Revista Colombiana de Computación
Vol. 21, No. 2. July - December 2020, pp. 71-82
e-ISSN: 2539-2115, **https://doi.org/10.29375/25392115.4035**

REVISTA COLOMBIANA DE COMPUTACIÓN

Posibles aportes del razonamiento analógico al problema de la abstracción y transferencia en la enseñanza de programación

Possible contributions of analogical reasoning to the problem of abstraction and transfer in programming teaching

Verónica D'Angelo[1]

[1]Universidad Abierta Interamericana, Rosario, Argentina
veronica.dangelo@uai.edu.ar

(Recibido: 25 Julio 2020; aceptado: 20 Octubre 2020; Publicado en Internet: 1 Diciembre 2020)

Resumen. En este artículo de revisión se explora una posible contribución de las investigaciones sobre razonamiento analógico al problema de la transferencia en programación -en la transición entre el aprendizaje de conceptos en la escuela media y su aplicación en la universidad. La facilidad con que los alumnos construyen programas en entornos multimedia conlleva la desventaja de una dificultad para trasladar esos conceptos a los lenguajes "reales" basados en texto, probablemente porque no se ha trabajado suficiente en promover abstracciones en el *nivel del problema*. Según investigaciones en enseñanza de la programación, los alumnos suelen tener mayor dificultad en los niveles de abstracción superior (la comprensión del problema) que en los niveles inferiores (como la codificación). La comparación de problemas mediante razonamiento analógico es una estrategia proveniente de la psicología cognitiva extendida a diversas disciplinas. Sugerimos que su aplicación en el campo de la enseñanza de la programación podría contribuir a solucionar el problema de la dificultad de abstracción en el nivel del problema, y facilitar la transferencia.

Palabras clave: Programación de ordenadores, Razonamiento analógico, Abstracción, Transferencia.

Abstract. This review article explores a possible contribution of research on analogical reasoning to the problem of transfer in programming -in the transition between the learning of concepts in middle school and their application at university. The ease with which students construct programs in multimedia environments carries the disadvantage of translating these concepts into "real" text-based languages, probably because not enough work has been done on the *problem level*. According to research in teaching programming, students tend to have greater difficulty at higher levels of abstraction (understanding the problem) than at lower levels (such as coding). Comparing problems through analogical reasoning is a strategy from cognitive psychology extended to various disciplines. We suggest that its application in programming teaching could contribute to solving the problem of the difficulty of abstraction at the problem level and facilitate the transfer.

Keywords: Computer programming, Analogical reasoning, Abstraction, Transfer.

Tipo de artículo: Artículo de revisión.

1 Introducción

Dado que este artículo trata sobre abstracción y transferencia de conceptos en programación, en primer lugar, se caracterizará la programación de ordenadores como un campo específico dentro de las ciencias de la computación (CC), cuya especificidad lo distingue de otras actividades, a saber: el uso de las tecnologías de la información y las comunicaciones (TIC), la manipulación de datos para la generación de información (ofimática), la resolución de problemas, o el pensamiento computacional, siendo éste último el concepto más general y popular, dentro de las que se denominan "iniciativas" de introducción a las CC (e.g., csunplugged.org, bebras.org, code.org.). El término "iniciativas" engloba tanto investigaciones como intervenciones educativas que suelen incluir tanto diseños colaborativos para intercambios sociales como enseñanza de la programación. Dado que no existe consenso acerca de cuáles son todos los procesos cognitivos involucrados en el pensamiento computacional (aunque se menciona la capacidad de abstracción como uno de sus pilares), este artículo sólo se enfocará en la programación de ordenadores digitales, que data desde la invención de las computadoras, y atiende a un conjunto conocido de objetivos de aprendizaje que podrían resumirse en: (1) identificación de problemas del mundo real cuya solución es

"algoritmizable", (2) algoritmización mediante estructuras de control -que incluyen los conceptos de (3) secuencia temporal de acciones, (4) iteración, (5) bifurcación, (6) concurrencia-, tratamiento de datos –ya sea (7) la entrada de datos, (8) la salida, o (9) el almacenamiento, (10) prueba de programas y, eventualmente, compilación en algún lenguaje particular y depuración. Los conceptos mínimos necesarios para programar, definidos en estos diez puntos, no incluyen ni deben confundirse con las actividades TIC, por ejemplo, la comunicación a través de una red social ni el diseño de animaciones. Éstas forman parte de entornos que han sido diseñados con el objeto de motivar a los jóvenes a introducirse en la temática de la programación y facilitar el aprendizaje de los objetivos de programación antes mencionados. Para verificar si esos entornos han logrado los objetivos de motivación y, además, los objetivos de aprendizaje, los investigadores deben realizar estudios que trasciendan la pura intervención educativa y pongan a prueba sus supuestos para orientar a los docentes hacia las estrategias más efectivas.

El punto de vista del movimiento computacional iniciado hace 14 años (Wing, 2006) era general, se instaba a "todas" las personas a adquirir un "pensamiento computacional". Este movimiento marchaba en paralelo con las preocupaciones en los ministerios de educación y en el ámbito empresarial, acerca de la falta de profesionales y la pérdida del interés por las carreras técnicas, que motivaron proyectos, cambios en planes de estudio e incluso nuevas carreras (e.g., en Argentina, la Fundación Sadosky, 2013; PEFI, 2012/2016). El lema del pensamiento computacional de "incluir a todos", basándose fuertemente en el juego en entornos multimediales y en actividades recreativas *on line* (con excepción de los enfoques *unplugged,* por ejemplo, Bell et al. (2012) lograron captar la atención de los más jóvenes. Mientras los kits de electrónica pusieron la robótica (ensamblado de partes funcionales) al alcance de los alumnos, la gran flexibilidad de entornos como Scratch, motivó que profesionales de distintas áreas, por ejemplo, los cientistas de la educación, se involucraran en la enseñanza de la programación con objetivos sociales más variados y enriquecidos que las tradicionales metas de la ingeniería informática, logrando diseñar e implementar entornos colaborativos y comunidades de aprendizaje para transmitir el patrimonio cultural nacional, entre otras contribuciones, al tiempo que incentivaron la creatividad.

Sin dejar de reconocer la importancia del aporte social que han realizado dichas iniciativas, se reivindica la propuesta inicial hacia las esperadas vocaciones en ingeniería, que hoy demanda nuevas investigaciones para averiguar qué transferencias se están logrando entre el nivel medio y la universidad en el área específica de ciencias de la computación.

En este punto cabe incorporar una nueva distinción entre los conceptos de aprendizaje y transferencia. El aprendizaje de un concepto, en sentido amplio, suele limitarse a su comprensión inmediata o a corto plazo. La transferencia, en cambio, es lograr aplicar un conocimiento aprendido en un entorno distante -física y temporalmente- de aquél en el cual se aprendió. Es por ello que algunos consideran que el verdadero aprendizaje incluye la transferencia o, lo que es lo mismo, que un aprendizaje sin transferencia no es aprendizaje. Vislumbrar qué tipo de relación mediaría entre dichos aprendizajes o qué distancia deberían mantener los dominios para posibilitar la transferencia de uno a otro, han sido preocupaciones de los educadores desde principios del siglo XX aún no resueltas. Barnett & Ceci (2002) ofrecen una revisión al respecto y un marco taxonómico para evaluar cuándo se produce la transferencia entre dominios distantes.

Siendo la programación de ordenadores una asignatura esencial en el curriculum de ciencias de la computación, es fundamental hacer un seguimiento del aprendizaje (y su transferencia) en la etapa de transición que va desde la finalización de la escuela media donde se adquieren los conceptos de programación en entornos multimediales (e.g., Scratch), hasta el reconocimiento y aplicación de estos conceptos en lenguajes de programación tradicionales durante el ciclo básico universitario (e.g., C++) durante las asignaturas que constituyen las primeras experiencias de algorítmica que derivan en la implementación (codificación) en lenguajes de alto nivel (e.g., Programación estructurada, Programación orientada a objetos).

A medida que se profundiza el conocimiento de los principios básicos de las Ciencias de la Computación (CC) (Denning, 1985, 2003, 2017) la abstracción se presenta como uno de sus rasgos esenciales (Denning et al., 1989) y uno de los objetivos fundamentales de enseñanza, según destacados investigadores (Armoni, 2013; Hazzan & Kramer, 2007, 2016; Statter & Armoni, 2016) y según estándares educativos (Seehorn, 2011).

Las iniciativas de introducir a los estudiantes jóvenes en el pensamiento computacional enfatizan que es fundamental enseñar conceptos, no sólo habilidades (Wing, 2006, 2008), para lo cual es fundamental desarrollar en los estudiantes la capacidad abstractiva. Paralelamente, desde hace décadas, los investigadores en educación en CC han desarrollado entornos para motivar a los jóvenes a introducirse en el ámbito informático por medio de juegos y escenarios visual y auditivamente atractivos. El interés que los alumnos demuestran por interactuar con estas aplicaciones no debe confundirse con la motivación por

aprender conceptos abstractos. Una preocupación emergente entre algunos pedagogos es que los niños dediquen demasiado tiempo al juego y muy poco a la conceptualización, ya que ésta no se da espontáneamente (Armoni, 2013; Armoni & Ben-Ari, 2013). Por otra parte, algunos conceptos fundamentales de programación, por ejemplo, la inicialización, que son críticos en algunos lenguajes, no son visibles en estos entornos, y no pueden ser transferidos a los entornos "reales" de programación a menos que se diseñen estrategias específicas de transferencia (Franklin et al., 2016).

2 ¿Qué es la abstracción?

La abstracción "*es un proceso de identificación de un conjunto de características esenciales invariantes de una cosa*" (Burgoon et al., 2013, p. 502). En tanto proceso cognitivo, es algo "*que se hace*" no algo que se observa, pero en CC se suele utilizar el término abstracción también para referir a los productos del proceso de abstraer. Así, por ejemplo, llamamos abstracciones de datos a las estructuras de datos, tales como un arreglo bidimensional, o una base de datos.

Lo que se espera de los estudiantes de programación es que logren identificar (abstraer) la estructura algorítmica común que subyace a problemas del mundo real, aún cuando dichos problemas sean aparentemente distintos por ocurrir en diferentes dominios. Por ejemplo, el problema "el cliente del banco ingresa su clave en un cajero" es estructuralmente similar al problema "el alumno completa el campo 'sexo' en un formulario electrónico de ingreso a la universidad".

Tanto el ingreso de "M" o "F" en el campo "sexo" como el ingreso de una clave en el cajero, están mediados por un algoritmo de validación. Un programador experto, frente a la situación "el usuario ingresa su clave en un cajero", percibe este problema como la manipulación de una secuencia de datos y no como un simple ingreso, ya que al ser un problema de validación, se anticipa que podría haber una secuencia de datos inválidos que deben ser rechazados hasta que el usuario introduzca el dato válido. Si el programador experto pensara además en el algoritmo, (suponiendo que no hubiera límite en la cantidad de intentos permitidos), identificaría rápidamente una estructura subyacente de iteración con número de repeticiones indefinido (*REPETIR HASTA*) en el algoritmo, cuya condición de fin es "que la clave ingresada sea correcta". De modo análogo, si el programador se dedicara al problema de restringir el ingreso de letras en un campo de formulario a sólo "M" o "F" para "sexo", anticiparía el mismo tipo de secuencia de datos inválidos posibles (todas las letras que no son ni "M" ni "F") y la misma estructura de iteración con número de repeticiones indefinido cuya condición de fin es "que se ingrese una "M" o una "F". A nivel abstracto, ambos problemas son equivalentes. A nivel superficial, tratan sobre temas distintos.

Algunos pedagogos de la enseñanza de programación, clasifican los niveles de abstracción. Por ejemplo, las abstracciones que contienen detalles como el tipo de estructura de iteración a utilizar, sin especificar el lenguaje ni la sintaxis, están ubicadas en un nivel de abstracción que denominan "nivel de algoritmo", mientras que el "nivel del problema" es más general, por ejemplo, puede aludir a conjuntos de datos de entrada o salida. Por otro lado, denominan "nivel del programa", al código con todo el detalle de su implementación en un lenguaje particular. Lo más difícil para los alumnos, afirman, es pasar de un nivel de abstracción a otro. Se argumentará que esta dificultad podría deberse a que no se ha identificado la estructura abstracta subyacente a nivel de problema y algoritmo, ya que se suele invertir cada vez menos tiempo de enseñanza en estos niveles, bien porque los alumnos se apresuran por pasar a la codificación, o porque han aprendido a programar con bloques de código antes de introducirse en el análisis y comparación de situaciones. Se sugerirá, además, que realizar comparaciones de ejemplos de situaciones problemáticas podría contribuir a reforzar las abstracciones (esquemas adquiridos) en el nivel de problema y promover la transferencia.

3 Niveles de abstracción

Perrenet et al. (2005) establecen una taxonomía de niveles de abstracción específicos de programación con objetivos pedagógicos.

Si bien algunos investigadores en CC conciben la abstracción como una capacidad con la que el alumno debe contar para iniciarse en carreras de informáticas, otros la ven como un proceso que al desarrollarse promueve dicha capacidad (Armoni, 2013; Hazzan, 2008). Según esta postura, el profesor debería indicar

explícitamente cuándo se realiza un pasaje a otro nivel, 'subiendo o bajando' en el nivel de abstracción. Así, por ejemplo, cuando el profesor habla sobre el problema del cajero puede mencionar la estructura de control ($REPETIR\ HASTA\ QUE\ clave - ingresada = clave\_valida$) pero aclarando que ha "descendido" a otro nivel de abstracción. Para ayudar al alumno a desarrollar esa capacidad y auto regularse es recomendable mencionar explícitamente los niveles de abstracción y los cambios de nivel.

Hazzan (1999, 2003) postula que los estudiantes, enfrentados a tareas demasiado abstractas, tienden a reducir el nivel de abstracción utilizando diversos mecanismos inconscientes para lograr que los conceptos se vuelvan accesibles para ellos. En otras palabras, los alumnos tienen dificultades para trabajar en niveles de abstracción elevados. A diferencia de otros autores, que sostienen que la abstracción debe enseñarse indirectamente, Hazzan afirma que es necesario un metacomponente explícito reflexivo. Los alumnos deberían aprender a "despegarse" de los niveles concretos hacia niveles de abstracción superiores, y realizar pasajes de un nivel a otro siendo conscientes de ello.

Tomando como base la teoría de Hazzan, Perrenet et al. (2005) y Perrenet & Kaasenbrood (2006) diseñaron una jerarquía de niveles de abstracción que utilizaron para investigar la comprensión que los estudiantes tienen del concepto de algoritmo (Perrenet, 2010).

1) El nivel más bajo es el nivel de ejecución. Cualquier proceso que se esté ejecutando en un momento determinado en una computadora concreta para resolver una tarea específica.
2) El siguiente nivel, es el programa escrito en un lenguaje de programación específico, considerando la sintaxis de ese lenguaje. Se trata de una implementación. Por ejemplo, un programa en C para ordenar un conjunto de números.
3) El siguiente nivel, Armoni (2013) sugiere denominarlo nivel de 'algoritmo'. Se trata de una especificación. Nos referimos a un conjunto de instrucciones para realizar funciones, sin especificar detalles de sintaxis, sin estar ligado a un lenguaje de programación. En las cátedras de programación se suele programar en pseudocódigo y se recalca la necesidad de no mencionar detalles de sintaxis, sino sólo las funciones que se realizan.
4) El nivel más alto de abstracción es el nivel de problema. Aquí se pueden pensar los problemas como pertenecientes a familias de problemas de un tipo.

Por ejemplo, en el nivel 4, un problema de búsqueda dicotómica puede representarse con actividades kinestésicas (juegos), puede explicarse verbalmente o por escrito con expresiones generales, tales como "dividir el conjunto en dos mitades, buscar en una de las mitades, si no hay nada descartar la mitad completa…". El problema puede incluso ser identificado dentro de una familia de problemas y no es necesario usar tecnicismos ni sintaxis. En el nivel 3, el algoritmo, se escribe en pseudocódigo, un código que 'parece' un lenguaje de computadora, pero utiliza términos del lenguaje natural para referir a acciones que ejecutará un autómata.

Trabajar con Scratch o con cualquier lenguaje concreto refiere al nivel 2. Los autores sugieren que en las clases de programación se comience hablando siempre en el nivel del problema, y luego se vaya descendiendo a los niveles concretos, porque en las investigaciones se halló que los alumnos suelen preferir los niveles concretos, y cuando están inmersos en ellos, por ejemplo, en un lenguaje visual como Scratch, o un lenguaje artificial profesional, se focalizan en manipular la implementación (el código o los bloques visuales) pero no logran resolver el problema, para lo cual deberían poder pensar en un nivel más abstracto.

El nivel de ejecución también debe ser identificado y diferenciado. Por ejemplo, la instrucción $REPETIR\ 100\ veces$: $acción\ x$, es una única instrucción, pero el ordenador ejecutará 100 ciclos, 100 veces la acción x. El tiempo de ejecución en una máquina concreta es un nivel distinto al de la escritura del programa.

4 Enseñar conceptos

Bajo el mismo principio de diferenciar niveles de abstracción y comenzar por lo general, Meerbaum-Salant et al. (2013, 2010) proponen comenzar enseñando los conceptos y luego pasar a su implementación en Scratch, en lugar de que los alumnos comiencen jugando en Scratch desde el inicio. Para tal fin, utilizaron el libro de texto "*Computer Science Concepts in Scratch*" (Armoni & Ben-Ari, 2013) desarrollado por ellos específicamente para la enseñanza. A medida que se mencionan los conceptos para resolver una tarea específica (y se proporcionan ordenadamente en el libro) se va diferenciando el nivel conceptual del nivel

práctico concreto. Cuando un proceso de solución implica la introducción de un nuevo concepto, ésta se realiza primero a nivel conceptual, y solo más adelante en Scratch (Nivel 2). Perrenet et al., (2005) establecen una taxonomía de niveles de abstracción específicos de programación con objetivos pedagógicos.

Se utilizan ayudas lingüísticas para diferenciar entre los niveles 2 y 3. La sintaxis de Scratch es relativamente natural y un script en Scratch se lee igual que el lenguaje natural. Sin embargo, al igual que cualquier otro lenguaje de programación, esta sintaxis se compone de "palabras reservadas". Por lo tanto, limitan el uso de esas palabras al nivel 2 y utilizan otras palabras más generales para el Nivel 3, así como frases muy naturales, flexibles y variables pero muy precisas y sin ambigüedades.

5 Lenguajes visuales vs lenguajes basados en texto

Desde los inicios de la programación estructurada, los cientistas de la computación propusieron métodos normativos de modelado de algoritmos acordes a ciertas restricciones de claridad, modularidad y enfoque *top-down* (Dahl et al., 1972; Wirth, 1976), así como técnicas de diagramación (Nassi & Shneiderman, 1973; Shneiderman et al., 1977). Algunos de los primeros trabajos en psicología de la programación cuestionaron que los enfoques *top-down* no resultaban intuitivos para los programadores (J. Hoc et al., 1990). El construccionismo de Papert (Harel & Papert, 1991; Papert & Harel, 1991), promovió mediante el lenguaje Logo (Papert, 1980) una forma exploratoria de programar que prometía adaptarse a todos los estilos cognitivos y culturales (Turkle & Papert, 1990) y sirvió como modelo para el desarrollo de Scratch (Resnick et al., 2009), un lenguaje interactivo basado en bloques visuales que permite programar a través del diseño de animaciones sin la necesidad de aprender la sintaxis compleja de los lenguajes basados en texto (LBT).

En los lenguajes basados en bloques visuales (LBBV) como Alice, Scratch o La Playa, en lugar de escribir programas en forma de texto, los usuarios arrastran y colocan con el mouse los bloques visuales que seleccionan en pantalla (como expresiones, condiciones, y declaraciones) según los consideren adecuados para cada problema en particular. Este modo de interacción evita la frustración de los alumnos ante los errores de sintaxis triviales .

La enseñanza tradicional de programación en entornos universitarios y profesionales continúa siendo la escritura de programas con lenguajes basados en texto (LBT) comenzando por el llamado pseudocódigo hasta llegar a los lenguajes profesionales. El primero es un conjunto de palabras (convención) del lenguaje natural que han sido seleccionadas para referir a determinadas operaciones básicas en un computador, por ej. MOSTRAR ('a') exhibe la letra a en pantalla.

Los alumnos ingresantes a carreras en CC, cursan durante el primer año una asignatura anual dividida en dos etapas (o dos asignaturas cuatrimestrales) dedicada a introducir los conceptos básicos de programación.

En la primer etapa (primer cuatrimestre) se introducen los conceptos básicos de programación utilizando pseudocódigo (texto), y durante el segundo cuatrimestre se presentan los mismos conceptos programados en lenguajes específicos, por ejemplo C++, Python, Java.

En los últimos años, se ha promocionado la actividad de programar a través de LBBV. En Argentina, el Plan Conectar Igualdad (2010) reorientó la inclusión de las TIC –como asignatura obligatoria en escuelas medias desde la ley 26206- hacia el aprendizaje de la Ciencia de la Computación, incorporando software específico como Scratch o Alice, también accesible por Internet. De modo tal que la mayoría de los alumnos ingresantes a carreras de Ingeniería en Computación, han adquirido nociones de programación en dichos entornos.

Otra diferencia importante entre ambos entornos de programación, está en el tipo de problemas a resolver. A diferencia de las aplicaciones *on line,* los problemas de programación que se resuelven en la universidad, suelen ser de cierta envergadura, muchos de ellos abiertos, admitiéndose más de una solución posible. En los entornos LBBV, en cambio, se suele trabajar con problemas que tratan sobre la transformación espacio temporal de objetos visuales (animaciones). Dicha transformación es interactiva, cada cambio introducido por el programador/diseñador en los valores de las variables, produce un efecto visible de modo inmediato, con lo cual, en este modo de programar, no es necesaria una etapa previa de comprensión del problema en un tiempo diferido a la implementación, el problema se resuelve y se modifica constantemente. La ventaja más evidente es la facilidad para efectuar modificaciones, una desventaja no tan obvia es que el diseñador no tiene oportunidad de distanciarse de los objetos concretos porque no hay una interrupción temporal mediando entre el diseño y la acción, esto podría, a nuestro entender, dificultar

los procesos de abstracción necesarios para la formulación de estrategias[1]. La Figura 1 presenta la diferencia entre lenguajes basados en bloques visuales (LBBV) y lenguajes basados en texto (LBT).

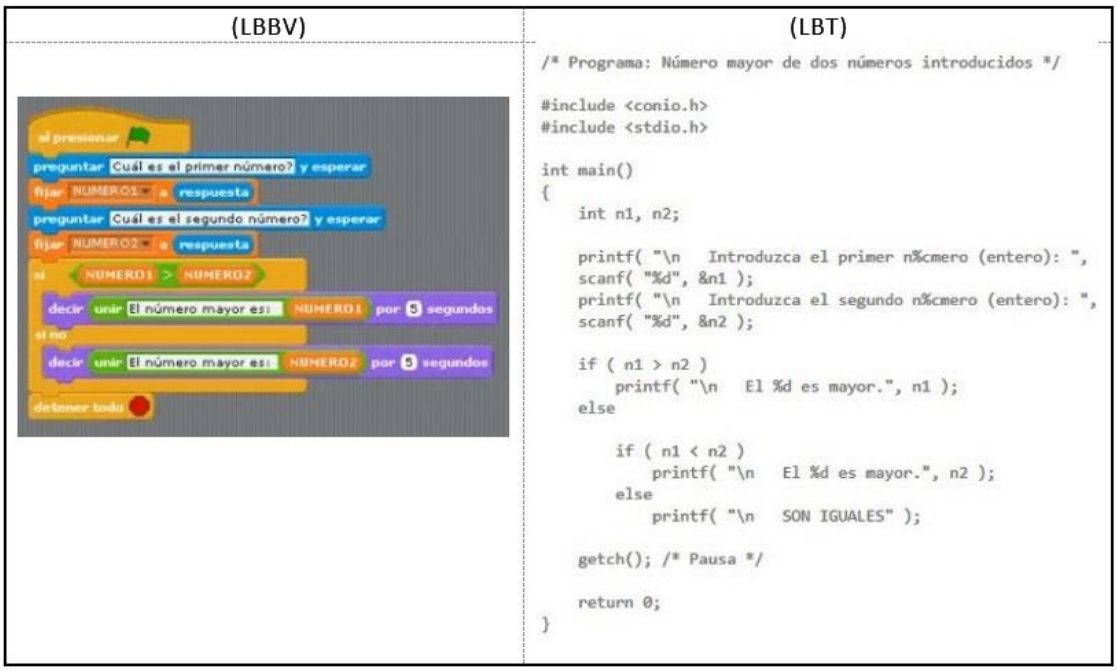

Figura 1. LBBV vs. LBT

6 Antecedentes sobre transferencia de conceptos desde Scratch

Resnick et al. (2009) afirman que los objetivos principales del entorno de programación Scratch son mejorar la creatividad permitiendo la libre exploración según los fundamentos del construccionismo (Kafai & Resnick, 1996). Se trataría de una herramienta para dar soporte al pensamiento creativo (Resnick et al., 2005). Al respecto, surgieron interrogantes en torno a varios puntos:

A. La transición entre los entornos visuales y los entornos 'reales'.
B. La carga cognitiva de la manipulación de la interface.
C. La necesidad de acompañamiento (docente) durante la actividad interactiva.
D. Los niveles de abstracción implicados.

Si bien algunos educadores proponen a Scratch como un excelente recurso para que los estudiantes universitarios se inicien en programación (Malan & Leitner, 2007) es sabido que son importantes las diferencias entre el entorno de aprendizaje en Scratch y la escritura de programas con lenguajes tradicionales, con formato de texto, teniendo que lidiar con la sintaxis, la organización secuencial de los procesos y la comprensión de las palabras reservadas, entre otras dificultades. Sin embargo, los conceptos básicos de programación son los mismos, es decir, si los estudiantes captan esos conceptos, el camino hacia la programación tradicional estaría allanado. Aunque los estudiantes no aprenden la sintaxis de C++ o Java usando Scratch, aprenden el tipo de problema y tipos de estructuras para resolverlo. Algunos estudios trabajan sobre la transición entre ambos entornos. Franklin et al. (2016) estudió la transferencia de conocimiento entre LBBV y LBT con respecto a la operación de inicialización en estudiantes de entre 9 y

[1] Diferencia entre estrategias y rutinas: Las rutinas o procedimientos son conocimientos de tipo procedural y pueden aprenderse fácilmente y automatizarse. Cada estructura de control en programación (iteración, bifurcación, algoritmos básicos) es una rutina y funciona de un modo predecible, pero una estrategia implica la toma de decisiones sobre qué rutinas seleccionar en cada problema en particular, cómo combinarlas y por qué, por lo tanto, es un proceso metacognitivo.

12 años, analizando las diferencias entre ambos entornos y sugiriendo estrategias de "*bridging*" y "*hagging*" (Perkins & Salomon, 1988) para mejorar la transferencia.

Los interrogantes acerca de si la carga cognitiva de la manipulación de la interface interfiere con la comprensión de conceptos abstractos que subyacen al mismo no han sido respondidos. Çakiroğlu et al. (2018) evaluaron el nivel de la carga cognitiva auto percibida en alumnos de sexto grado concluyendo que algunos componentes en Scratch podrían conducir a una mayor carga cognitiva, en especial para el concepto de secuencia, y aportan sugerencias didácticas.

Respecto de la necesidad de instrucción, la mayoría de las investigaciones optimistas sobre un proceso autodidacta han realizado observaciones desde un punto de vista afectivo o actitudinal, y han mostrado un cambio positivo evidente en la actitud de los estudiantes respecto de las ciencias computacionales, así como una preferencia por utilizar Scratch en vez de lenguajes tradicionales y se ha documentado el aprendizaje de conceptos clave de programación incluso en ausencia de intervenciones de instrucción con docentes expertos (Maloney et al., 2008) lo que sugiere que con Scratch cualquier persona puede aprender como autodidacta (sin instrucción). Sin embargo, como señalan (Meerbaum-Salant et al., 2013, p. 241, 2010, p. 70), cuando los investigadores señalan que los estudiantes utilizan estructuras de secuencia y de concurrencia en los proyectos "sin darse cuenta" de que lo hacen, se evidencia que no han investigado si el uso de la estructura implica la comprensión del concepto. (Meerbaum-Salant et al., 2013), contrariamente a la postura autodidacta, hallaron que los aprendizajes significativos de los conceptos de CC en Scratch se producen sólo como resultado de la instrucción, en los casos en que esos conceptos han sido explícitamente enseñados por un docente a la par de las actividades interactivas. Advierten además, en (Meerbaum-Salant et al. (2011) sobre algunos hábitos que se engendran en este tipo de programación, contrarios a los esperados en CC. En Armoni et al. (2015) analizaron la transición entre estudiar con el entorno visual de Scratch en la escuela media a estudiar con un lenguaje tradicional basado en texto (C++ o Java) con estudiantes entre 15 y 16 años, y hallaron que la experiencia de programación de los estudiantes que habían aprendido Scratch facilitó el aprendizaje del material más avanzado acelerando los tiempos de aprendizaje y logrando niveles cognitivos de comprensión superiores, aunque al final del proceso de enseñanza no hubo diferencias significativas en comparación con estudiantes que no habían estudiado Scratch.

7 Razonamiento analógico y transferencia

Enmarcadas dentro de las ciencias cognitivas, las teorías del razonamiento analógico, surgieron inicialmente para explicar cómo las personas entienden las analogías (Gentner, 1983; Holyoak & Thagard, 1989) y se extendieron luego al problema de la transferencia de conocimiento. En dichas teorías, las analogías se definen como comparaciones basadas en puntos en común entre dos análogos, por ejemplo, en la analogía entre el modelo atómico de Rutherford y el sistema solar, se suelen resaltar los aspectos comunes a ambos sistemas. Por ejemplo, en ambos casos hay objetos de menor tamaño girando alrededor de un objeto más grande, la relación entre sus masas es similar (Gentner & Markman, 1997).

El hecho de que las personas tengan dificultades para aplicar sus conocimientos aprendidos a situaciones estructuralmente análogas pero superficialmente diferentes ha sido explicado por los investigadores del pensamiento analógico como un obstáculo inherente a nuestra capacidad de razonamiento basada en un mecanismo que resultó adaptativo, en tanto económico: las señales estructuralmente similares (análogas), serían mucho más costosas en términos de procesamiento (Gentner, 1989). Es así que en la mayoría de los estudios experimentales de recuperación analógica (Gentner et al., 1993; Trench & Minervino, 2017), los participantes proponen analogías intradominio (dentro del mismo tema). Por ejemplo, si se presentara a los alumnos ingresantes la situación del cajero y la situación del alumno ingresante, probablemente no hallarían similitudes, salvo que ambos utilizan una computadora, pero ésa es una similitud superficial, no profunda. En cambio, un programador con experiencia, que posee esquemas formados a partir de una gran cantidad de problemas resueltos, advertirá que en ambas situaciones hay un problema de validación. Los expertos en un área tienden a identificar estructuras conceptuales a diferencia de los novatos (Chi et al., 1981).

Gick & Holyoak (1980) demostraron que el recuerdo espontáneo de un problema anterior por coincidencia estructural con un problema actual, es muy raro. Adicionalmente, Gick & Holyoak (1983), descubrieron que puede inducirse un esquema a partir de la comparación de dos análogos, y lograr efectos sobre la transferencia. Algunos participantes recibieron un único problema para leer, y otros recibieron dos problemas análogos. El 45% de los participantes en la condición de dos análogos generaron espontáneamente la solución de convergencia, mientras solo el 21% de los participantes en la condición de

un solo problema lo hicieron. Es decir, la transferencia de conocimiento se produjo por una serie de procesos: en primer lugar, cuando las personas leyeron ambos problemas formaron una representación mental de dichos problemas en su memoria de trabajo (MT). El sistema cognitivo estableció correspondencias entre los elementos y acciones comunes a ambas situaciones (alineación y mapeo estructural). Como resultado, generaron un esquema común a ambos problemas y luego pudieron identificar un tercer problema como portador del mismo esquema (transfirieron). En los experimentos de seguimiento, Gick y Holyoak descubrieron que el acoplamiento de la actividad de comparación con una declaración verbal o un diagrama que destacaba los aspectos compartidos de las fuentes tenía un claro efecto positivo en la transferencia.

El aprendizaje analógico basado en la comparación de ejemplos, mostró efectividad en la transferencia, sobre todo en procesos guiados de comparación. Comparar dos situaciones en las cuales subyace un mismo principio, puede promover la abstracción del esquema y luego transferirse a nuevas situaciones (Catrambone & Holyoak, 1989; Faries & Reiser, 1988; Gentner et al., 2003; Goldstone & Son, 2005; Kurtz & Loewenstein, 2007; Minervino et al., 2017).

8 Aplicación a un caso de programación en Scratch

Volviendo al planteo en la sección 3, acerca de la dificultad de los estudiantes para trabajar en los niveles de abstracción superior "del problema" y "del algoritmo", se hace notar que este nivel se refiere a la representación del contexto del problema: las situaciones de la vida real cargadas con contenidos propios de cada dominio. Lo realmente difícil para los estudiantes es encontrar la estructura abstracta subyacente en cada dominio, porque se enfocan en los detalles del dominio. Precisamente, las investigaciones sobre razonamiento analógico en resolución de problemas han mostrado que la estructura abstracta se resalta al comparar situaciones de dominios distintos pero con estructura abstracta coincidente, de lo que inferimos que la técnica comparación de problemas podría también mejorar la captación de los aspectos estructurales en los problemas de programación, si se dedica un tiempo suficiente a analizar los problemas en su contexto, antes de pasar a la codificación.

¿Cuántos problemas y qué distancia debe mediar entre los dominios? Cuanto mayor es la distancia entre dominios (más disímiles los escenarios) mayor es la nitidez del esquema abstracto formado. Por lo que una buena práctica sería presentar problemas diversos con estructura similar. Por el contrario, presentar problemas en un único dominio (por ejemplo, el dominio de las animaciones, donde todas las acciones son desplazamientos o transformaciones de *sprites*), dificulta al estudiante el pasaje a dominios distantes con posterioridad.

Se tomará como ejemplo el problema de repetición simple, la estructura de repetición más básica en programación, que se presenta en la actividad "No me canso de saltar" disponible online (Factorovich & O'Connor, 2016, p. 30).

En la actividad mencionada, se proponen dos ejemplos para un mismo tipo de problema, ambos en el entorno *on line* de Scratch. En el primer ejemplo, un personaje (el gato) ejecuta una acción cada vez que el estudiante se lo pide mediante una instrucción. El docente que guía la actividad le pide al alumno que piense cómo resolvería la situación, si el gato, en lugar de saltar 2 o 3 veces, tuviera que saltar 23 veces. El docente lo guía hacia la solución del problema que consiste en utilizar una estructura de iteración llamada repetir, que en Scratch se representa mediante un bloque visual. Esa solución, la inclusión de una estructura generalizadora "Repetir 23 veces", en vez de escribir 23 veces la acción saltar. Éste sería un aprendizaje importante si, mediando un tiempo de separación entre dicha actividad y su oportunidad de aplicación, el estudiante lograra utilizar el mismo concepto en otro dominio. Por ejemplo, si se presentara a los alumnos un problema matemático, como el de generar la serie de Fibonacci con una iteración, existiría una gran diferencia entre los números y el escenario del gato, que haría improbable su aplicación. Los alumnos podrían resolver el problema por otra vía, no por analogía con el anterior. Sería altamente improbable que dedujeran la acción "repetir 81 veces la suma de los números anteriores" como consecuencia de haber utilizado previamente la acción del gato "repetir 23 veces saltar".

Se ha observado una dificultad en los estudiantes para pasar de un nivel de abstracción a otro, en especial del nivel del programa (nivel 2) al nivel de algoritmo (nivel 3), o del nivel del programa al nivel del problema (nivel 4) (Armoni, 2013; Hazzan, 2008; Perrenet, 2010). Por ejemplo, en este caso, los estudiantes que trabajan en Scratch observan que el personaje del gato salta varias veces, pueden comprender fácilmente que hay una acción 'saltar' que se ejecuta durante un tiempo determinado (nivel 1), que esa

ejecución se diferencia de la instrucción para saltar que escribe el programador (nivel 2). Luego ven la instrucción condensada en una estructura de repetición llamada REPETIR, y entienden que se trata de un programa porque ven la secuencia de estructuras visuales fácilmente comprensible, pero aparentemente, el nivel de abstracción es tan concreto (niveles 1 y 2) que no les permitiría una transferencia directa a otros problemas del mismo tipo, porque tendrían que haber comprendido el problema a un nivel superior, del algoritmo (nivel 3) o del problema (nivel 4) para poder transferir. Según Hazzan (2008), los estudiantes tenderán a bajar de nivel o a quedarse en un nivel bajo de concreción si no hay ayudas explícitas por parte del docente para ascender a un nivel de comprensión más abstracto. En este caso, podría esperarse que aunque los participantes comprendan el concepto de repetición en Scratch, no lo transfieran a la resolución de un problema matemático donde deben repetir una suma de números, a menos que hayan comprendido el sentido de la iteración en programación a nivel abstracto: la iteración se aplica cuando una acción se ejecuta un número tan elevado de veces que no es conveniente dar una instrucción para cada acción sino que debe utilizarse una instrucción generalizadora.

Desde el punto de vista de los modelos del pensamiento analógico, son dos situaciones estructuralmente análogas, pero semánticamente distantes, la transferencia de conocimiento de una a la otra no está garantizada. Los alumnos pueden aprender un concepto en Scratch y no volver a aplicarlo cuando se enfrenten a la misma estructura de problema en un lenguaje como C o Java.

Teniendo en cuenta que la mayoría de los alumnos usan la computadora sin guía docente, se beneficiarían con actividades de conceptualización en las que se realicen comparaciones entre problemas de dominios diversos (tan distantes como sea posible) antes de ser programadas en Scratch. Por ejemplo, los alumnos podrían comparar problemas utilizando parejas de dominios distantes, uno de los problemas podría pertenecer al dominio de las animaciones (e.g., repetir saltos, vueltas o sonidos), el otro problema podría pertenecer al dominio de las matemáticas (e.g., sumar series, acumular, ordenar), pero la estructura de las operaciones debe coincidir (e.g., repetición simple, repeticiones anidadas, árbol de decisiones, validaciones, ingreso de datos), de tal modo, que al comparar los problemas y sus soluciones los alumnos descubran la coincidencia estructural.

9 Conclusiones

Cuanto mayor es la carga de detalles superficiales en la información manipulada, mayor es la dificultad para percibir la estructura subyacente (lo común a todas las situaciones). Se sugiere que el alumno aprenda a comprender problemas del mundo real (de nivel elevado de abstracción), efectuando comparaciones entre problemas de estructura análoga, al tiempo que se vinculan con las estructuras algorítmicas correspondientes a sus soluciones, de modo tal que aprendan a categorizar situaciones al tiempo que aprenden a construir algoritmos. Caso contrario, si los alumnos comienzan jugando con algoritmos que exhiben una gran cantidad de información visual, sin una guía docente, se dificulta el pasaje a niveles elevados de abstracción.

Por la gran accesibilidad de los medios tecnológicos, cuando los alumnos están motivados para aprender a programar, lo hacen interactuando con lenguajes visuales mucho antes de iniciar una carrera de grado en ciencias de la computación. Si estos lenguajes facilitan la construcción de programas pero obstaculizan la transferencia, es oportuno diseñar estrategias pedagógicas para facilitar la transición entre las nociones de programación aprendidas en la escuela media y los conceptos básicos necesarios en la universidad. Según los investigadores hay déficits en la abstracción a nivel de problema. Las investigaciones sobre razonamiento analógico han utilizado con éxito la comparación de problemas en distintos campos disciplinares. Se sugiere evaluar la posibilidad de utilizar la comparación de problemas para promover una elaboración más abstracta de los mismos, y aumentar el tiempo de trabajo de los alumnos en este nivel antes de pasar al código.

Declaración de conflicto de intereses

Los autores declaran no tener conflicto de intereses con respecto a la investigación, autoría o publicación de este artículo.

Referencias

Armoni, M. (2013). On Teaching Abstraction in CS to Novices. *Journal of Computers in Mathematics and Science Teaching*, *32*(3), 265–284. https://www.learntechlib.org/p/41271

Armoni, M., & Ben-Ari, M. (2013). *Computer Science Concepts in Scratch*. Department of Science Teaching , Weizmann Institute of Science. https://stwww1.weizmann.ac.il/scratch/scratch_en/

Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From Scratch to "Real" Programming. *ACM Transactions on Computing Education*, *14*(4), 1–15. https://doi.org/10.1145/2677087

Barnett, S. M., & Ceci, S. J. (2002). When and where do we apply what we learn?: A taxonomy for far transfer. *Psychological Bulletin*, *128*(4), 612–637. https://doi.org/10.1037/0033-2909.128.4.612

Bell, T., Rosamond, F., & Casey, N. (2012). Computer Science Unplugged and Related Projects in Math and Computer Science Popularization. En H. L. Bodlaender, R. Downey, F. V. Fomin, & D. Marx (Eds.), *The Multivariate Algorithmic Revolution and Beyond. Lecture Notes in Computer Science, vol 7370* (pp. 398–456). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-30891-8_18

Burgoon, E. M., Henderson, M. D., & Markman, A. B. (2013). There Are Many Ways to See the Forest for the Trees: A Tour Guide for Abstraction. *Perspectives on Psychological Science*, *8*(5), 501–520. https://doi.org/10.1177/1745691613497964

Çakiroğlu, Ü., Sude, S., B., K., Sari, A., Yildiz, S., & Öztürk, M. (2018). Exploring perceived cognitive load in learning programming via Scratch. *Research in Learning Technology*, *26*. https://doi.org/10.25304/rlt.v26.1888

Catrambone, R., & Holyoak, K. J. (1989). Overcoming contextual limitations on problem-solving transfer. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *15*(6), 1147–1156. https://doi.org/10.1037/0278-7393.15.6.1147

Chi, M. T. H., Feltovich, P. J., & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science*, *5*(2), 121–152. http://www.sciencedirect.com/science/article/pii/S0364021381800298

Dahl, O.-J., Dijkstra, E. W., & Hoare, C. A. R. (1972). *Structured programming*. Academic Press Ltd.

Denning, P.J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., & Young, P. R. (1989). Computing as a discipline. *Computer*, *22*(2), 63–70. https://doi.org/10.1109/2.19833

Denning, Peter J. (1985). The Science of Computing: What is computer science? *American Scientist*, *73*(1), 16–19. http://www.jstor.org/stable/27853057

Denning, Peter J. (2003). Great Principles of Computing. *Communications of the ACM*, *46*(11), 15–20. https://doi.org/10.1145/948383.948400

Denning, Peter J. (2017). *Computational thinking in science*.

Factorovich, P., & O'Connor, F. S. (2016). *Cuaderno para el docente. Actividades para aprender a programar*. Fundación Sadosky. http://programar.gob.ar/descargas/manual-docente-descarga-web.pdf

Faries, J. M., & Reiser, B. J. (1988). *Access and Use of Previous Solutions in a Problem Solving Situation*. https://apps.dtic.mil/dtic/tr/fulltext/u2/a224717.pdf

Franklin, D., Hill, C., Dwyer, H. A., Hansen, A. K., Iveland, A., & Harlow, D. B. (2016). Initialization in Scratch. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16*, 217–222. https://doi.org/10.1145/2839509.2844569

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, *7*(2), 155–170. https://doi.org/https://doi.org/10.1016/S0364-0213(83)80009-3

Gentner, D. (1989). The mechanisms of analogical transfer. En S. Vosniadou & A. Ortony (Eds.), *Similarity and Analogical Reasoning* (pp. 199–242). Cambridge University Press.

Gentner, D., Loewenstein, J., & Thompson, L. (2003). Learning and transfer: A general role for analogical encoding. *Journal of Educational Psychology*, *95*(2), 393–408. https://doi.org/10.1037/0022-0663.95.2.393

Gentner, D., & Markman, A. B. (1997). Structure mapping in analogy and similarity. *American Psychologist*, *52*(1), 45–56. https://doi.org/10.1037/0003-066X.52.1.45

Gentner, D., Rattermann, M. J., & Forbus, K. D. (1993). The Roles of Similarity in Transfer: Separating Retrievability From Inferential Soundness. *Cognitive Psychology*, *25*(4), 524–575. https://doi.org/10.1006/cogp.1993.1013

Gick, M. L., & Holyoak, K. J. (1980). Analogical problem solving. *Cognitive Psychology*, *12*(3), 306–355. https://doi.org/10.1016/0010-0285(80)90013-4

Gick, M. L., & Holyoak, K. J. (1983). Schema induction and analogical transfer. *Cognitive Psychology*, *15*(1), 1–38. https://doi.org/10.1016/0010-0285(83)90002-6

Goldstone, R. L., & Son, J. Y. (2005). The Transfer of Scientific Principles Using Concrete and Idealized Simulations. *Journal of the Learning Sciences*, *14*(1), 69–110. https://doi.org/10.1207/s15327809jls1401_4

Harel, I., & Papert, S. (1991). *Constructionism.* Ablex Publishing.

Hazzan, O. (1999). Reducing Abstraction Level When Learning Abstract Algebra Concepts. *Educational Studies in Mathematics*, *40*(1), 71–90. https://doi.org/10.1023/A:1003780613628

Hazzan, O. (2003). How Students Attempt to Reduce Abstraction in the Learning of Mathematics and in the Learning of Computer Science. *Computer Science Education*, *13*(2), 95–122. https://doi.org/10.1076/csed.13.2.95.14202

Hazzan, O. (2008). Reflections on teaching abstraction and other soft ideas. *ACM SIGCSE Bulletin*, *40*(2), 40–43.

https://doi.org/10.1145/1383602.1383631
Hazzan, O., & Kramer, J. (2007). Abstraction in Computer Science & Software Engineering: A Pedagogical Perspective. *Frontier Journal*, *4*(1), 6–14.
Hazzan, O., & Kramer, J. (2016). Assessing abstraction skills. *Communications of the ACM*, *59*(12), 43–45. https://doi.org/10.1145/2926712
Hoc, J., Green, T., Samurçay, R., & Gilmore, D. (1990). Part 1: Theoretical and Methodological Issues. En J. M. Hoc (Ed.), *Psychology of Programming*. London: Academic.
Holyoak, K. J., & Thagard, P. (1989). Analogical Mapping by Constraint Satisfaction. *Cognitive Science*, *13*(3), 295–355. https://doi.org/10.1207/s15516709cog1303_1
Kafai, Y., & Resnick, M. (1996). *Constructionism in Practice: Designing, Thinking, and Learning in a Digital World* (Y. Kafai & M. Resnick (eds.)). Routledge.
Kurtz, K. J., & Loewenstein, J. (2007). Converging on a new role for analogy in problem solving and retrieval: when two problems are better than one. *Memory & Cognition*, *35*(2), 334–341. https://doi.org/10.3758/BF03193454
Malan, D., & Leitner, H. (2007). Scratch for Budding Computer Scientists. *SIGCSE 2007: 38th SIGCSE Technical Symposium on Computer Science Education*, *39*. https://doi.org/10.1145/1227310.1227388
Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by Choice: Urban Youth Learning Programming with Scratch. *Proceedings of the 39th SIGCSE technical symposium on Computer science education - SIGCSE '08*, 367–371. https://doi.org/10.1145/1352135.1352260
Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2011). Habits of programming in scratch. *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education - ITiCSE '11*, 168–172. https://doi.org/10.1145/1999747.1999796
Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (Moti). (2013). Learning computer science concepts with Scratch. *Computer Science Education*, *23*(3). https://doi.org/10.1080/08993408.2013.832022
Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (Moti). (2010). Learning computer science concepts with Scratch. *Proceedings of the Sixth international workshop on Computing education research - ICER '10*, 69–76. https://doi.org/10.1145/1839594.1839607
Minervino, R. A., Olguín, V., & Trench, M. (2017). Promoting interdomain analogical transfer: When creating a problem helps to solve a problem. *Memory & Cognition*, *45*(2), 221–232. https://doi.org/10.3758/s13421-016-0655-2
Nassi, I., & Shneiderman, B. (1973). Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, *8*(8), 12–26. https://doi.org/10.1145/953349.953350
Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books.
Papert, S., & Harel, I. (1991). Situating Constructionism. En I. Harel & S. Papert (Eds.), *Constructionism* (p. 518). Ablex Publishing Corporation.
Perkins, D. N., & Salomon, G. (1988). Teaching for Transfer. *Educational Leadership*, *46*(1), 22–32. https://eric.ed.gov/?id=EJ376242
Perrenet, J. C. (2010). Levels of thinking in computer science: Development in bachelor students' conceptualization of algorithm. *Education and Information Technologies*, *15*(2), 87–107. https://doi.org/10.1007/s10639-009-9098-8
Perrenet, J., Groote, J. F., & Kaasenbrood, E. (2005). Exploring students' understanding of the concept of algorithm. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '05*, 64–68. https://doi.org/10.1145/1067445.1067467
Perrenet, J., & Kaasenbrood, E. (2006). Levels of abstraction in students' understanding of the concept of algorithm. *ACM SIGCSE Bulletin*, *38*(3), 270–274. https://doi.org/10.1145/1140123.1140196
Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: programming for all. *Communications of the ACM*, *52*(11), 60–67. https://doi.org/10.1145/1592761.1592779
Resnick, M., Myers, B., Nakakoji, K., Shneiderman, B., Pausch, R., & Eisenberg, M. (2005). Design Principles for Tools to Support Creative Thinking. *Report of Workshop on Creativity Support Tools*, *20*, 25–36.
Seehorn, D. (2011). K-12 Estándares para las Ciencias de la Computación. En *Asociación de Maestros de Ciencias de la Computación (CSTA)*.
Shneiderman, B., Mayer, R., McKay, D., & Heller, P. (1977). Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, *20*(6), 373–381. https://doi.org/10.1145/359605.359610
Statter, D., & Armoni, M. (2016). Teaching Abstract Thinking in Introduction to Computer Science for 7th Graders. *Proceedings of the 11th Workshop in Primary and Secondary Computing Education on ZZZ - WiPSCE '16*, 80–83. https://doi.org/10.1145/2978249.2978261
Trench, M., & Minervino, R. A. (2017). Cracking the Problem of Inert Knowledge. *Psychology of Learning and Motivation*, *66*, 1–41. https://doi.org/10.1016/bs.plm.2016.11.001
Turkle, S., & Papert, S. (1990). Epistemological Pluralism: Styles and Voices within the Computer Culture. *Signs: Journal of Women in Culture and Society*, *16*(1), 128–157. https://doi.org/10.1086/494648
Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35. https://doi.org/10.1145/1118178.1118215

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *366*(1881), 3717–3725. https://doi.org/10.1098/rsta.2008.0118

Wirth, N. (1976). *Algorithms and Data Structures*. Pearson Education.