo teniendo en cuenta la predicción de fluidez es no predecir si el estudiante va a abandonar o no un ejercicio, sino intentar predecir el tipo de error que tuvo el estudiante. Para poder verificar la correctitud de las predicciones, se requiero esfuerzo de muchos docentes para realizar una clasificación manual del error. Haldeman et al. en [51] implementan una metodología donde los docentes identifican manualmente los conceptos de programación de cada ejercicio, diseñando un conjunto completo de test. Una vez que el programa es enviado se verifican los casos de test. Luego los docentes etiquetan los errores a conceptos y habilidades, validando manualmente la clasificación automática de errores. La experiencia fue implementado en nivel universitario utilizando Java. Piech et al. en [97] intentan utilizar una metodología para reducir el costo de las anotaciones manuales pero aplicado en un entorno como Code.org con un lenguaje de bloques con una expresividad muy limitada. Por lo tanto, los errores que se encuentran son acotados y predecibles. Podemos ver entonces que los lenguajes de programación de texto, utilizados industrialmente o para introducir a la programación a estudiantes universitarios son más expresivos y mayor variabilidad. Nuevamente, se presenta la dificultad de poder cubrir de forma automática el espectro de programas generados utilizando lenguajes de texto.

Bibliografía

[1] Andrea Alliaud. *Los maestros y su historia.* Nuevas perspectivas en educación. Ediciones Granica SA, Buenos Aires, 1 edition, 6 2007.

[2] Carlos Areces, Luciana Benotti, Joshep Cortez-Sanchez, Raul Fervari, E. Garcia, , Cecilia Martinez, Martin Onetti, Eduardo Rodriguez-Pesce, Nicolas. Wolovick, and Marcos J. Gómez. *Ciencias de la Computacion para el aula: 2do ciclo de Primaria.* Ciencias de la Computacion para el aula. Fundacion Sadosky: Program.ar, 1 edition, 8 2018.

[3] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. From scratch to real programming. *ACM Transactions on Computing Education (TOCE)*, 14(4):25, 2015.

[4] Valerie Barr and Deborah Trytten. Using turing's craft codelab to support cs1 students as they learn to program. *Association for Computing Machinery Inroads*, 7(2):67–75, 2016.

[5] Valentin Basel. Proyecto icaro: Robótica pedagógica con software y hardware libre. 2012.

[6] David Bau, D. Anthony Bau, Mathew Dawson, and C. Sydney Pickens. Pencil code: Block code for a text world. In *Proceedings of the 14th International Conference on Interaction Design and Children*, IDC '15, pages 445–448, New York, NY, USA, 2015. ACM.

[7] Andrew Begel. Logoblocks: A graphical programming language for interacting with the world. *Electrical Engineering and Computer Science Department, MIT, Boston, MA*, pages 62–64, 1996.

[8] Tim Bell, Jason Alexander, Isaac Freeman, and Mick Grimley. Computer science unplugged: School students doing real computing without computers. *The New Zealand Journal of Applied Computing and Information Technology*, 13(1):20–29, 2009.

[9] Tim Bell, Peter Andreae, and Lynn Lambert. Computer science in new zealand high schools. In *Proceedings of the Twelfth Australasian Conference on Computing Education-Volume 103*, pages 15–22, 2010.

[10] Tim Bell, Peter Andreae, and Anthony Robins. A case study of the introduction of computer science in nz schools. *ACM Trans. Comput. Educ.*, 14(2), June 2014.

[11] Timothy C Bell, Ian H Witten, and Mike Fellows. *Computer Science Unplugged: Off-line activities and games for all ages.* Citeseer, 1998.

[12] Luciana Benotti, Federico Aloi, Franco Bulgarelli, and Marcos J. Gomez. The effect of a web-based coding tool with automatic feedback on students' performance and perceptions. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 2–7, New York, NY, USA, 2018. ACM.

[13] Luciana Benotti, Marcos J Gómez, and Cecilia Martínez. Unc++ duino: A kit for learning to program robots in python and c++ starting from blocks. In *Robotics in Education*, pages 181–192. Springer, 2017.

[14] Luciana Benotti, María Cecilia Martínez, and Fernando Schapachnik. Engaging high school students using chatbots. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 63–68. ACM, 2014.

[15] Luciana Benotti, María Cecilia Martínez, and Fernando Schapachnik. A tool for introducing computer science with automatic formative assessment. *IEEE Transactions on Learning Technologies*, 11(2):179–192, 2017.

[16] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences*, 23:561–599, 11 2014.

[17] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[18] María Belén Bonello et al. Diez preguntas frecuentes (y urgentes) sobre pensamiento computacional. *Virtualidad, Educación y Ciencia*, 11(20):156–167.

[19] Hilda Borko. Professional development and teacher learning: Mapping the terrain. *Educational researcher*, 33(8):3–15, 2004.

[20] Christian P. Brackmann, Marcos Román-González, Gregorio Robles, Jesús Moreno-León, Ana Casali, and Dante Barone. Development of computational thinking skills through unplugged activities in primary school. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*, WiPSCE '17, page 65–72, New York, NY, USA, 2017. Association for Computing Machinery.

[21] Royal Society (Great Britain). *Shut down or restart?: The way forward for computing in UK schools*. Royal Society, 2012.

[22] Peter Brusilovsky, Stephen Edwards, Amruth Kumar, Lauri Malmi, Luciana Benotti, Duane Buck, Petri Ihantola, Rikki Prince, Teemu Sirkiä, Sergey Sosnovsky, Jaime Urquiza, Arto Vihavainen, and Michael Wollowski. Increasing adoption of smart learning content for computer science education. In *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference*, ITiCSE-WGR '14, pages 31–57, New York, NY, USA, 2014. ACM.

[23] Kathy Carter. Teachers' knowledge and learning to teach. *Handbook of research on teacher education*, pages 291–310, 1990.

[24] François Chollet et al. Keras. `https://keras.io`, 2015.

[25] M. Y. Chuttur. Overview of the Technology Acceptance Model: Origins, Developments and Future Directions. *Sprouts: Working Papers on Information Systems*, 9(37), 2009.

[26] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the International Conference on Functional Programming*, ICFP, pages 268–279, 2000.

[27] Eve V Clark. Conversational repair and the acquisition of language. *Discourse Processes*, pages 1–19, 2020.

[28] Douglas H Clements and Julie Sarama. Teaching with computers in early childhood education: Strategies and professional development. *Journal of Early Childhood Teacher Education*, 23(3):215–226, 2002.

[29] K-12 Computer Science Framework Steering Committee et al. K-12 computer science framework. 2016.

[30] Stephen Cooper, Wanda Dann, and Randy Pausch. Alice: a 3-d tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5):107–116, 2000.

[31] Stephen Cooper, Wanda Dann, and Randy Pausch. Teaching objects-first in introductory computer science. *SIGCSE Bull.*, 35(1):191–195, January 2003.

[32] Valentina Dagiene, Linda Mannila, Timo Poranen, Lennart Rolandsson, and Par Söderhjelm. Students' performance on programming-related tasks in an informatics contest in finland, sweden and lithuania. In *Proceedings of the 2014 Conference on Innovation; Technology in Computer Science Education*, pages 153–158. ACM, 2014.

[33] Allison Druin, James A Hendler, and James Hendler. *Robots for kids: exploring new technologies for learning.* Morgan Kaufmann, 2000.

[34] Caitlin Duncan, Tim Bell, and Steve Tanimoto. Should your 8-year-old learn coding? In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, pages 60–69. ACM, 2014.

[35] Alessandro D'Ausilio. Arduino: A low-cost multipurpose lab equipment. *Behavior research methods*, 44(2):305–313, 2012.

[36] Alex Daniel Edgcomb, Frank Vahid, Roman Lysecky, Andre Knoesen, Rajeevan Amirtharajah, and Mary Lou Dorf. *Student performance improvement using interactive textbooks: A three-university cross-semester analysis.* American Society for Engineering Education, 2015.

[37] Susan Einhorn. Microworlds, computational thinking, and 21st century learning. *LCSI White Paper*, pages 1–10, 2012.

[38] Barbara Ericson, Mark Guzdial, and Maureen Biggers. Improving secondary CS education: progress and problems. In *SIGCSE Bulletin*, volume 39, pages 298–301, 2007.

[39] Louise P Flannery and Marina Umaschi Bers. Let's dance the "robot hokey-pokey!çhildren's programming approaches and achievement throughout early cognitive development. *Journal of Research on Technology in Education*, 46(1):81–101, 2013.

[40] Louise P Flannery, Brian Silverman, Elizabeth R Kazakoff, Marina Umaschi Bers, Paula Bontá, and Mitchel Resnick. Designing scratchjr: support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children*, pages 1–10. ACM, 2013.

[41] Allan Fowler, Johanna Pirker, Ian Pollock, Bruno Campagnola de Paula, Maria Emilia Echeveste, and Marcos J. Gómez. Understanding the benefits of game jams: Exploring the potential for engaging young learners in stem. In *Proceedings of the 2016 ITiCSE Working Group Reports*, ITiCSE '16, pages 119–135, New York, NY, USA, 2016. ACM.

[42] N Fraser et al. Blockly: A visual programming editor. *URL: https://code. google. com/p/blockly*, 2013.

[43] Michael S Garet, Andrew C Porter, Laura Desimone, Beatrice F Birman, and Kwang Suk Yoon. What makes PD effective? Results from a national sample of teachers. *American educational research journal*, 38(4):915–945, 2001.

[44] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. Ask-elle: an adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, pages 1–36, 2016.

[45] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. An interactive functional programming tutor. In *Proceedings of the Conference on Innovation & technology in computer science education*, pages 250–255. ACM, 2012.

[46] Marcos J. Gomez, Marco Moresi, and Luciana Benotti. Text-based programming in elementary school: A comparative study of programming abilities in children with and without block-based experience. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, pages 402–408, New York, NY, USA, 2019. ACM.

[47] Joanna Goode. If you build teachers, will students come? the role of teachers in broadening computer science learning for urban youth. *Journal of Educational Computing Research*, 36(1):65–88, 2007.

[48] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[49] Lindsey Ann Gouws, Karen Bradshaw, and Peter Wentworth. Computational thinking in educational activities: an evaluation of the educational game light-bot. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 10–15. ACM, 2013.

[50] Jeff Gray, Hal Abelson, David Wolber, and Michelle Friend. Teaching cs principles with app inventor. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 405–406. ACM, 2012.

[51] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D Nguyen. Providing meaningful feedback for autograding of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 278–283, 2018.

[52] Brian Harvey, Daniel D Garcia, Tiffany Barnes, Nathaniel Titterton, Omoju Miller, Dan Armendariz, Jon McKinsey, Zachary Machardy, Eugene Lemon, Sean Morris, et al. Snap!(build your own blocks). In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 749–749. ACM, 2014.

[53] Patrick GT Healey, Christine Howes, Julian Hough, and Matthew Purver. Better late than now-or-never: The case of interactive repair phenomena. *Behavioral and Brain Sciences*, 2016.

[54] Poul Henriksen and Michael Kölling. Greenfoot: combining object visualisation with interaction. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 73–82. ACM, 2004.

[55] Michael Homer and James Noble. Combining tiled and textual views of code. In *2014 Second IEEE Working Conference on Software Visualization*, pages 1–10. IEEE, 2014.

[56] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)*, pages 1606–1612, New York, 2016. IJCAI/AAAI Press.

[57] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, et al. Educational data mining and learning analytics in programming. In *Proceedings of Innovation & Technology in Computer Science Education Conference*, ITiCSE-WGR, pages 41–63. ACM, 2015.

[58] Ken Kahn. Toontalktm—an animated programming environment for children. *Journal of Visual Languages & Computing*, 7(2):197–217, 1996.

[59] Filiz Kalelioğlu. A new way of teaching programming skills to k-12 students: Code. org. *Computers in Human Behavior*, 52:200–210, 2015.

[60] Masahiro Kaneko, Tomoyuki Kajiwara, and Mamoru Komachi. Tmu system for slam-2018. In *Proceedings of the Thirteenth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 365–369, 2018.

[61] Elizabeth R Kazakoff, Amanda Sullivan, and Marina U Bers. The effect of a classroom-based intensive robotics and programming workshop on sequencing ability in early childhood. *Early Childhood Education Journal*, 41(4):245–255, 2013.

[62] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[63] Michael Kölling. The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):14, 2010.

[64] Amruth N. Kumar. The effect of using problem-solving software tutors on the self-confidence of female students. *Special Interest Group in Computer Science Education (SIGCSE) Bulletin*, 40(1):523–527, March 2008.

[65] Amruth N. Kumar. The effectiveness of visualization for learning expression evaluation. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 362–367, New York, NY, USA, 2015. ACM.

[66] Fred Lin. Blocklyduino, a web-based editor for arduino. *Arduino LLC*, 2015.

[67] Richard Lobb and Jenny Harlow. Coderunner: A tool for assessing computer programming skills. *ACM Inroads*, 7(1):47–51, February 2016.

[68] Pablo E Martínez López, Daniel Ciolek, Gabriela Arévalo, and Denise Pari. The gobstones method for teaching computer programming. In *2017 XLIII Latin American Computer Conference (CLEI)*, pages 1–9, 2017.

[69] Stéphane Magnenat, Jiwon Shin, Fanny Riedo, Roland Siegwart, and Morderchai Ben-Ari. Teaching a core cs concept through robotics. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*, pages 315–320, NY, USA, 2014. ACM.

[70] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[71] L. Mannila, V. Dagiene, B. Demo, N. Grgurina, C. Mirolo, L. Rolandsson, and A. Settle. Computational thinking in k-9 education. In *Proceedings of the Working Group Reports of the 2014 on Innovation &; Technology in Computer Science Education Conference*, ITiCSE-WGR '14, pages 1–29, 2014.

[72] Cecilia Martinez, Marcos J. Gomez, and Luciana Benotti. A comparison of preschool and elementary school children learning computer science concepts through a multilanguage robot programming platform. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pages 159–164, New York, NY, USA, 2015. ACM.

[73] María Cecilia Martinez and María Emilia Echeveste. Representaciones de estudiantes de primaria y secundaria sobre las ciencias de la computación y su oficio. *Revista de Educación a Distancia*, (46), 2015.

[74] María Cecilia Martinez, Marcos J. Gomez, Marco Moresi, and Luciana Benotti. Lessons learned on computer science teachers professional development. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 77–82, New York, NY, USA, 2016. ACM.

[75] María Cecilia Martinez and Marcos Javier Gomez. Programar computadoras en educación infantil. *Edutec. Revista Electrónica de Tecnología Educativa*, (65):40–53, 2018.

[76] Pablo Martinez-Lopez, Daniel Ciolek, Gabriela Arevalo, and Denise Pari. The gobstones method for teaching computer programming. In *2017 XLIII Latin American Computer Conference (CLEI)*, pages 1–9, Sept 2017.

[77] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 168–172. ACM, 2011.

[78] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Learning computer science concepts with scratch. *Computer Science Education*, 23(3):239–264, 2013.

[79] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[80] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[81] Frederic P Miller, Agnes F Vandome, and John McBrewster. Levenshtein distance: Information theory, computer science, string (computer science), string metric, damerau? levenshtein distance, spell checker, hamming distance. 2009.

[82] Francesco Mondada, Michael Bonani, Fanny Riedo, Manon Briod, Léa Pereyre, Philippe Rétornaz, and Stéphane Magnenat. The thymio open-source hardware robot. *IEEE Robotics & Automation Magazine*, 1070(9932/17):2, 2017.

[83] Jesús Moreno-León, Gregorio Robles, et al. Dr. scratch: a web tool to automatically evaluate scratch projects. In *WiPSCE*, pages 132–133, 2015.

[84] Leonel Morgado, Ken Kahn, and Maria GB Cruz. Preschool cookbook of computer programming topics. *Australasian Journal of Educational Technology*, 26(3):309–326, 2010.

[85] Andreas Mühling, Peter Hubwieser, and Torsten Brinda. Exploring teachers' attitudes towards object oriented modelling and programming in secondary schools. In *Proc of the Sixth international workshop on Computing education research*, pages 59–68. ACM, 2010.

[86] Lijun Ni and Mark Guzdial. Who am I?: understanding high school CS teachers' professional identity. In *Proc of the Technical Symposium on Computer Science Education*, pages 499–504. ACM, 2012.

[87] Opssi. Reporte anual sobre el sector de software y servicios informáticos de la república argentina. 2019.

[88] Lluís Padró and Evgeny Stanilovsky. Freeling 3.0: Towards wider multilinguality. In *LREC2012*, 2012.

[89] Mark Palatucci, Dean Pomerleau, Geoffrey E Hinton, and Tom M Mitchell. Zero-shot learning with semantic output codes. In *Advances in neural information processing systems*, pages 1410–1418, 2009.

[90] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA, 1980.

[91] Nick Parlante. CodingBat. `http://codingbat.com`, 2017. [Online; accessed 06-March-2020].

[92] Dale Parsons and Patricia Haden. Programming osmosis: Knowledge transfer from imperative to visual programming environments. In *Procedings of The Twentieth Annual NACCQ Conference*, pages 209–215, 2007.

[93] Hadi Partovi. Transforming us education with computer science. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 5–6, New York, NY, USA, 2014. ACM.

[94] Harold Pashler, Mark McDaniel, Doug Rohrer, and Robert Bjork. Learning styles: Concepts and evidence. *Psychological science in the public interest*, 9(3):105–119, 2008.

[95] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. In *ACM sigcse bulletin*, volume 39, pages 204–223. ACM, 2007.

[96] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[97] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. *arXiv preprint arXiv:1505.05969*, 2015.

[98] Shaileen Crawford Pokress and José Juan Dominguez Veiga. Mit app inventor: Enabling personal mobile computing. *arXiv preprint arXiv:1310.2830*, 2013.

[99] Kris Powers, Stacey Ecott, and Leanne M. Hirshfield. Through the looking glass: Teaching cs0 with alice. *SIGCSE Bull.*, 39(1):213–217, March 2007.

[100] Matthew Purver, Julian Hough, and Christine Howes. Computational models of miscommunication phenomena. *Topics in cognitive science*, 10(2):425–451, 2018.

[101] Noa Ragonis, Orit Hazzan, and Judith Gal-Ezer. A survey of CS teacher preparation programs in Israel tells us: CS deserves a designated high school teacher preparation! In *Proceedings of the 41st Symposium on Computer Science Education*, pages 401–405. ACM, 2010.

[102] Katie Redmond, Sarah Evans, and Mehran Sahami. A large-scale quantitative study of women in computer science at stanford university. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 439–444. ACM, 2013.

[103] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[104] Mitchel Resnick, Stephen Ocko, et al. *LEGO/logo–learning through and about design.* Epistemology and Learning Group, MIT Media Laboratory Cambridge, MA, 1990.

[105] Mitchel Resnick, Stephen Ocko, and Seymour Papert. Lego, logo, and design. *Children's Environments Quarterly*, pages 14–18, 1988.

[106] Mark Rollins. *Beginning Lego Mindstorms Ev3.* Apress, 2014.

[107] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: A neural code search. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 31–41, New York, NY, USA, 2018. ACM.

[108] Alfredo Sanzo, Fernando Schapachnik, Pablo Factorovich, and Federico Sawady O'Connor. Pilas bloques: A scenario-based children learning platform. In *Learning Technologies (LACLO), 2017 Twelfth Latin American Conference on*, pages 1–6. IEEE, 2017.

[109] Burr Settles, Chris Brust, Erin Gustafson, Masato Hagiwara, and Nitin Madnani. Second language acquisition modeling. In *Proceedings of the thirteenth workshop on innovative use of NLP for building educational applications*, pages 56–65, 2018.

[110] Burr Settles, Chris Brust, Erin Gustafson, Masato Hagiwara, and Nitin Madnani. Second language acquisition modeling. In *Proceedings of the Thirteenth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 56–65, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[111] Jason H. Sharp. Using codecademy interactive lessons as an instructional supplement in a python programming course. In *Proceedings of the Fourth EDSIG Conference on Information Systems and Computing Education, At Norfolk, VA*. 2018.

[112] Valerie Shute. Focus on formative feedback. *Review of Educational Research*, 78:153–189, 03 2008.

[113] Richard Socher, Milind Ganjoo, Christopher D Manning, and Andrew Ng. Zero-shot learning through cross-modal transfer. In *Advances in neural information processing systems*, pages 935–943, 2013.

[114] Cynthia J Solomon and Seymour A Papert. A case study of a young child doing turtle graphics in logo. 1976.

[115] Jaime Spacco, Paul Denny, Brad Richards, David Babcock, David Hovemeyer, James Moscola, and Robert Duvall. Analyzing student work patterns using programming exercise data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 18–23, New York, NY, USA, 2015. ACM.

[116] Amanda Sullivan, Mollie Elkin, and Marina Umaschi Bers. Kibo robot demo: engaging young children in programming and engineering. In *Proceedings of the 14th international conference on interaction design and children*, pages 418–421, 2015.

[117] David Thompson and Tim Bell. Adoption of new CS high school standards by New Zealand teachers. In *Proc of the Workshop in Primary and Secondary Computing Education*, pages 87–90. ACM, 2013.

[118] Giovanni Maria Troiano, Sam Snodgrass, Erinç Argımak, Gregorio Robles, Gillian Smith, Michael Cassidy, Eli Tucker-Raymond, Gillian Puttick, and Casper Harteveld. Is my game ok dr. scratch?: Exploring programming and computational thinking development via metrics in student-designed serious games for stem. In *Proceedings of the 18th ACM International Conference on Interaction Design and Children*, pages 208–219. ACM, 2019.

[119] Sherry Turkle and Seymour Papert. Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, 11(1):3–33, 1992.

[120] Peter D. Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *J. Artif. Int. Res.*, 37(1):141–188, January 2010.

[121] Laurens Valk. *Lego mindstorms Ev3 Discovery Book: A beginner's guide to building and programming robots*. No Starch Press, 2014.

[122] Ashok Kumar Veerasamy, Daryl D'Souza, and Mikko-Jussi Laakso. Identifying novice student programming misconceptions and errors from summative assessments. *Journal of Educational Technology Systems*, 45(1):50–73, 2016.

[123] Amber Wagner, Jeff Gray, Jonathan Corley, and David Wolber. Using app inventor in a k-12 summer camp. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 621–626, 2013.

[124] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. Deep knowledge tracing on programming exercises. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, pages 201–204, New York, NY, USA, 2017. ACM.

[125] Christopher Watson and Frederick W.B. Li. Failure rates in introductory programming revisited. In *Proceedings of the Conference on Innovation & Technology in Computer Science Education*, ITiCSE, pages 39–44, 2014.

[126] David Weintrop and Uri Wilensky. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *Proceedings of the Eleventh*

Annual International Conference on International Computing Education Research, pages 101–110, 2015.

[127] David Weintrop and Uri Wilensky. Between a block and a typeface: Designing and evaluating hybrid programming environments. In *Proceedings of the 2017 Conference on Interaction Design and Children*, pages 183–192. ACM, 2017.

[128] Cameron Wilson. Hour of code: We can solve the diversity problem in computer science. *ACM Inroads*, 5(4):22–22, December 2014.

[129] Mark Windschitl and Kurt Sahl. Tracing teachers' use of technology in a laptop computer school: The interplay of teacher beliefs, social dynamics, and institutional culture. *American educational research journal*, 39(1):165–205, 2002.

[130] David Wolber. App inventor and real-world motivation. In *SIGCSE*, volume 11, pages 601–606, 2011.

[131] David Wolber, Hal Abelson, Ellen Spertus, and Liz Looney. *App Inventor*. O'Reilly Media, Inc., 2011.

[132] Mike Wu, Milan Mosse, Noah Goodman, and Chris Piech. Zero shot learning for code education: Rubric sampling with deep learning inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 782–790, 2019.

[133] Mor Friebroon Yesharim and Mordechai Ben-Ari. Teaching robotics concepts to elementary school children. In *International Conference on Robotics and Education RiE 2017*, pages 77–87. Springer, 2017.

Apéndice A

Interfaz de Mumuki

En este apéndice describimos Mumuki teniendo en cuenta la interfaz en base el usuario que lo este utilizando. Los usuarios de Mumuki pueden clasificar en aquellos que tienen permisos para resolver ejercicios, a los que llamaremos *estudiantes*, y aquellos que pueden crear cursos, definir ejercicios de programación, entre otras cosas, a los que llamaremos *docentes*. En la Sección A.1 se describe la interfaz de Mumuki desde el punto de vista del estudiante. En la Sección A.2 detallamos la interfaz de Mumuki teniendo en cuenta el rol del docente. Por último en la Sección A.3 describimos el conjunto de datos registrados por Mumuki que se usan para este el desarrollo de este trabajo.

A.1. Perspectiva del estudiante

A continuación describimos la experiencia que tiene el estudiante cuando aprender a programar utilizando Mumuki. Los ejercicios, como se puede ver en la Figura A.1, están distribuidos en capítulos, de forma similar a un libro de texto. En cada capítulo se introducen, de modo incremental, nuevos conceptos de programación con un lenguaje de programación específico. Cada capítulo se centra en un solo lenguaje de programación. Por ejemplo el capítulo *Fundamentos de programación* utiliza el lenguaje de programación Gobstones [68], introduciendo los conceptos de condicional, procedimiento, entro otros. En los capítulos siguientes introducen los paradigmas de programación, funcional, imperativo y orientado a objetos con sus conceptos propios. Cada institución educativa puede diseñar sus propios capítulos eligiendo ejercicios de una biblioteca de miles de ejercicios que brinda Mumuki o diseñando sus propios ejercicios.

Cada capítulo está compuesto por varias lecciones. Cada lección presenta un concepto nuevo o se integran conceptos nuevos a los presentados en las lecciones anteriores. En la Figura A.2 observamos la primera y la última lección disponible para el Capítulo 3, correspondientes al lenguaje de programación Haskell.[1]

Cada lección está compuesta por conjunto de ejercicios qué están diseñados para trabajar un concepto en particular. En general el título de la lección está directamente relacionado con el concepto a trabajar.

Como puede verse en la Figura A.2 cada ejercicio está precedido por un icono con posibles diferentes colores. Los mismos hacen referencia a la evaluación que Mumuki realiza sobre el último programa enviado por el estudiante para ese ejercicio como describimos en el Capítulo 5. Esto permite al estudiante tener una visión general de su avance dentro de cada lección.

[1] Puede ver la lista completa de lecciones para Haskell en https://mumuki.io/central/chapters/82-programacion-funcional

Figura A.1: Algunos de los capítulos disponibles que brinda Mumuki.

En la Figura A.3 observamos la interfaz del estudiante al momento de resolver un ejercicio. La interfaz del entorno incluye una barra de progreso, una sección de enunciado y un editor donde el estudiante debe escribir el programa. En el Capítulo 5 describimos la interfaz y feedback formativo generado por Mumuki para los ejercicios.

En la sección siguiente analizamos el entorno Mumuki desde la perspectiva del docente. Describimos las herramientas que se utilizan para construir los ejercicios, definir los casos de tests y las expectativas. Además presentamos herramientas para realizar un seguimiento de los estudiantes y reconstruir la trayectoria realizada por cada uno de ellos dentro del sistema a partir de los programas enviados.

A.2. Perspectiva del docente

Desde el punto de vista del usuario docente, Mumuki brinda dos funcionalidades llamadas **biblioteca** y **classroom**.

En la biblioteca el docente tiene el control sobre las guías de ejercicios. Esta funcionalidad es la que le permite diseñar las guías para trabajar diferentes conceptos. Cuando el docente crea un nuevo ejercicio, tiene que resolver dos problemas. Necesita escribir la descripción del ejercicio y con una herramienta que brinda Mumuki definir el feedback automático para los programas enviados en base a dos niveles: corrección y calidad. Para ser correctos, el docente debe definir un conjunto de test, los cuales son ejecutados luego de que el estudiante envía un programa. Los casos de tienen por objetivo comprobar que el programa enviado cumpla con el ejercicio propuesto en diferentes escenarios. Si bien la corrección de los programas no puede ser asegurada con este método, es lo suficientemente buena dada la complejidad de los ejercicios propuestos.

Para poder evaluar la calidad del programa enviado, el docente cuenta con un conjunto de características llamadas expectativas las cuales pueden ser seleccionadas como necesarias o no en el programa enviado por el estudiantes. Las expectativas permiten de algún modo mantener ciertos lineamiento en los programas enviados por los estudiantes. Por ejemplo, mediante una expectativa, se puede definir qué función se espera que un ejercicio reuse. O evitar ciertas características del código como comparaciones innecesarias de valores booleanos.

Figura A.2: Primera y última lección correspondientes al lenguaje de programación Haskell

Otra de las funcionalidades que provee Mumuki al docente es Classroom donde puede ver el progreso detallado de cada uno de los alumnos registrados en sus cursos tal cual se muestra en la Figura A.4.

Esta herramienta le permite ver al docente la cantidad de ejercicios intentados, la cantidad de programas enviados y la evaluación automática que realiza Mumuki apara cada uno de los programas enviados por el estudiante. El docente a cargo del curso puede ver todas los programas enviados por un estudiante en un sistema de versiones y de este modo puede reconstruir la trayectoria del estudiante para resolver un ejercicio puntual. En la Figura A.5 se muestra una secuencia de imágenes las cuales corresponden a un fragmento de los programas enviados por un usuario para el mismo ejercicio, el texto resaltado en verde indica una parte nueva en el programa, respecto de la programa previo mientras que lo resaltado en rojo es lo que se quitó respecto del programa anterior.

Esta secuencia de imágenes muestra parte del trayecto realizado por un estudiante para intentar completar el ejercicio “Recortar Tuits”, el cual se presentó en la Figura A.3. Los tres programas que se muestran en la Figura A.5 son consecutivas y todas tienen errores de tipo. Si nos enfocamos en el tiempo transcurrido entre los programas, podemos observar que la distancia es corta, lo cual nos da una idea que el estudiante se puede estar frustrando. Y más aún si tenemos en cuenta la diferencia entre los programas enviados es muy pequeña considerando la cantidad de caracteres modificados o eliminados entre programas. Notemos también que el estudiante envía un programa sin realizarle ningún cambio a pesar de que el programa previo no cumplía con todos los test. Luego de esto introduce algunos cambios de modo que el programa tiene más errores de tipos la primera de esta secuencia. Esta secuencia nos está mostrando que el estudiante no entiende como corregir su programa, generando de algún modo frustración y aumentando así la probabilidad de abandono del ejercicio.

Figura A.3: Ejercicio Mumuki desde la vista de usuario

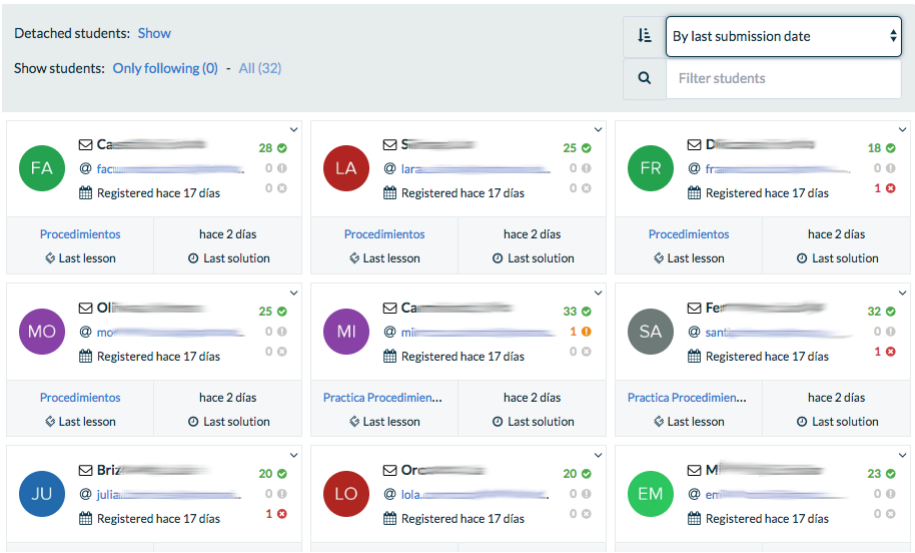


Figura A.4: Vista del curso desde la perspectiva del docente.

A.3. Conjunto de datos Mumuki

Cada vez que un estudiante presiona el botón enviar el programa enviado es evaluado por Mumuki y se guarda en la base de datos el programa enviado junto con información de la evaluación de los test unitarios, la evaluación de las expectativas y cierta información referida al usuario.

En la Figura A.6 se muestran los campos relevantes de un ejemplo correspondiente a un programa enviado para el ejercicio “Recortar tuits”-

La información relevante, para nuestro trabajo, se encuentra en los siguientes campos:

Figura A.5: Extracto del trayecto realizado por un estudiante antes de abandonar un ejercicio,

- **content**: Contenido escrito por el estudiante para el programa enviado.
- **created_at**: Fecha y hora en la cual el estudiante envió el programa.
- **status**: Resultado de la evaluación realizada por Mumuki, la misma se constituye de compilación, tests y comprobación de expectativas. El campo status de las programas enviados tienen cuatro opciones posibles, que se explicaron previamente en la Sección A.1: rojo oscuro, rojo claro, amarillo y verde.
- **exercise.name**: Nombre del ejercicio como lo ve el estudiante al resolverlo.
- **submission.count**: Número de programa enviado por el estudiante para ese ejercicio.
- **expectations_results**: Guarda los resultados de evaluar las expectativas definidas para el ejercicio.
- **test_results**: Guarda los resultados de los casos de test definidos para cada ejercicio, de forma similar en expectations results son guardadas las evaluaciones de las expectativas.

```
{
    "content" : "recortar :: [(String,String)] -> [(String,String)] \r\n recortar [] = [] \r\n
recortar ((x,y):xs) = (x,(take 15 y)):(recortar xs)",
    "status" : "Passed",
    "created_at" : "2016-04-17T02:32:26",
    "name" : "Recortar tuits",
    "guide.slug.name" : "mumuki-guia-funcional-listas",
    "submissions_count" : 1,
    "expectation_results" : [
        {
            "binding" : "take",
            "inspection" : "HasBinding",
            "result" : "passed"
        },
        {
            "binding" : "recortar",
            "inspection" : "HasTypeSignature",
            "result" : "passed"
        }
    ],
    "test_results" : [
        {
            "title" : "recortar [] == []",
            "status" : "passed",
        },
        {
            "title" : "recortar [(“@tuitero”, “Solo un tweet de prueba”)] == [(“@tuitero”, “Solo un
tweet d”)]",
            "status" : "passed",

        }
    ],
}
```


Figura A.6: Ejemplo de datos almacenados por el sistema luego de que un estudiante envía un programa.

Apéndice B

Ingeniería de características

En el capítulo 6 describimos algunas de las características definidas para entrenar modelos de regresión logística para evaluar el nivel de fluidez de un estudiante. En este apéndice presentamos todas las características definidas para este trabajo, ejemplificando el cálculo de cada una de ellas. En la Sección B.1 presentamos las características definidas para la dimensión estudiante. Luego, en la Sección B.2 describimos las características definida para la dimensión ejercicio.

B.1. Dimensión estudiante

Como describimos en el Capítulo 6 clasificamos las características de la dimensión estudiante en nivel de experiencia, nivel de abandono y nivel de insistencia. Para cada uno de estos niveles definimos y ejemplificamos cada una de las características.

Nivel de experiencia

El nivel de experiencia intenta capturar la experiencia en programación del estudiante en base a los programas enviados. Para ello definimos diferentes características que intentan capturar el nivel de experiencia del estudiante. Para poder ejemplificar y explicar la definición de cada una de las características, utilizaremos un conjunto de datos generado con ese objetivo. enviadas previamente definiendo diversas características. En la Tabla B.1 se muestra el conjunto de datos de ejemplo para calcular las características correspondientes para este nivel.

id	Fecha de envío	Estudiante	Ejercicio	Evaluación Mumuki
1	1/1/18 17:13:23	st1@mail.com	calcular	Error de caso de test
2	1/1/18 17:13:30	st1@mail.com	calcular	Programa correcto
3	2/1/18 17:16:23	st1@mail.com	esBisiesto	Programa correcto
4	4/1/18 17:20:20	st1@mail.com	pinos	Error de sintaxis
5	4/1/18 17:20:25	st1@mail.com	pinos	Error de caso de test

Tabla B.1: Conjunto de datos generado para ejemplificar y calcular cada una de las características correspondientes al nivel de experiencia

Promedio con aplazos (PCA): Para representar el nivel de prueba y error de un estudiante, definimos la característica PCA. Definimos promedio con aplazos como el total de ejercicios que intento resolver un estudiante sobre la cantidad total de programas enviados para los mismos. Un ejercicio se define como intentado por el estudiante, si el estudiante envió al menos un

programa para intentar resolver ese ejercicio. En la Tabla B.1, podemos ver que el estudiante intentó resolver 3 ejercicios diferentes para los cuales envió un total de 5 programas. Por lo tanto el valor de promedio con aplazos es el siguiente:

$$PCA(st1@mail.com) = \frac{3}{5} = 0{,}6$$

Promedio sin aplazos (PSA): Definimos promedio sin aplazos como la cantidad de ejercicios resueltos correctamente por un estudiante sobre la cantidad de programas enviados por el estudiante para resolver los mismos. Sean $\{e_1, .., e_n\}$ los ejercicios resueltos de forma **correcta** por el estudiante. Sean $\{s_1, ..., s_n\}$, las cantidades de soluciones que el estudiante necesitó para resolver cada ejercicio uno de los n ejercicios. Definimos promedio sin aplazos como:

$$PSA = \frac{n}{\sum(s_1, .., s_n)}$$

Como podemos ver en la Tabla B.1 el estudiante con email *st1@email.com* envió 5 programas. Los programas enviados están distribuidos para 3 ejercicios intentados: `esBisiesto`, `calcular` y `pinos`. Hay 2 ejercicios que se resolvieron correctamente (es decir que el programa enviado es sintácticamente correcto y pasa todos los casos de test): `calcular` y `esBisiesto`. Para resolver correctamente `calcular` el estudiante envió 2 programas. Para resolver `esBisiesto` solo le hizo falta enviar 1 programa. Por lo tanto para el estudiante `st1@email.com`, el cáculo de PSA sería

$$PSA(st1@mail.com) = \frac{2}{2+1} = 0{,}66$$

.

Notar que PSA siempre será un número entre 0 y 1 porque todo estudiante necesitará enviar al menos un programa para resolver de forma correcta un ejercicio. Si un estudiante, siempre resuelve los ejercicios de forma correcta en su primer intento, su PSA será 1.

Nivel de abandono

El nivel de abandono tiene como objetivo representar si un estudiante abandona frecuentemente los ejercicios. Además intenta modelar bajo qué condiciones decide abandonar un ejercicio. Para ello definimos la característica proporción de abandonos. Del mismo modo que hicimos con el nivel de experiencia luego de definir cada una de las características usaremos un conjunto de datos de muestra para ejemplificar el cálculo de cada una de ellas.

En la Tabla B.2 se muestra el conjunto de datos para ejemplificar y calcular la característica definida. Este pequeño conjunto de datos generado incluye a un estudiante que intentó resolver 3 ejercicios.

A continuación presentamos las diferentes características que permiten capturar el nivel de abandono de los estudiantes.

Proporción de programas abandonados (PA): representamos el historial de abandono de un estudiante con la característica PA. PA es el total de los programas anotados como abandono para un estudiante sobre el total de programas enviados por el estudiante.

La Tabla B.2 muestra los programas enviados por el estudiante *st1@mail.com.* El estudiante envió en total ocho programas, de los cuales 4 están anotados como abandono. Por lo tanto, el valor de PA para el estudiante es

$$PA(st1@mail.com) = \frac{4}{8} = 0{,}5$$

Proporción de ejercicios abandonados (EA): un ejercicio es considerado abandonado por un estudiante si al menos un programa enviado por el estudiante para resolver el ejercicio

id	Fecha de envío	Estudiante	Ejercicio	Evaluación Mumuki	Abandono
1	1/1/18 17:13:23	st1@mail.com	calcular	Error en caso de test	No
2	1/1/18 17:13:30	st1@mail.com	calcular	Programa correcto	No
3	2/1/18 17:16:23	st1@mail.com	esBisiesto	Error de sintaxis	Si
4	2/1/18 17:16:30	st1@mail.com	esBisiesto	Error de sintaxis	Si
5	2/1/18 17:20:20	st1@mail.com	Pinos	Error de sintaxis	Si
6	2/1/18 17:20:25	st1@mail.com	Pinos	Error en caso de test	Si
7	3/1/18 17:20:25	st1@mail.com	Pinos	Error en caso de test	No
8	3/1/18 17:20:30	st1@mail.com	Pinos	Programa correcto	No

Tabla B.2: Conjunto de datos generado para ejemplificar y calcular cada una de las características correspondientes al nivel de abandono. En la columna Fecha de envío se informa hora y fecha en la cual el programa fue enviado. En las columnas Estudiante y Ejercicio se informa el nombre del estudiante y ejercicio para el cual el programa fue enviado. En la columna Evaluación Mumuki se informa el estado del programa enviado por el estudiante en base a la evaluación realizada por Mumuki. En la columna Abandono se informa si el programa enviado fue anotado como abandono en base a la definición que presentamos en el Capítulo 6

considera un ejercicio es anotado como abandono. EA es calculado como la cantidad total de ejercicios considerados como abandonados sobre la cantidad de ejercicios intentados por un estudiante. A diferencia de la característica anterior, esta característica se calcula a nivel de ejercicio y no de programas.

Como se puede ver en la Tabla B.2 el estudiante intentó tres ejercicios `calcular`, `esBisiesto` y `Pinos`. Los ejercicios `esBisiesto` y `Pinos` tienen al menos una programa anotado como abandono, por lo tanto estos dos ejercicios son considerados como abandonados. No es relevante que en próximos programas el ejercicio sea resuelto como el caso de `Pinos`. Notar que la línea que separa los programas 6 y 7 está denotando un cambio de sesión. Por lo tanto la característica de cantidad de ejercicios abandonados se calcula de la siguiente manera,

$$EA(st1@mail.com) = \frac{2}{3} = 0{,}66$$

Nivel de insistencia

A través del nivel de insistencia queremos capturar la intensidad del estudiante al momento de enviar programas al no poder resolver un ejercicio puntual. Definimos tres características para intentar capturar el nivel de insistencia de un estudiante: promedio de tiempo transcurrido entre soluciones consecutivas, promedio de distancia de Levenshtein [81]. Tabla **??** se muestra el conjunto de datos para ejemplificar y calcular las características definidas.

Promedio de tiempo transcurrido (PTT): con el objetivo de medir con qué frecuencia el estudiante envía sus programas dentro de una sesión, calcularemos el promedio de tiempo transcurrido entre programas consecutivos (PTT) dentro de todas las sesiones llevadas a cabo por el estudiante, sin considerar el tiempo de inactividad, es decir el tiempo entre sesiones.

En la Tabla B.3 la columna `Dist. en segundos` corresponde al tiempo transcurrido del programa enviado con el programa anterior, cuyo valor está expresado en segundos. Por lo tanto el promedio de tiempo transcurrido entre programas enviados consecutivos para este estudiante es

$$PTT = (\frac{7 + 2 + 5 + 230 + 5 + 5 + 20 + 10}{8}) = 35,5$$

id	Fecha	Dist. en segundos	Dist. de Leveshtein	Estudiante	Ejercicio	# Prog. previos	Evaluación Mumuki
1	1/1/18 17:13:23	98293	24	st1@mail.com	calcular	1	Error en caso de test
2	1/1/18 17:13:30	7	5	st1@mail.com	calcular	2	Programa correcto
3	2/1/18 17:16:23	86573	80	st1@mail.com	esBisiesto	1	Error de sintaxis
4	2/1/18 17:16:25	2	4	st1@mail.com	esBisiesto	2	Error de sintaxis
5	2/1/18 17:16:30	5	2	st1@mail.com	esBisiesto	3	Error de sintaxis
6	2/1/18 17:20:20	230	40	st1@mail.com	pinos	1	Error de sintaxis
7	2/1/18 17:20:25	5	2	st1@mail.com	pinos	2	Error en caso de test
8	3/1/18 17:20:25	86400	1	st1@mail.com	pinos	3	Error en caso de test
9	3/1/18 17:20:30	5	2	st1@mail.com	pinos	4	Error de sintaxis
10	3/1/18 17:20:50	20	20	st1@mail.com	pinos	5	Error en caso de test
11	3/1/18 17:21:00	10	2	st1@mail.com	pinos	6	Programa correcto

Tabla B.3: Conjunto de datos generado para ejemplificar y calcular cada una de las características correspondientes al nivel de insistencia. En la columna Fecha se informa hora y fecha en la cual el programa fue enviado. En la las columnas Dist. en segundos y Dist. de Leveshtein se informa el tiempo transcurrido en segundos y la distancia de Leveshtein entre el programa enviado y al anterior. En las columnas Estudiante y Ejercicio se informa el nombre del estudiante y ejercicio para el cual el programa fue enviado. En la columna #Prog. previos se informa la cantidad de programas previos que envió el estudiante para resolver el ejercicio. En la columna Evaluación Mumuki se informa el estado del programa enviado por el estudiante en base a la evaluación realizada por Mumuki.

Notar que para el cálculo de PTT no se incluye el tiempo correspondiente a los programas con id 1, 3 y 8. Son las primeros programas de cada sesión, y el tiempo que se observa es el tiempo transcurrido entre sesiones. Tiempo en el que el estudiante no estuvo trabajando en Mumuki. El denominador de este promedio será la cantidad de programas enviados menos la cantidad de sesiones. En este caso el estudiante envió 11 programas en 3 sesiones, por lo tanto el denominador es 8.

Promedio distancia de Levenshtein (PDL): intentando capturar que tanto el estudiante diseña el programa antes enviarlo, calculamos el promedio distancia de Levenshtein [81] entre el contenido de los programas sobre un mismo ejercicio, para todas las sesiones realizadas por el estudiante.

La columna Dist. de Levenshtein en la Tabla B.3 muestra la distancia de Levenshtein [81] entre programas consecutivos del mismo ejercicio. En las primeros programas enviados para cada ejercicio (id 1, 3 y 5) la distancia de Levenshtein se calcula contra un programa vacío lo cual permite ver que tan grande es el primer programa enviado. Para el resto de los programas de un mismo ejercicio la comparación se realiza con el programa anterior. De esta forma el valor de PDL se calcula de la siguiente forma,

$$PDL(st1@mail.com) = (\frac{24 + 5 + 80 + 4 + 2 + 40 + 2 + 1 + 2 + 20 + 2}{11}) = 16,54$$

B.2. Dimensión ejercicio

En esta dimensión intentamos representar que tan difícil es cada uno de los ejercicios a partir del desempeño de los estudiantes en los mismos. Para ello se definen cuatro características, promedio de cantidad de soluciones para aprobar (PCSA), abandonos por estudiante (APE) y cantidad de abandonos por ejercicio (CAPE) y completitud (COMP). Para ejemplificar las definiciones y facilitar la comprensión de cada una de las características usaremos un extracto del conjunto de datos correspondiente al ejercicio `calcular` el cual se puede ver en la Tabla B.4.

Estudiante	Ejercicio	Evaluación Mumuki	Abandono	id
e1@mail.com	calcular	Error de caso de test	Si	1
e1@mail.com	calcular	Error de caso de test	Si	1
e2@mail.com	calcular	Error de caso de test	No	1
e2@mail.com	calcular	Programa correcto	No	1
e3@mail.com	calcular	Programa correcto	No	1
e4@mail.com	calcular	Error de sintaxis	Si	1
e4@mail.com	calcular	Error de sintaxis	Si	1
e4@mail.com	calcular	Error de sintaxis	Si	1

Tabla B.4: Soluciones enviadas correspondientes al ejercicio *calcular* para ejemplificar el cálculo de las métricas correspondiente a la dificultad del ejercicio

- **Promedio de cantidad soluciones para aprobar (PCSA):** con esta característica intentamos identificar la cantidad de soluciones necesarias para completar con éxito cada ejercicio. Para ello se consideran todas las trayectorias que finalizaron con el ejercicio con un programa correcto y se calculará el promedio de la cantidad de programas enviados para lograr resolver ese ejercicio.

En la Tabla B.4 podemos ver que los estudiantes *e2@mail.com* y *e3@mail.com* son quienes completaron el ejercicio `calcular` de modo correcto, necesitando dos y un programa enviado respectivamente. Por lo tanto el promedio de cantidad soluciones para aprobar del ejercicio `calcular` queda de la siguiente forma,

$$PCSA(calcular) = \frac{2+1}{2} = 1{,}5$$

- **Abandonos por estudiante (APE):** una forma de ver la dificultad del ejercicio puede ser viendo cuántos programas envían los estudiantes antes de abandonarlo. Calculamos la proporción entre el número de estudiantes que abandonaron el ejercicio sobre la cantidad de programas enviados anotados como abandono. Esta característica intenta modelar cuantas soluciones en promedio envía un estudiante hasta que decide abandonarlo.

  Para el ejercicio que estamos usando de ejemplo sólo dos estudiantes lo abandonaron, el estudiante `e1@mail.com` quien envió dos programas antes de abandonarlo definitivamente y el estudiante `e4@mail.com` quien envió tres programas antes de abandonarlo. Habiendo observado estos datos podemos calcular la característica de abandonos por estudiante de la siguiente forma,

$$APE(calcular) = \frac{2+3}{2} = 2{,}5$$

- **Cantidad de abandonos por ejercicio (CAPE):** en esta característica nos interesa analizar la distribución de los programas enviados para el ejercicio. En particular nos interesa ver la proporción de programas enviados anotados como abandono sobre cantidad de programas enviados para el ejercicio. De este modo si esa proporción está cerca de 1 implica que sería un ejercicio que ha sido abandonado frecuentemente.

  Como se puede ver en la Tabla B.4 se registraron ocho soluciones para el ejercicio `calcular` de las cuales cinco están anotadas como abandono, por lo tanto la característica CAPE se calcula de la siguiente forma,

$$CAPE(calcular) = \frac{5}{8} = 0{,}625$$

- **Completitud (COMP):**, con esta característica nos interesa modelar cuántos estudiantes pudieron completar el ejercicio. Para ello se calcula la proporción de estudiantes que completaron con éxito el ejercicio, es decir tiene al menos una solución en verde para ese ejercicio, sobre la cantidad estudiantes que intentaron el ejercicio.

  Como vimos previamente, sólo dos de cuatro estudiantes que intentaron el ejercicio pudieron completarlo satisfactoriamente. Considerando eso podemos calcular la característica completitud de la siguiente manera,

$$COMP(calcular) = \frac{2}{4} = 0{,}5$$

Apéndice C

Publicaciones internacionales

En este apéndice incluimos la primer hoja de cada una de las siguientes 6 publicaciones internacionales que introdujeron el Capítulo 1.

- María Cecilia Martínez, Marcos J. Gómez, Luciana Benotti. *A Comparison of Preschool and Elementary School Children Learning Computer Science Concepts through a Multilanguage Robot Programming Platform*. Proceedings of the 20th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITICSE 2015). **Número de citas: 35**.

- M. Cecilia Martinez, Marcos J. Gómez, Marco Moresi y Luciana Benotti. *Lessons Learned on Computer Science Teachers Professional Developmen*t. Proceedings of the 21th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITICSE 2016). **Número de citas: 14**.

- Luciana Benotti, Marcos J. Gómez, and M. Cecilia Martinez. *UNC++Duino: A kit for learning to program robots in Python and C++ starting from blocks*. Robotics in Education (pp. 181-192). **Número de citas: 5**.

- Allan Fowler, Johanna Pirker, Ian Pollock, Bruno Campagnola de Paula, Maria Emilia Echeveste, and Marcos J. Gómez. 2016. *Understanding the benefits of game jams: Exploring the potential for engaging young learners in STEM*. In Proceedings of the 2016 ITiCSE Working Group Reports (ITiCSE '16). **Número de citas: 13**.

- Luciana Benotti, Federico Aloi, Franco Bulgarelli, and Marcos J. Gomez. *The Effect of a Web-based Coding Tool with Automatic Feedback on Students' Performance and Perceptions*. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18). **Número de citas: 14**.

- Marcos J. Gómez, Marco Moresi y Luciana Benotti. *Text-based programming in elementary school: A comparative study of programming abilities in children with and without block-based experience*. Proceedings of the 24th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITICSE 2019). Association for Computing Machinery. **Número de citas: 0**.

A Comparison of Preschool and Elementary School Children Learning Computer Science Concepts through a Multilanguage Robot Programming Platform

Cecilia Martínez
Facultad de Filosofía y Humanidades
Universidad Nacional de Córdoba/CONICET
Córdoba, Argentina
cecimart@gmail.com

Marcos J. Gómez
FAMAF, Universidad Nacional de Córdoba
Córdoba, Argentina
mgomez4@famaf.unc.edu.ar

Luciana Benotti
Logic, Interaction and Intelligent Systems Group
FAMAF, Universidad Nacional de Córdoba/CONICET
Córdoba, Argentina
benotti@famaf.unc.edu.ar

ABSTRACT

This paper describes a school intervention to teach fundamental Computer Science (CS) concepts to 3-11 year old students with a multilanguage robot programming platform (using drag and drop, Python and C++ languages) in Argentina. We analyze students' performance and learning process based on multiple choice test and classroom observations. Data show that all students can intuitively learn sequence, conditional, loops and parameters and that girls performed slightly better than boys. Older students can easily combine these concepts to write a program. The multilanguage platform promotes student spontaneous exploration of more sophisticated CS concepts and languages. These findings imply that introducing CS in mandatory schooling from an inquiry based approach is both achievable and beneficial.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms

Education

Keywords

Computer science K-7 outreach, robots, experimental evaluation, iconic programming language

1. INTRODUCTION

There has been a lot of debate on whether preschool and elementary school children should use computers or are developmentally ready to learn programming [3]. On one side, children are using computers much earlier each decade [15].

On the other side, this intensive use that many children have of computers may not be contributing to early access of Computer Science as a discipline (CS) which includes notions of creating and developing technology, programming, designing, and automatizing actions. We could argue that children become software consumers very early but they do not learn some basics of how this technology works. Bergen [1] points out that re-programmable toys such as robots, give children the possibility of creating, imagining, programming and exploring rather than developing procedural digital competences.

Previous research suggests [3] that early introduction of some basic CS concepts benefits both children cognitive development and learning about CS. Nevertheless, we still are debating what kind of CS concepts should be taught in preschool and elementary school [16]. Some countries such as Estonia and the UK have recently introduced CS in these levels [6], but most others—including ours—are still deliberating when would it be appropriate to introduce CS in schools and what content is most suitable for the general basic education. While research on teaching CS in different educational levels is vast, and colleagues have proposed curricular designs [10, 4], comparisons among different age groups performance that inform curriculum selection and scope is not as common [6, 12].

With the purpose of both investigating how children learn basic CS concepts in schools using programmable toys and contributing to a CS curriculum selection and scope; we designed an exploratory study to compare how preschool children (ages 3 and 5), and elementary school children (ages 8 to 11) learn some basics CS concepts. We piloted CS lessons in a real school setting focusing on loops, variables, conditionals, sequence, and parameters; and their application to robot programming. We analyzed children' learning of CS using a multilanguage robot programming platform and compared boys and girls performance. The main contributions of this paper are: 1) Analyzing how different age group of children learn fundamental CS concepts. 2) Introducing a multilanguage robot programming platform that permits students to discover new CS concepts on their own, growing with the platform. 3) Evaluating gender and age

ITiCSE'15, July 6–8, 2015, Vilnius, Lithuania.

ACM 978-1-4503-3440-2/15/07 ...$15.00.
http://dx.doi.org/10.1145/2729094.2742599.

UNC++Duino: A kit for learning to program robots in Python and C++ starting from blocks

Luciana Benotti, Marcos J. Gómez, and Cecilia Martínez

Universidad Nacional de Córdoba,
Haya de la Torre y Medina Allende, Córdoba, Argentina
firstname.lastname@unc.edu.ar
http://masmas.unc.edu.ar

Abstract. We present UNC++Duino, an open source educative software for learning to program a robotic kit in C++ and Python. Besides of these two industry programming languages, UNC++Duino can be programmed using 2 high level languages based on blocks are free of syntax errors. One of the block based languages included is completely iconic allowing for its use with preliterate children. The hardware we use with UNC++Duino, the open RobotGroup robotic kit, can be used to build different automated constructions based on an Arduino board, sensors and actuators. UNC++Duino was developed within Argentinean K-12 schools by the Universidad Nacional de Córdoba with the collaboration and support of the Argentinean National Ministry of Science and the RISE program in Google for Education. Its goal is to provide an engaging tool for learning to program in different programming languages with increasing difficulty and control of the hardware.

1 Introduction

Why is it necessary to find effective and innovative ways of engaging more students into Computer Science (CS)? Taking our country as an example, we know that Argentinean universities graduate approximately 4000 CS students a year (compared to 10000 in Law and 15000 in Economics) while the national industry needs to hire twice that amount per year. The lack of human resources in CS has an economical impact in Argentina. The Information and Communication Technology (ICT) industry in Argentina, despite having grown intensely in the last ten years (four times in the number of employees, and nine times in the amount of exports), is still struggling to find qualified workers for its workforce.

Part of the reason why students do not choose to pursue a career in CS is that, neither programming, nor other CS techniques and concepts, are taught at school. Previous studies show that this lack of early CS education can influence career choices; students may not be selecting CS simply because they do not know what CS is [4,10]. Since it is not taught at school, misconceptions about what the discipline actually is are commonplace. Indeed, recent work found that although more than 90% of Argentinean high school students surveyed use computers, most of them believe that programming means installing programs [2].

Lessons Learned on Computer Science Teachers Professional Development

M. Cecilia Martínez
Centro de Investigaciones FFyH/CONICET, Universidad Nacional de Córdoba
Córdoba, Argentina
cecimart@gmail.com

Marcos J. Gómez
Logic, Interaction and Intelligent Systems Group
FAMAF, Universidad Nacional de Córdoba
Córdoba, Argentina
mgomez4@famaf.unc.edu.ar

Marco Moresi
Computer Science Section
FAMAF, Universidad Nacional de Córdoba
Córdoba, Argentina
mrc.moresi@gmail.com

Luciana Benotti
Logic, Interaction and Intelligent Systems Group
FAMAF, Universidad Nacional de Córdoba/CONICET
Córdoba, Argentina
benotti@famaf.unc.edu.ar

ABSTRACT

This paper describes an introductory Computer Science (CS) Professional Development (PD) course for K-12 teachers in Argentina that integrates pedagogical content knowledge and teacher classroom practice. We analyzed teachers' learning of what CS entails and the implementation of inquiry-based programming lessons in their schools. Based on pre and post teachers surveys and classroom observations, we found that most teachers learned about the CS object of study and about fundamental programming concepts such as conditionals, loops, variables, etc. Teachers were more likely to replicate the same activities they experienced during PD workshops in their classrooms than to produce their own. Teachers who had a previous background on CS provided in-depth explanations of CS concepts to their students while other teachers superficially introduced the content knowledge. We describe PD activities and characteristics that could explain teachers' learning and incorporation of programming lessons. Findings imply that a PD program that integrates pedagogical content knowledge and teachers classroom practice can effectively improve inquiry-based CS teaching, but may be insufficient preparation for teachers with no previous background on CS.

CCS Concepts

•**Social and professional topics** → **K-12 education;**

ITiCSE '16, July 09-13, 2016, Arequipa, Peru

DOI: http://dx.doi.org/10.1145/2899415.2899460

Keywords

Computer science K-12 outreach, experimental evaluation, Computer Science (CS) Professional Development (PD)

1. INTRODUCTION

As many countries are moving forward in their efforts to introduce Computer Science (CS) massively into the mandatory school curriculum, one topic of debate between academics, policy makers and the whole educational community is who is going to teach CS in schools and how are teachers going to be prepared. Currently, one of the major challenges for teaching CS is the lack of teacher subject knowledge [8].

While most researchers and policymakers agree that teacher certification degrees offered by Universities are the best option to prepare highly qualified teachers [13], some countries are considering training teachers "on the job" (such as the UK, New Zealand, the US and Germany) [14, 7, 11, 4]. These countries face the dilemma to move forward with their CS curriculum reform whit a shortage of CS professionals willing to move into a teaching path. Hence, they opt for in service training or short term Professional Development (PD). In Argentina, for the last five years members of the National Secretary of Science and Productive Innovation together with National Universities have been promoting the teaching of CS, and in particular CS programming, in mandatory education.

Researchers have documented that effective PD programs, that include an explicit focus on the subject matter and analyze student thinking, can promote teacher learning [6]. Requiring teacher active learning through workshops or study groups within a coherent program where learning goals, content and activities are aligned, has significant effects on positively changing teacher learning [6]. When it comes to teacher PD in technology, first hand experiences with technology, combined with observation and analysis of other teachers using technology are effective strategies to change teachers beliefs and practices about technology [15]. While

Understanding the benefits of game jams

Exploring the potential for engaging young learners in STEM

Allan Fowler
Kennesaw State University
Marietta, GA
USA
afowle56@kennesaw.edu.edu

Johanna Pirker
Graz University of Technology
Graz
Austria
jpirker@iicm.edu

Ian Pollock
California State University
East Bay
Hayward, CA, USA
ian.pollock@csueastbay.edu

Bruno Campagnola de Paula
Pontifical Catholic University of Parana
Brazil
bruno.paula@pucpr.br

Maria Emilia Echeveste
Universidad Nacional de Córdoba
Córdoba
Argentina
meecheveste@gmail.com

Marcos J. Gómez
Universidad Nacional de Córdoba
Córdoba
Argentina
mgomez4@famaf.unc.edu.ar

ABSTRACT

There is a wide range of implementations of game jams throughout the world . Game jams have been organized in a number of different formats, themes, and timeframes [43]. What they all have in common is the opportunity for participants to make a game within a specified constraint such as time, location, technology, or theme. Additionally, game jams as social experience support active and collaborative learning formats. In this paper, we discuss the potential of game jams for young learners, describe successful jam events in this context, and provide a list of tools useful for organizing game jams for this target group.

CCS Concepts

•Social and professional topics → Computer science education;

Keywords

Game Jams, Game Design, Game Development, Programming.

1. INTRODUCTION

Game Jams to introduce young learners to computer science concepts in a fun and engaging way have the potential to introduce computer science principles. Introducing these concepts at an early age in an entertaining way, it is possible to influence or improve perceptions of computer science as a career.

Working with an easy to use game development tool, Fowler and Cusack [39] report that participants found the experience fun and engaging. Using tools such as Kodu[1], Scratch[2], GameMaker[3], and Greenfoot[4] young learners discover computer programming concepts using visual programming environments [40].

According to Fowler, Fristoe, and MacLaurin [40], Kodu is entirely event driven. Programming involves the placement of tiles in a meaningful sequence to form a condition and action based on each rule. As a result, if the user does not enter the correct code, the system will still run, but the actions will not be performed, thus reducing some of the frustrations commonly experienced by novice learners. Fristoe et al. [46] and Jones [58] also found that Kodu and similar tools are effective tools to introduce young students to some foundation programming concepts.

The lack of diversity and underrepresented minorities in STEM careers has been discussed and debated in the literature [19, 53, 74, 100]. While the representation of underrepresented minorities (URM) in STEM subjects is improving, there is still some progress to be made in the computer sciences. Through introducing CS principles through game jams, the authors intend to address the imbalance of URM through introducing a game jam and development program in both formal and informal learning environments.

2. GAME JAMS

In the literature several definitions for game jams can be found [63]. Most game jam events share similar characteristics and a process flow. In the next section those elements are listed and briefly described.

2.1 Characteristics of Game Jams

Different characteristics to describe game jams [41, 42]. The most common ones are listed below.

- *Social:* The participants are encouraged to work in small teams (2-5) to brainstorm ideas and to develop their games.

ITiCSE '16 Arequipa, Peru

DOI: http://dx.doi.org/10.1145/3024906.3024913

[1]http://www.kodugamelab.com/
[2]https://scratch.mit.edu/
[3]http://www.yoyogames.com/studio
[4]http://greenfoot.org

The Effect of a Web-based Coding Tool with Automatic Feedback on Students' Performance and Perceptions

Luciana Benotti
Universidad Nacional de Cordoba / CONICET
Cordoba, Argentina
benotti@famaf.unc.edu.ar

Federico Aloi
Universidad Nacional de Quilmes
Buenos Aires, Argentina
federico.aloi@unq.edu.ar

Franco Bulgarelli
Mumuki Project
Buenos Aires, Argentina
franco@mumuki.org

Marcos J. Gomez
Universidad Nacional de Cordoba
Cordoba, Argentina
mgomez4@famaf.unc.edu.ar

ABSTRACT

In this paper we do three things. First, we describe a web-based coding tool that is open-source, publicly available and provides formative feedback and assessment. Second, we compare several metrics on student performance in courses that use the tool versus courses that do not use it when learning to program in Haskell. We find that the dropout rates are significantly lower in those courses that use the tool at two different universities. Finally we apply the technology acceptance model to analyse students perceptions.

CCS CONCEPTS

• **Language Classifications** → **Applicative (functional) languages**; • **Computers and Education** → **Computer-assisted instruction**;

KEYWORDS

Functional programming; Haskell; Coding tools; Replication; Automatic assessment

ACM Reference format:
Luciana Benotti, Federico Aloi, Franco Bulgarelli, and Marcos J. Gomez. 2018. The Effect of a Web-based Coding Tool with Automatic Feedback on Students' Performance and Perceptions. In *Proceedings of The 49th ACM Technical Symposium on Computer Science Education, Baltimore, MD, USA, February 21–24, 2018 (SIGCSE '18),* 6 pages.
https://doi.org/10.1145/3159450.3159579

1 INTRODUCTION

While many tools are developed in order to support the teaching and learning of programming, few such tools ever achieve widespread adoption and use. A survey [2] found that one of the most common reasons is the teachers' unwillingness to adopt a new system until its usefulness has been demonstrated.

In general, the evaluations of such tools use ad-hoc metrics and are conducted on a single course (for notable exceptions see [8]). As argued in [8], there is a critical need for replicating studies in different contexts and for evaluating various contributing factors. It is also necessary to use standard metrics in order to better understand the reasons why certain results occur. In this paper we hypothesize that coding tools may have an effect on the high dropout rates observed in introductory programming courses [15].

The main contributions of this paper are:

- We describe a web-based coding tool that is open-source and provides formative feedback and assessment.
- We compare several metrics on student performance in courses that use the tool versus courses that do not use it when learning to program in Haskell.
- We find that the dropout rates are significantly lower in those courses that use the tool at two different universities.
- We apply a standard model, the technology acceptance model (TAM) to analyse students perceptions.

According to a recent review [9], less than 10% of the educational programming tools support functional programming. This paper focuses on the language Haskell, a traditional functional programming language that has gained popularity in recent years.

We begin the paper by reviewing previous work on educational coding tools for functional programming and on previous studies on the effect of such tools on students' performance and perceptions. Then, we describe the online coding tool that we use, called Mumuki, and its rationale. A screenshot can be seen in Figure 1. Moreover, we address the study design followed by our findings. We analyse student passing, failure and dropout rates. Finally, we discuss our findings when applying the technology acceptance model on the courses.

2 PREVIOUS WORK

The topic of online coding tools has received plenty of attention during the last years. Well known examples are Codingbat [13], CodeRunner [12], CodeLab [1] and CloudCoder [14]. These online tools have been shown to be useful helping students' master the syntax of imperative programming languages such as Java and Python [8]. In spite of the attention this topic has received from researchers, the use of online coding tools is not widespread in education [2]. There are studies [8] that use these tools in order to identify students at-risk of dropout. However, longitudinal studies,

SIGCSE '18, February 21–24, 2018, Baltimore, MD, USA

ACM ISBN 978-1-4503-5103-4/18/02…$15.00
https://doi.org/10.1145/3159450.3159579

Text-based Programming in Elementary School: A Comparative Study of Programming Abilities in Children with and without Block-based Experience

Marcos J. Gomez
Universidad Nacional de Cordoba
Cordoba, Argentina
mgomez4@famaf.unc.edu.ar

Marco Moresi
Universidad Nacional de Cordoba
Cordoba, Argentina
mrc.moresi@gmail.com

Luciana Benotti
Universidad Nacional de Cordoba / CONICET
Cordoba, Argentina
benotti@famaf.unc.edu.ar

ABSTRACT

This paper describes an elementary school intervention to teach a text-based programming language to 10-11 year old students. We compare students with no previous programming experience with students with 3 semesters of experience with a block-based programming language. We analyze students' performance and learning based on detailed logs in an online programming platform and on multiple choice tests. Although both groups have a similar percentage of syntactical errors, the experienced group showed a better performance on exam scores and a lower number of test case errors. These findings suggest that, 10-11 year old students benefit from block-based experience when learning a new text-based programming language.

CCS CONCEPTS

• **Social and professional topics** → **K-12 education**; • **Applied computing** → *Computer-assisted instruction*; • **Software and its engineering** → *Imperative languages.*

KEYWORDS

Learning analytics; elementary education; text-based languages; block-based languages

ACM Reference Format:
Marcos J. Gomez, Marco Moresi, and Luciana Benotti. 2019. Text-based Programming in Elementary School: A Comparative Study of Programming Abilities in Children with and without Block-based Experience. In *Innovation and Technology in Computer Science Education (ITiCSE '19), July 15–17, 2019, Aberdeen, Scotland UK*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3304221.3319734

1 INTRODUCTION

In the last years, governments, universities, companies and organizations around the world have joined forces to bring the teaching of Computer Science (CS) at elementary, middle and high schools [11, 14, 22, 29]. In this context, there is still an ongoing debate on whether elementary school children are developmentally ready to learn programming or even to use computers (see for example [6]).

While research on teaching programming in higher educational levels is vast, and there is increasing consensus on curricular designs (e.g., [15]), studies over elementary schools are not as common [8]. Research based on teaching CS in elementary school can be classified (according to [2]) in three categories: unplugged activities [3], block-based visual programming [28] and robotics [5, 21]. Most of the documented experiences use block-based platforms such as Scratch [24, 28], Snap! [16], Alice [9], Code.org [17]. Block-based languages are supposed to allow students to focus on concepts, leaving aside the syntactic complexity present when learning to program using a text-based language. But, how well can elementary school students transition from block-based languages to text-based languages? In other words, our research question Q1: *do children with block-based experience have programming abilities that make it easier to learn a text-based language*?

In order to address this question we pose two non directional hypotheses. Ha is related to syntactic errors, Hb is related to semantic errors, defined as follows. A student makes a syntactic error when her/his program does not comply with the grammar of the programming language. A student makes a semantic error when she/he holds a misconception about the program behavior. In Section 3 we formally define these two kinds of errors.

Ha (null): There is no significant difference in the amount of syntactic errors that students with and without previous block-based programming experience make.

Hb (null): There is no significant difference in the amount of semantic errors that students with and without previous block-based programming experience make.

With the purpose of testing these hypotheses and contributing to a CS curriculum selection and scope, we designed an observational study for elementary school children. We carried out programming lessons in a real school setting focusing on loops, conditionals, sequences, and parameters; and their application to programming. The main contributions of this paper are three. First, we introduce an online coding tool that logs an student learning process while programming, registering each submission made by the student along with formative feedback automatically generated by the tool. Second, we analyze comparatively how 10-11 year old children with and without previous programming experience create programs in a text-based programming language. Finally we evaluate the

ITiCSE '19, July 15–17, 2019, Aberdeen, Scotland UK

ACM ISBN 978-1-4503-6301-3/19/07…$15.00
https://doi.org/10.1145/3304221.3319734