UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

FACULTAD DE
INGENIERÍA
UDELAR

Un modelo de investigación en didáctica de la programación

Federico Gómez

Tesis de Maestría presentada a la Facultad de Ingeniería de la Universidad de la República en cumplimiento parcial de los requerimientos para la obtención del título de Magíster en Informática.

Tutora

Dra. Sylvia da Rosa

Tribunal

Dra. Aiala Rosá
Dr. Crediné Silva de Menezes
Dr. Javier Blanco (revisor)

Montevideo, Uruguay
Noviembre de 2021

Un modelo de investigación en didáctica de la programación

Autor:
Federico Gómez
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
fgfrois@fing.edu.uy
Noviembre 2021

Supervisora y orientadora:
Dra. Sylvia da Rosa
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
darosa@fing.edu.uy

Revisor:
Dr. Javier Blanco
Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba
Córdoba, Argentina
blanco@famaf.unc.edu.ar

Resumen

En este trabajo se presenta un modelo de investigación en didáctica de la programación. La didáctica es un área específica enmarcada dentro de la ciencia de la computación, que estudia en profundidad temas relativos a educación en la disciplina, con fundamentos epistemológicos.

El marco teórico es la *Epistemología Genética*, que explica la construcción de conocimiento científico, a partir de, por un lado, estudios empíricos sobre la construcción de conceptos a nivel *psicogenético*, realizados por Piaget y colaboradores a lo largo de muchos años, y por el otro, de un *análisis histórico crítico de la evolución de las teorías científicas,* realizado por Piaget y García a partir de la sociogénesis de la ciencia. El modelo surge de la aplicación de principios de la teoría, especialmente la tríada de etapas *intra-inter-trans*, al campo concreto de conocimiento relativo a *algoritmos*, *estructuras de datos* y *programas*. Explica el proceso completo de construcción de conocimiento sobre algoritmos y estructuras de datos (pasaje de la etapa *intra* a la etapa *inter*), los *programas* que los implementan usando lenguajes de programación y su ejecución en computadora (pasaje de la etapa *inter* a la etapa *trans*). Para ilustrar cómo el modelo explica el proceso, se presenta un estudio empírico detallado sobre un problema de búsqueda, el algoritmo que lo resuelve y el correspondiente programa, en dos lenguajes de programación imperativos. El estudio fue llevado a cabo con estudiantes de un curso inicial de programación de una carrera de Ingeniería en Computación. Se incluyen extractos de entrevistas a los estudiantes y se presentan los resultados del estudio.

El modelo permite comprender el concepto de *programa* desde la perspectiva de la Epistemología Genética y constituye un aporte a la didáctica de la programación. Asimismo, contribuye al desarrollo del área didáctica de la informática, dado que elementos del modelo también pueden aplicarse para investigación acerca de la construcción de conocimiento sobre otros conceptos de informática. Los resultados de las investigaciones constituyen insumos para la elaboración de pautas didácticas basadas en fundamentos teóricos sólidos.

Palabras clave

Computación. Didáctica de la programación. Construcción de conocimiento. Epistemología genética. Algoritmos y estructuras de datos. Programas.

Agradecimientos

En primer lugar, un agradecimiento muy especial al tribunal, Aiala Rosá, Crediné Silva de Menezes y Javier Blanco, quien además fue mi revisor.

Al Instituto de Computación (InCo) de la Facultad de Ingeniería, Universidad de la República (Uruguay), por brindarme su apoyo, en muy diversas formas, para que este trabajo pudiera realizarse. También, a mis compañeros y compañeras del InCo por sus buenos deseos y estimularme, cada quien a su manera, a completar el trabajo.

A quienes, junto conmigo, integran el grupo de investigación en Didáctica de la Informática del InCo y el Núcleo Interdisciplinario Filosofía de la Ciencia de la Computación, por tanta inspiración, enseñanzas, y ayudar a formarme en esta área que es tan especial para mí.

A mis colegas docentes del curso de Programación 1 dictado por el InCo, por tantos años de valioso trabajo en equipo y por haber contribuido, casi "sin querer", a la elección del tema para el estudio empírico presentado en este trabajo.

A mis estudiantes, Aaron, Ignacio D, Ignacio U, Joaquín, Juan, Martín, Mónica, Nicolás, Pablo M, Pablo P, Tomás, Valeria y Ximena, por haber participado en forma voluntaria del estudio empírico, y a Kevin, por su participación previa en la experiencia piloto.

A todos mis amigos, amigas y familiares cercanos, por su amor incondicional, en las buenas y en las malas, y por darme continuos ánimos para completar este trabajo.

Finalmente, un profundo agradecimiento a mi supervisora y orientadora, Sylvia da Rosa, por su cariño, dedicación y enorme paciencia para conmigo a lo largo de los años. Este trabajo no habría sido posible sin su guía, sus valiosas enseñanzas, consejos e incansables correcciones. ¡Gracias!

Contenido

Capítulo 1

Introducción: Computación, didáctica y un modelo de investigación

El propósito del presente trabajo es presentar un modelo para investigación en didáctica de la programación, desarrollado en el marco de la teoría epistemológica de Jean Piaget. Dicha teoría plantea una visión *constructivista* del conocimiento y explica cómo el mismo se construye desde dos perspectivas. Por un lado, desde la *psicología genética*, que explica la construcción de conocimiento por parte de los sujetos, a partir de datos de estudios empíricos realizados por Piaget a lo largo de muchos años de investigación y por el otro, desde un *análisis crítico de la evolución y desarrollo de las teorías científicas*, que explica la construcción del conocimiento científico a nivel sociogenético. Este análisis, realizado por Piaget y Rolando García, se describe en [13], donde los autores presentan una síntesis de ambas perspectivas, denominada tríada de etapas *intra-inter-trans*. Dicha síntesis explica el pasaje, en cualquier dominio de conocimiento científico, desde un nivel de conocimiento considerado inferior a otro nivel superior, tanto a nivel de la psicogénesis como del desarrollo histórico y sociogenético. El modelo que se presenta en este trabajo se basa en la teoría de Piaget en general y en la mencionada tríada en particular. Se le llama *modelo* porque explica cómo se construye conocimiento (sobre programación) y propone una metodología para investigación, cuyos resultados puedan luego volcarse a la formación de docentes de programación y elaboración de pautas didácticas para educar en la disciplina.

El modelo constituye una instancia de la teoría de Piaget para investigar la construcción de conocimiento sobre *algoritmos*, *estructuras de datos y programas* y surge como resultado de múltiples investigaciones realizadas a lo largo de muchos años. Las primeras investigaciones estudiaron la construcción de conocimiento sobre algoritmos y estructuras de datos y en todas ellas se constató la importancia de las experiencias informales de los sujetos en el proceso de construcción de conocimiento, previas a cualquier formalización. Las siguientes investigaciones incursionaron progresivamente en la construcción de conocimiento sobre programas a partir del conocimiento previamente construido. En este trabajo se presenta una interpretación de todas las investigaciones previas junto con un estudio empírico que profundiza específicamente en la construcción de conocimiento sobre *programas*, incluyendo su formalización en un *lenguaje de programación*, siendo este el principal objetivo del presente trabajo. Se espera que el modelo, que responde a principios de una teoría

epistemológica que explica la construcción de conocimiento científico, constituya un aporte al desarrollo de la didáctica de la programación en particular y, más en general, al de la didáctica de la informática, dado que diversos elementos del modelo pueden aplicarse a otros conceptos de informática.

En las secciones 1.1 y 1.2 de este capítulo se introducen dos nociones fundamentales relativas al contexto del trabajo: *informática* (también llamada *computer science*, CS) y *didáctica de la informática* (también llamada *computer science education*, CSE). También se realiza una breve síntesis que describe el surgimiento de la didáctica como disciplina específica. En la sección 1.3 se discute la importancia de contar con un marco teórico sólido para investigar en didáctica. En la sección 1.4 se presenta un resumen del estado del arte en la disciplina, mediante la enumeración de algunos trabajos realizados en el área CSE y su relación con el modelo. Por último, en la sección 1.5 se incluyen algunos breves comentarios relativos al estado de CSE en Uruguay, país donde se realizó este trabajo. El resto del documento se organiza de la siguiente manera: en el capítulo 2 se introduce la teoría de Piaget como marco teórico para el modelo y se describen las características del mismo; en el capítulo 3 se presenta un estudio empírico que ilustra en detalle cómo el modelo explica el proceso completo de construcción de conocimiento sobre *programas*. Finalmente, en el capítulo 4 se presentan las conclusiones del trabajo y algunas líneas de trabajo futuro.

1.1 Ciencia de la computación o informática

Como se explica más adelante (en la sección 1.2), la didáctica de la informática constituye un área específica dentro de la ciencia de la computación. Para comprender esa especificidad, es necesario precisar primero qué se entiende (y qué no) por ciencia de la computación. La *informática*, o *ciencia de la computación*, es una disciplina científica que posee las mismas características y problemas generales de definición que otras ciencias, como por ejemplo matemática, física o química. En el mundo anglosajón se la suele llamar *computer science* (ciencia de la computación), mientras que en el mundo franco-germano se la suele llamar *informatique* (Francia) o *informatik* (Alemania). A lo largo del presente trabajo se utiliza, indistintamente, una u otra denominación. Los objetos que estudia la informática son los datos y las técnicas que permiten trabajar con los mismos. Así pues, temas tan diversos como estructuras de datos, algoritmos, bases de datos o teoría de grafos, entre muchos otros, caen dentro del dominio de la ciencia de la computación. Desde sus inicios, la informática se ha desarrollado notoriamente y su influencia sobre diversas áreas se expande continuamente, dando origen a múltiples visiones sobre qué es la informática. Cada visión determina a su vez qué se entiende por educación en informática, dado que los aspectos de educación están estrechamente ligados a la visión de la disciplina.

La visión adoptada en este trabajo se alinea con las definiciones propuestas por Holmboe, McIver y George [46] y Dowek [38], donde los primeros proponen definir *computer science* (en adelante, CS) como: *the collection of scientific disciplines oriented towards the electronic or digital storing and processing of information* (*colección de*

disciplinas científicas orientadas al almacenamiento y procesamiento electrónico o digital de información, traducción del autor, en adelante t.d.a). En este sentido, entienden que la informática es una disciplina académica y con base científica que estudia temas como algoritmos, procesamiento de datos y lenguajes de programación. Por su parte, Dowek afirma que: *l'informatique est structurée par quatre concepts: algorithme, machine, langage et information* (*la informática está estructurada por cuatro conceptos: algoritmos, máquinas, lenguajes e información*, t.d.a), estableciendo una diferencia con otras áreas vinculadas al uso de computadoras y tecnología, las cuales no se consideran CS.

Algunas de esas áreas son, por ejemplo, alfabetización tecnológica (uso de aplicaciones, como ser procesadores de texto o planillas de cálculo) o tecnología para la educación (uso de herramientas, tales como recursos audiovisuales para educar en otras disciplinas o plataformas en línea para interacción entre docentes y estudiantes). Históricamente, estas áreas han estado presentes en la enseñanza preuniversitaria, lo que muchas veces ha generado confusión en la ciudadanía entre lo que es informática y lo que es uso de tecnologías. En tanto, la educación en informática (conforme a la visión propuesta en [46] y [38]) ha estado restringida mayormente al contexto terciario o superior, más específicamente a carreras cuyo propósito es la formación de técnicos o profesionales vinculados a la computación, tales como tecnicaturas, licenciaturas o carreras en ingeniería.

Por otra parte, Eden establece en [40] tres paradigmas para definir ciencia de la computación: *racionalista*, *tecnocrático* y *científico*. El paradigma racionalista define a la informática como: *a branch of mathematics, treats programs on a par with mathematical objects, and seeks certain, a priori knowledge about their 'correctness' by means of deductive reasoning* (*una rama de las matemáticas, trata a los programas a la par de los objetos matemáticos y busca conocimiento certero, a priori sobre su 'corrección', por medio de razonamiento deductivo*, t.d.a). El paradigma tecnocrático la define como: *an engineering discipline, treats programs as mere data, and seeks probable, a posteriori knowledge about their reliability empirically using testing suites* (*una disciplina ingenieril, trata los programas como meros datos, y busca un conocimiento empírico verificable, a posteriori acerca de su confiabilidad, usando casos de prueba*, t.d.a). El paradigma científico la define como: *a natural (empirical) science, takes programs to be entities on a par with mental processes, and seeks a priori and a posteriori knowledge about them by combining formal deduction and scientific experimentation* (*una ciencia natural (empírica), toma los programas como entidades a la par de los procesos mentales, y busca conocimiento a priori y a posteriori sobre ellos combinando deducción formal con experimentos científicos*, t.d.a).

Las definiciones propuestas en [46] y [38] son lo bastante amplias como para incluir a los tres paradigmas enumerados en [40] dado que, más allá de sus diferencias, los tres entienden la computación como una disciplina con base científica, sin importar si la misma tiene un carácter más formal o más experimental. Esta visión de la informática se despega notoriamente de otras que la ven desde perspectivas diferentes,

como ser alfabetización tecnológica o tecnología para la educación, o incluso de otras visiones como, por ejemplo, la vinculación de personas mediante redes sociales. El enfoque a tomar desde la didáctica queda determinado por la visión que se tenga de la computación, dado que la didáctica es un área específica dentro de ella, como se explica en la sección 1.2. Puesto que la visión sobre computación adoptada en este trabajo es la de una disciplina con base científica, los temas vinculados a educación en computación se abordan desde esta visión. Así pues, temas como educación en uso de herramientas informáticas, plataformas educativas, software para ofimática, o trabajo con redes sociales no son cuestiones abordadas por la didáctica de la informática.

1.2 Didáctica de la informática o CSE

En los últimos años, académicos e investigadores de varios países han logrado, con mayor o menor éxito, impulsar en la sociedad la idea de que educar en informática no solamente es importante para la formación de profesionales en la disciplina, sino que además es fundamental como parte de la formación general de todo individuo. La ausencia de una adecuada educación en computación en la ciudadanía es lo que lleva a la confusión entre informática y uso de tecnologías, mencionada en la sección anterior. Se plantea la necesidad de comenzar a educar en informática en etapas más tempranas de los sistemas educativos. Esto se promueve desde varias organizaciones, tales como el grupo *Computing at School (CAS)* en Reino Unido [69], la organización *CODE* de Estados Unidos [65], la asociación *Computer Science Teachers Association (CSTA)* [67] que agrupa docentes de diversos países, la *Association Enseignement Public & Informatique* [64] o la *Académie des sciences* de Francia [63].

Siguiendo los planteos de distintos investigadores, ya desde la década de 1980 se desarrollan proyectos de educación en informática para niños y adolescentes en diversas escuelas. Por ejemplo, Papert plantea en [54] que: *children can learn to program and that learning to program can affect the way they learn everything else* (*los niños pueden aprender a programar y ese aprendizaje puede afectar la manera en que aprenden todo lo demás*, t.d.a). Por otra parte, Dowek expresa en [39] que: *L'apprentissage de la programmation et de l'algorithmique est de nature à apporter beaucoup aux lycéennes et lycéens dans leur développement intellectuel, car il permet un travail par projets et demande de mettre en application des connaissances acquises. Et également car il permet de construire un pont entre le langage et l'action et montre l'utilité de la rigueur scientifique* (*el aprendizaje de la programación y de la algoritmia tiene mucho que aportar a las y los liceales en su desarrollo intelectual, porque les permite trabajar en proyectos y demanda la aplicación de conocimientos adquiridos. De igual manera, porque permite construir un puente entre el lenguaje y la acción y mostrar la utilidad del rigor científico*, t.d.a).

Planteos como los mencionados ponen sobre la mesa que la educación en informática es de importancia para la formación de cualquier educando y no sólo para quienes luego sigan estudios superiores en dicha disciplina. Esto lleva al surgimiento de algunas interrogantes, como ser *¿qué significa aprender ciencia de la computación en*

general, y programación en particular?, *¿qué académicos deberían dar respuesta a la pregunta anterior?* o *¿qué docentes están formados para educar en CS a estudiantes pre universitarios?* Para educar en informática es necesaria un área de estudio que aborde en profundidad los problemas de índole educativa propios de la disciplina. La búsqueda de respuestas a estas preguntas lleva a un desarrollo y un fortalecimiento considerable de la didáctica de la informática y dentro de ella a investigaciones sobre cómo educar en temas de computación, particularmente en programación, en la infancia y adolescencia. Estos esfuerzos pueden constatarse en variadas conferencias y congresos del área de los últimos años (SIGCSE, ITiCSE, ICER, PPIG, CIESC, ISSEP, CSERC, entre otros) y publicaciones (Computer Science Education, ACM Transactions on Computing Education, Informatics in Education, etc.).

La didáctica de la informática surge entonces como un área específica dentro de la ciencia de la computación y cuenta con fundamentos y métodos propios, los cuales estudian en profundidad aspectos relacionados con educación en temas vinculados a la disciplina. Holmboe, McIver y George definen en [46] a la didáctica de la informática (*Computer Science Education*, por su denominación en inglés, en adelante CSE) como: *the subject specific educational research for the subject computer science* (*la disciplina específica cuyo objeto de estudio es la investigación en educación para la asignatura ciencia de la computación*, t.d.a). Esto significa que, tal como sucede en otras áreas de conocimiento, la informática cuenta con un área que es su didáctica, en la cual las investigaciones han de ser llevadas a cabo por informáticos académicos especializados en los problemas didácticos. Concretamente, la didáctica de la programación es la sub-área que aborda específicamente tópicos de educación en temas de programación y es en ella donde pone el foco el presente trabajo. El modelo que se propone se enmarca dentro de esta última sub-área y busca estudiar en detalle el proceso completo de construcción de conocimiento, por parte de estudiantes novatos, sobre algoritmos, estructuras de datos y programas. Dicho modelo surge de la aplicación de conceptos de la teoría de Piaget (en particular la tríada *intra-inter-trans*), que explica en detalle la construcción de conocimiento científico, al terreno concreto de conocimiento sobre programación, como se explica en el capítulo 2.

1.2.1 Didáctica y pedagogía

Con frecuencia, suele haber confusión entre las nociones de *didáctica* y *pedagogía*, muchas veces siendo usadas indistintamente. Parte de esta confusión proviene del mundo anglosajón, donde el término *pedagogía* engloba casi todos los temas relativos a educación, mientras que el término *didáctica* casi no se utiliza. El término *CSE* se usa en el Reino Unido para lo que en el resto de Europa se denomina *didáctica de la informática*. Andrews señala en [18] que, en la educación Europea (mundo no anglosajón), el término *didáctica* refiere a: *theories of teaching and learning but from the subject-specific perspective* (*teorías de enseñanza y aprendizaje, pero desde la perspectiva inherente a la asignatura específica*, t.d.a). Por ejemplo, la didáctica de la matemática estudia estrategias para educar en temas como aritmética y álgebra,

mientras que la didáctica de la física lo hace con temas como cinemática y dinámica, habiendo notables diferencias entre las estrategias vinculadas a una y otra ciencia.

En cuanto al término *pedagogía*, Andrews dice que: *pedagogy includes an examination of the curriculum, in both broad and narrow forms, and the underlying systemic aims and objectives of education. It is a broad and inclusive concept that transcends subject boundaries but acknowledges general theories of teaching and learning* (*la pedagogía incluye una examinación de la currícula, tanto en lo amplio como en lo concreto, y los fines sistémicos subyacentes y objetivos de la educación. Es un concepto amplio e inclusivo que trasciende las fronteras de las asignaturas, reconociendo teorías generales de enseñanza y aprendizaje*, t.d.a). La pedagogía refiere a cuestiones generales de educación, independientes de la especificidad de cada asignatura, tales como estrategias para planificar la currícula de un curso o la organización de un plan de estudios en el marco de un sistema educativo, etc. En el mundo anglosajón, la palabra *pedagogía* se utiliza indistintamente para referirse tanto a aspectos genéricos de educación, como a aquellos específicos a cada asignatura, por lo que muchas veces se utiliza el término *pedagogía* para lo que en el continente (mundo no anglosajón) se denomina *didáctica*. Sobre esto, Andrews dice que: *it is probably true to say that the English have no word for the general concept, familiar to continental colleagues as pedagogy, as we tend to reserve the word for what they would describe as didactics* (*probablemente sea cierto decir que el Inglés no tiene una palabra para el concepto general, familiarizado como pedagogía para los colegas del continente, dado que tendemos a reservar la palabra para lo que ellos describirían como didáctica*, t.d.a).

La didáctica surge, a partir del siglo XVII, como una disciplina general [5], transversal a distintas áreas de conocimiento (similar a la pedagogía) y evoluciona en forma progresiva hasta adquirir un carácter específico, enmarcada dentro de cada área concreta de conocimiento. Bolívar señala en [19] que, a principios del siglo XX, se comienza a hacer la distinción entre una *didáctica general*, que *desarrolla los cánones para la enseñanza basados en la psicología* y una *didáctica especial* que *aplica dichas reglas generales a los contenidos específicos de cada una de las materias escolares*. Hasta ese momento, se concibe a la didáctica como un conjunto de principios genéricos que son aplicables a cualquier disciplina. Posteriormente, esa visión comienza a ser reemplazada por otra que establece la necesidad de una *didáctica específica* de cada materia, con su propia identidad y métodos concretos para educación en la misma.

Los matemáticos son pioneros en la consolidación de la didáctica de la matemática como un área más del conjunto de saberes matemáticos, desarrollada por matemáticos académicos especializados en didáctica. Por ejemplo, Dubinsky, con su teoría *APOS* [7] y Brousseau, con su *teoría de las situaciones didácticas* [2]. Se trata de un cuerpo teórico específico del conocimiento matemático, que no surge de la simple aplicación de una teoría desarrollada previamente para otros dominios, como la pedagogía o la psicología. Chevallard, con su propuesta de *transposición didáctica* [4] contribuye a la consolidación de la didáctica de la matemática como disciplina autónoma, enmarcada dentro del campo de la matemática, estableciendo que: *toda ciencia debe asumir, como*

primera condición, pretenderse ciencia de un objeto, de un objeto real, cuya existencia es independiente de la mirada que lo transformará en objeto de conocimiento, dando pie a la posibilidad de surgimiento de distintas *didácticas específicas*, cada una de ellas enmarcadas dentro de un área de conocimiento en particular.

Conforme culmina el siglo XX, la evolución de los sistemas educativos lleva a la necesidad de consolidar el desarrollo de las didácticas específicas en el marco de las distintas áreas de conocimiento que integran. Bolívar explica en [19] que, para fines de la década de 1980, surge la necesidad de poner el foco en la didáctica de las materias de enseñanza y que la formación en métodos genéricos debe reemplazarse por estudios orientados a la enseñanza y el aprendizaje de las disciplinas específicas. A raíz de esto, en otras áreas de conocimiento (entre ellas, la informática) paulatinamente se va construyendo una didáctica específica enmarcada en las mismas. Compete a cada una el desarrollo de un cuerpo teórico propio, que integre tanto conocimiento específico del área en cuestión con temas de educación en la misma. Nótese que esto difiere de la noción de trabajo conjunto entre académicos del área con profesionales del campo de la educación o la psicología, donde unos dominan saberes específicos de la disciplina mientras que otros dominan temas de educación, pero sin que haya integración alguna entre ambos tipos de saber. En el caso concreto de la informática, esto supone el desarrollo de un campo cuyos académicos integren conocimientos específicos de computación con aspectos relativos a cognición y aprendizaje de los mismos.

1.2.2 Conocimiento didáctico del contenido o PCK

Un concepto central a la didáctica de diversas disciplinas, entre ellas informática, es el llamado *conocimiento didáctico del contenido*. Entre investigadores en CSE hay consenso en adoptar la definición de Shulman [59] de *pedagogical content knowledge* (*conocimiento pedagógico del contenido*, en adelante, PCK, por sus siglas en inglés): *the ways of representing and formulating the subject that make it comprehensible to others* (*el conjunto de formas de representar y formular la disciplina para hacerla comprensible a otros*, t.d.a). Nótese que, por las mismas razones explicadas por Andrews en [18], es más adecuado traducir *pedagogical content knowledge* como *conocimiento didáctico del contenido*. Diversos autores provienen del mundo anglosajón, por lo cual utilizan la primera denominación en lugar de la segunda. En el marco del presente trabajo, se utiliza indistintamente una u otra, así como la sigla PCK. En sus investigaciones en didáctica de la informática, Saeli, Perrenet, Jochems y Zwaneveld [56] especifican el PCK como: *concept that combines the knowledge of the content (e.g., maths, informatics, etc.) to the knowledge of the pedagogy (e.g., how to teach maths, how to teach informatics, etc.), giving insights into educational matters relative to the learning and teaching of a topic. Teachers with good PCK are teachers who can transform their knowledge of the subject into something accessible for the learners* (*concepto que combina conocimiento del contenido (ejemplo: matemática, informática, etc.) con el conocimiento de la pedagogía (ejemplo: cómo enseñar matemática, cómo enseñar informática, etc.), dando una mirada a aspectos relativos al*

aprendizaje y enseñanza de un tópico. Profesores con buen PCK son aquellos que pueden transformar su conocimiento de la materia en algo accesible para los aprendices, t.d.a).

Por otra parte, Bolívar explica en [19] que los profesores con un buen PCK son capaces de generar representaciones de un determinado tema que sean entendibles para los estudiantes. Además del conocimiento de la materia y del conocimiento pedagógico, los docentes también deben desarrollar un conocimiento particular: cómo educar específicamente en su materia. Si bien es indispensable el conocimiento sobre la materia, este no genera por sí mismo ideas de cómo presentar un contenido particular a estudiantes, por lo que se requiere un buen PCK. El mismo implica comprender el significado de la enseñanza de un tópico en particular, así como de sus principios y sus formas didácticas de representación. Bolívar señala que el PCK se construye sobre tres tipos de conocimiento: *conocimiento de la materia, conocimiento pedagógico general y conocimiento de los estudiantes*. Afirma que es más que una simple conjunción entre el conocimiento de la materia y los principios generales didácticos y pedagógicos. Es la capacidad para transformar el conocimiento de la materia en representaciones didácticas significativas y comprensibles para los estudiantes.

1.3 Necesidad de un marco teórico para investigación en CSE

A partir de la necesidad de consolidar cada didáctica específica como una disciplina integrada al área de conocimiento a la que corresponde, resulta fundamental realizar investigaciones que posibiliten el desarrollo de cada una. En el caso de la didáctica de la informática, se han desarrollado investigaciones según diferentes enfoques, donde en todos es clara la necesidad de dominar los temas de informática para su estudio desde la perspectiva didáctica. Holmboe, McIver y George enumeran en [46] un espectro de investigaciones tales como: *new, untested ideas* (ideas novedosas para investigación en CSE), *reports from the trenches* (experiencias puntuales realizadas en clases de informática) o *empirical studies* (estudios empíricos orientados a analizar fenómenos específicos de programación, para detectar dificultades y comportamientos de estudiantes a la hora de aprender temas puntuales de programación).

Por otra parte, Hubwieser, Armori, Giannakos y Mittermeir brindan en [47] una comparación del estado de la investigación, en varios países, de temas vinculados a CSE en escuelas primarias y secundarias (Reino Unido, Nueva Zelanda, Estados Unidos, Israel, Francia, Suecia, Rusia e Italia). Concluyen que, en la mayoría de esos trabajos, *proper teacher education in substantial extent and depth seems to be one of the most critical factors for the success of rigorous computer science education* y agregan que *programming in one form or another, seems to be absolutely necessary for a future oriented CSE* (*una formación docente adecuada en un grado y profundidad sustanciales parece ser uno de los factores más críticos para el éxito de una CSE rigurosa,* y agregan que *la programación, en una forma u otra, parece ser absolutamente necesaria para una CSE orientada al futuro*, t.d.a). También que: *despite the fact that there have been several important developments in K-12 CS education in recent years, there is still much that can be done, especially in K-12 educational systems*

with low CS subject integration (*a pesar de que ha habido muchos e importantes desarrollos en CSE a nivel de la educación en CS en K-12 en años recientes, aún hay mucho por hacer, especialmente en sistemas educativos K-12 con poca integración de asignaturas de ciencia de la computación*, t.d.a).

Tras analizar diversas investigaciones como las recién enumeradas, se observa que no alcanza solamente con saber de computación para investigar en didáctica de la informática, sino que además es fundamental conocer aspectos específicos relativos a cognición y aprendizaje para realizar investigación integral al respecto. Holmboe, McIver y George afirman en [46] que: *there has been a lack of reference to pedagogical theory, underlying most past research studies. This has resulted in a failure to provide teachers with "pedagogical content knowledge", critical to gaining useful insights into cognitive and educational issues surrounding learning* (*ha habido una carencia subyacente de referencias a teorías pedagógicas en la mayoría de las investigaciones pasadas realizadas. Esto ha resultado en una falla en proveer a los profesores con "conocimiento didáctico del contenido", crítico para ganar una visión útil en temas cognitivos y educativos vinculados al aprendizaje*, t.d.a). También, Peyton Jones et al afirman en [21] que: *one reason for the lack of expertise in computer science teaching is the background of the teachers.* Agregan que *CAS is preparing to play a role in the planning and provision of teacher CPD (continuous professional development)* (*una razón para la falta de experiencia en CSE es la formación de los docentes.* Agregan que *el grupo "Computación en la escuela" (Computing At School, CAS) se está preparando para jugar un rol en la planificación y provisión de desarrollo profesional y continuo de profesores*, t.d.a).

Así como ocurre en otras disciplinas, se plantea el problema de adoptar un contexto que abarque las investigaciones tanto desde la perspectiva disciplinar como didáctica. Para ello, resulta fundamental el desarrollo de un marco teórico que sustente las investigaciones que se realicen. Un buen punto de partida es analizar qué se ha hecho en investigación en didáctica dentro de otras ramas de conocimiento. El ejemplo paradigmático es la didáctica de la matemática, que se ha desarrollado como una disciplina específica dentro de la matemática, con sus propias investigaciones llevadas a cabo por matemáticos especializados en didáctica. No obstante, en otras áreas de conocimiento también se han realizado investigaciones. Holmboe, McIver y George mencionan en [46] que: *looking to the variety of work being published in more established fields (e.g. science education, mathematics education and teaching and learning of foreign languages) may give several useful pointers to researchers in computer science education* (*mirar la variedad de trabajos publicados en campos más establecidos (ejemplo: educación en ciencias, educación en matemática, enseñanza y aprendizaje de lenguas extranjeras) puede dar muchos puntos útiles a los investigadores en CSE,* t.d.a). En la sección 1.4 se enumeran otros trabajos realizados en el área CSE. Algunos de ellos están basados en el mismo marco teórico utilizado para el presente trabajo, mientras que los restantes se basan en otros enfoques o marcos teóricos diferentes.

Para realizar investigaciones en CSE que la consoliden como una disciplina específica dentro de CS y posibiliten la formación de docentes con un buen PCK, resulta esencial contar con un marco teórico que sustente las mismas, de modo que las investigaciones que se realicen no se reduzcan a observaciones realizadas a partir de experiencias aisladas ni a cuestiones de opinión por quienes las lleven a cabo, sino que requieren ser tratadas con el mismo rigor que cualquier otra actividad de investigación científica. Holmboe, McIver y George afirman en [46] que: *the future work of CSE must have a stronger connection to the theoretical frameworks of education-related disciplines such as pedagogy, epistemology, curriculum studies and psychology* (*el trabajo futuro en CSE debe tener una conexión más fuerte con marcos teóricos de disciplinas relacionadas con educación como pedagogía, epistemología, estudios curriculares y psicología*, t.d.a). Sostienen que la fuerte conexión de CSE con disciplinas relacionadas a educación constituye la argumentación teórica de la investigación como una manera de proveer evidencia de su efectividad. Sintetizan y aplican estas ideas para el caso de CSE afirmando que: *the aim (of CSE) should rather be to describe the different ways in which students come to understand, or not to understand, the subject matter. These descriptions, accompanied by knowledge of general pedagogical theory, epistemology and solid subject knowledge will make the foundation for answering the traditional didactical questions of why, what, how and for whom to educate in CS* (*el propósito (de CSE) debería ser describir las diferentes formas en las que los estudiantes llegan a entender, o no entender, los aspectos de la materia. Estas descripciones, acompañadas de conocimientos de pedagogía, epistemología y sólido conocimiento de la materia, sientan las bases para responder las tradicionales preguntas didácticas de qué, cómo, por qué y para quién educar en informática,* t.d.a).

En el marco de las cuatro preguntas didácticas tradicionales: *qué*, *cómo*, *por qué* y *para quién* educar en informática, este trabajo aborda específicamente la construcción de conocimiento sobre algoritmos básicos, estructuras de datos y programas (*qué*) y su introducción en la educación secundaria o superior inicial (*para quién*). Como se dijo anteriormente, hay consenso entre los investigadores sobre los beneficios que ese aprendizaje significa para todos los estudiantes, no solamente para los que luego seguirán estudios superiores en informática, como lo plantean Papert en [54] y Dowek en [39] (*por qué*). Con respecto al *cómo*, muchos autores interpretan la pregunta del siguiente modo: *¿cómo pueden hacer los docentes para enseñar determinado tema de una mejor manera?* Hacen hincapié en proponer estrategias y/o metodologías generales para la introducción de dicho tema por parte del docente, por ejemplo: mediante conceptos fundamentales (Schwill [58]), fomentando la abstracción (Wing [62]), o mediante proyectos (Dowek [39]), es decir enfocándose en cómo el docente debería proceder.

Por el contrario, el modelo presentado en este trabajo, aborda la pregunta del *cómo* desde la perspectiva del estudiante. Esto es, *¿cómo aprende el estudiante conceptos de programación?* Esta formulación es un caso particular del problema general de cómo aprenden las personas, estudiado en profundidad por Piaget y colaboradores (entendido "aprenden" como construcción de conocimiento, no sólo como aprendizaje

formal). A raíz de amplios estudios empíricos y del análisis crítico de la historia de las ciencias, Piaget elaboró explicaciones sobre la construcción de conocimiento, empezando por el que los sujetos construyen, mediante interacción con su medio, desde el nacimiento (*psicogenésis*) y también por el origen de las ideas científicas (*sociogenésis*). Mediante la aplicación de la teoría a la construcción de conocimiento sobre algoritmos, estructuras de datos y programas, este trabajo introduce un modelo para investigación en didáctica de la programación.

En contraste con otras propuestas para guiar a los estudiantes en el aprendizaje de la programación que involucran desde el comienzo algún lenguaje de programación o herramienta de computadora (como por ejemplo [24, 43, 44, 45, 51, 60]), el modelo que se propone surge de observaciones de situaciones en la vida cotidiana en las que las personas utilizan con éxito métodos para resolver problemas o realizar tareas tales como juegos, ordenación de objetos, búsquedas, etc. para construir conceptos de programación a partir de ellas y formalizarlos posteriormente. En dichas situaciones, una acción o una secuencia de acciones se repite hasta que se alcanza un estado especial, que puede resolverse mediante una acción directa. Las descripciones de las personas incluyen frases como "*hago lo mismo hasta que…*" o "*ahora sé cómo hacerlo*", refiriéndose a los casos en que usan el mismo método y llegan a un estado especial de resolución fácil o inmediata, respectivamente. Estas descripciones se relacionan con la programación en el sentido de que la repetición de acciones, hasta que se alcanza un caso especial de terminación, se formaliza mediante instrucciones recursivas o iterativas. A su vez, los objetos manipulados por las personas al resolver problemas como los mencionados, son asimilables a las estructuras de datos manipuladas en los programas. Por ejemplo, la ordenación de una baraja de cartas españolas, primero por palo y luego por número, es comparable a la ordenación de un arreglo de parejas (palo, número) en un lenguaje de programación, donde dicho arreglo constituye la estructura de datos manipulada.

Estas observaciones, en el marco de la teoría epistemológica de Piaget llevan a la formulación de preguntas tales como: *¿Existe alguna conexión entre el 'saber hacer' (conocimiento instrumental) de las personas al resolver problemas y los algoritmos formales?* En caso afirmativo, *¿cuál es la naturaleza de dicha conexión? ¿Cuál es el rol del conocimiento instrumental en el proceso de conceptualización y formalización de los conceptos? ¿Cómo se pueden tomar en cuenta los algoritmos que las personas aplican en situaciones cotidianas para conocer la construcción de conocimiento sobre programas?* Aplicando principios de la teoría de Piaget en el diseño, desarrollo y análisis de variados estudios empíricos a lo largo de años, se han logrado elaborar respuestas a dichas preguntas. El trabajo que aquí se presenta complementa dichos estudios previos y profundiza en investigar la construcción de conocimiento *formal* sobre programación. Integra sus resultados con los resultados de las investigaciones previas en un modelo que explica todas las etapas en la construcción de conocimiento sobre algoritmos, estructuras de datos y programas, partiendo desde el conocimiento *instrumental* de los sujetos en el plano de la acción (resolución de instancias concretas de problemas algorítmicos en situaciones cotidianas), pasando por conocimiento

conceptual, evidenciado por la capacidad de los sujetos de expresar en lenguaje natural el método empleado y las razones de éxito en la resolución, y llegando a la construcción de conocimiento *formal*, evidenciado por la capacidad de escribir *programas* que dan solución general a dichos problemas, comprender y explicar cómo y por qué funcionan cuando se ejecutan en una *computadora*.

1.4 Estado del arte: algunos trabajos relacionados

Las preguntas de la sección anterior, así como sus respuestas, son de carácter *epistemológico*. Abren la puerta a nuevas preguntas de carácter *didáctico*. Es decir, lo que se conoce sobre la construcción de conocimiento sobre conceptos de programación *¿podría ayudar a mejorar la enseñanza y aprendizaje de la programación? ¿cómo debería llevarse a cabo?* O como ya se ha dicho: *¿cómo pueden hacer los docentes para educar sobre determinado tema de una mejor manera?* Este tipo de preguntas constituye una línea de trabajo futuro de aplicación del modelo. El mismo se basa en un marco teórico sólido que explica la construcción de conocimiento y se espera que contribuya al enriquecimiento del PCK de docentes de programación, posibilitando la elaboración de pautas didácticas para aplicación en el aula que tomen en cuenta la forma en que los estudiantes aprenden a programar, en base a las explicaciones provistas por el modelo.

Por el contrario, otros trabajos en didáctica de la informática buscan aportar elementos a los docentes para mejorar su práctica al enseñar diferentes temas, muchas veces basándose en enfoques de teorías *psicológicas* en vez de *epistemológicas*. Algunos trabajos se basan en algún marco teórico y otros, como se discute en la sección anterior, no se basan en ninguno, o al menos no es mencionado en forma explícita. En algunos casos, las características de dichos trabajos permiten inferir elementos de algún marco teórico subyacente en los mismos. En esta sección se presenta un resumen del estado del arte relativo a investigación en CSE, mediante la enumeración de varias líneas de trabajo junto con ejemplos de trabajos concretos realizados en el marco de las mismas. También se realiza una breve discusión de similitudes y diferencias entre el modelo y el enfoque de cada línea. La lista no pretende ser exhaustiva, sino una muestra de diversos enfoques que tienen presencia en la disciplina CSE. Tres de ellos se basan explícitamente en la teoría de Piaget (*teoría APOS*, *teoría neo-Piagetiana*, *teoría de las situaciones didácticas*), al igual que el modelo propuesto. También se enumeran otras líneas (*Computer Science Unplugged*, *Active learning, teoría de los modelos mentales*, *neurociencia y psicología cognitiva*, *fenomenografía*). Algunas de estas líneas constituyen aportes a la investigación en CSE, mientras que otras son directamente propuestas de aplicación didáctica en el aula.

1.4.1 Teoría APOS

La teoría *APOS* (*Actions, Processes, Objects, Schemes*) es una teoría constructivista surgida en el marco de la didáctica de la matemática. Esta teoría toma como base conceptos de la teoría epistemológica de Piaget y desarrolla un modelo orientado a explicar cómo los sujetos construyen conocimiento sobre temas de matemática. De acuerdo con Dubinsky y McDonald en [7], *mathematical knowledge consists in an individual's tendency to deal with perceived mathematical problem situations by constructing mental actions, processes, and objects and organizing them in schemas to make sense of the situations and solve the problems* (*el conocimiento matemático consiste en la tendencia de un individuo a lidiar con situaciones de problemas matemáticos percibidas mediante la construcción de acciones mentales, procesos y objetos para organizarlos en esquemas de modo que las situaciones cobren sentido y permitan resolver los problemas*, t.d.a).

Según Dubinsky y McDonald, las repetidas *acciones* realizadas por un individuo al intentar resolver un problema matemático se transforman, tras realizar reflexiones sobre las mismas, en *procesos* mentales. Los *objetos* son elementos matemáticos mentales que surgen de la aplicación de los procesos cuando el sujeto toma conciencia de los cambios que los procesos infligen en los elementos matemáticos concretos manipulados. Finalmente, los *esquemas* representan colecciones de acciones, procesos y objetos que el sujeto construye mentalmente como marco de comprensión del concepto matemático involucrado.

Si bien la teoría *APOS* surge en la didáctica de la matemática, la misma propone el uso de lenguajes de programación como herramientas para explicar cómo se construye conocimiento sobre conceptos de matemática. A raíz de esto, se han hecho algunas investigaciones para aplicar esta teoría a la investigación de construcción de conocimiento sobre conceptos de programación. Por ejemplo, Cetin presenta en [25] un trabajo de aplicación de la misma para analizar los distintos niveles de comprensión de estructuras *iterativas* (incluyendo *anidación* de iteraciones) por parte de un grupo de estudiantes de un curso introductorio de programación.

Dado que *APOS* se basa en una visión constructivista del conocimiento y su marco teórico es la teoría de Piaget, la misma tuvo influencia en el diseño de los primeros trabajos realizados en Uruguay para investigar cómo se construye conocimiento sobre programación. *APOS* propone una metodología de investigación específica, que fue adaptada a esos primeros estudios. Uno de ellos fue llevado a cabo por da Rosa en [6], en el cual presenta una investigación acerca de la construcción de conocimiento sobre algoritmos recursivos y su formalización en un lenguaje de programación funcional. Dicho estudio, en conjunto con otros en la misma línea, sentaron las bases para la construcción del modelo que se presenta en este trabajo (en el capítulo 2 se brindan detalles acerca de sus orígenes).

1.4.2 Teoría neo-Piagetiana

La teoría *neo-Piagetiana* es una teoría psicológica que comparte con la teoría de Piaget que la construcción de conocimiento sobre cualquier tópico evoluciona en forma progresiva desde niveles inferiores hasta niveles superiores, pero se diferencia de ella en relación a la capacidad de los sujetos para alcanzar niveles más abstractos de conocimiento según sus edades. A este respecto, Lister afirma en [50] que: *in Piaget's view, a child who exhibits a certain level of abstract reasoning on a given problem will tend to exhibit that same level of abstract reasoning on many other problems. Subsequent work in psychology, however, has shown that children (and adults) exhibit different levels of abstract reasoning on different problems* (*en la visión de Piaget, un niño que exhibe un cierto nivel de razonamiento abstracto en un problema dado, tenderá a exhibir ese mismo nivel de razonamiento abstracto en muchos otros problemas. Trabajos subsecuentes en psicología, sin embargo, han mostrado que niños (y adultos) exhiben diferentes niveles de razonamiento abstracto en diferentes problemas*, t.d.a).

Lister agrega además que: *people, regardless of their age, are thought to progress through increasingly abstract forms of reasoning as they gain expertise in a specific problem domain. Thus a person who is a novice in one domain (e.g. chess) will exhibit less abstract forms of reasoning than that same person will exhibit in a domain where he/she is expert (e.g. calculus)* (*se piensa que las personas, sin importar sus edades, progresan a través de formas de razonamiento incrementalmente abstractas a medida que ganan experiencia en un problema de un dominio específico. Por tanto, una persona que es novata en un dominio (ej: ajedrez) exhibirá menos formas de razonamiento abstracto que esa misma persona exhibirá en un dominio donde sea experta (ej: cálculo)*, t.d.a). Se han realizado varios estudios en CSE basados en la teoría neo-Piagetiana para analizar dificultades de los sujetos al construir conocimiento sobre conceptos de computación. Por ejemplo, el propio Lister presenta en [50] uno que realiza un análisis del grado de éxito de estudiantes novatos, en cursos iniciales de programación, dependiendo de qué tanto hayan desarrollado su capacidad de razonamiento abstracto.

Si bien la teoría de Piaget explica la construcción de conocimiento científico en todas las etapas de la vida del individuo, una de las fuentes de datos empíricos para la teoría provino de estudios sobre la construcción de conocimiento desde el nacimiento y hasta la adolescencia. Ello dio origen al surgimiento de la *psicología genética*, que posteriormente fue ampliamente aplicada en la educación de niños. Esto llevó a malinterpretar esa rama de la psicología como contribución más importante de la teoría de Piaget, desconociendo muchas veces que se trata de una teoría *epistemológica* y no *psicológica*. En particular, muchos trabajos en didáctica de la informática basados en la teoría neo-Piagetiana se centran en los estadios del desarrollo del conocimiento infantil, que el propio Piaget y colaboradores superaron en sus últimos trabajos epistemológicos, abarcándolos en la tríada de etapas, la cual se describe en detalle en el capítulo 2.

1.4.3 Teoría de las situaciones didácticas

La *Teoría de las situaciones didácticas*, desarrollada por el matemático Brousseau [2] es un modelo didáctico concebido inicialmente en el marco de la didáctica de la matemática. Se basa en la teoría de Piaget y surge como una propuesta didáctica para su aplicación en el aula, concretamente en temas de matemática. A pesar de esto, las características de esta teoría (que se describen en breve) son tales que permiten también su aplicación en otros temas, particularmente de informática. A grandes rasgos, esta teoría se basa en la idea de dos tipos de interacciones, denominadas *situación adidáctica* y *situación didáctica*.

Una situación adidáctica es una *interacción entre un sujeto y un medio, a propósito de un conocimiento*. Se trata de una interacción del estudiante con una problemática concreta, a partir de la cual se espera que construya un nuevo conocimiento, sin intervención del docente. El docente le propone al estudiante la situación y deja que ponga en práctica sus conocimientos para resolverla, teniendo que modificarlos y adaptarlos para poder resolver la problemática planteada, construyendo así conocimiento nuevo durante el proceso. El docente no hace explícito cuál es el conocimiento que espera que el estudiante construya, sino que diseña la actividad de modo tal que sea el propio estudiante quien lo descubra durante el proceso. Una situación didáctica es otra situación que se da posteriormente, con la intervención del docente y eventualmente otros estudiantes, en la cual se hace explícito el nuevo conocimiento construido durante la situación adidáctica, así como su formalización e integración con los conocimientos previos del estudiante.

Si bien esta teoría surge en el marco de la didáctica de la matemática (al igual que la teoría *APOS*), la misma sirvió como guía, durante el proceso de construcción del modelo, para la elaboración de pautas didácticas para introducir conceptos de programación. Por ejemplo, en [37] da Rosa y Gómez presentan una propuesta inspirada en esta teoría para la introducción de *arreglos* en un curso inicial de programación. Se propone al estudiante interactuar con instancias concretas de arreglos en problemas puntuales, a partir de una guía de preguntas elaboradas por el docente. El estudiante debe hacerlo en forma autónoma y con antelación a tratar el tema en clase. Posteriormente, el estudiante presenta sus resultados en clase ante el docente y otros estudiantes, y a partir de ahí se generaliza la noción de arreglo y se formaliza en un lenguaje de programación. El trabajo previo del estudiante constituye una situación adidáctica y su posterior puesta a punto en clase una situación didáctica. En [27] da Rosa propone otra aplicación de esta teoría para la introducción de *punteros* y *listas encadenadas*, donde se plantea al estudiante una actividad donde interactúa, sin intervención del docente con cajas numeradas y flechas que las conectan, generando una secuencia ordenada en forma ascendente (situación adidáctica). Posteriormente, con intervención del docente, se formaliza (en un lenguaje de programación) un algoritmo para insertar un valor en una lista encadenada ordenada (situación didáctica).

1.4.4 Computer Science Unplugged

La corriente *Computer Science Unplugged* plantea el aprendizaje de conceptos de computación mediante actividades cenestésicas, sin utilizar una computadora. Dichas actividades están concebidas para aprender diversos conceptos de CS, no solamente de programación. También propone actividades para el aprendizaje de conceptos de otras ciencias, como por ejemplo matemática. Consiste en la realización de actividades que hacen uso de objetos tales como tarjetas, fichas de plástico, cartulinas, etc. Según dice el sitio oficial de *CS Unplugged* [68]: *los alumnos reciben desafíos basados en unas cuantas reglas sencillas y en el proceso de resolver esos desafíos descubren poderosas ideas por cuenta propia. No solo es una forma más memorable de aprender, sino que además los empodera para darse cuenta de que estas son ideas que están a su alcance. Las actividades también son bastante cenestésicas, cuanto más grandes sean los materiales, mejor*.

Como parte de la construcción del modelo que se propone en este trabajo, se han realizado algunos estudios basados en *Computer Science Unplugged*. Por ejemplo, en [29] da Rosa presenta uno, realizado con estudiantes de educación secundaria, sobre los preconceptos que pueden tener acerca de la ejecución de programas. Una de las actividades de dicho estudio consiste en el uso de vasos de distintos materiales que contienen líquidos de distintos colores. Los vasos simulan ser variables en la memoria de una computadora, los materiales simulan ser los tipos de datos de dichas variables y los líquidos de colores simulan ser los valores en ellas.

Por fuera del modelo, también se han realizado trabajos de aplicación didáctica inspirados en *Computer Science Unplugged*. Por ejemplo, Friss de Kereki presenta en [42] una propuesta basada en el uso de actividades cinestésicas para la introducción de diversos conceptos en un curso inicial de programación orientada a objetos. En dicho trabajo, presenta varias actividades, que no requieren el uso de una computadora, para trabajar conceptos de programación. Una de ellas propone la resolución de un juego de mesa con un tablero de madera y bolitas de colores para ejercitar la escritura de pasos para resolver una instancia de un problema algorítmico en el contexto del juego. Otra actividad propone el uso de plastilina y moldes de plástico con diversas formas (autos, animales, etc.) para simular la creación de objetos de distinto tipo en memoria (considerando el paradigma orientado a objetos), entendiendo que la totalidad de la plastilina disponible representa la masa total de memoria del programa, de la cual se toma plastilina para rellenar los distintos moldes y así construir diversos objetos.

Computer Science Unplugged no hace explícito un marco teórico que explique por qué el tipo de actividades que promueve son beneficiosas para el aprendizaje. No obstante, las características de dichas actividades están implícitamente alineadas con el marco teórico dado por la teoría de Piaget, dado que posibilitan a los sujetos construir conocimiento a partir de su interacción con objetos concretos en el plano de la acción. Por ello, resultan adecuadas tanto en trabajos de investigación sobre cómo se construye conocimiento como en actividades pensadas para su aplicación en el aula. El

estudio empírico que se presenta en el capítulo 3 también propone una actividad inspirada en esta corriente, donde los participantes construyen conocimiento sobre la estructura de datos *arreglo* a partir de la manipulación de una hilera de tarjetas numeradas, cuya disposición en el plano informal se asemeja a dicha estructura de datos en el plano formal.

1.4.5 Active learning

Active learning es una metodología educativa que promueve participación activa del estudiante en todas las etapas del proceso de aprendizaje. De acuerdo con Bonwell y Eison en [1]: *students must do more than just listen: they must read, write, discuss, or be engaged in solving problems. Most important, to be actively involved, students must engage in such higher-order thinking tasks as analysis, synthesis, and evaluation* (*los estudiantes deben hacer más que solo escuchar: deben leer, escribir, discutir o participar en resolver problemas. Más importante, para involucrarse activamente, los estudiantes deben participar en tareas de pensamiento de alto orden como análisis, síntesis y evaluación*, t.d.a). De acuerdo con McConnell en [52], las actividades pueden ser de distinta índole, tales como: *reading, writing, discussing, solving a problem, or responding to questions that require more than factual answers* (*lectura, escritura, discusión, resolución de un problema, o respuestas a preguntas que requieren más que respuestas textuales*, t.d.a). Esta corriente promueve la participación activa del estudiante en la resolución de múltiples actividades de diversa naturaleza, resultando aplicable a múltiples disciplinas. En este sentido, constituye una metodología *pedagógica* más que *didáctica*. No hace explícito un marco teórico concreto.

El propio McConnel describe en [52] algunas técnicas genéricas de *Active learning* aplicables al campo de CS. Las técnicas que menciona incluyen, por ejemplo, lectura de artículos por parte de grupos reducidos de estudiantes y posterior discusión grupal con toda la clase, o también ejecución manual (nuevamente en grupos reducidos de estudiantes) de distintos programas, para luego comparar los resultados obtenidos con los resultados arrojados por el programa correspondiente tras ser ejecutado en una computadora.

1.4.6 Teoría de los modelos mentales

Una teoría psicológica influyente en educación es la teoría de los *modelos mentales*, la cual se basa en el concepto de modelo mental. Distintas definiciones del concepto de modelo mental han sido dadas a lo largo de los años. Schwamb recoge varias de ellas en [57] y las sintetiza diciendo: *mental models are representations of knowledge dealing with particular objects and / or people in particular situations* (*los modelos mentales son representaciones del conocimiento que tratan con objetos particulares y/o personas en situaciones particulares*, t.d.a). Muy a grandes rasgos, esta teoría sostiene que el cerebro humano construye modelos de la realidad a pequeña escala, los cuales se utilizan para comprender nuevos eventos relacionados a dicha realidad, lo que lleva luego a la construcción de nuevos modelos mentales en un proceso que es gradual, permanente y progresivo.

Se han realizado distintos estudios en CSE, basados en esta teoría, que apuntan a identificar qué modelos mentales poseen los estudiantes previo al aprendizaje de temas de programación y cómo se puede hacer uso de dichos modelos mentales para aprender nuevos tópicos. Por ejemplo, Bubica y Boljat presentan en [22] un estudio de distintos modelos mentales que estudiantes novatos pueden tener cuando inician estudios de programación. Proponen distintas estrategias orientadas a transformar dichos modelos mentales preexistentes en nuevos modelos que posibiliten una comprensión adecuada de conceptos básicos de programación, tales como variables, asignación o iteración. Por otra parte, Lewis presenta en [49] un estudio de distintos modelos mentales sobre *recursión* que estudiantes novatos presentan al evaluar funciones matemáticas definidas en forma recursiva y cómo podrían ser utilizados para mejorar el aprendizaje de la recursión en cursos introductorios de programación.

1.4.7 Neurociencia y psicología cognitiva

En los últimos años, los avances en neurociencia y su aplicación a la psicología cognitiva han llevado al desarrollo de algunas líneas de investigación en CSE basadas en este enfoque. En [41], Escera se apoya en diversos autores y propone una definición de neurociencia como la *disciplina que busca entender cómo la función cerebral da lugar a las actividades mentales, tales como la percepción, la memoria, el lenguaje e incluso la consciencia*. En cuanto a la *psicología cognitiva*, Parkin la define en [11] como *la rama de la psicología que intenta proporcionar una explicación científica de cómo el cerebro lleva a cabo funciones mentales complejas como la visión, la memoria, el lenguaje y el pensamiento*.

Distintos estudios aplican neurociencia y psicología cognitiva al área de CSE. Por ejemplo, en [20], Bower presenta uno que analiza en detalle las distintas etapas subyacentes al proceso de aprendizaje, identificadas en los campos de la neurociencia y la psicología cognitiva, y cómo las mismas son instanciadas para el aprendizaje de conceptos de programación. Las etapas que menciona son: captura de la atención, percepción, comprensión, filtrado de información relevante, síntesis, memorización de aspectos importantes y, finalmente, abstracción de conceptos. Presenta un caso de estudio concreto relativo a la escritura de un programa que hace uso de una estructura iterativa para resolver un problema de cálculo concreto. Señala la importancia de presentarle a los educadores en informática, como parte de su formación docente, explicaciones acerca del funcionamiento cerebral, a efectos de permitirles diseñar actividades educativas que tomen en cuenta cómo se da el proceso de aprendizaje a partir de las etapas mencionadas.

1.4.8 Fenomenografía

La *fenomenografía* es una metodología de investigación orientada a describir y clasificar cómo las personas perciben y comprenden diversos fenómenos. Procura categorizar las diferentes maneras en las cuales un mismo concepto es entendido por diferentes grupos de individuos, para lograr tener una visión global de los distintos grados de comprensión del mismo e intentar buscar razones por las cuales se dan estos grados de diferencia. De acuerdo con la fenomenografía, cualquier fenómeno puede ser comprendido o percibido en un número limitado de formas diferentes. El propósito es descubrir y clasificar las distintas maneras de comprensión de un mismo fenómeno.

De acuerdo con Marton [10], *phenomenography investigates the qualitatively different ways in which people experience or think about various phenomena* (*la fenomenografía investiga las formas cualitativamente diferentes en las cuales las personas experimentan o piensan acerca de varios fenómenos*, t.d.a). Esta metodología fue desarrollada a partir de 1979 por un grupo de investigación del departamento de educación de la Universidad de Gotemburgo, Suecia.

Se han realizado estudios basados en fenomenografía aplicados a disciplinas muy diversas, desde aquellas con base en ciencias experimentales (tales como física o medicina) hasta otras (como ser historia, estudios sociales o aprendizaje de lenguas extranjeras). En todas ellas, se investiga cuáles son las distintas maneras en la que distintos grupos de individuos perciben distintos conceptos vinculados a las mismas y qué grado de comprensión alcanzan sobre ellos.

En los últimos años, también se han empezado a realizar estudios de fenomenografía sobre la comprensión de conceptos de informática. Por ejemplo, Bucks y Oakes presentan en [23] un trabajo para investigar los distintos niveles de comprensión que un grupo de personas de la Universidad Purdue en Indiana, EE.UU, tienen de dos conceptos específicos de programación (selección e iteración). El estudio propone a los participantes escribir un algoritmo para adivinar un número al azar dentro de un rango de valores posibles. Dicho algoritmo requiere aplicar estructuras tanto de selección como de iteración. Posteriormente, se realizan entrevistas a los participantes acerca del algoritmo, para recabar información que permita clasificar los distintos niveles de comprensión que tienen acerca de ambos conceptos. Los participantes del estudio son elegidos dentro de un espectro amplio, desde estudiantes novatos hasta estudiantes avanzados (e incluso algunos docentes) a efectos de abarcar el mayor abanico posible de niveles de comprensión posibles. En base a las respuestas dadas por los participantes, los investigadores logran identificar entre cinco y seis niveles de comprensión posibles de cada concepto, desde el nivel más bajo (comprensión muy pobre o nula) hasta el nivel más alto (comprensión avanzada, capacidad de abstracción y habilidad para aplicar los conceptos en situaciones de alta complejidad). El estudio permite a los investigadores formar una visión global de qué tan profundamente los estudiantes llegan a comprender los conceptos involucrados. También analizan brevemente las posibilidades que esto puede brindar para la evaluación y el desarrollo de pautas didácticas para el abordaje de estos conceptos en cursos de programación.

Los estudios basados en *fenomenografía*, así como los trabajos basados en *neurociencia y psicología cognitiva*, la teoría *neo-Piagetiana* y la teoría de los *modelos mentales*, se diferencian del modelo propuesto en que siguen un enfoque *psicológico* más que *epistemológico*. Si bien sus aportes al área CSE son valiosos, están orientados al estudio de los procesos cognitivos de los sujetos más que a la naturaleza específica de los temas de computación sobre los cuales construyen conocimiento, ni a su interacción con ellos por parte de los sujetos. Explican dificultades o preconceptos de los sujetos al construir conocimiento sobre determinados temas, pero sin que las características inherentes a ellos sean preponderantes.

El modelo propuesto en este trabajo tiene como objetivo explicar cómo se construye conocimiento sobre conceptos de *programación*, lo cual requiere tomar en cuenta la naturaleza específica de la disciplina, que además es *científica*. La teoría de Piaget es *epistemológica* y, como tal, explica cómo se construye conocimiento (en particular científico), donde el dominio de conocimiento involucrado es relevante, así como la interacción de los sujetos con objetos relativos a dicho dominio. Además, dado que es un modelo para investigación en *didáctica* de la programación, la especificidad de la disciplina debe ser tomada en cuenta, razón por la cual un enfoque como el adoptado por esos trabajos se considera insuficiente. La interacción de los sujetos con objetos vinculados a temas de programación es de relevancia. Tales objetos pueden ser *físicos* (por ejemplo tarjetas, fichas de un juego, cajas, etc.) sobre los cuales los sujetos realizan tareas asimilables a instancias concretas de algoritmos, o pueden ser *abstractos* (por ejemplo, el texto de un programa o una representación en memoria de una estructura de datos). A lo largo del resto del trabajo se trabaja con objetos como los mencionados y se explica cómo los sujetos construyen conocimiento sobre programación a partir de su interacción con los mismos.

1.5 Estado de la disciplina CSE en Uruguay

Conforme a lo expuesto en las secciones anteriores, el desarrollo de la disciplina CSE a nivel global aún está lejos de posicionarla como área académica establecida, si bien se han hecho avances prometedores en los últimos años. El presente trabajo fue desarrollado en Uruguay, país cuya situación no es diferente a la de otros países.

En relación a la investigación en CSE, el grupo de Didáctica de la Informática del InCo (Instituto de Computación [72]) de la Facultad de Ingeniería, Universidad de la República (UdelaR), del cual el autor del presente trabajo forma parte, ha realizado aportes a la disciplina a lo largo de los años que llevaron a la construcción del modelo. Algunas investigaciones fueron realizadas junto a otros participantes, destacando el *Núcleo interdisciplinario Filosofía de la Ciencia de la Computación* (NI FCC) [74] conformado por el grupo de investigación en Didáctica de la Informática del InCo, integrantes del Instituto de Filosofía de la Facultad de Humanidades y Ciencias de la Educación (UdelaR) [73] y docentes egresados del Profesorado de Informática (Administración Nacional de Educación Pública, ANEP, Uruguay) [76]. También se han realizado trabajos en el ámbito universitario privado.

En cuanto a formación docente en CS, en Uruguay no existe oferta académica para formación de docentes universitarios de informática. La única oferta es el Profesorado de Informática, pero está dirigido a la formación de docentes para educación media y no pertenece al ámbito universitario, sino a ANEP, organismo que gestiona los sistemas de educación primaria, secundaria, técnica y formación docente pre-universitaria a nivel nacional. El Instituto de Computación colaboró con la creación del mismo, aportando desde el conocimiento específico de la disciplina (informática), así como a nivel didáctico (en esto último, participó el grupo de Didáctica del InCo).

Se considera importante avanzar en Uruguay no solo en investigación en CSE sino también en profesionalizar la carrera docente en informática a nivel universitario, siendo necesario contar con una formación que avale la elaboración de pautas didácticas y la práctica docente en general. Excepto por algunas experiencias aisladas, el desarrollo de pautas didácticas para cursos de informática en general (y de programación en particular) sigue estando basado en cuestiones de opinión de los equipos docentes. Las mismas suelen elaborarse en base a ideas aisladas de cómo los docentes creen que los temas serán comprendidos por los estudiantes o a partir de replicación de experiencias realizadas en otros contextos. A título personal, el autor de este trabajo ha dictado cursos de programación en carreras universitarias durante más de 15 años y ha vivido en carne propia las carencias de no contar con formación docente en el área, teniendo que "aprender docencia sobre la marcha" y diseñar "por instinto" pautas didácticas para sus clases, sin contar con un marco teórico que avale su efectividad. Con el tiempo, su preocupación por estos temas, compartida por el grupo de Didáctica del InCo, llevó a su acercamiento al mismo, comenzando así su formación en CSE. Colaboró en varias de las investigaciones que llevaron al desarrollo del modelo y finalmente realizó este trabajo en el marco de su tesis de maestría.

Como comentario final, durante el desarrollo de este trabajo se presentaron dos avances del mismo. El primero se titula "*Hacia un modelo de investigación en didáctica de la programación*", presentado en la *XLV Conferencia Latinoamericana de Informática* (CLEI 2019, Panamá) [35]. El segundo es una versión extendida del primero y se titula "*A research model in didactics of programming*", publicado en CLEI Electronic Journal (2020) [36].

Capítulo 2

Marco teórico para el modelo: La teoría epistemológica de Jean Piaget

En este capítulo se introduce la teoría epistemológica de Jean Piaget como marco teórico para el modelo de investigación propuesto en este trabajo. La teoría explica el proceso de construcción de conocimiento, en particular conocimiento *científico*, dando descripciones detalladas de los mecanismos y herramientas cognitivas que intervienen en dicha construcción. Es aplicable a todos los dominios de conocimiento científico y en todos los niveles de desarrollo. La teoría, denominada *Epistemología Genética*, tiene como punto central explicar cómo se da la transición desde un nivel de conocimiento hacia otro nivel de más conocimiento [71]. En este sentido, en el resto del trabajo se usa, en múltiples ocasiones, la palabra *conceptualización* como sinónimo de dicha transición (aunque en otros contextos puede no tener el mismo significado).

Los datos que dan sustento a la teoría provienen principalmente de dos fuentes. Por un lado, de estudios empíricos llevados a cabo por Piaget sobre la construcción de conocimiento por parte de los sujetos, desde el nacimiento y hasta la adolescencia, los cuales dieron origen a la *psicología genética* de Piaget [14, 15, 17]. Por otro lado, de un *análisis crítico de la historia de las ciencias*, elaborado por Piaget y García, para investigar el origen y desarrollo de las ideas, conceptos y teorías científicas. En [13], los autores presentan una síntesis de sus investigaciones y brindan una perspectiva de sus explicaciones acerca de la construcción del conocimiento. Encuentran un posible paralelismo entre los mecanismos de desarrollo psicogenéticos relacionados con la evolución de la inteligencia en los niños y el desarrollo sociogenético sobre la evolución de las ideas y teorías principales en diversos dominios de la ciencia. A lo largo de los capítulos, presentan ejemplos de este paralelismo relativos a la historia del álgebra, la geometría, la mecánica y el conocimiento de la física en general.

La teoría establece ciertos paralelismos entre los mecanismos generales que permiten pasar de una forma de conocimiento a otra, tanto en la psicogénesis como en la evolución histórica de las ideas y las teorías científicas, donde la noción más importante de dichos mecanismos es la tríada de etapas, llamadas *intra*, *inter* y *trans* por Piaget y García. A grandes rasgos, la tríada explica el proceso de construcción del conocimiento mediante el paso de una primera etapa enfocada en objetos aislados (etapa *intra*), a otra que toma en cuenta las relaciones entre los objetos y sus transformaciones (etapa *inter*), lo que lleva a la construcción de estructuras generales

que involucran los elementos generalizados y sus transformaciones (etapa *trans*), integrando las construcciones de las etapas previas como casos particulares.

En la sección 2.1 se describen brevemente los principios básicos de la teoría y sus fuentes. En la sección 2.2 se presenta un componente central de la teoría: la *ley general de la cognición*. En la sección 2.3 se enumeran las *herramientas cognitivas* que intervienen en el proceso de construcción del conocimiento. En la sección 2.4 se describe la *tríada de etapas* (*intra*, *inter* y *trans*). Finalmente, en la sección 2.5 se introduce el modelo de investigación en didáctica de la programación, propuesto en base al marco teórico dado por las secciones anteriores.

2.1 Principios básicos y fuentes de la teoría

En la Teoría de Piaget, el conocimiento humano se considera esencialmente activo. Según dice el propio Piaget en [71]: *knowing an object does not mean copying it - it means acting upon it. It means constructing systems of transformations that can be carried out on or with this object* (*conocer un objeto no significa copiarlo - significa actuar sobre él. Significa construir sistemas de transformaciones que pueden ser llevadas a cabo con o sobre dicho objeto*, t.d.a).

Piaget sostiene que el conocimiento, en particular el conocimiento científico, no es estático, sino que está en constante evolución. Según Piaget, el conocimiento en cualquier área científica no se *adquiere*, sino que se *construye* en forma progresiva, en un proceso de construcción y reorganización continua. La teoría no se ocupa de estudiar el significado del conocimiento en sí mismo, sino sus mecanismos de construcción. Esta postura representa un cambio importante respecto de la forma en la cual se estudiaba el problema del conocimiento previo a la Epistemología Genética. Según dice García en [8], su estudio era competencia de la *filosofía especulativa* antes de su "derrumbe" a inicios del siglo XX. García proporciona explicaciones detalladas de los intrincados sucesos históricos que, en cierta forma, "quitaron" a la filosofía el estudio del problema del conocimiento y lo instalaron en el campo de la ciencia, posibilitando así el surgimiento de la *Epistemología Genética* como una teoría científica que explica la construcción del conocimiento. En el marco de dicha ruptura, el conocimiento deja de considerarse una noción existente *a priori* y pasa a considerarse como una que se *construye*, tanto a nivel del sujeto (*psicogénesis*) como del desarrollo histórico (*análisis crítico*).

La teoría plantea que la construcción del conocimiento constituye un proceso cognitivo continuo. Provee un modelo aplicable al estudio de la construcción de conocimiento científico en todos los dominios y en cualquier etapa del desarrollo. Explica la construcción a partir de los orígenes psicológicos de las nociones y operaciones elementales en las que se basa el conocimiento, de su historia y de su sociogénesis. El conocimiento se construye a través de un proceso continuo de interacción con el medio, cuyas fuentes son las acciones sobre lo concreto que, por medio de la reflexión, se transforman en operaciones en el plano del pensamiento. Todo el proceso se

desarrolla de manera gradual y dialéctica, sin que existan límites claramente demarcados entre una etapa y la siguiente. Se trata de una transición permanente, proactiva y retroactiva, en la cual hay una reorganización continua del pensamiento y de las ideas relativas al conocimiento que se está construyendo.

La *Epistemología Genética* estudia y explica los mecanismos de transición desde un estado de menor hacia otro estado de mayor conocimiento. Las nociones de "*menor*" y "*mayor*" tienen connotaciones formales inherentes a la propia naturaleza del dominio de conocimiento que se está construyendo. Es decir, no compete a esta teoría determinar si un determinado estado de conocimiento es superior o inferior a otro, sino a la propia disciplina a la cual corresponden dichos estados (matemática, física, biología, etc.). La teoría explica el proceso de construcción de conocimiento en forma independiente de su naturaleza científica específica. En palabras del propio Piaget en [71]: *We can formulate our problem in the following terms: by what means does the human mind go from a state of less sufficient knowledge to a state of higher knowledge?* (*podemos formular nuestro problema en los siguientes términos: mediante qué mecanismos pasa la mente humana de un estado de conocimiento menos suficiente a otro estado de mayor conocimiento?*, t.d.a).

Las dos fuentes de las cuales se nutre la teoría son la *psicología genética* y el *análisis crítico de la historia de las ciencias*. La *psicología genética* [14, 15, 17] estudia los mecanismos mentales que llevan a la construcción de conocimiento en el sujeto. Surgió con el objetivo de constituir un programa de investigaciones para el análisis de dichos mecanismos mentales, mediante los cuales se construye conocimiento en general y conocimiento científico en particular. Piaget buscó de esta manera obtener evidencia empírica para su teoría epistemológica. De acuerdo con García en [75], al plantearse el estudio del problema del conocimiento, Piaget recurre a la realización de estudios con niños y adolescentes, lo cual resulta novedoso para la época. Tales estudios permiten ver de qué forma el niño empieza a construir conceptos tales como espacio, tiempo o número, así como una lógica que contiene la semilla de las formas más elementales de razonamiento. Según García, la teoría establece que la construcción es continua en el sentido de que no hay discontinuidad en los instrumentos constructivos, lo cual es aplicable a cualquier persona, desde un niño hasta un científico. Establece que la forma en que un niño construye conocimiento no es diferente a la forma en que lo hace un joven, un adulto no "corrompido" por la ciencia, o un científico.

Respecto del *análisis crítico de la historia de las ciencias* [13], Piaget y García introducen dicha expresión con el fin de designar con precisión el tipo de material histórico requerido para un análisis epistemológico del desarrollo del conocimiento científico. No interesó a los autores realizar una cronología del desarrollo de la ciencia, sino más bien un estudio comparativo de los procesos que llevaron a la construcción del conocimiento científico en diversas civilizaciones y contextos socioculturales y económicos. La interpretación de dicho material histórico constituye una base fundamental para entender todo aquello que, posteriormente, fue catalogado como actividad científica y qué papel cumple en la construcción de conocimiento.

En resumen, la teoría propone explicaciones sobre cómo la humanidad construye conocimiento científico, tanto a nivel de los sujetos (*psicología genética*), como a nivel del desarrollo histórico de las ciencias (*análisis crítico de la historia de las ciencias*). El presente trabajo pone el foco en la construcción de conocimiento a nivel de los sujetos, pues busca investigar cómo es que las personas construyen conocimiento relativo a conceptos de programación. En particular, para explicar la construcción de conocimiento en los sujetos, la teoría formula una ley denominada *ley general de la cognición*, que se describe en la siguiente sección.

2.2 Ley general de la cognición

El problema principal del desarrollo cognitivo consiste en determinar el papel de la experiencia y las estructuras operativas del sujeto y en examinar los instrumentos mediante los cuales el conocimiento se ha construido en forma previa a su formalización. Este problema fue estudiado en detalle por Piaget en sus experimentos sobre construcción de conocimiento desde el nacimiento y hasta la adolescencia (dando lugar a la *psicología genética*), en base a cuyos resultados formuló la ley general de la cognición [14]. La misma regula la relación entre el conocimiento y su construcción, generada en la interacción entre el sujeto y los objetos que manipula para resolver problemas o llevar a cabo tareas. Tales objetos pueden ser físicos (por ejemplo, un juguete) o abstractos (por ejemplo: la noción de número). Se trata de una relación dialéctica en la que, a veces, las acciones guían el pensamiento y, otras veces, el pensamiento guía las acciones.

La teoría establece que la fuente del conocimiento se remonta a las acciones del sujeto en interacción con objetos en su medio. Tales acciones constituyen un conocimiento en el plano de la acción, un *saber hacer* sobre los objetos. Tras un proceso reflexivo y nuevas interacciones, las acciones son progresivamente interiorizadas y se transforman en conceptos, dando lugar a lo que Piaget denomina *operaciones* (acciones en el plano del pensamiento). La teoría da cuenta de la continuidad en el proceso de construcción de conocimiento, desde conceptos básicos hasta conceptos complejos, en un proceso regulado por la ley general de la cognición, (también conocida como ley de la toma de conciencia). Los mecanismos que intervienen en la construcción del conocimiento son siempre los mismos (ya sea que se trate de un niño construyendo la noción de número o bien de un matemático investigando una sofisticada teoría nueva). La teoría explica dichos mecanismos y cómo actúan en el proceso de construcción. Piaget ilustra la ley general de la cognición mediante el siguiente diagrama:

$$C \leftarrow P \rightarrow C'$$

donde P es lo que Piaget llama la *periferia* y C y C' lo que llama *centros*. La periferia es la reacción más inmediata y externa del sujeto al confrontar los objetos para resolver un problema o realizar una tarea. Esta reacción se asocia a la búsqueda de un objetivo y al logro de resultados, sin que haya ninguna reflexión sobre cómo se logra el objetivo y por qué lo que se hace funciona (o no funciona). La transición $P \rightarrow C$ representa el

proceso de conceptualización de la coordinación de las acciones realizadas sobre los objetos manipulados (centro *C*), mientras que la transición P → C' representa el proceso de conceptualización de los cambios que tales acciones imponen a los objetos, así como sus propiedades intrínsecas (centro C'). El proceso de transición de P a los centros, representado por las flechas, es gradual y dialéctico, y desemboca en la construcción de los conceptos. Es un proceso de toma de conciencia de métodos y razones donde los elementos *inconscientes* se vuelven *conscientes*.

Las transiciones hacia C y C' explican la conceptualización de las acciones realizadas y de los cambios que ellas producen sobre los objetos. En este proceso, el sujeto utiliza un objeto para un determinado fin y toma nota del resultado (a nivel inconsciente), asimilando el mismo a un esquema mental previo a la utilización del objeto. Esto desencadena que inicialmente no sea capaz de explicar cómo hizo lo que hizo con el objeto y su primera reacción inconsciente es atribuir el resultado a las propiedades del objeto. Progresivamente, las acciones realizadas y la reflexión sobre las mismas generan una reestructura de su esquema mental. Conforme esto sucede, el sujeto comprende que son sus acciones las que generan efectos sobre el objeto, lo cual le permite construir conocimiento. Deja de atribuir los efectos a las propiedades del objeto y comprende que dichos efectos son consecuencia de las acciones. Este proceso se explica por las herramientas cognitivas que se describen en la siguiente sección.

Los mecanismos por los cuales el esquema mental del individuo se reestructura se denominan *asimilación y acomodación* [15]. La asimilación consiste en amoldar los hechos constatados en el plano de la acción al esquema mental del sujeto, lo cual produce un desequilibrio en dicho esquema. Este desequilibrio lleva a un proceso de acomodación, que implica una reestructura del esquema mental que posibilita la integración de los hechos constatados en la acción, generando así un nuevo esquema mental. La asimilación y la acomodación son mutuamente indisociables, se dan en forma permanente y simultánea y ocurren en la construcción de conocimiento de cualquier índole y en todo nivel.

2.3 Herramientas cognitivas

Piaget describe dos herramientas cognitivas que intervienen en el proceso de la toma de conciencia regulado por la ley general de la cognición, denominadas *abstracción* (en sus dos formas: *empírica y reflexiva*) y *generalización* (también en dos formas: *inductiva* y *constructiva*) [14, 15, 16, 17]. En la sección 2.3.1 se describen las dos formas de la abstracción y en la sección 2.3.2 se describen las dos formas de la generalización. Luego, en la sección 2.3.3 se detalla cómo ambas herramientas cognitivas explican la construcción de nuevo conocimiento. Junto con las descripciones, se proponen algunos ejemplos concretos para ilustrar las mismas.

2.3.1 Abstracción (empírica y reflexiva)

La *abstracción empírica* consiste en la abstracción de lo percibido por el sujeto a partir de los objetos. Se trata de la extracción de características comunes a una clase de objetos a partir de las características observadas en un objeto concreto. Es un primer nivel de abstracción, que surge de lo inmediato en la interacción con objetos concretos. Inicialmente, el sujeto intenta naturalmente asimilar dicha interacción concreta como algo general a su esquema mental. Su primer impulso inconsciente es construir relaciones generales, en el plano del pensamiento, a partir de las relaciones concretas experimentadas en el plano de la acción. Cuando el sujeto realiza alguna tarea con un objeto y tiene éxito (o no), inicialmente atribuye el éxito (o el fracaso) a las propiedades del objeto, sin tomar en cuenta las acciones que ha realizado sobre el mismo.

La *abstracción reflexiva* es un proceso doble. En primer lugar, constituye una transposición al plano del pensamiento de relaciones establecidas en el plano de la acción. Las coordinaciones concretas logradas en el plano de la acción se convierten en coordinaciones inferenciales a nivel del pensamiento. Este proceso puede ser más o menos lento, porque implica reconstruir en el último las relaciones establecidas en el primero. Qué tan lento (o no) sea depende de cada sujeto así como de su contexto. Según Coll y Miras en [26], el desarrollo cognitivo depende de cuatro tipos de factores especificados por Piaget (biológicos, de equilibrio, de intercambio entre individuos y de transmisión educativa y cultural). En el plano de la acción, la coordinación se logra por ensayos sobre el conjunto de resultados posibles, los que permiten seleccionar inconscientemente los resultados favorables. En el plano del pensamiento, en cambio, la coordinación inferencial debe construirse generalizando todas las acciones posibles y ubicando a la acción actual en ese conjunto de posibilidades.

En segundo lugar, la abstracción reflexiva es una reconstrucción, en el plano del pensamiento, de las relaciones establecidas en el plano de la acción. Esta reconstrucción agrega un nuevo elemento: la comprensión de las condiciones y las motivaciones, generada por lo que Piaget llama la *búsqueda de razones de éxito (o falla)* [14]. Esta comprensión coloca el caso de éxito como uno más entre los casos posibles realizables en condiciones similares. Al realizar una tarea, el sujeto ya no atribuye el éxito (o el fracaso) de la misma a las propiedades del objeto, sino que comprende que son las acciones realizadas las que producen efectos sobre el mismo.

Para ilustrar las dos formas de abstracción, se brinda el siguiente ejemplo concreto. Supóngase un niño que empuja un juguete sobre una mesa y éste cae al suelo. El primer impulso del niño es pensar que "*se cayó porque es pesado*" (abstracción *empírica*). Tras múltiples interacciones con el juguete, el niño asimila que es su acción sobre el objeto la que genera el cambio de estado, y no la propiedad de "*ser pesado*". Llega un punto en que ya no piensa que "*se cayó porque es pesado*", sino que "*se cayó porque yo lo empujé*" (abstracción *reflexiva*). El caso de éxito en este ejemplo está dado por la caída del juguete cuando es empujado. El niño reconstruye mentalmente la acción realizada

y logra así concientizar que la acción aplicada por él sobre el objeto es la que genera un cambio de estado en el mismo.

2.3.2 Generalización (inductiva y constructiva)

Piaget denomina *generalización inductiva* a un proceso mediante el cual el sujeto intenta generalizar, hacia otros problemas nuevos, el conocimiento construido en un problema anterior, sin tener en cuenta las características del nuevo problema. Al enfrentar un nuevo problema que presenta similitudes y diferencias con respecto al anterior, el individuo intenta aplicar (de forma inconsciente) el conocimiento construido en ese problema previo. La generalización inductiva consiste solamente en transferir hacia el nuevo problema los hechos o relaciones constatados en el problema anterior. Constituye una generalización de "*algunos*" a "*todos*" en la cual se suman ideas preconcebidas, traídas del problema anterior, que falsean la lectura que el sujeto hace del nuevo problema. Esto provoca un desequilibrio de su esquema mental, que debe transformarse para alcanzar un nuevo equilibrio.

Piaget denomina *generalización constructiva* a un proceso que, a diferencia de la generalización inductiva, no consiste en asimilar nuevos contenidos a formas ya construidas, sino a engendrar nuevas formas y contenidos, que conducen a nuevas organizaciones estructurales, haciendo posible la construcción de conocimiento para el nuevo problema. Un proceso de inferencias y reflexiones sobre las acciones u operaciones del sujeto, mediante generalización constructiva, da lugar a nuevos métodos y posibilita construir nuevas estructuras. La generalización constructiva no implica una generalización de "*algunos*" a "*todos*", sino un proceso de adaptación que permite resolver el nuevo problema a partir del conocimiento construido a raíz del problema anterior, posibilitando la construcción de conocimiento nuevo, adaptado al nuevo problema.

Para ilustrar las dos formas de generalización, se propone la siguiente ampliación al ejemplo de la sección anterior. Una vez que el niño comprende que el juguete se cae al empujarlo, empuja luego una pared, esperando inconscientemente el mismo resultado: que la pared se caiga (*generalización inductiva*). Tiene la idea preconcebida de que cualquier objeto se mueve cuando se lo empuja. Constatar que la pared no se mueve al empujarla, como sí ocurre con el juguete, genera un desequilibrio en su esquema mental. El caso de éxito en este ejemplo habría sido que la pared se mueva, tal como sucede con el juguete. Ante la falla, el niño eventualmente llega a comprender que no todo objeto se mueve al empujarlo. Concluye que no alcanza con empujar un objeto para que se mueva, sino que hay otros factores que lo impiden, por ejemplo, que la pared está adherida a la casa (*generalización constructiva*).

2.3.3 Herramientas cognitivas y construcción de conocimiento

La *abstracción empírica* y la *generalización inductiva* permiten realizar constataciones en el plano de la acción pero no permiten construir conocimiento nuevo por sí solas. La *abstracción reflexiva* y la *generalización constructiva* hacen posible la toma de conciencia de la coordinación de las acciones y de los cambios que imponen a los objetos involucrados, procediendo de la periferia a los centros (en el marco de la ley general de la cognición). Posibilitan la construcción de nuevas formas y nuevos contenidos, construyendo así conocimiento nuevo. La *abstracción reflexiva* lo explica para el caso de situaciones puntuales, en tanto la *generalización constructiva* lo explica a nivel de nuevas situaciones. La combinación de ambas herramientas cognitivas explica la expansión en la construcción de conocimiento por parte del sujeto, permitiendo la apertura a la construcción de nuevas formas de conocimiento.

En la distinción entre *abstracción empírica* y *abstracción reflexiva* es que se construye conocimiento sobre un problema dado. A partir de la coordinación de las acciones realizadas sobre el objeto, el sujeto pasa a un nivel de representación mental de tales acciones. La abstracción reflexiva consiste en relacionar dichas representaciones en un todo ordenado. Así como los procesos de asimilación y acomodación son mutuamente indisociables, la abstracción empírica y la abstracción reflexiva no se dan de manera aislada y por separado, sino que por momentos predomina una y por momentos predomina la otra. Ambas formas de abstracción se dan en forma gradual y dialéctica, siendo la abstracción reflexiva la que posibilita reestructurar el esquema mental del sujeto, conduciendo así a la construcción de conceptos. En el marco de este proceso es que opera la ley general de la cognición.

Para ilustrar el vínculo entre las dos formas de abstracción y la ley general de la cognición, se considera nuevamente el ejemplo del niño de las dos secciones anteriores. Inicialmente, cuando el niño empuja el juguete y atribuye el resultado a la propiedad de "*ser pesado*" (*abstracción empírica*) su pensamiento está en la periferia (P). Posteriormente, se traslada progresivamente hacia los centros C y C'. El centro C está dado por la comprensión de que la acción aplicada sobre el juguete genera el cambio de estado. El centro C' está dado por la comprensión de que tiene la propiedad de "*ser pesado*", pero que por sí sola no basta para producir movimiento en el objeto, sino que es la acción de empujarlo la que genera el cambio (*abstracción reflexiva*).

De la misma forma, la distinción entre *generalización inductiva* y *generalización constructiva* permite construir conocimiento sobre un nuevo problema a partir del conocimiento construido sobre un problema anterior. Nuevamente, la ley general de la cognición opera en este nuevo proceso. El conocimiento construido anteriormente constituye la periferia (P) en el nuevo problema. A partir de ese punto, el pensamiento inicia nuevas transiciones hacia centros C y C' enmarcados en el contexto del nuevo problema. Para ilustrar esto, se retoma nuevamente el ejemplo del niño, quien inicialmente cree que la pared, como el juguete, se caerá tras empujarla (*generalización inductiva*). Esto constituye la periferia (P) del nuevo problema. El centro C está dado

por la comprensión de que la acción de empujar la pared no es suficiente para moverla, mientras que el centro C' está dado por la comprensión de que la pared se encuentra adherida a la casa, lo que impide que se mueva a pesar de la acción ejercida (*generalización constructiva*). El niño comprende finalmente que no cualquier objeto se mueve tras ser empujado. El conocimiento construido en la situación previa sirve como base para la construcción de nuevo conocimiento.

2.4 Tríada de etapas: *intra*, *inter* y *trans*

Como se indica al inicio del capítulo, la noción más importante que describe los mecanismos generales de pasaje de un nivel de conocimiento considerado inferior a otro nivel superior es la denominada tríada de etapas *intra*, *inter* y *trans* por parte de Piaget y García [13]. De acuerdo con ellos, el conocimiento (cualesquiera sean su dominio y nivel de desarrollo) se construye pasando por una primera etapa enfocada en objetos aislados (etapa *intra*), pasando luego por una segunda etapa que toma en cuenta relaciones entre dichos objetos y sus transformaciones (etapa *inter*) y llegando a una tercera etapa en la cual se construye un esquema general que involucra tanto los objetos generalizados como sus transformaciones (etapa *trans*).

De acuerdo con la teoría, el pasaje de cada etapa a la siguiente es gradual y dialéctico, ocurre en cualquier dominio de conocimiento y en cualquier nivel de desarrollo. El pasaje ocurre tanto a nivel de la construcción de conocimiento por parte del sujeto (*psicogénesis*), como a nivel del desarrollo del conocimiento científico por parte de la humanidad (*análisis crítico de la historia de las ciencias*). En el primer caso, se trata del pasaje individual que el pensamiento de cada persona realiza por las tres etapas durante su proceso de construcción sobre cualquier área en particular. Por ejemplo, cómo una persona construye conocimiento sobre termodinámica. En el segundo caso, se trata de la evolución del conocimiento de cada disciplina científica a nivel global, en el marco del desarrollo histórico de la misma. Por ejemplo, cómo evolucionó la termodinámica como disciplina científica a lo largo de la historia. El modelo que se propone en el presente trabajo se concentra en el primer caso, ya que busca estudiar cómo los sujetos construyen conocimiento relativo a temas de programación. Es decir, cómo su pensamiento transita por las tres etapas de la tríada durante el proceso.

Según Piaget y García, la tríada constituye la expresión de las condiciones que los mecanismos de asimilación y acomodación imponen a toda construcción de conocimiento. A partir de la interacción de un sujeto con objetos aislados (etapa *intra*) se produce un desequilibrio en su esquema mental que lo lleva a un proceso de acomodación. Los hechos constatados en la interacción con objetos concretos se asimilan a su esquema mental anterior, construyendo así un nuevo esquema mental. En forma gradual y dialéctica, esta reestructura da paso a la construcción de relaciones entre los objetos manipulados y a la caracterización de los cambios sufridos por tales objetos a raíz de la manipulación (etapa *inter*). Posteriormente, otro proceso de asimilación y acomodación (nuevamente gradual y dialéctico) lleva al sujeto a una nueva modificación de su esquema mental para construir estructuras generales (etapa

trans), en las cuales los hechos específicos inicialmente constatados al manipular objetos concretos se integran como casos particulares de dichas estructuras generales.

Desde la perspectiva psicogenética, cada etapa de la tríada refiere al nivel de conocimiento del sujeto en cada una, los mecanismos que describen cómo pasa de una etapa a la siguiente se explican por la *ley general de la cognición* (sección 2.2) y en cada pasaje intervienen las herramientas cognitivas: *abstracción* y *generalización* (sección 2.3). La etapa *intra* refiere al conocimiento en el plano de la acción al manipular objetos concretos. La etapa *inter* refiere al conocimiento que se conceptualiza a partir de este, donde las acciones se transforman en operaciones en el pensamiento. La etapa *trans* refiere al conocimiento que se generaliza a partir del saber conceptualizado y se integra en estructuras mentales generales.

Piaget y García proponen en [13] explicaciones detalladas de la naturaleza de cada etapa de la tríada tanto en el campo de la física (por ejemplo, involucrando nociones como fuerza, peso, calor, etc.) como del conocimiento lógico-matemático (por ejemplo, nociones como número, conjunto, etc.). Más allá de las diferencias inherentes a cada dominio del conocimiento, concluyen que el pasaje por las tres etapas se da en toda construcción de conocimiento científico, sin importar si se trata de ciencias naturales o exactas. También detallan cómo operan las herramientas cognitivas: *abstracción* (*empírica* y *reflexiva*) y *generalización* (*inductiva* y *constructiva*) durante el proceso de construcción en cualquier dominio científico. Brindan ejemplos relativos a la construcción de nociones vinculadas a ciencias naturales (por ejemplo: peso y velocidad en física) y a ciencias exactas (por ejemplo: área y distancia en geometría). Sin importar la naturaleza específica del conocimiento, concluyen que ambas herramientas cognitivas siempre intervienen en todo el proceso de construcción.

Un aspecto importante es que el pasaje de cada etapa de la tríada a la etapa siguiente no se caracteriza por un incremento en el volumen de conocimientos construidos respecto a la etapa anterior, sino por un proceso de reorganización del esquema mental del individuo. El desarrollo cognitivo no se da por acumulación de conocimientos, sino que implica, en cada nivel, una reconstrucción de lo conceptualizado en el nivel anterior. Se trata de una reorganización de conocimientos a la luz de nuevos elementos y de una reinterpretación de los conocimientos de base previamente construidos.

La transición de cada etapa a la siguiente se da de manera gradual y dialéctica y forma parte de un proceso que no culmina tras haber construido determinado conocimiento en particular sino que se repite permanentemente. Esto posibilita la construcción de nuevo conocimiento en niveles superiores. Este proceso se puede pensar como una secuencia de muchas tríadas *intra-inter-trans* encadenadas, una a continuación otra, en un proceso continuo que no termina. Las estructuras que un sujeto construye al alcanzar la etapa *trans* vinculada a la construcción de un conocimiento determinado dan paso a su vez a una nueva etapa *intra* correspondiente a un nivel de conocimiento superior, la que a su vez conducirá a una nueva etapa *inter*, a una nueva etapa *trans*, y así sucesivamente.

Para ilustrar la tríada de etapas, se propone como ejemplo hipotético un niño que cuenta piedritas repartidas en dos montones, habiendo tres piedritas en el primer montón y cinco en el segundo. Cuenta las del primer montón y luego las del segundo y posteriormente lo hace al revés. En ambos casos, obtiene como resultado ocho piedritas. El niño manipula objetos concretos (las piedritas) y realiza una operación específica sobre ellas (las cuenta). Constata, en el plano de la acción, que la suma es una operación conmutativa, pero aún no tiene conceptualizada dicha propiedad como algo general. Su pensamiento está en la etapa *intra*. Piensa en piedritas y no en números. La etapa *intra* supone el análisis de casos particulares no vinculados entre sí, o vinculados de manera insuficiente. El niño cuenta inicialmente piedritas, pero también cuenta otros objetos en otros momentos, como ser juguetes o lápices. En todos los casos, constata en el plano de la acción el cumplimiento de la propiedad conmutativa de la suma, pero siempre trabajando con los objetos concretos y sin tener conciencia aún de que se trata de una propiedad general que trasciende a dichos objetos concretos. En forma progresiva, su esquema mental pasa por un proceso de acomodación y asimila que puede contar las piedritas empezando por cualquiera de los dos montones y obtener el mismo resultado.

Las operaciones concretas inicialmente constatadas en la acción se transforman luego en operaciones en el plano del pensamiento. El niño comprende que siempre que cuente las piedritas, obtendrá el mismo resultado tras contarlas en cualquier orden, sin necesitar volver a realizarlo expresamente con piedritas concretas. Lo mismo sucede con las cuentas de otros objetos. En todos los casos, reconstruye dichas cuentas a nivel del pensamiento, logrando abstraerse de la manipulación de objetos específicos (etapa *inter*) pero sin integrarlos aún a un esquema general aplicable a todos los casos. Más adelante, el niño comprende que contar objetos de cualquier tipo, sin importar en qué orden lo haga, produce siempre el mismo resultado, aun cuando se trate de objetos concretos que nunca ha contado expresamente, llegando eventualmente a comprender que siempre que sume dos números cualesquiera, obtiene el mismo resultado, sin importar en qué orden sume. Generaliza finalmente la propiedad conmutativa de la suma (etapa *trans*), habiendo integrado la interacción inicial con objetos concretos (piedritas, juguetes o lápices) como casos particulares. Todo el proceso (desde la interacción con las piedritas hasta la construcción de la noción de conmutatividad de la suma) es gradual y dialéctico y se explica por la ley general de la cognición. En él intervienen las herramientas cognitivas mencionadas en la sección anterior. A su vez, este conocimiento sobre la propiedad conmutativa de la suma será la base para que el niño construya nuevo conocimiento relativo a, por ejemplo, la conmutatividad de otras operaciones, como ser la multiplicación. Dicho conocimiento previo formará parte de una nueva etapa *intra* vinculada a la construcción de conocimiento nuevo. Su pensamiento pasará luego por una nueva etapa *inter* y llegará eventualmente a una nueva etapa *trans*, construyendo así conocimiento sobre la propiedad conmutativa de la multiplicación.

2.5 Modelo de investigación en didáctica de la programación

El modelo de investigación en didáctica de la programación que se propone en este trabajo constituye una síntesis de múltiples investigaciones realizadas y procura contribuir a explicar cómo las personas construyen conocimiento sobre conceptos de programación. Se basa en la aplicación de diversos conceptos de la Teoría de Piaget, destacando entre ellos la tríada de etapas *intra-inter-trans*. Para ello, el modelo interpreta cada etapa de la tríada en el campo específico de algoritmos, estructuras de datos y programas. Explora la relación entre el *saber hacer* en el plano de la acción (conocimiento *instrumental*, etapa *intra*) de los sujetos al resolver tareas, aplicando métodos asimilables a *instancias* concretas de algoritmos, el conocimiento que construyen en el plano del pensamiento (conocimiento *conceptual*, etapa *inter*) sobre los *algoritmos* aplicados y el conocimiento *formal* (etapa *trans*) sobre los *programas* que los implementan. El modelo estudia cómo el conocimiento en el primer nivel se transforma en conocimiento en los siguientes dos niveles. Esta interpretación de las tres etapas surge tras estudios empíricos llevados a cabo durante años de trabajo, incluyendo el que se presenta en el capítulo 3. Las palabras *instrumental*, *conceptual* y *formal* se introducen como terminología en el modelo para referir específicamente a conocimiento sobre programación y se utilizan para denotar, respectivamente, los niveles de conocimiento en las etapas *intra*, *inter* y *trans*.

En el contexto de la investigación sobre construcción de conocimiento relativo a conceptos de programación, el modelo interpreta que el conocimiento de todo sujeto en la etapa *intra* se evidencia por la aplicación de instancias concretas de algoritmos en el plano de la acción (tales como resolución de juegos, ordenación de objetos, búsquedas, etc.). Dicho conocimiento se construye por sucesivas repeticiones de la tarea. Por ejemplo, si se trata de un juego, el sujeto intenta jugar repetidas veces hasta que logra dominarlo en la acción, pero aún no expresa de manera consciente cómo lo resuelve. En la etapa *inter*, el conocimiento se evidencia por su capacidad para describir en lenguaje natural los algoritmos conceptualizados a partir de las instancias concretas iniciales y explicar por qué funcionan y producen el resultado esperado. En la etapa *trans*, el conocimiento se evidencia por su capacidad de formalizar dichos algoritmos en un lenguaje *formal*, mediante la escritura de *programas* que los implementan, junto con la habilidad de explicar cómo y por qué funcionan cuando se ejecutan en una computadora.

En la sección 2.5.1 se brindan algunos antecedentes que llevaron a la construcción del modelo. En la sección 2.5.2 se describe cómo la postura ontológica sobre la noción de *programa* influyó en el modelo. En la sección 2.5.3 se enuncia una extensión a la ley general de la cognición para explicar construcción de conocimiento sobre programas. Finalmente, en la sección 2.5.4 se presenta una síntesis del modelo y se introduce el estudio a desarrollar en el capítulo 3.

2.5.1 Antecedentes del modelo

Las primeras investigaciones realizadas en Uruguay, basadas en la teoría Piagetiana, para investigar la construcción de conocimiento sobre conceptos de programación, fueron llevadas a cabo en la primera década del siglo XXI [6, 31, 32, 34]. Entre otras razones, fueron motivadas con el fin de establecer un área de educación en informática basada en fundamentos teóricos sólidos, como sucede con otras áreas, por ejemplo la didáctica de la matemática. En particular, la teoría *APOS* (presentada en la sección 1.4.1), tuvo influencia en los primeros estudios y en la adopción de la teoría de Piaget como marco teórico, aunque con diferencias en su aplicación por el carácter específico de la disciplina informática. La evolución de los mencionados estudios llevó a la conformación del grupo de investigación en Didáctica de la Informática del InCo (Instituto de Computación) de la Facultad de Ingeniería, Universidad de la República (UdelaR), del cual el autor del presente trabajo forma parte.

Los primeros estudios empíricos tuvieron como objetivo investigar el proceso de construcción de conocimiento sobre algoritmos básicos y estructuras de datos por parte de estudiantes novatos y se centraron mayormente en estudiar el pasaje de la etapa *intra* a la etapa *inter*. La metodología aplicada en ellos fue fundamentalmente la conducción de entrevistas clínicas similares a las de Piaget en sus estudios de psicología genética. Las herramientas teóricas que guiaron las investigaciones son la *abstracción* y *generalización*, en sus distintas formas, junto con la *ley general de la cognición*, presentadas en las secciones anteriores. Por otra parte, el estudio del concepto de *programa* desde un punto de vista ontológico e histórico, junto con nuevos estudios [29, 30, 33] centrados en investigar la construcción de conocimiento sobre la implementación de algoritmos y su ejecución en una computadora, sentaron las bases para estudiar el pasaje de la etapa *inter* a la etapa *trans* (en el cual el presente trabajo pone el foco). Tales estudios condujeron a una extensión de la ley general de la cognición de Piaget (que se presenta en la sección 2.5.3) para abarcar el conocimiento sobre *programas* como *objetos ejecutables*. Esta elaboración teórica constituye una de las principales contribuciones del grupo de investigación, abordando los estudios epistemológicos y didácticos sobre ciencia de la computación desde la perspectiva de una teoría científica del conocimiento como es la teoría de Piaget. El modelo constituye la síntesis del proceso de investigación regulado por la tríada de etapas *intra-inter-trans*, donde el pasaje de cada etapa a la siguiente es explicado por medio de los estudios empíricos realizados y su relación con la teoría.

2.5.2 Naturaleza dual de los programas

La introducción del enfoque histórico y ontológico de la noción de *programa* permitió completar la interpretación de la tríada *intra-inter-trans* mencionada en la sección anterior. Para ello, fue fundamental el aporte del *Núcleo interdisciplinario Filosofía de la Ciencia de la Computación* (NI FCC) [74]. En el marco del trabajo realizado por el NI FCC, se estudió la naturaleza *dual* del concepto de programa, en línea con la noción propuesta por Moor en [53]: *a computer program can be understood on two levels. Physically, computer programs may be a series of punched cards, configurations on*

magnetic tape, or in any number of other forms. Symbolically, computer programs are understood as instructions to a computer (*un programa de computadora puede entenderse en dos niveles. Físicamente, los programas de computadora pueden ser una serie de tarjetas perforadas, configuraciones en cinta magnética, o cualquier número de otras formas. Simbólicamente, los programas de computadora se entienden como instrucciones a una computadora*, t.d.a). Según esta noción, los programas pueden entenderse tanto desde una perspectiva *física* como desde una perspectiva *simbólica*. Por un lado, como instrucciones ejecutadas por una computadora, cuyo significado no es relevante para la misma. Por otro lado, como símbolos que tienen un significado para las personas que los escriben. En línea con dicha noción, en el marco del presente modelo, una comprensión cabal del concepto de *programa* implica la construcción de conocimiento sobre dos aspectos: la parte simbólica del programa (parte *textual*) y la parte física (parte *ejecutable*).

En relación a la parte *textual*, la construcción de conocimiento en la etapa *trans* implica lograr representar, de manera *formal*, el conocimiento sobre el algoritmo ya conceptualizado (etapa *inter*) para lo cual se requiere hacer uso de un *formalismo*. Un formalismo o lenguaje formal consiste en una notación distinta del lenguaje natural que permite expresar, con cierto grado de rigor, conceptos pertenecientes a algún dominio de naturaleza científica. Posee sus propias reglas de sintaxis y semántica y constituye por sí mismo un objeto sobre el cual el sujeto construye conocimiento. Existen múltiples formalismos aplicables a diversos dominios científicos, algunos más rigurosos que otros. Por ejemplo, la notación para fórmulas lógicas de primer orden o la notación utilizada para expresar ecuaciones algebraicas mediante variables, igualdades y operadores aritméticos, son ejemplos de formalismos utilizados en lógica y matemática. A su vez, la notación de vectores, en términos de magnitud, dirección y sentido es un ejemplo de un formalismo usado en física. En el marco del presente modelo, el formalismo utilizado es un *lenguaje de programación*.

Para expresar conceptos utilizando un formalismo, el sujeto necesita construir conocimiento sobre ese formalismo. Para ello, su pensamiento transita las mismas etapas que para construir conocimiento científico de cualquier índole. En el proceso, construye conocimiento tanto acerca de la *sintaxis* del formalismo (cómo expresar conceptos de manera correcta siguiendo sus reglas sintácticas) como de la *semántica* del mismo (qué significado tienen los símbolos utilizados). Piaget y García denominan *thematized knowledge* (*conocimiento tematizado*, t.d.a) en [13] al conocimiento que involucra el uso de formalismos. Allí brindan abundantes ejemplos, especialmente en el terreno de la matemática y de la física, de la relación existente entre la etapa *trans* y el uso de lenguajes formales para expresar conocimiento formal. En ocasiones, el conocimiento sobre el formalismo se construye en paralelo con el conocimiento sobre el algoritmo en cuestión y otras veces ocurre con antelación (cuando el sujeto usa un lenguaje ya conocido para implementar un nuevo algoritmo sobre el cual construye conocimiento).

Estudios previos sugieren que el uso de un *formalismo intermedio* entre el lenguaje natural y el lenguaje de programación resulta de utilidad para facilitar al sujeto expresar el algoritmo usando el lenguaje formal (y para construir conocimiento sobre el propio lenguaje formal) [6, 29, 30, 31, 32, 33]. El formalismo intermedio puede ser *pseudocódigo*, *máquinas de estado*, *diagramas de flujo* o cualquier otro cuyas reglas de sintaxis y semántica sean menos rigurosas que las de un lenguaje de programación, pero lo suficiente como para expresar con claridad los pasos del algoritmo como una primera aproximación a su expresión en el lenguaje formal, sin que aspectos propios de este último introduzcan dificultades adicionales. Contribuye a consolidar la conceptualización del algoritmo (etapa *inter*) y a la vez resulta de ayuda para iniciar el pasaje a la etapa *trans*, como se ve en el capítulo 3. El *pseudocódigo* es el formalismo intermedio usado para el estudio empírico en dicho capítulo, presentado mediante un conjunto de instrucciones con reglas de sintaxis y semántica fáciles de comprender, pues están a medio camino entre lenguaje natural (que el sujeto ya domina) y un lenguaje de programación (que puede dominar en mayor o menor medida).

En relación a la parte *ejecutable*, la construcción de conocimiento implica comprender que un *agente externo* (la computadora) se encarga de ejecutar las instrucciones del programa. La construcción de conocimiento sobre programas, entendidos como objetos ejecutables por una computadora, introduce como desafío la comprensión de aspectos relativos a la ejecución por parte de una *máquina*. Es necesario que el sujeto comprenda cómo hace la computadora para realizar acciones sobre las estructuras de datos de un programa. Para explicar este proceso de construcción de conocimiento, se propone una extensión a la ley general de la cognición, que se describe en la siguiente sección. Dicha extensión surgió del estudio empírico realizado en [33] ante la necesidad de explicar la construcción de conocimiento sobre la parte ejecutable a partir de las acciones ejecutadas por el agente externo, en vez de las acciones realizadas por el propio sujeto y de sus explicaciones sobre cómo y por qué tiene éxito en su solución.

De forma similar al uso de un formalismo intermedio para ayudar al sujeto en la construcción de conocimiento sobre la parte *textual* del programa, estudios previos [29, 30] sugieren que la introducción de un mecanismo denominado *automatización* resulta de utilidad para ayudarlo a construir conocimiento sobre la parte *ejecutable*. Consiste en que otra persona oficia de *agente externo*, actuando como un robot imaginario que ejecuta los pasos (*formalismo intermedio*) o las instrucciones (*lenguaje de programación*) escritos por el sujeto. La automatización ayuda en varias formas a facilitar la construcción. Permite visualizar el comportamiento generado por los pasos o instrucciones que el sujeto escribe, facilitando su comprensión así como la detección de errores y/o comportamientos no previstos. Cuando se aplica sobre las *instrucciones* del programa, posibilita además observar su efecto sobre las *estructuras de datos* manipuladas y la ocurrencia de situaciones que podrían generar errores de *ejecución*. Si bien su mayor beneficio radica en la parte *ejecutable*, indirectamente también ha probado ser útil para detectar errores de sintaxis y/o semántica, ayudando de este modo en la construcción de conocimiento sobre el propio formalismo y sobre la parte *textual*, debido a la relación dialéctica entre ésta y la parte *ejecutable*.

2.5.3 Extensión a la ley general de la cognición

Desde la perspectiva de la *psicogénesis*, la teoría señala la interacción con los objetos como la fuente de construcción de conocimiento y la describe mediante la *ley general de la cognición* (introducida en la sección 2.2). Piaget estudió en profundidad los elementos que intervienen en dicha construcción para problemas y soluciones algorítmicas aplicadas por los sujetos a objetos (físicos y/o abstractos) manipulados por ellos mismos (mediante acciones manuales y/o mentales). Para el caso del conocimiento sobre *programas* como objetos *ejecutables*, los estudios empíricos realizados por el grupo de investigación revelaron que la ley general de la cognición no era suficiente, ya que las acciones de un programa no son ejecutadas por los propios sujetos, sino por un *agente externo*: la computadora.

En [33] se describe una investigación que tuvo, como uno de sus resultados, la formulación de una *extensión* a la mencionada ley para explicar esto. Cabe señalar que se da en el marco de la construcción de conocimiento *formal* dado que un *programa*, para ser ejecutado, debe estar escrito en un *lenguaje de programación*. El conocimiento *formal* fue estudiado en profundidad por Piaget y Garcia desde la perspectiva del *análisis crítico de la historia de las ciencias* [13], previamente al establecimiento de la informática como una de ellas. Por lo tanto, fue necesario elaborar la extensión a la ley para estudiar la construcción de conocimiento sobre programas. Dicha construcción implica la conceptualización, por parte de los sujetos, de aspectos relativos a la relación de causa y efecto entre la parte *textual* del programa y la parte *ejecutable*. Esa relación se da entre las *instrucciones* escritas por ellos y el resultado de su ejecución por la máquina, así como entre la representación formal de las *estructuras de datos* y sus representaciones *físicas* en memoria durante la ejecución. Deben comprender las condiciones que hacen que la computadora ejecute exitosamente el programa originalmente expresado en el texto del mismo. En analogía con el diagrama de Piaget, se propuso el siguiente diagrama en [33] para describir la extensión propuesta:

$$\underbrace{C \leftarrow P \rightarrow C'}$$
$$newC \leftarrow newP \rightarrow newC'$$

La primera línea expresa la ley general de la cognición tal y como fue formulada por Piaget, la cual, en el contexto de la construcción de conocimiento sobre temas de algoritmia, regula la construcción de conocimiento *conceptual* en relación a un problema algorítmico y al algoritmo que le da solución, tras su aplicación con éxito en la acción. El pensamiento del sujeto parte de la periferia P (conocimiento *instrumental* al aplicar el algoritmo por sí mismo) y transita a los centros C y C' (conceptualización de las *acciones* realizadas en la aplicación del algoritmo y de los *cambios* que imponen a los objetos, respectivamente), interviniendo en el proceso la herramienta cognitiva *abstracción* (en sus dos formas, *empírica* y *reflexiva*). En términos de la tríada, la primera línea expresa el pasaje de la etapa *intra* a la etapa *inter*.

El sujeto debe ser capaz de escribir un *programa* que implemente el algoritmo conceptualizado, lo cual implica la construcción de conocimiento *formal* sobre el mismo. La relación de causa y efecto entre las instrucciones del programa escritas por el sujeto y el comportamiento de las mismas al ser ejecutadas por la computadora se representa mediante la segunda línea del diagrama. El conocimiento *conceptual* construido por el sujeto constituye lo que se denomina *newP* (nueva periferia) en el diagrama. La construcción de conocimiento sobre la relación de causa y efecto mencionada se explica mediante las transiciones hacia nuevos centros (*newC* y *newC'*) los cuales representan, respectivamente, la conceptualización de cómo la computadora ejecuta las *instrucciones* del programa y los cambios que dichas instrucciones imponen sobre las *estructuras de datos* del mismo. Nuevamente, la herramienta cognitiva que interviene es la *abstracción* (tanto empírica como *reflexiva*).

El pasaje de la etapa *inter* a la etapa *trans* representa la construcción de conocimiento *formal* sobre el programa que implementa el algoritmo. Dicha construcción se logra por medio de las transiciones desde *newP* hacia *newC* y *newC'*, junto con un factor extra: la comprensión de la relación dialéctica que subyace entre la primera y la segunda línea del diagrama. Esto es, por un lado, la comprensión de la correspondencia entre las *acciones* del algoritmo realizadas por el sujeto y su contraparte en términos del programa, dada por las *instrucciones* que las expresan (conocimiento sobre la parte *textual*). Por otro lado, la comprensión de la correspondencia entre los *cambios* que las acciones del sujeto imponen a los objetos y su contraparte, dada por las modificaciones que la ejecución de las *instrucciones* imponen a las estructuras de datos del programa (conocimiento sobre la parte *ejecutable*). El conocimiento *formal* sobre el programa se construye cuando el sujeto comprende la relación de causa y efecto entre las instrucciones del programa y sus efectos en la ejecución (segunda línea del diagrama) y además comprende la correspondencia entre el *programa* y el *algoritmo* previamente conceptualizado (relación dialéctica de la segunda línea del diagrama con la primera).

2.5.4 Ilustración del modelo mediante un estudio empírico

El modelo propuesto en el presente trabajo surge como síntesis de las distintas investigaciones llevadas a cabo por el equipo de investigación a lo largo de los años. Las primeras se centraron principalmente en estudiar la construcción de conocimiento *conceptual* sobre temas de algoritmia y las siguientes empezaron a profundizar en el estudio de la construcción de conocimiento *formal* sobre programas. La interpretación del proceso *completo* de construcción de conocimiento (desde lo *instrumental* hasta lo *formal*) en términos de la tríada *intra-inter-trans* no surgió desde el inicio de las investigaciones sino que, por el contrario, resulta de una revisión global de todo el trabajo realizado. Parte de dicha revisión fue llevada a cabo como preparación del trabajo que aquí se presenta. Además, las investigaciones previas condujeron a formular la extensión a la ley general de la cognición presentada en la sección anterior. Tras la introducción de la reinterpretación en términos de la tríada, se constató que los primeros trabajos ponían el foco en el estudio del pasaje de la etapa *intra* a la etapa *inter* para problemas algorítmicos diversos y que resultaba necesario profundizar en

el estudio del pasaje de la etapa *inter* a la etapa *trans*. Es, por tanto, en este último pasaje donde pone énfasis el presente trabajo.

En el capítulo 3 se presenta un estudio empírico detallado que complementa el trabajo previo que llevó a la formulación de la extensión a la ley general de la cognición para explicar la construcción de conocimiento de *programas* como objetos con característica *dual* (parte *textual* y parte *ejecutable*). Dicho estudio analiza el proceso *completo* (desde la etapa *inter* hasta la etapa *trans*) de construcción de conocimiento sobre un problema concreto: *búsqueda de un elemento en una secuencia no ordenada de elementos* (también conocido como *búsqueda lineal*). Se analizan las etapas desde la resolución, por parte de las personas que participan del estudio, de una instancia concreta del problema en el plano instrumental (etapa *intra*), pasando por la construcción de conocimiento conceptual sobre el algoritmo que construyen durante el proceso (etapa *inter*) y finalizando con la escritura de un programa que lo implementa en un lenguaje de programación (etapa *trans*). El estudio ilustra cómo el modelo explica el proceso completo de construcción de conocimiento para el problema elegido, siendo además el primer estudio que analiza en detalle el proceso de transición del pensamiento desde *newP* hacia *newC* y *newC'*, expresado por la segunda línea del diagrama de la ley extendida, así como su relación dialéctica con la primera línea. En particular, se explora el rol que juegan la *automatización* y el uso de un *formalismo intermedio* como facilitadores del pasaje de la etapa *inter* a la etapa *trans*, a partir de la evidencia sobre sus beneficios sugerida por los estudios previos.

La etapa *trans* abarca no solo la construcción de conocimiento formal sobre un problema en particular, sino también de un esquema más general que involucra objetos generalizados junto con sus transformaciones, conforme a lo expresado en la sección 2.4. Si bien el estudio empírico se centra mayormente en el proceso de construcción completo desde la etapa *intra* hasta la etapa *trans* para el problema elegido, también se decidió incluir, como parte del mismo, algunos elementos relativos a la construcción de dicho esquema más general, a efectos de sentar algunas bases para trabajos futuros que profundicen en ello. En particular, se estudian algunos aspectos relativos a la construcción de la noción de *elemento genérico* y también cómo el conocimiento formal construido sobre la búsqueda lineal posibilita enfrentar a los participantes con un *nuevo* problema que presenta similitudes y diferencias con el problema ya resuelto, de modo de avanzar a la construcción de conceptos más generales y complejos.

El estudio empírico surge a partir de una instancia concreta del problema de búsqueda lineal, en la cual los participantes deben buscar, en el plano de la acción, un elemento dado dentro de una secuencia con una cantidad *específica* de elementos (por ejemplo, una secuencia concreta de 10 valores numéricos que se les brinda al inicio del estudio, en forma de tarjetas numeradas). Más tarde, al escribir el programa, utilizan una estructura de *arreglo* que almacena dichos valores al implementar el programa que resuelve la búsqueda. Parte del estudio consiste en analizar cómo construyen la noción de *elemento genérico* al pasar de una cantidad *específica* a una cantidad *genérica* de *N*

elementos al escribir el programa (dado por la constante *N* que define el tamaño del arreglo). Esto implica pasar de un caso particular a un algoritmo más general para el problema propuesto.

Lo anterior constituye una instancia del problema del *elemento genérico*. Este problema ha sido estudiado en profundidad por Matalon en un capítulo de [9] titulado *Recherches sur le nombre quelconque* (*Investigaciones sobre el 'número cualquiera'*, t.d.a). Matalon analiza el problema de realizar el salto de casos particulares al caso general e introduce variables para su referencia. Por ejemplo, explica que, previo a que se adoptara el uso de variables en matemáticas, Fermat hacía sus demostraciones aritméticas usando un número particular, pero tratándolo como un número genérico, por ejemplo, el 17. En la medida de que ninguna propiedad específica del 17 se involucra en la prueba, entonces la misma se considera válida para todos los números. Matalon agrega además que, en geometría, al probar una propiedad para un triángulo genérico, se dibuja un triángulo concreto, evitando que sea rectángulo, equilátero o isósceles y sin involucrar en la prueba ninguna propiedad específica del triángulo dibujado. Matalon concluye que para construir la noción de *elemento genérico*, es necesario realizar una *acción genérica* que, por repetición sucesiva, permite construir la noción de elemento genérico. Esto resulta fundamental en el pasaje a la etapa *trans* y por ello se analiza como parte del estudio empírico.

Por otra parte, cuando los participantes logran construir un programa que resuelve la *búsqueda lineal*, se les solicita, como parte del estudio, escribir un segundo programa que resuelva un nuevo problema (*filtrado* de valores). Se trata de otro problema que presenta similitudes con el anterior (trabaja sobre la misma estructura de datos: *arreglo*) y diferencias (requiere emplear un algoritmo diferente para su resolución). Se busca analizar cómo adaptan el conocimiento construido en el problema previo para dar solución al nuevo problema, lo cual también contribuye a la construcción de un esquema más general en la etapa *trans*. La herramienta cognitiva que interviene en dicha construcción es la *generalización* (en sus dos formas: *inductiva* y *constructiva*), presentada en la sección 2.3.2.

Capítulo 3

Un estudio empírico basado en el modelo: Conocimiento formal sobre búsqueda lineal

En este capítulo se presenta un estudio empírico en el que se investiga en detalle el proceso de construcción de conocimiento sobre un problema algorítmico específico, su solución algorítmica y la implementación y ejecución de un *programa* que lo resuelve en computadora. El problema elegido es la *búsqueda de un elemento en una secuencia no ordenada de elementos* (también conocido como *búsqueda lineal*). El estudio fue realizado con trece estudiantes que cursaron el primer año de la carrera Ingeniería en Computación en el año 2019 y se divide en dos partes. La primera parte analiza el proceso de construcción desde la etapa de conocimiento *instrumental* (*intra*) hasta la de conocimiento *conceptual* (*inter*). La segunda parte describe el objetivo principal de este trabajo, que consiste en investigar el proceso de construcción de conocimiento *formal* (etapa *trans*) a partir del conocimiento *conceptual* construido.

En la primera parte se analiza el proceso que transcurre desde que el estudiante se enfrenta a una instancia concreta del problema (y lo resuelve con éxito en el plano de la acción) hasta que expresa en lenguaje natural cómo lo resolvió y por qué funciona. La correcta descripción del algoritmo y de las razones de éxito en la solución da cuenta del pasaje de la etapa *intra* a la etapa *inter*. Para esta parte, se usan los mismos resultados y recomendaciones de trabajos previos [6, 31, 32] que abordan esa etapa del proceso en profundidad. La segunda parte analiza el proceso de formalización de los conceptos construidos en la primera parte, por medio de la escritura de un programa y su compilación y ejecución en una computadora, lo cual evidencia el pasaje de la etapa *inter* a la etapa *trans*. Se utiliza por primera vez el modelo propuesto en el capítulo 2 para explicar el proceso de construcción de conocimiento *formal* sobre programas, regulado por la extensión a la ley general de la cognición (presentada en la sección 2.5.3 de dicho capítulo), la cual se ilustra mediante el siguiente diagrama:

$$\overbrace{C \leftarrow P \rightarrow C'}$$
$$newC \leftarrow newP \rightarrow newC'$$

La extensión a la ley explica el proceso por el cual el pensamiento del estudiante pasa del conocimiento *conceptual* construido en el pasaje de *intra* a *inter* (representado por *newP*) al conocimiento *formal* que construye en el pasaje de *inter* a *trans* (representado por *newC* y *newC'*). Es decir, cómo el estudiante construye conocimiento sobre la correspondencia entre la aplicación de una instancia concreta del algoritmo (sobre objetos *físicos* manipulados por sí mismo) y la instrucción a la computadora para que ejecute el algoritmo en forma general, sobre objetos *computacionales*. Dicha correspondencia se representa por la relación entre las dos líneas del diagrama. Los resultados de este estudio permiten explicar el proceso *completo* de construcción de conocimiento (de *intra* a *trans*) para el problema elegido y abren la puerta hacia nuevas investigaciones a realizar como trabajos futuros.

Para el estudio se opta por la *búsqueda lineal* dado que los problemas de búsqueda en general (y el de la *búsqueda lineal* en particular) se estudian en cursos iniciales de programación y se consideran fundamentales en la formación de profesionales en informática. El autor de este trabajo ha participado durante más de 15 años en el dictado de cursos de programación introductorios. Durante ese tiempo ha constatado que la *búsqueda lineal* es un problema tal que su conceptualización y formalización presentan varias dificultades para estudiantes iniciales y requiere la combinación de diversos conceptos de programación para su resolución. Sin embargo, es un problema bien conocido a nivel instrumental, pues es aplicado con éxito y de manera automática en múltiples situaciones cotidianas por parte de los estudiantes. Esto brinda la posibilidad de estudiar en detalle la conceptualización y formalización desde la fuente de construcción de conocimiento dada por las acciones. Por ello, se entiende que el problema, aún en su naturaleza simple, presenta suficiente riqueza para extraer conclusiones interesantes del proceso de construcción de conocimiento y su explicación mediante el modelo.

En la sección 3.1 se presentan los criterios para la selección de los estudiantes participantes junto con la metodología empleada para el estudio. En las secciones 3.2 y 3.5 se proponen los diseños en detalle para ambas partes del estudio. En las secciones 3.3 y 3.6, se describe la realización de cada parte con los estudiantes y se analizan los resultados correspondientes. Finalmente, en las secciones 3.4 y 3.7 se presentan las conclusiones correspondientes a cada parte del estudio.

3.1 Selección de estudiantes y metodología empleada

Para realizar el estudio se seleccionaron trece estudiantes que cursaron el primer año de la carrera Ingeniería en Computación en el año 2019. Se definieron cinco criterios para la selección, orientados a reducir lo más posible cualquier sesgo en la información obtenida. Los criterios definidos son los siguientes. Al momento de realizar el estudio:

1. Todos estaban realizando el primer curso de programación de la carrera.
2. Ninguno había estudiado temas de programación antes de comenzar el curso.
3. Ninguno había estudiado formalmente el problema de la búsqueda lineal.

4. Todos habían trabajado en clase con los siguientes temas: resolución de problemas de programación sencillos, sintaxis básica de un lenguaje de programación imperativa, variables, tipos de datos elementales, expresiones e instrucciones simples del lenguaje, estructuras de control (tanto de *selección* como de *iteración*).
5. Ninguno había trabajado con *arreglos* hasta el momento (el estudio se planteó justo antes de empezar a trabajar en clase con dicha estructura de datos).

Los trece estudiantes participaron del estudio en forma voluntaria. Siete de ellos pertenecían a un grupo que usaba el lenguaje de programación *C*, mientras que los restantes seis pertenecían a otro grupo que usaba el lenguaje *Pascal*. Los contenidos temáticos vistos hasta el momento eran los mismos en ambos grupos, excepto por el lenguaje de programación utilizado. La principal razón por la cual se eligió realizar el estudio con estudiantes que trabajaban con ambos lenguajes fue de índole práctica ya que coincidió que el autor del presente trabajo era docente en ambos grupos a la vez, por lo que también se decidió analizar, como parte del estudio, si la diferencia en el lenguaje utilizado aporta o no algún elemento relevante para la formalización.

El estudio es de naturaleza *cualitativa* y la metodología consiste en llevar a cabo una entrevista individual con cada estudiante, como las realizadas en los estudios previos que siguen el estilo de la entrevista clínica, elegido por Piaget. Durante la entrevista, se le pide al estudiante que realice una serie de actividades y se le hacen preguntas orientadas a inducirlo a reflexionar sobre lo realizado y sus resultados. Se plantea la realización de una actividad para la primera parte y otras cuatro actividades para la segunda parte, las cuales se describen en este capítulo.

Para el diseño de las actividades, se elaboró un borrador inicial y se realizó una experiencia *piloto* con otro estudiante que no formaba parte de los trece que participaron finalmente en el estudio. La inclusión de la experiencia piloto se tomó de otros estudios previos y su propósito es validar las actividades propuestas en relación al resultado esperado del estudio. Ello permitió realizar una serie de ajustes a las actividades, tanto en cuestiones relativas a la formulación de las mismas, como al orden de su realización. También posibilitó la reformulación de algunas de modo que se pusiera el foco en aspectos específicos que se deseaba investigar, dejando de lado otros aspectos potencialmente distractores. Por ejemplo, durante la parte de escritura del programa en la experiencia piloto, se constató que brindarle al estudiante un programa con el arreglo (sobre el que se ejecutaría la búsqueda) previamente cargado con valores permitió poner el foco en el problema de búsqueda en sí, sin desviar la atención del estudiante hacia otro problema (carga previa de los valores del arreglo).

3.2 Diseño del estudio: primera parte (*intra* → *inter*)

La primera parte del estudio fue diseñada siguiendo los resultados de la experiencia piloto junto con los resultados y recomendaciones de trabajos previos [6, 31, 32] que estudian, para otros problemas algorítmicos, el proceso de transformación de conocimiento *instrumental* (etapa *intra*) en conocimiento *conceptual* (etapa *inter*). La actividad diseñada para esta parte se describe en detalle en la sección 3.2.1 y consiste en solicitar a cada estudiante que resuelva una instancia concreta del problema de la búsqueda lineal (en el plano de la acción) y que, luego de lograrlo, conteste preguntas orientadas a obtener una descripción precisa de cómo lo resuelve y por qué tiene éxito. Este proceso constituye un primer paso hacia la conceptualización, como se explica en la sección 3.2.2, donde se detallan los fundamentos teóricos empleados para su diseño.

3.2.1 Actividad propuesta

Al comienzo de la actividad, se le presenta al estudiante una hilera de tarjetas numeradas sobre una mesa y se le dice que simulan ser números de puertas de casas en una calle. Debajo de cada tarjeta hay otra tarjeta que contiene otro número, el cual representa el documento de identidad (cédula) de la persona que vive en la casa correspondiente. Asimismo, se explica que se asume que en cada casa vive solo una persona y que no hay dos personas distintas con el mismo número de documento. Los números de puerta están ordenados en forma secuencial y son visibles para el estudiante, mientras que los números de documento no están ordenados y están ocultos a la vista del estudiante (simulando ser las personas dentro de sus casas).

Al diseñar este tipo de actividades, es preciso enfocarse en el resultado *final* esperado. En este caso, se espera que el estudiante construya conocimiento sobre el problema en el plano *formal*, evidenciado por la implementación de un *programa* de búsqueda lineal de un valor dado en un *arreglo*. Por eso, se presenta la instancia como la disposición de tarjetas simulando puertas, lo cual se asemeja a un *arreglo* de enteros en un lenguaje de programación. Los números de puerta representan los *índices* (ordenados) del arreglo, en tanto los números de documento representan los *valores* (no ordenados) guardados en sus celdas. El arreglo es una representación *computacional*, en el plano *formal*, de la hilera de puertas manipulada en el plano *instrumental* y conceptualizada mediante la descripción en lenguaje natural de la manipulación realizada (plano *conceptual*). Varios estudios previos [6, 29, 30, 31] han mostrado que establecer una correspondencia entre objetos *físicos* y objetos *computacionales* resulta de utilidad en el diseño de actividades de este tipo, facilitando luego al sujeto la comprensión de lo que significa una *estructura de datos* en el plano formal.

Se pide al estudiante que busque un determinado número de documento en la hilera de tarjetas, de modo análogo a como lo haría estando físicamente posicionado al comienzo de la calle, frente a la primera puerta. Se le pide realizar este proceso dos veces; una para buscar un documento que se encuentra en la hilera y otra para buscar otro que no (en ninguna se le dice de antemano si efectivamente está o no). En este

punto, la metodología establece que la participación del entrevistador sea mínima. Por su naturaleza, se espera que cada estudiante resuelva el problema por sí mismo de forma exitosa en la acción, tanto en caso de que el documento buscado se encuentre en alguna puerta (finalizando la búsqueda en ese momento) como en caso de que no (finalizando tras haber visitado todas las puertas). No se le menciona el objetivo del estudio ni el nombre del problema, simplemente se pide que lo resuelva. Tampoco se le dice que las tarjetas constituyen una representación informal de la estructura de datos (*arreglo*) a manipular luego en la etapa formal (escritura del programa).

Una vez resuelto el problema, se le pide que describa oralmente cómo lo hizo y por qué tuvo éxito, tanto cuando el documento buscado se encuentra en la hilera, como cuando no (en este último caso el éxito significa el fin de la búsqueda). Esto estimula al estudiante a separarse de las instancias concretas del problema resueltas en la acción y tratar de brindar una descripción *general* del método aplicado y por qué funciona. Lo primero lo induce a tomar conciencia de la coordinación de las *acciones* realizadas y lo segundo a tomar conciencia de los *cambios* que dichas acciones imponen a los objetos manipulados. El avance hacia la construcción de conceptos está dado por el grado de toma de conciencia sobre ambos (acciones y cambios).

En el marco del problema, las acciones fundamentales son tres: la comparación del documento buscado con el de la persona que vive tras cada puerta, el avance hacia la siguiente puerta de la hilera y la repetición de estas dos en cada puerta visitada. La repetición de acciones genera una reducción progresiva en la cantidad de puertas que restan visitar, lo que conduce eventualmente a que el estudiante observe un cambio en los objetos debido a las acciones. Los posibles cambios observables son dos: un cambio en la relación resultante de la comparación de documentos (que pasa de *distintos* a *iguales*) o un cambio en la cantidad de puertas que restan visitar tras el avance (que pasa de ser *mayor* que cero a ser *igual* a cero). En términos del algoritmo, estos cambios se interpretan como sus dos condiciones de parada posibles: detener la búsqueda cuando se llega a una puerta que contiene un documento igual al buscado o detenerla cuando no quedan más puertas por visitar. En el primer caso, la búsqueda finaliza habiendo encontrado el documento, en tanto que, en el segundo, finaliza sin haberlo encontrado (el documento no está en la hilera).

Si bien es esperable que todos los estudiantes logren el éxito en la acción, el grado de corrección y completitud en la descripción puede variar de un estudiante a otro. Por lo tanto, a cada estudiante se le realizan preguntas orientadas a ayudarlo a que logre describir oralmente y con precisión las acciones realizadas y los cambios que las mismas imponen a los objetos, explicando además por qué tiene éxito, como un primer paso hacia la construcción de conceptos. Puede ser necesario volver a realizar la tarea en la acción para que el estudiante tome conciencia de lo que *dice* en relación a lo que *hizo*, donde se plantean nuevas preguntas según las respuestas del estudiante. Cuando finalmente la descripción en lenguaje natural coincide con su accionar, se le pide que la ponga *por escrito*. La descripción escrita constituye una primera expresión en

lenguaje natural del algoritmo de búsqueda lineal y será usada como punto de partida para la segunda parte del estudio (pasaje de *inter* a *trans*).

3.2.2 Fundamentación teórica

El diseño de la actividad propuesta se fundamenta por la *ley general de la cognición* [14] de la Teoría de Piaget (presentada en el capítulo 2, sección 2.2). En esta parte, se usa dicha ley para explicar el proceso de construcción de conocimiento *conceptual, a* partir del *instrumental.*

$$C \leftarrow P \rightarrow C'$$

Al enfrentarse al problema, el pensamiento del estudiante se encuentra inicialmente en la periferia (P), lo cual significa que está enfocado en lograr el resultado deseado y solucionarlo. En el proceso utiliza conocimiento instrumental (etapa *intra*), sin que las acciones realizadas ni los cambios que imponen sobre los objetos sean conscientes. Dicho conocimiento instrumental fue construido en forma previa, a partir de la interacción del sujeto con el medio, tras realizar tareas de búsqueda similares en su vida cotidiana. En el marco de la ley, la transformación en conocimiento *conceptual* (etapa *inter*) surge de un proceso por el cual el pensamiento transita desde P hacia los centros (C y C'). El centro C corresponde a la conceptualización de la coordinación de las acciones, en tanto C' corresponde a la conceptualización de los cambios que ellas imponen sobre los objetos. El conocimiento se construye cuando el estudiante toma conciencia de los cambios que sus acciones provocan en los objetos, lo cual se evidencia cuando explica cuáles son esas acciones y las razones de éxito. Observar que las transiciones desde P hacia C y C' son dialécticas, en el sentido de que el avance hacia cada centro influye en el otro. La herramienta cognitiva que interviene en ambas transiciones es la *abstracción* [14, 15, 17] en sus dos formas: *empírica y reflexiva* (presentadas en el capítulo 2, sección 2.3.1). La primera explica la reacción inmediata del sujeto en atribuir el éxito puramente a sus acciones o a propiedades de los objetos, sin tener conciencia del vínculo entre ellos. La segunda permite reconstruir, en el plano del pensamiento, las acciones efectuadas en el plano de la acción junto con los cambios observados en los objetos manipulados.

El pensamiento del estudiante se evidencia en la periferia (P) cuando, tras resolver el problema en la acción, su descripción oral inicial está centrada en el resultado logrado. Se ha constatado en estudios previos [6, 31, 32] que, para el caso de algoritmos básicos, la conceptualización de la relación entre acciones y cambios presenta la dificultad de que, una vez resuelto el problema, y frente al requerimiento de explicar *cómo* lo hizo, el estudiante responde *qué* hizo. Por ejemplo, para este problema concreto, si dice "*busqué el documento entre las puertas*", su pensamiento está centrado en el resultado alcanzado. En este caso, se hace énfasis en pedirle que describa, *paso a paso*, *cómo* hizo para resolverlo. Esto lo induce a que reflexione sobre la coordinación de las acciones realizadas (comparación, avance, repetición) posibilitando así que dé una descripción más completa y su pensamiento comience a transitar hacia el centro C.

También se le pregunta cuáles son las razones de su éxito al alcanzar la solución. Aplicado a este problema, significa en qué caso(s) tiene éxito al completar la búsqueda. Esto lo induce a reflexionar sobre los cambios que sus acciones generan en los objetos (detiene la búsqueda cuando encuentra un documento *igual* al buscado o cuando no queda *ninguna* puerta por visitar) para que su pensamiento comience a transitar hacia C'. Dependiendo de lo que conteste, se le hacen preguntas para ayudarlo en la reflexión. Es un proceso dinámico, que requiere prestar atención a lo que dice para decidir qué preguntarle a continuación. Además, puede implicar pedirle que vuelva a la acción para que realice nuevamente la tarea y dé una nueva descripción.

Se ha observado también que el planteo de un *método alternativo* en la vuelta a la acción juega un rol determinante en ayudar a la conceptualización en casos en los que el estudiante atribuye el éxito puramente a sus acciones o puramente a propiedades de los objetos. La introducción de otro método que, manteniendo las mismas acciones o propiedades, *no* conduce a un resultado exitoso es de gran ayuda, pues induce al estudiante a mover el foco. Si está centrado en sus acciones, lo estimula a pensar en los cambios que ellas generan. Si está centrado en propiedades, lo estimula a pensar en las acciones que necesita realizar para que se observe algún cambio. Se pueden proponer diversos métodos alternativos, según dónde ponga el foco. A continuación, se brindan dos ejemplos de posibles respuestas que podría dar un estudiante para este problema junto con posibles métodos alternativos (no necesariamente únicos).

Ejemplo 1: Si dice "*encontré el documento porque hago lo mismo en cada puerta*", significa que su pensamiento está centrado en sus acciones y no toma en cuenta los cambios que ellas imponen a los objetos. En este caso, se le pide que vuelva a la acción y se le van agregando tarjetas adicionales a la hilera, conforme va avanzando, sin que ninguna de ellas tenga el número buscado. El objetivo es enfrentarlo a una situación en la cual *hace lo mismo en cada puerta* pero que, sin embargo, no conduce a un resultado exitoso. El propósito en este caso es inducirlo a que tome conciencia de que la disminución en la cantidad de puertas, a medida que avanza, es una condición esencial para alcanzar la terminación y que el éxito no depende solo de sus acciones, sino también del efecto que las mismas tienen sobre la cantidad de puertas restantes.

Ejemplo 2: Si dice "*encuentro el documento porque las puertas están ordenadas*", significa que su pensamiento está centrado en una propiedad de la hilera y no toma en cuenta sus acciones. En este caso, se le propone otro método alternativo en el cual se le pide que siempre vuelva a golpear en la primera puerta. El objetivo es enfrentarlo a una situación en la cual *las puertas están ordenadas* pero que tampoco conduce a un resultado exitoso. Ahora el propósito es inducirlo a que tome conciencia de que la acción de avance es fundamental para alcanzar la terminación y que el éxito no depende solo de la propiedad de orden de la hilera, sino de cómo sus acciones generan cambios sobre la misma, a medida que la recorre siguiendo ese orden.

En resumen, si bien tanto las preguntas realizadas como el orden en que se formulan pueden variar de un estudiante a otro, en todos los casos tienen como objetivo que el

estudiante tome conciencia de la coordinación de las acciones realizadas, los cambios que pueden observarse en los objetos a raíz de ellas, y que logre expresar ambos con claridad en su descripción. La toma de conciencia sobre la relación entre acciones y cambios se evidencia cuando el estudiante es capaz de explicar las razones por las que tiene éxito en resolver el problema, tanto cuando encuentra el documento buscado como cuando detiene la búsqueda porque ya no quedan más puertas. En dicha toma de conciencia radica la conceptualización. Es decir, el conocimiento *instrumental* (etapa *intra*) se transforma en *conceptual* (etapa *inter*).

3.3 Desarrollo del estudio: primera parte (*intra* → *inter*)

En esta sección se presentan extractos de las descripciones en lenguaje natural escritas por los estudiantes al finalizar la primera parte del estudio (pasaje de la etapa *intra* a la etapa *inter*) junto con un análisis preliminar del grado de conceptualización del algoritmo que las mismas evidencian. Cuanto mayor es el grado de conceptualización en relación a la coordinación de acciones, mayor es el avance desde P hacia C, y cuanto mayor es en relación a los cambios observados en los objetos, mayor es el avance desde P hacia C'. Las descripciones completas de todos los estudiantes y sus análisis detallados (estudiante por estudiante) se encuentran disponibles en el anexo A.

En la sección 3.3.1 se incluyen algunos extractos de las descripciones junto con un análisis preliminar del grado de conceptualización evidenciado por ellas. En la sección 3.3.2 se realiza un análisis global de los resultados, incluyendo a todos los estudiantes. Luego, en la sección 3.4, se presentan las conclusiones de la primera parte del estudio.

3.3.1 Descripciones de los estudiantes y análisis preliminar

Se presentan algunos extractos de descripciones dadas por los estudiantes, agrupadas en dos categorías; acciones realizadas y cambios en objetos. En cada categoría se analiza el extracto según el grado de conceptualización que evidencia. Cada estudiante se identifica por su nombre de pila (y también por la inicial de su apellido, en caso de repetirse el nombre).

Acciones realizadas

El mayor grado de conceptualización de las acciones se da cuando en la descripción del *cómo* se logra resolver el problema, las mismas aparecen claramente expresadas y concuerdan con lo hecho previamente en la acción. Hay estudiantes que las describen de forma totalmente explícita. Por ejemplo, Nicolás lo hace, como muestra el siguiente extracto de su descripción:

*Se recorren **las puertas** en orden, abriéndolas y **preguntando a la persona detrás de ellas su número de cédula**. **Si este es el que estábamos buscando,** se finaliza la búsqueda, Sino, **pasamos a la siguiente puerta**.*

Expresa con claridad tanto la comparación (*preguntando a la persona detrás de ellas su número de cédula. Si este es el que estábamos buscando*) como el avance secuencial

(*pasamos a la siguiente puerta*). También deja en claro que repite las acciones en cada una (describe todas las acciones refiriéndose siempre a *las puertas*, en plural).

Otros estudiantes exhiben un grado intermedio de conceptualización al no expresar todas las acciones tan explícitamente, ya sea porque lo hacen para algunas puertas en particular, o bien porque las dejan implícitas en algunos casos. Por ejemplo, Joaquín, en el siguiente extracto:

***Evalúo la primera puerta verificando si ésta contiene el número buscado**, si coincide finalizo la búsqueda. **De lo contrario golpeo la siguiente puerta**, así hasta que o bien se encuentre el número buscado o se acaben las puertas.*

Aclara expresamente que pregunta por el documento en la primera puerta y que lo compara con el que busca (*Evalúo la primera puerta verificando si ésta contiene el número buscado*) pero no lo hace explícito para las demás (lo deja implícito en *así hasta que*). Lo mismo con el avance (*De lo contrario golpeo la siguiente puerta*). Tampoco dice claramente que recorre el resto de la hilera de manera secuencial ni que repite el proceso en las demás puertas (solo dice *así hasta que*).

El menor grado de conceptualización se da cuando ninguna de las acciones se explicita en la descripción, o bien cuando apenas se sugieren. Por ejemplo, Aaron, en el siguiente extracto:

Para encontrar la cédula deseada busqué o recorrí puerta por puerta siguiendo un orden hasta encontrarla. De no encontrarla, paré luego de verificar todas las puertas.

Nunca dice (ni sugiere) que compara el documento buscado con el que se encuentra en cada puerta visitada. Tampoco expresa con claridad el avance hacia la siguiente puerta, solo lo sugiere al decir *recorrí puerta por puerta siguiendo un orden*, ni menciona que repite acciones en ellas.

Cambios en objetos

Hay estudiantes que expresan los cambios que sus acciones producen en los objetos de manera totalmente explícita, al detallar claramente las dos posibles condiciones de parada del algoritmo. Estos estudiantes muestran el mayor grado de conceptualización en relación a los cambios. Por ejemplo, Valeria, en el siguiente extracto:

Pueden ocurrir 2 casos:

1) ***que encuentre en alguna puerta el nº de cédula** (por lo que **dejo de preguntar en las restantes**)*
2) ***no encuentre coincidencia de cédula** (por lo que **recorrí todas las puertas**)*

En el primer caso expresa la detención luego de que la comparación entre el documento buscado y el de alguna puerta da que son iguales (*encuentre en alguna puerta el nº de cédula… dejo de preguntar en las restantes*) mientras que en el segundo expresa la detención tras terminarse las puertas sin haberlo encontrado (*no encuentre coincidencia de cédula… recorrí todas las puertas*).

Otros estudiantes muestran un grado intermedio de conceptualización, al explicitar solo una de las condiciones y dejar implícita la otra, o bien dejar ambas implícitas, Algunos incluso las hacen explícitas solamente para puertas concretas, pero no para todas (similar a lo que hacen con las acciones). Por ejemplo, Joaquín, en el siguiente extracto:

*Evalúo la **primera puerta** verificando si ésta contiene el número buscado, **si coincide finalizo la búsqueda**. De lo contrario golpeo la siguiente puerta, así **hasta que o bien se encuentre** el número buscado **o se acaben las puertas**.*

Dice que detiene la búsqueda si encuentra el documento específicamente en la primera puerta (*si coincide finalizo la búsqueda*) pero lo deja implícito en caso de que lo encuentre en alguna de las demás puertas (*hasta que o bien se encuentre*). Lo mismo ocurre con la otra condición de parada (*…o se acaben las puertas*).

Todos los estudiantes (excepto uno) mencionan ambas condiciones de parada de algún modo, ya sea de forma totalmente explícita, explícita para puertas concretas, o bien implícita. Pablo P es el único estudiante que hace explícita una de ellas, sin mencionar la otra en absoluto.

*Primero preguntamos en la primer puerta si la persona tiene el número de cédula que buscamos. Como no fue el caso continuamos con la siguiente puerta y volvemos a preguntar si la persona tiene el número de cédula que buscamos. Como la respuesta es negativa, continuamos a la siguiente. Repetimos el proceso de preguntar **si la persona tiene el número de cédula que buscamos**. **Como la respuesta es afirmativa, detenemos la búsqueda**.*

Pablo P describe una instancia concreta que resolvió en la acción en vez de dar una descripción general del algoritmo. Concretamente, describe un caso puntual en que el documento buscado está en la tercera puerta. Como consecuencia, solamente expresa una condición de parada (encontrar el documento: *Como la respuesta es afirmativa, detenemos la búsqueda*). El hecho de que las dos condiciones de parada se hagan explícitas evidencia que se han conceptualizado los cambios que las acciones provocan en los objetos como razones de éxito. Si una de las condiciones no aparece es porque la conceptualización es incompleta.

3.3.2 Resultados de la primera parte

Los trece estudiantes resuelven correctamente el problema en la acción, iniciando la búsqueda en la primera puerta y recorriendo la hilera en forma secuencial, deteniendo la búsqueda al encontrar el documento buscado o bien cuando se terminan las puertas. Sin embargo, no todos logran dar una descripción en lenguaje natural completamente acorde a su accionar y no todos expresan con claridad las razones por las cuales el algoritmo finaliza. Alcanzan distintos grados de conceptualización del algoritmo, en relación a acciones realizadas y/o cambios en objetos. Para ilustrar dichos grados, se presentan dos tablas y un análisis global luego de cada una.

La tabla para acciones realizadas tiene tres columnas, que corresponden a cada una de las acciones: comparación, avance y repetición. La tabla para cambios en objetos tiene dos columnas, que corresponden a cada una de las condiciones de parada. Cada columna es independiente de las otras y agrupa estudiantes según el grado de conceptualización alcanzado en cada componente. Todos los estudiantes dentro de un mismo grupo exhiben un grado equivalente. Cuanto más arriba está un grupo en una columna, significa que sus estudiantes logran un mayor grado de conceptualización en el componente (acción o cambio) correspondiente a esa columna. Cuanto más abajo, significa menor grado de conceptualización.

<table>
<tr><th colspan="3">ACCIONES REALIZADAS</th></tr>
<tr><th>Comparación</th><th>Avance</th><th>Repetición</th></tr>
<tr><td>Explícita en cada puerta
Ignacio D, Nicolás, Pablo M</td><td>Explícita en cada puerta
Nicolás, Pablo M, Tomás</td><td>Explícita en cada puerta
Ignacio D, Nicolás, Pablo M, Tomás</td></tr>
<tr><td>Explícita en puertas concretas
Ignacio U, Joaquín, Juan, Martín, Pablo P, Ximena</td><td>Explícita en puertas concretas
Ignacio U, Joaquín, Juan Pablo P</td><td>Explícita en puertas concretas
Pablo P</td></tr>
<tr><td>Implícita
Mónica, Tomás, Valeria</td><td>Implícita
Aaron</td><td rowspan="2">No menciona
Aaron, Ignacio U, Joaquín, Juan, Martín, Mónica, Valeria, Ximena</td></tr>
<tr><td>No menciona
Aaron</td><td>No menciona
Ignacio D, Martín, Mónica, Valeria, Ximena</td></tr>
</table>

Tabla 3.3.2.1 – Acciones realizadas

Respecto a la acción de comparación, nueve estudiantes la hacen explícita en mayor o menor medida, otros tres la sugieren de forma implícita y hay un estudiante (Aaron) que no manifiesta conciencia de la misma, al no hacer mención alguna (directa ni indirecta) a la comparación (las cantidades de estudiantes no son relevantes para la naturaleza *cualitativa* del estudio, se incluyen solamente a título descriptivo). En cuanto al avance hacia la siguiente puerta, siete explicitan esta acción en mayor o menor medida, mientras que uno la sugiere de forma implícita y otros cinco estudiantes no la mencionan (ni sugieren). Algo similar ocurre con la repetición de las acciones en cada puerta, donde la relación es de cinco a ocho. Estos ocho no manifiestan conciencia de la importancia que tiene volver a comparar documentos y avanzar. En términos de la ley general de la cognición, el pensamiento de aquellos estudiantes que dejan implícita o no mencionan alguna de las tres acciones se encuentra más en la periferia que para los demás estudiantes. No construyen de manera tan sólida la noción de que la realización de las acciones es fundamental para así alcanzar eventualmente la solución. Su grado de transición hacia el centro C es menor que para los demás estudiantes.

<table>
<tr><th colspan="2">CAMBIOS EN OBJETOS</th></tr>
<tr><th>Encontrar documento buscado</th><th>Se acaban las puertas de la hilera</th></tr>
<tr><td>Totalmente explícito
Martín, Nicolás, Pablo M, Valeria</td><td>Totalmente explícito
Aaron, Martín, Mónica, Pablo M, Tomás, Valeria</td></tr>
<tr><td>Explícito en 1ra. puerta, implícito restantes
Joaquín, Tomás</td><td rowspan="2">Implícito
Ignacio D, Joaquín, Juan, Nicolás, Ximena, Ignacio U (algoritmo general pero atado a cantidad concreta de 7 puertas)</td></tr>
<tr><td>Explícito solo en puerta concreta
Ignacio U, Mónica, Ximena (1ra puerta)
Pablo P (3ra puerta, instancia concreta)</td></tr>
<tr><td>Implícito
Aaron, Ignacio D, Juan</td><td>No menciona
Pablo P (instancia concreta que no toma en cuenta esta condición de parada)</td></tr>
</table>

Tabla 3.3.2.2 – Cambios en objetos

Respecto a la detención al encontrar el documento buscado, diez estudiantes la explicitan en mayor o menor medida, mientras que tres no lo hacen, pero la sugieren de manera implícita. En cuanto a la detención de la búsqueda al terminarse las puertas, seis la hacen explícita, otros seis la sugieren de forma implícita y un estudiante (Pablo P) no evidencia ningún grado de conceptualización relativo a esta condición de parada. El pensamiento de los estudiantes que dejan implícita o no mencionan alguna condición de parada está más en la periferia que para los demás estudiantes. No construyen de manera tan sólida la noción de que los cambios impuestos a los objetos son fundamentales para alcanzar con éxito la solución. Su grado de transición hacia el centro C' es menor que para los demás estudiantes.

Pablo P e Ignacio U constituyen ejemplos de estudiantes que no logran dar una descripción *general* del algoritmo. Pablo P describe una instancia *concreta* que resuelve en la acción (en la cual encuentra el documento buscado específicamente en la tercera puerta) y por eso no toma en cuenta la otra condición de parada. La descripción de Ignacio U (ver anexo A) muestra que tampoco logra apartarse del todo de la instancia concreta. Si bien da una descripción general del método aplicado, lo hace atado a la cantidad puntual de 7 puertas de su instancia concreta, en vez de hacerlo para una cantidad *cualquiera* de puertas. Todos los demás logran describir un algoritmo que resuelve el problema en forma general (cualesquiera sean el documento a buscar y la cantidad de puertas), si bien tienen matices en cuanto al grado de conceptualización alcanzado en relación a los cambios en los objetos.

3.4 Conclusiones de la primera parte (*intra* → *inter*)

Conforme a lo explicado por la ley general de la cognición, cuanto más sólida es la conceptualización del algoritmo, mayor es el grado de transición de la periferia (P) a los centros (C y C'). Es decir, la transformación de conocimiento *instrumental* (etapa *intra*) en *conceptual* (etapa *inter*) es más profunda. Los resultados de la primera parte muestran que el grado de conceptualización en relación a cada aspecto (acciones y cambios) varía según cada estudiante. Al tomar en cuenta ambos aspectos combinados, se tiene evidencia del grado de conceptualización global del algoritmo.

Hay dos estudiantes (Nicolás y Pablo M) que muestran una sólida construcción, pues presentan un alto grado de conceptualización en todos o bien en la mayoría de sus componentes: comparación, avance y repetición (acciones) y condiciones de parada (cambios). Todos los demás presentan diversidad en los grados de conceptualización (más altos en algunos componentes y más bajos en otros), no habiendo ningún estudiante en los niveles inferiores en los cinco componentes en simultáneo. Aaron es el único que alcanza una conceptualización algo más pobre que el resto en tres de los cinco, al mostrar el grado más bajo en ellos (los demás exhiben el grado más bajo en, como mucho, dos componentes). No obstante, el propio Aaron exhibe el grado más alto en una de las condiciones de parada (terminación cuando se acaban las puertas de la hilera), incluso más alto que otros estudiantes que no muestran el mayor grado en ningún componente.

En conclusión, se observa bastante heterogeneidad en los grados de conceptualización alcanzados por los estudiantes en relación al algoritmo. La transición desde P hacia C es más profunda en algunos estudiantes y menos en otros, así como la transición desde P hacia C'. Una sólida conceptualización en relación a la coordinación de las acciones no necesariamente implica una conceptualización equivalente en relación a los cambios en los objetos, y viceversa, lo cual también se ha observado en estudios previos [6, 31, 32]. La teoría da cuenta (especialmente en dos de las obras de Piaget [14, 17]) de la complejidad en el proceso de construcción de conocimiento y de cuán relevante es el rol del conocimiento instrumental de los sujetos (que se construye al realizar tareas repetidamente en la acción) sin que dicho conocimiento necesariamente se transforme en conceptual. En dichas obras se explican obstáculos que se presentan en la conceptualización y cómo se superan por medio de la repetición (de las tareas) y la reflexión. Por ello, la primera actividad de la segunda parte del estudio plantea a los estudiantes escribir una versión del algoritmo en *pseudocódigo* a partir de su descripción en lenguaje natural, tal y como se describe en la sección 3.5.1. Dicha actividad posibilitará nuevas repeticiones de la tarea, lo cual permitirá a cada estudiante consolidar la conceptualización de aquellos componentes del algoritmo en los cuales la transición hacia el centro correspondiente no fue tan marcada, logrando así consolidar la construcción de conocimiento *conceptual* en relación al algoritmo. A su vez, dicha actividad dará inicio a la construcción de conocimiento *formal* sobre el mismo (inicio del pasaje a la etapa *trans*).

3.5 Diseño del estudio: segunda parte (*inter* → *trans*)

La segunda parte del estudio constituye el objetivo principal de este trabajo. Esto es, investigar la construcción de conocimiento *formal* (etapa *trans*) sobre el problema de búsqueda lineal y el algoritmo empleado en su solución, en base al conocimiento *conceptual* (etapa *inter*) construido a partir del *instrumental* (etapa *intra*). Se busca obtener información acerca del proceso de construcción de conocimiento sobre el *programa* y su ejecución y explicarlo en términos de la extensión a la ley general de la cognición, presentada en el capítulo 2 (sección 2.5.3).

Según lo expuesto en dicho capítulo (sección 2.5.2), el pasaje a la etapa *trans* implica un proceso de construcción de conocimiento que abarca dos aspectos. El primero es que, por tratarse de conocimiento *formal*, involucra el uso de un *formalismo* o *lenguaje formal* (en el marco del presente modelo, dicho formalismo está dado por un *lenguaje de programación*). El segundo es que la ejecución del programa resultante no es realizada por el propio individuo, sino por un *agente externo*: la computadora. Así como la ley general de la cognición explica el proceso de construcción cuando es el propio sujeto quien realiza las acciones y construye conocimiento conceptual sobre el algoritmo, la extensión a la mencionada ley lo explica cuando las acciones están dadas por *instrucciones* del programa ejecutadas por el agente externo.

Para esta parte, se diseñaron tres actividades que guían al estudiante en el pasaje de la descripción en lenguaje natural (dada al finalizar la primera parte) a la escritura, compilación y ejecución de un *programa*, escrito en un *lenguaje de programación*, que resuelve el problema en una *computadora*. Debe lograr escribir y compilar el programa así como comprender y explicar cómo y por qué funciona al ejecutarse en máquina. Luego se diseñó una cuarta actividad que estudia la construcción de un esquema más general en la etapa *trans* y sienta algunas bases para trabajos futuros. En el diseño de las actividades se tuvo en cuenta tres aspectos. Primero, recomendaciones de trabajos previos, en particular el estudio empírico en el cual la ley extendida fue formulada por primera vez [33] y otros dos donde se aborda la escritura de algoritmos usando un *formalismo intermedio* y el agente externo está dado por una persona que hace de *robot imaginario* que ejecuta sus pasos [29, 30]. Segundo, distintos elementos del *lenguaje formal* con los que el estudiante trabajó en clase (previo al estudio). Tercero, el resultado esperado (evidencia del conocimiento *formal* construido, dada por la escritura, compilación y ejecución del programa y explicación de razones de éxito).

En la sección 3.5.1 se describe el diseño de la primera actividad, que guía al estudiante en la escritura de una versión en *pseudocódigo* para el algoritmo. En las secciones 3.5.2 y 3.5.3 se describen, respectivamente, los diseños de la segunda y tercera actividad, que lo guían en la escritura del *programa*, a partir de la versión en pseudocódigo. En la sección 3.5.4 se describe el diseño de la cuarta actividad, la cual sienta las bases para una línea de trabajos futuros a partir del conocimiento formal construido. Finalmente, en la sección 3.5.5 se detallan los fundamentos del marco teórico empleados para el diseño de las cuatro actividades.

3.5.1 Primera actividad: *pseudocódigo* para búsqueda lineal

En esta actividad se le pide al estudiante que, partiendo de su descripción en lenguaje natural, escriba una primera versión del algoritmo en *pseudocódigo*. Se trata de un *formalismo intermedio* utilizado para consolidar la conceptualización del algoritmo y facilitar el inicio del pasaje a la etapa *trans*. Posee reglas de sintaxis y semántica menos rigurosas que un lenguaje formal, siendo más sencillas de comprender para el estudiante y permitiéndole dar una primera expresión del algoritmo pensando en instruir a un *agente externo* (distinto del propio estudiante) para que lo ejecute en forma automática. En este caso, el agente externo está dado por un robot imaginario (interpretado por el entrevistador) en vez de por una computadora, lo que permite al estudiante visualizar la lógica del algoritmo, dejando para después la comprensión de aspectos propios de la ejecución en máquina. En el marco del presente trabajo, se llama *lógica* del algoritmo al orden de ejecución de los pasos expresados en pseudocódigo, pero sin considerar aún aspectos de su posterior implementación en el lenguaje de programación como ser, por ejemplo, la *estructura de datos* a utilizar luego en el programa. En el estudio, se elige implementar la búsqueda lineal sobre un *arreglo*, pero también se podría hacer sobre una *lista encadenada*. En ambas implementaciones, varía la estructura de datos, pero la lógica del algoritmo sigue siendo la misma. Lo que determina la lógica del algoritmo es el comportamiento producido cuando el agente externo ejecuta los pasos del pseudocódigo en el orden definido.

La introducción de un formalismo distinto del lenguaje natural exige tener en cuenta que también es un objeto sobre el cual el estudiante construye conocimiento, además del conocimiento que construye sobre el problema algorítmico propuesto. Previo al estudio, algunos estudiantes ya habían trabajado en clase con pseudocódigo. De todas formas, se repasa con cada estudiante algunas reglas de sintaxis y semántica usadas en este formalismo intermedio, concretamente, reglas para trabajar con *expresiones booleanas* y estructuras de *selección* e *iteración*. Para ello, se prepararon unas tarjetas de cartulina que ilustran la sintaxis genérica de cada una, las cuales se le presentan al estudiante durante el transcurso de la actividad.

Respecto a las expresiones booleanas, las tarjetas ilustran la sintaxis genérica de expresiones que hacen uso de los operadores *and*, *or* y *not*:

condición1 ***and*** *condición2*
condición1 ***or*** *condición2*
not *condición*

Con las tarjetas a la vista, se le pide al estudiante que describa oralmente el comportamiento de cada operador, para así detectar su grado de conceptualización sobre ellos. Si su descripción evidencia una conceptualización pobre, se le pide que escriba pequeños ejemplos concretos de expresiones (por ejemplo: *llueve and hace frío*) y razone el resultado de su evaluación, hasta que logre comprender y explicar la semántica de cada operador. En línea con el marco teórico adoptado, el conocimiento que el estudiante construye sobre dichos operadores surge de su interacción con

instancias concretas que hacen uso de ellos. Por tratarse de *pseudocódigo*, y tomando en cuenta la continuidad del proceso de construcción de conocimiento, en esta etapa se repasa la semántica de cada operador, pero no se menciona aún cómo el lenguaje de programación hace la evaluación. Para *and* y *or*, puede hacerla por *circuito corto* o por *circuito completo*. En el primer caso, evalúa *condición1* y, dependiendo del resultado, evalúa *condición2* solo si es necesario. En el segundo caso, evalúa ambas sin importar el resultado de la primera. Cómo el lenguaje hace la evaluación es tratado en la tercera actividad (escritura del *programa*).

En cuanto a las estructuras de *selección* e *iteración*, también se presentan tarjetas que ilustran la sintaxis genérica de cada una. Se introducen cuatro estructuras, tres de las cuales contienen condiciones que hacen uso de *expresiones booleanas* como las vistas anteriormente. Dos de ellas son estructuras de selección (*si* y *si / sino*) y las otras dos son estructuras de iteración (*mientras* y *para cada*). Si bien algunos de los estudiantes también han trabajado en clase con otras estructuras más, se optó por trabajar solo con estas cuatro para el estudio planteado, pues son suficientes para las actividades propuestas. La sintaxis presentada en las tarjetas es:

Selección con una única secuencia de pasos posible:

si condición
    paso 1
    paso 2
    ...
fin

Selección con dos secuencias de pasos posibles:

si condición
    paso 1a
    paso 2a
    ...
sino
    paso 1b
    paso 2b
    ...
fin

Iteración condicional:

mientras condición
    paso 1
    paso 2
    ...
fin

Iteración por subrango:

para cada elemento
    paso 1
    paso 2
    …
fin

Así como con las expresiones booleanas, se pide al estudiante que describa oralmente el comportamiento de cada una. Se busca detectar su grado de conceptualización en relación a las estructuras. De ser necesario, se proponen nuevamente ejemplos concretos hasta que reconozca las similitudes y diferencias entre ellas y comprenda cuándo corresponde usar cada una, según las características del problema. Por tratarse de *pseudocódigo*, en esta etapa no se menciona cómo el lenguaje de programación implementa cada una. Por ejemplo, la implementación de *para cada* en *C* exige al programador variar explícitamente la variable que controla la cantidad de veces que se realiza la iteración. *Pascal*, en cambio, hace dicha variación en forma implícita. Tampoco se abordan aún otras cuestiones propias de la ejecución en máquina, como ser un error de *salida de rango* o acceder a una celda con valor indefinido (*basura*). Esos temas se tratan en la tercera actividad (escritura del *programa*).

La experiencia del autor de este trabajo como docente del curso también fue una guía en el diseño del estudio. Durante años, ha constatado que un error que los estudiantes cometen con frecuencia es la elección de *para cada* en situaciones en las que es más adecuado usar *mientras*, aun habiendo visto en clase cuándo es más adecuado utilizar cada estructura. El curso sigue un enfoque de programación *estructurada*, según el cual corresponde utilizar *para cada* cuando se conoce de antemano la cantidad de veces a iterar y *mientras* cuando *no* se conoce. Al usar *para cada* no es admisible cortar la iteración antes de completar el total de repeticiones. En el caso de *mientras*, la iteración debe terminar únicamente cuando la condición *deja* de cumplirse, no siendo admisible cortarla por otros medios. Para el problema planteado, el uso de *para cada* implica recorrer todas las puertas, en cambio *mientras* permite detener la búsqueda cuando corresponda. Es esperable que haya estudiantes que elijan *para cada*, aun habiendo expresado correctamente la detención en lenguaje natural. Por ejemplo, un estudiante que expresó *pregunto puerta a puerta y paro si encuentro la cédula* en su descripción, pero en pseudocódigo escribe *para cada puerta*, no usa la más adecuada. En tal caso, se lo induce a corregir su pseudocódigo según su propia versión en lenguaje natural.

Durante la actividad se le pide al estudiante que escriba *todas* las versiones en pseudocódigo necesarias para el algoritmo, como forma de consolidar gradualmente los conceptos y corregir errores entre versión y versión, hasta llegar finalmente a una correcta. El objetivo es que tome conciencia de que las acciones (comparación, avance y repetición) ahora son ejecutadas por el *agente externo*, que es el que impone cambios a los objetos (pasar de documentos *distintos* a *iguales* y pasar de que *queden puertas* en la hilera a que *no quede ninguna*) para alcanzar con éxito la solución.

Entre cada versión y la siguiente, se recurre a la *automatización*. Este mecanismo, introducido en [29, 30] y mencionado en el capítulo 2 (sección 2.5.2) consiste en que el entrevistador oficia de *agente externo*, actuando como un robot imaginario que ejecuta los pasos mientras el estudiante observa su comportamiento. El objetivo es inducir al estudiante a reflexionar sobre los errores detectados y corregirlos. Al igual que en la primera parte, se hace al menos dos veces: una para buscar un documento que está en la hilera y otra para buscar otro que no, intercalando preguntas similares a las de dicha parte. Los errores detectados pueden deberse a cuestiones de sintaxis y/o semántica, a problemas en la lógica del algoritmo (los pasos generan que el robot imaginario realice acciones erróneas o que no conducen a la solución), o incluso a una conceptualización incompleta del algoritmo en la primera parte. Cualquiera sea el caso, la automatización ha probado ser de utilidad para ayudar a su detección y corrección. El proceso se repite hasta que el robot ejecuta el pseudocódigo y logra resolver correctamente el problema en cualquier caso, lo que evidencia un avance en la conceptualización del estudiante.

3.5.2 Segunda actividad: *sintaxis* y *semántica* del lenguaje formal

Previo a la escritura del *programa* en el lenguaje de programación, en esta actividad se le pide al estudiante resolver algunos pequeños ejercicios para que se familiarice con las reglas de sintaxis y semántica para trabajar con *arreglos* y también repasar otros elementos del lenguaje de programación trabajados en clase antes del estudio. Conforme a lo expresado en la sección 3.1, el estudio fue planteado justo antes de comenzar a trabajar con la estructura de datos *arreglo*, por lo que el estudiante aún no la conoce. También se repasan las reglas de sintaxis y semántica del lenguaje de programación para trabajar con *expresiones booleanas* y las estructuras de *selección* e *iteración* vistas en pseudocódigo. Nuevamente se presentan tarjetas, las cuales ilustran la sintaxis genérica de las mismas, pero ahora en el lenguaje de programación usado por el estudiante (*C* a la izquierda, *Pascal* a la derecha).

Expresiones booleanas:

```
expresion1 && expresion2        expresion1 and expresion2
expresion1 || expresion2        expresion1 or expresion2
!expresion                      not expresion
```

Selección con una única secuencia de pasos posible:

```
if (expresion)                  if expresion then
{                               begin
    instruccion1;                   instruccion1;
    instruccion2;                   instruccion2;
    ...                             ...
}                               end
```

Selección con dos secuencias de pasos posibles:

```
if (expresion)                    if expresion then
{                                 begin
    instruccion1a;                    instruccion1a;
    instruccion2a;                    instruccion2a;
    ...                               ...
}                                 end
else                              else
{                                 begin
    instruccion1b;                    instruccion1b;
    instruccion2b;                    instruccion2b;
    ...                               ...
}                                 end
```

Iteración condicional:

```
while (expresion)                 while expresion do
{                                 begin
    instruccion1;                     instruccion1;
    instruccion2;                     instruccion2;
    ...                               ...
}                                 end
```

Iteración por subrango:

```
for (i = ini; i <= fin; i++)      for i := ini to fin do
{                                 begin
    instruccion1;                     instruccion1;
    instruccion2;                     instruccion2;
    ...                               ...
}                                 end
```

Para introducir al estudiante el concepto de *arreglo* se le brinda la siguiente descripción informal: "*un **arreglo** es una variable que permite almacenar varios valores en forma secuencial, todos del mismo tipo. Está dividida en casillas llamadas **celdas**. Cada celda contiene un valor y se accede por un **índice**. Los índices pertenecen a un **subrango** finito de valores consecutivos*". Junto con la descripción, se le muestra un dibujo que ilustra una instancia concreta de dicha estructura de datos, conteniendo valores enteros, y también se presenta la definición sintáctica en el lenguaje usado por el estudiante (*C* o *Pascal*). Además, se indica que la sintaxis `arre[i]` se utiliza para acceder al valor de la i-ésima celda, siendo i un índice válido del rango de índices del arreglo. En el caso de *C* siempre se usan índices empezando en 0, mientras que en *Pascal* se usan índices empezando en 1 (pues así se haría con frecuencia en dicho grupo en el resto del curso, si bien *Pascal* también admite índices empezando en otros valores). Se muestra a continuación el dibujo mostrado a los estudiantes del grupo de *Pascal* (el de *C* es igual, pero con índices empezando en cero) junto con la definición de tipo en ambos lenguajes y la declaración de una variable (*C* a la izquierda, *Pascal* a la derecha). En ambos grupos, la distinción entre tipo y variable había sido trabajada previamente en clase.

1	2	3	4	5	6	7	8
17	25	103	14	92	36	87	55

```
const int N = 8;
typedef int arreglo[N];
arreglo arre;
```

```
const N = 8;
type arreglo = array [1..N] of integer;
var arre : arreglo;
```

Para ejercitar la sintaxis e introducir al estudiante en la semántica de dicha estructura de datos, la actividad le propone que escriba en papel cuatro pequeños fragmentos de programa (los cuales escribe en el lenguaje usado por su grupo) que resuelven problemas sencillos relativos al manejo de arreglos y además permiten ejercitar su integración con otros elementos del lenguaje (trabajados en clase previo al estudio). Los fragmentos piden resolver los siguientes problemas, asumiendo que las celdas de `arre` fueron previamente cargadas con valores:

1. Sumar los valores de las primeras dos celdas y guardar el resultado en una variable entera (`suma`)
2. Mostrar en pantalla un mensaje indicando si el valor en la celda 3 es par o impar
3. Determinar si el valor en la celda 1 es o no igual al valor en la última celda y guardar el resultado en una variable booleana (`iguales`)
4. Recorrer el arreglo de la primera a la última celda e ir mostrando por pantalla sus valores

Los fragmentos 1, 2 y 3 buscan que el estudiante ejercite el acceso a celdas concretas del arreglo y comprenda que puede operar con ellas de la misma manera que con cualquier otra variable entera, realizando las mismas operaciones y construyendo expresiones tanto enteras como booleanas. Los fragmentos 3 y 4 buscan que el estudiante ejercite el acceso a celdas utilizando índices dados por *variables* o *expresiones* en vez de por valores *constantes*. En los fragmentos 2, 3 y 4, el estudiante debe elegir la estructura de control más adecuada de entre las presentadas y explicar por qué razón la elige.

Un aspecto importante es que esta actividad ayuda al estudiante a despegarse de una instancia concreta del problema (caso de las puertas) y lo acerca al problema más general de buscar un valor en un arreglo cualquiera, que es lo que deberá programar en la siguiente actividad. Los fragmentos 3 y 4 plantean deliberadamente problemas relativos a la *última* celda, cuyo índice surge de la constante N, pero sin mencionar el valor concreto de dicho índice en el enunciado de la actividad. Por ejemplo, para N = 8 en *Pascal*, si el estudiante la expresa como `arre[8]` en vez de `arre[N]` en el fragmento 3, o escribe ocho instrucciones separadas para mostrar en pantalla el contenido de cada celda en el fragmento 4, ello indica que el estudiante está apegado a los casos concretos. Para ayudarlo a separarse de ellos, se cambia el valor de la constante N repetidas veces si es necesario, hasta que comprenda que puede usarla en forma general.

Al igual que en la actividad anterior, en cada fragmento se pide al estudiante que escriba *todas* las versiones que sean necesarias, recurriendo nuevamente a la *automatización* hasta que logre hacerlo correctamente e induciendo a la reflexión por medio de preguntas en caso de errores. En esta actividad (y en las siguientes) la interacción es ahora con representaciones de objetos *formales* del lenguaje (un arreglo dibujado en papel) en lugar de con objetos *físicos* (tarjetas numeradas en la hilera) que representan objetos del problema original (puertas de una calle). En este punto, el agente externo es la computadora. Durante la automatización, la misma nuevamente está representada por el entrevistador, que ejecuta las instrucciones del estudiante como si fuera la computadora, trabajando sobre el arreglo dibujado.

3.5.3 Tercera actividad: *programa* para búsqueda lineal

En esta actividad se busca analizar el proceso de construcción de conocimiento del estudiante enfrentado al problema de escribir, compilar y ejecutar *un programa* que resuelva el problema de *buscar un valor dado dentro de un arreglo de valores enteros*. Se le pide que lo escriba a partir de la versión en *pseudocódigo* escrita en la primera actividad, haciendo uso de las reglas de sintaxis y semántica trabajadas en la segunda actividad. Tras lo decidido en el estudio piloto, se asumen ya declaradas las variables necesarias y previamente cargadas con valores. El estudiante debe traducir al *lenguaje formal* los pasos escritos usando e*l formalismo intermedio*. La traducción implica un proceso reflexivo, en el cual debe establecer una correspondencia entre los pasos realizados sobre la hilera de tarjetas y las *instrucciones* ejecutadas sobre la *estructura de datos* que la representa (el arreglo). Como las dos actividades previas, esta también implica la construcción de conocimiento sobre dos aspectos: uso de un *formalismo* (ahora *lenguaje de programación*) y ejecución por parte de un *agente externo* (ahora la *computadora*). La construcción abarca no sólo la parte *textual* del programa sino también su parte *ejecutable*, y la relación entre ambas.

El diseño de la actividad debe tener en cuenta que la lógica del algoritmo fue construida en la primera actividad. Dado que ahora la *computadora* es el agente externo, se debe buscar que el estudiante se enfoque en cuestiones propias de la ejecución en máquina, no necesariamente vinculadas a la lógica del algoritmo. Algunas son, por ejemplo, un error de *salida de rango* o una iteración que no termina (*loop infinito*). Lo primero ocurre si alguna una instrucción accede a una celda mediante `arre[exp]`, siendo `exp` una expresión cuyo valor no pertenece al rango de índices válidos del arreglo. El estudiante debe tomar conciencia de que la expresión `exp` no puede adquirir cualquier valor entero, por más que su tipo sea entero. Lo segundo ocurre si nunca se cumple una condición que detenga la ejecución de la iteración que realiza la búsqueda (por ejemplo, usa `while` con una variable `i` para acceder a cada celda, pero nunca la incrementa). Debe tomar conciencia de que las instrucciones deben ir reduciendo la cantidad de celdas que restan visitar hasta que, en algún momento, se llegue a cumplir una condición que detenga la iteración. Si bien errores como estos están presentes en el *texto* del programa, es en la ejecución cuando se revelan sus consecuencias.

La metodología que guía el desarrollo de esta actividad es igual que para las dos actividades previas: el estudiante escribe una primera versión en papel de su programa y se recurre a la automatización (robot imaginario interpretado por el entrevistador, simulando ser la computadora) al menos dos veces: una para buscar un valor que está en el arreglo y otra para buscar otro que no. El arreglo está dibujado en papel junto con una flecha o marca para señalar el índice de la celda actual. El comportamiento observado durante la automatización permite al estudiante corregir errores y escribir nuevas versiones. Se le hacen preguntas dirigidas a situar su pensamiento en lo que la *computadora* debe hacer para resolver el problema (no el estudiante). Por ejemplo, un error de *salida de rango* guarda estrecha relación con la evaluación de expresiones booleanas por *circuito corto* o por *circuito completo*. La intención de las preguntas es dirigir el pensamiento del estudiante hacia esa característica de los lenguajes de programación (la mayoría evalúa por circuito corto) moviéndolo desde una "semántica humana" donde el orden de los operadores booleanos no es relevante hacia una "semántica computacional" donde sí es relevante. Como en la primera parte, las preguntas concretas se adaptan a cada estudiante según su versión del programa. Todo el proceso se repite hasta que el estudiante construye una versión final que, desde el punto de vista de la ejecución sobre el arreglo dibujado en papel, resuelve el problema correctamente y el estudiante logra explicar cómo y por qué.

La actividad finaliza pidiéndole que transcriba a la computadora el texto del programa escrito en papel, lo compile y lo ejecute al menos dos veces: una para buscar un valor que existe en el arreglo y otra para buscar otro que no. La ejecución en máquina constituye la validación del conocimiento construido sobre la parte *ejecutable*, dado que es la interacción entre el sujeto y el verdadero agente externo en acción (la *computadora*). Las instrucciones para cargar el arreglo con valores y solicitar el ingreso del valor a buscar ya se le proporcionan previamente escritas en un archivo fuente. El estudiante debe completar la porción faltante, correspondiente a la búsqueda lineal y luego emitir un mensaje indicando el resultado. Se espera que no surjan errores de compilación ni de ejecución, debido a la consolidación de los conceptos alcanzada en las etapas anteriores. No obstante, como el proceso es dialéctico, puede suceder (quizás debido a una distracción en la transcripción del papel a la máquina, o alguna instrucción errónea pasada por alto durante el proceso). De producirse algún error de compilación, significa que persiste algún error de sintaxis o semántica, en cuyo caso se procede a leerlo y se le pregunta qué debe hacer para corregirlo. De ocurrir algún error durante la ejecución, se vuelve al arreglo en papel y se repite nuevamente automatización y preguntas, a efectos de detectar el problema y volver a la computadora para corregirlo y luego compilar y ejecutar otra vez el programa.

3.5.4 Cuarta actividad: programa para *nuevo* problema

El proceso descrito por medio de las tres actividades previas aumenta el conocimiento del estudiante desde un nivel *instrumental* (etapa *intra*) a un nivel *formal* (etapa *trans*), pasando por un nivel *conceptual* (etapa *inter*), cada uno evidenciado por los logros del estudiante en cuanto a la formulación de soluciones y explicación de las razones de éxito. El proceso de construcción de conocimiento es *continuo* [75] y en particular la etapa *trans* posibilita enfrentar al estudiante con *nuevos* problemas que presentan similitudes y diferencias con los ya resueltos, de modo de avanzar a la construcción de conceptos más generales y complejos. En el marco del estudio, se decidió explorar este aspecto como introducción para trabajos futuros, para lo cual se diseñó una cuarta actividad destinada a sentar algunas bases para dichos trabajos.

Esta actividad consiste en pedirle al estudiante que escriba, *directamente* en la computadora, otro programa que resuelva un *nuevo* problema: *obtener un listado de todos los valores pares almacenados en el arreglo* y que explique por qué lo resuelve del modo en que lo hace. Se trata de un problema similar en la *estructura de datos* pero diferente en el *algoritmo* a aplicar. En vez de ser un problema de *búsqueda*, se trata de un problema de *filtrado*, que requiere una recorrida *completa* del arreglo para su resolución. Este tipo de estudio ha sido profundamente analizado en la teoría y esbozado para algunos problemas y soluciones algorítmicas en trabajos previos [6, 30, 31]. Esta actividad busca obtener información sobre la construcción de conocimiento al trabajar directamente sobre *programas* (sin pasar por *pseudocódigo*) y comenzar a explorar la relación entre el conocimiento *formal* ya construido para un problema previo y el conocimiento relativo a un nuevo problema.

Tras escribir el nuevo programa, se pide al estudiante que proceda directamente a compilarlo y ejecutarlo en máquina. De producirse algún error de compilación, se le pregunta cómo haría para corregirlo (como en la actividad anterior). De ocurrir algún error en la ejecución, se recurre nuevamente a la automatización sobre un arreglo dibujado en papel y se plantean preguntas como en las actividades previas. Si resuelve correctamente el problema, pero hay algún elemento del lenguaje que no es el más adecuado, se le pregunta al respecto y se repasan los elementos sintácticos y/o semánticos necesarios. Por ejemplo, si usa nuevamente `while`, a pesar de que sería preferible usar `for`, dada la naturaleza del nuevo problema. En tal caso, se puede incluso recurrir nuevamente a la automatización para que constate que itera sobre el arreglo completo usando una estructura iterativa que no es la más adecuada. Posteriormente, se vuelve a la computadora para que corrija el programa, compile y ejecute de nuevo, repitiendo el proceso de ser necesario, hasta llegar finalmente a una versión correcta.

3.5.5 Fundamentación teórica

El diseño de las primeras tres actividades se fundamenta por la extensión a la ley general de la cognición presentada en el capítulo 2 (sección 2.5.3), según la cual un aspecto fundamental en el pasaje a la etapa *trans* es la comprensión de que las acciones del algoritmo ya no son realizadas por el propio sujeto, sino por un agente externo (la *computadora*). Las acciones son las *instrucciones* ejecutadas por la misma, mientras que los objetos son las representaciones en memoria de las *estructuras de datos* del programa (en este caso, el *arreglo*).

Existe una relación de causa y efecto entre las instrucciones del programa (parte *textual*) y el comportamiento ejecutado por la computadora (parte *ejecutable*). La construcción de conocimiento sobre *programas* implica una conceptualización sobre ambos aspectos (parte textual y parte ejecutable) y su relación. El segundo aspecto implica además la concepción del programa como *objeto ejecutable*, en línea con la naturaleza *dual* de los programas establecida por Moor en [53], lo cual es también un factor preponderante en el proceso, ya que implica la comprensión de cómo la computadora ejecuta las instrucciones. Por ejemplo, entender que el orden de los operandos al evaluar expresiones booleanas con `and` y `or` es relevante (según la evaluación se haga por *circuito corto* o por *circuito completo*) es consecuencia de concebir al programa como un objeto ejecutable.

Así como la ley general la cognición regula el proceso de construcción de conocimiento *conceptual* sobre el algoritmo, su extensión regula el proceso de construcción de conocimiento *formal* sobre el *programa* que lo implementa. Este proceso se ilustra por la segunda línea del diagrama que expresa la ley extendida (*newC* ← *newP* → *newC*'). El conocimiento *conceptual* construido por cada estudiante (evidenciado por la descripción de su solución en lenguaje natural) sitúa su pensamiento en una nueva periferia (*newP*). A partir de allí, se continúa el proceso que busca que tome conciencia de, por un lado, la formalización del algoritmo en *instrucciones* que, al ser ejecutadas por la computadora, le permiten a la misma resolver el problema (transición hacia *newC*) y por el otro, el hecho de que el éxito en la solución radica en las modificaciones que la ejecución de las instrucciones imponen a las *estructuras de datos* (transición hacia *newC*'). En dicha toma de conciencia radica la construcción de conocimiento sobre el programa y su ejecución.

El medio que se utiliza para instruir a la computadora es el *lenguaje de programación* (lenguaje *formal*), el cual constituye *otro* objeto sobre el cual el sujeto construye conocimiento. Dicha construcción puede darse en paralelo con la construcción de conocimiento sobre el algoritmo en cuestión, o puede haberse dado con antelación. Las personas a menudo aprenden nuevos algoritmos en conjunto con el aprendizaje de algún lenguaje de programación, y otras veces los aprenden y luego los formalizan usando un lenguaje que ya conocen de antemano. En el marco del presente estudio, el estudiante construye conocimiento sobre la *sintaxis* y *semántica* para la manipulación

de *arreglos* en el lenguaje de programación (*C* o *Pascal*) a la vez que construye conocimiento formal sobre la búsqueda lineal.

El proceso implica la escritura de *instrucciones* en el lenguaje formal que expresen acciones equivalentes a las que el sujeto describió en lenguaje natural durante el pasaje a la etapa *inter*, pero pensando en que será la computadora quien las ejecute sobre las estructuras de datos. El pasaje de la descripción en lenguaje natural a instrucciones del lenguaje de programación es un proceso complejo, que requiere establecer una correspondencia entre las acciones realizadas sobre los objetos manipulados por sí mismo en la acción (las tarjetas) y las instrucciones ejecutadas por la computadora sobre las estructuras de datos que los representan (las celdas del arreglo). En este pasaje subyace la relación dialéctica entre la primera y la segunda línea del diagrama que expresa la ley extendida. El conocimiento *conceptual* (etapa *inter*) se transforma en *formal* (etapa *trans*) cuando el sujeto establece dicha correspondencia y además su pensamiento consolida las transiciones desde *newP* hacia *newC* y *newC*'.

Conforme a lo expresado en el capítulo 2 (sección 2.5.2), el uso de un *formalismo intermedio* (pseudocódigo) constituye un facilitador del pasaje de lenguaje natural a instrucciones, pues ayuda al sujeto a aproximarse gradualmente a la formalización sin que los elementos sintácticos y semánticos del lenguaje de programación constituyan una dificultad extra. Particularmente cuando, en paralelo, se construye conocimiento sobre elementos del lenguaje de programación a utilizar para escribir el programa (como sucede en este caso). El uso del formalismo intermedio permite que luego, cuando se pasa del mismo a la escritura del *programa*, el foco esté puesto en cómo usar las reglas de sintaxis y semántica del lenguaje formal para expresar los pasos ya escritos en pseudocódigo. Por ejemplo, al usar la estructura `while` para expresar la iteración, el sujeto se concentra en usar adecuadamente la sintaxis de dicha estructura sin necesitar reflexionar acerca de por qué es la más adecuada, pues ya lo hizo al momento de elegir *Mientras* para la versión en pseudocódigo. A su vez, la elección de dicha estructura había surgido de la descripción en lenguaje natural dada inicialmente.

El pasaje de la descripción del algoritmo en lenguaje natural a una nueva expresión del mismo usando el formalismo intermedio también se rige por la ley de la cognición extendida. La misma explica el proceso de construir conocimiento para instruir a un *agente externo*. En el caso del programa, el agente externo es la *computadora* mientras que, en el algoritmo en pseudocódigo, el agente externo es el *robot imaginario* que ejecuta sus pasos. En el marco de la ley extendida, el pensamiento del estudiante parte de *newP* e inicia las transiciones hacia *newC* y *newC*'. Debe tomar conciencia de que las acciones son ahora los *pasos* del algoritmo llevados a cabo por el robot (avance hacia *newC*) y que el éxito en la solución surge de los cambios que ellos imponen sobre la hilera de tarjetas (avance hacia *newC*'). Dichas transiciones continúan luego durante el pasaje del *pseudocódigo* al texto del *programa*.

El pensamiento del estudiante se evidencia en *newP* cuando la primera versión en pseudocódigo no expresa adecuadamente la relación entre acciones y cambios. Por

ejemplo, si expresa la detención solamente al encontrar el documento (*Mientras no encuentre el documento hacer…*) denota que está centrado en el resultado deseado y solo toma en cuenta uno de los dos cambios. Necesita entonces avanzar hacia *newC*', para incluir el otro (parar cuando no quedan puertas). Por otra parte, si no incluye todas las acciones necesarias (por ejemplo, no avanza a la siguiente puerta), no es consciente de que instruir al agente externo para realizar cada acción es esencial para alcanzar la terminación. Necesita avanzar hacia *newC*. El estudiante ha construido conocimiento conceptual sobre el algoritmo (tras su aplicación por sí mismo), pero aún no es capaz de instruir adecuadamente al agente externo para su aplicación.

La estrategia empleada para inducir las transiciones desde *newP* hacia *newC* y *newC*' (tanto al pasar de la descripción en lenguaje natural a pseudocódigo como de este último al programa) se basa en el mecanismo de *automatización* [29, 30], combinado con preguntas similares a las de la primera parte, las cuales se adaptan a cada estudiante. La automatización resulta de gran utilidad, pues evidencia qué pasos del algoritmo (*instrucciones*, para el caso del programa) o ausencia de ellos producen un comportamiento incorrecto o no esperado. Ayuda al estudiante a visualizar la relación entre las acciones y los cambios que imponen a los objetos (*estructuras de datos*, para el caso del programa), posibilitando la escritura de nuevas versiones mejoradas hasta llegar a una correcta. En el caso del *programa*, también resulta fundamental para visualizar y comprender resultados de la ejecución en computadora que generarían errores en tiempo de ejecución (por ejemplo, *salida de rango*) o situaciones no deseadas (por ejemplo, *loop infinito*), lo cual es fundamental para consolidar las transiciones desde *newP* hacia *newC* y *newC*' y así la transformación de conocimiento *conceptual* en conocimiento *formal*.

La *automatización* contribuye no solamente a la detección de errores y facilitar la conceptualización, sino también a la construcción de un programa *general*. Si bien en la teoría de Piaget el salto de instancias concretas al caso general se investiga en cualquier etapa de la construcción de conocimiento (problema del *elemento genérico*, Matalon [9]), en este estudio se analiza especialmente en el pasaje a la etapa *trans*. A grandes rasgos, Matalon concluye sus investigaciones diciendo que para construir la noción de elemento *genérico*, es necesario realizar una *acción genérica* que, por repetición sucesiva, permite construir dicha noción. En este estudio se interpretan los resultados de Matalon instando a los estudiantes a trabajar con arreglos de diferentes tamaños, para construir la noción de *cantidad genérica* de elementos, utilizando la constante N en el texto del programa para referirse al tamaño del arreglo en vez de hacer mención al valor concreto de N en una instancia concreta.

En resumen, la extensión a la ley general de la cognición explica la continuidad del proceso de construcción de conocimiento sobre algoritmos, estructuras de datos y programas desde lo *instrumental* a lo *formal*, pasando por lo *conceptual*. Al construir conocimiento sobre *programas*, los mismos se toman como objetos de naturaleza *dual* (textual y ejecutable). Las acciones realizadas sobre los objetos manipulados en la etapa *instrumental* (las tarjetas) se transforman en acciones mentales en el proceso de

conceptualización y luego en instrucciones ejecutadas por la computadora sobre las estructuras de datos que los representan (las celdas del arreglo) en el proceso de *formalización*.

Finalmente, el diseño de la cuarta actividad busca abrir la posibilidad de analizar la aplicación del conocimiento ya construido a la resolución de *nuevos* problemas. En la teoría de Piaget, este aspecto es una de las principales características de la etapa *trans*, la cual no solo implica la formalización de problemas concretos (por ejemplo, la búsqueda lineal), sino también la construcción de estructuras generales (lo que Piaget denomina *systèmes d'ensemble* en [13]). La herramienta cognitiva fundamental en el proceso de aplicación de conocimiento construido previamente a nuevos problemas es la *generalización* [16] en sus dos formas: *inductiva y constructiva* (presentadas en el capítulo 2, sección 2.3.2). En la cuarta actividad se analiza el pasaje a la etapa *trans* específicamente para la construcción de conocimiento formal sobre un nuevo problema (filtrado de valores pares del arreglo). Los resultados preliminares de la misma serán usados como insumo para futuras investigaciones, junto con resultados de otros trabajos previos [6, 30, 31], en los que se han propuesto nuevos problemas que presentan similitudes y diferencias en relación a un problema anteriormente resuelto con éxito. Se espera a futuro expandir el modelo para analizar la construcción de conocimiento formal sobre *familias de algoritmos*, entre otros aspectos.

3.6 Desarrollo del estudio: segunda parte (*inter* → *trans*)

En esta sección se presentan extractos de las respuestas escritas por los estudiantes para las cuatro actividades correspondientes a la segunda parte del estudio (pasaje de la etapa *inter* a la etapa *trans*) y un análisis global por cada una. Dichas actividades consisten en la escritura de:

1. El algoritmo en *pseudocódigo* que resuelve el problema de la búsqueda lineal
2. Los cuatro fragmentos de programa para ejercitar las reglas de *sintaxis* y *semántica* del lenguaje de programación (necesarias para las siguientes dos actividades)
3. El código del *programa* que resuelve el problema de la búsqueda lineal
4. El código del programa que resuelve el *nuevo* problema (listar valores pares del arreglo)

En esta parte del estudio, se le pide a cada estudiante que escriba *todas* las versiones necesarias para cada respuesta, dado que el objetivo es investigar el proceso *completo* al pasar de la etapa *inter* a la etapa *trans*. El propósito final en cada actividad es que el estudiante logre construir una solución correcta por sí mismo y de forma progresiva. La cantidad de versiones escritas puede variar de un estudiante a otro, dependiendo de los errores cometidos, que pueden estar en aspectos de sintaxis y/o semántica como de comportamiento. Entre cada versión y la siguiente, se recurre a la *automatización* para que un *agente externo* (primero un robot imaginario para el caso de *pseudocódigo* y luego la computadora para el caso de los *programas*, ambos interpretados por el

entrevistador) lleve a cabo los pasos de la solución y el estudiante observe el comportamiento generado por su ejecución y responda preguntas orientadas a que reflexione sobre el mismo. De ser necesario, se vuelven a repasar las reglas de sintaxis y semántica involucradas.

Al igual que en la primera parte, se incluyen extractos de las respuestas de los estudiantes junto con sus análisis preliminares, concretamente del algoritmo en *pseudocódigo* para la búsqueda lineal (3.6.1 y 3.6.2), las reglas de *sintaxis* y *semántica* del lenguaje de programación (3.6.4), el código del *programa* para la búsqueda lineal (3.6.6 y 3.6.7) y el código del *programa* para el *nuevo* problema (3.6.9). Los respectivos análisis globales de resultados, incluyendo la totalidad de los estudiantes, se presentan en las secciones 3.6.3, 3.6.5, 3.6.8 y 3.6.10. Luego, en la sección 3.7, se presentan las conclusiones de la segunda parte del estudio.

3.6.1 Primera actividad: *pseudocódigo* para búsqueda lineal

Para ilustrar un ejemplo del proceso *completo* de construcción del algoritmo en *pseudocódigo*, se presenta a continuación el trabajo de un estudiante concreto (Juan). Se incluyen *todas* las versiones que escribe hasta llegar a una final que resuelve el problema. Se realiza también un análisis preliminar de distintos aspectos en cada versión y del proceso de construcción por parte del estudiante. La totalidad de las versiones escritas por todos los estudiantes, junto con sus análisis correspondientes, se encuentran en el anexo B, incluyendo también preguntas realizadas a cada estudiante entre versión y versión, junto con sus respuestas. El proceso comienza a partir de la descripción en lenguaje natural dada por el estudiante al finalizar la primera parte del estudio. Para el caso de Juan, su descripción es la siguiente:

Comienzo a recorrer las puertas comenzando por la que tengo más cerca. Toco en la primera, pregunto por el número de cédula, si no es el que busco, sigo con la siguiente, y así sucesivamente hasta encontrarlo. En caso de no encontrarlo, recorro todas las puertas.

Juan constituye un ejemplo de estudiante que no conceptualizó sólidamente todos los aspectos relevantes del algoritmo en la primera parte (su análisis detallado está en el anexo A). Hace explícitas las acciones realizadas solamente para la primera puerta, pero las deja implícitas para las demás, así como los cambios en los objetos. Según lo expresado en la sección 3.4, esta actividad posibilita nuevas repeticiones de la tarea, lo cual le permitirá consolidar la conceptualización de dichos aspectos, a la vez que da inicio al pasaje a la etapa *trans*. La primera versión en pseudocódigo de Juan es:

Mientras (***no encuentro nro cedula***)
Golpeo puerta
Pregunto por cedula
Verifico si es la buscada
Si es
Paro
Sino
Termino
Fin
Fin

En esta versión inicia la búsqueda posicionado frente a la primera puerta, previo a consultar el documento en ella. Empieza sin haber encontrado el documento buscado y lo expresa mediante la condición de la estructura de iteración *Mientras* (*no encuentro nro cédula*), que inicialmente es verdadera, por lo que comienza a iterar. En relación a los cambios en objetos, pretende parar si encuentra el documento, pero no incluye una segunda condición para parar cuando no haya más puertas. Sin embargo, toma en cuenta la detención por dicha razón en su descripción inicial (*En caso de no encontrarlo, recorro todas las puertas*).

De las tres acciones a realizar, la comparación de documentos (expresada por *si no es el que busco* en su descripción en lenguaje natural) se traduce al paso *Verifico si es la buscada* en pseudocódigo. Respecto a la repetición, en lenguaje natural la deja implícita al decir y *así sucesivamente*, pero ahora la vuelve explícita al utilizar la estructura de iteración. El formalismo intermedio le permite extender a más puertas acciones previamente implícitas o bien explícitas solamente para el caso de la primera puerta. El avance a la siguiente puerta está incluido en su descripción inicial (*sigo con la siguiente*), pero no en pseudocódigo (pone *Termino* tras *Sino*).

El pensamiento de Juan está iniciando dos nuevas transiciones, desde *newP* hacia *newC* y *newC'*. Su primera versión muestra que está centrado en el resultado deseado: encontrar el documento (*newP*). Ha conceptualizado aspectos del algoritmo tras su aplicación por parte de él mismo y ahora debe hacerlo en términos de la instrucción al robot imaginario para que lo aplique. Por estar centrado en el resultado deseado, no considera el otro cambio posible (que se terminen las puertas sin encontrar el documento). Esto lleva a que en su pseudocódigo solo tome en cuenta una de las dos condiciones de parada (su pensamiento necesita avanzar hacia *newC'*). Tampoco explicita todas las acciones, plasmando solamente dos de ellas: comparación y repetición, pero no avance de puerta (necesita avanzar hacia *newC*). La automatización le permite mejorar en este sentido, al recurrir a ella constata que llega a inspeccionar únicamente la primera puerta, tras lo cual escribe una segunda versión.

Mientras (*no encuentro nro cedula*)
Golpeo puerta
Pregunto por cedula
Verifico si es la buscada
Si es
Paro
Sino
Sigo a la siguiente puerta
Fin

Fin

Su segunda versión corrige el problema al cambiar *Termino* por *Sigo a la siguiente puerta*. Su pensamiento ha mostrado un avance en la transición a *newC*, ya que ahora toma en cuenta la acción de avance a la siguiente puerta, pero todavía sigue sin contemplar la otra condición de parada. Al recurrir nuevamente a la automatización, detecta el problema al intentar buscar un documento que no existe. Pretende seguir buscando luego de consultar la última puerta y no haberlo encontrado, constatando la necesidad de considerar la otra condición, tras lo cual escribe una tercera versión.

Mientras ((*no encuentro nro cedula*) ***OR* (*no hay mas puertas*)**)
Golpeo puerta
Pregunto por cedula
Verifico si es la buscada
Si es
Paro
Sino
Sigo a la siguiente puerta
Fin
Fin

Su tercera versión intenta incluir ambas condiciones de parada, pero lo hace usando el operador *OR*, de manera incorrecta. Si bien ha avanzado hacia *newC*' al procurar reflejar ambas, aún presenta dificultad en la conceptualización de los cambios. En su descripción inicial dice *hasta encontrarlo* para referirse a la primera condición. Al combinarla con la segunda, hace uso de la disyunción, como lo haría en lenguaje natural (*o que no haya mas puertas*). Esto lo induce a usar inicialmente *OR* en lugar de *AND*. Es otra manifestación de que su pensamiento necesita seguir transitando hacia *newC*'. Debe reflexionar sobre la relación entre la semántica de los operadores booleanos y de la estructura *Mientras*, a efectos de expresar la detención en forma acorde (debe hacerlo de manera que la iteración finaliza cuando alguna de las condiciones *deja* de cumplirse). El estudiante ha trabajado en clase con operadores booleanos y *Mientras*, previo a la realización del estudio, pero su conceptualización aún no es lo bastante sólida.

Nuevamente, la automatización le permite mejorar en este sentido. Se le pide ahora buscar un documento que no existe, y constata que la condición compuesta se sigue

cumpliendo tras visitar la última puerta (debido al uso de *OR*). Consultado al respecto, efectivamente piensa que ello implica parar la recorrida. Se le señala que la semántica de la estructura *Mientras* es tal que para de iterar cuando la condición compuesta *deja* de cumplirse (en vez de parar cuando aún se cumple) y escribe una cuarta versión en la cual procura detenerse cuando la condición ya no se cumple.

Mientras (**(*encuentro nro cedula*) *AND* (*hay puertas*)**)
    Golpeo puerta
    Pregunto por cedula
    Verifico si es la buscada
    Si es
        Paro
    Sino
        Sigo a la siguiente puerta
    Fin
Fin

En su cuarta versión cambia el uso de *OR* por *AND* y quita la negación en ambas condiciones, mostrando un avance en la conceptualización, pero aún necesita pulir la combinación de ambas. Entiende que existe una relación entre el cambio de operador y la negación de las condiciones involucradas, pero falla en hacer las modificaciones de forma que logren el comportamiento deseado (debía quitar la negación solamente en la segunda condición). Al volver una vez más a la automatización, constata el error al comprobar que ahora la iteración no se realiza ni una vez (al instante de empezar, nota que la primera condición es falsa al evaluarla por primera vez), tras lo cual escribe una quinta versión.

Mientras ((**no** *encuentro nro cedula*) *AND* (*hay puertas*))
    Golpeo puerta
    Pregunto por cedula
    Verifico si es la buscada
    Si es
        Paro
    Sino
        Sigo a la siguiente puerta
    Fin
Fin

Finalmente, su quinta versión es correcta. Contempla ambas condiciones de forma correcta y las tres acciones son explícitas: repetición (*Mientras*) comparación (*Verifico si es la buscada*) y avance (*Sigo a la siguiente puerta*). Ha construido una versión correcta que resuelve el problema. Su pensamiento finalmente se aparta de *newP* al comprender que el éxito en la solución se da debido a los cambios impuestos por el agente externo a las tarjetas (avance hacia *newC*') como consecuencia de las acciones realizadas (avance hacia *newC*). Dichas transiciones continuarán luego en el pasaje del pseudocódigo al texto del *programa* (tercera actividad, sección 3.6.6).

Un proceso como el de Juan fue llevado a cabo con los demás estudiantes. Todos necesitaron escribir al menos dos versiones hasta llegar a una final correcta. Durante el transcurso, se obtuvo información variada sobre el proceso de construcción de conocimiento acerca de las acciones realizadas por el agente externo y los cambios que ellas imponen a los objetos. A continuación, se presenta una selección de algunas versiones del algoritmo escritas por otros estudiantes, junto con un análisis preliminar del proceso de construcción observado en relación a ambos aspectos (acciones y cambios). Al igual que en la primera parte, se presentan en dos grupos, uno para cada aspecto. Las versiones que se presentan no necesariamente son todas finales, por lo que el grado de conceptualización evidenciado varía según cada versión.

Acciones realizadas

Todos los estudiantes usan alguna estructura de iteración y comienzan la búsqueda posicionados frente a la primera puerta. Algunos eligen consultar el documento en dicha puerta dentro del cuerpo de la iteración (como Juan, al comienzo de la sección) y otros eligen hacerlo previo a comenzar a iterar (como Nicolás, ver en breve).

La mayoría explicita las tres acciones, mostrando un alto grado de conceptualización sobre ellas. Algunos ya lo hacen incluso en versiones iniciales del algoritmo que no son correctas. Por ejemplo, Nicolás considera las tres en su primera versión: comparación (*cédula es distinta de x*), avance (*golpear puerta siguiente*) y repetición (*Mientras*):

golpear la primera puerta
Mientras (***cédula es distinta de x***)
    golpear puerta siguiente
    preguntar cédula
Fin

Otros estudiantes muestran un grado de conceptualización algo menor, al explicitar dos de las acciones y dejar implícita la tercera. Esto se constata no solo en versiones iniciales, sino también en versiones posteriores, llegando incluso a algunas finales, como es el caso de Ignacio D, cuya versión final es la siguiente:

Mientras (*no haya encontrado a la persona*) *AND* (*no se hayan acabado las puertas*)
    Pregunto en la puerta que estoy quien vive
    Avanzo a la siguiente puerta
Fin

Ignacio D hace explícitas la repetición (*Mientras*) y el avance (*Avanzo a la siguiente puerta*), pero deja implícita la comparación. Asume que una respuesta a *pregunto en la puerta que estoy quien vive* implica la obtención del documento en dicha puerta, dejando implícita su comparación con el documento buscado en la próxima evaluación de la primera condición de *Mientras*. Esto no es un problema en pseudocódigo, ya que igual es lo bastante claro para ser interpretado por una persona haciendo de robot imaginario en vez de una computadora. Tendrá oportunidad de hacer explícita la comparación en la tercera actividad (escritura del programa, sección 3.6.6).

La repetición es la única acción que la totalidad de estudiantes incluye en todas sus versiones, al hacer uso de una estructura iterativa ya desde su primera versión. La mayoría utiliza *Mientras* desde el inicio, excepto por dos estudiantes (Mónica y Valeria) que usan *Para cada* en su versión inicial. Por ejemplo, la de Mónica es la siguiente:

Para cada puerta *hacer*
    Golpear puerta
    Preguntar C.I
    Si encuentro la C.I entonces
        Termino
    Sino
        Sigo preguntando
    Fin
Fin

Conforme a la semántica de *Para cada*, la recorrida continúa hasta el final de la hilera, aún tras encontrar el documento. Sin embargo, el paso *Termino* sugiere la detención en ese momento, mostrando que quizás la estudiante piensa que puede cortar la iteración anticipadamente. Consultada al respecto (previo a la automatización) expresa la intención de recorrer todas las puertas, a pesar de haber escrito ese paso. Por lo tanto, no se trata de una diferencia entre lo que escribe y lo que piensa. Esto puede deberse a una conceptualización incompleta en la primera parte del estudio, pues su descripción en lenguaje natural (ver anexo A) explicita la detención solamente cuando encuentra el documento en la primera puerta pero no la hace explícita en caso de encontrarlo en alguna de las otras. Quizás realmente piensa que necesita recorrer toda la hilera. Tras volver a la automatización para buscar un documento que está en una puerta que no es la última, toma conciencia de la necesidad de parar cuando llega a ella (cambia a *Mientras* en su siguiente versión). Lo mismo sucede con la otra estudiante (Valeria).

Todos los estudiantes incluyen las tres acciones (con mayor o menor explicitud) en todas sus versiones, excepto únicamente por Juan (cuyo proceso completo se describe arriba, al comienzo de la sección). Es el único que incluye solo dos (repetición y comparación) en su versión inicial, agregando la tercera (avance) a partir de su segunda versión.

Cambios en objetos

Hay un solo estudiante (Ignacio D) que contempla los dos cambios posibles ya desde su primera versión, incluyendo las condiciones correspondientes en su estructura *Mientras*. El problema es que las combina con *OR* en lugar de *AND*, de tal modo que no expresa adecuadamente la detención.

Mientras ***no haya encontrado a la persona OR no se hayan acabado las puertas***
    Pregunto en la puerta que estoy quien vive
    Avanzo a la siguiente puerta
Fin

Los demás estudiantes incluyen solamente una condición en su versión inicial. Toman conciencia de la necesidad de incluir la otra tras recurrir a la automatización. En este sentido, Ignacio D ya muestra un grado de conceptualización más sólido desde el inicio, pues sus dos condiciones son correctas, pero falla al combinarlas mediante *OR*. De los demás, aproximadamente la mitad expresa la detención solamente al encontrar el documento (por ejemplo, Juan) mientras que el resto la expresa solamente cuando se acaba la hilera. Por ejemplo, Tomás, en su primera versión:

Mientras (***hayan puertas por ver***)
    preguntar si es el número
    Si (*está el número*)
        elijo esa puerta
    Sino
        sigo
    Fin
Fin

Tomás, al igual que otros estudiantes que incluyen solamente esta condición, también está centrado en el mismo resultado deseado (*newP*) que los que incluyen solamente la otra (parar cuando se encuentra el documento). A pesar de que el algoritmo escrito expresa la detención solamente cuando no hay más puertas por ver, igualmente tiene intención de parar al encontrar el documento (lo que escribe en su pseudocódigo no es acorde a lo que piensa). Esto se detecta al consultarle al respecto al volver a la automatización. Cree que la inclusión del paso *elijo esa puerta* alcanza para terminar de iterar en ese momento, sin volver a evaluar una vez más la condición de *Mientras* (se le hace notar que no es así, conforme a la semántica vista en clase). Como sucede con otros estudiantes (Juan entre ellos), esto se vincula a una conceptualización pobre de la semántica de la estructura *Mientras*.

3.6.2 Aspectos inherentes al *algoritmo* y al *formalismo intermedio*

En la sección anterior se presentaron diversos ejemplos de versiones del algoritmo escritas por los estudiantes y se realizó un análisis preliminar de aspectos vinculados a la construcción de conocimiento sobre el mismo (acciones realizadas y cambios en objetos). En esta sección se presentan más ejemplos, pero ahora se analizan desde otra perspectiva, tomando en cuenta otros aspectos detectados durante el estudio. Por ejemplo, cuestiones de manejo de sintaxis y semántica del *formalismo intermedio* o de la *lógica* del algoritmo, no necesariamente vinculadas al proceso de conceptualización. Los ejemplos se presentan nuevamente en grupos, según los aspectos detectados (son cinco: *cuestiones de sintaxis, cuestiones de semántica, estrategia de búsqueda, salto de instancia concreta al caso general* y *problemas de borde*).

Cuestiones de sintaxis del formalismo intermedio

Por tratarse de un formalismo intermedio, el *pseudocódigo* no posee reglas estrictas de sintaxis (a diferencia de un lenguaje de programación). Algunas ambigüedades

propias de dicha falta de rigor se evidencian en varias versiones de los estudiantes. Por ejemplo, Aaron es ambiguo en su manejo del operador *NOT* en una de sus versiones intermedias:

Ver la C.I de la puerta 1
Mientras ***NOT*** (*sea la que busco*) *OR* (***No*** *tengo mas Puertas*)
    Veo siguiente C.I
Fin

En la primera condición, usa *NOT* expresamente como un operador que antecede a otra condición, conforme a la sintaxis presentada en las tarjetas. En cambio, la segunda condición expresa la negación como haría en lenguaje natural (mediante la palabra *No*). Sin embargo, pretende lograr el mismo efecto en ambas, sin que la semántica de la negación se vea afectada ni que dificulte su comprensión por parte del estudiante. Este tipo de ambigüedades sintácticas son admisibles en pseudocódigo pero no así en el lenguaje de programación.

Por otra parte, hay tres estudiantes que ya incorporan algunos elementos propios del lenguaje de programación en sus versiones en pseudocódigo (variables, asignación, expresiones aritméticas y booleanas). Dado que esto no contradice las reglas de sintaxis dadas en las tarjetas para las estructuras en pseudocódigo, se les permitió hacer uso de dichos elementos. En consecuencia, las versiones escritas por estos estudiantes presentan menos ambigüedad sintáctica que las de otros estudiantes (lo cual no significa que carezcan de otros problemas). Por ejemplo, Ignacio U ya utiliza estos elementos en su primera versión:

nocedula := FALSE
contador := 1
Mientras (***nocedula = FALSE***)
    Analizo si en puerta 1 está la cédula
    Si (*no está*)
        contador := contador + 1
    Fin
    Si (***contador = 7***)
        nocedula := TRUE
    Fin
Fin

Cuestiones de semántica del formalismo intermedio

Varias versiones (no finales) de estudiantes tienen problemas relativos a la semántica de los operadores booleanos *AND* y *OR*. En algunos casos, son problemas propios de la no comprensión del comportamiento de cada uno, mientras que en otros se asocian a una conceptualización pobre de la semántica de *Mientras*. Por ejemplo, Juan (al inicio de la sección anterior) usa *OR* en una de sus versiones pensando que la semántica de *Mientras* es tal que para de iterar cuando la condición aún se cumple (en vez de parar cuando *deja* de cumplirse).

Otros estudiantes también muestran problemas en comprender la semántica de *Mientras* incluso sin operadores booleanos involucrados. Por ejemplo, Pablo M (en una versión intermedia):

Mientras (*haya puertas en la cuadra*)
golpeo la puerta
pregunto por fulanito (*cédula*)
Si (*fulanito está en la puerta*)
Termino la búsqueda
Sino
Voy a la siguiente puerta
Fin

Pablo M pretende detener la iteración ni bien llega al paso *Termino la búsqueda*, sin evaluar una vez más la condición de *Mientras* antes de terminar (esto se constata en la automatización, cuando manifiesta su intención de finalizar la búsqueda en ese instante). No ha conceptualizado que la semántica de *Mientras* exige evaluar una vez más la condición (se le hace notar esto previo a la escritura de su siguiente versión) sin importar la presencia del paso *Termino la búsqueda* en el pseudocódigo.

Estrategia de búsqueda usada

Todos los estudiantes comienzan la búsqueda posicionados frente a la primera puerta antes de empezar a iterar. Al observar la versión final escrita por cada estudiante, se constata que sigue una (y solo una) de las siguientes estrategias para la búsqueda:

1. ***A priori***: consulta el documento de la primera puerta **antes** de comenzar la iteración. Si el documento buscado no está en la hilera, queda posicionado en la última puerta al finalizar la recorrida.

2. ***A posteriori***: consulta el documento de la primera puerta **luego** de comenzar la iteración (es decir, dentro de la misma). Si el documento buscado no está en la hilera, avanza una vez más tras consultar la última puerta y recién entonces finaliza la recorrida.

Para denominar ambas estrategias, se introduce la clasificación *a priori* y *a posteriori*, a efectos de su fácil identificación en el resto del trabajo. Cuatro estudiantes siguen la estrategia *a priori* en su versión final, en tanto los nueve restantes siguen *a posteriori*. Todos llegan por sí solos a una de las dos estrategias, conforme escriben sucesivas versiones. En algunos casos, la estrategia empleada ya es notoria en la versión inicial y en otros se hace evidente conforme se suceden las versiones. Al diseñar el estudio no se pensó específicamente en ninguna de las dos estrategias. El arribo de cada estudiante hacia alguna de las dos fue algo que se constató en su realización.

Los estudiantes que siguen la estrategia *a priori* nunca llegan a posicionarse más allá de la última puerta. Tanto si el documento buscado está en la última puerta, como si no existe en la hilera, quedan igualmente posicionados en la última, variando la razón

por la que se detiene la iteración (una u otra condición de *Mientras*) dependiendo de si el documento está o no. Por ejemplo, Aaron sigue esta estrategia, como se aprecia en su versión final.

```
Ver la C.I de la puerta 1
Mientras NOT (sea la que busco) AND (tengo mas Puertas)
        Veo siguiente C.I
Fin
```

Los estudiantes que siguen la estrategia *a posteriori* avanzan una vez más tras haber visitado la última puerta cuando el documento buscado no está en la hilera, pero controlan adecuadamente la detención, evitando consultar por el documento de una puerta que no existe. Por ejemplo, Pablo M sigue esta estrategia en su versión final.

```
Mientras (haya puertas en la cuadra) AND (no encuentre a fulanito)
        golpeo la puerta
        pregunto por fulanito (cédula)
        Si (fulanito está en la puerta)
                Termino la búsqueda
        Sino
                Voy a la siguiente puerta
        Fin
Fin
```

Comienza posicionado frente a la primera puerta, sin haber visitado ninguna ni haber encontrado el documento de la persona buscada (a la cual llama *fulanito*). La primera vez que consulta por el documento es dentro de la iteración, tras haber evaluado por primera vez las dos condiciones de su estructura *Mientras* (las cuales son ambas verdaderas, por lo que empieza a iterar). Si el documento buscado no existe en la hilera, avanza una vez más tras visitar la última puerta y recién entonces finaliza la iteración (al evaluar, por última vez, la primera condición de *Mientras*).

Salto de instancia concreta a caso general

Al igual que en la primera parte, hay estudiantes que no completan el salto de la instancia concreta hacia el caso general (problema del *elemento genérico*, estudiado por Matalon [9]) ya que mantienen elementos de la misma (resuelta en la acción) en la versión que escriben en pseudocódigo. Es decir, no escriben un algoritmo que sirva para cualquier documento y cantidad de puertas, sino que lo hacen para el documento concreto y/o la cantidad puntual de puertas correspondientes a su instancia concreta.

A diferencia de la primera parte, en la que un estudiante (Pablo P) describe una instancia concreta para un documento que está en la tercera puerta, en la segunda ningún estudiante lo hace así. No obstante, algunos escriben su algoritmo atados a algún elemento propio de la instancia concreta. Por ejemplo, en una versión intermedia, Joaquín ya escribe un algoritmo aplicable a cualquier cantidad de puertas, pero atado al documento concreto buscado (417).

```
puerta := 1
abrir puerta := x
número := 417
Mientras (x <> número)
        puerta := puerta + 1
        abrir puerta := x
Fin
```

Otros estudiantes escriben versiones atadas a la cantidad de puertas de la instancia concreta, pero no al documento concreto. Por ejemplo, el propio Pablo P (que describe una instancia concreta del algoritmo en la primera parte) ahora escribe un algoritmo que busca un documento cualquiera, pero para el caso concreto de 5 puertas (las numera de 0 a 4) en su versión final:

```
numero = 0
puerta = 0
CI = numero conocido
Mientras (puerta ≤ 4) AND (numero != CI)
        pregunto numero en puerta
        Si (numero != CI)
                puerta = puerta + 1
        Fin
Fin
```

Si bien algunos estudiantes mantienen elementos de la instancia concreta incluso en sus versiones finales, no se les pide más versiones en pseudocódigo, dado que el salto al caso general se trata con más profundidad en la segunda actividad (sección 3.6.4).

Problemas de borde

Finalmente, hay dos estudiantes que presentan *problemas de borde* vinculados a la última puerta de la hilera en versiones intermedias del algoritmo (que corrigen tras volver a la automatización). Uno de ellos es Ignacio U, que no llega a consultar el documento de la última puerta (las numera de 1 a 7). Su segunda condición detiene la iteración cuando llega a la séptima puerta, sin llegar a consultar el documento en ella (luego cambia 7 por 8 en la siguiente versión).

```
nocedula := FALSE
contador := 1
Mientras (nocedula = FALSE) AND NOT (contador = 7)
        Analizo si en puerta contador está la cédula
        Si (no está)
                contador := contador + 1
        Sino
                nocedula := TRUE
        Fin
Fin
```

El otro estudiante (Nicolás) pretende consultar (también en una versión intermedia) el documento de la puerta siguiente a la última (puerta que no existe). Numera las puertas de 1 a 7 (como Ignacio U) e intenta consultar el documento de la octava puerta, previo a finalizar la iteración (luego cambia ≤ por < en su siguiente versión).

golpear la primera puerta
preguntar cédula
Mientras (*cédula es distinta de x*) *AND* (nro de puerta ≤ 7)
    golpear puerta siguiente
    preguntar cédula
Fin

Problemas de borde como el de Ignacio U no producirían error en tiempo de ejecución al trabajar sobre un *arreglo* en una computadora, si fueran instrucciones de un *programa*. Su solución no sería correcta en el sentido de no resolver bien el problema, pero no implicaría un error de ejecución o que haga que el programa se comporte de modo impredecible. En cambio, un problema como el de Nicolás generaría un error de *salida de rango* en la ejecución. Cuestiones como estas son inherentes a la construcción de conocimiento sobre aspectos propios de la ejecución en computadora, que se abordan en la tercera actividad (sección 3.6.6).

3.6.3 Resultados de la primera actividad

Ninguno de los estudiantes escribe una versión correcta en pseudocódigo al primer intento. Todos necesitan escribir varias versiones hasta llegar a una correcta, según los errores detectados entre cada intento y el siguiente (Ignacio D, Martín y Pablo P escriben dos, Ignacio U siete, el resto entre tres y cinco). Más allá de la cantidad de intentos de cada estudiante (que depende de factores diversos) el pensamiento de cada uno consigue iniciar las transiciones desde *newP* hacia *newC* y *newC*', si bien tienen matices en el grado de conceptualización, de modo análogo a lo que sucede en el pasaje de la etapa *intra* a la etapa *inter* en relación a las transiciones desde P hacia C y C'.

Como en la primera parte del estudio, se presentan dos tablas que resumen los grados de conceptualización alcanzados por los estudiantes al finalizar la primera actividad y un análisis global tras cada una. Cada tabla se enfoca en uno de los dos aspectos usuales (acciones o cambios) y toma en cuenta las versiones finales del algoritmo escritas por los estudiantes. No obstante, en cada análisis se hace mención a versiones intermedias cuando ello aporta al análisis. Los criterios para organizar las tablas y distribuir a los estudiantes en grupos son idénticos a los de la sección 3.3.2. Finalmente, se presenta una tercera tabla que sintetiza los aspectos inherentes al algoritmo y al formalismo intermedio (presentados en la sección anterior). Dicha tabla no agrupa a los estudiantes según su grado de conceptualización, sino según los aspectos constatados.

ACCIONES REALIZADAS		
Comparación	***Avance***	***Repetición***
Explícita Aaron, Juan, Martín, Nicolás, Pablo M, Tomás, Ximena (pseudocódigo "puro") Ignacio U, Joaquín, Pablo P (ya con elementos del lenguaje de programación)	***Explícita*** Aaron, Ignacio D, Juan, Martín, Nicolás, Pablo M (pseudocódigo "puro") Ignacio U, Joaquín, Pablo P (ya con elementos del lenguaje de programación)	***Explícita*** Aaron, Ignacio D, Ignacio U, Joaquín, Juan, Martín, Mónica, Nicolás, Pablo M, Pablo P, Tomás, Valeria, Ximena
Implícita Ignacio D, Mónica, Valeria	***Implícita*** Mónica, Tomás, Valeria, Ximena	

Tabla 3.6.3.1 – Acciones realizadas

Diez de los trece estudiantes hacen explícita la comparación. Siete en pseudocódigo "puro" (por ejemplo: *preguntar si es el número*) y tres con elementos del lenguaje de programación (por ejemplo: *x <> número*). Si bien el grado de explicitud de todos ellos es suficiente para la versión en pseudocódigo, los tres que usan elementos del lenguaje son aún más claros. Los tres estudiantes restantes dejan implícita la comparación (por ejemplo: *Si encuentro la C.I*). Con la acción de avance sucede lo mismo. Nueve estudiantes la hacen explícita. Seis en pseudocódigo (por ejemplo: *Voy a la siguiente puerta*) y tres con elementos del lenguaje (por ejemplo: *puerta := puerta + 1*). Los cuatro restantes la dejan implícita (por ejemplo: *sigo preguntando*, sin aclarar expresamente que preguntan en la puerta inmediatamente siguiente). Hay un solo estudiante (Juan, sección 3.6.1) que no la menciona en su primera versión, pero la incluye en la segunda. En cuanto a la repetición, todos la hacen explícita, pues usan una estructura de iteración (*Mientras*). De hecho, todos lo hacen ya desde la primera versión, incluidas las dos que usan *Para cada* (Mónica y Valeria). Incluso Juan (sección 3.6.1) entiende la necesidad de la repetición de acciones en su versión inicial (también usa *Mientras*) a pesar de fallar en no incluir la acción de avance en dicha versión.

El pensamiento de los estudiantes que dejan implícita la comparación o el avance está más en *newP* que para los demás estudiantes. No comprenden tan sólidamente que el robot no realiza dichas acciones en forma automática, sino que debe ser instruido. Necesitan avanzar en la transición hacia *newC*. Todos tendrán oportunidad de profundizar su grado de conceptualización en relación a estas dos acciones en la tercera actividad, al pasar del pseudocódigo al código de un *programa* que resuelva el problema (sección 3.6.6).

CAMBIOS EN OBJETOS	
Encontrar documento buscado	***Se acaban las puertas de la hilera***
Explícito Aaron, Ignacio D, Juan, Martín, Mónica, Nicolás, Pablo M, Tomás, Valeria, Ximena (pseudocódigo "puro") Ignacio U, Joaquín, Pablo P (ya con elementos del lenguaje de programación)	***Explícito*** Aaron, Ignacio D, Joaquin, Juan, Martín, Mónica, Nicolás, Pablo M, Tomás, Valeria, Ximena (pseudocódigo "puro") Ignacio U, Pablo P (ya con elementos del lenguaje de programación)

Tabla 3.6.3.2 – Cambios en objetos

Respecto a los cambios en los objetos, todos expresan ambas condiciones de la estructura *Mientras* con claridad en sus versiones finales y las combinan de manera adecuada con *AND*. Al igual que para las acciones, la mayoría lo hace en pseudocódigo "puro", mientras que solo tres lo hacen con elementos del lenguaje de programación (dos para el caso de la detención cuando se acaban las puertas). Si bien estos últimos son aún más claros, todos plasman la detención por cualquiera de las dos condiciones.

A diferencia de las acciones, en las que la no explicitud en algunos casos no es de gravedad, las condiciones deben quedar explícitas para garantizar la terminación. Aún en pseudocódigo, no explicitar ambas implica que el robot no termina o tiene un comportamiento indefinido. La ventaja de trabajar con un formalismo (aunque sea intermedio) es que obliga al estudiante a ser explícito en los cambios, ya que de otro modo el agente externo (robot en este caso) no es capaz de realizar la tarea. Esto estimula al estudiante a comprender las razones por las cuales el robot tiene éxito, y así tomar conciencia del efecto que tienen las acciones (*newC*) sobre las tarjetas para comprender los cambios (*newC'*) que conducen a la solución. Ningún estudiante expresa o combina las condiciones adecuadamente desde el inicio. La automatización resulta esencial para ayudar a conceptualizar los cambios.

Durante la actividad se observa bastante heterogeneidad en las dificultades que los estudiantes enfrentan en la conceptualización de los cambios (como se aprecia en la sección 3.6.1 y el anexo B). Por ejemplo: Ignacio D comienza tomando en cuenta ambas condiciones, pero falla al combinarlas, usando *OR*. En cambio, Pablo P considera solo una en su versión inicial, pero en su segunda versión incorpora la otra usando *AND* de manera correcta, sin siquiera pensar en probar con *OR*. Se constata que el proceso varía mucho de un estudiante a otro pero, en todos los casos, la automatización ayuda a avanzar en las transiciones desde *newP* hacia *newC* y *newC'*.

ASPECTOS INHERENTES AL ALGORITMO Y AL FORMALISMO INTERMEDIO
Cuestiones de sintaxis del formalismo intermedio
no en vez de *NOT*: Aaron, Ignacio D, Joaquín, Juan, Mónica, Pablo M, Tomás, Valeria, Ximena
Incluye elementos del lenguaje de programación: Ignacio U, Joaquín, Pablo P
Cuestiones de semántica del formalismo intermedio
Intenta primero con *OR*: Aaron, Ignacio D, Ignacio U, Juan, Valeria
Duda brevemente entre *OR* y *AND* (se decide por *AND*): Nicolás, Ximena
Usa directamente *AND*: Joaquín, Martín, Mónica, Pablo M, Pablo P, Tomás
Problemas con semántica *Mientras*: Juan, Mónica, Pablo M, Tomás, Ximena
Estrategia de búsqueda
A priori: Aaron, Joaquín, Martín, Nicolás
A posteriori: Ignacio D, Ignacio U, Juan, Mónica, Pablo M, Pablo P, Tomás, Valeria, Ximena
Salto de instancia concreta a caso general
Usa documento concreto: Joaquín, Ximena (solo en versión inicial)
Refiere a cantidad concreta de puertas: Ignacio U, Mónica, Nicolás, Pablo P
Problemas de borde
Documento en última puerta sin consultar: Ignacio U (versión no final)
Consulta documento en puerta inexistente: Nicolás (versión no final)

Tabla 3.6.3.3 – Aspectos inherentes al algoritmo y al formalismo intermedio

Al igual que con el proceso de conceptualización de los cambios en los objetos, se observa bastante heterogeneidad en relación a los aspectos inherentes al formalismo intermedio y a cuestiones del algoritmo en sí. En algunos casos, esto guarda relación directa con el proceso de conceptualización. Por ejemplo, errores vinculados a temas de semántica de operadores booleanos y de la estructura *Mientras*. En otros casos, se trata de aspectos independientes del grado de comprensión de los estudiantes, como ser la estrategia de búsqueda construida (por ellos mismos) durante la sucesión de versiones. Sin importar las dificultades relativas a la conceptualización enfrentadas durante el proceso, cada estudiante arriba por sí solo a una de las dos estrategias. De hecho, la estrategia empleada influye luego en cuestiones relativas a la ejecución del *programa* en computadora (tercera actividad, sección 3.6.6).

3.6.4 Segunda actividad: *sintaxis* y *semántica* del lenguaje formal

Esta actividad introduce por primera vez el concepto de *arreglo* y procura que cada estudiante se familiarice con el mismo y su integración con otros elementos del lenguaje de programación, trabajados en clase previo al estudio (uso de variables, expresiones y estructuras de selección e iteración). Conforme a lo expresado en la sección 3.5.5, esto implica un *doble* proceso de construcción de conocimiento. Por un

lado, sobre las reglas de *sintaxis* y *semántica* necesarias para escribir instrucciones que trabajen con dicha estructura de datos (parte *textual*) y, por el otro, sobre cómo la computadora ejecuta dichas instrucciones (parte *ejecutable*), habiendo una relación de causa y efecto entre ambos aspectos. La actividad pide escribir cuatro pequeños fragmentos de programa (cada estudiante los escribe en el lenguaje usado por su grupo) que resuelven problemas sencillos sobre un arreglo de enteros llamado `arre`, asumiendo que sus celdas fueron previamente cargadas con valores.

Al igual que en las demás actividades de la segunda parte, cada estudiante escribe *todas* las versiones necesarias. Siempre que sea oportuno, se recurre a la *automatización* y el estudiante responde preguntas entre cada versión y la siguiente. La totalidad de las versiones escritas por todos los estudiantes, junto con sus análisis correspondientes, se encuentran en el anexo C, incluyendo preguntas realizadas a cada uno y sus respuestas. Se presenta aquí una selección de algunas versiones escritas por varios estudiantes y un análisis preliminar del proceso de construcción observado y de aspectos propios del código fuente escrito en cada caso.

Fragmento 1: Sumar los valores de las primeras dos celdas y guardar el resultado en una variable entera (`suma`).

Cada estudiante necesita escribir, como mucho, dos versiones hasta llegar a una correcta. Los errores cometidos por quienes necesitan dos versiones involucran aspectos de *sintaxis* y los corrigen tras detectarlos por sí mismos o repasar la sintaxis para acceso a celdas del arreglo. Por ejemplo, Juan (lenguaje *C*) necesita escribir dos versiones. La primera es:

```
int suma = 0;
suma = arre.0 + arre.1;
```

Si bien no es un error, inicializa la variable con `0` al momento de su declaración (*C* lo permite) a pesar de no ser necesario, pues en la instrucción siguiente modifica su valor. Presenta un error de sintaxis en el acceso a celdas (utiliza punto en lugar de paréntesis rectos). Quizás hace una lectura similar a la que haría en matemática, usando punto para intentar expresar cada índice como subíndice en una sucesión ($arre_0 + arre_1$). Se le indica que usa un operador no definido por las reglas del lenguaje. Se le muestra nuevamente la tarjeta que ilustra la sintaxis de acceso a celdas y lo corrige, resultando en su segunda versión (que es correcta):

```
int suma = 0;
suma = arre[0] + arre[1];
```

Fragmento 2: Mostrar en pantalla un mensaje indicando si el valor en la celda 3 es par o impar.

Al igual que en el fragmento 1, la mayoría de los estudiantes necesita escribir, como mucho, dos versiones hasta llegar a una correcta. Un solo estudiante (Juan) escribe seis (disponibles en el anexo C) hasta lograrlo. Los errores cometidos por quienes

escriben más de una versión involucran aspectos de *sintaxis* y/o *semántica* y los corrigen tras repasar las reglas correspondientes. Por ejemplo, Ignacio U (lenguaje *Pascal*) necesita escribir dos versiones. La primera es:

```
if arre[3] = mod 2 then
    write ('es par')
else
    write ('es impar')
```

Esta versión tiene un error de sintaxis en la condición de la sentencia `if`. A la izquierda del operador `mod` debería haber un operando entero en lugar de =. Quizás el estudiante hace una lectura similar a como haría en lenguaje natural (escribe `= mod 2` para decir que *es igual a múltiplo de 2*), queriendo *generalizar* al lenguaje *formal* su conocimiento del lenguaje *natural* (la herramienta cognitiva que interviene es la generalización *inductiva* [16]). Necesita adaptar su pensamiento para escribir la expresión teniendo en cuenta que será evaluada por la *computadora* en vez de por una persona, para lo cual debe usar la sintaxis del lenguaje *formal*. Se le señala el error, se da cuenta del mismo y lo corrige, quedando una segunda versión (que es correcta):

```
if arre[3] mod 2 = 0 then
    write ('es par')
else
    write ('es impar')
```

Mónica (lenguaje *C*) también necesita escribir dos versiones. La primera es:

```
if (arre[3] % 2 = 0)
    printf ("El número es par");
else
    printf ("El número es impar");
```

Esta versión tiene un error de sintaxis en la condición. En vez de usar comparación (==), utiliza asignación (=). Es un error común que los estudiantes cometen al escribir en *C*, pues están acostumbrados a usar el símbolo = para expresar comparación en matemática (nuevamente, generalización *inductiva*). Esto generaría que el compilador intente interpretar la expresión a la izquierda de dicho operador como si fuera una variable, resultando en un error de sintaxis. Al preguntarle, toma conciencia y lo cambia, resultando en su segunda versión (que es correcta):

```
if (arre[3] % 2 == 0)
    printf ("El número es par");
else
    printf ("El número es impar");
```

Fragmento 3: Determinar si el valor en la celda 1 es o no igual al valor en la última celda y guardar el resultado en una variable booleana (`iguales`).

Para este fragmento, la cantidad de versiones escritas por los estudiantes varía bastante (entre una y seis versiones). De los que escriben más de una, algunos cometen

errores de *sintaxis* y/o de *semántica* (los que corrigen tras repasar las reglas) y todos necesitan pasar por varios intentos para despegarse de instancias *concretas* (arreglos dibujados en papel con cantidades *específicas* de celdas) y lograr expresar el índice de la *última* celda en forma *general*, en función de la constante N en lugar del número de índice concreto que identifica a la última celda. En todos los casos, logran hacerlo tras recurrir a la *automatización* y responder preguntas.

Aaron (lenguaje *Pascal*) necesita escribir dos versiones. La primera es:

```
iguales := arre[1] = arre[8];
```

Si bien es correcta, está atada a la instancia concreta manipulada (arreglo de ocho celdas, con índices 1 y 8 para referirse a la primera y última, respectivamente). Consultado acerca de cómo haría si fuese cualquier otra cantidad de celdas (por ejemplo, 20) logra el salto al caso general y escribe su segunda versión (también correcta), pero ahora en función de la constante N:

```
iguales := arre[1] = arre[N];
```

Ximena (lenguaje C) necesita escribir seis versiones. La primera es:

```
boolean iguales;
if (arre[0] == arre[7])
    boolean iguales = TRUE;
else
    boolean iguales = FALSE;
```

En esta versión, pretende declarar tres veces la variable `iguales`. La estudiante no lo sabe, pero en realidad declara tres variables diferentes con el mismo nombre, una global y otras dos locales a las cláusulas `if` y `else` (las nociones de *local* y *global* aún no han sido trabajadas en clase). Sí ha visto en clase que toda variable debe declararse *una* sola vez, pero aún muestra dificultad en su conceptualización. Se le consulta al respecto y cree que, por tener el mismo nombre, se trata de la *misma* variable en los tres casos. Se repasa con ella el concepto de unicidad en la declaración y corrige el fragmento, resultando en su segunda versión:

```
boolean iguales;
if (arre[0] == arre[7])
    iguales = TRUE;
else
    iguales = FALSE;
```

Si bien es correcta, está atada a la instancia concreta manipulada (arreglo de ocho celdas, con índices `0` y `7` para referirse a la primera y última, respectivamente). Consultada acerca de cómo haría si fuese cualquier otra cantidad de celdas, necesita pasar por tres versiones más (con valores *concretos*) antes de lograr el salto al caso general, lo cual consigue en su sexta versión (que también es correcta), pero ahora en función de la constante N:

```
boolean iguales;
if (arre[0] == arre[N-1])
    iguales = TRUE;
else
    iguales = FALSE;
```

Fragmento 4: Recorrer el arreglo de la primera a la última celda e ir mostrando por pantalla sus valores.

Para este fragmento, cada estudiante escribe, como mucho, tres versiones hasta llegar a una correcta. Los trece estudiantes usan una estructura iterativa y hacen uso de la constante N para expresar el límite superior del rango de índices de la iteración. Aún presentando errores diversos, todos lo hacen ya desde su primera versión. Si bien es la primera vez que estos estudiantes realizan una recorrida sobre un arreglo, probablemente el haber trabajado (en el fragmento anterior) con la noción de cantidad *genérica* de elementos indujo a cada estudiante a escribir el fragmento de modo que recorra el arreglo en función de N, cualquiera sea su valor.

Los errores cometidos por quienes escriben más de una versión involucran aspectos tanto de *sintaxis* y/o *semántica* como de *ejecución*. Según cada tipo de error, todos logran corregirlos tras repasar las reglas correspondientes o bien recurrir a la *automatización* y responder preguntas. Hay solo dos estudiantes (Pablo M y Tomás, cuyas versiones se pueden ver en el anexo C) que inicialmente eligen `while`, a pesar de no ser la estructura más adecuada para este problema. Todos los demás eligen `for` (ya en su primera versión) y explican bien por qué.

Ignacio U (lenguaje *Pascal*) necesita escribir dos versiones. La primera es:

```
for i := arre[1] to arre[N] do
    write (arre[i]);
```

Si bien no hay errores de sintaxis, tiene un error en relación a los límites del rango de iteración en la sentencia `for`. En vez de usar `1` y `N`, usa `arre[1]` y `arre[N]` como límites, pensando que así itera de la primera celda a la última (a lo largo de los años, el autor ha constatado que muchos estudiantes cometen este error al empezar a trabajar con arreglos). Además, podría producir un error de *salida de rango* (según los valores concretos en dichas celdas) pero no se aborda aún este tema. Quizás hace una lectura propia de una persona (*desde la celda 1 hasta la celda N*). Necesita adaptar su pensamiento para escribir los límites considerando que la *computadora* ejecutará la instrucción, debiendo iterar sobre el rango de *índices* y conceptualizar la diferencia entre *índices* y *valores* de las celdas. Se le pregunta cuánto valen `arre[1]` y `arre[N]` (en su instancia concreta) y dice "*17 y 55*". Se le indica que su iteración irá de 17 a 55 en vez de 1 a 8 y toma conciencia del error, cambiando a su segunda versión (correcta):

```
for i := 1 to N do
    write (arre[i]);
```

Juan (lenguaje *C*) también necesita escribir dos versiones. La primera es:

```
int i = 0;
FOR (i = 0; i < N-1; i++) {
    printf ("%d", ARRE[i]);
}
```

En el lenguaje *C*, la palabra `for` se escribe en minúsculas, pero el estudiante usa mayúsculas. Esto daría un error de sintaxis al compilar. El tema no se le menciona, ya que se trata luego en el pasaje del código escrito en papel a la computadora (tercera actividad, sección 3.6.6). Además, al igual que en su primer fragmento, inicializa la variable al declararla (en forma innecesaria). Si bien el estudiante parece comprender la importancia de la inicialización, quizás cree que siempre debe hacerlo junto con la declaración. Necesita conceptualizar que puede hacerse después, cuando sea necesario hacer uso de un valor válido al trabajar con la misma.

Esta versión presenta un *problema de borde*. Deja sin mostrar el valor almacenado en la última celda. El hecho de que los índices en *C* empiezan en cero, lleva en ocasiones a los estudiantes que trabajan con dicho lenguaje a expresar incorrectamente el límite superior de la iteración al recorrer las N celdas. Los estudiantes que trabajan con *Pascal* no suelen cometer este error cuando usan índices que empiezan en 1. Se recurre a la automatización sobre una instancia concreta de 8 celdas, el estudiante detecta el problema ni bien se llega a la celda con índice 7 y lo corrige, resultando en una segunda versión (correcta, a menos de `FOR` en mayúsculas):

```
int i = 0;
FOR (i = 0; i <= N-1; i++) {
    printf ("%d", ARRE[i]);
}
```

3.6.5 Resultados de la segunda actividad

Los trece estudiantes construyen conocimiento sobre las reglas de *sintaxis* y *semántica* para trabajar con un *arreglo* de enteros y su integración con otros elementos del lenguaje (parte *textual*) y también sobre aspectos de la ejecución de instrucciones que manipulan dicha estructura de datos (parte *ejecutable*). En las distintas versiones, todos logran corregir errores ya sea repasando las reglas correspondientes, o bien recurriendo a la automatización. Esta última posibilita también que logren expresar el índice de la *última* celda en forma *general* (en función de la constante N), despegándose de instancias *concretas* del arreglo. Se presentan a continuación cuatro tablas (una por cada fragmento) que sintetizan varios aspectos relativos a la conceptualización constatados durante la actividad y un análisis global luego de cada una. El criterio para agrupar a los estudiantes en cada tabla es por cantidad de intentos realizados. Cuanto más arriba está un grupo, significa que sus estudiantes cometen menos errores y necesitan escribir menos versiones hasta llegar a una correcta.

FRAGMENTO 1: sumar valores de primeras dos celdas
No cometen errores Pablo M, Pablo P, Tomás, Valeria, Ximena (*C*, una sola versión) Aaron, Ignacio D, Ignacio U, Martín, Nicolás (*Pascal*, una sola versión)
Cometen errores de sintaxis Juan, Mónica (*C*, dos versiones) Joaquín (*Pascal*, dos versiones)

Tabla 3.6.5.1 - Aspectos constatados en fragmento 1

Juan es el único que presenta un error de *sintaxis* en el acceso a las celdas y lo corrige tras repasar la sintaxis correspondiente. Mónica y Joaquín olvidan asignar el resultado a la variable `suma`, lo cual detectan y corrigen por sí mismos durante la escritura.

FRAGMENTO 2: emitir mensaje indicando si valor en celda 3 es par o impar
No cometen errores Pablo P, Tomás, Valeria, Ximena (*C*, una sola versión)
Cometen errores puramente de sintaxis Aaron, Ignacio D, Martín, Nicolás (*Pascal*, dos versiones)
Comenten errores por problemas en conceptualización de semántica Mónica, Pablo M (*C*, dos versiones) Ignacio U, Joaquín (*Pascal*, dos versiones)
Cometen los dos tipos de error anteriores Juan (*C*, seis versiones)

Tabla 3.6.5.2 - Aspectos constatados en fragmento 2

Los cuatro estudiantes que cometen errores en aspectos puramente de *sintaxis*, lo hacen en la escritura de la sentencia `if/else` necesaria para el requerimiento. Manejan mal el uso de `begin/end` y/o de punto y coma para separar instrucciones. Otros cuatro también cometen errores de sintaxis, pero como resultado de problemas al interpretar la *semántica* de la expresión booleana usada para determinar si el valor es par (algunos errores cometidos son propiamente de semántica). Los que usan *C* confunden comparación (==) con asignación (=). Los que usan *Pascal* quizás hacen una lectura de la expresión como si fuese *lenguaje natural*. En todos los casos, esto es consecuencia de *generalizar* al lenguaje *formal* elementos ya conocidos de matemática o del lenguaje natural. Juan es el único que comete errores de ambos tipos. Todos corrigen los errores tras repasar las reglas sintácticas correspondientes.

FRAGMENTO 3: determinar si valores en primera y última celda son iguales
No cometen errores y logran con facilidad salto a N celdas Pablo M, Tomás, Valeria (*C*, una o dos versiones) Ignacio D, Ignacio U, Joaquín, Martín, Nicolás (*Pascal*, una o dos versiones) Aaron (*Pascal*, dos versiones y es el único que asigna expresión booleana)
No comenten errores y necesitan más intentos para lograr salto a N celdas Mónica, Pablo P (*C*, tres o cuatro versiones)
Comenten errores y necesitan más intentos para lograr salto a N celdas Juan, Ximena (*C*, cinco o seis versiones)

Tabla 3.6.5.3 - Aspectos constatados en fragmento 3

Aaron es el único estudiante que asigna una expresión *booleana* (comparación con =) a una variable *booleana*. Los demás usan `if` o `if/else` para expresar la comparación entre las celdas y asignar expresamente `TRUE` o `FALSE` a la variable, según el resultado. Esta característica también ha sido observada por el autor repetidamente en clase. Muchos estudiantes presentan dificultad en conceptualizar que pueden asignar expresiones de tipo *boolean* a variables (si bien lo hacen sin dificultad cuando se trata de expresiones enteras o reales) y por ello recurren al uso de selección para hacerlo. Además de Aaron, otros ocho estudiantes logran con facilidad despegarse del valor concreto del índice de la última celda y expresarlo en función de N. Lo escriben directamente así o necesitan intentar, como mucho, con una única instancia concreta previo a usar la constante. Otros cuatro necesitan más de dos intentos. Los que usan *Pascal* expresan el acceso mediante `arre[N]` y los que usan *C* mediante `arre[N-1]`. Hay solo dos estudiantes (Juan y Ximena) que cometen errores. Ambos pretenden declarar más de una vez la variable booleana y lo corrigen tras repasar que cada variable debe ser declarada una única vez previo a ser usada.

FRAGMENTO 4: recorrer el arreglo y desplegar todos sus valores
No comenten errores Mónica, Pablo P, Tomás, Valeria (*C*, una sola versión) Ignacio D, Joaquín, Nicolás (*Pascal*, una sola versión)
Cometen errores puramente de sintaxis Aaron, Martín (*Pascal*, dos versiones)
Comenten errores relativos a aspectos de ejecución Juan, Ximena (*C*, dos versiones) Ignacio U (*Pascal*, dos versiones)
Cometen los dos tipos de error anteriores Pablo M (*C*, tres versiones)

Tabla 3.6.5.4 - Aspectos constatados en fragmento 4

De los siete estudiantes que no cometen errores, Tomás es el único que inicialmente elige `while` (anexo C) pero ni siquiera llega a escribir una versión que use dicha estructura. Enseguida cambia a `for` tras ser consultado y la usa correctamente. Los demás eligen `for` como primera opción y también la usan bien y explican por qué.

Los dos estudiantes que cometen errores en aspectos puramente de *sintaxis*, lo hacen en la escritura de la sentencia `for`. Manejan mal el uso de `begin/end` o del operador de asignación para inicializar la variable de control. Corrigen los errores tras repasar las reglas sintácticas correspondientes. Otros tres cometen errores relativos a aspectos de la ejecución. Dos de ellos (Juan y Ximena) presentan un *problema de borde*, dejando sin mostrar el valor en la última celda, mientras que el tercero (Ignacio U) confunde los *índices* con los *valores* almacenados en las celdas. Los tres corrigen estos errores tras recurrir a la automatización, logrando así construir conocimiento sobre dichos aspectos, que son propios de la ejecución en máquina.

Pablo M es el único estudiante que comete errores de ambos tipos a lo largo de sus versiones (disponibles en el anexo C). Inicialmente, intenta resolver el problema con `while` pero escribe una iteración que nunca termina (*loop infinito*) y comete errores de sintaxis durante la escritura. Luego cambia a `for`, pero lo hace buscando una alternativa para evitar el *loop* y no porque resulte la estructura más adecuada para el problema. Esto evidencia una conceptualización pobre, por parte del estudiante, acerca de en qué situaciones es más adecuado usar una u otra iteración (tema que fue trabajado en clase, previo al estudio). La automatización junto con el repaso de las reglas de sintaxis correspondientes, le permiten corregir los errores y llegar a una versión correcta que resuelve el problema usando `for`.

3.6.6 Tercera actividad: *programa* para búsqueda lineal

En esta actividad, los estudiantes escriben, compilan y ejecutan un *programa* que implementa la búsqueda de un valor dentro de un arreglo de enteros, partiendo de la versión en *pseudocódigo* escrita en la primera actividad. El foco está puesto en la construcción de conocimiento sobre el *programa* en sí, dado que la construcción de la *lógica* del algoritmo fue trabajada en dicha actividad. Lo escriben inicialmente en papel (*todas* las versiones necesarias) y después lo transcriben a la computadora para su compilación y ejecución (las declaraciones e instrucciones para la carga del arreglo ya se incluyen en el archivo fuente dado a cada estudiante). Nuevamente, se repasan las reglas del lenguaje formal y se recurre a la *automatización* y preguntas cuando es necesario. La totalidad de versiones escritas por todos los estudiantes y sus análisis correspondientes, se encuentran en el anexo D, incluyendo preguntas realizadas a cada uno junto con sus respuestas. Se presentan aquí extractos de algunas versiones escritas por varios estudiantes y un análisis preliminar.

Se analiza el proceso de construcción de conocimiento de cada estudiante en relación a dos aspectos. Por un lado, la correspondencia entre los *pasos* realizados sobre la hilera de puertas en pseudocódigo y las *instrucciones* ejecutadas sobre el *arreglo* que la representa (conocimiento sobre la parte *textual* del programa). Por el otro, la

comprensión de cómo la computadora las ejecuta y la búsqueda de estrategias para la corrección de errores en la ejecución (conocimiento sobre la parte *ejecutable*). También se pide al estudiante que escriba (directamente en la computadora) una instrucción para emitir un mensaje en pantalla indicando el resultado de la búsqueda (sin darle mayor indicación que esa). Interesa analizar cómo elige hacerlo y su vínculo con la razón por la cual detuvo la búsqueda (a nivel *textual* y *ejecutable*).

El proceso de conceptualización de las *acciones* (comparación, avance y repetición) realizadas por el robot imaginario y cómo ellas imponen *cambios* a los objetos (dados por las condiciones de parada) comenzó en la primera actividad. Todos los estudiantes lograron expresar en forma explícita repetición y cambios, pero no así comparación y avance (hubo cinco estudiantes que dejaron implícita al menos una de dichas acciones, sección 3.6.3). El pasaje del pseudocódigo al texto del programa permitirá a todos terminar de expresar todas ellas de manera explícita. El uso del lenguaje *formal* exige que así sea, pues posee reglas estrictas de sintaxis y semántica. Además, por tratarse de un lenguaje de *programación*, la compilación chequea su cumplimiento y la ejecución permite visualizar el comportamiento que las instrucciones generan. Esto ayuda a que el pensamiento del estudiante consolide la relación entre acciones y cambios. El *agente externo* deja de ser el robot imaginario, capaz de interpretar pasos que poseen cierto grado de ambigüedad, y pasa a ser la computadora, que no admite ningún grado de ambigüedad.

A continuación, se presenta una selección de algunas versiones del programa escritas por varios estudiantes, junto con un análisis preliminar del proceso de construcción observado. Se presentan en tres grupos. El primero analiza el pasaje del algoritmo en pseudocódigo al texto del programa (conocimiento sobre la parte *textual*). El segundo ilustra los errores de ejecución más comunes detectados en el proceso y analiza las estrategias empleadas para su corrección (conocimiento sobre la parte *ejecutable*). El tercero muestra las tres formas elegidas por los estudiantes para decidir qué mensaje emitir y su vínculo con la razón por la cual detienen la búsqueda. Las versiones presentadas en los tres grupos no necesariamente son todas finales.

Pasaje de pseudocódigo a programa

Todos los estudiantes construyen conocimiento sobre la parte *textual* del programa. Cada uno necesita escribir, como mucho, tres versiones del programa hasta llegar a una correcta. Todos mantienen la estrategia usada en pseudocódigo (*a priori* o *a posteriori*) a lo largo de sus versiones. En su versión final, los que siguen *a priori* nunca asignan a la variable usada para marcar el índice de cada celda (durante la iteración) un valor que exceda el rango de índices válidos del arreglo. Como mucho, queda con el índice de la última celda, tanto si el valor buscado existe en el arreglo, como si no. Los que siguen *a posteriori* asignan a dicha variable un valor más del último índice válido, pero sin acceder a una celda fuera de rango.

Por otra parte, todos los estudiantes, excepto dos, respetan la lógica de su algoritmo en pseudocódigo. Mantienen el orden tanto en las instrucciones que representan los pasos en pseudocódigo como en las condiciones de la estructura *Mientras* (la cual traducen a `while`). Incluso los dos que no respetan totalmente la lógica de su última versión en pseudocódigo (Pablo M y Valeria) se mantienen bastante acordes a ella. Las desviaciones que cometen se muestran entre los extractos que se presentan más adelante. Por otro lado, todos los estudiantes completan el salto al caso *general*, en el sentido de que su programa busca un valor *cualquiera* en un arreglo de *cualquier* tamaño (dado por la constante N de la declaración del arreglo).

Tres estudiantes escriben una versión correcta y acorde al algoritmo en pseudocódigo en un único intento, sin necesitar ninguna corrección. Por ejemplo, Aaron lo hace, como se muestra en el siguiente extracto (lenguaje *Pascal* y estrategia *a priori*).

```
Puerta := 1;
cedula := arre[1];
While Not (cedula = ci) AND (Puerta < N) DO
Begin
    Puerta := Puerta + 1;
    cedula := arre[Puerta];
end
```

La versión que escribe se corresponde en forma acorde con su pseudocódigo:

Ver la C.I de la puerta 1
Mientras NOT (*sea la que busco*) *AND* (*tengo mas Puertas*)
    Veo siguiente C.I
Fin

Mantiene el orden de las condiciones y su combinación mediante *AND*, así como el orden de las instrucciones. Expresa la condición *NOT* (*sea la que busco*) mediante `Not (cedula = ci)` y *tengo mas Puertas* mediante `(Puerta < N)`. Necesita utilizar dos instrucciones para expresar el paso *Ver la C.I de la puerta 1*, una para posicionarse en el índice de la celda (`Puerta := 1`) y otra para obtener su contenido (`cedula := arre[1]`). Lo mismo para *Veo siguiente C.I*.

Otros ocho estudiantes también mantienen la lógica de su algoritmo en pseudocódigo, pero necesitan escribir más de una versión (siete escriben dos y una escribe tres). Los que escriben dos versiones cometen pocos errores en la primera (que pueden ser de *sintaxis*, *semántica* o *ejecución*) y los corrigen en la segunda (repasando las reglas correspondientes o recurriendo a la automatización) la cual compila y se ejecuta correctamente. Con la estudiante que escribe tres versiones (Ximena, lenguaje *C* y estrategia *a posteriori*) pasa lo mismo, solo que necesita un intento más para llegar a una versión correcta. La primera versión de Ximena es la siguiente:

```
int ced;
boolean c = FALSE
int i = 0
while (i =< n-1) && (c == TRUE)
{
    if (arre[i] == ced) {
        c = TRUE;
         }
    else
          i++;
}
```

En esta versión intenta ser acorde a su última versión en pseudocódigo, pero tiene cinco errores. Cuatro son de sintaxis y uno es de comportamiento en la ejecución. Los de sintaxis son que omite poner punto y coma tras las dos primeras inicializaciones, usa =< en vez de <=, usa n minúscula en vez de N mayúscula para referirse a la constante (el lenguaje *C* es *case sensitive*) y omite poner paréntesis rodeando ambas condiciones en la estructura `while` (*C* lo exige). El de comportamiento es porque compara (`c == TRUE`) en la segunda condición, generando que no se itera ni siquiera una vez. La estudiante probablemente piensa en la condición para *detener* la iteración en vez de para *seguir* iterando, lo cual muestra que quizás aún persista alguna dificultad en su conceptualización de la semántica de la estructura `while`. El código anterior asume que la variable `ced` fue previamente inicializada con el valor a buscar (al transcribir a la computadora, se inicializa con un valor leído por teclado).

Mientras hallan puertas AND no encuentre la cédula que busco hacer
    abrir puerta
    Si cédula es la que busco
        terminé de buscar
    Sino
        sigo recorriendo puertas
    Fin
Fin

Mantiene el orden de las condiciones y su combinación mediante *AND* (usa `&&` en *C*) así como el orden de las instrucciones. Expresa la condición *hallan puertas* mediante (`i =< n-1`) y *no encuentre la cédula que busco* mediante (`c == TRUE`) (la cual genera el error). Unifica en una sola condición (`arre[i] == ced`) dos pasos separados de su pseudocódigo (*abrir puerta* y *Si cédula es la que busco*). Tras la automatización, detecta y corrige el error de comportamiento, cambiando por (`c == FALSE`) en su segunda versión (todavía en papel). Luego transcribe a la computadora y al compilar se detectan los errores de sintaxis. Los corrige tras repasar las reglas correspondientes, resultando en su tercera versión, que compila y se ejecuta correctamente.

Por último, los dos estudiantes que se desvían de la lógica de su pseudocódigo necesitan escribir tres versiones del programa (siendo la última correcta). La versión final que escriben es correcta en el sentido de que igualmente resuelve la búsqueda. En sus

primeras dos versiones cometen errores (como los ocho anteriores) además de apartarse de su lógica inicial. Como ejemplo, se muestra la segunda versión de Pablo M (lenguaje *C*, estrategia *a posteriori*):

```
int a = 0; boolean igual;
while (a <= N-1) && (cedula != arre[a]) {
    if (arre[a] == cedula) {
        igual = True
                    }
    else {
        igual = False
         }
        a++
      }
```

Pablo M tuvo dificultad en comprender la consigna planteada en esta actividad. En su primera versión (disponible en el anexo D), en vez de resolver lo solicitado (buscar un valor en el arreglo), lo busca en una secuencia de valores ingresados progresivamente por teclado (sin usar el arreglo). Tras repasar la consigna, escribe la segunda versión que aquí se muestra y es en ella donde altera la ubicación de una instrucción, apartándose así de la lógica de su algoritmo en pseudocódigo.

Mientras (*haya puertas en la cuadra*) *AND* (*no encuentre a fulanito*)
    golpeo la puerta
    pregunto por fulanito (*cédula*)
    Si (*fulanito está en la puerta*)
        Termino la búsqueda
    Sino
        Voy a la siguiente puerta
    Fin
Fin

En su pseudocódigo, avanza a la siguiente puerta solo cuando el documento buscado no está en la puerta actual (el paso *Voy a la siguiente puerta* se ubica dentro de la cláusula *Sino*), mientras que en su programa avanza *siempre* a la siguiente celda (coloca `a++` por *fuera* de la instrucción `if/else`) sin importar si el valor buscado está o no en la celda actual (en su primera versión estaba *dentro* del bloque `else`, acorde a su pseudocódigo). Excepto por el movimiento de `a++`, el resto es bastante acorde al pseudocódigo. Mantiene el orden de las condiciones y su combinación con *AND*. Unifica en una sola condición (`arre[a] == cedula`) tres pasos separados del pseudocódigo (*golpeo la puerta*, *pregunto por fulanito (cédula)* y *Si* (*fulanito está en la puerta*)). Además, presenta dos errores de sintaxis: faltan varios puntos y comas y un juego de paréntesis que rodee toda la condición de la sentencia `while` (*C* lo exige). Dichos errores son detectados tras transcribirlo a la computadora y compilar. Los corrige (manteniendo el resto del código igual), resultando en su tercera versión, que resuelve la búsqueda y se ejecuta correctamente.

El estudiante asigna repetidamente `false` a la variable `igual` en forma innecesaria a medida que avanza (podría haberla inicializado una vez antes de empezar la recorrida) y no la usa en la segunda condición para cortar la iteración (luego de asignarle `true`) ni la consulta luego para emitir el resultado. Esto le fue señalado al estudiante, pero manifestó no saber cómo modificarlo. Si bien no son errores, esto evidencia una conceptualización pobre acerca del uso (o no) de la variable booleana para cortar la iteración. Su solución es un híbrido entre una que hace uso de dicha variable (como la de Ximena) y una que no lo hace (como la de Aaron). De todas formas, construye conocimiento sobre el programa, pues llega a una solución, pero lo hace de forma menos sólida que otros estudiantes. Resuelve la búsqueda, pero se aparta un poco de la lógica del algoritmo que ideó en pseudocódigo y hace un manejo confuso de la variable booleana, que no aporta a la solución. Su comprensión del vínculo entre la lógica del algoritmo y el programa es menos marcada que para los demás estudiantes.

Errores de ejecución y estrategias para su corrección

Todos los estudiantes construyen conocimiento sobre la parte *ejecutable* del programa, tanto sobre la resolución de la búsqueda en sí como sobre la emisión del mensaje para indicar el resultado. A lo largo de sus versiones, seis estudiantes detectan errores de *ejecución*, los cuales corrigen tras recurrir a la automatización. Son de ejecución en el sentido de que la cortan de manera abrupta o bien producen un comportamiento inadecuado. Los errores detectados son por *salida de rango*, *problemas de borde* o no iterar ni siquiera una vez (este último es cometido solo por una estudiante, Ximena, ver más arriba en esta misma sección). Dependiendo de la naturaleza del error y el lugar del programa donde ocurre, cada estudiante aplica una estrategia diferente para su corrección. Se muestran a continuación tres extractos a modo de ejemplo, dos por *salida de rango* y uno por *problema de borde*.

Martín (lenguaje *Pascal*, estrategia *a priori*) escribe dos versiones en total. En la primera, comete un error de *salida de rango*, como se muestra a continuación:

```
i := 1;
ci := arre[1];
while (cedula <> arre[i]) and (i ≤ n) do
begin
    i := i+1;
    ci := arre[i];
end
```

Si el valor buscado no está en el arreglo, tras consultar la última celda, la instrucción `i := i+1` genera que la variable `i` tome el valor `N+1`, quedando fuera del rango de índices válidos. Al acceder a `arre[i]` en la siguiente instrucción (`ci := arre[i]`), se produce el error. Tras recurrir a la automatización, decide corregirlo cambiando ≤ por <, resultando en su segunda versión (correcta). Esta corrección resulta adecuada para su estrategia *a priori*. Se asegura de que `i` siempre quede con el valor del índice de la última celda al finalizar la iteración (tanto si el valor buscado está en el arreglo, como si no), evitando así la salida de rango.

Valeria (lenguaje *C*, estrategia a *posteriori*) escribe tres versiones en total. En la primera, comete un error de *salida de rango*, como se muestra a continuación (pone `If` y `Else` con la primera letra en mayúscula, más tarde las cambia a minúsculas en computadora):

```
int CI, i = 0;
boolean encontreCI = false;
while (arre[i] != CI && i < N) {
    If (arre[i] == CI) {
        encontreCI = true;
                        }
    Else {
          i++;
        }
}
```

El código anterior asume que la variable `CI` fue previamente inicializada con el valor a buscar (al transcribir a la computadora, se inicializa con un valor leído por teclado). Si el valor buscado no está en el arreglo, tras consultar la última celda, la instrucción `i++` genera que la variable `i` tome el valor `N`, quedando fuera del rango de índices válidos. Al acceder luego a `arre[i]` en la primera condición de la estructura `while`, se produce el error. El código que escribe es acorde a su algoritmo en pseudocódigo, en el cual expresa: *Mientras* (*no encuentre nro C.I AND no pregunté en última puerta*). Mantiene el orden de las condiciones en pseudocódigo (así como el orden de sus instrucciones). En términos del *formalismo intermedio* esto es correcto, dado que el operador *AND* es conmutativo, siendo irrelevante el orden de las condiciones. La estudiante debe conceptualizar que la ejecución es realizada por la computadora y tomar en cuenta cómo la máquina evalúa expresiones booleanas.

El compilador usado por la estudiante genera un programa ejecutable que evalúa `&&` por *circuito corto*. Esto es, primero evalúa la expresión de la izquierda y, según el resultado, evalúa la de la derecha solo si es necesario. El orden ahora pasa a ser relevante, por lo cual `&&` deja de ser conmutativo. La expresión `(arre[i] != CI)` se ubica a la izquierda de `&&`, por lo cual se evalúa antes que `(i < N)`, produciendo el error. Tras recurrir a la automatización y notar el error, la estudiante pregunta si puede asumir que no se evalúa la segunda y se le dice que sí (la distinción entre *circuito corto* y *circuito completo* había sido mencionada en clase previo al estudio). Decide corregir el error invirtiendo el orden de las condiciones, quedando una segunda versión con `while (i < N && arre[i] != CI)` y manteniendo todo el resto igual. Asumiendo evaluación por *circuito corto*, esta corrección resulta adecuada para su estrategia *a posteriori*. Ahora `(i < N)` se evalúa primero, dando `FALSE` como resultado y dejando sin evaluar `(arre[i] != CI)`, evitando así la salida de rango.

Al igual que el programa de Pablo M (ver más arriba en esta sección), el de Valeria deja de ser totalmente acorde a su algoritmo en pseudocódigo, en este caso debido a la inversión en el orden de las condiciones. Por lo demás, se mantiene acorde al mismo.

A diferencia de Pablo M, Valeria introduce su cambio de manera *consciente* para corregir un error surgido de un aspecto propio de la ejecución en máquina y construye conocimiento formal en el proceso. Entiende la necesidad del cambio, por lo que su comprensión del vínculo entre el programa y la lógica del algoritmo no se ve afectada.

Tras la corrección anterior, la propia Valeria constata un *problema de borde* al ejecutarse la instrucción que escribe luego de la iteración para emitir el resultado (la escribe directo en la computadora y el problema es constatado al ejecutar en máquina):

```
if (encontreci == TRUE) {
    printf("Se encontro ci"); }
else {
    printf("No se encontro cI");
     }
```

Elige usar la variable booleana para decidir qué mensaje emitir. Si bien la búsqueda funciona correctamente, en el sentido de que para cuando corresponde y ya no tiene el error de salida de rango, se produce un *problema de borde* cuando el número buscado está en la *última* celda, emitiendo un mensaje incorrecto en ese caso. La iteración se detiene al evaluar la segunda condición `(arre[i] != CI)`, sin llegar a ejecutarse la asignación `encontreCI = true`. La estudiante lo constata al ejecutar el programa para buscar un valor que está en la última celda (previamente constata que funciona bien al buscar un valor que está en una celda anterior a la última y otro que no se encuentra en el arreglo). Tras recurrir nuevamente a la automatización, decide corregirlo cambiando la segunda condición de `while`, resultando en su tercera versión: `while (i < N && encontreci == FALSE)` y todo el resto igual. Con este cambio, siempre asigna `TRUE` a `encontreci` cuando encuentra el valor, sin importar en qué celda se encuentre. Su segunda versión dejaba `FALSE` en dicha variable si lo encontraba en la última.

Emisión de mensaje indicando el resultado

Cada estudiante transcribe su programa a computadora para compilar y ejecutarlo. Luego de la iteración que resuelve la búsqueda, escribe (directamente en máquina) una instrucción para emitir un mensaje indicando el resultado. No se le da ninguna indicación de cómo hacerlo. Se busca analizar qué variable decide consultar para determinar si tuvo éxito en hallar el valor y emitir un mensaje acorde. Esto se vincula a la construcción de conocimiento sobre el concepto de *invariante* de la iteración. Si bien no se incluyó expresamente como parte del estudio, se quiere recabar información sobre cómo lo abordan los estudiantes. Once de ellos logran hacerlo bien al primer intento. Otro lo hace en dos intentos y hay uno que lo logra en tres. Los que necesitan más de un intento cometen algún error que (como en el pasaje del pseudocódigo al programa) puede ser de *sintaxis*, *semántica* o *ejecución* y lo corrigen tras repasar las reglas correspondientes o recurrir nuevamente a la automatización.

Cinco estudiantes eligen consultar una variable entera usada durante la búsqueda para almacenar el valor de la celda actual (en cada entrada a la iteración) y compararlo con el valor buscado. Por ejemplo, Joaquín (lenguaje *Pascal*), quien además lo hace bien al primer intento.

```
if x = numero then
    writeln ('La cedula fue encontrada')
else
    writeln ('La cedula no fue encontrada');
```

Durante la recorrida, guarda en la variable `x` el valor de cada celda. Al terminar la iteración, `x` queda con el valor de la última celda visitada, por lo tanto siempre puede comparar ese valor con el buscado (dado en la variable `numero`) para emitir el mensaje correspondiente.

Otros seis estudiantes eligen trabajar con una variable booleana y usarla para cortar la iteración (en caso de encontrar el valor) y luego determinar el mensaje a emitir. La inicializan en `false` antes de la recorrida y le asignan `true` dentro de la misma si encuentran el valor. Por ejemplo, Ignacio U (lenguaje *Pascal*), quien también lo hace bien al primer intento.

```
if nocedula = true then
    writeln('la cedula esta')
else
    writeln('la cedula no esta');
```

Por último, hay solo dos estudiantes que eligen usar el valor del índice para determinar qué mensaje emitir como resultado. Durante la recorrida, usan una variable para indicar el índice de la celda actual. Ambos estudiantes siguen la estrategia *a posteriori* y su programa deja dicho índice con un valor más del último índice válido cuando el valor buscado no está en el arreglo. Por ejemplo, Pablo M (lenguaje *C*), quien también lo hace bien al primer intento.

```
if (a > N-1) {
    printf ("Fulanito no esta en ninguna de las puertas");
}
else {
    printf ("Fulanito esta en alguna de las puertas");
}
```

Se trata del mismo estudiante (ver más arriba en esta misma sección) que inicialmente intentó trabajar con una variable booleana, pero luego terminó sin usarla. Su variable `a` queda con el valor del índice siguiente a la última celda únicamente en caso de que el valor buscado no esté en el arreglo, por lo tanto siempre puede usar su valor para determinar si estaba o no.

3.6.7 Aspectos inherentes al *programa* y al *lenguaje formal*

En la sección anterior se presentaron diversos ejemplos de versiones del programa escritas por los estudiantes y un análisis preliminar del proceso de construcción de conocimiento sobre dos aspectos (parte *textual* y parte *ejecutable*). En esta sección se presentan más ejemplos, pero ahora se analizan desde otra perspectiva, tomando en cuenta otros aspectos constatados en el estudio. Concretamente, algunas cuestiones vinculadas a la implementación en el *programa* de la estrategia de búsqueda definida en pseudocódigo (*a priori* o *a posteriori*) además de otras relativas al uso que hacen los estudiantes del *lenguaje de programación* con el que trabajan (*C* o *Pascal*), no necesariamente vinculadas al proceso de conceptualización. Los ejemplos se presentan nuevamente en grupos (son tres: *programas que siguen estrategia a priori, programas que siguen estrategia a posteriori y uso del lenguaje de programación*).

Programas que siguen estrategia *a priori*

Si bien no es necesario, los cuatro estudiantes que siguen la estrategia *a priori* eligen usar una variable auxiliar durante la búsqueda para almacenar el valor de la celda actual (en cada entrada a la iteración) y usarla luego en la instrucción para determinar qué mensaje emitir indicando el resultado. Además, ninguno de ellos utiliza estructura de selección (`if` ni `if/else`) dentro de la iteración. Como ejemplo, se muestra la versión final de Nicolás (lenguaje *Pascal*), quien utiliza una variable `cedula` para ir almacenando el valor de cada celda. Se incluye también la instrucción que coloca luego de la iteración para emitir el mensaje con el resultado de la búsqueda:

```
cedula := arre[1];
i := 1;
WHILE (cedula <> x) AND (i < N) DO
BEGIN
    cedula := arre[i+1];
    i := i+1
END;
IF cedula = x THEN
    writeLn('La cedula fue encontrada')
ELSE
    writeLn('La cedula no fue encontrada');
```

Por otra parte, en la estrategia *a priori* resulta irrelevante si la evaluación del operador `AND` en la condición de la estructura `while` se realiza por *circuito corto* o por *circuito completo*. Incluso en el caso de que se acceda a `arre[i]` en alguna de las condiciones, no hay riesgo de *salida de rango*, dado que `i` queda, como mucho, con el valor del último índice válido. De los cuatro estudiantes que usan esta estrategia, solamente Martín (ver sección anterior) lo hace (tuvo error de salida de rango y lo corrigió asegurándose de que el valor de `i` no exceda el último índice válido). Los otros tres usan la variable auxiliar (definida para almacenar el valor actual) en alguna de las condiciones. A pesar de no ser necesario, el uso de dicha variable quizás se explica por una intención, por parte de los estudiantes, de reducir el riesgo de salirse de rango.

Entienden que si almacenan en ella el valor de la celda actual (dentro de la iteración), no tienen riesgo de acceder a una celda con índice inválido en ninguna de las condiciones de la estructura `while`. No son conscientes de que la estrategia empleada ya de por sí garantiza no salirse de rango cuando la variable correspondiente al índice se maneja bien.

Programas que siguen estrategia *a posteriori*

Ocho de los nueve estudiantes que siguen *a posteriori* no usan una variable auxiliar para almacenar el valor de la celda actual y utilizan una estructura de selección (`if` o `if/else`) dentro de la iteración. Como ejemplo, se muestra la versión final de Mónica (lenguaje *C*) incluyendo además su instrucción para emitir el mensaje con el resultado:

```
boolean encontre = FALSE;
int i = 0;
while ((!encontre) && (i ≤ N-1))
{
    if (arre[i] == CI)
        encontre = TRUE;
    else
        i++;
}
if (encontre == TRUE)
    printf ("Encontré la CI");
else
    printf ("No encontré la CI");
```

En la estrategia *a posteriori* puede ser relevante si la evaluación del operador `&&` se realiza por *circuito corto* o por *circuito completo*. Dado que `i` finaliza con un valor más del último índice válido (cuando el valor buscado no existe), no es posible acceder a `arre[i]` en la condición a la izquierda del operador. La única que inicialmente lo hace es Valeria (ver sección anterior) y lo corrige invirtiendo el orden de las condiciones, aprovechando la evaluación por *circuito corto*. Los demás acceden a `arre[i]` en la de la derecha o bien no lo hacen y trabajan con una variable booleana (que asignan dentro de la iteración). Para emitir el resultado, seis de los ocho usan una variable booleana para decidir qué mensaje emitir (como Mónica), mientras que los dos restantes usan el índice (como Pablo M, ver sección anterior). Al igual que los que siguen *a priori*, el uso de la variable auxiliar (en este caso, booleana) no es necesario en la estrategia *a posteriori* cuando la evaluación es por *circuito corto* y quizás también responde a una intención para reducir el riesgo de salirse de rango.

El noveno estudiante que sigue la estrategia *a posteriori* (Ignacio D, lenguaje *Pascal*) constituye la única excepción a todo lo anterior. No usa `if` ni `if/else` y tampoco trabaja con una variable booleana. Además, utiliza una variable auxiliar (`cedula`) para tomar el valor de la celda actual en cada entrada a la iteración (como los cuatro que siguen la estrategia *a priori*). Su versión final, incluyendo la instrucción para emitir el mensaje con el resultado, es la siguiente:

```
cedula := arre[1];
puerta := 1;
while (cedula <> cedulaAbuscar) and (puerta ≤ N) do
Begin
    cedula := arre[puerta];
    puerta := puerta + 1;
End
if (cedula = cedulaAbuscar) then
    write ('se encontró la persona')
else
    write ('no se encontró la persona');
```

Su estrategia es *a posteriori* en el sentido de que `puerta` queda con el valor del índice siguiente a la última celda cuando el valor buscado no está. Sin embargo, la estructura de su código se asemeja más a la estrategia *a priori*. De hecho, obtiene el valor de la primera celda antes de iniciar la iteración, aunque vuelve a obtenerlo dentro (en la primera entrada), como los demás que emplean *a posteriori*. Para mostrar el resultado, inicialmente usó el índice, pero tuvo un *problema de borde* (ver anexo D) si el valor estaba en la última celda. Para corregirlo, decidió emplear la variable `cedula`. Si bien su pseudocódigo (se muestra a continuación) define una estrategia *a posteriori*, quizás el pensamiento del estudiante cambió (de manera inconsciente) cuando hacía el pasaje al programa, resultando en una versión que combina elementos de ambas.

Mientras (*no haya encontrado a la persona*) *AND* (*no se hayan acabado las puertas*)
    Pregunto en la puerta que estoy quien vive
    Avanzo a la siguiente puerta
Fin

Aspectos relativos al uso del *lenguaje de programación*

En primer lugar, se constata que hay estudiantes que *generalizan* al lenguaje de programación (tanto entre quienes escriben en *C* como en *Pascal*) elementos conocidos del lenguaje natural o de lenguaje matemático, al igual que en la segunda actividad. Si bien ocurre con menos frecuencia, igualmente persiste en la tercera. Como sucede en cualquier proceso de construcción de conocimiento, el que se construye sobre el uso del lenguaje *formal* es gradual y dialéctico, lo cual explica que siga pasando. Por ejemplo, Juan (lenguaje *C*) escribe `while ((encontre = FALSE) && (i ≤ N-1))` en su primera versión, confundiendo asignación (=) con comparación (==). Joaquín (lenguaje *Pascal*) escribe `WHILE (x <> NUMERO) AND ARRE[i] <> ARRE[N]` en su primera versión, confundiendo los índices de las celdas con sus valores. Todos corrigen estos errores en sus segundas versiones del programa. También se constata que varios estudiantes siguen teniendo dificultad en conceptualizar que una variable booleana puede ser usada por sí misma como una expresión. Por ejemplo, Ignacio U (lenguaje *Pascal*) pone `if nocedula = true` en vez de `if nocedula`, o Ximena (lenguaje *C*) pone `if (c == TRUE)` en vez de `if (c)`.

En segundo lugar, se constatan algunas diferencias bastante marcadas entre los que usan *Pascal* y los que usan *C*. De los seis que usan *Pascal*, cuatro siguen *a priori*, mientras que solamente dos siguen *a posteriori*. De esos dos, Ignacio D (ver más arriba en esta sección) escribe un programa cuya estructura se asemeja más a *a priori*. Cinco de los seis no utilizan estructura de selección (`if` ni `if/else`) dentro de la búsqueda y tampoco usan variable booleana (solo el sexto lo hace, Ignacio U). Esos mismos cinco usan una variable auxiliar de tipo entero para ir almacenando el valor de la celda actual en la recorrida. Por otra parte, los siete que trabajan con *C* siguen *a posteriori*. Todos usan `if` o `if/else` dentro de la búsqueda, cinco usan variable booleana (seis, si se cuenta a Pablo M, quien intentó hacerlo y terminó sin darle uso real) y ninguno usa variable auxiliar de tipo entero.

Las diferencias anteriores resultan llamativas, pues las dos estrategias se pueden implementar en ambos lenguajes, los cuales permiten trabajar con los mismos elementos (variables enteras, booleanas, estructuras de selección y de iteración). Además, la construcción de la estrategia de búsqueda surgió en forma *previa* a la escritura del programa en el lenguaje formal. Ocurrió al definir la lógica del algoritmo durante la escritura en *pseudocódigo*, cuando los trece estudiantes emplearon el *mismo* formalismo intermedio. En la próxima sección se propone una posible explicación para ello, pero sería oportuno realizar más estudios a futuro para profundizar en el tema.

3.6.8 Resultados de la tercera actividad

Sin considerar la emisión del mensaje con el resultado, tres estudiantes logran escribir una versión correcta del programa en un solo intento, siete en dos y tres en tres. En cuanto a la emisión del mensaje, diez estudiantes lo logran en un intento, dos en dos y uno en tres. Los que necesitan más de un intento cometen pocos errores que corrigen sin mayor dificultad. El pasaje del *pseudocódigo* al *programa* parece resultar más sencillo para los estudiantes que el pasaje de la descripción en *lenguaje natural* a *pseudocódigo*, lo cual requirió la escritura de más versiones para la mayoría de los estudiantes (de dos a siete). El pensamiento de todos había iniciado las transiciones desde *newP* hacia *newC* y *newC*' con la escritura del algoritmo en pseudocódigo y profundiza en ellas con la escritura del programa, su compilación y ejecución.

Una primera lectura de esto es que la construcción de la *lógica* del algoritmo parece resultar más compleja para los estudiantes que su *formalización*. El foco al escribir el programa ya no está en la conceptualización de las *acciones* del agente externo (comparación, avance y repetición) y su vínculo con la imposición de *cambios* a los objetos (detener la búsqueda al encontrar un valor igual al buscado o cuando la cantidad de valores que resta consultar llega a cero), sino en la correspondencia entre el *formalismo intermedio* y el *lenguaje formal*, tanto a nivel *textual* (vínculo entre *pasos* del pseudocódigo e *instrucciones* del programa) como a nivel *ejecutable* (efecto que las *instrucciones* tienen sobre las *estructuras de datos* cuando el agente externo es la computadora). La conceptualización de esta correspondencia parece más simple para los estudiantes que la conceptualización del vínculo entre acciones y cambios.

Para la primera actividad (sección 3.6.3) se presentaron dos tablas (una para *acciones* y otra para *cambios*). Ahora también se presentan dos, pero con otro enfoque. La primera se centra en la construcción de conocimiento sobre la parte *textual* (escritura del *programa* a partir del *pseudocódigo*, sin la emisión del mensaje) y la segunda se enfoca en la construcción sobre la parte *ejecutable* (errores de *ejecución* y estrategias de corrección). A diferencia de las tablas de la sección 3.6.3, que agrupan estudiantes según el grado de conceptualización alcanzado al *finalizar* la actividad, ahora se agrupan según el proceso de construcción observado *durante* la actividad. La razón es que el nivel de conocimiento alcanzado al finalizar es equivalente para todos los estudiantes, con excepción de uno (Pablo M, cuyo caso se analiza en esta sección).

Excepto por él, al terminar el proceso todos construyen conocimiento a la par sobre el programa y su vínculo con el algoritmo en pseudocódigo. El análisis ahora se centra en el desarrollo del proceso más que en la culminación. El criterio de agrupación en cada tabla ahora es por cantidad de intentos realizados. Cuanto más arriba está un grupo, significa que sus estudiantes necesitan escribir menos versiones para llegar a una solución correcta. Junto a cada tabla, se incluye un análisis global relativo al aspecto correspondiente. Posteriormente, se presentan otras dos tablas. Una sintetiza las tres formas empleadas por los estudiantes para decidir qué mensaje emitir como resultado y la otra resume los aspectos inherentes al programa y al lenguaje formal (presentados en la sección anterior). Estas tablas no agrupan a los estudiantes según su cantidad de intentos, sino según los aspectos constatados.

***PASAJE DE PSEUDOCÓDIGO A PROGRAMA* (parte *textual*)**
No cometen errores y mantienen lógica algoritmo
Aaron, Ignacio D (*Pascal*, una versión)
Pablo P (*C*, una versión)
Pocos errores y mantienen lógica algoritmo o la cambian deliberadamente
Ignacio U (*Pascal*, cambia `8` por `N+1`, dos versiones)
Joaquín (*Pascal*, `begin`, nombre variable, confunde índice c/celda, dos versiones)
Juan (*C*, cambia = por ==, dos versiones)
Martín (*Pascal*, cambia <= por <, dos versiones)
Mónica (*C*, cambia `7` por `N-1`, dos versiones)
Nicolás (*Pascal*, falta inicializar variable, dos versiones)
Tomás (*C*, cambia <= por <, dos versiones)
Valeria (*C*, cambia orden condiciones, modifica segunda condición, tres versiones)
Ximena (*C*, errores de sintaxis, `c == TRUE` en vez `c == FALSE`, tres versiones)
Pocos errores y cambian inadvertidamente lógica algoritmo
Pablo M (*C*, cambia lugar `a++`, manejo confuso variable booleana, tres versiones)

Tabla 3.6.8.1 – Pasaje de pseudocódigo a programa

Tres estudiantes escriben el programa correctamente al primer intento y mantienen la lógica de su última versión en pseudocódigo. En cuanto a los nueve del segundo grupo, cometen pocos errores y los corrigen con facilidad. La única estudiante de este grupo que modifica la lógica de su algoritmo en pseudocódigo es Valeria, pero lo hace de forma consciente para corregir un error. A efectos de su conceptualización sobre la parte *textual* del programa, está a la par del resto en el sentido de que corrige sin dificultad el texto y comprende por qué.

Los errores cometidos pueden ser de *sintaxis*, *semántica* o *ejecución*. En términos del pensamiento del estudiante, la naturaleza y/o efectos de cada error pueden no ser tan evidentes. Lo que se analiza es la habilidad en corregir el *texto* para eliminar el error. Por ejemplo, en su primera versión, Nicolás omite inicializar la variable `i` que usa para iterar, siendo ese su *único* error. Técnicamente es grave, porque generaría un problema de *ejecución* que podría manifestarse como *salida de rango*, según qué valor por defecto tome `i` en el programa ejecutable. Sin embargo, no necesita constatar la salida de rango para tomar conciencia. Lo detecta y corrige al instante, sin tener que recurrir a la automatización hasta que se manifieste el problema. Más allá de las implicancias de cada error, estos nueve los detectan y corrigen con facilidad y construyen conocimiento sobre la parte *textual*, así como los tres del primer grupo.

El único que construye conocimiento en menor grado es Pablo M. Su confusión inicial respecto de la consigna, el cambio de lugar de `a++` y su manejo confuso de la variable booleana evidencian un grado de conceptualización sobre el texto de su programa no tan marcado. El grado de transición de su pensamiento hacia *newC* no es tan profundo, pues no comprende el significado de todas sus instrucciones. Asimismo, ocurre con la transición hacia *newC*'. No termina de comprender que hay instrucciones en su código que no generan cambios (las que involucran el uso de la variable booleana). Tampoco comprende, con la misma profundidad que el resto, la relación entre la lógica de su algoritmo en pseudocódigo y el programa que lo implementa. Su pseudocódigo expresa su intención de no continuar cuando encuentra a la persona, al igual que su descripción en lenguaje natural. En cambio, su programa avanza a la siguiente celda incluso luego de encontrar el valor buscado. En términos de la extensión a la ley general de la cognición, su comprensión del vínculo entre el conocimiento *conceptual* sobre el algoritmo y el conocimiento *formal* sobre el programa (relación dialéctica entre la primera y la segunda línea del diagrama) es menos sólida que para el resto.

***ERRORES DE EJECUCIÓN y ESTRATEGIAS CORRECCIÓN* (parte *ejecutable*)**
No cometen errores que requieran visualizar cómo afectan la ejecución
Aaron (*Pascal*, una versión búsqueda, una mensaje)
Ignacio U (*Pascal*, dos versiones búsqueda, una mensaje)
Joaquín (*Pascal*, dos versiones búsqueda, una mensaje)
Juan (*C*, dos versiones búsqueda, una mensaje)
Mónica (*C*, dos versiones búsqueda, una mensaje)
Nicolás (*Pascal*, dos versiones búsqueda, una mensaje)
Pablo M (*C*, tres versiones búsqueda, una mensaje)
Cometen errores que afectan la ejecución y los corrigen tras automatización
Ignacio D (*Pascal*, problema borde mensaje, una versión búsqueda, tres mensaje)
Martín (*Pascal*, salida rango búsqueda, dos versiones búsqueda, una mensaje)
Pablo P (*C*, salida rango mensaje, una versión búsqueda, dos mensaje)
Tomás (*C*, salida rango búsqueda, dos versiones búsqueda, una mensaje)
Valeria (*C*, salida rango búsqueda, problema borde mensaje, tres versiones búsqueda, una mensaje)
Ximena (*C*, no itera ni una vez búsqueda, tres versiones búsqueda, dos mensaje)

Tabla 3.6.8.2 – Errores de ejecución y estrategias de corrección

Hay siete estudiantes que no cometen errores de ejecución, o bien los cometen pero los corrigen sin necesitar recurrir a la automatización para visualizar cómo afectan la ejecución. Por ejemplo, en su primera versión, Mónica escribe `(i ≤ 7)` como segunda condición de su estructura `while`, manteniendo la cantidad de 7 puertas de su versión en pseudocódigo. Técnicamente, produciría un error de *salida de rango* cuando el valor de la constante `N` es inferior a `8`. Sin embargo, no necesita recurrir a la automatización. Cambia por `N-1` sin necesitar volver a interactuar con arreglos de varios tamaños para lograr saltar al caso general. Los restantes seis cometen algún error que requiere volver a la automatización para constatar el efecto que genera en la ejecución y, en base a ello, idear una estrategia para su corrección. Construyen conocimiento sobre la parte *ejecutable* del programa, más allá de haber cometido algún error en el proceso.

En cuanto a Pablo M, no comete ningún error que afecte la ejecución (de hecho, ya su primera versión realiza una búsqueda sin errores de ejecución, salvo que no busca en un arreglo, sino en una secuencia de valores leídos por teclado). En este aspecto, está a la par de los otros en el sentido de que comprende cómo funciona la ejecución de las instrucciones de su programa que aportan a resolver la búsqueda pero no comprende del todo por qué las restantes no lo hacen (las que involucran la variable booleana). Por ello, su conocimiento sobre la parte ejecutable es menos sólido que para el resto.

EMISIÓN DE MENSAJE CON RESULTADO
Usan variable entera con último valor visitado para decidir mensaje a emitir Aaron, Joaquín, Martín, Nicolás (*Pascal*, *a priori*) Ignacio D (*Pascal*, *a posteriori* con estructura símil *a priori*)
Usan variable booleana que indica si encontró para decidir mensaje a emitir Ignacio U (*Pascal*, *a posteriori*) Juan, Mónica, Tomás, Valeria, Ximena (*C*, *a posteriori*)
Usan índice que indica si excedió rango para decidir mensaje a emitir Pablo M, Pablo P (*C*, *a posteriori*)

Tabla 3.6.8.3 – Emisión de mensaje con resultado

En sus versiones finales del programa, los cuatro estudiantes que siguen la estrategia *a priori* eligen consultar la variable entera con el valor de la última celda visitada en la búsqueda. Ello es adecuado para la estrategia empleada, pues es invariante que contiene el valor buscado (si existe en el arreglo) y que no lo contiene (si no). De los nueve que siguen *a posteriori*, los seis que utilizan variable booleana eligen consultarla, dado que es invariante que vale `TRUE` o `FALSE`, según el valor exista o no (no se cuenta a Pablo M, pues no le da uso real). Los dos que usan el valor del índice al finalizar la recorrida lo hacen porque es invariante que está dentro del rango de índices válidos (si el valor buscado existe) o que no lo está (sino). En ambos casos, la elección es adecuada para la estrategia *a posteriori*.

El noveno estudiante que sigue *a posteriori*, Ignacio D, es el único que no consulta una variable típicamente acorde a la estrategia de búsqueda empleada. En su primera versión, usa el índice (`if (puerta > N)`), lo cual suele ser adecuado para la estrategia *a posteriori*. Sin embargo, tiene un *problema de borde* (mensaje incorrecto cuando el valor buscado está en la *última* celda). Tras recurrir a la automatización, cambia al mismo criterio que los cuatro que siguen *a priori*. Como se vio en la sección anterior, la estructura de su programa se ajusta más a dicha estrategia, por lo cual su cambio de criterio resulta adecuado para corregir el problema.

ASPECTOS INHERENTES AL PROGRAMA Y AL LENGUAJE FORMAL
Programas que siguen estrategia a priori
No usan `if` ni `if/else` en búsqueda: Aaron, Joaquín, Martín, Nicolás (*Pascal*)
Usan variable entera mensaje: Aaron, Joaquín, Martín, Nicolás (*Pascal*)
No importa *circuito corto* o *completo*: Aaron, Joaquín, Martín, Nicolás (*Pascal*)
Programas que siguen estrategia a posteriori
Usan `if` o `if/else` en búsqueda: Ignacio U (*Pascal*), Juan, Mónica, Pablo M, Pablo P, Tomás, Valeria, Ximena (*C*)
No usan `if` ni `if/else` en búsqueda: Ignacio D (*Pascal*, estructura símil *a priori*)
Usan variable entera mensaje: Ignacio D (*Pascal*, estructura símil *a priori*)
Usan variable booleana mensaje: Ignacio U (*Pascal*), Juan, Mónica, Tomás, Valeria, Ximena (*C*)
Usan índice mensaje: Pablo M, Pablo P (*C*)
No importa *circuito corto* o *completo*: Ignacio D, Ignacio U (*Pascal*), Juan, Mónica, Tomás, Ximena (*C*)
Sí importa *circuito corto* o *completo*: Pablo M, Pablo P, Valeria (*C*)

Tabla 3.6.8.4 – Aspectos inherentes al programa y al lenguaje formal

Esta tabla sintetiza las diferencias detectadas entre los estudiantes que trabajan con *C* y los que trabajan con *Pascal*. La mayoría de los que usan *Pascal* emplean la estrategia *a priori* (o similar, si se cuenta a Ignacio D) sin uso de selección ni variable booleana, mientras que la mayoría de los que usan *C* emplean la estrategia *a posteriori* con uso de selección y variable booleana. Esto resulta llamativo, pues puede hacerse de ambas formas en ambos lenguajes y además la definición de la estrategia y elección de estructuras surgieron previamente, durante la escritura en *pseudocódigo*. Todos trabajaron con las *mismas* reglas de sintaxis y semántica del *formalismo intermedio*. Además, como se menciona en la sección 3.6.2, la elección de la estrategia ni siquiera fue pensada en el diseño del estudio, sino que el arribo de cada estudiante hacia una de las dos se constató en su realización. Surge la pregunta de si el conocimiento *previo* del lenguaje *formal* podría influir en el pensamiento del estudiante al construir conocimiento sobre la *lógica* de un nuevo algoritmo, *antes* de su formalización.

Al iniciar el estudio, cada estudiante conocía solamente el lenguaje usado en su grupo y ninguno había trabajado con *arreglos*. Todos habían trabajado con los mismos elementos (variables, tipos de datos básicos, expresiones e instrucciones simples, estructuras de *selección* e *iteración*). Para buscar una explicación a lo constatado, el autor de este trabajo revisó los temarios de ambos grupos y detectó una única diferencia que puede tener relación. Los del grupo de *Pascal* trabajan con el compilador Free Pascal [70] y los de *C* con Code::Blocks para *C*/*C*++ [66]. En el primero, se había hecho mucho hincapié (previo al estudio) sobre la diferencia entre evaluación por *circuito corto* o *circuito completo* y se estimulaba a evaluar por *circuito corto*, pues así se hace por defecto en Free Pascal. En el segundo grupo, si bien la diferencia era conocida, no se hacía igual hincapié en el tema, aun cuando Code::Blocks también

evalúa por defecto usando *circuito corto*. Por lo tanto, los estudiantes de *C* quizás asumen (inconscientemente) *circuito completo*, tendiendo a usar variables booleanas (y por extensión usan `if` o `if/else`). Los de *Pascal* quizás tienen más presente *circuito corto*, evitando recurrir a variables booleanas y al uso de `if` o `if/else`.

Según afirma Cellérier [3], el conocimiento que los sujetos construyen sobre distintos dominios se integra a sus esquemas mentales y luego participa en la construcción de nuevos conceptos. En el marco del estudio, si bien los estudiantes definen la lógica del algoritmo en la etapa de *pseudocódigo*, antes del estudio habían conocido la evaluación por *circuito corto* o *circuito completo*. Por más que hubieran trabajado con dichas formas de evaluación en el lenguaje *formal*, la manera en que funcionan se integró previamente a su esquema mental. Por ello, las emplearon (a nivel inconsciente) al definir la lógica en *pseudocódigo*, resultando en que los que usan *Pascal* tendieron a no usar selección ni variables booleanas y los que usan *C* sí lo hicieron. Aun así, persisten otras interrogantes. Por ejemplo, dado que los estudiantes del grupo de *Pascal* parecían tener más presente la evaluación por *circuito corto*, por qué no hubo más estudiantes en dicho grupo que escribieran un programa *a posteriori* sin usar selección ni variable booleana. Podría hacerse perfectamente e incluso sería más esperable entre quienes tienden a evaluar por *circuito corto*. Sería oportuno hacer más estudios a futuro para indagar en estas cuestiones.

3.6.9 Cuarta actividad: programa para *nuevo* problema

En esta actividad, cada estudiante escribe, *directamente* en la computadora, otro programa que resuelve un *nuevo* problema: *listar todos los valores pares almacenados en el arreglo* (se proporciona nuevamente cargado con valores). Se busca analizar cómo lo hace sin pasar por *pseudocódigo* y explorar la relación entre el conocimiento *formal* construido en el problema previo y el nuevo problema. Nuevamente, se pide escribir *todas* las versiones necesarias, compilar y ejecutar. Se vuelven a repasar las reglas del lenguaje y se recurre a *automatización* y preguntas, de ser necesario. La totalidad de versiones escritas por todos los estudiantes junto con sus análisis se encuentran en el anexo E, incluyendo preguntas realizadas a cada uno y sus respuestas. Se presenta aquí una selección de versiones de varios estudiantes y un análisis preliminar del proceso de construcción. Las versiones presentadas no necesariamente son todas finales.

Por la naturaleza del nuevo problema, la solución más adecuada consiste en una combinación de la estructura de iteración `for` con selección (`if` sin `else`). Se trata de un problema de *filtrado* en vez de *búsqueda*, por lo que la estructura `while` (usada en el problema anterior) deja de ser la más apropiada. La herramienta cognitiva que interviene en el proceso de construcción de conocimiento sobre un nuevo problema es la *generalización* [16]. En particular, interesa ver si los estudiantes intentan emplear alguna variante del *mismo* algoritmo del problema anterior para resolver el nuevo problema (generalización *inductiva*), a pesar de que tiene una naturaleza *diferente* y analizar de qué manera logran adaptar el conocimiento ya construido para construir una solución adecuada para el nuevo (generalización *constructiva*).

Todos los estudiantes necesitan escribir, como mucho, cuatro versiones del programa hasta llegar a una correcta. Seis de ellos lo hacen en un único intento, sin necesitar ninguna corrección. Por ejemplo, Nicolás (lenguaje *Pascal*) lo hace, como muestra el siguiente extracto:

```
FOR i := 1 TO N DO
    IF (arre[i] MOD 2 = 0) THEN
        writeLn(arre[i])
```

Consultado acerca de la elección de estructuras de control para este nuevo problema, responde correctamente por qué elige cada una (tanto `for` como `if`). No se producen errores de compilación y constata que funciona correctamente al ejecutarlo. Logra construir una solución para el nuevo problema, elige estructuras de control adecuadas y comprende por qué funciona correctamente.

Otros seis estudiantes escriben dos versiones. En la primera no hay error pero necesitan realizar algún ajuste, o bien cometen un solo error (de *sintaxis*, *semántica* o *ejecución*), que corrigen en la segunda. Por ejemplo, Pablo P (lenguaje *C*) tiene un *problema de borde* en su primera versión:

```
for (i=0; i < (N-1); i++)
{
    if (arre[i]%2 == 0)
    {
        printf("\n%d", arre[i]);
    }
}
```

Deja sin consultar el valor almacenado en la última celda. Al iterar sobre un arreglo en *C*, hay estudiantes que expresan en forma incorrecta el límite superior de la iteración (se confunden porque los índices empiezan en cero). Tras recurrir a la automatización, detecta y corrige el problema (cambia < por <= en su segunda versión). También explica correctamente por qué elige las estructuras de control y comprende por qué el programa funciona.

Por último, hay un solo estudiante que necesita escribir cuatro versiones. Dicho estudiante (Pablo M, lenguaje *C*) es el único que presenta dificultad con el nuevo problema. En su primera versión, elige `for` (lo cual es correcto) pero no `if`, por lo que despliega *todos* los valores del arreglo. Lo constata tras ejecutarlo en máquina y se le pregunta cómo podría hacer para mostrar solamente los que son pares, tras lo cual escribe una segunda versión, que se muestra a continuación:

```
for (a=0; arre[a]%2 == 0; a++) {
     printf("\n%d", arre[a]);
}
```

Esta versión tampoco resuelve el problema. Muestra solamente los valores pares entre la celda cero y la primera celda con un valor impar (sin incluirla). Además, podría producir error de *salida de rango* si no hay ningún impar. El estudiante no es

consciente de que su código se comporta como si usara una estructura `while` en lugar de `for`. Si bien *C* permite usar `for` de esta manera, en clase (previo al estudio) se desestimó su uso en casos así. Su solución se asemeja más a una *búsqueda lineal* que carece de una segunda condición para controlar la detención cuando llega al final del rango de índices válidos del arreglo. Durante el curso, se hizo hincapié en el uso de la estructura de control más adecuada para cada problema. Si se quisiera buscar el primer valor impar, sería más adecuado usar `while` (como en la tercera actividad).

El estudiante tiene dificultad en conceptualizar la semántica de `for`. Quizás piensa que la condición `arre[a] % 2 == 0` es para *filtrar* los valores pares, sin ser consciente de que *detiene* la recorrida cuando deja de cumplirse. Se recurre a la automatización y constata el problema al llegar a una celda con un valor impar. Se revisa junto a él la sintaxis en *C* de cada estructura (tanto de selección como de iteración) y también se repasa cuándo resulta adecuado usar cada una. Se le pregunta qué tendría que verificar sobre cada valor a medida que recorre, tras lo cual cae en cuenta de la necesidad de incluir selección y escribe una tercera versión:

```
for (a=0; a <= N-1; a++) {
    if(arre[a]%2 == 0) {
     printf("\n%d", arre[a]);
}
```

Esta versión ahora resuelve el problema pero tiene un error de sintaxis (hay dos llaves que abren bloques, pero solo una que cierra). Tras compilar y analizar el mensaje de error, agrega otra llave que cierra, quedando su cuarta versión (que es correcta). Constata que compila y funciona correctamente en máquina y finalmente logra explicar por qué usa no solamente iteración por subrango, sino también selección.

3.6.10 Resultados de la cuarta actividad

Doce estudiantes eligen correctamente `for` e `if` (sin `else`) ya en su primera versión. Excepto por algún error o ajuste (que corrigen con facilidad), todos resuelven el problema correctamente. Incluso los que tienen error expresan, ya en su primera versión, la condición de la estructura `if` para controlar si el valor en cada celda es par. Como mucho, hay algún error (de *sintaxis*, *semántica* o *ejecución*) pero controlan la paridad. El estudiante restante (Pablo M) inicialmente solo elige `for` e incorpora `if` después. Se presenta a continuación una tabla que resume el proceso constatado en la actividad y un análisis global tras la misma. Como en la sección 3.6.8, el análisis se centra en el desarrollo del proceso más que en su culminación. El criterio para agrupar estudiantes en la tabla es nuevamente por cantidad de intentos realizados.

PROGRAMA PARA NUEVO PROBLEMA
No cometen errores
Aaron, Ignacio U, Nicolás (*Pascal*, una versión)
Juan, Tomás, Valeria (*C*, una versión)
Cometen un solo error o bien necesitan hacer un ajuste
Joaquín (*Pascal*, muestra texto 'arre[i]' en vez de valor `arre[i]`, dos versiones)
Ignacio D (*Pascal*, pone = en vez de `:=`, dos versiones)
Martín (*Pascal*, faltó `end` en `for`, dos versiones)
Mónica (*C*, pone = en vez de `==`, dos versiones)
Pablo P (*C*, problema de borde, dos versiones)
Ximena (*C*, agrega tabulador para separar valores desplegados, dos versiones)
Cometen varios errores y presentan dificultad con el nuevo problema
Pablo M (*C*, muestra todos, maneja mal `for` y no usa `if`, falta `}`, cuatro versiones)

Tabla 3.6.10.1 – Programa para nuevo problema

Los doce estudiantes que integran los primeros dos grupos construyen conocimiento sobre el nuevo problema en forma sólida, tanto a nivel *textual* como *ejecutable*. Más allá de corregir un error o hacer algún ajuste, todos eligen las estructuras de control adecuadas para resolverlo y explican por qué lo hacen. Comprenden la diferencia entre el problema de *filtrado* y el problema de *búsqueda* resuelto anteriormente. En cuanto a Pablo M, también logra construir conocimiento, pero se consolida al finalizar el proceso. Sus dificultades previas con la conceptualización de la semántica de `for` le impiden hacerlo en menos intentos. No obstante, al terminar comprende (casi a la par del resto) cómo y por qué su solución funciona, dado que logra realizar los cambios necesarios y explicar finalmente por qué tiene éxito en su solución.

Al enfrentarse a un nuevo problema, todo sujeto habitualmente intenta (de manera inconsciente) aplicar conocimiento previo para resolverlo, muchas veces sin adaptarlo a la nueva situación. Esto se explica en términos de la herramienta cognitiva *generalización inductiva* [16]. Cuando la aplicación directa del conocimiento previo no resuelve el nuevo problema adecuadamente, comienza (en el plano del pensamiento) un proceso de adaptación de dicho conocimiento para resolver la nueva situación. El esquema mental se equilibra y se construye así nuevo conocimiento (generalización *constructiva* [16]). Debido a esto, inicialmente se pensó que algunos estudiantes quizás intentarían utilizar `while` para resolver el nuevo problema, tras usarlo previamente para resolver la búsqueda lineal. Sin embargo, ningún estudiante lo hace. Todos eligen `for` como primera opción (incluso Pablo M, si bien presenta dificultad en su uso).

Se proponen dos posibles explicaciones para lo anterior (no excluyentes entre sí). La primera es que la naturaleza del problema de *filtrado* quizás resulta más simple para los estudiantes que la del problema de *búsqueda*. La experiencia del autor como docente del curso da cuenta de ello. Durante años, ha constatado en clase que muchos

problemas que requieren el uso de `for` para su resolución suelen resultar más fáciles para los estudiantes y luego intentan usarlo en nuevos problemas para los cuales `while` resulta más adecuado. Quizás, si el diseño del estudio hubiese sido al revés (proponiendo primero el problema de *filtrado* y luego el de *búsqueda*), se habría constatado esto último. Se requieren nuevos estudios a futuro para indagar en ello. La segunda explicación se basa en Cellérier [3] (como en la sección 3.6.8). El hecho de que todos los estudiantes usen `for` quizás se deba a que antes, en la segunda actividad (sección 3.6.4), se les pidió un fragmento de programa para listar *todos* los valores del arreglo. La mayoría usó `for` para resolverlo, construyendo conocimiento sobre el uso de `for` como la estructura adecuada para recorrer todo el arreglo e integrándolo entonces a su esquema mental. Enfrentados luego al nuevo problema en esta actividad, aplican dicho conocimiento previo y logran adaptarlo para sumarle el uso de `if`.

Los trece estudiantes que participan del estudio empiezan a construir conocimiento sobre la estructura de datos *arreglo* a partir de su introducción en la segunda actividad. Tras completar la tercera y la cuarta, construyen conocimiento sobre dos *programas* que manipulan dicha estructura. En este último, lo hacen aplicando y adaptando conocimiento previamente construido, sin necesidad de resolver instancias concretas del problema a nivel *instrumental*, dar una descripción en lenguaje natural como evidencia de construcción de conocimiento *conceptual*, ni pasar nuevamente por el uso de pseudocódigo como *formalismo intermedio*. Logran construir una solución al nuevo problema trabajando *directamente* en el plano *formal* (además, casi todos lo hacen con facilidad). Esto permite extraer algunas conclusiones preliminares que sientan las bases para futuros estudios, las cuales se presentan en la próxima sección.

3.7 Conclusiones de la segunda parte (*inter* → *trans*)

Anteriormente, en la primera parte del estudio, se constató que el grado de construcción de conocimiento *conceptual* en relación al algoritmo para la *búsqueda lineal* varió de un estudiante a otro. No todos lograron una sólida conceptualización de las acciones realizadas ni de los cambios que imponen a los objetos. Posteriormente, en la segunda parte, la introducción del pseudocódigo como *formalismo intermedio* contribuye a consolidar la conceptualización, a la vez que ayuda a iniciar el pasaje a la etapa *trans*, siendo la primera herramienta utilizada para instruir a un *agente externo* (robot imaginario) para realizar las acciones e imponer los cambios. Esto marca el inicio de un nuevo proceso de construcción de conocimiento regulado por la extensión a la ley general de la cognición. El conocimiento *conceptual* construido previamente sitúa el pensamiento de cada estudiante en una nueva periferia (*newP*) y a partir de allí, construye nuevo conocimiento sobre el agente externo como ejecutor de las acciones (inicio de transición hacia *newC*) que imponen cambios, los cuales conducen al éxito en la solución (inicio de transición hacia *newC*').

Tras completar la escritura del algoritmo en pseudocódigo, se constata que el grado de conceptualización de los estudiantes se vuelve más homogéneo. A diferencia de lo que sucede con las descripciones en lenguaje natural en la primera parte, ahora todos

logran expresar los cambios de manera explícita y comprenden por qué los mismos permiten al agente externo resolver el problema. En cuanto a las acciones, todos explicitan repetición y solo unos pocos dejan implícitas comparación y avance. El propio formalismo intermedio introduce la necesidad de hacer más explícitas acciones y cambios, pues de otro modo el agente externo no es capaz de realizar la tarea. Además, se comprueba que la *automatización* efectivamente resulta útil para visualizar la ejecución de las acciones expresadas en pseudocódigo. Ayuda a comprender el efecto que tienen sobre los objetos, facilita la reflexión y permite detectar y corregir errores. Este mecanismo, combinado con el *formalismo intermedio*, contribuye a que el pensamiento de cada estudiante avance en las transiciones hacia *newC* y *newC*'.

La automatización prueba ser de utilidad no solo durante la escritura en pseudocódigo, sino también en la escritura del *programa* en el *lenguaje formal* (lenguaje de programación). La ejecución de las instrucciones sobre representaciones de objetos *formales* del lenguaje (arreglo dibujado en papel) por el nuevo agente externo (robot imaginario que simula ser la computadora) permite a cada estudiante visualizar el efecto de las instrucciones sobre la estructura de datos y corregir errores, de igual forma que para el pseudocódigo. Además, lo ayuda a terminar de despegarse de instancias concretas y lograr expresar el programa en forma general, de modo que funcione para buscar *cualquier* valor en un arreglo de *cualquier* tamaño.

Todos los estudiantes (con excepción de Pablo M) profundizan en la conceptualización de la relación entre acciones y cambios al escribir el *programa*. El lenguaje de programación exige que ambos queden totalmente explícitos (las acciones por medio de las *instrucciones* y los cambios por medio de las *expresiones booleanas* usadas en la estructura `while`). Al pasar del pseudocódigo al texto del programa, todos logran establecer una correspondencia entre los pasos del pseudocódigo y las instrucciones, construyendo en el proceso conocimiento sobre la parte *textual* del mismo y sobre el vínculo entre la lógica del algoritmo y el programa. Asimismo, la comprensión (ayudada por la automatización) de los efectos generados por las instrucciones sobre la estructura de datos posibilita que construyan conocimiento sobre la parte *ejecutable*. Al comprender la relación dialéctica entre ambas partes, el pensamiento profundiza en el avance hacia *newC* y *newC*' iniciado con la escritura del algoritmo en pseudocódigo. En cuanto a Pablo M, su grado de construcción de conocimiento sobre ambas partes es menos marcado que para el resto. No consigue completar la correspondencia entre su pseudocódigo y el programa, al desviarse un poco de la lógica inicial de su algoritmo y dejar instrucciones en el texto que no aportan a la solución del problema. Tampoco comprende del todo el comportamiento de las mismas en la ejecución. No obstante, igualmente construye conocimiento en el proceso.

Por otro lado, los resultados de la segunda parte muestran que la construcción de la *lógica* del algoritmo resulta más compleja que su posterior expresión en el *lenguaje formal*. El inicio del proceso de transformación de conocimiento *conceptual* en conocimiento *formal* resulta más dificultoso para los estudiantes dado que, en ese punto, su pensamiento necesita reestructurarse para pasar de conocer el algoritmo (a

nivel *conceptual*) a expresarlo por primera vez para instruir al agente externo. Contar con el formalismo intermedio y el mecanismo de automatización como facilitadores los ayuda en el proceso, de forma gradual y dialéctica, hasta que lo consiguen. Una vez construida la lógica del algoritmo, expresada en el pseudocódigo, el pasaje al lenguaje formal resulta más simple. Excepto por Pablo M, todos los estudiantes consiguen hacerlo con bastante poco esfuerzo.

En paralelo con el análisis del proceso de construcción de conocimiento, se detecta que cada estudiante construye la lógica de su algoritmo siguiendo una de dos estrategias posibles para su solución, denominadas *a priori* y *a posteriori*. La construcción de la estrategia se da durante la escritura en pseudocódigo e inicialmente no parece haber razón evidente por la cual cada estudiante emplea una u otra. Luego, en el pasaje al programa, cada uno mantiene su estrategia, siendo Ignacio D el único que cambia "sobre la marcha", escribiendo inicialmente una solución *a posteriori* y arribando a una versión formal que parece más *a priori*. En los distintos programas se observan marcadas diferencias entre quienes usan *C* y quienes usan *Pascal*, no solamente en términos de la estrategia, sino también de los elementos del lenguaje que emplean en sus soluciones (uso o no de variable booleana y de selección dentro de la iteración).

Al diseñar el estudio, se previó analizar si la diferencia en el lenguaje de programación aporta o no algún elemento relevante para la formalización. Del análisis de los resultados de la tercera actividad se concluye que no parece ser la diferencia en el lenguaje lo que explica las diferencias constatadas, sino el conocimiento previo de los estudiantes sobre la evaluación de expresiones booleanas por *circuito corto* o *circuito completo*. A raíz de la diferencia en el énfasis hecho en cada grupo (previo al estudio) sobre ambas formas de evaluación, los del grupo de *Pascal* quizás tenían más presente la diferencia entre ellas que los del grupo de *C*, explicando así por qué difieren los elementos del lenguaje de programación usados por los estudiantes de cada grupo. Además, la forma que cada estudiante elige para determinar por qué razón finalizó la búsqueda al mostrar el mensaje con el resultado puede estar también relacionada con esto. Por último, según lo planteado por Cellérier [3], se interpreta que el conocimiento previo sobre ambas formas de evaluación influye también en la elección de la estrategia durante la etapa de escritura del pseudocódigo, lo cual explicaría por qué cada estudiante arriba a una u otra al idear la lógica del algoritmo, previo a escribir el texto del programa. Se concluye que es necesario realizar más estudios a futuro para arrojar luz sobre estas cuestiones.

En relación al programa para el nuevo problema (*filtrado*), contrariamente a lo que inicialmente se pensó que podría pasar, todos los estudiantes lo escriben sin intentar aplicar una variante al problema original (*búsqueda*). Además, todos lo hacen casi sin dificultad (salvo Pablo M), tras completar el programa para la búsqueda. Se interpreta que ello puede deberse a que el nuevo problema es de naturaleza más simple que el original y a que uno de los fragmentos pedidos en la segunda actividad puede haber contribuido al conocimiento previo de los estudiantes, facilitando así su adaptación al nuevo problema (en línea nuevamente con lo planteado por Cellérier). Al margen de

esto, la actividad permite concluir que cuando el nuevo problema guarda razonables similitudes y diferencias con otros problemas ya conocidos, es posible construir conocimiento trabajando directamente en el plano *formal*, sin transitar por un proceso completo desde el plano *instrumental* y pasando por el *conceptual*. Esto abre la puerta para investigar la construcción de conocimiento relativa a *familias* de algoritmos, transitando el proceso para algunos algoritmos de una familia dada (de la etapa *inter* a la etapa *trans*) y usando el conocimiento construido sobre ellos para construir nuevo conocimiento formal sobre otros algoritmos de la misma familia.

Como conclusión final, se constata una vez más que toda construcción de conocimiento implica un proceso complejo que rara vez es lineal. La teoría de Piaget ya lo afirma [14, 17] para la construcción de conocimiento en general y en este estudio se observa específicamente para la construcción de conocimiento sobre algoritmos, estructuras de datos y programas. Una sólida conceptualización del algoritmo al pasar de la etapa *intra* a la etapa *inter*, no asegura una construcción de conocimiento igualmente sólida al pasar de *inter* a *trans*. Estudiantes que exhiben un grado de conceptualización pobre al inicio pueden ganar solidez posteriormente, y viceversa. Esto se constata, por ejemplo, en Aaron y Pablo M. En la primera parte, Aaron mostró un grado bajo de comprensión del algoritmo a nivel *conceptual*. Sin embargo, logra consolidar dicho conocimiento en la primera actividad de la segunda parte y construye conocimiento *formal* con facilidad en las siguientes actividades. Pablo M, en cambio, mostró una sólida comprensión del algoritmo a nivel *conceptual* (de hecho, en la primera parte, su descripción en lenguaje natural fue de las más completas) pero tiene más dificultades que el resto en el pasaje al conocimiento *formal*, tanto al escribir su programa para la búsqueda lineal, como al construir la solución para el nuevo problema.

Conforme a lo explicado por Perales en [55], esto último puede explicarse en términos de lo que Piaget denomina un "*decalage*" vertical, refiriéndose a que en ocasiones hay individuos que, enfrentados a la comprensión de conocimiento científico nuevo en el plano *formal*, revierten a estadios cognitivos inferiores antes de lograr operar plenamente en el plano formal, produciéndose un desfasaje entre la estructura cognitiva previa del sujeto y el nuevo conocimiento formal al que se enfrenta. El caso de Pablo M abre la puerta a trabajos futuros para investigar este aspecto en el campo del conocimiento sobre programación y lograr así enriquecer el modelo para brindar explicaciones más precisas de todo el proceso de construcción de conocimiento.

Capítulo 4

Conclusiones y trabajos futuros: Síntesis, expansión y aportes del modelo

El modelo presentado en este trabajo constituye una síntesis de muchos años de investigaciones acerca de construcción de conocimiento sobre *algoritmos*, *estructuras de datos* y *programas*, en el marco de la teoría epistemológica de Jean Piaget. Los primeros estudios se centraron en la construcción de conocimiento *conceptual* sobre algoritmos y estructuras de datos, a partir del conocimiento *instrumental*. Este último, a su vez, es construido por los sujetos a partir de su interacción con el medio, al resolver tareas o problemas aplicando métodos asimilables a instancias concretas de algoritmos. Dichos estudios se basaron en principios de la teoría para explicar el proceso de construcción de conocimiento *conceptual* (*ley general de la cognición*, *abstracción* y *generalización*). Las siguientes investigaciones incursionaron en el estudio de la construcción de conocimiento sobre *programas*, los cuales utilizan lenguajes de programación como *formalismos*, por lo que fue necesario tomar otros principios de la teoría para explicar la construcción de conocimiento *formal*. Esto llevó a iniciar una reinterpretación, de todo el trabajo realizado, en términos de la tríada de etapas *intra-inter-trans*. La construcción de conocimiento *conceptual* en los sujetos, a partir del *instrumental*, se reinterpreta como el pasaje de la etapa *intra* a la etapa *inter*, en tanto la construcción de conocimiento *formal*, a partir del *conceptual*, como el pasaje de la etapa *inter* a la etapa *trans*. Las investigaciones sobre este último pasaje llevaron a la necesidad de formular la *extensión* a ley general de la cognición, estableciéndose así el contexto a partir del cual se desarrolló el presente trabajo, cuyo principal objetivo fue estudiar en profundidad este último pasaje.

Para ello, se llevó a cabo el estudio empírico presentado en el capítulo 3, que permitió explicar en detalle dicho pasaje para el problema elegido: la *búsqueda lineal*. Se aplicó la extensión a la ley general de la cognición por primera vez desde su formulación para explicar la construcción de conocimiento sobre el *programa* que resuelve dicho problema. Posibilitó consolidar la reinterpretación en términos de la tríada en un modelo de investigación capaz de explicar el proceso *completo* de construcción de conocimiento sobre algoritmos, estructuras de datos y programas, partiendo del conocimiento *instrumental* (etapa *intra*), pasando por el *conceptual* (etapa *inter*) y llegando al *formal* (etapa *trans*). En la sección 4.1 se presentan las conclusiones del trabajo y en la sección 4.2 se proponen algunas líneas de trabajo futuro.

4.1 Conclusiones

En esta sección se presentan algunas conclusiones de todo el trabajo, tanto del estudio empírico realizado (sección 4.1.1), como otras relativas al modelo de investigación en general y sus aportes a CSE como una disciplina específica dentro de CS (sección 4.1.2).

4.1.1 Surgidas del estudio empírico

En el capítulo 3 se presentaron, por separado, conclusiones específicas para cada parte del estudio (primera parte: pasaje de *intra* a *inter* y segunda parte: pasaje de *inter* a *trans*). En esta sección se presenta un resumen de todo el estudio y se extraen algunas conclusiones globales del mismo y su vínculo con el modelo.

El estudio tuvo como objetivo observar el proceso *completo* de construcción de conocimiento transitado por cada estudiante en relación al problema de búsqueda lineal. El proceso comienza con la aplicación, en el plano de la acción, de una instancia concreta de un algoritmo que lo resuelve y concluye con su formalización en un lenguaje de programación, por medio de la escritura de un programa que lo implementa y su ejecución por parte de una computadora. Una primera conclusión es que el modelo permite explicar todo el proceso en forma satisfactoria. Al igual que en otras investigaciones previas, los datos obtenidos en la primera parte del estudio muestran que la transición del pensamiento de cada estudiante desde la etapa *intra* hasta la etapa *inter* se explica fundamentalmente por medio de la *ley general de la cognición* y la herramienta cognitiva *abstracción*. La segunda parte constituye la primera constatación de cómo la extensión a la ley general de la cognición explica la transición desde la etapa *inter* hasta la etapa *trans*.

En cuanto a la metodología empleada para el estudio, la realización de entrevistas clínicas, al estilo de las conducidas por Piaget, probó nuevamente ser de utilidad para la observación del proceso de construcción por parte de los participantes y la extracción de datos empíricos, al igual que en todos los estudios previos. El diseño de este tipo de entrevistas requiere mucha precisión, siendo fundamental enfocarse tanto en el resultado final esperado como en el desarrollo de todo el proceso, a efectos de obtener información representativa del mismo. Las actividades a plantear en las entrevistas deben permitir a los sujetos interactuar, en el plano *instrumental*, con instancias concretas del problema y posibilitar luego la transición de su pensamiento hacia los planos *conceptual* y *formal*. Además, se siguió la recomendación de realizar una experiencia *piloto* para validar y/o ajustar el diseño del estudio. También deben definirse criterios adecuados para la selección de participantes, con el fin de reducir lo más posible cualquier sesgo en la información obtenida, no empañada por ideas preconcebidas de los participantes sobre el problema planteado o por conocimiento conceptual o formal acerca del mismo, construido en forma previa.

Para el estudio, se aplicaron resultados y recomendaciones de trabajos previos en el diseño de la actividad propuesta para estudiar el pasaje de la etapa *intra* a la etapa *inter*. En cuanto al pasaje de la etapa *inter* a la etapa *trans*, las cuatro actividades se

diseñaron con el propósito de explorar el uso de un *formalismo intermedio* (entre el lenguaje natural y un lenguaje formal) y el rol del mecanismo de *automatización* (ejecución de pasos o instrucciones por parte de otra persona que oficia de *agente externo*) como elementos facilitadores, en base a la evidencia sobre sus beneficios recogida en estudios previos. A partir de los resultados del estudio, se concluye que efectivamente facilitan la construcción de conocimiento formal, tanto sobre las reglas de sintaxis y semántica del *lenguaje formal* (lenguaje de programación), como sobre el *programa* construido (tanto a nivel *textual* como *ejecutable*). Se trata del primer estudio empírico, basado en el modelo, expresamente diseñado para combinar ambas herramientas metodológicas con el marco teórico dado por la ley extendida.

El estudio permitió además obtener información acerca de aspectos inherentes al *algoritmo* y al *programa* construidos por cada estudiante durante el proceso. En primer lugar, se constata que la construcción de la *lógica* del algoritmo resulta más compleja que su posterior expresión en el *lenguaje formal* (lenguaje de programación). Además, se detectan dos posibles estrategias de búsqueda, no previstas en el diseño, las cuales fueron denominadas *a priori* y *a posteriori*. Las mismas se vinculan a la lógica del algoritmo y surgen durante su escritura en pseudocódigo (*formalismo intermedio*). Posteriormente, en la escritura del programa, se constatan diferencias en el uso de elementos del lenguaje de programación entre los que utilizan *C* y los que utilizan *Pascal*, a pesar de que ambos lenguajes permiten usar los mismos elementos para la formalización de la búsqueda lineal sobre un *arreglo*. Como es de esperar, se detecta cierta relación entre esto y la estrategia empleada. Se deduce que el conocimiento *formal* previo de cada estudiante sobre la evaluación de expresiones booleanas por *circuito corto* o *circuito completo* influye de forma inconsciente en la elección de la estrategia, a pesar de que la misma se construye *previo* a la formalización. Se concluye que es necesario investigar más a fondo todas estas cuestiones.

En relación a la construcción de un esquema más general que involucra elementos generalizados y sus transformaciones (características de la etapa *trans*), en el estudio se planteó un *nuevo* problema algorítmico (*filtrado*) para investigar la aplicación de conocimiento *formal* previamente construido sobre el problema de *búsqueda* y su explicación mediante la herramienta cognitiva *generalización*. El estudio permite concluir que, tal y como prevé la teoría, especialmente en línea con lo explicado por Cellérier [3] y según se detalla en la sección 3.6.8 del capítulo 3, es posible construir conocimiento nuevo en el plano *formal*, a partir de problemas previos que guardan similitudes y diferencias adecuadas. Esto abre la puerta a líneas de trabajo futuro relativas a la expansión del modelo.

Por último, durante el estudio se detectó el caso de un estudiante, no contemplado de antemano en el diseño (Pablo M), quien presentó mayor dificultad que el resto en el pasaje a la etapa *trans*, sin haber presentado dificultades previas en el pasaje a la etapa *inter*. La teoría explica este tipo de casos en la construcción de conocimiento científico por medio de lo que Piaget denomina un "*decalage*" vertical (sección 3.6.10 del capítulo 3). En el estudio se constata específicamente para el dominio concreto de conocimiento

sobre *programación*, lo que abre la puerta a la necesidad de investigar la cuestión más a fondo, para explicar con más detalle cómo opera el pensamiento de los sujetos cuando enfrentan este tipo de dificultades en la construcción de conocimiento sobre programación y poder así integrar este tipo de casos al modelo.

4.1.2 Relativas al modelo y sus aportes a CSE

En el capítulo 1 se planteó la importancia de investigar en CSE a partir de la integración de conocimiento sobre la disciplina informática con conocimiento sobre aspectos de cognición y educación. El relevamiento sobre el estado del arte en la disciplina arrojó que dicha integración aún constituye un gran debe en el desarrollo de la misma. Esto se explica, en parte, por el poco grado de evolución, en términos históricos, de CSE en comparación con otras didácticas más consolidadas como, por ejemplo, la didáctica de la matemática. Conforme a lo expuesto en dicho capítulo, hay consenso entre diversos autores en la necesidad de investigar en didáctica de la informática a partir de un marco teórico que tome en cuenta la mencionada integración. En este sentido, se concluye que el modelo constituye un aporte al desarrollo de la disciplina, pues se basa en un marco teórico sólido, dado por una teoría epistemológica que explica la construcción de conocimiento científico. Dicha teoría, integrada con conocimiento específico sobre la disciplina programación, posibilita el desarrollo del modelo, siendo capaz de brindar explicaciones satisfactorias acerca de cómo las personas aprenden a programar.

De la mano de lo anterior, muchos trabajos en didáctica de la informática no surgen de investigaciones sobre cuestiones de aprendizaje de conceptos de computación sino que son directamente propuestas de aplicación en el aula que no cuentan con un marco teórico que explique por qué son beneficiosas para el aprendizaje, o al menos no es explícito. Esto no significa que sean propuestas inadecuadas, sino que no expresan una fundamentación teórica que respalde sus beneficios, como surge del relevamiento realizado en el capítulo 1. Además, algunas propuestas realizan estudios cuantitativos para medir sus beneficios en el proceso de aprendizaje, lo cual constituye una constatación y no una explicación de por qué lo que se hace funciona (o no).

Al contar con una fundamentación teórica, el modelo brinda aportes significativos a la disciplina CSE. Por un lado, para el desarrollo de nuevas *investigaciones* acerca de cómo se construye conocimiento sobre programación y, por otro, a nivel de propuestas de *aplicación didáctica*. Respecto del primero, la integración del marco teórico dado por la teoría de Piaget con los lineamientos metodológicos del modelo abre la puerta para el diseño de nuevos estudios empíricos que investiguen cómo se construye conocimiento sobre conceptos de programación de diversa índole. En cuanto al segundo, en línea con lo expresado en el capítulo 1 en relación a las cuatro preguntas tradicionales de la didáctica: *qué*, *cómo*, *por qué* y *para quién* educar en informática, el modelo abre una línea de aplicación del mismo para la elaboración de *pautas didácticas* que tomen en cuenta *cómo* los estudiantes aprenden a programar.

Contar con un marco teórico que explica en detalle cómo construyen conocimiento sobre programación posibilita el diseño de actividades para trabajar en clase basadas en el proceso que transita el pensamiento de los estudiantes durante la construcción de conocimiento. Algunas propuestas preliminares [27, 37] fueron desarrolladas en los trabajos previos, durante la construcción del modelo y en la próxima sección se introduce una propuesta a futuro, a partir de los resultados del presente trabajo.

4.2 Trabajos futuros

Existen múltiples líneas posibles para trabajos futuros. En esta sección se proponen cuatro líneas específicas que, a partir del trabajo realizado, se consideran puntos de partida adecuados para el crecimiento del modelo y contribución al desarrollo de CSE. Se espera que el modelo, que es abierto, se enriquezca con nuevos aportes, ajustes y/o modificaciones a medida que se concreten nuevas investigaciones. Las primeras tres líneas son de *investigación* y la cuarta es de *aplicación didáctica*. Tales líneas son: profundización en el estudio de la construcción de conocimiento sobre *búsqueda lineal* (sección 4.2.1), capitalización de lo aprendido en relación a las herramientas teóricas y metodológicas para estudiar construcción de conocimiento sobre *nuevos* problemas (sección 4.2.2), nuevas investigaciones para *expansión* del modelo (sección 4.2.3) y elaboración de *pautas didácticas* basadas en el modelo (sección 4.2.4).

4.2.1 Profundización en el estudio de la *búsqueda lineal*

La primera línea se compone de cuatro propuestas para profundizar en la investigación sobre construcción de conocimiento acerca de la búsqueda lineal, iniciada en el estudio realizado en el capítulo 3. La primera apunta a refinar aspectos de la construcción de conocimiento sobre la búsqueda lineal en la estructura de datos empleada (*arreglo*) que, por cuestiones de alcance, no llegaron a ser incluidas en el diseño del estudio. La segunda extiende el estudio del problema hacia otras estructuras de datos. La tercera consiste en la construcción de versiones más *generales* de soluciones al problema de búsqueda, con el fin de consolidar el conocimiento en la etapa *trans* sobre dicho problema algorítmico. Por último, en línea con lo expresado en las conclusiones acerca de la construcción de un esquema más general propio de la etapa *trans*, la cuarta trata el estudio de la construcción de conocimiento sobre *nuevos* problemas algorítmicos que guardan razonables similitudes y diferencias con la búsqueda lineal, además de lo ya hecho con el problema de *filtrado* en el estudio realizado.

Profundización en búsqueda lineal sobre arreglo

La primera propuesta consiste en extender el estudio para obtener más información relativa a la construcción de conocimiento sobre la búsqueda lineal en un arreglo. Como se mencionó al inicio del capítulo 3, se trata de un problema considerado fundamental en la formación de profesionales en informática, por lo que profundizar en su estudio sería de utilidad, no solo para enriquecer la investigación realizada, sino también para contar con más elementos que posibiliten el diseño a futuro de pautas didácticas basadas en datos recogidos. Por ejemplo, profundizar en cómo se construye

conocimiento sobre la noción de *invariante* relativa a la iteración que realiza la búsqueda, tema que fue abordado de forma tangencial en el estudio. En este sentido, el diseño debería incluir preguntas para volver *conscientes* los aspectos de la noción de invariante. Por ejemplo, que cada entrada a la iteración corresponde a una búsqueda parcial y que en cada una siempre se cumple determinada condición (invariante).

Búsqueda lineal sobre otras estructuras de datos

Como segunda propuesta, se plantea estudiar la construcción de conocimiento acerca del *mismo* problema algorítmico (búsqueda lineal), pero implementado sobre *otras* estructuras de datos (por ejemplo, *arreglos dinámicos* o *listas encadenadas*). Esto permitiría analizar no solamente la construcción de conocimiento cuando varía la estructura de datos, sino también obtener información sobre aspectos inherentes a los *programas* que implementen la búsqueda sobre dichas estructuras, de modo similar a la información obtenida para la estructura *arreglo* (sección 3.6.7 del capítulo 3). Esto abre la puerta para profundizar en la relación entre la *lógica* del algoritmo (construida al escribir la versión en pseudocódigo, secciones 3.6.1 y 3.6.2 del capítulo 3) y posibles implementaciones sobre distintas estructuras.

Versiones más generales de la búsqueda lineal

En tercer lugar, se propone estudiar la construcción de conocimiento sobre versiones más *generales* de la búsqueda lineal. Por ejemplo, investigar la construcción cuando los elementos del arreglo son de *cualquier* tipo. El estudio realizado abordó el caso concreto en que los elementos son *números enteros* y analizó el salto al caso general específicamente para la noción de tamaño *genérico* del arreglo, dado por la constante N de la declaración (secciones 3.6.4 y 3.6.6 del capítulo 3). Se propone analizar el salto al caso general para el *tipo* de los elementos, además de para el *tamaño*, a efectos de transitar hacia una generalización que abarque ambos aspectos para este problema y establecer lineamientos para el estudio a futuro de variantes más genéricas de otros problemas algorítmicos.

Nuevos problemas algorítmicos a partir de la búsqueda lineal

Por último, así como se hizo con el estudio la construcción de conocimiento sobre el problema de *filtrado* (sección 3.6.9 del capítulo 3), trabajando directamente en el plano *formal*, se propone realizar estudios análogos para otros problemas algorítmicos que también guarden razonables similitudes y diferencias con la búsqueda lineal sobre un arreglo. Por ejemplo, problemas de *conteo* de elementos que cumplan una cierta propiedad, o incluso otros que además integren variantes del problema de búsqueda original como, por ejemplo, buscar una determinada cantidad de *ocurrencias* de un valor dado en el arreglo. Dicho problema guarda razonable similitud con la búsqueda (pues también requiere dos condiciones de parada) y diferencia (se integra el conteo de ocurrencias). La búsqueda debe finalizar cuando se llega a contar la cantidad de ocurrencias dada o cuando no hay más celdas. A su vez, en línea con la segunda propuesta, se propone luego realizar el mismo proceso para otras estructuras de datos.

4.2.2 Estudio de nuevos problemas y familias de algoritmos

La segunda línea consiste en volcar el aprendizaje, tanto teórico como metodológico, de las herramientas usadas en el estudio sobre búsqueda lineal, a la investigación de construcción de conocimiento sobre *nuevos* problemas algorítmicos. Se presentan tres propuestas. Las primeras dos capitalizan el uso de los principios teóricos (*ley de la cognición* y su *extensión*, *herramientas cognitivas*, *elemento genérico*) y herramientas metodológicas (*entrevista clínica*, *formalismo intermedio*, *automatización*) a efectos de ampliar el espectro de problemas algorítmicos a estudiar en el marco del modelo. La tercera profundiza en el rol de la *automatización* en la construcción de conocimiento sobre algoritmos, estructuras de datos y programas y busca retroalimentar el modelo para perfeccionar el uso de dicha herramienta en futuros estudios.

Estudio de nuevos problemas algorítmicos

El estudio sobre búsqueda lineal fue el primero en emplear los principios teóricos junto con el uso de un *formalismo intermedio* y *automatización* para explicar la construcción de conocimiento formal sobre programas. La primera propuesta es realizar nuevos estudios que usen las mismas herramientas teóricas y metodológicas para investigar la construcción de conocimiento sobre otros problemas algorítmicos. En los estudios previos, estas herramientas se introdujeron en los planos *instrumental* y *conceptual*, quedando planteada la necesidad de profundizar sobre ellas en el plano *formal*. El estudio del capítulo 3 fue una primera experiencia de investigación en ese sentido, siendo necesario consolidar el uso de las mismas en nuevos estudios, lo cual además constituye un aprendizaje significativo sobre el propio modelo en sí. Por ejemplo, estudiar problemas de *ordenación*, *inserción* o *eliminación* en diversas estructuras de datos (*arreglos*, *listas encadenadas*, *árboles*, *estructuras de hash*, etc). Esto abre el abanico de problemas algorítmicos a estudiar en el marco del modelo y permite enriquecerlo a partir de más datos empíricos y así seguir profundizando en el rol del formalismo intermedio y de la automatización en el proceso de construcción. Por otra parte, como en el estudio realizado, esto también posibilita obtener datos sobre aspectos inherentes a los *algoritmos* y *programas* que solucionan los nuevos problemas y su vinculación con la construcción de conocimiento.

Construcción de conocimiento sobre familias de algoritmos

En segundo lugar, se propone investigar construcción de conocimiento sobre *familias* de algoritmos. Conforme a lo expresado en la sección 2.4 del capítulo 2, la tríada de etapas está presente en un proceso cíclico. El proceso de construcción de conocimiento en cada nivel (*instrumental*, *conceptual* y *formal*) y pasando por las tres etapas, constituye un proceso continuo que se prolonga hacia la construcción de estructuras generales propias de la etapa *trans* (lo que Piaget denomina *systèmes d'ensemble* en [13]). La construcción de conocimiento en el plano *formal* sobre un algoritmo concreto y un programa que lo implementa abre la posibilidad de investigar construcción de conocimiento relativa a familias de algoritmos.

En una primera etapa, para estudiar la construcción de conocimiento sobre una familia de algoritmos determinada, la propuesta consiste en elegir un algoritmo concreto de dicha familia y realizar el estudio completo para el mismo desde la etapa *intra* hasta la etapa *trans*, como lo hecho en el estudio para la búsqueda lineal sobre un arreglo. Luego, proponer actividades similares a la del nuevo problema en dicho estudio (*filtrado*), que requieran la escritura de nuevos programas para implementar otros algoritmos de la misma familia, directamente en el plano *formal*, a partir de sus similitudes y diferencias con el programa que implementa el algoritmo original. Por ejemplo, para algoritmos de *búsqueda*, se puede proponer la escritura de otros programas que implementen *búsqueda lineal ordenada* y *búsqueda binaria*, a partir del conocimiento formal construido previamente sobre la *búsqueda lineal*. Además, en línea con la segunda y la tercera propuesta de la sección anterior, se puede realizar nuevas variantes de estos estudios para investigar el salto a tamaños y tipos *genéricos* y/o implementación de los mismos algoritmos de cada familia sobre *otras* estructuras de datos. Las combinaciones son múltiples y abren la puerta hacia más propuestas para enriquecimiento del modelo.

Los estudios propuestos en el párrafo anterior pueden llevarse a cabo empleando las mismas herramientas (teóricas y metodológicas) usadas en el capítulo 3. También se propone extender el modelo, tanto a nivel teórico como metodológico, para investigar cómo el conocimiento sobre una familia de algoritmos determinada se integra con conocimiento sobre otras familias en una estructura de conocimiento en la etapa *trans* más amplia y superior. Esto abre la puerta hacia otra línea de trabajo futuro relativa a construcción de conocimiento sobre el vínculo entre problemas algorítmicos diferentes y familias de algoritmos que les dan solución. Por ejemplo, cómo el conocimiento sobre algoritmos de *búsqueda* se integra con conocimiento sobre algoritmos de *ordenación*, siendo la construcción de conocimiento acerca de esta última familia estudiada de la misma forma, a partir de algún algoritmo concreto (por ejemplo, ordenación por *burbuja*), desde el plano *instrumental* y hasta el *formal* e implementando después otros algoritmos, trabajando directamente en el plano formal, a partir de sus similitudes y diferencias con el original (por ejemplo, ordenación por *selección*, ordenación por *inserción*, etc). Se piensa que esta línea de trabajo resulta fundamental para consolidar la construcción de conocimiento en la etapa *trans* relativa a algoritmos, estructuras de datos y programas.

Profundización en el rol de la automatización

La *automatización*, entendida como un mecanismo en el cual otra persona, distinta del sujeto que escribe los pasos de un algoritmo, oficia de *agente externo* y los ejecuta, ha probado ser de gran utilidad para ayudar a los sujetos a construir conocimiento sobre algoritmos y estructuras de datos. La evidencia recogida en los estudios previos así lo sugería y se constató de nuevo en el estudio presentado en el capítulo 3. La tercera propuesta consiste en analizar el rol de la automatización con más profundidad para el caso de *programas*, donde el agente externo es la *computadora*. Esta propuesta podría enmarcarse en los nuevos estudios de las propuestas anteriores de esta misma

sección, ya que podría ser beneficiosa para los mismos, además de analizar el rol de la automatización en sí. Si bien la "simulación" de la computadora por parte de otra persona ha probado ser efectiva en el estudio del capítulo 3, interesa diseñar estudios que incorporen a la propia computadora como agente externo ejecutor durante la construcción de conocimiento. Esto podría hacerse sacando provecho de diversas aplicaciones informáticas ya existentes. Por ejemplo, las propias herramientas de depuración (*debugger*) ya incorporadas en muchos entornos de desarrollo (*IDE*) o lenguajes de programación que permiten simular ejecución de modo visual, pensados expresamente para el aprendizaje de la disciplina, como por ejemplo *Scratch* [77]. La propuesta no es usar tales herramientas como fin en sí mismo, sino sacar provecho de su incorporación a estudios basados en el modelo, siempre con el fin de analizar la construcción de conocimiento. Esto también abre la puerta al diseño de mejores representaciones de *estructuras de datos* (con más elaboración que un simple dibujo en papel, como se hizo en el estudio del capítulo 3) que ayuden aún más a construir conocimiento sobre las mismas. Además, posibilita otra línea futura de expansión del modelo que incluya construcción de conocimiento relativo a *verificación* de programas.

4.2.3 Nuevas investigaciones para expansión del modelo

Como tercera línea, se presentan cinco propuestas para expansión del modelo, en términos de cuestiones relativas a construcción de conocimiento sobre programación que, a diferencia de las líneas anteriores, no están vinculadas directamente al estudio de problemas algorítmicos en sí. La primera propone explorar la noción de *lógica* de un algoritmo. La segunda busca profundizar en la construcción de conocimiento sobre programas *recursivos*. La tercera indaga en la construcción de conocimiento acerca de *formalismos*. La cuarta incursiona en otros *paradigmas* de programación (el estudio del capítulo 3 abarcó solamente algoritmos *iterativos* bajo un paradigma *imperativo* y *estructurado*). Por último, la quinta sienta las bases para un *análisis histórico crítico* relativo a la evolución de la programación como disciplina científica.

Rol de la lógica de los algoritmos en la construcción de conocimiento

En el capítulo 3 (sección 3.5.1), se propuso definir el concepto de *lógica* de un algoritmo como el orden de la ejecución de los pasos expresados en pseudocódigo, pero sin considerar todavía aspectos de su posterior implementación en un lenguaje de programación. El autor de este trabajo considera que dicha definición es insuficiente, pues no expresa el concepto con la exactitud deseada. No se trata solamente del orden en sí de la ejecución de los pasos, sino que subyace "algo más" relativo a la génesis que lleva a un sujeto a proponer dicho orden por sobre otro al idear los pasos del algoritmo (como se constató en el estudio, con las estrategias de búsqueda *a priori* y *a posteriori*). No obstante, entiende que la definición propuesta permitió ilustrar el concepto con suficiente claridad en relación al propósito del estudio.

Durante la realización de este trabajo, se buscó en la literatura académica alguna definición del concepto (incluso bajo otras denominaciones que no involucren la palabra *lógica*) que estuviese alineada con la noción propuesta en el capítulo 3, pero

no se encontró ninguna. Lo más cercano que se halló es la propuesta de Kowalski en [48]: *An algorithm can be regarded as consisting of a logic component, which specifies the knowledge to be used in solving problems, and a control component, which determines the problem-solving strategies by means of which that knowledge is used* (*se puede considerar que un algoritmo consiste en un componente lógico, que especifica el conocimiento a utilizar en la resolución de problemas, y un componente de control, que determina las estrategias de resolución de problemas por medio de la cual se usa ese conocimiento*, t.d.a). No obstante, dicha propuesta no refleja la misma noción, pues asocia la idea de *lógica* a *especificación* más que a *comportamiento*, asociando este último a lo que llama *control*. Además, en el resto del trabajo, Kowalski vincula ambas nociones (lógica y control) con las estructuras de datos usadas en la implementación, apartándose de la idea de que la *lógica* de un algoritmo (por ejemplo, para resolver el problema de la *búsqueda lineal*) no depende de la estructura sobre la cual luego se implemente en un lenguaje de programación (por ejemplo, un algoritmo que resuelve la búsqueda lineal tiene la misma lógica tanto si se implementa sobre un *arreglo* como sobre una *lista encadenada*).

En el marco del modelo propuesto, que estudia construcción de conocimiento sobre algoritmos, estructuras de datos y programas, comprender la construcción de la *lógica* de un algoritmo resulta fundamental, dado que la misma atraviesa todo el proceso de construcción, desde su génesis a nivel *instrumental* (etapa *intra*), pasando por lo *conceptual* (etapa *inter*) y llegando a lo *formal* (etapa *trans*). En todos los estudios realizados se constató que dicha construcción resulta compleja. La evidencia recogida muestra que los esquemas previamente construidos influyen en dicha construcción, lo cual se fundamenta por Cellérier [3], abarcando incluso el conocimiento *formal* previo, como se constató en el estudio con la evaluación de expresiones booleanas por *circuito corto* o *circuito completo* (sección 3.6.8 del capítulo 3). Por ello, se propone investigar más a fondo cómo los sujetos construyen la *lógica* de un algoritmo en forma previa a su formalización, lo cual se vincula a la lógica de acciones y significaciones elaborada por Piaget y García en [12]. De la mano con esto, indagar en la causa de dificultades como las mostradas por Pablo M durante el estudio, al desviarse de la lógica de su algoritmo, en el pasaje a la etapa *trans* (interpretadas como "*decalage*" vertical) puede aportar a una mejor comprensión de esta parte del proceso. El diseño de futuros estudios en esta línea requiere estudiar más a fondo la teoría de Piaget, en particular en lo relativo al rol de los *decalages* en la construcción de conocimiento.

El contraste entre la dificultad en hallar una definición del concepto de *lógica* de un algoritmo y la presencia subyacente del mismo en todos los estudios realizados durante la construcción del modelo, hace pensar que la obtención de datos empíricos a través de esta línea de trabajo puede eventualmente llevar al surgimiento de una definición más exacta. En la opinión del autor de este trabajo, esto está estrechamente ligado a la ausencia de una definición más exacta del concepto de *pensamiento computacional*. Desde que Wing popularizó una caracterización del mismo en [61]: *Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science* (*el pensamiento*

computacional implica resolver problemas, diseñar sistemas y comprender el comportamiento humano, basándose en los conceptos fundamentales de la informática, t.d.a) la literatura ha propuesto múltiples intentos de definición del concepto, todos ellos incompletos, como menciona da Rosa en [28]. En ese mismo trabajo, da Rosa sugiere que una definición adecuada del concepto podría surgir de la *extensión a la ley general de la cognición*, formulada en [33] y empleada en este trabajo para explicar el pasaje de la etapa *inter* a la etapa *trans*. Se piensa que la profundización en esta línea de trabajo puede llegar a aportar definiciones más exactas y completas para ambos conceptos, a la vez que permitiría explicar, con mayor profundidad, cómo los sujetos construyen conocimiento sobre algoritmos, estructuras de datos y programas.

Construcción de conocimiento sobre programas recursivos

La segunda propuesta consiste en estudiar en detalle la construcción de conocimiento sobre programas *recursivos*, específicamente en el plano *formal*. Varios trabajos previos que llevaron a la construcción del modelo abordaron el tema [6, 31, 32], pero sin profundizar en el pasaje a la etapa *trans*. Se propone realizar estudios empíricos que analicen el proceso completo de construcción, desde el plano *instrumental* hasta el plano *formal*, para soluciones recursivas a problemas algorítmicos. Ello requiere estudiar la construcción de conocimiento *formal* sobre el concepto de recursión, complementando los aspectos instrumentales y conceptuales desarrollados en las investigaciones previas. Además, esto posibilita dar inicio a otra línea de trabajo para investigar la construcción de conocimiento sobre el vínculo entre soluciones *recursivas* y sus contrapartes *iterativas*, como dos formalizaciones posibles de un mismo método que resuelve un problema algorítmico mediante repetición de acciones. Un posible punto de partida podría ser estudiar nuevamente el problema de la búsqueda lineal, mediante la implementación de una solución recursiva en vez de iterativa. Se aplicaría por primera vez la ley extendida para explicar el proceso de construcción para esta clase de algoritmos junto con automatización y uso de un formalismo intermedio, realizando posibles ajustes al modelo en base a los resultados.

Construcción de conocimiento sobre formalismos

En todos los estudios que llevaron a la construcción del modelo, el uso de *lenguajes formales* (lenguajes de programación) y *formalismos intermedios* ha sido siempre funcional al estudio de construcción de conocimiento sobre algoritmos, estructuras de datos y programas. Si bien el foco ha estado puesto en esto (ya que los formalismos se usan justamente como *medio* para expresar conceptos, en este caso de programación), siempre se ha tenido presente la necesidad de estudiar específicamente la construcción de conocimiento sobre los formalismos en sí mismos. En particular, cómo se construye conocimiento sobre sus reglas de sintaxis y de semántica. La evidencia recogida en todos los estudios muestra que dicha construcción surge de la interacción del sujeto con el formalismo, como ocurre con el conocimiento sobre cualquier objeto a partir de la interacción del sujeto con el mismo, conforme explica la teoría de Piaget.

Por lo tanto, la tercera propuesta es realizar estudios pensados expresamente para analizar la interacción entre sujeto y formalismo, más que estudiar cómo se construye conocimiento sobre problemas algorítmicos mediante su aplicación. Dado que hay una relación dialéctica entre los formalismos en programación y el comportamiento expresado por ellos, se espera retroalimentar el modelo a partir de estos estudios y perfeccionar la metodología para futuros estudios sobre algoritmos, estructuras de datos y programas. Una mejor comprensión acerca de cómo los sujetos construyen conocimiento relativo a formalismos puede ayudar a comprender mejor cómo lo construyen sobre programas que dan solución a problemas algorítmicos. Además, se propone explorar el vínculo entre el *lenguaje natural*, que es el primer formalismo usado para expresar conocimiento sobre algoritmos en el plano *conceptual*, y su relación con otros formalismos (tanto *intermedios* como lenguajes *formales*). Por ejemplo, analizar qué tipo de correspondencia se genera entre las construcciones lingüísticas del lenguaje natural y las reglas de sintaxis y de semántica de un formalismo en programación. El estudio del capítulo 3 se limitó únicamente a reconocer un paralelismo entre los pasos expresados en pseudocódigo y su contraparte dada por las instrucciones del lenguaje de programación (sección 3.6.6) pero sería oportuno profundizar en dicha correspondencia en todo el proceso de construcción.

Extensión del modelo para abarcar otros paradigmas de programación

Todos los estudios llevados a cabo durante la construcción del modelo trabajaron con lenguajes de programación *imperativos*, *estructurados* y/o *funcionales*. Excepto por el estudio presentado en este trabajo, ninguno de ellos profundizó en el pasaje a la etapa *trans*. En todos se usó un lenguaje formal como herramienta utilitaria, pero no se ahondó en la construcción de conocimiento sobre el propio lenguaje ni sobre los programas correspondientes. Por ende, la cuarta propuesta es realizar estudios similares al presentado, pero haciendo uso de otros paradigmas para implementar los programas resultantes. Esta propuesta guarda estrecha relación con la anterior, dado que *paradigma* y *formalismo* están fuertemente vinculados. No es lo mismo programar en un lenguaje imperativo que en uno funcional o en otro orientado a objetos, habiendo notorias diferencias en términos de sintaxis y semántica según el paradigma. Por lo tanto, la investigación de la construcción de conocimiento sobre programas en el plano formal se bifurca en varias líneas de investigación, según el paradigma que se utilice.

Un posible punto de partida puede ser retomar nuevamente el problema de búsqueda lineal y estudiar el pasaje a la etapa *trans* cuando la solución se implementa, por ejemplo, en un lenguaje *orientado a objetos*, poniendo el foco en el análisis de la influencia del paradigma en la construcción de conocimiento formal, haciendo ajustes pertinentes al modelo, en caso de ser necesario. Otro enfoque podría ser analizar el cambio de paradigma trabajando directamente en el plano *formal*, nuevamente pensando en la construcción de esquemas generales en la etapa *trans*. Por ejemplo, partiendo del programa que resuelve la búsqueda lineal en un lenguaje *estructurado* (como ser *C* o *Pascal*), usar ese conocimiento formal para escribir una nueva versión en un lenguaje *orientado a objetos* (como ser *C#* o *Java*).

Análisis histórico crítico de la programación como disciplina científica

Como se menciona al inicio del capítulo 2, las dos fuentes de datos que nutren a la teoría de Piaget son la *psicología genética* y un *análisis crítico de la historia de las ciencias*, donde la primera estudia los mecanismos de pensamiento que llevan a la construcción de conocimiento en los sujetos y el segundo constituye un análisis epistemológico y sociogenético del desarrollo del conocimiento científico en la historia. El modelo propuesto consiste en una instancia de la teoría de Piaget al dominio concreto de conocimiento sobre programación. La mayoría de los estudios realizados en su construcción se centraron en la perspectiva *psicogenética*, con poca incursión en el *análisis histórico* de la evolución de la programación como disciplina científica. Por tanto, la quinta propuesta consiste en avanzar en dicho análisis, a efectos de que el modelo constituya una instancia más completa de la teoría, abarcando ambas fuentes.

Piaget y García estudiaron en detalle el vínculo entre el conocimiento *psicogenético* y el *sociogenético* [13], expresado a través de la tríada presente en ambos procesos de construcción de conocimiento y la relación dialéctica entre ellos. El análisis de uno de los procesos aporta elementos para explicar el otro y viceversa, como lo describen para disciplinas científicas tales como física, geometría o álgebra. Un ejemplo notable es la constatación hecha por los autores de que, en el campo de la geometría, la noción de espacio se construye, a nivel psicogenético, partiendo de experiencias topológicas y siguiendo posteriormente con proyectivas y euclidianas. En cambio, a nivel histórico, el desarrollo de la geometría euclidiana como disciplina formal ocurrió en forma previa al de las geometrías proyectivas y la topología. Desde hace años, el grupo de Didáctica del InCo tiene como objetivo a largo plazo abordar el análisis histórico crítico de la programación como disciplina científica, en el entendido de que no sólo resultará útil desde el punto de vista histórico, sino también profundizar en el conocimiento de la psicogénesis de los conceptos de programación en relación con su sociogénesis, contribuyendo así a mejorar su didáctica. Dado que el análisis histórico crítico está estrechamente vinculado a la etapa *trans* de la tríada, se entiende que la concreción del estudio presentado en este trabajo sienta las bases adecuadas para su abordaje.

Esta línea está vinculada a las tres anteriores en el sentido de que dos puntos de partida adecuados para el análisis histórico crítico son, por un lado, una revisión histórica de los conceptos de *recursión* e *iteración* y, por el otro, una revisión histórica de la evolución de los *lenguajes de programación*. Los primeros están fuertemente ligados a la noción de *algoritmo* a nivel *instrumental* y *conceptual*, así como a la noción de *programa* a nivel *formal* y los segundos constituyen expresiones de conocimiento *tematizado* (*thematized knowledge*), en palabras de Piaget y García en [13]. Los datos aportados por esta línea de trabajo podrían retroalimentar las tres propuestas anteriores, que proponen abordar dichas construcciones a nivel de los sujetos (perspectiva *psicogenética*). Una visión global de la evolución de los lenguajes de programación aportaría datos valiosos al diseño de estudios sobre cómo los sujetos construyen conocimiento sobre formalismos, paradigmas y construcción de soluciones recursivas y/o iterativas a problemas algorítmicos.

4.2.4 Pautas didácticas basadas en el modelo

Las propuestas de las secciones anteriores son para *investigación* en CSE. En esta sección se presenta una propuesta de *aplicación didáctica*. Uno de los objetivos de las investigaciones que guiaron la construcción del modelo es proveer fundamentos para elaborar pautas didácticas de aplicación en el aula. Algunas propuestas preliminares fueron presentadas en [27, 37] y se propone volcar los resultados del presente trabajo para la elaboración de nuevas pautas. La propuesta consiste en avanzar a la formación de equipos especializados en el diseño de pautas didácticas, integrados por docentes con formación en CSE junto con *pedagogos* y otros actores, dado que no solo debe tomarse en cuenta la especificidad de la disciplina (que es fundamental), sino también otros factores inherentes a los sistemas educativos, como ser planes de estudios, cantidad de estudiantes, modalidad de dictado, recursos, etc. Se espera que esto contribuya a mejorar aspectos de educación en informática, así como a enriquecer el PCK y a la profesionalización de la formación de docentes en la disciplina.

Un posible punto de partida es la creación de proyectos de diseño de pautas didácticas, guiadas por el modelo, para introducción de nuevos temas en cursos iniciales de programación. Se trata de pautas que parten del conocimiento *instrumental* de los estudiantes en relación a dichos temas y siguen luego con la construcción de conocimiento *conceptual* y *formal*. Este último, a su vez, puede usarse como base para la elaboración de nuevas pautas para la introducción de nuevos conceptos. En el anexo F se proponen una serie de recomendaciones basadas en el modelo para comenzar a trabajar en esta dirección. También se brinda un ejemplo concreto de pautas didácticas que siguen las mismas, como forma de ilustrar un posible enfoque que se podría tomar para introducir un tema específico de programación en el contexto de una clase.

Bibliografía

[1] Bonwell, C., Eison, J., *Active Learning: Creating Excitement in the Classroom*. Ashe-Eric Higher Education Report nº 1. Washington D.C: The George Washington University, School of Education and Human Development (1991).

[2] Brousseau, G., *Iniciación al estudio de la Teoría de las Situaciones Didácticas* (trad. Español). Buenos Aires: Libros del Zorzal (2007).

[3] Cellérier, G., *Structures and Functions*. En: Inhelder, B., de Caprona, D., Cornu-Wells, A., *Piaget today*. Lawrence Erlbaum Associates Ltd. Publishers, UK (1987).

[4] Chevallard, Y., *La transposición didáctica* (trad. Español). Buenos Aires: Ed. Aique (1997).

[5] Comenius, J., *Didáctica Magna* (1632). Re-Edición en Madrid: Ed. Akal (1986).

[6] da Rosa, S., *The Learning of Recursive Algorithms and their Functional Formalization* [Tesis de doctorado, PEDECIBA Informática, Uruguay] (2005).

[7] Dubinsky, E., Mcdonald, M.A, APOS: *A Constructivist Theory of Learning in Undergraduate Mathematics Education Research*. En: Holton, D., Artigue, M., Kirchgräber, U., Hillel, J., Niss, M., Schoenfeld, A., *The Teaching and Learning of Mathematics at University Level*. New ICMI Study Series, vol 7. Springer (2001).

[8] García, R., *El conocimiento en construcción*. Barcelona: Editorial Gedisa (2000).

[9] Gréco, P., Inhelder, B., Matalon, B., Piaget, J., *La Formation des raisonnements récurrentiels*. Paris: Presses Universitaires de France (1963).

[10] Marton, F., *Phenomenography: A Research Approach to Investigating Different Understandings of Reality*. En: Sherman, R., Webb, R., *Qualitative Research in Education: Focus and Methods*. London: The Falmer Press (1988).

[11] Parkin, A., *Exploraciones en neuropsicología cognitiva*. Madrid: Editorial Panamericana, pag.3 (1999).

[12] Piaget, J., García, R., *Hacia una Lógica de Significaciones* (trad. Español). Barcelona: Editorial Gedisa (1987).

[13] Piaget, J., García, R., *Psychogenesis and the History of Sciences*. New York: Columbia University Press (1980).

[14] Piaget, J., *La prise de conscience*. Paris: Presses Universitaires de France (1974).

[15] Piaget, J., *L'equilibration des Structures Cognitives: Probleme Central du Developpement*. Paris: Presses Universitaires de France (1975).

[16] Piaget, J., *Recherches sur la Généralisation*. Paris: Presses Universitaires de France (1978).

[17] Piaget, J., *Success and Understanding*. Harvard University Press (1978).

[18] Andrews, P., "Conditions for learning: a footnote on pedagogy and didactics". ATM Mathematics Teaching Journal 204, p 22 (2007).

[19] Bolívar, A., "Conocimiento Didáctico del Contenido y Didácticas Específicas". Universidad de Granada. Revista de currículum y formación del profesorado vol 9, nº 2, pp 1-39 (2005).

[20] Bower, M., "Lessons from Cognitive Science for Computer Science Education". Macquarie University, Australia. (2008).

[21] Brown, N., Kölling, M., Crick, T., Peyton Jones, S., Humphreys, S., Sentance, S., "Bringing Computer Science Back into Schools: Lessons from the UK". SIGCSE '13, Proceedings of the 44th ACM technical symposium on Computer science education, pp 269-274 (2013).

[22] Bubica, N., Boljat, I., "Programming Novices' Mental Models". 7th International Conference on Education and New Learning Technologies, Barcelona (2015).

[23] Bucks, G., Oakes, W., "Phenomenography as a Tool for Investigating Understanding of Computing Concepts". American Society for Engineering Education, Annual Conference & Exposition (2011).

[24] Budd, T., "An Active Learning Approach to Teaching the Data Structure Course". SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education, pp 143–147 (2006).

[25] Cetin, I., "Students' Understanding of Loops and Nested Loops in Computer Programming: An APOS Theory Perspective". Canadian Journal of Science, Mathematics and Technology Education, vol 15, nº 2, pp 155-170 (2015).

[26] Coll, C., Miras, M., "Los Factores Socio-Culturales y el Desarrollo Cognitivo en la Teoría de Genética de Jean Piaget". Anuario de Psicología. The UB Journal of Psychology nº 9, pp 3-24 (1973).

[27] da Rosa, S., "El rol del estudiante como diseñador de contenidos". SADIO Electronic Journal vol 19, nº 2, pp 151-166 (2020).

[28] da Rosa, S., "Piaget and Computational Thinking". CSERC '18: Proceedings of the 7th Computer Science Education Research Conference, pp 44-50 (2018).

[29] da Rosa, S., "Studying preconceptions of novice learners about program execution". Proceedings of the 27th Psychology of Programming Interest Group Workshop, Cambridge, UK, paper 22 (2016).

[30] da Rosa, S., "The construction of knowledge of basic algorithms and data structures" Proceedings of the 26th Psychology of Programming Interest Group Workshop, Bournemouth, UK, paper 7 (2015).

[31] da Rosa, S., "The Construction of the Concept of Binary Search Algorithm". Proceedings of the 22th Psychology of Programming Interest Group Workshop, Madrid, Spain, pp 100-111 (2010).

[32] da Rosa, S., "The Learning of Recursive Algorithms from a Psychogenetic Perspective". Proceedings of the 19th Psychology of Programming Interest Group Workshop, Joensuu, Finland, pp 201-215 (2007).

[33] da Rosa, S., Aguirre, A., "Students teach a computer how to play a game". En: Informatics in Schools. Fundamentals of Computer Science and Software Engineering. Lecture Notes in Computer Science, vol. 11169, pp 55-67. Springer-Verlag (2018).

[34] da Rosa, S., "Designing Algorithms in High School Mathematics". Lecture Notes in Computer Science vol 3294. Springer-Verlag (2004).

[35] da Rosa, S., Gómez, F., "Hacia un modelo de investigación en didáctica de la programación" Proceedings of the 2019 XLV Latin American Computing Conference (CLEI), pp. 1-8 (2019).

[36] da Rosa, S., Gómez, F., "A research model in didactics of programming". CLEI Electronic Journal vol 23, nº 1, paper 5 (2020).

[37] da Rosa, S., Gómez, F., "An educational methodology based on the work of students". CLEI Electronic Journal vol 13, nº 2, paper 6 (2010).

[38] Dowek, G., "Les quatre concepts de l'informatique". Sciences et technologies de l'information et de la communication en milieu éducatif. Patras, Grèce, pp 21-29 (2011).

[39] Dowek, G., "Quelle informatique enseigner au lycée?". APMEP Bulletin 480, pp 93-97 (2009).

[40] Eden, A., "Three Paradigms in Computer Science". Minds and Machines 17(2), pp 135–167 (2007).

[41] Escera, C., "Aproximación histórica y conceptual a la Neurociencia Cognitiva". Revista Cognitiva vol 16, nº 2, pp 141-162 (2004).

[42] Friss de Kereki, I., "Incorporation of Kinesthetic Learning Activities to Computer Science 1 course: Use and Results". CLEI Electronic Journal vol 13, nº 2, paper 1 (2010).

[43] George, C., "Experiences with Novices: The importance of Graphical Representations in Supporting Mental Models" en A.F.Blackwell & E.Bilotta (Eds). Proc. PPIG 12, pp 33-44 (2000).

[44] Gomes, A., "Learning to program - difficulties and solutions". International Conference on Engineering Education ICEE, pp 283-287 (2007).

[45] Götschi, T., Sanders, I., Galpin, V., "Mental models of recursion". SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education, pp 346–350 (2003).

[46] Holmboe, C., McIver, L., George, C., "Research Agenda for Computer Science Education". G.Kadoda (Ed). Proc. PPIG 13, pp 207-223 (2001).

[47] Hubwieser, P., Armoni, M., Giannakos, M., Mittermeir, R., "Perspectives and Visions of Computer Science Education in Primary and Secondary (K-12) Schools". Journal ACM Transactions on Computing Education (TOCE), Special Issue on Computing Education in (K-12) Schools. vol 14, nº 2, article nº 7 (2014).

[48] Kowalski, R., "Algorithm = Logic + Control". Communications of the ACM, vol 22, nº 7, pp 424-436 (1979).

[49] Lewis, C., "Exploring Variation in Students' Correct Traces of Linear Recursion". Proceedings of the 2014 ACM Conference on International Computing Education Research (ICER), pp 67–74 (2014).

[50] Lister, R., "Concrete and other neo-Piagetian forms of reasoning in the novice programmer". Proceedings of ACE 2011: The 13th Australasian Computing Education Conference, vol 114 pp 9-18 (2011).

[51] Mannila, L., Peltomäki, M., Salakoski, T., "What about a simple language? Analyzing the difficulties in learning to program". International Conference on Engineering Education ICEE, pp 211-227 (2007).

[52] McConnell, J., "Active Learning and its use in Computer Science". ACM SIGCSE Bulletin vol 28, issue SI, pp 52–54 (1996).

[53] Moor, J., "Three Myths of Computer Science". The British Journal for the Philosophy of Science, vol 29, nº 3, pp. 213-222. Oxford University Press (1978).

[54] Papert, S., "What's the big idea? Toward a pedagogy of idea power". IBM Systems Journal vol 39, Issue 3-4, pp 720–729 (2000).

[55] Perales, F., "Desarrollo Cognitivo y Modelo Constructivista en la Enseñanza-aprendizaje de las Ciencias". Revista Interuniversitaria de Formación del Profesorado, nº 13 pp. 173-189 (1992).

[56] Saeli, M., Perrenet, J., Jochems, W., Zwaneveld, B., "Teaching Programming in Secondary School: A Pedagogical Content Knowledge Perspective". Informatics in Education, vol 10, nº 1, Vilnius University, pp 73–88 (2011).

[57] Schwamb, K., "Mental Models: A Survey". Department of Information and Computer Science. University of California (1990).

[58] Schwill, A., "Computer Science Education Based on Fundamental Ideas". Proceedings of the IFIP TC3 WG3.1/3.5 joint working conference on Information technology: supporting change through teacher education, pp 285-291 (1997).

[59] Shulman, L., "Those Who Understand: Knowledge Growth in Teaching". Educational Researcher, vol 15, nº 2, pp 4-14 (1986).

[60] Stephens-Martinez, K., Ju, A., Parashar, K., Ongowarsito, R., Jain, N., Venkat, S., Fox, A., "Taking Advantage of Scale by Analyzing Frequent Constructed-Response, Code Tracing Wrong Answers". Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER), pp 56-64 (2017).

[61] Wing, J., "Computational thinking". Communications of the ACM, vol 49, nº 3, pp 33-35 (2006).

[62] Wing, J., "Computational thinking and thinking about computing". Philosophical Transactions of the Royal Society. A Mathematical Physical and Engineering Sciences 366(1881), pp 3717–3725 (2008).

[63] Académie des sciences. Sitio Web https://www.academie-sciences.fr. Última consulta Sep. 2021.

[64] Association Enseignement Public & Informatique (EPI). Sitio Web https://www.epi.asso.fr. Última consulta Sep. 2021.

[65] Code.org. Sitio Web https://code.org. Última consulta Sep. 2021.

[66] Code::Blocks. Sitio Web https://www.codeblocks.org. Última consulta Sep. 2021.

[67] Computer Science Teachers Association (CSTA). Sitio Web https://www.csteachers.org. Última consulta Sep. 2021.

[68] Computer Science Unplugged. Sitio Web https://csunplugged.org/es. Última consulta Sep. 2021.

[69] Computing at School (CAS). Sitio Web https://www.computingatschool.org.uk. Última consulta Sep. 2021.

[70] Free Pascal. Sitio Web https://www.freepascal.org. Última consulta Sep. 2021.

[71] Genetic Epistemology, a series of lectures delivered by Piaget at Columbia University. Publicado por Columbia Univesity Press. Disponible en https://www.marxists.org/reference/subject/philosophy/works/fr/piaget.htm. Última consulta Sep. 2021.

[72] Instituto de Computación. Facultad de Ingeniería (UdelaR). Sitio web https://www.fing.edu.uy/inco. Última consulta Sep. 2021.

[73] Instituto de Filosofía. Facultad de Humanidades y Ciencias de la Educación (UdelaR). Sitio web https://www.fhuce.edu.uy/index.php/filosofia. Última consulta Sep. 2021.

[74] Núcleo Interdisciplinario Filosofía de la Ciencia de la Computación. Sitio Web https://www.fing.edu.uy/grupos/nifcc. Última consulta Sep. 2021.

[75] Piaget, las ciencias y la dialéctica. Entrevista a Rolando García. Disponible en https://herramienta.com.ar/articulo.php?id=753. Última consulta Sep. 2021.

[76] Profesorado de Informática. Sitio web http://www.cfe.edu.uy/index.php/planes-y-programas/planes-vigentes-para-profesorado/44-planes-y-programas/profesorado-2008/365-informatica. Última consulta Sep. 2021.

[77] Scratch. Sitio Web https://scratch.mit.edu. Última consulta Sep. 2021.

Anexo A

Descripciones en lenguaje natural (primera parte: *intra* → *inter*)

En este anexo se presentan las descripciones en lenguaje natural escritas por los trece estudiantes participantes al finalizar la primera parte del estudio (pasaje de la etapa *intra* a la etapa *inter*) junto con un análisis detallado de cada una. En cada análisis, se usan cursivas para citar partes concretas de sus descripciones. Se tomaron fotografías de las hojas de papel con las descripciones originales escritas por los estudiantes al finalizar las entrevistas, las cuales fueron transcritas aquí.

Aaron

Para encontrar la cédula deseada busqué o recorrí puerta por puerta siguiendo un orden hasta encontrarla. De no encontrarla, paré luego de verificar todas las puertas.

En cuanto a las acciones realizadas, Aaron nunca dice (ni sugiere) que compara el documento buscado con el que se encuentra en cada puerta visitada. Tampoco expresa claramente que avanza hacia la siguiente puerta, aunque sugiere que recorre la hilera de manera secuencial (*recorrí puerta por puerta siguiendo un orden*). No aclara que repite el proceso en cada puerta.

Respecto a los cambios en los objetos, no dice expresamente que detiene la búsqueda en caso de encontrar el documento buscado. Dice *hasta encontrarla* pero no menciona expresamente la detención, la sugiere de forma implícita. Sí hace explícita la detención en caso de haber pasado por todas las puertas (*De no encontrarla, paré luego de verificar todas las puertas*).

Ignacio D

Para resolver el problema debemos golpear una a una las puertas, preguntando si en esa puerta vive la persona con el determinado número de cédula. Debemos repetir este procedimiento hasta que encontremos a la persona o hasta que se acaben las puertas.

En cuanto a las acciones realizadas, Ignacio D expresa claramente que compara el documento buscado con el que está en cada puerta por la que pasa (*preguntando si en esa puerta vive la persona con el determinado número de cédula*). Lo que no dice expresamente es que avanza hacia la siguiente puerta recorriendo la hilera de manera

secuencial, lo más que dice es *debemos golpear una a una las puertas*. Sí hace explícito que repite el proceso en las siguientes puertas (*Debemos repetir este procedimiento*).

Respecto a los cambios en los objetos, nunca dice expresamente que detiene la búsqueda. No lo hace en caso de encontrar el documento buscado, lo deja implícito (*hasta que encontremos a la persona*). Lo mismo sucede en caso de haber pasado por todas las puertas (*hasta que se acaben las puertas*).

Ignacio U

Avanzo en orden preguntando primero en la puerta 1. Si la cédula es la que busco, paro ahí. Caso contrario, sigo a la 2 y así sucesivamente hasta la puerta 7. Si la cédula no estaba en ninguna, significa que la persona no vivía en la calle.

Ignacio U da una descripción general del algoritmo, aunque atado a la cantidad específica de 7 puertas correspondiente a la instancia concreta resuelta en la acción.

En cuanto a las acciones realizadas, aclara expresamente que pregunta por el documento en la primera puerta y que lo compara con el que busca (*preguntando primero en la puerta 1* y *Si la cédula es la que busco*) pero no hace explícito que vuelve a comparar en las restantes puertas. Lo mismo sucede para el avance hacia la siguiente puerta, solamente lo explicita cuando está en la primera puerta (*sigo a la 2*). No lo dice expresamente en puertas posteriores, aunque sugiere que recorre la hilera de manera secuencial (*Avanzo en orden* y *así sucesivamente hasta la puerta 7*). No hace explícita la repetición del proceso en las restantes puertas.

Respecto a los cambios en los objetos, dice expresamente que detiene la búsqueda si encuentra el documento en la primera puerta (*preguntando primero en la puerta 1* y *Si la cédula es la que busco, paro ahí*) pero no lo dice en caso de que lo encuentre en alguna de las demás puertas. Tampoco dice expresamente que para en caso de acabarse las puertas, sino que lo deja implícito (*así sucesivamente hasta la puerta 7* y *Si la cédula no estaba en ninguna, significa que la persona no vivía en la calle*).

Joaquín

Evalúo la primera puerta verificando si ésta contiene el número buscado, si coincide finalizo la búsqueda. De lo contrario golpeo la siguiente puerta, así hasta que o bien se encuentre el número buscado o se acaben las puertas.

En cuanto a las acciones realizadas, Joaquín dice expresamente que compara el documento en la primera puerta con el que busca (*Evalúo la primera puerta verificando si ésta contiene el número buscado*) pero no lo hace explícito para las restantes puertas. Lo mismo sucede para el avance hacia la siguiente puerta, solamente lo explicita cuando está en la primera puerta (*De lo contrario golpeo la siguiente puerta*) pero no cuando está en alguna puerta posterior. No menciona claramente que recorre el resto de la hilera de manera secuencial ni que repite el proceso en las restantes puertas (lo más que dice es *así hasta que*).

Respecto a los cambios en los objetos, dice expresamente que detiene la búsqueda si encuentra el documento en la primera puerta (*Evalúo la primera puerta verificando si ésta contiene el número buscado, si coincide finalizo la búsqueda*) pero no lo dice en caso de que lo encuentre en alguna de las demás puertas, lo deja implícito (*hasta que o bien se encuentre*). Lo mismo ocurre con la segunda condición de parada (*hasta que… o se acaben las puertas*).

Juan

Comienzo a recorrer las puertas comenzando por la que tengo más cerca. Toco en la primera, pregunto por el número de cédula, si no es el que busco, sigo con la siguiente, y así sucesivamente hasta encontrarlo. En caso de no encontrarlo, recorro todas las puertas.

En cuanto a las acciones realizadas, Juan aclara expresamente que pregunta por el documento en la primera puerta y que lo compara con el que busca (*Toco en la primera, pregunto por el número de cédula, si no es el que busco*) pero no lo aclara para las restantes puertas. Lo mismo sucede para el avance hacia la siguiente puerta, solamente lo explicita (*sigo con la siguiente*) cuando está en la primera puerta, pero no cuando está en alguna puerta posterior. Sugiere que recorre la hilera de puertas de manera secuencial (*sigo con la siguiente, y así sucesivamente*) pero no hace explícito que repite el proceso en las restantes puertas.

Respecto a los cambios en los objetos, nunca dice expresamente que detiene la búsqueda. No lo hace en caso de encontrar el documento buscado (*si no es el que busco, sigo con la siguiente*), apenas sugiere implícitamente que lo hace (*hasta encontrarlo*). Hace lo mismo en caso de haber pasado por todas las puertas (*En caso de no encontrarlo, recorro todas las puertas*). Su descripción expresa que continúa buscando en caso de no darse ninguna de las dos condiciones de parada, pero no hace explícita la detención en caso de que sí se cumpla cualquiera de ellas.

Martín

Comienzo golpeando sobre la puerta número uno y preguntando por la persona que busco. En caso de haberla encontrado, entro a la casa y termino ahí mi objetivo. En caso contrario, continúo hasta recorrer el número finito de casas y detenerme donde se encuentra la persona que busco. En caso de recorrer todas las casas y no encontrar a la persona, mi búsqueda finaliza puesto que no se encontraba en ninguna de las casas.

En cuanto a las acciones realizadas, Martín aclara expresamente que pregunta por el documento en la primera puerta y que lo compara con el que busca (*golpeando sobre la puerta número uno y preguntando por la persona que busco. En caso de haberla encontrado*) pero no lo aclara para las restantes puertas. Nunca hace explícito el avance a la siguiente puerta ni que recorre la hilera de manera secuencial (lo más que dice es *continúo hasta recorrer el número finito de casas*). Tampoco expresa que repite el proceso en las restantes puertas.

Respecto a los cambios en los objetos, dice expresamente que detiene la búsqueda, tanto si encuentra el documento en la primera puerta (*golpeando sobre la puerta número uno y preguntando por la persona que busco. En caso de haberla encontrado, entro a la casa y termino ahí mi objetivo*) como en alguna de las demás (*y detenerme donde se encuentra la persona que busco*). Lo mismo ocurre con la segunda condición de parada (*mi búsqueda finaliza puesto que no se encontraba en ninguna de las casas*).

Mónica

Golpeo la puerta nº o, si encuentro la cédula termino. Sino golpeo en las siguientes puertas hasta encontrar la cédula, puede pasar que no se encuentre la cédula por lo que voy a finalizar cuando termine de recorrer todas las puertas.

En cuanto a las acciones realizadas, Mónica no dice expresamente que compara el documento buscado con el de cada puerta por la que pasa, lo da como algo implícito (*si encuentro la cédula* y *puede pasar que no se encuentre la cédula*). Tampoco hace explícito el avance a la siguiente puerta ni que recorre el resto de la hilera de manera secuencial (lo más que dice, tras visitar la primera puerta, es *golpeo en las siguientes puertas*, pero sin aclarar por cuál de las siguientes continúa). Tampoco expresa que repite el proceso en las restantes puertas.

Respecto a los cambios en los objetos, dice expresamente que detiene la búsqueda si encuentra el documento en la primera puerta (*Golpeo la puerta nº o, si encuentro la cédula termino*) pero no lo dice en caso de que lo encuentre en alguna de las siguientes puertas. Sí expresa claramente la segunda condición de parada (*voy a finalizar cuando termine de recorrer todas las puertas*).

Nicolás

Suponiendo que se quiere encontrar la persona con cédula x, se recorren las puertas en orden, abriéndolas y preguntando a la persona detrás de ellas su número de cédula. Si este es el que estábamos buscando, se finaliza la búsqueda, Sino, pasamos a la siguiente puerta. Si se recorren todas las puertas y no se encuentra a la persona de cédula x, se deduce que ésta no reside detrás de las puertas estudiadas o no existe.

En cuanto a las acciones realizadas, Nicolás dice expresamente que pregunta por el documento en cada puerta y que lo compara con el que busca (*preguntando a la persona detrás de ellas su número de cédula. Si este es el que estábamos buscando*). También expresa que avanza hacia la siguiente puerta (*pasamos a la siguiente puerta*) y que recorre la hilera de puertas de manera secuencial (*se recorren las puertas en orden*). Además, deja en claro que repite el proceso en cada una (describe todas las acciones refiriéndose a *las puertas*, en plural).

Respecto a los cambios en los objetos, expresa claramente que detiene la búsqueda si encuentra el documento en alguna puerta (*Si este es el que estábamos buscando, se finaliza la búsqueda*) pero no lo dice expresamente en caso de que se acaben las

puertas, sino que lo deja implícito (*Si se recorren todas las puertas… se deduce que ésta no reside detrás de las puertas estudiadas o no existe*).

Pablo M

1) *Llego a la cuadra*
2) *Me acerco a la puerta más cercana*
3) *Golpeo la puerta*
4) *Consulto con la persona si fulanito (con la cédula buscada) vive en esa casa*
5a) *En caso de que sí viva, terminé mi búsqueda*
5b) *Si fulanito no vive en esa casa, me acerco a la siguiente en orden de cercanía*
6) *Repito el proceso anterior*
  a) *hasta encontrar a fulanito*
  b) *hasta que no queden casas en la cuadra (marco que fulanito no vive en esa cuadra)*

En cuanto a las acciones realizadas, Pablo M expresa claramente que compara el documento buscado con el que se encuentra en cada puerta por la que pasa (*Consulto con la persona si fulanito (con la cédula buscada) vive en esa casa*). Si bien da a entender que podría vivir más de una persona (la consigna inicial era que en cada casa vive una sola persona), esto no afecta su grado de conceptualización de las acciones realizadas. También expresa que avanza hacia la siguiente puerta, deja en claro que siempre recorre la hilera de puertas de manera secuencial (*me acerco a la siguiente en orden de cercanía*) y que repite el proceso en cada una (*Repito el proceso anterior*).

Respecto a los cambios en los objetos, hace explícito que detiene la búsqueda cuando se da cualquiera de las dos condiciones de parada (*En caso de que sí viva, terminé mi búsqueda* y *hasta que no queden casas en la cuadra (marco que fulanito no vive…)*).

Pablo P

Primero preguntamos en la primer puerta si la persona tiene el número de cédula que buscamos. Como no fue el caso continuamos con la siguiente puerta y volvemos a preguntar si la persona tiene el número de cédula que buscamos. Como la respuesta es negativa, continuamos a la siguiente. Repetimos el proceso de preguntar si la persona tiene el número de cédula que buscamos. Como la respuesta es afirmativa, detenemos la búsqueda ya que no hay incentivos para continuarla.

Pablo P describe una instancia concreta del algoritmo en lugar de dar una descripción general del mismo. Concretamente, el caso en que el documento buscado se encuentra en la tercera puerta.

Como consecuencia, expresa las acciones realizadas atadas a su instancia concreta. Lo hace tanto para la comparación entre el documento buscado y el de la puerta actual (*preguntamos en la primer puerta si la persona tiene el número de cédula*, *volvemos a preguntar* y *repetimos el proceso de preguntar… si la persona tiene el número de cédula que buscamos*) como para el avance hacia la siguiente puerta (*continuamos con la siguiente* y *continuamos a la siguiente*). Expresa claramente que recorre la hilera de

puertas de manera secuencial y la repetición del proceso, a pesar de hacerlo para el caso de su instancia concreta (*Como no fue el caso continuamos con la siguiente puerta*, *Como la respuesta es negativa, continuamos a la siguiente* y *Repetimos el proceso de preguntar si la persona tiene el número de cédula que buscamos*).

Respecto a los cambios en los objetos, solamente expresa uno de ellos (detener la búsqueda cuando encuentra el documento buscado), que es el que se da en su instancia concreta (*Como la respuesta es afirmativa, detenemos la búsqueda ya que no hay incentivos para continuarla*). Nunca expresa el otro cambio posible (parar cuando no quedan más puertas), pues no ocurre en la instancia concreta que describe.

Tomás

Me dirijo a la primer puerta y busco el número deseado. Si encuentro el número, selecciono esa puerta y no me fijo en el resto. Si no la encuentro paso a la siguiente y repito el mismo proceso hasta encontrar el número. En caso de no encontrarlo, al finalizar el paso por todas las puertas me retiro.

En cuanto a las acciones realizadas, Tomás no dice expresamente que compara el documento buscado con el de cada puerta por la que pasa, sino que lo deja implícito (*busco el número deseado* y *Si encuentro el número*). Sí hace explícito el avance hacia la siguiente puerta (*paso a la siguiente*). Deja en claro que recorre la hilera de puertas de manera secuencial y que repite el proceso (*repito el mismo proceso*).

Respecto a los cambios en los objetos, dice expresamente que detiene la búsqueda si encuentra el documento en la primera puerta (*Me dirijo a la primer puerta* y *Si encuentro el número, selecciono esa puerta y no me fijo en el resto*). De encontrarlo en alguna de las siguientes, lo sugiere pero no lo dice expresamente (*repito el mismo proceso hasta encontrar el número*). Sí expresa claramente la segunda condición de parada (*En caso de no encontrarlo, al finalizar el paso por todas las puertas me retiro*).

Valeria

Con el número de cédula, golpeo cada puerta preguntando por el nº de cédula del que vive ahí. Pueden ocurrir 2 casos:
1) *que encuentre en alguna puerta el nº de cédula (por lo que dejo de preguntar en las restantes)*
2) *no encuentre coincidencia de cédula (por lo que recorrí todas las puertas)*

En cuanto a las acciones realizadas, Valeria no dice expresamente que compara el documento buscado con el que se encuentra en cada puerta visitada, sino que lo deja implícito (*preguntando por el nº de cédula del que vive ahí*). Tampoco hace explícito que avanza hacia la siguiente puerta ni que recorre la hilera de puertas de manera secuencial, lo más que dice es *golpeo cada puerta preguntando*). Tampoco dice expresamente que repite el proceso en cada puerta visitada.

Respecto a los cambios en los objetos, explicita la detención por ambas condiciones (*que encuentre en alguna puerta el nº de cédula (por lo que dejo de preguntar en las restantes)* y *no encuentre coincidencia de cédula (por lo que recorrí todas las puertas)*).

Ximena

Dada una cierta cantidad de puertas, verifico una por una hasta encontrar la cédula que estoy buscando. Empiezo buscando la cédula en la puerta cero. Veo si la cédula coincide con la que tengo. Si coincide, terminé de buscar. Si no coincide, sigo recorriendo puertas hasta encontrarla o hasta que no queden más puertas.

En cuanto a las acciones realizadas, Ximena dice expresamente que pregunta por el documento en la primera puerta y que lo compara con el que busca (*Empiezo buscando la cédula en la puerta cero. Veo si la cédula coincide con la que tengo*) pero no lo dice para las demás puertas (lo más que dice es *verifico una por una*). No hace explícito que avanza hacia la siguiente puerta ni que recorre la hilera de puertas de manera secuencial, lo más que dice es *sigo recorriendo puertas*. Tampoco hace explícito que repite el proceso en las demás puertas.

Respecto a los cambios en los objetos, dice expresamente que detiene la búsqueda si encuentra el documento en la primera puerta (*Empiezo buscando la cédula en la puerta cero* y *Si coincide, terminé de buscar*) pero no lo dice en caso de que lo encuentre en alguna de las siguientes puertas (solo dice *sigo recorriendo puertas hasta encontrarla*). Tampoco explicita la detención tras haber pasado por todas las puertas, lo más que dice es *hasta que no queden más puertas*.

Anexo B

Pseudocódigo para búsqueda lineal (segunda parte: *inter* → *trans*)

En este anexo se presentan las versiones en *pseudocódigo* escritas por los trece estudiantes para el algoritmo de búsqueda lineal durante la primera actividad de la segunda parte del estudio (pasaje de la etapa *inter* a la etapa *trans*). Se presentan *todas* las versiones dadas por cada estudiante junto con un análisis detallado de cada una y del proceso realizado. En cada análisis, se usan cursivas entre comillas para citar respuestas orales de los estudiantes durante las entrevistas y cursivas sin comillas para citar partes concretas escritas en sus versiones en pseudocódigo y en sus descripciones iniciales en lenguaje natural (dadas en la primera parte). Las entrevistas fueron grabadas y se tomaron fotografías de las hojas de papel conteniendo las distintas versiones en pseudocódigo escritas por los estudiantes, que fueron transcritas aquí.

Aaron

Su primera versión del algoritmo es la siguiente:

Mientras (*No encuentre la C.I*)
    Ver si la C.I de esa puerta es la que busco
    Si es la C.I buscada paro
    Sino sigo verificando la siguiente
    Fin
Fin

En esta versión, Aaron refleja solo uno de los dos cambios posibles (parar al encontrar el documento) y expresa la condición en forma acorde a la semántica de *Mientras* (continúa iterando mientras se cumple que no encuentra la cédula). No toma en cuenta el otro (parar cuando se acaban las puertas), a pesar de haberlo incluido en su descripción al terminar la primera parte (*De no encontrarla, paré luego de verificar todas las puertas*). Su pensamiento necesita avanzar en la transición hacia *newC*'. Debe comprender la necesidad de instruir al agente externo para detenerse en ese caso.

Al recurrir a la automatización, cambia de opinión sobre la marcha y manifiesta su intención de escribir una segunda versión "*más cortita*" y que revise la primera cédula antes de comenzar a iterar. Dice "*podría haber cambiado las cosas... este ver que está acá...*" (refiriéndose a *Ver si la C.I de esa puerta es la que busco*) "*...haberlo puesto antes*

del Mientras para saber si entro o no porque puede ser que lo encuentre a la primera… este Si, también, podría cambiarlo".

Ver la C.I de la puerta 1
Mientras (*No sea la que busco*)
    Ver siguiente C.I
Fin

Su segunda versión efectivamente es más compacta que la primera, dado que deja de usar la estructura *Si* para la comparación y la unifica con la condición de la estructura *Mientras*. Conforme a lo que manifestó, separa el caso de ver la primera cédula antes de comenzar la iteración. De todas formas, aún mantiene el error de la primera versión, pues sigue tomando en cuenta solamente la misma condición de parada. Se le pide volver a la automatización para buscar un documento que no está en la hilera. Cuando llega al paso *Ver siguiente C.I* tras haber visitado la última puerta, detecta el error. Al preguntarle al respecto, contesta "*no hay más*". En este momento, toma conciencia de la necesidad de incluir la otra condición de parada. Para ello, se introduce en este punto la utilidad de usar operadores booleanos para combinar condiciones.

Ver la C.I de la puerta 1
Mientras NOT (*sea la que busco*) *OR* (*No tengo mas Puertas*)
    Veo siguiente C.I
Fin

En su tercera versión, mantiene el estilo más compacto de la segunda (revisando la primera cédula antes de la iteración). Presenta cierta ambigüedad en el manejo de la sintaxis del operador *NOT*. En la primera condición, lo utiliza expresamente como un operador que antecede a la condición, mientras que en la segunda lo escribe como parte de la misma (mediante la palabra *No*). De todas formas, no afecta la semántica de la negación ni su comprensión por parte del estudiante. Por otra parte, ahora intenta contemplar la segunda condición de parada, pero usa un operador incorrecto (*OR*). Ha avanzado en la transición hacia *newC*', pero hace uso de la disyunción, como lo haría en lenguaje natural. El uso de *OR* junto con su segunda condición muestra que está pensando en lo que debe cumplirse para *parar* de iterar. Precisa reflexionar sobre la relación entre la semántica de los operadores booleanos y de la estructura *Mientras*, a efectos de expresar la detención en forma acorde (debe hacerlo de manera que la iteración finaliza cuando alguna de las condiciones *deja* de cumplirse).

Se le pide nuevamente volver a la automatización para buscar un documento que no está en la hilera y detecta el error tras haber visitado la última puerta. Dice "*va a llegar a la última puerta, no va a ser la que yo busco y va a seguir buscando*". El estudiante toma conciencia de que debe expresar las condiciones de forma que la iteración pare cuando alguna *deja* de cumplirse, por lo que cambia *OR* por *AND* y quita la negación de la segunda condición.

Ver la C.I de la puerta 1
Mientras NOT (*sea la que busco*) *AND* (*tengo mas Puertas*)
    Veo siguiente C.I
Fin

Finalmente, su cuarta versión es correcta. Expresa cambios en objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse), y las acciones realizadas: comparación (*sea la que busco*), avance (*Veo siguiente C.I*) y repetición (*Mientras*). Estructura su algoritmo de forma que inicia posicionado frente a la primera puerta. Consulta el primer documento antes de iniciar la iteración y queda posicionado en la última puerta visitada al finalizar la iteración, tanto si encuentra en ella el documento como si no está en la hilera (estrategia de búsqueda a *priori*).

Ignacio D

Su primera versión del algoritmo es la siguiente:

Mientras no haya encontrado a la persona OR no se hayan acabado las puertas
    Pregunto en la puerta que estoy quien vive
    Avanzo a la siguiente puerta
Fin

En esta versión, Ignacio D usa la palabra *no* como parte de cada condición, en vez de usar el operador *NOT* que antecede a la condición. A pesar de eso, es consistente en su uso en ambas y no afecta la semántica. Ya en su primera versión intenta reflejar los dos posibles cambios en los objetos, pero lo hace usando un operador incorrecto (*OR*). Comparado con otros estudiantes, su pensamiento ya inicia más próximo a *newC*' al contemplar ambas condiciones y querer expresarlas de modo que para de iterar cuando alguna *deja* de cumplirse (pone *no* en ambas). El problema es que hace uso de la disyunción, como lo haría en lenguaje natural. Quizás sea una dificultad en la conceptualización de la semántica de dicho operador booleano, más que de los cambios, dado que las condiciones en sí están bien expresadas.

Se le pide volver a la automatización para buscar un documento que está en la tercera puerta. Al hacerlo, detecta que continúa buscando luego de haberlo encontrado. Dice "*va a seguir haciendo los pasos y queremos que pare de hacerlos… cambiar el OR por un AND*". En este punto, toma conciencia de la necesidad de cambiar de operador.

Mientras (*no haya encontrado a la persona*) *AND* (*no se hayan acabado las puertas*)
    Pregunto en la puerta que estoy quien vive
    Avanzo a la siguiente puerta
Fin

Su segunda versión es correcta, en el sentido de que resuelve el problema. Expresa los cambios en los objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse) pero, en cuanto a las acciones, solamente explicita el avance (*Avanzo a la siguiente puerta*) y la repetición (*Mientras*), dejando implícita la

comparación. Asume que una respuesta a *pregunto en la puerta que estoy quien vive* implica obtener el documento a comparar y supone implícita la comparación en la primera condición de la estructura *Mientras*. Si bien su transición hacia *newC* es menor que para otros estudiantes (que sí explicitan las tres acciones) esto no es un problema en términos del formalismo intermedio (*pseudocódigo*), ya que igual es lo bastante claro para ser interpretado por una persona haciendo de robot imaginario en vez de una computadora, por lo que no se le pide una nueva versión. Tendrá oportunidad de profundizar en dicha transición en la tercera actividad (escritura del programa).

Por otra parte, siempre avanza una vez más hacia la siguiente puerta, incluso enseguida de haber encontrado el documento. Esto implica un paso más de lo necesario, pero no es un problema para la solución, ya que igualmente detiene la búsqueda en la siguiente evaluación de la primera condición de *Mientras*. Estructura su algoritmo de tal forma que inicia posicionado frente a la primera puerta y siempre hace la comparación dentro de la iteración (incluso en la primera puerta). Tras visitar la última, la acción *avanzo a la siguiente puerta* no implica riesgo de consultar por un documento en una puerta inexistente, pues lo controla en la siguiente evaluación de la segunda condición de la estructura *Mientras* (estrategia de búsqueda a *posteriori*).

Ignacio U

Su primera versión del algoritmo es la siguiente:

```
nocedula := FALSE
contador := 1
Mientras (nocedula = FALSE)
        Analizo si en puerta 1 está la cédula
        Si (no está)
                contador := contador + 1
        Fin
        Si (contador = 7)
                nocedula := TRUE
        Fin
Fin
```

Ignacio U ya incorpora algunos elementos del lenguaje de programación en su pseudocódigo. Concretamente, uso de variables, operador de asignación y expresiones aritméticas y booleanas. Dado que esto no contradice las reglas sintácticas dadas para la escritura en pseudocódigo, se deja que use dichos elementos. Inicializa las variables correctamente y opera con ellas de forma adecuada. Como su descripción en lenguaje natural (primera parte), su pseudocódigo constituye una descripción del algoritmo que busca un documento cualquiera, pero atado a la cantidad específica de 7 puertas de la instancia concreta manipulada durante la automatización.

La primera versión presenta varios problemas. Uno de ellos es que, en cada entrada a la iteración, siempre compara el documento de la primera puerta, sin importar en cuál se encuentre durante la recorrida. Su pensamiento necesita transitar hacia *newC*, a

efectos de comprender que, en cada entrada, la comparación debe hacerse con el documento de una nueva puerta. Al volver a la automatización, detecta el problema. Tras visitar la primera puerta dice "*siempre está preguntando por la 1 nomás*". Consultado acerca de qué puerta debería consultar, contesta "*por contador*". En este punto, toma conciencia de la necesidad de variar el documento consultado.

```
nocedula := FALSE
contador := 1
Mientras (nocedula = FALSE)
        Analizo si en puerta contador está la cédula
        Si (no está)
                contador := contador + 1
        Fin
        Si (contador = 7)
                nocedula := TRUE
        Fin
Fin
```

Su segunda versión cambia *1* por *contador* en el primer paso de la iteración, pero mantiene los demás problemas. Otro de ellos es que refleja solamente uno de los dos cambios posibles (parar cuando se terminan las puertas, lo cual hace cuando *contador* llega a 7), sin tomar en cuenta el otro (parar al encontrar el documento buscado). Su pensamiento necesita avanzar en la transición hacia *newC*'. Debe comprender la necesidad de instruir al agente externo para detenerse en ese caso. Quizás puede deberse a una conceptualización incompleta en la primera parte del estudio, donde nunca dijo expresamente que debía detener la búsqueda en ese caso (lo más que puso fue *Si la cédula no estaba en ninguna, significa que la persona no vivía en la calle,* sin expresar la detención en caso de encontrarla). El nombre que da a su variable booleana (*nocedula*) sugiere alguna intención de parar al encontrar el documento, pero el comportamiento expresado por los pasos que escribe no conduce a eso.

Al volver a la automatización, detecta el problema. Se le pide buscar un documento que está en una puerta que no es la última. Al encontrarlo, nunca corta la iteración y queda en un *loop infinito*, posicionado en la puerta donde encuentra el documento. Consultado al respecto, dice: "*queda en un bucle*". En este punto, toma conciencia de la necesidad de parar al encontrarlo.

```
nocedula := FALSE
contador := 1
Mientras (nocedula = FALSE)
        Analizo si en puerta contador está la cédula
        Si (no está)
                contador := contador + 1
        Sino
                nocedula := TRUE
        Fin
```

```
        Si (contador = 7)
                nocedula := TRUE
        Fin
Fin
```

Su tercera versión corrige el problema anterior (ahora detiene la búsqueda cuando encuentra el documento, asignando *TRUE* a *nocedula*), pero mantiene otro: deja sin consultar el documento de la última puerta (*problema de borde*). La instancia concreta usada en la acción tiene 7 puertas. Cuando el documento buscado no está en ninguna de las 6 primeras, la variable *contador* toma el valor 7. Enseguida, asigna *TRUE* a *nocedula* y se detiene, sin llegar a consultar el documento en la séptima. Se le pide volver a la automatización para buscar un documento que está en la última y detecta el problema. Dice "*me quedó una sin analizar*". Además, pretende usar la variable *nocedula* tanto para detenerse tanto cuando encuentra el documento como cuando se acaban las puertas. Su pensamiento necesita seguir transitando hacia *newC*'. Ahora es consciente de las dos posibles razones por las cuales debe parar, pero necesita expresar con más claridad por cuál de ellas es que lo hace en cada caso. Para ello, se introduce en este punto la utilidad de usar operadores booleanos para combinar condiciones.

```
nocedula := FALSE
contador := 1
Mientras (nocedula = FALSE) OR (contador < 7)
        Analizo si en puerta contador está la cédula
        Si (no está)
                contador := contador + 1
        Sino
                nocedula := TRUE
        Fin
Fin
```

Su cuarta versión elimina la segunda estructura *Si* dentro de la iteración (que era donde pretendía controlar la detención al terminarse las puertas) y ahora intentar contemplar los dos cambios posibles, usando una condición compuesta en la estructura *Mientras*. El problema es que combina las condiciones de forma incorrecta, usando *OR*. Ha avanzado en la transición hacia *newC*', pero hace uso de la disyunción, como haría en lenguaje natural. Quizás sea una dificultad en conceptualizar la semántica de dicho operador booleano, más que de los cambios, dado que las condiciones en sí están bien expresadas, en el sentido de que expresan lo que debe cumplirse para *seguir* iterando.

Tras volver a la automatización para buscar un documento que está en una puerta anterior a la última, detecta que sigue recorriendo tras haberlo encontrado, pues usa *OR* y la segunda condición se sigue cumpliendo. Dice "*acá va a entrar igual porque el contador no está en 7*".

```
nocedula := FALSE
contador := 1
Mientras (nocedula = FALSE) AND (contador = 7)
        Analizo si en puerta contador está la cédula
        Si (no está)
                contador := contador + 1
        Sino
                nocedula := TRUE
        Fin
Fin
```

Su quinta versión cambia el uso de *OR* por *AND* y modifica la segunda condición. Toma conciencia de que debe expresar las condiciones de forma que la iteración pare cuando alguna *deja* de cumplirse, pero se confunde al modificar la segunda condición. Esto introduce un nuevo problema, pues genera que no itera ni siquiera una vez. Al volver a la automatización, constata este error y dice "*ya no entró... era NOT, capaz que tendría que haber puesto un NOT*".

```
nocedula := FALSE
contador := 1
Mientras (nocedula = FALSE) AND NOT (contador = 7)
        Analizo si en puerta contador está la cédula
        Si (no está)
                contador := contador + 1
        Sino
                nocedula := TRUE
        Fin
Fin
```

En su sexta versión corrige este problema, utilizando correctamente los operadores *AND* y *NOT* para combinar adecuadamente ambas condiciones y parar cuando alguna deja de cumplirse. Sin embargo, todavía mantiene el problema de borde detectado en la tercera versión: deja sin consultar el documento en la última puerta. Tras volver a la automatización por última vez, al llegar a la última puerta y constatar que sigue sin consultar el documento en ella, dice "*era 8*".

```
nocedula := FALSE
contador := 1
Mientras (nocedula = FALSE) AND NOT (contador = 8)
        Analizo si en puerta contador está la cédula
        Si (no está)
                contador := contador + 1
        Sino
                nocedula := TRUE
        Fin
Fin
```

Finalmente, su séptima versión es correcta, en el sentido de que resuelve el problema, pero no lo hace para cualquier cantidad de puertas, sino para la cantidad puntual de su instancia concreta. No se le menciona nada, pues la generalización para cualquier cantidad se trabaja luego en la segunda y tercera actividad. Expresa cambios en objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse) y hace explícitas las acciones: comparación (*Analizo si en puerta contador está la cédula*), avance (*contador := contador + 1*) y repetición (*Mientras*). A diferencia de otros estudiantes, Ignacio U incorpora elementos del lenguaje de programación, lo cual puede generar la aparición de problemas propios de la ejecución en computadora. Por ejemplo, el *problema de borde* surgido de usar 7 en vez de 8. Por otro lado, el uso de esos mismos elementos hace que las acciones sean aún más explícitas, por lo cual su transición hacia *newC* en este sentido es mayor que para otros estudiantes. Estructura su algoritmo de forma que inicia posicionado frente a la primera puerta y siempre hace la comparación dentro de la iteración (incluso en la primera). Tras visitar la última, la acción *contador := contador + 1* no implica riesgo de consultar por un documento en una puerta inexistente, pues lo controla en la siguiente evaluación de la segunda condición de *Mientras* (estrategia de búsqueda a *posteriori*).

Joaquín

Su primera versión del algoritmo es la siguiente:

```
Mientras (puerta <> número)
        puerta := 1 + puerta
Fin
```

Joaquín ya incorpora algunos elementos del lenguaje de programación en su pseudocódigo (uso de variables, operador de asignación, expresiones aritméticas y booleanas). Dado que no contradice las reglas sintácticas dadas para pseudocódigo, se deja que use dichos elementos. Sin embargo, presenta algunos problemas por esto. El primero es que, ni bien termina de escribir, detecta que no inicializa la variable *puerta*, no dejando claro que debe empezar por la primera. Su incorporación en esta etapa de elementos propios del lenguaje formal lo lleva a la necesidad de conceptualizar tempranamente aspectos de la ejecución en máquina, como la importancia de inicializar variables. Se recurre a la automatización y toma conciencia de esto ni bien empieza. Dice "*tendría que haber definido puerta antes… arrancaría en 1*".

```
puerta := 1
Mientras (puerta <> número)
        puerta := 1 + puerta
Fin
```

En su segunda versión, compara el número de puerta con el número de documento buscado, en vez de hacer una comparación entre documentos. Necesita conceptualizar la diferencia entre el identificador de la puerta y el número de documento contenido

en ella. Detecta esto al volver a la automatización y manifiesta la necesidad de obtener el documento en dicha puerta. Dice "*nunca la abrí*" (refiriéndose a la puerta).

```
puerta := 1
abrir puerta := x
número := 417
Mientras (x <> número)
        puerta := puerta + 1
Fin
```

Su tercera versión intenta incorporar dos asignaciones: *abrir puerta := x* y *número := 417* antes de comenzar a iterar. En la primera pretende guardar en una nueva variable (x) el documento de la primera puerta. Tras escribirla, el propio estudiante observa que si se tratase del lenguaje de programación, sería necesario invertir el orden (*x := abrir puerta*). Dice "*pero acá va al revés*". Por ser pseudocódigo, se deja que lo mantenga así como está. En la segunda asignación, asigna el mismo documento de ejemplo usado al volver a la automatización (417). Su pseudocódigo no permite buscar un documento cualquiera, sino que está atado a su instancia concreta (en este caso, al documento concreto buscado). No se le menciona nada aún, ya que el salto hacia el caso general (para buscar un valor cualquiera) se trabaja luego en la tercera actividad. Por otra parte, decide invertir el orden de los operandos de la expresión *1 + puerta*, cambiando por puerta + 1 entre la segunda y la tercera versión. Dicha inversión no influye en el resultado, por lo que no se le menciona el tema.

Tras realizar los cambios anteriores, al volver a la automatización detecta que x siempre contiene el documento de la puerta 1, por lo que no varía a medida que avanza. Su pensamiento necesita transitar hacia *newC*, debiendo conceptualizar la importancia de obtener un documento nuevo en cada entrada a la iteración. Dice "*me faltó definir x*" (refiriéndose a la necesidad de asignarle un nuevo valor dentro de la iteración).

```
puerta := 1
abrir puerta := x
número := 417
Mientras (x <> número)
        puerta := puerta + 1
        abrir puerta := x
Fin
```

En su cuarta versión, agrega otra vez *abrir puerta := x*, pero ahora dentro de la iteración tras haber avanzado a la siguiente puerta, logrando de esta manera actualizar el valor de x con el documento de la próxima puerta en cada paso. Tras resolver este problema, ahora presenta otro: solamente considera uno de los dos cambios posibles (parar al encontrar el documento), sin tomar en cuenta el otro (parar cuando se terminan las puertas). Su pensamiento necesita avanzar en la transición hacia *newC'*. Debe comprender que también debe instruir al agente externo para parar en ese caso. Esto puede deberse a una conceptualización incompleta en la primera parte del

estudio, donde nunca dijo expresamente que debía detener la búsqueda en ese caso (lo dejó implícito al poner *hasta que o bien se encuentre el número buscado o se acaben las puertas*). Toma conciencia del problema tras volver a la automatización para buscar un documento que no está. Dice "*no hay más puertas… error*" tras haber visitado la última. En este punto, se introduce la utilidad de hacer uso de operadores booleanos para combinar condiciones.

```
puerta := 1
abrir puerta := x
número := 417
Mientras (x <> número) AND (no es la última puerta)
        puerta := puerta + 1
        abrir puerta := x
Fin
```

Finalmente, su quinta versión es correcta. Expresa los cambios en objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse) y las acciones realizadas: comparación (*x <> número*), repetición (*Mientras*) y avance (*puerta := puerta + 1*). No utiliza primero *OR*, sino que lo resuelve directamente con *AND*. A diferencia de otros estudiantes, muestra una conceptualización más sólida de la relación entre dicho operador y la semántica de *Mientras*. Expresa la negación como parte de la segunda condición usando *no*, en vez de hacer uso explícito del operador *NOT*, lo cual no afecta la semántica. Estructura su algoritmo de tal forma que inicia posicionado frente a la primera puerta. Consulta el primer documento antes de iniciar la iteración y queda posicionado en la última puerta visitada al finalizar la iteración, tanto si encuentra en ella el documento buscado como si el mismo no está en la hilera (estrategia de búsqueda a *priori*).

Juan

Su primera versión del algoritmo es la siguiente:

```
Mientras (no encuentro nro cedula)
        Golpeo puerta
        Pregunto por cedula
        Verifico si es la buscada
        Si es
                Paro
        Sino
                Termino
        Fin
Fin
```

Esta versión presenta varios problemas. Uno de ellos es que nunca avanza hacia la siguiente puerta, solamente inspecciona la primera. Su pensamiento necesita avanzar en la transición hacia *newC*. Debe conceptualizar la importancia de la acción de avance a la siguiente puerta. Tras volver a la automatización, detecta el problema: pretende

finalizar la iteración tanto si el documento se encuentra en la primera puerta ("*Paro*") como si no ("*Termino*"). Se le pregunta si eso es lo que quiere y dice " *no...seguir*".

Mientras (*no encuentro nro cedula*)
    Golpeo puerta
    Pregunto por cedula
    Verifico si es la buscada
    Si es
        Paro
    Sino
        Sigo a la siguiente puerta
    Fin
Fin

Su segunda versión corrige el problema pero mantiene otro: considera solo uno de los dos cambios posibles (parar al encontrar el documento), sin tomar en cuenta el otro (parar cuando no quedan puertas). Su pensamiento necesita avanzar en la transición hacia *newC*'. Debe comprender que necesita instruir al agente externo para detenerse en ese caso. Esto puede deberse a una conceptualización incompleta en la primera parte. Su descripción no dice expresamente que detiene la búsqueda, lo deja implícito (*En caso de no encontrarlo, recorro todas las puertas*). Detecta esto al volver a la automatización, al buscar un documento que no existe dice "*sigo a la siguiente, que no existe... no hay más puertas*" tras haber consultado el de la última. En este punto, se introduce la utilidad de usar operadores booleanos para combinar condiciones.

Mientras ((*no encuentro nro cedula*) *OR* (*no hay mas puertas*))
    Golpeo puerta
    Pregunto por cedula
    Verifico si es la buscada
    Si es
        Paro
    Sino
        Sigo a la siguiente puerta
    Fin
Fin

Su tercera versión intenta incluir ambas condiciones de parada, pero lo hace con *OR*, de modo incorrecto. Ha avanzado en la transición hacia *newC*', pero usa la disyunción, como lo haría en lenguaje natural. El uso de *OR* junto con su segunda condición muestra que está pensando en lo que debe cumplirse para *parar* de iterar. Precisa reflexionar sobre la relación entre la semántica de los operadores booleanos y de la estructura *Mientras*, a efectos de expresar la detención en forma acorde (debe hacerlo de manera que la iteración finaliza cuando alguna de las condiciones *deja* de cumplirse). Además, usa la palabra *no* en cada condición, en vez de usar expresamente el operador *NOT*, lo cual no afecta la semántica. Al volver a la automatización para buscar un documento que no existe, se le hace notar que la condición se sigue

cumpliendo tras la última puerta (debido al *OR*). Sin embargo, dice "*fin*" (piensa que termina de iterar). En realidad debería seguir, conforme a la semántica de la estructura *Mientras*. Se le hace notar eso y dice "*tengo que corregir la condición… te va a dar un error… cuando quiera entrar acá porque no tengo más puertas*".

Mientras ((*encuentro nro cedula*) *AND* (*hay puertas*))
    Golpeo puerta
    Pregunto por cedula
    Verifico si es la buscada
    Si es
        Paro
    Sino
        Sigo a la siguiente puerta
    Fin
Fin

En su cuarta versión cambia el uso de *OR* por *AND* y quita la negación en ambas condiciones. Toma conciencia de que debe expresar las condiciones de forma que la iteración pare cuando alguna *deja* de cumplirse. Entiende que existe una relación entre el cambio de operador y la negación de las condiciones involucradas, pero falla en hacer las modificaciones de forma que logren el comportamiento deseado (debía quitar la negación solamente en la segunda condición). Esto introduce un nuevo problema en la primera, pues genera que no itera ni siquiera una vez. Al volver a la automatización, constata el error y dice "*tengo que poner que no encontré la cédula*".

Mientras ((*no encuentro nro cedula*) *AND* (*hay puertas*))
    Golpeo puerta
    Pregunto por cedula
    Verifico si es la buscada
    Si es
        Paro
    Sino
        Sigo a la siguiente puerta
    Fin
Fin

Finalmente, su quinta versión es correcta. Expresa cambios en los objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse) y acciones realizadas: comparación (*Verifico si es la buscada*), repetición (*Mientras*) y avance (*Sigo a la siguiente puerta*). Estructura su algoritmo de tal forma que inicia posicionado frente a la primera puerta y siempre hace la comparación dentro de la iteración (incluso en la primera puerta). Tras visitar la última puerta, la acción *Sigo a la siguiente puerta* no implica riesgo de consultar por un documento en una puerta inexistente, pues lo controla en la siguiente evaluación de la segunda condición de la estructura *Mientras* (estrategia de búsqueda a *posteriori*).

Martín

Su primera versión del algoritmo es la siguiente:

1 - Conozco el número de la persona
2 - Voy a la primera puerta
Mientras (*el número de la persona dentro de la casa sea distinto del número que busco*)
    Paso 1 - Golpeo la puerta
    Paso 2 - Si (*el número de la persona es igual al mío*)
        Paso 1a - Entro
        Paso 2a - Termino mi búsqueda
    Sino
        Paso 1b - Voy a la siguiente puerta
    Fin
Fin

En esta versión, Martín numera algunos pasos de su algoritmo. Si bien repite los números 1 y 2, no le impide comprender el orden de realización. También compara dos veces el documento buscado con el de la persona en la puerta actual (una vez en la condición de la estructura *Si* y otra en la condición de la estructura *Mientras*). Si bien no constituye un error, resulta confuso. Su manejo de la primera puerta también es confuso, pues no la golpea antes de empezar a iterar, pero asume que ya conoce dicho documento (al volver a la automatización dice "*en este caso ya se abrió*"). El principal problema en esta versión es que contempla solo uno de los dos cambios posibles (parar al encontrar el documento), sin tomar en cuenta el otro (parar cuando se acaban las puertas), a pesar de haberlo incluido en su descripción al finalizar la primera parte (*mi búsqueda finaliza puesto que no se encontraba en ninguna de las casas*). Su pensamiento necesita avanzar en la transición hacia *newC'*. Se recurre a la automatización para buscar un documento que no está y detecta el problema. Dice "*no hay más puertas*" tras haber visitado la última. Debe conceptualizar que necesita instruir al agente externo para parar en ese caso. Para ello, se introduce en este punto la utilidad de usar operadores booleanos para combinar condiciones.

Conozco el número de la persona
Voy a la primera puerta
Golpeo la primera puerta
Mientras (*el número de la persona sea distinto del que busco*) *AND* (*haya mas puertas*)
    Paso 1 - Paso a la siguiente puerta
    Paso 2 - Golpeo la puerta
Fin

Su segunda versión es correcta. Expresa cambios en los objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse) y acciones: comparación (*el número de la persona sea distinto del que busco*), repetición (*Mientras*) y avance (*Paso a la siguiente puerta*). Reestructura su algoritmo de modo que ya no compara dos veces el documento. No utiliza primero *OR*, sino que lo resuelve directo con *AND*. A diferencia de otros estudiantes, muestra una conceptualización más sólida

de la relación entre dicho operador y la semántica de *Mientras*. Consulta el documento en la primera puerta antes de iniciar la iteración, quedando posicionado en la última puerta visitada al finalizar la misma, tanto si encuentra en ella el documento buscado como si no se encuentra en la hilera (estrategia de búsqueda a *priori*).

Mónica

Su primera versión del algoritmo es la siguiente:

```
Para cada puerta hacer
        Golpear puerta
        Preguntar C.I
        Si encuentro la C.I entonces
                Termino
        Sino
                Sigo preguntando
        Fin
Fin
```

Conforme a la semántica de *Para cada*, esta versión recorre la totalidad de puertas de la hilera, sin importar si el documento buscado está en ella o no. Quizás la estudiante no sea consciente de esto, ya que incluye el paso *Termino* dentro de la iteración, en caso de encontrarlo. Tras escribir esta versión y previo a recurrir a la automatización, se le pregunta cuántas puertas debería visitar su algoritmo y dice "*ocho*" (la cantidad de puertas de su instancia concreta) Su respuesta sugiere que realmente tiene intención de recorrerlas todas (a pesar de haber puesto *Termino*), acorde a la semántica de *Para cada*. Esto puede deberse a una conceptualización incompleta en la primera parte, ya que su descripción en lenguaje natural hace explícita la detención solo al encontrar el documento en la primera puerta (*Golpeo la puerta nº o, si encuentro la cédula termino*) pero no la explicita en caso de que lo encuentre en alguna de las siguientes puertas. Quizás realmente piensa que necesita recorrer la hilera hasta el final. Cualquiera sea el caso, necesita avanzar en la transición hacia *newC*' y comprender la importancia de instruir al agente externo para detenerse al encontrar el documento buscado, usando una estructura iterativa acorde a dicho propósito.

En la automatización, se procede a buscar un documento que está en la tercera puerta. Al llegar al paso *Termino*, tras encontrarlo, se da cuenta de que hay algo que no está bien. Se le pregunta cuántas puertas visitó y responde "*tres*". Se le hace notar que antes había dicho ocho puertas, a lo que responde "*entonces no es un para cada... debería ser un mientras*". Ha avanzado en la transición hacia *newC*', al tomar conciencia de la necesidad de expresar la detención cuando encuentra el documento, y hacerlo usando una estructura iterativa acorde (*Mientras*).

Mientras no encuentre la C.I hacer
Golpear puerta
Preguntar C.I
Si encuentro la C.I entonces
Termino
Sino
Sigo preguntando
Fin
Fin

En su segunda versión usa la palabra *no* dentro de la condición en vez de utilizar expresamente el operador *NOT*, lo cual no afecta la semántica. Tras cambiar a *Mientras*, su algoritmo ahora expresa la detención solo al encontrar el documento, sin tomar en cuenta el otro cambio posible (parar cuando no quedan más puertas). Su pensamiento necesita seguir transitando hacia *newC*', debe tomar en cuenta también la otra condición de parada. Se recurre nuevamente a la automatización dos veces, una para buscar un documento que está en la hilera y otra para buscar otro que no. En la primera vez, cuando encuentra el documento, al llegar al paso *termino*, dice: "*encuentro la cédula, entonces termino*" dando a entender que finaliza en ese instante. Se le hace notar que debe evaluar una vez más la condición de *Mientras*. Dice "*ah, ahí termino, claro*". Muestra una conceptualización incompleta de la semántica de dicha estructura, al omitir que necesita evaluar una vez más la condición antes de terminar. Piensa que alcanza con el paso *Termino* para expresar la detención. En la segunda vez, al llegar al paso *Sigo preguntando* tras haber visitado la última puerta dice "*acá ya no tengo más puertas para golpear*". En este punto, toma conciencia de la necesidad de incluir la otra condición de parada (cuando se acaban las puertas). Para ello, se introduce la utilidad de usar operadores booleanos para combinar condiciones.

Mientras (*no encuentre la C.I*) *AND* (*la puerta es menor o igual a 7*) *hacer*
Golpear puerta
Preguntar C.I
Si encuentro la C.I entonces
Termino
Sino
Sigo preguntando
Fin
Fin

Finalmente, su tercera versión es correcta, en el sentido de que resuelve el problema, pero no para cualquier cantidad de puertas, sino para la cantidad puntual de su instancia concreta. En este punto no se le menciona nada al respecto, pues la generalización para cualquier cantidad se trabaja luego en la segunda y tercera actividad. Expresa cambios en objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse). No intenta primero con *OR*, sino que lo resuelve directamente con *AND*. En este sentido, a diferencia de otros estudiantes, muestra una conceptualización más sólida de la relación entre dicho

operador y la semántica de *Mientras*. En cuanto a las acciones, solo expresa la repetición (*Mientras*), dejando implícitas las otras dos. No hace explícita la comparación, solo pone *Si encuentro la C.I*. Tampoco explicita el avance, sino que pone *Sigo preguntando* sin aclarar expresamente que lo hará en la siguiente puerta. Si bien su transición hacia *newC* es menor que en otros estudiantes (que sí expresan las tres acciones) esto no es un problema en términos del formalismo intermedio (*pseudocódigo*), ya que igual es lo bastante claro para ser interpretado por una persona haciendo de robot en vez de una computadora, por lo que no se le pide una nueva versión. Tendrá oportunidad de profundizar en dicha transición en la tercera actividad (escritura del programa). Estructura su algoritmo de tal forma que inicia posicionada frente a la primera puerta y siempre hace la comparación dentro de la iteración (incluso en la primera). Tras visitar la última, la acción *Sigo preguntando* no implica riesgo de consultar por un documento en una puerta inexistente, pues lo controla en la siguiente evaluación de la segunda condición de *Mientras* (estrategia de búsqueda a *posteriori*).

Nicolás

Su primera versión del algoritmo es la siguiente:

golpear la primera puerta
Mientras (*cédula es distinta de x*)
    golpear puerta siguiente
    preguntar cédula
Fin

Esta versión solo toma en cuenta uno de los dos cambios posibles (parar al encontrar el documento buscado) pero no el otro (parar cuando se acaban las puertas). Se recurre a la automatización para detectarlo. Al instante de comenzar la búsqueda (*golpear la primera puerta*) el estudiante dice "*me olvidé de algo… de preguntar la cédula después de golpear la primera puerta*". En términos de pseudocódigo, no sería un problema si se asume que la pregunta queda implícita en la acción de *golpear la primera puerta*. Sin embargo, el estudiante hace explícita dicha pregunta dentro de la iteración. Por lo tanto, entiende que es ambiguo preguntarlo dentro pero no antes, por lo cual también decide ponerla en forma explícita antes de la iteración.

golpear la primera puerta
preguntar cédula
Mientras (*cédula es distinta de x*)
    golpear puerta siguiente
    preguntar cédula
Fin

Su segunda versión mantiene el problema original: contempla solo una de las condiciones de parada (encontrar el documento, al cual llama *x*), sin tomar en cuenta la otra. Esto puede deberse a una conceptualización incompleta en la primera parte del estudio, donde nunca dijo expresamente que debía detener la búsqueda en caso de

terminarse las puertas, lo dejó implícito (*Si se recorren todas las puertas… se deduce que ésta no reside detrás de las puertas estudiadas o no existe*). Su pensamiento necesita avanzar en la transición hacia *newC*' a efectos de incluir dicho cambio en forma explícita. Tras volver a la automatización, detecta el error. Se le pide buscar un documento que no está. Al llegar al paso *golpear puerta siguiente* tras haber visitado la última, se le pregunta y contesta: "*vamos a la puerta siguiente… y no hay*". En este punto, toma conciencia de la necesidad de incluir la segunda condición. Para ello, se introduce la utilidad de usar operadores booleanos para combinar condiciones.

golpear la primera puerta
preguntar cédula
Mientras (*cédula es distinta de x*) *AND* (nro de puerta ≤ 7)
    golpear puerta siguiente
    preguntar cédula
Fin

Su tercera versión tiene ahora un *problema de borde*. Al buscar un documento que no está en la hilera, pretende consultar el documento de la puerta siguiente a la última, la cual no existe (su instancia concreta es de 7 puertas, e intenta consultar el de la octava). Otros estudiantes también avanzan tras visitar la última puerta, pero no llegan a consultar el documento en una puerta inexistente, solamente avanzan. Nicolás, en cambio, pregunta por dicho documento y constata que no existe tras volver a la automatización. Dice "*la puerta 7 cumple con el segundo parámetro*" (refiriéndose a la segunda condición) "*entonces iba a seguir… no existen otras puertas*". Un problema como este generaría un error de *salida de rango* si ocurriera durante la ejecución de un *programa*. Cuestiones como estas son inherentes a la construcción de conocimiento sobre aspectos propios de la ejecución en máquina, lo cual se aborda en la tercera actividad (escritura del programa). En cualquier caso, constituye un error a corregir, lo cual hace cambiando ≤ por < en la segunda condición.

golpear la primera puerta
preguntar cédula
Mientras (*cédula es distinta de x*) *AND* (nro de puerta < 7)
    golpear puerta siguiente
    preguntar cédula
Fin

Finalmente, su cuarta versión es correcta, en el sentido de que resuelve el problema, pero no lo hace para cualquier cantidad de puertas, sino para la cantidad puntual de su instancia concreta. En este punto no se le menciona nada al respecto, pues la generalización para cualquier cantidad se trabaja luego en la segunda y tercera actividad. Cuando escribe esta versión, inicialmente mantiene el operador *AND* de la versión anterior, duda un instante y lo cambia por *OR*. Enseguida duda de nuevo y lo vuelve a cambiar por *AND* (todo por iniciativa propia) mientras escribe, ni siquiera necesita volver a la automatización. Expresa los cambios en objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse) y las

acciones realizadas: comparación (*cédula es distinta de x*), repetición (*Mientras*) y avance (*golpear puerta siguiente*). Estructura su algoritmo de tal forma que inicia posicionado frente a la primera puerta. Consulta el primer documento antes de iniciar la iteración y queda posicionado en la última puerta visitada al finalizar la iteración, tanto si encuentra en ella el documento buscado como si el mismo no está en la hilera (estrategia de búsqueda a *priori*).

Pablo M

Su primera versión del algoritmo es la siguiente:

Mientras (*haya puertas en la cuadra*)
    golpeo la puerta
    pregunto por fulanito (*cédula*)
    Si (*fulanito está en la puerta*)
        Termino la búsqueda
    Sino
        Voy a la siguiente puerta
    Fin
    Si (*fulanito no está en ninguna puerta*)
        Marco que fulanito no vive en esa cuadra
    Fin
Fin

Su primera versión hace explícito solamente uno de los dos cambios posibles (parar cuando se terminan las puertas de la hilera), sin expresar adecuadamente el otro (parar al encontrar el documento buscado). Al recurrir a la automatización, queda en evidencia una conceptualización incompleta, por parte del estudiante, de que todos los pasos dentro de la iteración deben hacerse en forma *secuencial*. Esto es detectado al pretender buscar un documento que no está en la primera puerta. Al llegar al paso *Voy a la siguiente puerta*, el estudiante pretende que se evalúe nuevamente la condición de la estructura *Mientras*, salteando la segunda estructura *Si* luego de la primera. Se le hace notar que, antes de volver a evaluarla, primero hay que pasar por el segundo *Si*, a lo que dice: "*entonces este Si no iría*". Lo quita, resultando en la siguiente versión:

Mientras (*haya puertas en la cuadra*)
    golpeo la puerta
    pregunto por fulanito (*cédula*)
    Si (*fulanito está en la puerta*)
        Termino la búsqueda
    Sino
        Voy a la siguiente puerta
    Fin
Fin

Su segunda versión mantiene el problema original: no expresa adecuadamente la detención al encontrar el documento buscado. Su algoritmo continúa la búsqueda tras

haberlo encontrado. Quizás el estudiante no sea consciente de esto, ya que incluye el paso *Termino la búsqueda* dentro de la iteración, en caso de hallarlo. De hecho, en su descripción inicial al finalizar la primera parte fue explícito al expresar la detención por esta razón (*Repito el proceso anterior… hasta encontrar a fulanito*) por lo cual quizás piensa que es suficiente con incluir dicho paso para cortar la iteración. Su pensamiento necesita avanzar en la transición hacia *newC'*.

Se recurre nuevamente a la automatización para buscar un documento que está en la hilera. Cuando lo encuentra, efectivamente manifiesta intención de parar ni bien se llega al paso *termino la búsqueda* (sin evaluar una vez más la condición de *Mientras*). Muestra una conceptualización incompleta de la semántica de dicha estructura, al no tener claro que necesita volver a evaluar la condición antes de terminar. Se le hace notar que debe hacerse y dice: "*por más que cumplió el cometido, sigue repitiendo el proceso*". Se le pregunta cuándo terminaría de iterar y contesta: "*cuando se termine la cuadra… sin importar si encontramos a fulanito o no*". En este punto, el estudiante toma conciencia de la importancia de expresar la detención en forma acorde a la semántica de *Mientras*. Para ello, se introduce la utilidad de usar operadores booleanos para combinar condiciones y el estudiante escribe una nueva versión:

Mientras (*haya puertas en la cuadra*) *AND* (*no encuentre a fulanito*)
    golpeo la puerta
    pregunto por fulanito (*cédula*)
    Si (*fulanito está en la puerta*)
        Termino la búsqueda
    Sino
        Voy a la siguiente puerta
    Fin
Fin

Finalmente, su tercera versión es correcta. Utiliza la palabra *no* en la segunda condición, en vez de usar expresamente el operador *NOT*, lo cual no afecta la semántica. Expresa los cambios en objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse) y acciones: comparación (*pregunto por fulanito (cédula)* y *fulanito está en la puerta*), repetición (*Mientras*) y avance (*Voy a la siguiente puerta*). No intenta primero con *OR*, sino que lo resuelve directo con *AND*. En este sentido, a diferencia de otros estudiantes, muestra una conceptualización más sólida de la relación entre dicho operador y la semántica de *Mientras*. Estructura su algoritmo de tal forma que inicia posicionado frente a la primera puerta y siempre hace la comparación dentro de la iteración (incluso en la primera). Tras visitar la última, la acción *Voy a la siguiente puerta* no implica riesgo de consultar por un documento en una puerta inexistente, pues lo controla en la siguiente evaluación de la primera condición de la estructura *Mientras* (estrategia de búsqueda a *posteriori*).

Pablo P

Su primera versión del algoritmo es la siguiente:

```
numero = 0
puerta = 0
CI = numero conocido
Mientras (puerta ≤ 4)
        pregunto numero en puerta
        Si (numero != CI)
                puerta = puerta + 1
        Fin
Fin
```

Pablo P ya incorpora algunos elementos del lenguaje de programación (variables, operador de asignación, expresiones aritméticas y booleanas). Dado que no contradice las reglas sintácticas dadas para pseudocódigo, se deja que use dichos elementos. Inicializa las variables correctamente y opera con ellas de forma adecuada. A diferencia de su descripción en lenguaje natural dada en la primera parte (describió una instancia concreta en la cual encuentra el documento buscado específicamente en la tercera puerta), ahora describe el algoritmo tomando en cuenta la posibilidad de encontrarlo en cualquier puerta. No obstante, lo hace atado a la cantidad específica de 5 puertas (numeradas de 0 a 4) correspondiente a su instancia concreta.

Su primera versión presenta el siguiente problema: refleja solamente uno de los dos cambios posibles (parar cuando se terminan las puertas de la hilera), sin tomar en cuenta el otro (parar al encontrar el documento buscado). Pretende continuar la búsqueda luego de haber encontrado el documento, pero queda en *loop infinito*, posicionado en la puerta donde lo encuentra. A diferencia de lo expresado en su descripción de la primera parte, ahora toma en cuenta el cambio que había omitido en dicha descripción (parar cuando no quedan puertas). Su pensamiento necesita transitar hacia *newC*'. Debe incorporar la detención cuando encuentra el documento. Al volver a la automatización para buscar un documento que está en la hilera, detecta el problema. Luego de encontrarlo dice: "*va a seguir*". En este punto, toma conciencia de la necesidad de incluir la segunda condición. Para ello, se introduce la utilidad de usar operadores booleanos para combinar condiciones.

```
numero = 0
puerta = 0
CI = numero conocido
Mientras (puerta ≤ 4) AND (numero != CI)
        pregunto numero en puerta
        Si (numero != CI)
                puerta = puerta + 1
        Fin
Fin
```

Su segunda versión es correcta, en el sentido de que resuelve el problema, pero no lo hace para cualquier cantidad de puertas, sino para la cantidad puntual de 5 puertas de su instancia concreta. Además, funciona bien siempre que el número buscado no sea cero. Por tratarse de un documento de identidad, el estudiante asume que dicho número no puede ser cero y usa ese valor para inicializar su variable *número*. Expresa cambios en objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse) y acciones realizadas: comparación (*numero* != *CI*), repetición (*Mientras*) y avance (*puerta := puerta + 1*). Por otra parte, utiliza directamente *AND* (sin haber intentado antes con *OR*). En este sentido, a diferencia de otros estudiantes, muestra una conceptualización más sólida de la relación entre dicho operador y la semántica de la estructura *Mientras*. Estructura su algoritmo de tal forma que inicia posicionado frente a la primera puerta y siempre hace la comparación dentro de la iteración (incluso en la primera puerta). Tras visitar la última puerta sin encontrar en ella el documento buscado, la acción *puerta = puerta + 1* no implica riesgo de consultar por un documento en una puerta inexistente, pues lo controla en la siguiente evaluación de la primera condición de *Mientras* (estrategia de búsqueda a *posteriori*).

Tomás

Su primera versión del algoritmo es la siguiente:

```
Mientras (hayan puertas por ver)
        preguntar si es el número
        Si (está el número)
                elijo esa puerta
        Sino
                sigo
        Fin
Fin
```

Esta versión explicita solo uno de los dos cambios posibles (parar cuando se terminan las puertas de la hilera), sin hacer explícito el otro (parar al encontrar el documento). Esto puede deberse a una conceptualización incompleta en la primera parte, ya que en su descripción en lenguaje natural hace explícita la detención cuando encuentra el documento en la primera puerta (*Me dirijo a la primer puerta* y *Si encuentro el número, selecciono esa puerta y no me fijo en el resto*) pero no si lo encuentra en alguna de las siguientes (lo sugiere pero no lo hace explícito: *repito el mismo proceso hasta encontrar el número*). Su algoritmo continúa buscando tras encontrar el documento. Quizás el estudiante no es consciente de esto, ya que incluye el paso *elijo esa puerta* dentro de la iteración (en caso de encontrarlo) por lo cual a lo mejor piensa que alcanza con incluir ese paso para cortar la iteración. Cualquiera sea el caso, necesita avanzar en la transición hacia *newC*' y comprender la importancia de instruir al agente externo para parar al encontrar el documento, en forma acorde a la semántica de *Mientras*.

Se le pide volver a la automatización para buscar un documento que está en la hilera y dice: "*elijo esa puerta y fin*" cuando lo encuentra, manifestando efectivamente su intención de parar ni bien se llega a ese paso, sin volver a evaluar la condición de *Mientras*. Muestra una conceptualización incompleta de la semántica de dicha estructura, al no tener claro que necesita volver a evaluar la condición antes de terminar. Se le hace notar esto y se le pregunta qué debería hacer. Contesta: "*seguir viendo si hay puertas por ver*". Se le pregunta si es lo que se quiere y responde: "*no*". En este punto, el estudiante toma conciencia de la importancia de expresar la detención en forma acorde a la semántica de *Mientras*. Para ello, se introduce la utilidad de usar operadores booleanos para combinar condiciones.

```
Mientras (hayan puertas por ver AND está el número)
        preguntar si es el número
        Si (está el número)
                elijo esa puerta
        Sino
                sigo
        Fin
Fin
```

En su segunda versión, pretende detener la búsqueda cuando encuentra el documento buscado. Para ello, agrega una segunda condición utilizando *AND* (sin haber intentado primero con *OR*). En este sentido, a diferencia de otros estudiantes, muestra una conceptualización más sólida de la relación entre dicho operador y la semántica de la estructura *Mientras*. El problema es que lo hace de tal forma que no itera ni siquiera una vez (le falta negar dicha condición). Su segunda condición expresa lo que debe cumplirse para *parar* de iterar en vez de para *seguir* iterando. Al volver a la automatización, constata este error y dice: "*se cortaría ahí mismo, o sea que…*" mientras procede a borrar y modificar lo escrito "*…y no está el número*".

```
Mientras (hayan puertas por ver AND no está el número)
        preguntar si es el número
        Si (está el número)
                elijo esa puerta
        Sino
                sigo
        Fin
Fin
```

Finalmente, su tercera versión es correcta, en el sentido de que resuelve el problema. Expresa cambios en objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse). En cuanto a acciones, solamente hace explícita la comparación (*preguntar si es el número*) y (*si está el número*) y repetición (*Mientras*). Respecto al avance, solamente pone *sigo*, sin aclarar expresamente que irá a la siguiente. No hace explícito que la recorrida de la hilera se hace en forma secuencial. Si bien su transición hacia *newC* es menor que para otros estudiantes (que

sí explicitan las tres acciones), no es un problema en términos del formalismo intermedio (*pseudocódigo*), ya que igual es lo bastante claro para ser interpretado por una persona haciendo de robot imaginario en vez de una computadora, por lo que no se le pide una nueva versión. Tendrá oportunidad de profundizar en dicha transición en la tercera actividad (escritura del programa). Además, pone la palabra *no* en su segunda condición, en vez de usar expresamente el operador *NOT*, lo cual no afecta la semántica. Estructura su algoritmo de tal forma que inicia posicionado frente a la primera puerta y siempre hace la comparación dentro de la iteración (incluso en la primera puerta). Tras visitar la última puerta, la acción *sigo* no implica riesgo de consultar por un documento en una puerta inexistente, pues lo controla en la siguiente evaluación de la primera condición de *Mientras* (estrategia de búsqueda a *posteriori*).

Valeria

Su primera versión del algoritmo es la siguiente:

```
Para cada puerta hacer
        preguntar nro C.I
        Si (encuentro nro C.I)
                parar de preguntar
        Sino
                seguir preguntando
        Fin
Fin
```

Conforme a la semántica de *Para cada*, esta versión recorre toda la hilera, sin importar si el documento buscado se encuentra en ella o no. Esto puede deberse a una conceptualización incompleta en la primera parte, ya que en su descripción en lenguaje natural dijo *golpeo cada puerta preguntando*. Quizás realmente piensa que necesita recorrer hasta el final de la hilera, o quizás no sea consciente de que la recorre toda, ya que incluye el paso *parar de preguntar* dentro de la iteración, en caso de encontrarlo. Cualquiera sea el caso, necesita avanzar en la transición hacia *newC*' y comprender la importancia de instruir al agente externo para detenerse al encontrar el documento buscado, usando una estructura iterativa acorde a dicho propósito. Tras escribir esta versión y previo a pasar a la automatización, se le pregunta cuántas puertas debería visitar y responde: "*siete, es para cada puerta*" (todas las puertas de su instancia concreta). Manifiesta en forma expresa la intención de recorrer toda la hilera.

Se recurre a la automatización para buscar un documento que se encuentra en la tercera puerta. Al llegar al paso *parar de preguntar*, se le hace notar que no recorrió todas. Dice: "*tengo que parar acá… así que es con mientras*". Ha avanzado en la transición hacia *newC*', al tomar conciencia de la necesidad de expresar la detención cuando encuentra el documento usando una estructura iterativa acorde (*Mientras*).

Mientras (*no encuentre nro C.I*) *hacer*
    preguntar nro C.I
    Si (*encuentro nro C.I*)
        parar de preguntar
    Sino
        seguir preguntando
    Fin
Fin

En su segunda versión usa la palabra "*no*" dentro de la condición en vez de utilizar expresamente el operador *NOT*, lo cual no afecta la semántica. El problema es que contempla solamente uno de los dos cambios posibles (parar al encontrar el documento), sin tomar en cuenta el otro (parar cuando se terminan las puertas). Su pensamiento necesita continuar la transición hacia *newC*'. Debe comprender que necesita instruir al agente externo para detenerse en ese caso. Detecta el error al volver a la automatización. Se le pide buscar un documento que no está en la hilera. Al llegar al paso *seguir preguntando* tras haber visitado la última puerta, se le pregunta al respecto y contesta: "*ya pregunté en la última, me faltó parar*". En este punto, la estudiante toma conciencia de la importancia de expresar la segunda condición de parada. Para ello, se introduce la utilidad de usar operadores booleanos para combinar condiciones y escribe una nueva versión:

Mientras (*no encuentre nro C.I OR pregunté en última puerta*) *hacer*
    preguntar nro C.I
    Si (*encuentro nro C.I*)
        parar de preguntar
    Sino
        seguir preguntando
    Fin
Fin

Su tercera versión intenta incluir ambas condiciones de parada, pero lo hace utilizando *OR*, de manera incorrecta. Ha dado un paso hacia *newC*', pero hace uso de la disyunción, como lo haría en lenguaje natural. El uso de *OR* junto con su segunda condición muestra que está pensando en lo que debe cumplirse para *detener* la iteración. Precisa reflexionar sobre la relación entre la semántica de los operadores booleanos y de la estructura *Mientras*, a efectos de expresar la detención en forma acorde (debe hacerlo de manera que la iteración finaliza cuando alguna de las condiciones *deja* de cumplirse). Se le pide volver a la automatización para buscar un documento que no existe. Tras visitar la última puerta, se le hace notar que la primera condición se sigue cumpliendo. Dice "*entonces se rompe*" (refiriéndose a la ocurrencia de un error).

Mientras (*no encuentre nro C.I AND pregunté en última puerta*) *hacer*
    preguntar nro C.I
    Si (*encuentro nro C.I*)
        parar de preguntar
    Sino
        seguir preguntando
    Fin
Fin

En su cuarta versión cambia *OR* por *AND*, manteniendo sin cambiar ambas condiciones. Toma conciencia de que debe cambiar de operador, pero necesita modificar la segunda condición. Nuevamente se le pide volver a la automatización para buscar un documento que no existe. Constata que no entra a la iteración ni una vez, pues la segunda condición es falsa la primera vez que se evalúa. Dice "*acá también tiene que ser con no*" (señala la segunda condición). Agrega dicho *no*, quedando así:

Mientras (*no encuentre nro C.I AND no pregunté en última puerta*) *hacer*
    preguntar nro C.I
    Si (*encuentro nro C.I*)
        parar de preguntar
    Sino
        seguir preguntando
    Fin
Fin

Finalmente, su quinta versión es correcta, en el sentido de que resuelve el problema. Expresa cambios en objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse). En cuanto a acciones, solamente hace explícita la repetición (*Mientras*), dejando implícitas las otras dos. No explicita la comparación (solo pone *Si encuentro nro C.I.*) ni el avance (pone *seguir preguntando* sin aclarar que es en la siguiente). No hace explícito que la recorrida de la hilera se hace en forma secuencial, avanzando de a una puerta por vez. Si bien su transición hacia *newC* es menor que para otros estudiantes (que sí explicitan las tres acciones), no es un problema para el formalismo intermedio (*pseudocódigo*), ya que igual es lo bastante claro para ser interpretado por una persona haciendo de robot en vez de una computadora, por lo que no se pide una nueva versión. Tendrá oportunidad de profundizar en dicha transición en la tercera actividad (escritura del programa). Estructura su algoritmo de tal forma que inicia posicionada frente a la primera puerta y siempre hace la comparación dentro de la iteración (incluso en la primera puerta). Tras visitar la última, la acción *seguir preguntando* no implica riesgo de consultar por un documento en una puerta inexistente, ya que lo controla en la siguiente evaluación de la segunda condición de *Mientras* (estrategia de búsqueda *a posteriori*).

Ximena

Su primera versión del algoritmo es la siguiente:

```
cédula = 144
Mientras hallan puertas hacer
        buscar cédula
        Si cédula = 144
                terminé de buscar
        Sino
                Si cédula ≠ 144
                        sigo recorriendo puertas
        Fin
Fin
```

En esta versión, Ximena busca el documento de su instancia concreta (144). Su pseudocódigo no busca un documento cualquiera, sino que está atado a su instancia concreta (en este caso, al documento concreto buscado). Al preguntarle cómo haría para buscar cualquier documento dice "*que el usuario ingrese la cédula*". No se le pide que lo cambie, pues el salto hacia el caso general (para buscar un valor cualquiera) se trabaja luego en la tercera actividad. Se le pregunta por el significado del paso *buscar cédula* a lo que responde: "*abrir la puerta, verificar si la cédula de la persona que hay es igual a la que yo tengo*", aclarando que se refiere a la comparación de documentos.

Por otra parte, considera solo uno de los dos cambios posibles (parar cuando se acaban las puertas), sin tomar en cuenta el otro (parar al encontrar el documento). Su algoritmo continúa buscando tras haber encontrado el documento. Esto puede deberse a una conceptualización incompleta en la primera parte, ya que en su descripción en lenguaje natural hace explícita la detención al encontrar el documento en la primera puerta (*Empiezo buscando la cédula en la puerta cero* y *Si coincide, terminé de buscar*) pero no si lo encuentra en alguna de las siguientes (solo dice *sigo recorriendo puertas hasta encontrarla*). Capaz que no es consciente de que sigue buscando, ya que pone el paso *terminé de buscar* dentro de la iteración por lo cual quizás piensa que alcanza con eso para cortar la iteración. Cualquiera sea el caso, necesita avanzar en la transición hacia *newC*' y comprender la importancia de instruir al agente externo para parar al encontrar el documento, en forma acorde a la semántica de *Mientras*.

Se le pide volver a la automatización para buscar el 144 (que está en la hilera). Tras encontrarlo dice: " *terminé de buscar*" manifestando su intención de parar al llegar a ese paso, sin volver a evaluar la condición. Muestra una conceptualización incompleta de la semántica de *Mientras*, al no tener claro que necesita volver a evaluarla antes de terminar. Se le hace notar que el robot debe hacerlo y pregunta: "*¿pero por qué seguís haciendo eso?*". Ante esto, se repasa la semántica de *Mientras* y constata que igualmente continúa buscando tras encontrarlo. En este punto, toma conciencia de la importancia de expresar la detención en forma acorde a dicha estructura. Para ello, se introduce la utilidad de usar operadores booleanos para combinar condiciones:

```
Mientras hallan puertas AND no encuentre la cédula que busco hacer
        abrir puerta
        Si cédula es la que busco
                terminé de buscar
        Sino
                Si cédula es distinta a la que busco
                        sigo recorriendo puertas
        Fin
Fin
```

En su segunda versión, pone *no* en la segunda condición en vez de utilizar expresamente el operador *NOT*, lo cual no afecta la semántica. Además, realiza dos veces la comparación entre documentos, una por la afirmativa y otra por la negativa. Se le hace notar que no es necesario, pero no es un error y dice "*no, pero es confuso*". Al volver a la automatización, dice: "*lo vuelvo a escribir, cuando podría poner sigo recorriendo puertas*" (refiriéndose a que vuelve a comparar, cuando alcanza solamente con seguir recorriendo). Lo quitar, resultando en una nueva versión:

```
Mientras hallan puertas AND no encuentre la cédula que busco hacer
        abrir puerta
        Si cédula es la que busco
                terminé de buscar
        Sino
                sigo recorriendo puertas
        Fin
Fin
```

Finalmente, su tercera versión es correcta, en el sentido de que resuelve el problema. Duda un instante entre mantener *AND* o cambiar por *OR*. Opta por dejar *AND*, pero sin estar convencida. Tras volver a la automatización, se convence de que funciona bien. Expresa cambios en objetos (para de iterar cuando alguna de las condiciones, combinadas con *AND*, deja de cumplirse). En cuanto a las acciones, solamente explicita la repetición (*Mientras*) y la comparación (*Si cédula es la que busco*). Respecto al avance, pone *sigo recorriendo puertas* (en plural) sin aclarar expresamente que avanza de a una puerta por vez y que continúa con la puerta siguiente, en forma secuencial. Si bien su transición hacia *newC* es menor que para otros estudiantes (que sí explicitan las tres acciones), no es un problema para el formalismo intermedio (*pseudocódigo*), ya que igual es lo bastante claro para ser interpretado por una persona haciendo de robot en vez de una computadora, por lo que no se pide una nueva versión. Estructura su algoritmo de tal forma que inicia posicionada frente a la primera puerta y siempre hace la comparación dentro de la iteración (incluso en la primera puerta). Tras visitar la última puerta, la acción *sigo recorriendo puertas* no implica riesgo de consultar por un documento en una puerta inexistente, pues lo controla en la siguiente evaluación de la primera condición de *Mientras* (estrategia de búsqueda *a posteriori*).

Anexo C

Sintaxis y *semántica* del lenguaje formal (segunda parte: *inter* → *trans*)

En este anexo se presentan los fragmentos de programa escritos por los trece estudiantes durante la segunda actividad de la segunda parte del estudio (pasaje de la etapa *inter* a la etapa *trans*). Se trata de cuatro fragmentos puntuales que resuelven problemas sencillos relativos al uso de arreglos en el lenguaje de programación utilizado por cada estudiante (*C* o *Pascal*), permitiendo ejercitar las reglas de *sintaxis* y *semántica* para su manipulación y su combinación con otros elementos del lenguaje.

1. Sumar los valores de las primeras dos celdas y guardar el resultado en una variable entera (`suma`)
2. Mostrar en pantalla un mensaje indicando si el valor en la celda 3 es par o impar
3. Determinar si el valor en la celda 1 es o no igual al valor en la última celda y guardar el resultado en una variable booleana (`iguales`)
4. Recorrer el arreglo de la primera a la última celda e ir mostrando por pantalla sus valores

Se presentan *todas* las versiones escritas por cada estudiante para cada fragmento junto con un análisis detallado de cada una y del proceso realizado. En cada análisis, se usan cursivas entre comillas para citar respuestas orales de los estudiantes durante las entrevistas. Las entrevistas fueron grabadas y se tomaron fotografías de las hojas de papel conteniendo los distintos fragmentos escritos por los estudiantes, que fueron transcritos aquí.

Aaron (*Pascal*)

Fragmento 1: Escribe una única versión, la cual es correcta:

```
Suma := arre[1] + arre[2];
```

Fragmento 2: Su primera versión es la siguiente:

```
if (arre[3] Mod 2 = 0) then
    Write ('el valor es Par');
else
    Write ('el valor es impar');
```

Esta versión presenta un error de sintaxis, ya que Pascal no admite el uso de punto y coma previo a `else`. Se da cuenta del error por sí solo durante la escritura y lo tacha, resultando en la segunda versión (que es correcta). Consultado al respecto, dice "*el compilador llega hasta acá y me dice que encuentra un else... pero que no tiene... que no está bien inicializado*" (se refiere a que no está asociado con la sentencia `if` previa).

```
if (arre[3] Mod 2 = 0) then
    Write ('el valor es Par')
else
    Write ('el valor es impar');
```

Fragmento 3: Su primera versión es la siguiente:

```
iguales := arre[1] = arre[8];
```

Esta versión es correcta, en el sentido de que resuelve el problema, pero está atada a la instancia concreta manipulada, en la cual trabaja con un arreglo de ocho celdas y utiliza los índices 1 y 8 para referirse a la primera y última, respectivamente. Consultado acerca de cómo haría si fuese cualquier otra cantidad de celdas (por ejemplo, 20) dice "*cambiar el 8 por el valor de la última celda*" (se refiere al índice de la última celda) y agrega "*si estoy usando esto que dice acá, que la constante es igual a 8, yo pongo 1 con N*" (se refiere a que compara la celda 1 con la celda N).

```
iguales := arre[1] = arre[N];
```

Su segunda versión también es correcta, pero ahora es más general, al hacer referencia a la constante N definida en la declaración del tipo arreglo. Por lo tanto, la instrucción ahora resuelve el problema sin importar la cantidad de celdas del arreglo manipulado en la instancia concreta. Conforme a lo concluido por Matalon, la sucesiva repetición de una acción *genérica* (en este caso acceder a la *última* celda), permite al estudiante lograr el salto de casos particulares al caso *general*. Además, asigna (correctamente) una expresión booleana (la comparación con =) a una variable booleana. Al consultarle al respecto, dice: "*iguales, como es un boolean, yo puedo hacer una comparación, que por eso utilicé el iguales*" (se refiere a que puede asignar la expresión booleana a la variable booleana *iguales*, ya que son ambas del mismo tipo).

Fragmento 4: Su primera versión es la siguiente:

```
For i := 1 to N Do
    Write (arre[i]);
end;
```

Esta versión presenta un error de sintaxis, ya que Pascal no admite `end` en la sentencia `for` sin estar asociado a `begin`. Consultado al respecto dice: "*no es necesario el end este... porque es solamente una instrucción... yo sé que hay una que lleva end, me parece*" (se refiere a la instrucción `case` de Pascal, que vio en clase previo al estudio y que utiliza solamente `end` sin `begin`). Procede a quitar `end`, resultando en la siguiente versión (que es correcta).

```
For i := 1 to N Do
    write (arre[i]);
```

Consultado acerca de la elección de `for` para resolver el problema, dice "*porque me dicen que recorra todo el arreglo y que muestre los valores en cada caso*" (en cada celda). Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Ignacio D (*Pascal*)

Fragmento 1: Escribe una única versión, la cual es correcta:

```
SUMA := arre[1] + arre[2];
```

Fragmento 2: Su primera versión es la siguiente:

```
if arre[3] mod 2 = 0 then
    write ('es par')
else
    write ('es impar')
end
```

Esta versión presenta un error de sintaxis, ya que Pascal no admite `end` en la sentencia `if` sin estar asociado a `begin`. Se da cuenta por sí solo durante la escritura y tacha `end`, resultando en su segunda versión (correcta). Consultado al respecto dice: "*en realidad estaba mirando ahí el ejemplo y ví que al final había un end*" (se refiere la tarjeta donde se presenta la sintaxis genérica de la instrucción `if/else`, con `begin` y `end`) y continúa "*pero en realidad como no tenía ningún begin y era una instrucción sola, no necesita el end*" (refiriéndose a que puede prescindir de `begin/end` cuando dentro de la cláusula `else` hay una sola instrucción).

```
if arre[3] mod 2 = 0 then
    write ('es par')
else
    write ('es impar')
```

Fragmento 3: Escribe una única versión, la cual es correcta. No asigna una expresión booleana a la variable, sino que usa `if/else` para asignar el resultado dependiendo del cumplimiento o no de la condición que compara los valores en las celdas.

```
if arre[1] = arre[N] then
    iguales := true
else
    iguales := false
```

Directamente hace uso de la constante N para referirse al índice de la última celda, sin probar primero con el índice de su instancia concreta. No obstante, duda mientras escribe. Consultado al respecto, dice "*no estaba seguro, y en realidad declara N como 8 en la constante, así que poner 8 o N me pareció que era lo mismo*" (refiriéndose al valor dado a N en la declaración de la constante). Se le pregunta qué pasaría si el valor

dado a N fuese otro (por ejemplo, 20) y dice "*en la condición cambiarlo por 20*" (refiriéndose a cambiar `arre[N]` por `arre[20]` en la condición de la sentencia `if`). Si en vez de 20 fuese 50, dice "*cambiarlo por 50*". Dado que inicialmente puso N, se le consulta qué sucede si lo deja así, a lo que responde "*ya está, o sea, cambia automáticamente*", dando a entender que en realidad no necesita modificar la condición. Conforme a lo concluido por Matalon, la sucesiva repetición de una acción *genérica* (en este caso acceder a la *última* celda), permite al estudiante lograr el salto de casos particulares al caso *general*.

Fragmento 4: Escribe una única versión, la cual es correcta:

```
for i := 1 to N do
    write (arre[i])
```

Consultado acerca de la elección de `for` para resolver el problema, dice "*en el problema me planteaban que tenía que recorrer todo el arreglo y como ya sabía la cantidad de veces que tenía que hacer la repetición... me valía más la pena usar un for que un while*". Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Ignacio U (*Pascal*)

Fragmento 1: Escribe una única versión, la cual es correcta:

```
suma := arre[1] + arre[2];
```

Fragmento 2: Su primera versión es la siguiente:

```
if arre[3] = mod 2 then
    write ('es par')
else
    write ('es impar')
```

Esta versión tiene un error de sintaxis en la condición de la sentencia `if`. A la izquierda del operador `mod` debería haber un operando entero en lugar de `=`. Consultado al respecto dice: "*estoy preguntando si el contenido de la celda 3 es par, con mod 2*". Quizás hace una lectura de la expresión similar a como lo haría en lenguaje natural (pone = *mod 2* queriendo decir *es igual a múltiplo de 2*), queriendo *generalizar* al lenguaje *formal* su conocimiento del lenguaje *natural* (la herramienta cognitiva que interviene es la generalización *inductiva*). Necesita adaptar su pensamiento para escribirla teniendo en cuenta que será evaluada por la computadora en vez de por una persona. Precisa usar la sintaxis del lenguaje *formal*. Se le hace notar el error y se da cuenta del mismo (dice "*me tiene que dar resto cero*") y lo corrige.

```
if arre[3] mod 2 = 0 then
    write ('es par')
else
    write ('es impar')
```

Fragmento 3: Su primera versión es la siguiente:

```
if arre[1] = arre[8] then
    iguales := true
else
    iguales := false;
```

Esta versión es correcta, en el sentido de que resuelve el problema, pero está atada a la instancia concreta manipulada, en la cual trabaja con un arreglo de ocho celdas y utiliza los índices 1 y 8 para referirse a la primera y última, respectivamente. Consultado acerca de cómo haría si fuese cualquier otra cantidad de celdas (por ejemplo, 20) dice "*acá le habría puesto N*" (señalando el 8 en `arre[8]`) "*porque es la constante definida… y el array está definido arriba de 1 a N*". Si en vez de 20 fuese 50, dice "*no tenés que modificar… en caso de que varíe la constante esa*" (refiriéndose a que si se cambia el valor dado a la constante, no necesita volver a modificar la condición de la sentencia `if`).

```
if arre[1] = arre[N] then
    iguales := true
else
    iguales := false;
```

Ahora resuelve el problema en forma general, sin importar la cantidad de celdas del arreglo manipulado en la instancia concreta. Conforme a lo concluido por Matalon, la sucesiva repetición de una acción *genérica* (en este caso acceder a la *última* celda), permite al estudiante lograr el salto de casos particulares al caso *general*. Lo resuelve haciendo uso de `if/else` para asignar el resultado, no asigna una expresión booleana a la variable.

Fragmento 4: Su primera versión es la siguiente:

```
for i := arre[1] to arre[N] do
    write (arre[i]);
```

Si bien compila, esta versión tiene un error en relación a los límites del rango de iteración en la sentencia `for`. En lugar de utilizar 1 y N como límites, usa `arre[1]` y `arre[N]`, pensando que de ese modo itera desde la primera celda hasta la última. Además, podría producir un error de *salida de rango* (según los valores concretos en dichas celdas) pero no se aborda aún este tema. Quizás hace una lectura propia de una persona (*desde la celda 1 hasta la celda N*). Necesita adaptar su pensamiento para escribir los límites teniendo en cuenta que la instrucción será ejecutada por la computadora en vez de por una persona. Debe comprender que la iteración debe hacerse sobre el rango de *índices* del arreglo. Necesita conceptualizar la diferencia entre los *índices* y los *valores* contenidos en las celdas. La automatización resulta de ayuda para ayudarlo en dicha conceptualización. Se le pregunta por los valores de `arre[1]` y `arre[N]` (en su instancia concreta) y dice "*17 y 55*". Se le indica entonces que su iteración irá de 17 a 55 y se le pregunta si es lo que quiere. Responde "*no, no, tendría que haber puesto los índices… el for iría de 1 hasta N*".

```
for i := 1 to N do
    write (arre[i]);
```

Finalmente, su segunda versión es correcta. Consultado acerca de la elección de `for` para resolver el problema, dice "*porque ya sé desde antes cuántas veces lo voy a repetir*". Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Joaquín (*Pascal*)

Fragmento 1: Su primera versión es la siguiente:

```
ARRE[2] + ARRE[1];
```

Inicialmente escribe solamente la expresión correspondiente a la suma. Durante la escritura, nota por sí mismo que no realiza la asignación y la agrega, resultando en la siguiente versión (que es correcta).

```
SUMA := ARRE[2] + ARRE[1];
```

Fragmento 2: Su primera versión es la siguiente:

```
IF ARRE[3] MOD 2 THEN
    WRITE ('Es Par')
ELSE
    WRITE ('Es Impar');
```

Esta versión tiene un error de semántica en la condición de la sentencia `if`. Falta comparar el resultado de la operación realizada con 0, a efectos de verificar si se trata de un valor par y que quede una expresión booleana como condición. Quizás hace una lectura similar al lenguaje natural (*el valor en la celda 3 es múltiplo de 2*, interpretando mod como *es múltiplo de*), queriendo *generalizar* al lenguaje *formal* su conocimiento del lenguaje *natural* (la herramienta cognitiva que interviene es la generalización *inductiva*). Necesita comprender que `mod` es un operador aritmético y no booleano. Se le hace notar que el resultado es un número usado como condición. Consultado al respecto dice: "*está mal, tengo que tener un = 0 ahí*".

```
IF (ARRE[3] MOD 2 = 0) THEN
    WRITE ('Es Par')
ELSE
    WRITE ('Es Impar');
```

Fragmento 3: Su primera versión es la siguiente:

```
IF ARRE[1] = ARRE[N] THEN
    IGUALES := ARRE[1]
ELSE
    IGUALES := FALSE;
```

Durante la escritura, nota por sí mismo que no corresponde asignar `ARRE[1]` a la variable `IGUALES`, por lo cual tacha y corrige, resultando en la siguiente versión (es correcta). Lo hace con `if/else` para asignar el resultado, no asigna una expresión booleana a la variable.

```
IF ARRE[1] = ARRE[N] THEN
    IGUALES := TRUE
ELSE
    IGUALES := FALSE;
```

Directamente hace uso de la constante N para referirse al índice de la última celda, sin intentar primero con el índice de su instancia concreta. Consultado al respecto, dice "*porque así me queda para cualquier caso*" (refiriéndose a cualquier valor para N). El estudiante logra con facilidad el salto de casos particulares al caso general, sin necesitar repetir sucesivamente la acción *genérica* de acceder a la *última* celda.

Fragmento 4: Escribe una única versión, la cual es correcta:

```
FOR i := 1 TO N DO
    WRITE (ARRE[i])
```

Durante la escritura, inicialmente pretende utilizar `begin/end` para delimitar el alcance de la iteración. Cambia de opinión sobre la marcha y solamente escribe la instrucción para desplegar. Consultado al respecto dice "*iba a poner más instrucciones pero no era necesario*" (se refiere a que no necesita `begin/end` por tratarse de una sola instrucción). A diferencia de otros estudiantes, utiliza mayúsculas para las palabras reservadas `FOR` e `IF`, así como para llamar al procedimiento `WRITE`. Pascal admite ambas formas (mayúsculas y/o minúsculas). Consultado acerca de la elección de `for` para resolver el problema, dice "*porque me recorre todos los valores, uno por uno, desde el 1 hasta el último*". Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Juan (*C*)

Fragmento 1: Su primera versión es la siguiente:

```
int suma = 0;
suma = arre.0 + arre.1;
```

Si bien no es necesario, inicializa la variable suma con o antes de realizar la asignación pedida. Presenta un error de sintaxis en el acceso a las celdas (utiliza punto en lugar de paréntesis rectos). Quizás hace una lectura similar a como haría en matemática, usando el punto para expresar cada índice como si fuese subíndice en una sucesión ($arre_0 + arre_1$). Se le dice que usa un operador no definido. Se le muestra nuevamente la tarjeta que ilustra la sintaxis para acceso a celdas y lo corrige.

```
int suma = 0;
suma = arre[0] + arre[1];
```

Fragmento 2: Su primera versión es la siguiente:

```
printf ("El valor de la celda 3 es: %c" ...(no termina de escribir)
```

El especificador de conversión `%c` se usa en *C* para desplegar un carácter, lo que evidencia la intención de desplegar un mensaje (formado por caracteres). Dado que se le pide *mostrar un mensaje indicando si el valor es par o impar*, inicialmente intenta combinar, en una instrucción simple, la decisión de si es par con la emisión del mensaje. Necesita modificar su pensamiento para conceptualizar que debe instruir a la computadora para tomar primero la decisión y luego emitir el mensaje que corresponda, según el resultado. Consultado al respecto, dice "*lo quise resolver en una sola, pero no... primero tengo que evaluar qué valor tiene para saber si es par o impar*", tras lo cual escribe la siguiente versión.

```
if (arre[3] % 2) {
    printf ("La celda 3 es par");
else
    printf ("La celda 3 es impar");
}
```

Esta versión tiene dos problemas. Uno es un error de sintaxis por cómo hace uso de las llaves. Las coloca de forma que la cláusula `else` no queda asociada a la cláusula `if`. Se le muestra nuevamente la tarjeta que describe la sintaxis genérica de la sentencia `if/else` y se pide que la compare con lo escrito. Para corregirlo, agrega dos llaves más rodeando a la cláusula else.

```
if (arre[3] % 2) {
    printf ("La celda 3 es par");
} else {
    printf ("La celda 3 es impar");
}
```

Corregido ese error, aún mantiene el otro problema: un error de semántica en la condición de la sentencia `if`. Falta comparar el resultado de la operación realizada con 0, a efectos de verificar si es un valor par y que quede una expresión booleana como condición. Quizás hace una lectura de la expresión similar a como haría en lenguaje natural (*el valor en la celda 3 es múltiplo de 2*, interpretando `%` como *es múltiplo de*), queriendo *generalizar* al lenguaje *formal* su conocimiento del lenguaje *natural* (la herramienta cognitiva que interviene es la generalización *inductiva*). Necesita comprender que `%` es un operador aritmético en vez de booleano. Se le pregunta qué está haciendo en su condición y dice: "*calculando si el resto de dividirlo entre 2 es cero*". Se le hace notar que falta expresar dicha comparación con cero, tras lo cual la agrega.

```
if (arre[3] % 2 = 0) {
    printf ("La celda 3 es par");
} else {
    printf ("La celda 3 es impar");
}
```

Su tercera versión tiene ahora un error de sintaxis. En lugar de usar el operador de comparación (==), usa el operador de asignación (=). Es un error común al escribir en *C*, dado que en matemática se usa = para comparación (nuevamente, generalización *inductiva*). Esto generaría que el compilador intente interpretar la expresión a la izquierda del operador como si fuera una variable, resultando en un error de sintaxis. Cuando se le pregunta al respecto, dice "*esto es igual igual... estás comparando si el valor es igual*", tras lo cual, lo cambia, resultando en una versión final correcta:

```
if (arre[3] % 2 == 0) {
    printf ("La celda 3 es par");
} else {
    printf ("La celda 3 es impar");
}
```

Fragmento 3: Su primera versión es la siguiente:

```
if (arre[0] == arre[7]) {
    boolean iguales = TRUE;
} else {
    boolean iguales = FALSE;
}
```

En esta versión, pretende declarar dos veces la variable `iguales`. El estudiante no lo sabe, pero en realidad declara dos variables con el mismo nombre, cada una local al bloque entre llaves donde la declara (las nociones de *local* y *global* aún no han sido vistas en clase). Sí ha visto que toda variable debe declararse *una* vez, pero quizás tiene dificultad en su conceptualización. Se le pregunta cuántas veces corresponde declarar y dice "*una sola vez y arriba... esta declaración boolean iguales = false arriba*".

```
boolean iguales = FALSE;
if (arre[0] == arre[7]) {
    iguales = TRUE;
}
```

En su segunda versión, escribe una sola declaración y quita la cláusula `else`. Cuando se le pregunta al respecto, dice "*porque en realidad, al tener ya valor false, solamente en el caso en que se cumpla la condición, cambia el valor, sino queda igual*". No asigna una expresión booleana a la variable, sino `true` o `false` según se cumpla o no la condición. Esta versión es correcta, en el sentido de que resuelve el problema, pero está atada a la instancia concreta manipulada (arreglo de ocho celdas e índices 0 y 7 para referirse a la primera y última, respectivamente). Consultado acerca de cómo haría si fuese cualquier otra cantidad de celdas (por ejemplo, 20) dice "*el 7 tendría que sustituirlo por el 20*". Se le hace notar que en C los índices empiezan en cero y dice "*tendría que poner 19*" y tacha el 7, cambiando por 19.

```
boolean iguales = FALSE;
if (arre[0] == arre[19]) {
    iguales = TRUE;
}
```

Si en vez de 20 fuesen 30, dice "*tachar y 29*".

```
boolean iguales = FALSE;
if (arre[0] == arre[29]) {
    iguales = TRUE;
}
```

Se le consulta qué tendría que hacer si se sigue pidiendo cambiar el índice, y contesta "*con cada valor que cambia, tendría que venir y cambiar acá* " (señala el índice de la última celda en la condición). Se le pide que mire la declaración del arreglo para ver si hay algo que le permita no tener que cambiar el valor cada vez. Lo hace y dice: "*...sí, N*". Tacha nuevamente el índice y cambia por N, resultando en la siguiente versión:

```
boolean iguales = FALSE;
if (arre[0] == arre[N]) {
    iguales = TRUE;
}
```

Se le hace notar nuevamente que los índices empiezan en cero, por lo cual no existe la celda con índice N (en caso de ejecutarse, produciría un error de *salida de rango*). Lo cambia por última vez, poniendo N-1, quedando finalmente correcta. Al hacer uso de la constante N, resuelve ahora el problema en forma general, sin importar la cantidad de celdas del arreglo manipulado en la instancia concreta. Conforme a lo concluido por Matalon, la sucesiva repetición de una acción *genérica* (en este caso acceder a la *última* celda), permite al estudiante lograr el salto de casos particulares al caso *general*.

```
boolean iguales = FALSE;
if (arre[0] == arre[N-1]) {
    iguales = TRUE;
}
```

Fragmento 4: Su primera versión es la siguiente:

```
int i = 0;
FOR (i = 0; i < N-1; i++) {
    printf ("%d", ARRE[i]);
}
```

Esta versión presenta un *problema de borde*. Deja sin mostrar el valor en la última celda. El hecho de que los índices en *C* empiezan en cero, lleva en ocasiones a los estudiantes a expresar incorrectamente el límite superior de la iteración. Se recurre a la automatización sobre su instancia concreta de 8 celdas y el estudiante detecta el problema en el instante en que se llega a la celda con índice 7. Dice "*acá en realidad estoy preguntando si es menor, tendría que ser menor o igual para poder ver el dato este... me falta uno*" (refiriéndose al valor de la última celda), tras lo cual lo corrige.

```
int i = 0;
FOR (i = 0; i <= N-1; i++) {
    printf ("%d", ARRE[i]);
}
```

Finalmente, su segunda versión es correcta, en el sentido de que resuelve el problema. Si bien no es necesario, inicializa la variable i con 0 antes de iniciar la iteración. En el lenguaje C, la palabra `for` se escribe en minúsculas, pero el estudiante usa mayúsculas. Esto daría un error de sintaxis al compilar. El tema no se le menciona, ya que se trata luego en el pasaje del código escrito en papel a la computadora (tercera actividad). Consultado acerca de la elección de `for` para resolver el problema, dice "*porque necesariamente tengo que recorrer todo el arreglo… el while lo usaría en el caso de que no tuviera que recorrer todo el arreglo*". Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Martín (*Pascal*)

Fragmento 1: Escribe una única versión, la cual es correcta:

```
Suma := arre[1] + arre[2]
```

Fragmento 2: Su primera versión es la siguiente:

```
if (arre[3] mod 2 = 0) then
    write ('Es par')
else
    write ('Es impar')
end
```

Esta versión presenta un error de sintaxis, ya que Pascal no admite `end` en la sentencia `if` sin estar asociado a `begin`. Se le pide que compare lo escrito con la tarjeta que ilustra la sintaxis de la instrucción `if/else`. Consultado si está bien así, dice: "*no… si hubiéramos puesto begin después del else*" (refiriéndose a que, en ese caso, sería correcto colocar `end`) y lo tacha, resultando en la siguiente versión (que es correcta).

```
if (arre[3] mod 2 = 0) then
    write ('Es par')
else
    write ('Es impar')
```

Fragmento 3: Escribe una única versión, la cual es correcta. No asigna una expresión booleana a la variable, sino que hace uso de `if/else` para asignar el resultado según el cumplimiento o no de la condición que compara los valores en las celdas.

```
if (arre[1] = arre[n]) then
    iguales := true
else
    iguales := false
```

Directamente hace uso de la constante N para referirse al índice de la última celda, sin probar primero con el índice de su instancia concreta. Consultado al respecto, dice "*porque n podría no ser 8… ser cualquier otra cosa*" (refiriéndose a cualquier valor para la constante N). Escribe la constante en minúscula, lo cual es válido en *Pascal* (a pesar de estar declarada en mayúscula). El estudiante logra con facilidad el salto de

casos particulares al caso general, sin necesitar repetir sucesivamente la acción *genérica* de acceder a la *última* celda.

Fragmento 4: Su primera versión es la siguiente:

```
for i : 1 to n do
    write (arre[i])
```

Esta versión tiene un error de sintaxis en la inicialización de `i`. En vez del operador de asignación (`:=`), usa dos puntos (`:`). Cuando se le pregunta al respecto, dice "*me faltó el igual... para completar la asignación* ", tras lo cual, lo cambia, quedando correcto.

```
for i := 1 to n do
    write (arre[i])
```

Consultado acerca de la elección de `for` para resolver el problema, dice "*porque tiene que recorrer todas las celdas que hay*". Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Mónica (*C*)

Fragmento 1: Su primera versión es la siguiente:

```
arre[0]
```

Inicialmente escribe solamente la expresión de acceso a la primera celda. Durante la escritura, nota por sí misma que no realiza la asignación. Tacha lo escrito y escribe la siguiente versión (que es correcta).

```
suma = arre[0] + arre[1];
```

Fragmento 2: Su primera versión es la siguiente:

```
if (arre[3] % 2 = 0)
    printf ("El número es par");
else
    printf ("El número es impar");
```

Esta versión tiene un error de sintaxis en la condición. En vez de usar comparación (==), utiliza el operador de asignación (=). Este es un error común al escribir en *C*, dado que en matemática se usa el símbolo = para expresar comparación. La estudiante *generaliza* al lenguaje de *programación* su conocimiento del lenguaje *matemático* (la herramienta cognitiva que interviene es la generalización *inductiva*). Esto generaría que el compilador intente interpretar la expresión a la izquierda de dicho operador como si fuera una variable, resultando en un error de sintaxis. Cuando se le pregunta al respecto, dice "*ah, no, es otro... es así*" (cambia = por ==).

```
if (arre[3] % 2 == 0)
    printf ("El número es par");
else
    printf ("El número es impar");
```

Fragmento 3: Su primera versión es la siguiente. No asigna una expresión booleana a la variable, sino que usa `if/else` para asignar el resultado según el cumplimiento o no de la condición que compara los valores en las celdas.

```
boolean iguales;
if (arre[0] == arre[7])
    iguales = TRUE;
else
    iguales = FALSE;
```

Esta versión es correcta, en el sentido de que resuelve el problema, pero está atada a la instancia concreta manipulada, en la cual trabaja con un arreglo de ocho celdas y utiliza los índices 0 y 7 para referirse a la primera y última, respectivamente. Consultada acerca de cómo haría si fuese otra cantidad de celdas (por ejemplo, 20) dice "*acá, habría que poner 19*" (señala el 7, tacha y lo cambia por 19).

```
boolean iguales;
if (arre[0] == arre[19])
    iguales = TRUE;
else
    iguales = FALSE;
```

Si en vez de 20 fuesen 30, dice "*en este caso tendría que poner 29*" (tacha y cambia).

```
boolean iguales;
if (arre[0] == arre[29])
    iguales = TRUE;
else
    iguales = FALSE;
```

Se le muestra la declaración del arreglo y se le consulta qué tendría que poner para expresar el índice de la última celda, sin tener que cambiar el valor cada vez, a lo que responde "*N-1*" (siendo N la constante de la declaración). Tacha de nuevo el índice y lo cambia por `N-1`, resolviendo ahora el problema en forma general, sin importar la cantidad de celdas del arreglo en la instancia concreta. Conforme a lo concluido por Matalon, la sucesiva repetición de una acción *genérica* (en este caso acceder a la *última* celda), permite a la estudiante lograr el salto de casos particulares al caso *general*.

```
boolean iguales;
if (arre[0] == arre[N-1])
    iguales = TRUE;
else
    iguales = FALSE;
```

Fragmento 4: Escribe una única versión, la cual es correcta:

```
int i;
for (i = 0; i < N; i++)
    printf ("%d", arre[i]);
```

Consultada acerca de la elección de `for` para resolver el problema, dice "*porque tengo que recorrer todos los índices... todas las celdas*". Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Nicolás (*Pascal*)

Fragmento 1: Escribe una única versión, la cual es correcta:

```
suma := arre[1] + arre[2]
```

Fragmento 2: Su primera versión es la siguiente:

```
if arre[3] MOD 2 = 0 then
    WriteLn ('El valor en la celda 3 es par');
else
    WriteLn ('El valor en la celda 3 es impar')
```

Durante la escritura, omite inicialmente poner = 0. Se da cuenta por sí solo durante la escritura y lo agrega, quedando la versión presentada. No obstante, presenta un error de sintaxis, ya que Pascal no admite el uso de punto y coma previo a else. Consultado al respecto, dice "*como hay solo una, no precisa... no es necesario*" (se refiere a la primera llamada a `WriteLn`) "*o sea, no es una instrucción lo que hay después*" (se refiere a que el código que sigue a ese punto y coma, empezando en `else`, no sería una instrucción válida). Cuando dice que "*no es necesario* ", se le consulta de nuevo y aclara "*te da error de compilación*". Lo tacha, quedando una segunda versión (correcta).

```
if arre[3] MOD 2 = 0 then
    WriteLn ('El valor en la celda 3 es par')
else
    WriteLn ('El valor en la celda 3 es impar')
```

Fragmento 3: Escribe una única versión, la cual es correcta. No asigna una expresión booleana a la variable, sino que hace uso de `if/else` para asignar el resultado según el cumplimiento o no de la condición que compara los valores en las celdas.

```
if arre[1] = arre[N] then
    iguales := True
else
    iguales := False;
```

Directamente hace uso de la constante N para referirse al índice de la última celda, sin probar primero con el índice de su instancia concreta. Al respecto, dice "*porque si se cambia el valor de la constante, no hay que cambiarlo acá también*" (señala `arre[N]`). El estudiante logra con facilidad el salto de casos particulares al caso general, sin necesitar repetir sucesivamente la acción *genérica* de acceder a la *última* celda.

Fragmento 4: Escribe una única versión, la cual es correcta:

```
For i := 1 to N Do
    WriteLn (arre[i])
```

Consultado acerca de la elección de `for` para resolver el problema, dice "*porque voy a recorrer todo el array y sé la cantidad de elementos que tiene*". Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Pablo M (*C*)

Fragmento 1: Escribe una única versión, la cual es correcta:

```
int suma;
suma = arre[0] + arre[1];
```

Fragmento 2: Su primera versión es la siguiente:

```
if (arre[3] % 2 = 0) {
    printf ("El valor de la celda 3 es par");
                      }
else {
    printf ("El valor de la celda 3 es impar");
     }
```

Esta versión tiene un error de sintaxis en la condición. En lugar de utilizar el operador de comparación (==), utiliza el operador de asignación (=). Este es un error común al escribir en *C*, dado que en matemática se usa el símbolo = para comparar. El estudiante *generaliza* al lenguaje de *programación* su conocimiento del lenguaje *matemático* (la herramienta cognitiva que interviene es la generalización *inductiva*). Esto generaría que el compilador intente interpretar la expresión a la izquierda de dicho operador como si fuera una variable, resultando en un error de sintaxis. Consultado al respecto, dice "*no, estoy queriendo igualar*" (se refiere a comparar) mientras cambia = por ==.

```
if (arre[3] % 2 == 0) {
    printf ("El valor de la celda 3 es par");
                      }
else {
    printf ("El valor de la celda 3 es impar");
     }
```

Fragmento 3: Su primera versión es la siguiente. No asigna una expresión booleana a la variable, sino que hace uso de `if/else` para asignar el resultado según se cumpla o no la condición que compara los valores en las celdas:

```
boolean iguales;
if (arre[0] == arre[7]) {
     iguales = True;
   }
else {
     iguales = False;
     }
```

Durante la escritura, intenta hacer uso de `printf`. Se le recuerda que se le pide asignar el resultado a una variable en vez de mostrar por pantalla, tras lo cual tacha y cambia por el uso de la variable `iguales`, resultando en la versión presentada. Su versión es correcta, en el sentido de que resuelve el problema, pero está atada a la instancia concreta manipulada, en la cual trabaja con un arreglo de ocho celdas y utiliza los índices 0 y 7 para referirse a la primera y última, respectivamente. Consultado sobre qué poner si fuese otra cantidad de celdas (por ejemplo, 20 o 30) dice "*N-1… lo que te hace es que vayas siempre a esa última celda*" (refiriéndose a que sin importar el valor de N, accede a la última celda). Conforme a lo concluido por Matalon, la sucesiva repetición de una acción *genérica* (en este caso acceder a la *última* celda), permite al estudiante lograr el salto de casos particulares al caso *general*.

```
boolean iguales;
if (arre[0] == arre[N-1]) {
     iguales = True;
   }
else {
     iguales = False;
     }
```

Fragmento 4: Su primera versión es la siguiente:

```
int a = 0;
while (a <= N-1){
   printf ("\n%d", arre[a]);
}
```

Esta versión no tiene errores de sintaxis ni de semántica, pero corresponde a una iteración que nunca termina. Tras recurrir a la automatización, detecta el problema tras mostrar tres veces seguidas el valor de la celda 0. Dice "*ahí te queda en loop*" (refiriéndose a un *loop infinito*). Consultado acerca de cómo haría para evitar el *loop*, responde "*podrías hacer un a++, que sume 1*", tras lo cual escribe una segunda versión:

```
int a = 0;
while (a, a++, a <= N-1){
   printf ("\n%d", arre[a]);
}
```

Su segunda versión tiene un error de sintaxis, al escribir tres expresiones diferentes como condición de la estructura `while` (separadas por comas). El estudiante no lo dice en voz alta, pero lo que escribe sugiere que piensa en utilizar `for` ya que lo escrito parece asemejarse más a la sintaxis de dicha estructura iterativa. Se le hace notar que, así como está escrito, daría error de compilación. Tras repasar las tarjetas con la sintaxis de cada estructura, procede a escribir una tercera versión:

```
int a;
for (a = 0; a <= N-1; a++){
   printf ("\n%d", arre[a]);
}
```

Efectivamente, cambia el uso de `while` por `for`. Tras volver a la automatización, constata que funciona correctamente. Se procede entonces a consultarle en qué situación resulta adecuado usar `while` (dice "*cuando no sabemos cuántos elementos tenés*") y `for` (dice "*cuando sabés cuántos tenés*"). Si bien ambas estructuras permiten resolver el problema, en clase (previo al estudio) se discutió que resulta más adecuado usar `for` cuando se sabe la cantidad de elementos a procesar en la iteración, pero que `while` permite igualmente resolver el problema. No obstante, el estudiante no elige usar `for` porque sea una mejor práctica, sino porque no logra hacer que funcione bien con `while`. Al fallar en su intento de usar `while`, en realidad se pasa a `for` buscando una alternativa para resolver el problema.

Tras mencionar lo anterior al estudiante, se le pide que igualmente vuelva a intentar resolverlo con `while`, para que compruebe por sí mismo que es posible hacerlo. Se repasa nuevamente la sintaxis de la estructura `while` (haciendo énfasis en que no es posible poner tres condiciones separadas por comas, como intentó hacer en su segunda versión). Mueve la instrucción `a++` hacia el bloque entre llaves, quedando así:

```
int a = 0;
while (a <= N-1){
   printf ("\n%d", arre[a]);
   a++;
}
```

Tras volver a la automatización, constata que esta versión funciona correctamente, al igual que la versión con `for`. Consultado acerca de por qué termina (sin quedar en *loop*) responde "*va a parar cuando... después de que haga la suma del 7, esto sea 8*" (refiriéndose a la última vez que repite la iteración) "*y que 8, digamos, no cumpla la condición de que sea igual a 7*" (refiriéndose a que para porque deja de cumplirse).

Pablo P (*C*)

Fragmento 1: Escribe una única versión, la cual es correcta:

```
suma = arre[0] + arre[1];
```

Fragmento 2: Escribe una única versión, la cual es correcta:

```
if (arre[2] % 2 == 0)
{
    printf ("es par");
}
else
{
    printf ("es impar");
}
```

Fragmento 3: Su primera versión es la siguiente. Inicializa la variable con `false` y no incluye `else`. No asigna una expresión booleana a la variable, sino expresamente `true` o `false` según se cumpla o no la condición que compara los valores en las celdas.

```
boolean iguales = false;
if (arre[0] == arre[7])
{
    iguales = true;
}
```

Esta versión es correcta, en el sentido de que resuelve el problema, pero está atada a la instancia concreta manipulada, en la cual trabaja con un arreglo de ocho celdas y utiliza los índices 0 y 7 para referirse a la primera y última, respectivamente. Consultado acerca de cómo haría si fuese otra cantidad de celdas (por ejemplo, 20) dice "*habría que cambiar acá, el índice... el número de índice de la última celda*" (señala el 7, tacha y lo cambia por 19).

```
boolean iguales = false;
if (arre[0] == arre[19])
{
    iguales = true;
}
```

Si en vez de 20 fuesen 30, dice "*habría que tachar acá y poner 29*".

```
boolean iguales = false;
if (arre[0] == arre[29])
{
    iguales = true;
}
```

Se le muestra la declaración del arreglo y se le consulta qué tendría que poner para expresar el índice de la última celda, sin tener que cambiar el valor cada vez, a lo que responde "*que esto fuera... arre[N-1]*" (siendo N la constante de la declaración) "*no importa si... tu N va a ser 8, va a ser 12, o va a ser 20... siempre va a ser la última celda*" Tacha `arre[29]` y cambia por `arre[N-1]`, resolviendo ahora el problema en forma general, sin importar la cantidad de celdas del arreglo manipulado en la instancia concreta. Conforme a lo concluido por Matalon, la sucesiva repetición de una acción *genérica* (en este caso acceder a la *última* celda), permite al estudiante lograr el salto de casos particulares al caso *general*.

```
boolean iguales = false;
if (arre[0] == arre[N-1])
{
    iguales = true;
}
```

Fragmento 4: Escribe una única versión, la cual es correcta:

```
for (i = 0; i < N; i++)
{
    printf ("%d", arre[i]);
}
```

Consultado sobre la elección de `for` para resolver el problema, dice "*porque vamos a recorrer todas las celdas y sabemos cuántas celdas hay*". Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Tomás (*C*)

Fragmento 1: Escribe una única versión, la cual es correcta:

```
int suma;
suma = arre[0] + arre[1];
```

Fragmento 2: Escribe una única versión, la cual es correcta:

```
if (arre[3] % 2 == 0)
    printf ("Es par");
else
    printf ("Es impar");
```

Fragmento 3: Escribe una única versión, la cual es correcta:

```
boolean iguales;
if (arre[0] == arre[N-1])
    iguales = Verdadero;
else
    iguales = Falso;
```

Inicialmente empieza escribiendo `arre[7]`. Al momento de hacerlo, consulta "*¿en este caso yo sé qué tienen ocho?*" (refiriéndose a si son ocho celdas en el arreglo). Como respuesta, se le pregunta qué pasaría si fuera otra cantidad (por ejemplo, 10). Dice "*hasta 9*" (refiriéndose al índice de la última celda en ese caso). Para 20, responde "*19*". Cuando se le pregunta qué pasaría para un valor cualquiera para N, dice "*N-1*", y termina de escribir su versión con N-1. Conforme a lo concluido por Matalon, la sucesiva repetición de una acción *genérica* (en este caso acceder a la *última* celda), permite al estudiante lograr el salto de casos particulares al caso *general*. No asigna una expresión booleana a la variable, sino que asigna expresamente `true` o `false` según se cumpla o no la condición de la estructura `if` (pone `Verdadero` o `Falso`, en español, dado que el lenguaje *C* permite definir `boolean` como un tipo enumerado, de modo que el programador elige los nombres para las constantes del tipo).

Fragmento 4: Escribe una única versión, la cual es correcta:

```
for (i = 0; i < N; i++)
{
    printf ("%d", arre[i]);
}
```

Consultado acerca de la elección de `for`, Inicialmente elige `while` (antes de escribir). Cuando se le pregunta por qué, responde "*porque se tendría que imprimir para cada celda*". Se le señala que dice *para cada* y se le muestran nuevamente las tarjetas que ilustran la sintaxis de las estructuras en pseudocódigo. Si fuera en pseudocódigo, se le

pregunta cuál elegiría y señala la tarjeta con la sintaxis de *Para cada*. Se le pregunta entonces a cuál estructura en C corresponde *Para cada* (se le muestran de nuevo las tarjetas con la sintaxis en C), y señala la tarjeta que corresponde a `for` (la cual finalmente elige y utiliza en la versión que escribe).

Valeria (*C*)

Fragmento 1: Escribe una única versión, la cual es correcta. Si bien no es necesario, inicializa la variable suma con 0 antes de realizar la asignación pedida.

```
int suma = 0;
suma = arre[0] + arre[1];
```

Fragmento 2: Escribe una única versión, la cual es correcta en el sentido de que resuelve el problema. En el lenguaje C, las palabras `if` y `else` se ponen en minúsculas, pero ella usa mayúscula para la primera letra de cada una. Esto daría un error de sintaxis al compilar, pero no se le menciona, ya que se trata luego en el pasaje del código escrito en papel a la computadora (tercera actividad).

```
If (arre[3] % 2 == 0){
    printf ("La celda 3 de arre es par");
                      }
Else {
    printf ("La celda 3 de arre es impar");
  }
```

Fragmento 3: Su primera versión es la siguiente:

```
boolean iguales;
If (arre[0] == arre[7]){
      iguales = true;
               }
Else {
      iguales = false;
    }
```

Esta versión es correcta, en el sentido de que resuelve el problema, pero está atada a la instancia concreta manipulada, en la cual trabaja con un arreglo de ocho celdas y utiliza los índices 0 y 7 para referirse a la primera y última, respectivamente. Consultada acerca de cómo haría si fuese otra cantidad de celdas (por ejemplo, 20) dice "*cambiar acá, este índice*" (señala el 7). Se le muestra la declaración del arreglo y se le pregunta qué pondría para expresar el índice de la última celda, sin tener que cambiar el valor cada vez, a lo que dice "*poner N-1*". Conforme a lo concluido por Matalon, la sucesiva repetición de una acción *genérica* (en este caso acceder a la *última* celda), permite a la estudiante lograr el salto de casos particulares al caso *general*. No asigna una expresión booleana a la variable, sino que usa `if/else` para asignar el resultado según se cumpla o no la condición que compara los valores en las celdas.

```
boolean iguales;
If (arre[0] == arre[N-1]){
      iguales = true;
               }
Else {
      iguales = false;
    }
```

Fragmento 4: Escribe una única versión, la cual es correcta:

```
for (i = 0; i < N; i++)
  {
    printf ("%d", arre[i]);
  }
```

Durante la escritura, inicialmente pone N-1 en vez de N. Duda un instante, tacha y deja N, quedando la versión presentada. Consultada acerca de la elección de `for`, responde "acá *me pide que recorra todo, no es como cuando buscaba la cédula*" (se refiere la versión final de su algoritmo en pseudocódigo, en la cual usa *Mientras* para realizar la búsqueda y parar cuando encuentra el documento). Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Ximena (*C*)

Fragmento 1: Escribe una única versión, la cual es correcta:

```
int suma;
suma = arre[0] + arre[1];
```

Fragmento 2: Escribe una única versión, la cual es correcta:

```
if (arre[3] % 2 == 0)
    printf ("es par");
else
    printf ("es impar");
```

Fragmento 3: Su primera versión es la siguiente:

```
boolean iguales;
if (arre[0] == arre[7])
    boolean iguales = TRUE;
else
    boolean iguales = FALSE;
```

En esta versión, pretende declarar tres veces la variable `iguales`. La estudiante no lo sabe, pero en realidad declara tres variables con el mismo nombre, una global y otras dos locales a las cláusulas `if` y `else` (las nociones de *local* y *global* aún no han sido vistas en clase). Sí ha visto que toda variable debe declararse *una* vez, pero aún muestra dificultad en su conceptualización. Se le pregunta cuántas veces corresponde declarar la variable y dice "*una… y acá las asigné*" (señala las asignaciones con `TRUE` y `FALSE`). No tiene claro que al poner la palabra `boolean` en ambas, declara dos nuevas

variables, además de realizar las asignaciones. Piensa que al tener el mismo nombre, se trata de la misma variable las tres veces. Se le menciona esto, a lo que pregunta "*¿qué decís, que le saque el boolean?*" (se refiere a la palabra `boolean` en las dos declaraciones adicionales). Se le consulta por qué razón habría que sacarlas, y contesta "*porque ya se sabe que ésto es ésto*" (señala las dos declaraciones adicionales y luego la declaración inicial, dando a entender que se trata de la misma variable en los tres casos). Luego dice "*sólo pongo el nombre*" y tacha `boolean` en ambas.

```
boolean iguales;
if (arre[0] == arre[7])
    iguales = TRUE;
else
    iguales = FALSE;
```

Esta versión es correcta, en el sentido de que resuelve el problema, pero está atada a la instancia concreta manipulada, en la cual trabaja con un arreglo de ocho celdas y utiliza los índices 0 y 7 para referirse a la primera y última, respectivamente. Consultada acerca de cómo haría si fuese otra cantidad de celdas (por ejemplo, 20) dice "*cambiaría por 0 y 19*".

```
boolean iguales;
if (arre[0] == arre[19])
    iguales = TRUE;
else
    iguales = FALSE;
```

Si en vez de 20 fuesen 30, dice "*29*", y cambia nuevamente.

```
boolean iguales;
if (arre[0] == arre[29])
    iguales = TRUE;
else
    iguales = FALSE;
```

Se le muestra la declaración del arreglo y se le consulta qué tendría que poner para expresar el índice de la última celda, sin tener que cambiar el valor cada vez, a lo que responde "*esto - 1*" (se refiere a 30 - 1) y cambia nuevamente.

```
boolean iguales;
if (arre[0] == arre[30-1])
    iguales = TRUE;
else
    iguales = FALSE;
```

Se le señala que sigue escribiendo su código en función del valor concreto y se le consulta si hay alguna forma de no depender de dicho valor concreto. Contesta: "*puedo poner N-1*" tras lo cual tacha y lo modifica por última vez. Conforme a lo concluido por Matalon, la sucesiva repetición de una acción *genérica* (en este caso acceder a la *última* celda), permite a la estudiante lograr el salto de casos particulares al caso *general*.

```
boolean iguales;
if (arre[0] == arre[N-1])
    iguales = TRUE;
else
    iguales = FALSE;
```

Fragmento 4: Su primera versión es la siguiente:

```
for (i = 0; i < N-1; i++)
{
    printf ("%d", arre[i]);
}
```

Esta versión presenta un *problema de borde*, pues deja sin mostrar el contenido de la última celda. El hecho de que los índices en *C* empiezan en cero, lleva en ocasiones a los estudiantes a expresar incorrectamente el límite superior de la iteración al recorrer las N celdas. Lo constata al volver a la automatización, sobre su instancia concreta de 8 celdas. Al llegar a la celda con índice 7 dice "*ese no se muestra*", tras lo cual modifica.

```
for (i = 0; i ≤ N-1; i++)
{
    printf ("%d", arre[i]);
}
```

Utiliza el símbolo ≤ en vez de <=. Al escribirlo, dice "*igual, esto no funciona en el programa*", dando a entender que el símbolo ≤ no está definido en *C*. Se le dice que se puede corregir al momento de pasarlo a máquina. Excepto por eso, la versión es correcta. Consultada acerca de la elección de `for` (cuando está a punto de escribir su primera versión) elige la tarjeta que ilustra la sintaxis de dicha estructura y pregunta "*pero sabemos cuántas celdas hay, si?*" a lo que se le contesta que son N y responde "*ta, entonces usaría el for*". Comprende que necesita una estructura iterativa adecuada para iterar sobre un rango conocido de índices.

Anexo D

Programa para búsqueda lineal (segunda parte: *inter* → *trans*)

En este anexo se presentan las versiones del *programa* que resuelve el problema de la búsqueda lineal sobre un arreglo de enteros, escritas por los trece estudiantes durante la tercera actividad de la segunda parte del estudio (pasaje de la etapa *inter* a la etapa *trans*). Se presentan *todas* las versiones escritas por cada estudiante junto con un análisis detallado de cada una y del proceso realizado. En cada análisis, se usan cursivas entre comillas para citar respuestas orales de los estudiantes durante las entrevistas y cursivas sin comillas para citar partes concretas escritas en sus versiones previas en pseudocódigo. Las entrevistas fueron grabadas y se tomaron fotografías de las hojas de papel conteniendo las distintas versiones del programa escritas por los estudiantes, las cuales fueron transcritas aquí. También se guardaron los archivos fuentes resultantes de las transcripciones a computadora hechas por los estudiantes.

Aaron (*Pascal*)

Escribe una única versión del programa, la cual es correcta:

```
Puerta := 1;
cedula := arre[1];
While Not (cedula = ci) AND (Puerta < N) DO
Begin
    Puerta := Puerta + 1;
    cedula := arre[Puerta];
end
```

Esta versión se corresponde en forma acorde con su última versión en pseudocódigo:

Ver la C.I de la puerta 1
Mientras NOT (*sea la que busco*) *AND* (*tengo mas Puertas*)
    Veo siguiente C.I
Fin

Mantiene la estrategia de búsqueda usada en pseudocódigo (*a priori*), obteniendo el valor de la primera celda antes de iniciar la iteración y quedando `Puerta` con el valor del índice de la última celda al detener la iteración tanto cuando el valor buscado está en ella como cuando no existe. Debido a la estrategia utilizada, no se produce error de

salida de rango. Dado que ninguna de sus condiciones combinadas con `AND` en la estructura `while` pretende acceder a una celda del arreglo, es irrelevante si la evaluación se realiza por *circuito corto* o por *circuito completo*. Mantiene la estructura de control *Mientras* (la cual traduce a while), el orden de las condiciones y su combinación mediante *AND*, así como el orden de las instrucciones. Necesita usar dos instrucciones para expresar el paso *Ver la C.I de la puerta 1*, una para posicionarse en el índice de la celda (`Puerta := 1`) y otra para obtener su contenido (`cedula := arre[1]`). Lo mismo para el paso *Veo siguiente C.I*.

Tras recurrir a la automatización y constatar que funciona correctamente, procede a transcribirlo a la computadora y compilarlo. Al hacerlo, el compilador señala un error en la línea siguiente a la palabra `end` que cierra el bloque `while`. El mensaje dice:

```
Syntax error, ";" expected but "IF" found
```

El punto y coma debería estar como separador entre el bloque `while` y la instrucción que el estudiante pone después (`if`) para emitir al usuario un mensaje indicando si encontró o no el valor buscado. Consultado respecto del error, dice "*no cerré el while*". Se le hace notar que sí lo cierra (puso `end`) y se lee nuevamente junto a él el mensaje de error. Contesta "*me faltó este punto y coma*" y lo agrega, tras lo cual vuelve a compilar (esta vez con éxito). Posteriormente, procede a ejecutarlo dos veces en computadora, una para buscar un número que existe y otra para buscar otro que no, constatando que funciona bien en ambos casos. La instrucción (escrita directamente en la computadora) que el estudiante pone luego del bloque `while` para mostrar el resultado es la siguiente sentencia `if/else`:

```
if (cedula = ci) then
    writeln('la c.i fue encontrada')
else
    writeln('la c.i no fue encontrada');
```

El estudiante elige hacer uso de la variable `cedula` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `cedula` siempre queda con el valor de la última celda visitada durante la recorrida, por lo que siempre puede comparar su valor con el valor buscado (`ci`) para emitir un mensaje indicando si estaba o no en el arreglo.

Finalmente, el pensamiento del estudiante profundiza en las transiciones hacia *newC* y *newC'*. Escribe un programa acorde a la lógica de su algoritmo en pseudocódigo. Expresa cambios en objetos, que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`Not (cedula = ci)`) y (`Puerta < N`), combinadas con `AND`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`cedula = ci`), avance (`Puerta := Puerta + 1`) y repetición (`while`).

Ignacio D (*Pascal*)

Escribe una única versión del programa, la cual es correcta:

```
cedula := arre[1];
puerta := 1;
while (cedula <> cedulaAbuscar) and (puerta ≤ N) do
Begin
    cedula := arre[puerta];
    puerta := puerta + 1;
End
```

Esta versión se corresponde en forma acorde con su última versión en pseudocódigo:

Mientras (*no haya encontrado a la persona*) *AND* (*no se hayan acabado las puertas*)
    Pregunto en la puerta que estoy quien vive
    Avanzo a la siguiente puerta
Fin

Técnicamente, mantiene la misma estrategia de búsqueda usada en pseudocódigo (*a posteriori*), obteniendo el valor de la primera celda dentro de la iteración y quedando puerta con el valor del índice siguiente a la última celda al detener la iteración cuando el valor buscado no existe. Sin embargo, la estructura de su código parece más *a priori*. De hecho, obtiene el valor de la primera celda *antes* de iniciar la iteración, aunque vuelve a hacerlo por segunda vez *adentro*, como los demás que emplean a *posteriori*. Lo hace así para que cedula esté inicializada con un valor válido al momento de evaluar por primera vez la condición (`cedula <> cedulaAbuscar`). En cada una de las demás celdas, obtiene el valor almacenado solo una vez. No se produce error de *salida de rango* porque la segunda condición de `while` corta la iteración ni bien el valor de puerta supera el último índice válido, sin llegar a intentar acceder a una celda con índice inválido. Dado que ninguna de sus condiciones combinadas con `and` en la estructura `while` intenta acceder a una celda del arreglo, es irrelevante si la evaluación se realiza por *circuito corto* o por *circuito completo*. Mantiene la estructura de control *Mientras* (la cual traduce a `while`), el orden de las condiciones y su combinación mediante *AND*, así como el orden de las instrucciones.

Tras recurrir a la automatización, constata que funciona correctamente y procede a transcribirlo a la computadora. Compila y no se producen errores. Al transcribir, cambia el símbolo (≤) usado en papel por el operador <= de *Pascal*. Luego del bloque `while`, agrega la siguiente sentencia `if/else` (directamente en computadora) para mostrar el resultado:

```
if (puerta > N) then
    write ('no se encontró la persona')
else
    write ('se encontró la persona');
```

El estudiante elige hacer uso de la variable `puerta` para decidir qué mensaje emitir. Si bien la búsqueda funciona correctamente, esto produce un *problema de borde*, debido a la condición que define en esta nueva sentencia `if`. En caso de que el número buscado esté en la *última* celda, el mensaje es incorrecto. En ese caso, `puerta` finaliza con valor N+1, por lo cual el mensaje mostrado dice que no se encontró a la persona, a pesar de que el valor está en la última celda. Se recurre a la automatización, el estudiante constata el problema y decide cambiar la condición, haciendo uso de la variable `cedula`:

```
if (cedula = cedulaAbuscar) then
    write ('no se encontró la persona')
else
    write ('se encontró la persona');
```

Tras el cambio, procede a compilar nuevamente (no se producen errores) y ejecutarlo. Al probar con algunos valores que están en el arreglo, el mensaje mostrado en cada caso es que el valor no se encuentra en él. A este respecto, dice "*ya sé por qué... ahí en el if, cuando cambiamos la condición, tendríamos que haber cambiado los mensajes*". Los intercambia, quedando así:

```
if (cedula = cedulaAbuscar) then
    write ('se encontró la persona')
else
    write ('no se encontró la persona');
```

Tras volver a compilar con éxito, procede a ejecutarlo varias veces en computadora, algunas para buscar un número que existe y otras para buscar otro que no, constatando que funciona bien en todos los casos. Debido a cómo estructura su búsqueda en el bloque `while`, `cedula` siempre queda con el valor de la última celda visitada durante la recorrida, por lo tanto el estudiante elige ahora comparar su valor con el valor buscado (dado en `cedulaAbuscar`) para emitir un mensaje indicando si estaba o no.

Finalmente, el pensamiento del estudiante profundiza en las transiciones hacia *newC* y *newC'*. Escribe un programa acorde a la lógica de su algoritmo en pseudocódigo. Expresa cambios en objetos, los que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`cedula <> cedulaAbuscar`) y (`puerta <= N`), combinadas con `and`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`cedula <> cedulaAbuscar`), avance (`puerta := puerta + 1`) y repetición (`while`).

Ignacio U (*Pascal*)

Su primera versión del programa es la siguiente:

```
nocedula := False;
contador := 1;
while (nocedula = False) and not (contador = 8) do
    if (arre[contador] <> cedula) then
        contador := contador + 1
    else
        nocedula := True;
```

Esta versión se corresponde en forma acorde con su última versión en pseudocódigo:

nocedula := FALSE
contador := 1
Mientras (*nocedula = FALSE*) *AND NOT* (*contador = 8*)
    Analizo si en puerta contador está la cédula
    Si (*no está*)
        contador := contador + 1
    Sino
        nocedula := TRUE
    Fin
Fin

Mantiene la estrategia de búsqueda usada en pseudocódigo (*a posteriori*), obteniendo el valor de la primera celda dentro de la iteración y quedando `contador` con el valor del índice siguiente a la última celda al detener la iteración cuando el valor buscado no existe. No se produce error de *salida de rango* porque la segunda condición de la estructura `while` corta la iteración ni bien el valor de `contador` supera el último índice válido, sin llegar a intentar acceder a una celda con índice inválido. Dado que ninguna de sus condiciones, combinadas con `and`, en la estructura `while` pretende acceder a una celda del arreglo, es irrelevante si la evaluación se realiza por *circuito corto* o *circuito completo*. Mantiene la estructura de control *Mientras* (la cual traduce a `while`), el orden de las condiciones y su combinación con *AND*, así como el orden de las instrucciones. Unifica en la condición (`arre[contador] <> cedula`) dos pasos separados del pseudocódigo (*Analizo si en puerta contador está la cédula* y *Si* (*no está*)).

Tras volver a la automatización y constatar que funciona correctamente, procede a transcribirlo a la computadora y compilar (no se producen errores). Su versión es correcta, en el sentido de que resuelve el problema, pero sigue atada a la cantidad concreta de 7 puertas manipulada durante la escritura de la versión en pseudocódigo. Se le señala la constante N y se le pregunta cómo haría para adaptar su código, de modo que funcione cualquiera sea el valor de N. Contesta "*cambiar 8 por N+1* ", lo cual hace, quedando la siguiente versión (correcta). El salto de casos particulares al caso *general* había sido trabajado en la tercera actividad, por lo que el estudiante no presenta dificultad al realizar el cambio.

```
nocedula := False;
contador := 1;
while (nocedula = False) and not (contador = N+1) do
    if (arre[contador] <> cedula) then
        contador := contador + 1
    else
        nocedula := True;
```

Tras volver a compilar con éxito, lo ejecuta varias veces en máquina, algunas para buscar una cédula que existe y otras para buscar otra que no, constatando que funciona bien en todos los casos. La instrucción (escrita directo en la computadora) que pone luego del bloque `while` para mostrar el resultado es la siguiente sentencia `if/else`:

```
if nocedula = true then
    writeln('la cedula esta')
else
    writeln('la cedula no esta');
```

El estudiante elige usar la variable `nocedula` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `nocedula` queda con valor `true` cuando el valor buscado (dado en `cedula`) está en el arreglo y `false` cuando no, por lo tanto siempre puede usar su valor para emitir un mensaje indicando si está o no. Si bien la condición es correcta, la comparación con `true` es innecesaria (alcanza con poner `nocedula` como condición). Esta característica ha sido vista repetidamente en clase. Muchos estudiantes presentan dificultad en conceptualizar que una variable booleana es por sí misma una expresión booleana, sin ser necesaria su comparación con `true`. Ya usaban operadores de comparación en matemática, previo a conocer el tipo `boolean` y tienden a repetir dicha práctica al manipular variables booleanas, queriendo *generalizar* al lenguaje de *programación* su conocimiento de lenguaje *matemático* (la herramienta cognitiva que interviene es la generalización *inductiva*).

Finalmente, el pensamiento del estudiante profundiza en las transiciones hacia *newC* y *newC*'. Escribe un programa acorde a la lógica de su algoritmo en pseudocódigo. Expresa cambios en objetos, los que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`nocedula = False`) y (`not (contador = N+1)`), combinadas con `and`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`arre[contador] <> cedula`), avance (`contador := contador + 1`) y repetición (`while`).

Joaquín (*Pascal*)

Su primera versión del programa es la siguiente:

```
PUERTA := 1;
x := ARRE[1];
WHILE (x <> NUMERO) AND ARRE[i] <> ARRE[N]
    i := i+1; x := ARRE[i]
END
```

Esta versión intenta corresponderse en forma acorde con su última versión en pseudocódigo, pero tiene errores, dos de sintaxis y uno de interpretación semántica. Los de sintaxis son que pretende usar una variable llamada `puerta` para iterar sobre los índices del arreglo, pero en la iteración usa otra llamada `i`, y que utiliza `end` para cerrar el bloque `while` sin haber usado `begin` (y además falta la palabra `do`). El error de interpretación semántica es que confunde los índices con los valores contenidos en las celdas en su segunda condición (`ARRE[i] <> ARRE[N]`), en vez de (`i <> N`). Quizás hace una lectura propia de una persona (*desde la celda 1 hasta la celda N*). Necesita adaptar su pensamiento para expresarla teniendo en cuenta que será evaluada por la computadora en vez de por una persona. Debe comprender que la comparación debe hacerse entre los *índices* del arreglo. Necesita conceptualizar la diferencia entre los *índices* y los *valores* contenidos en las celdas:

puerta := 1
abrir puerta := x
número := 417
Mientras (*x <> número*) *AND* (*no es la última puerta*)
    puerta := puerta + 1
    abrir puerta := x
Fin

Consultado al respecto del primer error, dice "*acá tiene que ir puerta… puerta… puerta*" (señala cada lugar donde usa `i`, indicando que debe cambiar por `puerta` en cada uno). Se le sugiere cambiar `puerta` por `i` en la primera asignación, a efectos de cambiar una sola vez. En cuanto a los otros errores, dice "*tengo que comparar el índice*" y "*me faltó el begin*". Escribe una segunda versión donde corrige dichos errores:

```
i := 1;
x := ARRE[1];
WHILE (x <> NUMERO) AND (i <> N) DO
BEGIN
    i := i+1;
    x := ARRE[i]
END
```

Mantiene la estrategia de búsqueda usada en pseudocódigo (*a priori*), obteniendo el valor de la primera celda antes de iniciar la iteración y quedando `i` con el valor del índice de la última celda al detener la iteración tanto cuando el valor buscado está en ella como cuando no existe. Debido a la estrategia usada, no se produce error de *salida de rango*. Dado que ninguna de sus condiciones, combinadas con `AND`, en la estructura `while` pretende acceder a una celda del arreglo, es irrelevante si la evaluación se realiza por *circuito corto* o por *circuito completo*. Mantiene la estructura de control *Mientras* (la cual traduce a `while`), el orden de las condiciones y su combinación mediante *AND*, así como el orden de las instrucciones. Además, se independiza de la instancia concreta manipulada en pseudocódigo al quitar la asignación *número := 417* (se le dice que el valor a buscar ya se encuentra cargado en la variable).

Tras volver a la automatización y constatar que funciona correctamente, procede a transcribirlo a la computadora (al hacerlo, cambia a minúsculas lo que tenía en mayúsculas). Lo compila (no se producen errores) y lo ejecuta varias veces, tanto para buscar valores que existen como valores que no, constatando que funciona bien en todos los casos. La instrucción (escrita directamente en la computadora) que pone luego del bloque `while` para mostrar el resultado es la siguiente sentencia `if/else`:

```
if x = numero then
    writeln ('La cedula fue encontrada')
else
    writeln ('La cedula no fue encontrada');
```

El estudiante elige hacer uso de la variable `x` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `x` siempre queda con el valor de la última celda visitada durante la recorrida, por lo tanto siempre puede comparar su valor con el valor buscado (dado en `numero`) para emitir un mensaje indicando si estaba o no en el arreglo.

Finalmente, el pensamiento del estudiante profundiza en las transiciones hacia *newC* y *newC'*. Escribe un programa acorde a la lógica de su algoritmo en pseudocódigo. Expresa cambios en objetos, los que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`x <> NUMERO`) y (`i <> N`), combinadas con `AND`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`x <> NUMERO`), avance (`i := i+1`) y repetición (`while`).

Juan (*C*)

Su primera versión del programa es la siguiente:

```
boolean encontre = FALSE;
int i = 0;
while ((encontre = FALSE) && (i ≤ N-1)) {
    if (arre[i] == cedula) {
        encontre = TRUE;
    } else {
        i = i+1;
    }
}
```

Al escribir, inicialmente pone (`arre[i] != cedula`) como primera condición de su estructura `while`, pero decide tacharla y cambia por (`encontre = FALSE`) resultando en la versión mostrada (ni siquiera completa la versión con (`arre[i] != cedula`), cambia de opinión a medida que escribe). Esta versión intenta corresponderse en forma acorde con su última versión en pseudocódigo, pero tiene un error de semántica. Usa el operador de asignación (=) en vez de comparación (==) en la condición (`encontre = FALSE`). *C* interpreta eso como una asignación válida del lenguaje, pero no es lo que el estudiante pretende hacer. Este es un error común al escribir en *C*, dado que en matemática se usa = para expresar comparación. El estudiante *generaliza* al

lenguaje de *programación* su conocimiento del lenguaje *matemático* (la herramienta cognitiva que interviene es la generalización *inductiva*).

Mientras ((*no encuentro nro cedula*) *AND* (*hay puertas*))
    Golpeo puerta
    Pregunto por cedula
    Verifico si es la buscada
    Si es
        Paro
    Sino
        Sigo a la siguiente puerta
    Fin
Fin

Se le pregunta por el operador de asignación que usa en la condición y responde "*acá le estoy asignando el valor... es comparar*". Lo modifica, quedando la siguiente versión (que es correcta):

```
boolean encontre = FALSE;
int i = 0;
while ((encontre == FALSE) && (i ≤ N-1)) {
    if (arre[i] == cedula) {
        encontre = TRUE;
    } else {
        i = i+1;
    }
}
```

Mantiene la estrategia de búsqueda usada en pseudocódigo (*a posteriori*), obteniendo el valor de la primera celda dentro de la iteración y quedando `i` con el valor del índice siguiente a la última celda al detener la iteración cuando el valor buscado no existe. No se produce error de *salida de rango* porque la segunda condición de la estructura while corta la iteración ni bien el valor de `i` supera el último índice válido, sin tratar de acceder a una celda con índice inválido. Dado que ninguna de sus condiciones, combinadas con `&&` en la estructura `while`, pretende acceder a una celda del arreglo, es irrelevante si la evaluación se realiza por *circuito corto* o *circuito completo*. Mantiene la estructura de control *Mientras* (la traduce a `while`), el orden de las condiciones y su combinación mediante *AND*, así como el orden de las instrucciones. Unifica en la condición (`arre[i] == cedula`) cuatro pasos separados del pseudocódigo (*Golpeo puerta*, *Pregunto por cedula*, *Verifico si es la buscada* y *Si es*).

No necesita volver a la automatización para constatar que funciona bien (explica oralmente y de manera clara y correcta por qué funciona y en qué casos detiene la búsqueda), por lo que procede a transcribirlo a la computadora y compilar (no se producen errores). Al hacer la transcripción, cambia el símbolo (≤) usado en papel por el operador <= de *C*. Lo ejecuta dos veces, para buscar un valor que existe y otro que no, constatando que funciona bien en ambas. La instrucción (escrita directamente en

la computadora) que pone luego del bloque `while` para mostrar el resultado es la siguiente sentencia `if/else`:

```
if (encontre == FALSE) {
    printf ("No encontre la cedula");
} else {
    printf ("Encontre la cedula");
}
```

El estudiante elige hacer uso de la variable `encontre` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `encontre` queda con el valor `TRUE` cuando el valor buscado (dado en `cedula`) está en el arreglo y `FALSE` cuando no, por lo tanto siempre puede usar su valor para emitir un mensaje indicando si el número estaba o no. Si bien es correcta, la comparación con `FALSE` puede expresarse poniendo `!encontre` como condición). Esta característica ha sido vista repetidamente en clase. Muchos estudiantes presentan dificultad en conceptualizar que una variable booleana antecedida de una negación es por sí misma una expresión booleana, sin necesitar comparar con `FALSE`. Ya usaban operadores de comparación en matemática, previo a conocer el tipo `boolean` y tienden a repetir dicha práctica al manipular variables booleanas, queriendo *generalizar* al lenguaje de *programación* su conocimiento de lenguaje *matemático* (nuevamente, generalización *inductiva*).

Finalmente, el pensamiento del estudiante profundiza en las transiciones hacia *newC* y *newC'*. Su programa es acorde a la lógica de su algoritmo en pseudocódigo. Expresa cambios en objetos, los que se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`encontre == FALSE`) y (`i <= N-1`), combinadas con `&&`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`arre[i] == cedula`), avance (`i = i+1`) y repetición (`while`).

Martín (*Pascal*)

Su primera versión del programa es la siguiente:

```
i := 1;
ci := arre[1];
while (cedula <> arre[i]) and (i ≤ n) do
begin
    i := i+1;
    ci := arre[i];
end
```

Esta versión intenta corresponderse en forma acorde con su última versión en pseudocódigo, pero tiene un error de *salida de rango*. En caso de que el valor buscado no esté en el arreglo, tras consultar la última celda, la instrucción `i := i+1` genera que `i` tome el valor `N+1`, quedando fuera del rango de índices válidos. Al pretender acceder a `arre[i]` en la siguiente instrucción (`ci := arre[i]`), se produciría el error en tiempo de ejecución. La construcción de conocimiento relativa al *programa*

implica conceptualizar aspectos propios de la ejecución en máquina, como ser errores de este tipo y la búsqueda de estrategias para evitarlos. Nuevamente, la automatización cumple un rol esencial para su detección y ayudar en la conceptualización.

> *Conozco el número de la persona*
> *Voy a la primera puerta*
> *Golpeo la primera puerta*
> *Mientras* (*el número de la persona sea distinto del que busco*) *AND* (*haya mas puertas*)
> *Paso 1 - Paso a la siguiente puerta*
> *Paso 2 - Golpeo la puerta*
> *Fin*

Tras recurrir a la automatización, el estudiante constata el error cuando se intenta acceder a la celda con índice N+1. Consultado acerca de cómo podría arreglarlo, contesta "*cambiando el menor o igual solo por un menor*". De ese modo, se asegura de que `i` siempre quede con el valor del índice de la última celda al finalizar la iteración (tanto si el valor buscado está en el arreglo, como si no) evitando así salirse de rango. Lo cambia, quedando la siguiente versión (que es correcta):

```
i := 1;
ci := arre[1];
while (cedula <> arre[i]) and (i < n) do
begin
    i := i+1;
    ci := arre[i];
end
```

Mantiene la estrategia de búsqueda usada en pseudocódigo (*a priori*), obteniendo el valor de la primera celda antes de iniciar la iteración y quedando `i` con el valor del índice de la última celda al detener la iteración, tanto cuando el valor buscado está en ella como cuando no existe. Dado que corrigió el error de modo que ya nunca accede a una celda con índice inválido, deja de ser relevante si la evaluación del operador and se realiza por *circuito corto* o *circuito completo*. Mantiene la estructura de control *Mientras* (la cual traduce a `while`), el orden de las condiciones y su combinación mediante *AND*, así como el orden de las instrucciones. Tras volver a la automatización y constatar que funciona correctamente, procede a transcribirlo a la computadora. Compila (no se producen errores) y lo ejecuta varias veces, tanto para buscar valores que existen en el arreglo como valores que no, funcionando bien en todos los casos. La instrucción (escrita directamente en la computadora) que el estudiante pone luego del bloque `while` para mostrar el resultado es la siguiente sentencia `if/else`:

```
if ci = cedula then
    writeln('La persona se encuentra en:', i)
else
    writeln('No se encontro a la persona buscada');
```

El estudiante elige hacer uso de la variable `ci` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `ci` siempre queda con el valor de

la última celda visitada durante la recorrida, por lo tanto siempre puede comparar su valor con el valor buscado (dado en `cedula`) para emitir un mensaje indicando si estaba o no en el arreglo.

Finalmente, el pensamiento del estudiante profundiza en las transiciones hacia *newC* y *newC'*. Su programa es acorde a la lógica de su algoritmo en pseudocódigo. Expresa cambios en objetos, los que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`cedula <> arre[i]`) y (`i < n`), combinadas con `and`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`cedula <> arre[i]`), avance (`i := i+1`) y repetición (`while`).

Mónica (*C*)

Su primera versión del programa es la siguiente:

```
boolean encontre = FALSE;
int i = 0;
while ((!encontre) && (i ≤ 7))
{
    if (arre[i] == CI)
        encontre = TRUE;
    else
        i++;
}
```

Esta versión se corresponde en forma acorde con su última versión en pseudocódigo:

Mientras (*no encuentre la C.I*) *AND* (*la puerta es menor o igual a 7*) *hacer*
    Golpear puerta
    Preguntar C.I
    Si encuentro la C.I entonces
        Termino
    Sino
        Sigo preguntando
    Fin
Fin

Mantiene la estrategia de búsqueda usada en pseudocódigo (*a posteriori*), obteniendo el valor de la primera celda dentro de la iteración y quedando `i` con el valor del índice siguiente a la última celda al detener la iteración cuando el valor buscado no existe. No ocurre error de *salida de rango* porque la segunda condición de la estructura `while` corta la iteración ni bien el valor de `i` supera el último índice válido, sin llegar a intentar acceder a una celda con índice inválido. Dado que ninguna de sus condiciones, combinadas con `&&` en la estructura `while`, intenta acceder a una celda del arreglo, es irrelevante si la evaluación se realiza por *circuito corto* o *circuito completo*. Mantiene la estructura de control *Mientras* (la cual traduce a `while`), el orden de las condiciones y su combinación mediante *AND*, así como el orden de las instrucciones. Unifica en

una sola condición (`arre[i] == CI`) tres pasos separados del pseudocódigo (*Golpear puerta, Preguntar C.I* y *Si encuentro la CI*).

Tras volver a la automatización, constata que funciona correctamente. Su versión es correcta, en el sentido de que resuelve el problema, pero sigue atada a la cantidad concreta de 7 puertas manipulada durante la escritura de la versión en pseudocódigo. Se le hace notar esto, se le señala el cuarto fragmento escrito en la tercera actividad y se le pregunta que se había hecho en él. Contesta "*ah… N-1*". Lo cambia, quedando la siguiente versión (correcta), la cual resuelve el problema cualquiera sea el tamaño del arreglo. El salto de casos particulares al caso *general* había sido trabajado en la tercera actividad, por lo que la estudiante no presenta dificultad al realizar el cambio.

```
boolean encontre = FALSE;
int i = 0;
while ((!encontre) && (i ≤ N-1))
{
    if (arre[i] == CI)
        encontre = TRUE;
    else
        i++;
}
```

Procede a transcribirlo a la computadora y compilarlo (no se producen errores). Al hacer la transcripción, cambia el símbolo (≤) usado en papel por el operador <= de *C*. Lo ejecuta dos veces, para buscar un valor que existe y otro que no, constatando que funciona bien en ambas. La instrucción (escrita directo en la computadora) que pone luego del bloque `while` para mostrar el resultado es la siguiente sentencia `if/else`:

```
if (encontre == TRUE)
    printf ("Encontré la CI");
else
    printf ("No encontré la CI");
```

La estudiante elige usar la variable `encontre` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `encontre` queda con valor `TRUE` cuando el valor buscado (dado en `CI`) está en el arreglo y `FALSE` cuando no, por lo que siempre puede usar su valor para emitir un mensaje indicando si el número está o no. Si bien es correcto, comparar con `TRUE` es innecesario (alcanza con poner `encontre` como condición). Esta característica ha sido vista repetidamente en clase. Muchos estudiantes presentan dificultad en conceptualizar que una variable booleana es por sí misma una expresión booleana, sin necesitar comparar con `TRUE`. Ya comparaban en matemática, previo a conocer el tipo `boolean` y tienden a repetir dicha práctica con variables booleanas, *generalizando* al lenguaje de *programación* su conocimiento de lenguaje *matemático* (herramienta cognitiva: generalización *inductiva*).

Finalmente, el pensamiento de la estudiante profundiza en las transiciones hacia *newC* y *newC'*. Su programa es acorde a la lógica de su algoritmo en pseudocódigo. Expresa cambios en objetos, los que se traducen a cambios sobre el *arreglo*. Para de iterar

cuando alguna de las condiciones (`!encontre`) y (`i <= N-1`), combinadas con `&&`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`arre[i] == CI`), avance (`i++`) y repetición (`while`).

Nicolás (*Pascal*)

Su primera versión del programa es la siguiente:

```
cedula := arre[1];
WHILE (cedula <> x) AND (i < N) DO
BEGIN
    cedula := arre[i+1];
    i := i+1
END;
```

Esta versión intenta corresponderse en forma acorde con su última versión en pseudocódigo, pero omite inicializar la variable `i`:

golpear la primera puerta
preguntar cédula
Mientras (*cédula es distinta de x*) *AND* (nro de puerta < 7)
    golpear puerta siguiente
    preguntar cédula
Fin

Se recurre a la automatización y enseguida constata el error. Se le pregunta cuánto vale `i` y dice "*1*". Se le hace notar que no está escrito y responde "*ah, me faltó eso*". Se le pregunta qué implica eso respecto al valor de la variable y dice "*es indefinido*". La construcción de conocimiento relativa al *programa* implica conceptualizar aspectos propios de la ejecución en máquina, como la importancia de inicializar las variables. Agrega dicha inicialización, quedando la siguiente versión (correcta):

```
cedula := arre[1];
i := 1;
WHILE (cedula <> x) AND (i < N) DO
BEGIN
    cedula := arre[i+1];
    i := i+1
END;
```

Mantiene la estrategia de búsqueda usada en pseudocódigo (*a priori*), obteniendo el valor de la primera celda antes de iniciar la iteración y quedando `i` con el valor del índice de la última celda al detener la iteración tanto cuando el valor buscado está en ella como cuando no existe. Debido a la estrategia utilizada, no se produce error de *salida de rango*. Dado que ninguna de sus condiciones, combinadas con `AND` en la estructura `while`, pretende acceder a una celda del arreglo, es irrelevante si la evaluación se realiza por *circuito corto* o *circuito completo*. Mantiene la estructura de control *Mientras* (la cual traduce a `while`), el orden de las condiciones y su

combinación mediante *AND*. Al pasar de pseudocódigo a *Pascal*, invierte el orden de los pasos previos a la iteración. Coloca la instrucción `i := 1` (que representa el paso *golpear la primera puerta*) luego de `cedula := arre[1]` (que representa el paso *preguntar cédula*). Hace lo mismo con los pasos dentro de la iteración. A pesar de invertir el orden, mantiene la misma lógica de su versión en pseudocódigo, al usar `i+1` como índice en `cedula := arre[i+1]`, antes de incrementar `i`.

Tras volver a la automatización y constatar que funciona correctamente, procede a transcribirlo a la computadora. Compila (no se producen errores) y lo ejecuta varias veces, tanto para buscar valores que existen en el arreglo como valores que no, funcionando bien en todos los casos. La instrucción (escrita directamente en la computadora) que el estudiante pone luego del bloque `while` para mostrar el resultado es la siguiente sentencia `if/else`:

```
IF cedula = x THEN
    writeLn('La cedula fue encontrada')
ELSE
    writeLn('La cedula no fue encontrada');
```

El estudiante elige usar la variable `cedula` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `cedula` siempre queda con el valor de la última celda visitada en la recorrida, por lo tanto siempre puede comparar su valor con el buscado (dado en `x`) para emitir un mensaje indicando si estaba o no.

Finalmente, el pensamiento del estudiante profundiza en las transiciones hacia *newC* y *newC*'. Escribe un programa acorde a la lógica de su algoritmo en pseudocódigo. Expresa cambios en objetos, los que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`cedula <> x`) y (`i < N`), combinadas con `AND`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`cedula <> x`), avance (`i := i+1`) y repetición (`while`).

Pablo M (*C*)

Su primera versión del programa es la siguiente:

```
int a = 0; boolean igual;
int CP;
while (a <= N-1) && (cedula != CP) {
    printf ("Cual es su cedula? ");
    scanf ("%d ",&CP);
    if (CP == cedula) {
        igual = True
                    }
    else {
         a++
         igual = False
         }
      }
```

Inicialmente, el estudiante parece no haber comprendido la consigna planteada (*buscar un valor dentro del arreglo*). Conforme avanza en la escritura del programa, esto se vuelve evidente. Se lo deja continuar sin decirle nada, a efectos de ver lo que escribe. Cuando termina, su programa no busca un valor en el arreglo, sino que busca un valor dentro de una secuencia de N valores que se leen progresivamente por teclado (sin haberlos cargado en el arreglo). Su programa es bastante acorde a su última versión en pseudocódigo, pero bajo el supuesto de que los números de los documentos se van leyendo por teclado en vez de tenerlos previamente cargados en el arreglo:

Mientras (*haya puertas en la cuadra*) *AND* (*no encuentre a fulanito*)
    golpeo la puerta
    pregunto por fulanito (*cédula*)
    Si (*fulanito está en la puerta*)
        Termino la búsqueda
    Sino
        Voy a la siguiente puerta
    Fin
Fin

Cuando termina, se le hace notar lo anterior. El estudiante había interpretado que *pregunto por fulanito (cédula)* se refiere a *preguntarle al usuario del programa*. Se le explica nuevamente la consigna, haciendo énfasis en que los valores de las cédulas fueron previamente cargados en el arreglo y que *preguntar por fulanito* equivale a consultar el valor almacenado en cada celda visitada (en vez de preguntarle al usuario). Tras esto, escribe una segunda versión:

```
int a = 0; boolean igual;
while (a <= N-1) && (cedula != arre[a]) {
    if (arre[a] == cedula) {
        igual = True
                        }
    else {
          igual = False
         }
        a++
      }
```

En esta versión, sustituye la variable `CP` por `arre[a]`, donde `a` contiene el valor del índice de la celda actual del arreglo y quita la lectura de la cédula por teclado. Mantiene la estrategia de búsqueda usada en pseudocódigo (*a posteriori*), obteniendo el valor de la primera celda dentro de la iteración y quedando `a` con el valor del índice siguiente a la última celda al detener la iteración cuando el valor buscado no existe. Mantiene la estructura de control *Mientras* (la cual traduce a `while`), el orden de las condiciones y su combinación con *AND*. Unifica en una sola condición (`arre[a] == cedula`) tres pasos separados de su pseudocódigo (*golpeo la puerta*, *pregunto por fulanito (cédula)* y *Si* (*fulanito está en la puerta*)). Sin embargo, altera la ubicación de una instrucción. En su pseudocódigo avanza a la siguiente puerta solamente cuando el documento

buscado no está en la puerta actual (el paso *Voy a la siguiente puerta* se ubica dentro de la cláusula *Sino*), mientras que en su programa avanza *siempre* a la siguiente celda, sin importar si el valor buscado está o no en la celda actual (coloca `a++` por *fuera* de la instrucción `if/else`). En su primera versión (cuando leía las cédulas por teclado), `a++` estaba *dentro* de la cláusula `else`.

Presenta dos tipos de error de sintaxis: faltan varios puntos y coma y un juego de paréntesis para rodear toda la condición de la sentencia `while` (es necesario en *C*). No se le dice nada y se espera a que compile en la computadora. Si el valor buscado no existe en el arreglo, no se produce error de *salida de rango* dado que el compilador usado por el estudiante genera un programa ejecutable que evalúa la condición con `&&` de la estructura `while` por *circuito corto*. Si `a` llega a tomar el valor N, la primera condición es falsa y no se evalúa la segunda. Si fuese por *circuito completo*, se evaluaría igualmente la segunda y se produciría error de salida de rango, al pretender acceder a la celda `arre[N]`, la cual no está dentro del rango de índices válidos.

Sin considerar los errores de sintaxis, tras recurrir a la automatización constata que funciona correctamente. Se le hace notar que asigna repetidamente `false` a la variable `igual` (podría haberla inicializado en `false` antes de iniciar la iteración). Se le pregunta cómo podría evitar dicha repetición y contesta "*en vez de poner un else solo, poner un else if con un tipo de condición... no se me ocurre una manera de que no repita*". Dado que luce confundido con la pregunta y que su solución resuelve el problema, se opta por que la deje así. El estudiante muestra dificultad en comprender el significado de las instrucciones que involucran la variable booleana. Lo transcribe a la computadora (escribe `FALSE` y `TRUE` en mayúsculas cuando lo hace), lo compila y se emite el siguiente mensaje de error en la línea donde comienza la sentencia `while`:

```
error: expected identifier before '(' token
```

Se repasa la sintaxis de `while` (mostrando los paréntesis que deben rodear toda la condición) y se le pregunta cómo podría arreglarlo. Dice "*hago así*", mientras quita los paréntesis cercanos al operador `&&`, dejando solo los paréntesis exteriores. Así logra que toda la condición quede entre paréntesis, conforme a la sintaxis. Compila de nuevo y se emiten los mensajes de error correspondientes a los puntos y coma faltantes. Los agrega, quedando ahora la siguiente versión final (que resuelve el problema):

```
int a = 0; boolean igual;
while (a <= N-1 && cedula != arre [a]) {
    if (arre[a] == cedula) {
        igual = TRUE;
   }
else {
     igual = FALSE;
}
a++;
}
```

Ejecuta el programa dos veces, una para buscar un valor que existe y otro que no, constatando que funciona bien en ambas. La instrucción (escrita directamente en la computadora) que pone luego del bloque `while`, para mostrar el resultado, es la siguiente sentencia `if/else`:

```
if (a > N-1) {
    printf ("Fulanito no esta en ninguna de las puertas");
}
else {
       printf ("Fulanito esta en alguna de las puertas");
}
```

El estudiante elige hacer uso de la variable `a` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `a` queda con el valor del índice siguiente a la última celda únicamente en caso de que el valor buscado no esté en el arreglo, por lo tanto siempre puede usar su valor para determinar si el número buscado (dado en `cedula`) estaba o no.

Si bien resuelve finalmente el problema, compara dos veces el número buscado con el de la celda actual (una vez en la segunda condición de la estructura `while` y otra vez en la condición de la estructura `if`). Alcanza solo con la primera comparación. Vinculado a esto está el hecho de que la variable `igual` termina sin tener utilidad, dado que no la usa en la segunda condición de la estructura `while` para detener la búsqueda y tampoco en la sentencia `if` para decidir qué resultado emitir. Se le hace notar esto al estudiante. De hecho, al compilar, esto es reportado como advertencia:

```
warning: variable 'igual' set but not used
```

A raíz de esto, se podría eliminar la variable `igual`, quitar la instrucción `if/else` del bloque `while`, y aún así su programa resolvería el problema. Que haya ocurrido esto es consecuencia de tres factores. El primero, la confusión inicial del estudiante al escribir la primera versión que buscaba el número dentro de una secuencia de valores leídos por teclado, en vez de buscarlo en el arreglo. En dicha solución, quizás su intención fuera el uso de la variable `igual` para indicar el éxito o no en encontrar el número buscado y por eso luego la mantuvo. El segundo, el cambio de orden de la instrucción `a++` colocada por *fuera* de la instrucción `if/else` (cuando el paso *Voy a la siguiente puerta* se ubicaba *dentro* de la cláusula *Sino* en su pseudocódigo y `a++` dentro de la cláusula `else` en su primera versión). Tercero, el manejo inadecuado de la variable `igual`, al pretender inicializarla repetidamente en `FALSE` y terminar sin darle ningún uso real.

Por lo anterior, el pensamiento del estudiante avanza en las transiciones hacia *newC* y *newC'*, pero no lo hace tan profundamente. Logra construir un programa que resuelve la búsqueda lineal, pero se aparta de la lógica del algoritmo que ideó antes al escribir en pseudocódigo. Su comprensión del vínculo entre el conocimiento *conceptual* sobre el algoritmo y el conocimiento *formal* que construye sobre el programa es menos sólida que para los demás estudiantes al mantener, dentro del bloque `while`, la instrucción

`if/else` que no aporta nada a la solución del problema y no tener claro cómo evitar asignar repetidamente `FALSE` a la variable `igual`. Si bien ocurre en menor grado que para otros estudiantes, Pablo M igualmente construye conocimiento, pues expresa cambios en objetos, que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`a <= N-1`) y (`cedula != arre[a]`), combinadas con `&&`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`cedula != arre[a]`), avance (`a++`) y repetición (`while`).

Pablo P (*C*)

Escribe una única versión del programa, la cual es correcta:

```
int CI;
int puerta = 0;
printf ("ingrese el nro cedula que busca");
scanf ("%d", &CI);
while (puerta <= (N-1) && CI != arre[puerta])
{
    if (arre[puerta] != CI)
    {
        puerta = puerta + 1;
    }
}
```

Esta versión se corresponde en forma acorde con su última versión en pseudocódigo, excepto por algunas pequeñas diferencias que se enumeran en breve:

numero = 0
puerta = 0
CI = numero conocido
Mientras (*puerta ≤ 4*) *AND* (*numero* != *CI*)
    pregunto numero en puerta
    Si (*numero* != *CI*)
        puerta = puerta + 1
    Fin
Fin

Mantiene la estrategia de búsqueda usada en pseudocódigo (*a posteriori*), obteniendo el valor de la primera celda dentro de la iteración y quedando puerta con el valor del índice siguiente a la última celda al detener la iteración cuando el valor buscado no existe. Mantiene la estructura de control *Mientras* (la cual traduce a `while`), el orden de las condiciones y su combinación con *AND*. Difiere de su pseudocódigo en que deja de usar la variable *numero* y en su lugar cambia por `arre[puerta]`. Esto genera que su programa funcione bien cualquiera sea el número buscado, ya que su versión en pseudocódigo funciona bien solo cuando el número buscado no es cero. Excepto por este cambio (que en realidad es una mejora, en el sentido de que es una solución un poco más general), mantiene la misma lógica de su versión en pseudocódigo. Unifica en una condición (`arre[puerta] != CI`) dos pasos separados del pseudocódigo

(*pregunto numero en puerta* y *Si* (*numero != CI*)). Además, usa (`N-1`) en lugar de 4 para referirse al índice de la última celda, funcionando cualquiera sea el tamaño del arreglo. Si bien resuelve correctamente el problema, compara dos veces el número buscado con el de la celda actual (una vez en la segunda condición de `while` y otra vez en la condición de `if`). Alcanza solo con la primera. No se le menciona esto, pues igualmente su programa es acorde a la lógica de su versión en pseudocódigo.

En caso de que el valor buscado no exista en el arreglo, no se produce error de *salida de rango* debido a que el compilador usado por el estudiante genera un programa ejecutable que evalúa la condición con `&&` de la estructura `while` por *circuito corto*. En caso de que `puerta` llegue a valer `N`, la primera condición es falsa y no se evalúa la segunda. Si fuese por *circuito completo*, se evaluaría igualmente la segunda condición y se produciría error de salida de rango, al pretender acceder a la celda `arre[N]`, la cual no está dentro del rango de índices válidos. Tras recurrir a la automatización, constata que funciona correctamente y procede a transcribirlo a la computadora. Lo compila y no se producen errores. Luego del bloque `while`, agrega la siguiente sentencia `if/else` (directamente en la computadora) para mostrar el resultado:

```
if (CI != arre[puerta])
{
    printf("No se encontro la Cedula buscada");
}
else
{
    printf("Se encontro la Cedula buscada");
}
```

El estudiante elige hacer uso de la celda `arre[puerta]` para decidir qué mensaje emitir. Si bien la búsqueda funciona correctamente, se produce un error de *salida de rango* en la condición de `if`. En caso de que el número buscado no se encuentre en el arreglo, la variable `puerta` finaliza con el valor `N` tras ejecutarse el bloque `while`. Al pretender acceder a `arre[N]`, se produce el error. La construcción de conocimiento relativa al *programa* implica conceptualizar aspectos propios de la ejecución en máquina, como ser errores de este tipo y la búsqueda de estrategias para evitarlos. Se recurre a la automatización y el estudiante constata el error. Se le pide que piense una alternativa para corregirlo y decide cambiar la condición de la sentencia `if`, de modo que emite el mismo resultado, pero evitando acceder a una celda fuera de rango.

```
if ((N-1) < puerta)
{
    printf("No se encontro la Cedula buscada");
}
else
{
    printf("Se encontro la Cedula buscada");
}
```

El estudiante ahora elige hacer uso de la variable `puerta` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `puerta` queda con

el valor del índice siguiente a la última celda únicamente en caso de que el valor buscado no esté en el arreglo, por lo tanto siempre puede usar su valor para emitir un mensaje indicando si el número buscado (dado en `CI`) estaba o no. Compila de nuevo (no se producen errores) y lo ejecuta dos veces, una para buscar un valor que está en el arreglo y otra para buscar uno que no, funcionando bien en ambos casos.

Finalmente, el pensamiento del estudiante profundiza en las transiciones hacia *newC* y *newC'*. Su programa es acorde a la lógica de su algoritmo en pseudocódigo, excepto por los pequeños cambios enumerados anteriormente, los cuales en realidad brindan un carácter un poco más general a su solución. Expresa cambios en los objetos, que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`puerta <= (N-1)`) y (`CI != arre[puerta]`), combinadas con `&&`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`CI != arre[puerta]`), avance (`puerta = puerta + 1`) y repetición (`while`).

Tomás (*C*)

Su primera versión del programa es la siguiente:

```
boolean esta = FALSE;
int cedula, i = 0;
while (i <= N && (!esta)) {
    if (arre[i] == cedula)
        esta = TRUE;
    else
        i = i+1;
}
```

Esta versión intenta corresponderse en forma acorde con su última versión en pseudocódigo, pero tiene un error de *salida de rango*. Si el valor buscado no está en el arreglo, tras consultar la última celda, la instrucción `i = i+1` genera que la variable `i` tome el valor `N`, quedando fuera del rango de índices válidos. Al intentar acceder a `arre[i]` en la condición de `if`, se produce el error. La construcción de conocimiento relativa al *programa* implica conceptualizar aspectos propios de la ejecución en máquina, como ser errores de este tipo y la búsqueda de estrategias para evitarlos.

Mientras (*hayan puertas por ver AND no está el número*)
    preguntar si es el número
    Si (*está el número*)
        elijo esa puerta
    Sino
        sigo
    Fin
Fin

Nuevamente, la automatización cumple un rol esencial para su detección y ayudar al estudiante en la conceptualización. Tras recurrir a ella, constata el error cuando `i` toma el valor N. Consultado al respecto, dice "*al ser 3, no tengo ninguna puerta*" (en la

instancia concreta, N vale 3 y los índices van de 0 a 2). Corrige el error, cambiando <= por < en la primera condición de la estructura `while`, quedando la siguiente versión (correcta) y elimina el error de salida de rango:

```
boolean esta = FALSE;
int cedula, i = 0;
while (i < N && (!esta)) {
    if (arre[i] == cedula)
        esta = TRUE;
    else
        i = i+1;
}
```

Dado que ninguna de sus condiciones, combinadas con `&&` en la estructura `while`, pretende acceder a una celda del arreglo, es irrelevante si la evaluación se realiza por *circuito corto* o por *circuito completo*. Mantiene la estrategia de búsqueda usada en pseudocódigo (*a posteriori*), obteniendo el valor de la primera celda dentro de la iteración y quedando `i` con el valor del índice siguiente a la última al parar la iteración cuando el valor buscado no existe. Mantiene la estructura de control *Mientras* (la cual traduce a `while`), el orden de las condiciones y su combinación mediante *AND*, así como el orden de las instrucciones. Unifica en una condición (`arre[i] == cedula`) dos pasos separados (*preguntar si es el número* y *Si* (*está el número*)). Respecto a la variable `cedula`, durante la automatización se le dice al estudiante que asuma que la misma fue inicializada con un valor leído por teclado (lo cual será así al transcribir el programa a la computadora). Procede a transcribirlo a la computadora y compilar, tras lo cual se emite el siguiente mensaje de error:

```
error: expected ',' or ';' before 'printf'
```

Previo al código anterior, el estudiante declara las variables necesarias y usa `printf` para emitir al usuario un mensaje solicitando que ingrese la cédula a buscar. Entre la declaración de las variables y la llamada a `printf` omite poner punto y coma. Lo corrige, y además se da cuenta de un paréntesis que le faltó transcribir, tras lo cual lo agrega y compila nuevamente (no se produce ningún error). Lo ejecuta dos veces, una para buscar un valor que está en el arreglo y otra para buscar uno que no, funcionando bien en ambos casos. La instrucción que escribe luego del bloque `while` (directamente en la computadora) para mostrar el resultado es la siguiente sentencia `if/else`:

```
if (esta == TRUE)
    printf ("Esta\n");
else
    printf ("No esta\n");
```

El estudiante elige hacer uso de la variable `esta` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `esta` queda con valor `TRUE` cuando el valor buscado (dado en `cedula`) está en el arreglo y `FALSE` cuando no, por lo que siempre puede usar su valor para emitir un mensaje indicando si el número estaba o no. Si bien la condición es correcta, la comparación con `TRUE` es innecesaria (basta poner `esta` como condición). Esta característica ha sido vista repetidamente en clase. Muchos estudiantes presentan dificultad en conceptualizar que una variable booleana es por sí misma una expresión booleana, sin ser necesaria su comparación con `TRUE`. Ya usaban operadores de comparación en matemática, previo a conocer el tipo `boolean` y tienden a repetir dicha práctica al manipular variables booleanas, queriendo *generalizar* al lenguaje de *programación* su conocimiento de lenguaje *matemático* (la herramienta cognitiva que interviene es la generalización *inductiva*).

Finalmente, el pensamiento del estudiante profundiza en las transiciones hacia *newC* y *newC*'. Escribe un programa acorde a la lógica de su algoritmo en pseudocódigo. Expresa cambios en objetos, los que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`i < N`) y (`!esta`), combinadas con `&&` deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`arre[i] == cedula`), avance (`i = i+1`) y repetición (`while`).

Valeria (*C*)

Su primera versión del programa es la siguiente:

```
int CI, i = 0;
boolean encontreCI = false;
while (arre[i] != CI && i < N) {
    If (arre[i] == CI) {
        encontreCI = true;
                        }
    Else {
          i++;
        }
}
```

Esta versión intenta corresponderse en forma acorde con su última versión en pseudocódigo, pero tiene un error de *salida de rango*. Si el valor buscado no está en el arreglo, tras consultar la última celda, la instrucción `i++` genera que la variable `i` tome el valor N, quedando por fuera del rango de índices válidos. Al pretender acceder luego a `arre[i]` en la primera condición de la estructura `while`, se produce el error. La construcción de conocimiento relativa al *programa* implica conceptualizar aspectos propios de la ejecución en máquina, como ser errores de este tipo y la búsqueda de estrategias para evitarlos. Por otra parte, pone `If` y `Else` con la primera letra de cada una en mayúscula (más tarde las cambia a minúsculas al pasarlo a computadora). Respecto a la variable `CI`, se le dice a la estudiante que asuma que la misma fue inicializada con un valor leído por teclado (lo cual será así al transcribir el programa a la computadora).

Mientras (*no encuentre nro C.I AND no pregunté en última puerta*) *hacer*
preguntar nro C.I
Si (*encuentro nro C.I*)
parar de preguntar
Sino
seguir preguntando
Fin
Fin

Mantiene la estrategia de búsqueda usada en pseudocódigo (*a posteriori*), obteniendo el valor de la primera celda dentro de la iteración y quedando `i` con el valor del índice siguiente a la última al parar la iteración cuando el valor buscado no existe (lo cual genera su error de *salida de rango*). Mantiene la estructura de control *Mientras* (la cual traduce a `while`), el orden de las condiciones y su combinación mediante *AND*, así como el orden de las instrucciones. Unifica en una sola condición (`arre[i] == CI`) dos pasos separados de su pseudocódigo (*preguntar nro C.I* y *Si* (*encuentro nro C.I*)).

En términos del *formalismo intermedio* (pseudocódigo), el algoritmo es correcto, dado que *AND* es conmutativo, siendo irrelevante el orden de las condiciones. La estudiante debe conceptualizar que ahora la ejecución es en máquina y construir conocimiento sobre cómo la misma evalúa expresiones booleanas. En el *lenguaje formal* utilizado por la estudiante (*C*), el compilador usado genera un programa ejecutable que evalúa la condición con `&&` por *circuito corto*. Primero evalúa la expresión de la izquierda y, según el resultado, evalúa la de la derecha solo si es necesario. El orden de las expresiones ahora es relevante, por lo cual `&&` deja de ser conmutativo. La expresión (`arre[i] != CI`) se ubica a la izquierda de `&&`, por lo cual se evalúa antes que la otra (`i < N`), produciendo el error cuando `i` toma el valor N.

Nuevamente, la automatización cumple un rol esencial para la detección del error y ayudar a la estudiante en la conceptualización. Tras recurrir a ella, constata el error. Respecto del mismo, pregunta "*si la primera es falsa ¿no se hace la segunda?*" (se refiere a cuando en clase, previo al estudio, se habló de la posibilidad de evaluar por *circuito corto*). Se le dice que sí, a lo que responde "*entonces acá me conviene dar vuelta*" (señala las condiciones de la estructura `while` y se refiere a invertir el orden). Lo hace, escribiendo una segunda versión:

```
int CI, i = 0;
boolean encontreCI = false;
while (i < N && arre[i] != CI) {
    If (arre[i] == CI) {
        encontreCI = true;
                    }
    Else {
          i++;
        }
}
```

Tras invertir el orden, (i < N) se evalúa primero, dando `FALSE` como resultado y dejando sin evaluar la otra y se elimina el error de salida de rango. Si la evaluación fuese por *circuito completo*, se evaluaría igualmente la segunda condición y se produciría nuevamente el error. De ser ese el caso, la estudiante necesitaría buscar tra estrategia para evitar el error. Tras volver a la automatización y constatar que funciona correctamente, lo transcribe a la computadora. Al hacer la transcripción, pone `FALSE` y `TRUE` en mayúsculas e `if` y `else` en minúsculas. Luego escribe (directamente en la computadora) la siguiente sentencia `if/else` para mostrar el resultado:

```
if (encontreci == TRUE) {
     printf("Se encontro ci"); }
else {
     printf("No se encontro cI");
     }
```

La estudiante elige hacer uso de la variable `encontreci` para decidir qué mensaje emitir. Si bien la condición es correcta, la comparación con `TRUE` es innecesaria (basta poner `encontreci` como condición). Esta característica ha sido vista repetidamente en clase. Muchos estudiantes presentan dificultad en conceptualizar que una variable booleana es por sí misma una expresión booleana, sin ser necesaria su comparación con `TRUE`. Ya usaban operadores de comparación en matemática, previo a conocer el tipo `boolean` y tienden a repetir dicha práctica al manipular variables booleanas, queriendo *generalizar* al lenguaje de *programación* su conocimiento de lenguaje *matemático* (la herramienta cognitiva que interviene es la generalización *inductiva*).

Si bien la búsqueda funciona bien, en el sentido de que para cuando corresponde y ya no tiene el error de salida de rango, se produce un *problema de borde*, debido a la condición que define en esta nueva sentencia `if` para decidir qué resultado mostrar. Si el número buscado está en la *última* celda, el mensaje es incorrecto. En ese caso, la iteración se detiene al evaluar la segunda condición (`arre[i] != CI`) de la estructura `while`. En tal caso, no llega a ejecutarse la asignación `encontreCI = true`, por lo cual se muestra el mensaje `No se encontro cI`. La estudiante constata esto tras compilar el programa (no se producen errores) y ejecutarlo para buscar un valor que está en la última celda. Previamente constata que funciona bien tanto para buscar un valor que está en una celda anterior a la última como para buscar otro que no existe en el arreglo. Se recurre nuevamente a la automatización y decide cambiar la segunda condición de la estructura `while`, quedando:

```
int CI, i = 0;
boolean encontreCI = FALSE;
while (i < N && encontreCI == FALSE) {
    if (arre[i] == CI) {
           encontreCI = TRUE; }
    else {
           i++; }
}
```

Con este cambio, logra que `encontreci` siempre tome el valor `TRUE` sin importar en qué celda se encuentre el valor buscado. Su versión anterior asignaba `TRUE` a dicha variable siempre y cuando el valor se encontrase en cualquier celda anterior a la última. Finalmente, construye una versión en la cual puede hacer uso del valor en dicha variable para emitir un mensaje indicando si el valor estaba o no, sin importar la celda en la cual se encuentre. Tras el cambio, compila nuevamente (no se producen errores) y lo ejecuta para varios valores, constatando que ahora muestra el mensaje correcto (incluso cuando el valor buscado está en la última celda).

Finalmente, el pensamiento de la estudiante profundiza en las transiciones hacia *newC* y *newC'*. Si bien su programa no es totalmente acorde a la lógica de su algoritmo en pseudocódigo, debido al cambio de orden en las condiciones de la estructura `while`, introdujo dicho cambio de manera *consciente*, para corregir un error producido por un aspecto propio de la ejecución en máquina, sobre el cual la estudiante reflexionó y construyó conocimiento. Tiene claro por qué se necesitó el cambio. Por lo demás, mantiene la lógica del algoritmo ideado anteriormente. Al igual que en pseudocódigo, expresa cambios en objetos, los que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`i < N`) y (`encontreci == FALSE`), combinadas con `&&`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`arre[i] == CI`), avance (`i++`) y repetición (`while`).

Ximena (*C*)

Su primera versión del programa es la siguiente:

```
int ced;
boolean c = FALSE
int i = 0
while (i =< n-1) && (c == TRUE)
{
    if (arre[i] == ced) {
        c = TRUE;
         }
    else
          i++;
}
```

Esta versión intenta corresponderse en forma acorde con su última versión en pseudocódigo, pero tiene cinco errores. Cuatro de ellos son de sintaxis y uno es de comportamiento en la ejecución. Los de sintaxis son que omite poner punto y coma tras las dos primeras inicializaciones, usa =< en vez de <=, usa `n` minúscula en vez de `N` mayúscula para referirse a la constante (el lenguaje *C* es *case sensitive*) y omite poner paréntesis rodeando ambas condiciones en la estructura `while` (*C* lo exige). El de comportamiento es por comparar (`c == TRUE`) en la segunda condición. La estudiante quizás piensa en la condición para *detener* la iteración en vez de para *seguir* iterando, lo cual muestra que quizás aún persiste alguna dificultad en conceptualizar la semántica de `while`.

Respecto a la variable `ced`, se le dice a la estudiante que asuma que la misma fue inicializada con un valor leído por teclado (lo cual será así al transcribir el programa a la computadora).

Mientras hallan puertas AND no encuentre la cédula que busco hacer
    abrir puerta
    Si cédula es la que busco
        terminé de buscar
    Sino
        sigo recorriendo puertas
    Fin
Fin

Mantiene la estrategia de búsqueda usada en pseudocódigo (*a posteriori*), obteniendo el valor de la primera celda dentro de la iteración y quedando i con el valor del índice siguiente a la última celda al detener la iteración cuando el valor buscado no existe. No se produce error de *salida de rango* porque la segunda condición de la estructura while corta la iteración ni bien el valor de `i` supera el último índice válido, sin llegar a intentar acceder a una celda con índice inválido. Dado que ninguna de sus condiciones, combinadas con `&&` en la estructura `while`, pretende acceder a una celda del arreglo, es irrelevante si la evaluación se realiza por *circuito corto* o *circuito completo*. Mantiene la estructura de control *Mientras* (la traduce a `while`), el orden de las condiciones y su combinación con *AND*, así como el orden de las instrucciones. Unifica en una sola condición (`arre[i] == ced`) dos pasos separados de su pseudocódigo (*abrir puerta* y *Si cédula es la que busco*).

Por el momento, se ignoran los errores de sintaxis y se recurre a la automatización para detectar el error de comportamiento. Ni bien se evalúa por primera vez la segunda condición, constata que no itera ni una vez. Consultada al respecto dice "*lo cambiamos a FALSE*". Lo cambia, resultando en la siguiente versión, que es correcta (a menos de los errores de sintaxis):

```
int ced;
boolean c = FALSE
int i = 0
while (i =< n-1) && (c == FALSE)
{
    if (arre[i] == ced) {
        c = TRUE;
         }
    else
          i++;
}
```

Tras recurrir nuevamente a la automatización, constata que funciona correctamente. Procede a transcribirlo a la computadora y compilarlo, tras lo cual se muestran (uno por vez) los siguientes mensajes (que corresponden a los cuatro errores de sintaxis cometidos):

```
error: expected ',' or ';' before 'int'
error: expected primary-expression before '<' token
error: 'n' was not declared in this scope
error: expected identifier before '(' token
```

Se va leyendo cada error junto a la estudiante, se le consulta al respecto y los corrige uno a uno, resultando en la siguiente versión final (que compila correctamente):

```
int ced;
boolean c = FALSE;
int i = 0;
while ((i <= N-1) && (c == FALSE))
{
    if (arre[i] == ced) {
        c = TRUE;
    }
    else
          i++;
}
```

La instrucción que escribe (directamente en la computadora) para mostrar el resultado luego del bloque `while`, es la siguiente sentencia `if/else`:

```
if (c = TRUE)
    printf("Encontre la cedula \n");
else
    printf("No encontre la cedula \n");
```

Tras compilar nuevamente, se emite la siguiente advertencia en la condición de la nueva sentencia `if/else`:

```
warning: suggest parentheses around assignment used as truth
```

El lenguaje *C* permite realizar dicha asignación dentro de la condición, pero la estudiante no lo sabe. Se le consulta cuál fue su intención en relación a dicha condición. Responde "*si el valor booleano fue verdadero... me faltó el otro igual*" (se refiere a comparar con `==`, en vez de asignar). Este es un error común al escribir en *C*, dado que en matemática se usa = para expresar comparación. La estudiante *generaliza* al lenguaje de *programación* su conocimiento del lenguaje *matemático* (la herramienta cognitiva que interviene es la generalización *inductiva*). Lo corrige, quedando así:

```
if (c == TRUE)
    printf("Encontre la cedula \n");
else
    printf("No encontre la cedula \n");
```

La estudiante elige hacer uso de la variable `c` para decidir qué mensaje emitir. Debido a cómo estructura su búsqueda en el bloque `while`, `c` queda con valor `TRUE` cuando el valor buscado (dado en `ced`) está en el arreglo y `FALSE` cuando no, por lo tanto siempre puede usar su valor para emitir un mensaje indicando si el número estaba o

no. Compila una vez más (no se emite la advertencia) y lo ejecuta dos veces, una para buscar un valor que existe y otra para buscar otro que no, constatando que funciona bien en ambos casos. Si bien la condición es correcta, la comparación con `TRUE` es innecesaria (basta poner `c` como condición). Esta característica ha sido vista en clase repetidamente. Muchos estudiantes presentan dificultad en conceptualizar que una variable booleana es por sí misma una expresión booleana, sin ser necesaria su comparación con `TRUE`. Ya usaban operadores de comparación en matemática, previo a conocer el tipo `boolean` y tienden a repetir dicha práctica al manipular variables booleanas (nuevamente, generalización *inductiva*).

Finalmente, el pensamiento de la estudiante profundiza en las transiciones hacia *newC* y *newC*'. Escribe un programa acorde a la lógica de su algoritmo en pseudocódigo. Expresa cambios en objetos, los que ahora se traducen a cambios sobre el *arreglo*. Para de iterar cuando alguna de las condiciones (`i <= N-1`) y (`c == FALSE`), combinadas con `&&`, deja de cumplirse. Las acciones se expresan en el texto del programa: comparación (`arre[i] == ced`), avance (`i++`) y repetición (`while`).

Anexo E

Programa para *nuevo* problema (segunda parte: *inter* → *trans*)

En este anexo se presentan las versiones del programa que resuelve el *nuevo* problema: *listar todos los valores pares almacenados en el arreglo*, escritas por los trece estudiantes durante la cuarta actividad de la segunda parte del estudio (pasaje de la etapa *inter* a la etapa *trans*). Las versiones fueron escritas directamente en la computadora, sin pasar por escritura previa en papel ni pseudocódigo. Se presentan *todas* las versiones escritas por cada estudiante junto con un análisis detallado de cada una y del proceso realizado. En cada análisis, se usan cursivas entre comillas para citar respuestas orales de los estudiantes durante las entrevistas. Las entrevistas fueron grabadas y se guardaron los archivos fuente escritos por los estudiantes.

Aaron (*Pascal*)

Escribe una única versión del programa, la cual es correcta:

```
for i:=1 to N do
    if (arre[i] mod 2 = 0) then
        writeln(arre[i]);
```

Consultado acerca de la elección de estructuras de control para este nuevo problema, responde "*usé un for y un if… porque me pide que muestre todas las cédulas y eso quiere decir que tengo que verificar con cada uno de los casos… con el if* " (se refiere a verificar que cada celda contenga un valor par antes de mostrarlo) Tras compilar (no se producen errores) procede a ejecutarlo y constata que funciona bien. El estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Ignacio D (*Pascal*)

Su primera versión del programa es la siguiente:

```
for i = 1 to N do
    if (arre[i] mod 2 = 0) then
        writeln ( arre[i] );
```

Al compilar, se produce un error que dice:

```
Syntax error, ":=" expected but "=" found
```

Consultado sobre el error, dice "*el i era una asignación y no una condición*" y cambia = por := quedando así (compila correctamente):

```
for i := 1 to N do
    if (arre[i] mod 2 = 0) then
        writeln ( arre[i] );
```

Tras ejecutarla, constata que funciona bien. Consultado acerca de la elección de estructuras de control para este nuevo problema, responde "*lo que iba a hacer era recorrer todo el arreglo*" (refiriéndose a `for`) "*y después, en cada una de las celdas, ver si el número era par*" (y a `if`). El estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Ignacio U (*Pascal*)

Escribe una única versión del programa, la cual es correcta:

```
for i := 1 to N do
    if (arre[i] mod 2 = 0) then
        WriteLN(arre[i]);
```

Consultado acerca de la elección de estructuras de control para este nuevo problema, responde "*el for para recorrer todas las celdas del arreglo... y luego el if para preguntar eso mismo*" (se refiere a preguntar si el valor en cada celda es par) Tras compilar (no producen errores) procede a ejecutarlo y constata que funciona bien. El estudiante logra construir un programa que resuelve el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Joaquín (*Pascal*)

Su primera versión del programa es la siguiente:

```
for i := 1 to n do
    if (arre[i] mod 2 = 0) then
        writeln('arre[i]');
```

Al compilar, se produce un error debido a un carácter mal tipeado. Tras ser corregido, procede a ejecutarlo y constata que muestra varias veces el texto "arre[i]" (debido a las comillas que puso, mostró texto en vez de números). Procede a eliminarlas, quedando la siguiente versión, la cual compila y funciona bien (mostrando todos los números pares almacenados):

```
for i := 1 to n do
    if (arre[i] mod 2 = 0) then
        writeln(arre[i]);
```

Consultado acerca de la elección de estructuras de control para este nuevo problema, responde "*porque sé la cantidad de datos que tengo... en el arreglo... porque quiero encontrar todos los pares*" (refiriéndose a `for`). Dado que habla de *encontrar*, se le pregunta si quiere hacer una búsqueda y contesta "*no, no, o sea... todos los que sean pares... escribirlos*" (se refiere a mostrarlos). Sobre `if`, dice "*para verificar eso... si es par*". El estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Juan (*C*)

Escribe una única versión del programa, la cual es correcta:

```
for (i=0; i <= N-1; i++) {
    if (arre[i] % 2 == 0) {
        printf ("%d\n",arre[i]);
    }
}
```

Consultado acerca de la elección de estructuras de control para este nuevo problema, responde "*elegí el for para poder recorrer todo el arreglo... y el if para ir evaluando si el resto de dividir el valor de cada celda dividido 2 era cero... y en ese caso muestro el valor*". Tras compilar (no se producen errores) procede a ejecutarlo y constata que funciona bien. El estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Martín (*Pascal*)

Su primera versión del programa es la siguiente:

```
for i:= 1 to n do
begin
    if arre[i] mod 2 = 0 then
     writeln(arre[i]);
```

En esta versión, olvida colocar `end` para cerrar el bloque correspondiente a la sentencia `for`. Al compilar, se produce un error, en la última línea del programa:

```
Syntax error, ";" expected but "." found
```

El error ocurre porque el compilador intenta asociar la palabra `end` (que está luego de la sentencia for) a la palabra `begin` puesta por el estudiante. El problema es que dicho `end` tiene un punto a continuación (fue puesto para cerrar el bloque del programa principal) y, por tanto, no puede ser asociado a la palabra `begin` de la sentencia `for`. En realidad falta la palabra `end` que cierre el bloque `for`. Se consulta al respecto al estudiante y dice: "*ah, me faltó un end*". Se le pregunta qué fue lo que interpretó el compilador y contesta: "*suponer que este end pertenecía al for*" (el del programa principal). Agrega la palabra `end` faltante, quedando así (compila correctamente):

```
for i:= 1 to n do
begin
    if arre[i] mod 2 = 0 then
     writeln(arre[i]);
end;
```

Consultado acerca de la elección de estructuras de control para este nuevo problema, responde "*for porque tiene que revisarse el valor de cada una de las casillas... y el if lo usé para saber si es divisible entre 2... y en cuyo caso saber si es par o no*". Tras compilar (ahora sin errores) procede a ejecutarlo y constata que funciona bien. El estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Mónica (*C*)

Su primera versión del programa es la siguiente:

```
for (i=0; i <= N-1; i++)
{
    if (arre[i] % 2 = 0)
        printf ("%d ",arre[i]);
}
```

Esta versión tiene un error de sintaxis. Usa asignación (=) en vez de comparación (==) en la condición de `if`, lo que genera que el compilador intente interpretar la expresión a la izquierda de la asignación como si fuera una variable. Este es un error común al escribir en *C*, dado que en matemática se usa el símbolo = para comparación. La estudiante *generaliza* al lenguaje de *programación* su conocimiento del lenguaje *matemático* (la herramienta cognitiva que interviene es la generalización *inductiva*). Al compilar, se emite el siguiente mensaje de error.

```
error: lvalue required as left operand of assignment
```

Se le señala la línea donde se produce el error y se le consulta al respecto. Dice "*ah, me faltó...*" (mientras cambia = por ==) quedando la siguiente versión (correcta):

```
for (i=0; i <= N-1; i++)
{
    if (arre[i] % 2 == 0)
        printf ("%d ",arre[i]);
}
```

Tras compilar de nuevo (ahora sin errores) lo ejecuta y constata que funciona bien. Consultada acerca de la elección de estructuras de control para este nuevo problema, responde "*para recorrer todas las puertas*" (`for`) y "*para fijarme si son pares o no*" (`if`). La estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Nicolás (*Pascal*)

Escribe una única versión del programa, la cual es correcta:

```
FOR i := 1 TO N DO
    IF (arre[i] MOD 2 = 0) THEN
        writeLn(arre[i])
```

Consultado acerca de la elección de estructuras de control para este nuevo problema, responde "*for para que recorra todo el array... desde el primero hasta el último... y se fija con el if a ver si el resto de la división entera es cero*". Durante la escritura, considera por un instante agregar una cláusula `else` pero al final no lo hace. Al respecto, dice "*porque no quiero que haga nada si es impar*". Tras compilar (no se producen errores) procede a ejecutarlo y constata que funciona bien. El estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Pablo M (*C*)

Su primera versión del programa es la siguiente:

```
for (a=0; a <= N-1; a++) {
     printf("\n%d", arre[a]);
}
```

Esta versión compila correctamente, pero muestra todos los valores del arreglo. Lo constata al ejecutarlo en máquina. Se le pregunta cómo podría mostrar solamente los que son pares, y contesta "*acá, cambiar esto*", tras lo cual lo modifica, resultando en una segunda versión:

```
for (a=0; arre[a]%2 == 0; a++) {
     printf("\n%d", arre[a]);
}
```

Su segunda versión tampoco resuelve el problema. En vez de mostrar todos los valores pares, muestra solamente los pares que haya en el arreglo desde la celda cero hasta la primera celda con un valor impar (sin incluirla). Además, podría producir error de *salida de rango* si no existe ningún impar. El estudiante no es consciente de que su código se comporta como si fuese una estructura `while` en lugar de `for`. Si bien *C* permite usar `for` de esta manera, en clase (previo al estudio) se desestimó su uso en casos así. Su solución se asemeja más a una *búsqueda lineal* que carece de una segunda condición para controlar la detención cuando se llega al final del rango de índices válidos del arreglo. Durante el curso, se hizo hincapié en el uso de la estructura de control más adecuada para cada problema. Si se quisiera buscar el primer valor impar, sería más adecuado usar `while` (como en la tercera actividad).

El estudiante tiene dificultad en conceptualizar la semántica de `for`. Quizás piensa que la condición (`arre[a]%2 == 0`) es para *filtrar* los valores pares en la recorrida, sin ser consciente de que en realidad *detiene* la misma cuando deja de cumplirse. Se recurre a la automatización y constata el problema cuando llega a una celda con un valor impar. Consultado al respecto, dice "*mostró las que son pares hasta que se cortó*". Se repasa junto a él la sintaxis en *C* de cada estructura (tanto de selección como de iteración) y cuando resulta adecuado usar cada una. Se le pregunta qué tendría que verificar de cada valor a medida que recorre y contesta "*si son pares*". Se le pregunta cómo podría hacer para lograrlo y dice "*algo por fuera mismo del for... dejas esas tres condiciones bien y poner algo par, o sea un if*". Escribe una tercera versión:

```
for (a=0; a <= N-1; a++) {
    if(arre[a]%2 == 0) {
     printf("\n%d", arre[a]);
}
```

Esta versión ahora resuelve el problema pero tiene un error de sintaxis (dos llaves abren bloques, pero solo una cierra). Al compilar, se emite el siguiente mensaje:

```
error: expected '}' at end of input
```

Se le señala la línea donde se produce el error y se le consulta al respecto. Lo corrige agregando una segunda llave que cierra quedando la siguiente versión (correcta). Compila sin errores, la ejecuta en máquina y constata que ahora funciona bien.

```
for (a=0; a <= N-1; a++) {
    if(arre[a]%2 == 0) {
     printf("\n%d", arre[a]);
  }
}
```

Consultado acerca de la elección de estructuras de control para este nuevo problema, acerca de `for` responde "*para que te agarre solo el índice... para tomar el rango de todas las celdas que tenés vos... cuando tenés todos... cuando sabés los elementos* " (se refiere a recorrer todas las celdas) Respecto a if dice: "*para verificar múltiplo de 2... si era par...que se mostrase en pantalla*". El estudiante logra finalmente resolver el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente, si bien tuvo dificultades durante el proceso de construcción.

Pablo P (*C*)

Su primera versión del programa es la siguiente:

```
for (i=0; i < (N-1); i++)
{
    if (arre[i]%2 == 0)
    {
        printf("\n%d", arre[i]);
    }
}
```

Esta versión presenta un *problema de borde*. Deja sin consultar el valor almacenado en la última celda. El hecho de que los índices en *C* empiezan en cero, lleva en ocasiones a los estudiantes a expresar incorrectamente el límite superior de la iteración al recorrer las N celdas. Se recurre a la automatización y detecta el problema. Dice "*sino iba a terminar en 2*" (refiriéndose a la penúltima celda, el valor de la constante N es 3 en la versión usada para la automatización), tras lo cual lo corrige, quedando así:

```
for (i=0; i <= (N-1); i++)
{
    if (arre[i]%2 == 0)
    {
        printf("\n%d", arre[i]);
    }
}
```

Consultado acerca de la elección de estructuras de control para este nuevo problema, responde "*porque tenía que mostrar todos... sí o sí tenía que pasar por todos*" (`for`) y "*había que preguntar si era par o no*" (`if`). Tras compilar (no se producen errores) procede a ejecutarlo y constata que funciona bien. El estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Tomás (*C*)

Escribe una única versión del programa, la cual es correcta:

```
for (i=0; i < N; i++) {
    if (arre [i] % 2 == 0) {
        printf ("%d\n",arre [i]);
    }
}
```

Consultado acerca de la elección de estructuras de control para este nuevo problema, responde "*el for... tendría que ser para todas*" (se refiere a que necesita recorrer todas las celdas) y "*si es par, que me imprima... sino que no haga nada*" (se refiere a usar `if` sin `else` para chequear si el valor en cada celda es par). Tras compilar (no se producen errores) procede a ejecutarlo y constata que funciona bien. El estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Valeria (*C*)

Escribe una única versión del programa, la cual es correcta:

```
for (i=0; i < N; i++) {
      if (arre[i] % 2 == 0) {
           printf("\n%d",arre[i]); }
}
```

Consultada acerca de la elección de estructuras de control para este nuevo problema, responde "*for porque acá sí tengo que ir hasta el final*" (a diferencia del problema anterior) y "*if para ver si lo tengo que mostrar*" (se refiere a verificar si el contenido de cada celda es par, para mostrarlo). Tras compilar (no se producen errores) procede a ejecutarlo y constata que funciona bien. La estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Ximena (*C*)

Su primera versión del programa es la siguiente:

```
for (i=0; i <= N-1; i++)
{
    if ( arre[i] % 2 == 0)
     printf ("%d", arre [i]);
}
```

Tras compilar (no se producen errores), lo ejecuta. Efectivamente, muestra todos los valores pares, pero la estudiante cree que no. Esto se debe a que no pone ninguna separación entre cada valor y el siguiente en la llamada a `printf`, por lo que el programa los muestra todos de corrido (como si fuera un solo número con muchas cifras). Ante esto, la estudiante dice "*esos números no son todos pares... está mostrando todos*". Se le pide que mire la salida con más detenimiento y dice "*ah, pasa que está todo pegado*". Se le pregunta cómo puede hacer para separarlos y dice "*con el tabulador*". Lo agrega, quedando la siguiente versión:

```
for (i=0; i <= N-1; i++)
{
    if ( arre[i] % 2 == 0)
     printf ("%d\t", arre [i]);

}
```

Al igual que la anterior, esta versión es correcta, solo que ahora muestra un tabulador entre cada valor y el siguiente. Compila y ejecuta de nuevo, constatando que funciona correctamente. Consultada acerca de la elección de estructuras de control para este nuevo problema, responde "*para ir celda por celda*" (`for`) y "*meto un if adentro... para ver si es par o no... y si es que muestre*". Tras compilar (no se producen errores) lo ejecuta y constata que funciona bien. La estudiante logra construir un programa para el *nuevo* problema, eligiendo estructuras de control adecuadas y comprende por qué funciona correctamente.

Anexo F

Pautas didácticas basadas en el modelo

En este anexo se proponen algunas recomendaciones generales que podrían servir como guía para el desarrollo a futuro de una línea de trabajo especializada en la elaboración de pautas didácticas para introducción de temas de programación en cursos iniciales de la disciplina. Dichas recomendaciones se basan en el modelo, se fundamentan por todo el marco teórico presentado en el trabajo y están alineadas en algunos aspectos con elementos de dos propuestas presentadas en la sección 1.4 del capítulo 1 (la *teoría de las situaciones didácticas* y la corriente *Computer Science Unplugged*). Se trata de pautas didácticas guiadas por el proceso de construcción de conocimiento, que parten del conocimiento *instrumental* de los estudiantes en relación a dichos temas, y siguen después con la construcción de conocimiento *conceptual* y luego *formal*. Este último a su vez puede usarse como base para la elaboración de nuevas pautas para la introducción de otros conceptos. Se incluye también un ejemplo concreto de pautas que siguen dichas recomendaciones como forma de ilustrar un posible enfoque para introducir un tema específico de programación en una clase.

Recomendaciones para elaboración de pautas didácticas

Una primera recomendación es comenzar por diseñar actividades que impliquen la interacción de los estudiantes con objetos en el plano de la acción. Es fundamental que las mismas se planifiquen de forma que su resolución en el plano *instrumental* tenga un correlato en el plano *formal* con el concepto que se desea construir. Esto se diferencia de otros enfoques usados con frecuencia al elaborar pautas didácticas, que empiezan por introducir conceptos de manera expositiva directamente en el plano *formal*. Los objetos pueden ser tanto *físicos* como *abstractos*. Para temas nuevos, particularmente en cursos iniciales, se recomienda que sean *físicos*, aunque no es excluyente. Conforme a lo visto en la sección 3.2.1 del capítulo 3, el establecimiento de una correspondencia entre objetos *físicos* y objetos *computacionales* ayuda a construir conceptos a partir de la acción, particularmente sobre *estructuras de datos* y facilita su posterior formalización, lo cual además se alinea con lo propuesto por *Computer Science Unplugged*, que promueve el uso de objetos físicos para su uso en actividades cenestésicas. Cualesquiera sean los objetos, las actividades deben diseñarse buscando minimizar (o eliminar, si es posible) todo elemento distractor que desvíe la atención de los estudiantes hacia aspectos no vinculados al concepto a construir.

Se recomienda que sean actividades que puedan ser resueltas de forma *autónoma* por los estudiantes. Esto se alinea con las situaciones *adidácticas* de la *teoría de las situaciones*, que propone la realización de actividades por parte de los estudiantes, sin intervención del docente. El propósito es que utilicen conocimiento *instrumental* para resolverlas. Según el tema a introducir, pueden ser actividades que impliquen aplicar conocimiento instrumental construido en forma previa (como la actividad propuesta en el estudio sobre *búsqueda lineal*, la cual es análoga a otras resueltas en situaciones cotidianas previas) o puede requerir la construcción del mismo al momento de su realización (por ejemplo, un juego nuevo cuyas reglas los estudiantes no conocen de antemano). En este último caso, se sugiere destinar tiempo a que puedan construir conocimiento instrumental por sucesivas repeticiones (por ejemplo, para el caso del juego, que lo jueguen varias veces). Cualquiera sea la actividad, es fundamental que llegue un punto en que logren realizarla en forma autónoma.

Una vez resueltas las actividades, se recomienda la definición de pautas orientadas a que los estudiantes reflexionen sobre lo realizado, expliquen cómo se resolvió y por qué se tuvo éxito. Conforme a lo visto en la sección 3.2.2 del capítulo 3, esto induce a tomar conciencia de las acciones realizadas y los cambios que ellas imponen a los objetos, lo cual resulta fundamental en el proceso de conceptualización. Dependiendo del tema a introducir y otros factores del sistema educativo (cantidad de estudiantes en el grupo, modalidad de dictado, etc.), puede hacerse de múltiples maneras (en forma oral, por escrito, con discusiones grupales, etc.). Cualquiera sea la metodología, el propósito es que logren dar descripciones detalladas, acordes a su accionar, de lo que se hizo y por qué funciona. La capacidad de expresión en lenguaje natural evidencia la construcción de conocimiento en el plano *conceptual*. Nótese además que, nuevamente en línea con la *teoría de las situaciones*, en este punto el docente interviene (situación *didáctica*) para orientar el trabajo y guiar a los estudiantes en la elaboración de explicaciones, por medio de preguntas y/o estrategias que estimulen a brindar descripciones acordes a lo realizado.

A partir del conocimiento *conceptual* construido, proponer luego nuevas actividades orientadas a su evolución hacia conocimiento *formal*. Conforme a lo visto en la sección 3.5.5 del capítulo 3, el uso de algún *formalismo intermedio* es un gran facilitador del proceso. En este sentido, puede aportar al diseño de actividades que ayuden a los estudiantes a construir progresivamente la expresión del concepto en lenguaje *formal*. De acuerdo con lo visto, una aproximación *gradual* y *dialéctica* hacia la formalización contribuye a la construcción de conocimiento en la etapa *trans*. Además, en caso de que el tema en cuestión involucre la escritura y ejecución de *programas*, el diseño de actividades que incorporen el mecanismo de *automatización* ayuda a construir conocimiento sobre cómo la computadora procesa las instrucciones, según lo explicado por la extensión a la ley general de la cognición. En este punto, la reflexión sobre cómo la *computadora* ejecuta las instrucciones e impone cambios a las *estructuras de datos* involucradas también resulta esencial en el proceso.

Finalmente, el conocimiento *formal* construido sobre un tema determinado puede usarse como punto de partida para el diseño de nuevas pautas didácticas que introduzcan nuevos temas. Según lo explicado en la sección 3.5.5 del capítulo 3, la herramienta cognitiva que interviene es la *generalización*. En caso de que el nuevo tema guarde razonables similitudes y diferencias con el anterior, esto puede hacerse trabajando directamente en el plano *formal*, en línea con la construcción de esquemas generales en la etapa *trans*. En otro caso, es posible definir pautas para comenzar a trabajar nuevamente en el plano *instrumental*, pero haciendo uso del conocimiento relativo al tema previo como base para el nuevo tema. La adopción de uno u otro enfoque dependerá del grado de similitudes y diferencias entre ambos temas.

Ejemplo concreto de posibles pautas didácticas

Se presenta una lista de pautas didácticas que siguen las recomendaciones anteriores para ilustrar una posible forma de introducir un nuevo tema en un curso inicial de programación. Se trata de una estructura de datos denominada *arreglo con tope*, junto con cuatro operaciones fundamentales para su manipulación: *inicialización*, *inserción*, *búsqueda* y *listado*. Al definir pautas, es esencial tomar en cuenta el conocimiento *previo* de los estudiantes y el contexto de aplicación de las pautas. Para este ejemplo, se asume un grupo de estudiantes de un curso universitario inicial en modalidad presencial, que cumplen con las mismas condiciones del estudio realizado en el captulo 3, con los siguientes agregados: ya han trabajado en el plano *formal* con *arreglos*, *subprogramas* (procedimientos y funciones) y tipos *estructurados*. Para el ejemplo, se asume que trabajan con el lenguaje *Pascal* (también podría hacerse con *C* o con otros), que utiliza el constructor *record* para definición de tipos de datos estructurados (*registros*). Las pautas propuestas para la introducción del tema son las siguientes:

1. Introducción de la estructura en el plano instrumental y conceptualización
2. Inicialización e inserción en el plano instrumental y conceptualización
3. Formalización de la estructura y operaciones de inicialización e inserción
4. Operaciones de búsqueda y listado trabajando directamente en el plano formal
5. Generalización de la estructura y las operaciones definidas en el plano formal

Las primeras dos pautas parten del conocimiento *instrumental* (etapa *intra*) de los estudiantes e inducen su transformación en conocimiento *conceptual* (etapa *inter*) en relación a la estructura y las operaciones de inicialización e inserción, conceptos *nuevos* para los estudiantes. La tercera pauta guía el pasaje de lo anterior hacia conocimiento *formal* (etapa *trans*). Las últimas dos pautas trabajan la construcción de conocimiento nuevo en la etapa *trans*, usando conocimiento formal *previo* y trabajando a partir de sus similitudes y diferencias con las nuevas nociones a construir (operaciones de *búsqueda* y *listado* y variantes más *generales* del arreglo con tope).

Pauta 1: Se propone la siguiente situación: *Se tiene un estante en el cual se irán insertando libros. Todos los libros tienen el mismo tamaño y se desea guardarlos en el estante de izquierda a derecha, de a uno por vez. El estante tiene capacidad para 20 libros y se desea poder saber, en todo momento, la cantidad de libros ya colocados sin necesidad de contarlos uno por uno. ¿Cómo se podría hacer para resolver esto último?*

La disposición propuesta tiene un correlato con la estructura de datos a construir en el plano *formal*. Los libros se corresponden con los valores a almacenar en las celdas y sus posiciones en el estante con los índices del arreglo (estructura de datos ya conocida por los estudiantes). La actividad introduce la necesidad de incorporar un elemento nuevo: el *tope*. Se trata de un valor entero que indica la cantidad de elementos almacenados hasta el momento. La consigna plantea que los estudiantes propongan una solución, en el plano de la acción, para resolver lo solicitado. Dependiendo de los recursos disponibles, se puede trabajar con un estante y algunos libros en forma física, o bien con un dibujo que los represente. Se espera que los estudiantes propongan ideas tales como "*poner una marca*" o "*anotar la cantidad en un papelito*". En base a las propuestas, preguntar a los estudiantes por qué cada alternativa funciona (o por qué no). Por ejemplo, poner una "*marca*" implica la necesidad de contar los libros que la anteceden, lo cual no cumple con el requisito de evitar contarlos. Tener un "*papelito*" brinda una mejor solución, ya que evita tener que hacer dicha cuenta cada vez. Es fundamental que los propios estudiantes evalúen los pros y contras de cada opción que sugieran y el docente oriente la reflexión al respecto, de ser necesario.

Pauta 2: Se propone la siguiente actividad: *Dado un libro, insertarlo en el estante según el procedimiento descrito y luego explicar paso a paso cómo se hizo y por qué funciona*.

Con esta actividad, se busca que los estudiantes interactúen con los objetos en el plano de la acción y apliquen conocimiento instrumental para su resolución. Todos han guardado objetos en algún estante previamente, por lo que deberían poder resolverlo. Se pide que lo hagan tres veces: con el estante vacío, con algunos libros ya colocados y con el estante completo. Esto tiene como objetivo que tomen conciencia tanto del *caso general* como de los *casos borde*. Deben comprender la necesidad de inicializar el tope en cero en forma *previa* a la inserción del primer libro y controlar que el tope no supere la cantidad máxima de libros que caben.

Posteriormente, deben explicar paso a paso el método empleado y por qué funciona. Como se vio en el estudio empírico, es esperable que den descripciones incompletas, que centren las mismas en sus *acciones*, sin tomar en cuenta los *cambios* que ellas imponen, y viceversa. Para este problema, las acciones fundamentales son tres: la comparación del valor en el papelito con la capacidad del estante, la inserción del libro a continuación del último y el incremento del valor en el papelito. Los cambios observables son dos: un cambio en la cantidad de libros en el estante (se incrementa en 1) o un cambio en la relación entre el valor del papelito y la capacidad (que pasa de ser *menor* a ser *igual*). El docente debe prestar atención a lo que dicen y hacer preguntas que ayuden a mover el foco hacia la parte faltante.

También es útil proponer *métodos alternativos* que, manteniendo lo que dicen, no conduzcan al resultado deseado y volver a la acción de ser necesario. El propósito es que den finalmente descripciones precisas y acordes con su accionar. Para este ejemplo, una posible descripción final podría ser: "*Si el estante no está lleno, sumo 1 al papelito y coloco el libro al lado del último libro que ya está*". En cuanto a las razones de éxito, una posible respuesta podría ser "*funciona porque tenía anotada la cantidad anterior en el papelito y así pude saber si aún tenía lugar"* (observar que la cantidad anterior podría ser cero, siendo fundamental que los estudiantes tomen conciencia de la importancia de su inicialización previa).

Pauta 3: Se propone la siguiente actividad: *Escribir un algoritmo en pseudocódigo que inserte un libro en el estante, según el procedimiento empleado*.

Como se vio en la sección 3.5.5 del capítulo 3, el uso de un formalismo intermedio ayuda a consolidar la conceptualización y dar inicio al pasaje a la etapa *trans*. Particularmente al introducir una operación *nueva* que, como en este caso, no tiene equivalente similar en el conocimiento *formal* previo de los estudiantes. Hasta el momento, solamente han trabajado con arreglos comunes (sin tope). En un arreglo común, la operación de inserción no está definida, pues siempre se cargan todas sus celdas. La noción de inserción de un nuevo elemento en una estructura de datos está siendo trabajada por primera vez con la introducción del arreglo con tope. Puede ser necesario que escriban más de una versión y deban volver a realizar la tarea en la acción. Para este ejemplo, se espera una versión final similar a la siguiente:

```
Si el estante no está lleno entonces
        sumo 1 al papelito
        inserto libro a continuación del último
Fin
```

Posteriormente, se pide que piensen cómo podrían definir en *Pascal* una estructura de datos adecuada para representar el estante de libros y el "*papelito*". En este punto, el docente indica que cada libro se representa por un número entero que corresponde a su isbn, a efectos de no introducir elementos adicionales que desvíen la atención de los estudiantes. Esta actividad implica construcción de conocimiento sobre el uso del lenguaje de programación para definir el arreglo con tope. Los estudiantes han trabajado en forma separada con *arreglos* y con *registros*, y su integración permite definir la nueva estructura de datos. Aprovechando que no se requieren elementos sintácticos nuevos, la pauta pide a los estudiantes que, haciendo uso de lo que ya conocen del lenguaje, propongan una idea para definir la nueva estructura. Así como en las pautas anteriores, proponen opciones y explican por qué sirven (o no). El docente interviene solamente para preguntar en caso de ser necesario. Finalmente, se espera que produzcan una declaración similar a la siguiente:

```
type Estante = record
                   libros : array [1..20] of integer;
                   papelito : integer;
               end;
```

Conforme a lo visto en la sección 2.5.4 del capítulo 2, el salto de instancias concretas al caso general se produce por repetición sucesiva de acciones. Si bien una definición como la anterior es correcta desde el punto de vista sintáctico, la misma evidencia que el pensamiento de los estudiantes aún está atado a la instancia concreta de *libros* manipulada en la acción (lo cual es natural que ocurra). En este punto, se aprovecha para trabajar el salto al caso general y fomentar la abstracción de la instancia concreta. Para ello, el docente puede introducir en este punto la terminología formal (*arreglo con tope*) y preguntar cómo se haría si en vez de 20 fuese cualquier cantidad de valores (al igual que en el estudio). Dado que los estudiantes han trabajado previamente con arreglos y el uso de una constante simbólica para definición del tamaño, se espera que, tras algunas preguntas y respuestas, modifiquen la definición anterior hacia una nueva versión, similar a la siguiente:

```
type ArregloConTope = record
                          arreglo : array [1..N] of integer;
                          tope : integer;
                      end;
```

A continuación, se propone otra actividad que solicita a los estudiantes escribir los siguientes procedimientos (se asume que han trabajado antes con *subprogramas*):

```
procedure inicializar (var act : ArregloConTope);
procedure insertar (valor : integer; var act : ArregloConTope);
```

Se espera que no surjan mayores dificultades al escribir ambos subprogramas, debido a la construcción progresiva de conceptos surgida de las actividades anteriores. En particular, el segundo procedimiento implica una nueva expresión del algoritmo en pseudocódigo. Conforme a lo visto en la sección 3.6.8 del capítulo 3, este proceso es llevado a cabo con bastante facilidad en la mayoría de los casos. Durante la escritura, es esperable que los estudiantes cometan errores propios de la ejecución en máquina. Por ejemplo, un problema de borde con algún índice que produzca un error de *salida de rango*, o acceder a una celda con valor indefinido ("*basura*"). En tales casos, se recomienda recurrir a la *automatización*, a efectos de que construyan conocimiento relativo a la ejecución en computadora. Se espera que escriban versiones finales similares a las siguientes, evidenciando así la construcción de conocimiento *formal* sobre las operaciones de *inicialización* e *inserción*:

```
procedure inicializar (var act : ArregloConTope);
begin
    act.tope := 0
end;
```

```
procedure insertar (valor : integer; var act : ArregloConTope);
begin
    if act.tope < N then
    begin
        act.tope := act.tope + 1;
        act.arreglo [act.tope] := valor
    end
end;
```

Pauta 4: Esta pauta fue diseñada tomando en cuenta el conocimiento *previo* de los estudiantes (en el plano *formal*) de la estructura de datos *arreglo*. Partiendo del mismo, la pauta pide que escriban, trabajando directamente en el *lenguaje de programación*, las operaciones de *búsqueda* y *listado* sobre el arreglo con tope, y se proponen los siguientes encabezados:

```
function buscar (act : ArregloConTope; num : integer) : boolean;
procedure listar (act : ArregloConTope);
```

A diferencia de la operación de *inserción* (que es nueva) los estudiantes ya han implementado las operaciones de *búsqueda* y *listado* sobre un arreglo común. La pauta toma esto en cuenta y busca que adapten dicho conocimiento para la implementación de nuevas versiones, trabajando ahora sobre el arreglo con tope. Las mismas guardan razonables similitudes y diferencias con el arreglo común, lo cual brinda un contexto adecuado para la construcción de conocimiento nuevo trabajando en el plano *formal* (herramienta cognitiva *generalización*). Nuevamente, puede hacer falta recurrir a la *automatización* durante el proceso. Como ejemplo, se muestra una versión de `listar` similar a la que podrían haber escrito antes en el curso para un arreglo común y a continuación una nueva versión que podrían escribir para el arreglo con tope.

```
procedure listar (arre : Arreglo);
var i : integer;
begin
    for i := 1 to N do
        writeln (arre[i])
end;

procedure listar (act : ArregloConTope);
var i : integer;
begin
    for i := 1 to act.tope do
        writeln (act.arreglo[i])
end;
```

Una variante a la pauta podría ser solicitar primero que escriban nuevamente las versiones para el arreglo común (lo que además permite repasar lo aprendido previamente) y luego las nuevas versiones para el arreglo con tope, a efectos de tener a la vista las similitudes y diferencias entre ambas. Para el caso de `listar`, la similitud es el uso de `for` como estructura adecuada para iterar sobre todo el rango de valores válidos. Las diferencias son el límite superior de la iteración (`N` para el arreglo y

`act.tope` para el arreglo con tope) y la sintaxis de acceso a las celdas (`arre[i]` para el arreglo y `act.arreglo[i]` para el arreglo con tope). Además, se recomienda pedir que los propios estudiantes hagan explícitas las similitudes y diferencias, lo cual contribuye a la consolidación de los conceptos.

Pauta 5: Se propone la siguiente actividad: *Supóngase que, en vez de valores enteros, se desea almacenar parejas (formadas por un entero y un carácter) en el arreglo con tope. ¿Qué cambios sería necesario realizar a la declaración y a los subprogramas implementados?*

Al igual que la anterior, esta pauta parte del conocimiento *previo* de los estudiantes en el plano *formal*. A diferencia de la anterior, toma en cuenta el conocimiento construido en relación a la *nueva* estructura (arreglo con tope) y constituye un punto de partida para construir la noción de *tipo genérico*. La pauta no pretende abordar dicha construcción en profundidad, sino dar un primer paso en esa dirección. Conforme a lo visto en la sección 4.2.1 del capítulo 4, se requieren nuevas investigaciones a futuro que profundicen en la construcción de dicha noción. A efectos de esta pauta, se busca que comprendan que es posible almacenar elementos de *cualquier* tipo en las celdas del arreglo con tope. Nuevamente se requiere contrastar similitudes y diferencias entre la solución anterior y la nueva, trabajando una vez más en el plano *formal*. Como en las pautas anteriores, se debe estimular que las ideas surjan de los propios estudiantes y que expliquen por qué sirven o no. Nuevamente, el docente interviene solo para hacer preguntas o estimular que pongan el foco en aspectos no contemplados. Después, se pide que ellos mismos escriban nuevas versiones de la declaración del arreglo con tope y los subprogramas implementados. Para finalizar, se ilustra una posible declaración que podrían proponer, junto con una nueva versión del procedimiento `listar`. Se recomienda además que los estudiantes compilen y ejecuten en máquina, tanto las versiones anteriores como las nuevas, para validación del conocimiento construido, mediante el uso de la computadora como agente externo ejecutor en acción.

```
type Pareja = record
                ent : integer;
                car : char;
              end;
     ArregloConTope = record
                        arreglo : array [1..N] of Pareja;
                        tope : integer;
                      end;

procedure listar (act : ArregloConTope);
var i : integer;
begin
    for i := 1 to act.tope do
        writeln (act.arreglo[i].ent, ' ', act.arreglo[i].car)
end;
```