From One Language to the Next: Applications of Analogical Transfer for Programming Education

YVONNE KAO and BRYAN MATLEN, WestEd
DAVID WEINTROP, University of Maryland

The 1980s and 1990s saw a robust connection between computer science education and cognitive psychology as researchers worked to understand how students learn to program. More recently, academic disciplines such as science and engineering have begun drawing on cognitive psychology research and theories of learning to create instructional materials and teacher professional development materials based on theories of learning, to some success. In this paper, we follow a similar approach by highlighting common areas of interest between computer science education and cognitive psychology–specifically theories of analogical transfer–and discuss how cross-pollination of theoretical constructs between disciplines can support research on the teaching and learning of multiple programming languages. We will also discuss areas where computing education research can adapt the existing theories from cognitive psychology to develop domain-specific theories of knowledge transfer in computing and feed back into cognitive psychology research to inform larger debates about the nature of cognition and learning.

CCS Concepts: • **Social and professional topics → Computing education;**

Additional Key Words and Phrases: Cognitive psychology, transfer, computer science education, programming

ACM Reference format:
Yvonne Kao, Bryan Matlen, and David Weintrop. 2022. From One Language to the Next: Applications of Analogical Transfer for Programming Education. *ACM Trans. Comput. Educ.* 22, 4, Article 42 (November 2022), 21 pages.
https://doi.org/10.1145/3487051

1 INTRODUCTION

An increasing number of students are now taking computing courses in multiple grade bands and are likely to be taught with multiple programming languages. Many students first learn a block-based language, like Scratch, as their introduction to computer science and then learn conventional text-based programming languages as they advance. How can we help students build on prior conceptual knowledge as they progress through a multi-lingual course sequence? Understanding and supporting students' transitions to new programming languages is an area of active research in computer science education [126]. Effective pedagogy should help students successfully transfer

Authors' addresses: Y. Kao (corresponding author) and B. Matlen, WestEd, 730 Harrison St, San Francisco, California 94107; emails: {ykao, bmatlen}@wested.org; D. Weintrop, University of Maryland, 2226H Benjamin Building, 3942 Campus Drive, College Park, MD 20742; email: weintrop@umd.edu.

1946-6226/2022/11-ART42 $15.00
https://doi.org/10.1145/3487051

knowledge from one language to another to deepen their knowledge rather than treating each programming language as unrelated. We have two major goals for this article. First, we aim at highlighting the substantial synergy between theoretical work on transfer from cognitive psychology and recent work on transfer in computing education research and present an integrated overview of these two literatures. Second, we suggest ways to build upon this prior work to support the development of classroom interventions, multi-lingual assessments, and CS-specific theories of transfer.

The last two and a half decades have seen the emergence of cognitive psychology as an important driver of educational research in mathematics and the sciences [45]. There have been a large number of review articles synthesizing basic research in cognitive psychology and generating recommendations for applying findings to practice (e.g., [27, 84]). In addition, numerous projects have used theories developed from basic cognitive psychology research to engineer new educational interventions [19, 23, 73]. Cognitive psychology offers a key bridge between the basic science of how the brain learns new information and implications for designing instruction [15]. Other authors have already reviewed the historical connections between cognitive psychology and computing education and introduced key areas of cognitive psychology research to a computing education research audience [68, 97]. Readers will note that some of the same cognitive psychology studies discussed by Robins and colleagues [97] are also summarized here. We extend this work by presenting the cognitive psychology literature simultaneously with a discussion of the computing education literature on transfer in order to facilitate comparison and integration. We conclude the article with a discussion of developing CS-specific theories of transfer and how they can support the creation of instructional materials and multi-lingual assessments.

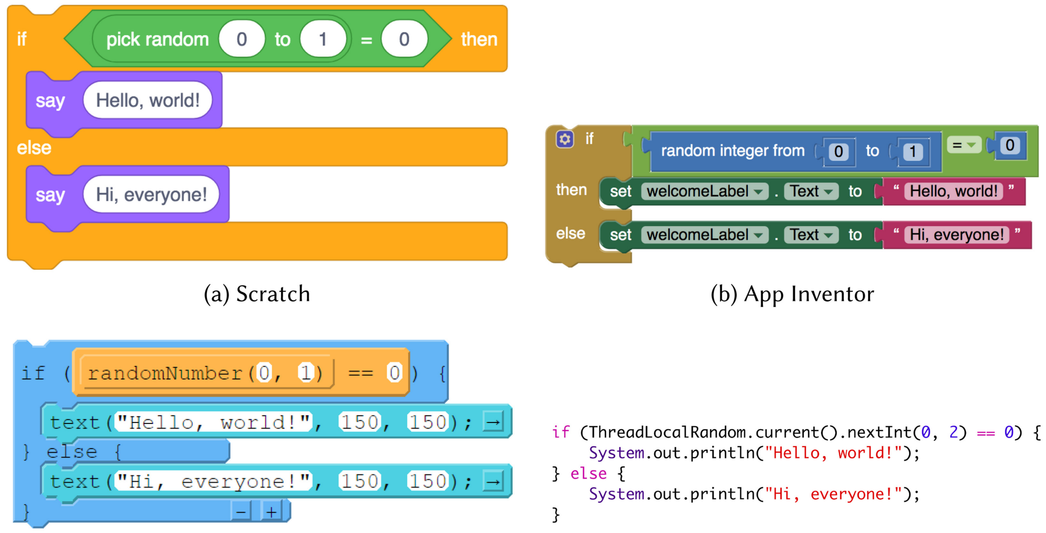
2 THEORIES OF TRANSFER

Analogical transfer–the act of applying knowledge from one context to another based on shared relations–is arguably one of the central goals of education. For instance, if a student accurately solves multiplication problems on a homework assignment, we expect this skill to transfer to solving multiplication problems on a standardized test, calculating costs while shopping, scaling recipes while cooking, and so on. In other words, a central goal of instruction is to support students in acquiring knowledge that they can extend to a range of situations within and beyond the classroom. However, the analogical transfer is notoriously difficult to achieve. Furthermore, the transfer can be counterproductive if erroneous inferences are made (referred to as *negative transfer*).

A prominent theory of analogy is Structure Mapping [34, 35], which explains that knowledge transfer involves aligning corresponding objects and their parts based on a shared relational structure. To demonstrate how Structure Mapping works, we present a simple "Hello, world"-typescript that randomly decides which of two greetings to use. This script is implemented in Scratch (Figure 1(a)), App Inventor (Figure 1(b)), the blocks-based Javascript used by Code.org's Game Lab (Figure 1(c)), and Java (Figure 1(d)). The scripts are structurally equivalent but differ substantially in their surface features. There are obvious differences between languages in how the code looks (i.e., whether blocks are used and what colors and shapes are used) and how functions are named. To transfer knowledge from one of these programming languages to another, students must be able to identify the underlying structure of the code and then align the equivalent segments of code across the two languages.

2.1 The Problem of Transfer

In the analogical transfer literature, there is an important distinction between surface similarity and structural similarity. As an illustration, "My butcher is a surgeon" has a completely different


if pick random 0 to 1 = 0 then
say Hello, world!
else
say Hi, everyone!
if random integer from 0 to 1 = 0
then set welcomeLabel . Text to " Hello, world! "
else set welcomeLabel . Text to " Hi, everyone! "
if (randomNumber(0, 1) == 0) {
text("Hello, world!", 150, 150);
} else {
text("Hi, everyone!", 150, 150);
}

```java
if (ThreadLocalRandom.current().nextInt(0, 2) == 0) {
    System.out.println("Hello, world!");
} else {
    System.out.println("Hi, everyone!");
}
```

(a) Scratch (b) App Inventor (c) Javascript (blocks) (d) Java

Fig. 1. An if-else statement is implemented in three block-based programming languages (a–c) and Java (d).

meaning than "My surgeon is a butcher", despite using the same words. The ability to understand these situations in terms of their structural relations forms the basis for our ability to engage in analogical transfer [40]. Despite our incredible ability to understand new situations by mapping the underlying relations, the spontaneous analogical transfer may be based on superficial relations or fail to occur. In a classic study of spontaneous transfer, Gick and Holyoak provided undergraduates with a short scenario [39, p. 3]:

> A general wish to capture a fortress located in the center of a country. There are many roads radiating outward from the fortress. All have been mined so that while small groups of men can pass over the roads safely, any large force will detonate the mines. A full-scale direct attack is therefore impossible. The general's solution is to divide his army into small groups, send each group to the head of a different road, and have the groups converge simultaneously on the fortress.

After studying this scenario, students are presented with another problem that, unbeknownst to them, can be solved in an analogous way (i.e., by using partitioning and convergence).

> Suppose you are a doctor faced with a patient who has a malignant tumor in his stomach. It is impossible to operate on the patient, but unless the tumor is destroyed the patient will die. There is a kind of ray that can be used to destroy the tumor. If the rays reach the tumor all at once at a sufficiently high intensity, the tumor will be destroyed. Unfortunately, at this intensity, the healthy tissue that the rays pass through on the way to the tumor will also be destroyed. At lower intensities, the rays are harmless to healthy tissue, but they will not affect the tumor either. What type of procedure might be used to destroy the tumor with the rays, and at the same time avoid destroying the healthy tissue? [39, p. 3]

The solution is to set up several, lower-powered rays that encircle the tumor, thereby allowing the rays to enter the healthy tissue at separate points, leaving the healthy tissue unharmed, while converging the full power of the ray at the site of the tumor (analogous to dividing up an army encircled around a fortress and attacking simultaneously).

Despite the similarity in the structures of the two problems, Gick and Holyoak [39] found that transfer to the ray-tumor problem was surprisingly low even though the problems were presented in close succession. Only 20–40% of participants across several experiments transferred knowledge to the new problem even when the researchers implemented strategies aimed at improving transfer, such as providing diagrams or an abstract description of the principle. Though this example is from research conducted decades ago, comparatively low rates of spontaneous analogical transfer have been documented in many domains and educational contexts [14, 32].

2.2 Why Transfer is Hard: Surface vs. Structural Similarity

What underlies the difficulty in analogical transfer? One key source of students' difficulty is failing to recognize that two problems share analogous structures [91]. This is especially true for domain novices that lack the expertise and experience to know which features are most relevant. Novices often focus on the surface aspects of problems (i.e., the perceptual attributes). Surface features are the most readily accessed by our perceptual system and are alluring in a context where the stimuli are novel and cognitive resources are limited. However, this surface-level focus often comes at the expense of attending to the underlying relational structure.

To use an example from physics education, Chi, Feltovich, and Glaser [17] asked novice physics students and physics experts to sort problems based on their similarity to one another. Novices sorted problems based on the surface features, such as sorting all problems containing inclined planes into one group and sorting all the problems containing pulleys into another group. Experts, on the other hand, sorted problems based on their underlying structural principles, such as grouping all problems involving conservation of energy. In another striking example, Perkins [87] describes physics students who, after learning in class how to calculate the time it would take for a ball to fall from the top of a tower to the ground, exhibited confusion on a test problem that required them to calculate the time for an object to fall from the top to the bottom of a well. The students lamented that they had not been given instruction on problems with wells.

In other words, whereas novices overlook the abstract relationships between problems of differing surface features, experts' knowledge and experience allow them to readily perceive or "see" the underlying similarity in structure, despite the lack of similarity in the literal presentation. This expert-novice difference in categorization and problem-solving has been demonstrated in a great variety of domains (e.g., [16, 99]), including in programming (e.g., [127]).

Paying too much attention to surface features has the potential to distract from the key relationships and results in over contextualization of knowledge, which inhibits transfer. This issue is highlighted in Barrett and Ceci's [7] taxonomy of transfer distance. Their framework describes transfer in terms of two factors: (1) content, or *what* is transferred; and (2) context, or *where and when* the content is transferred. Content that is practiced in a limited number of contexts is less likely to transfer. For instance, poor transfer performance has resulted from studying repeatedly in the same physical context [11] and repeatedly studying the same type of problem [98]. Furthermore, though concrete or grounded representations provide familiarity that can aid in initial learning, such representations studied in isolation are often insufficient to promote later transfer [51].

2.3 A Review of Computing-specific Theories

The 1980s and 1990s saw a robust connection between CS education and cognitive psychology as researchers worked to understand how students learn programming [2, 12, 109]. Since then, the fields have largely worked in parallel. Computing education researchers have independently developed theoretical models of transfer in programming education. In this section, we review three computing-specific theories that aim at predicting when the transfer will occur.

2.3.1 Theory of ACT-R and Programming Tutors. In the early 1990s, Anderson and colleagues conducted a series of studies to investigate how knowledge of one programming language transfers to another. This work is based on ACT-R, which is a cognitive architecture, i.e., a general theory of human cognition instantiated as a computational model [2, 95]. ACT-R contains a series of modules and buffers that enable the model to set and update goals, perceive stimuli and generate responses, and store and update information in memory. The core of the architecture is the production system, which selects and applies production rules. Production rules are *condition-action*, or *if-then*, pairs. The production system determines which rule applies in any given circumstance, performs the action, and then updates the system state. ACT-R learns new production rules through a process of analogy. Existing production rules are strengthened through repeated use [3].

Anderson and colleagues created an intelligent programming tutor to teach Lisp, Pascal, and Prolog that made use of the ACT-R theory. The tutor was organized around a model of an ideal student: a set of production rules that enables students to program in each language effectively. When human students interacted with the intelligent programming tutor, the tutor then used a *model-tracing* paradigm to determine how to respond to student actions. That is the tutor attempted to match, in real-time, the student's actual actions with a sequence of productions based on the ideal student model. When the student's behavior could not be matched with the ideal model, the tutor would provide corrective feedback.

The ideal student models for programming in Lisp, Pascal, and Prolog were quite different, reflecting the fact that these languages are quite different in character. The few production rules that were common across languages related to more conceptual aspects of programming, such as evaluating conditional statements. As there were no common production rules for writing code due to the differences in syntax and style for the three languages, researchers predicted that there may be little knowledge transfer from one language to another, and any transfer should relate to underlying conceptual knowledge. Anderson and colleagues then tested these predictions in laboratory and classroom studies in which students used the programming tutor to learn Lisp, Pascal, and Prolog. Though students were not more accurate when programming in their second language compared to their first, they learned the second language slightly more quickly. Another set of studies found that conceptual understanding of algorithms transferred readily between languages, but not procedural coding skills [95]. In explaining these results, Wu and Anderson [129] identified three levels of similarity that can facilitate transfer: syntactic, algorithmic, and problem. A major strength of the research with ACT-R is that the creation of the ideal student models necessitated a very careful analysis of the knowledge and skills needed for programming, enabling very precise predictions on what knowledge and skills might transfer across languages.

2.3.2 Mindshift Learning Theory. The Mindshift Learning Theory offers another theoretical framework for explaining the transfer, or lack thereof when learning new programming languages [5]. Rooted in the field of information systems, Mindshift Learning Theory (Figure 2) is based on Louis and Sutton's explanatory framework for shifting cognitive processes [66]. The theory posits that the perceived level of novelty affects the ease of learning new concepts. The model proposed three categories of transfer related to learning a new language (or paradigm): (1) carryover concepts that have a similar meaning from the known context, (2) changed concepts that are similar to the known context but have a different meaning, and (3) novel concepts that are new to the learner. Armstrong and colleagues developed and tested this model through a series of empirical studies of programmers familiar with procedural programming as they transitioned to programming with

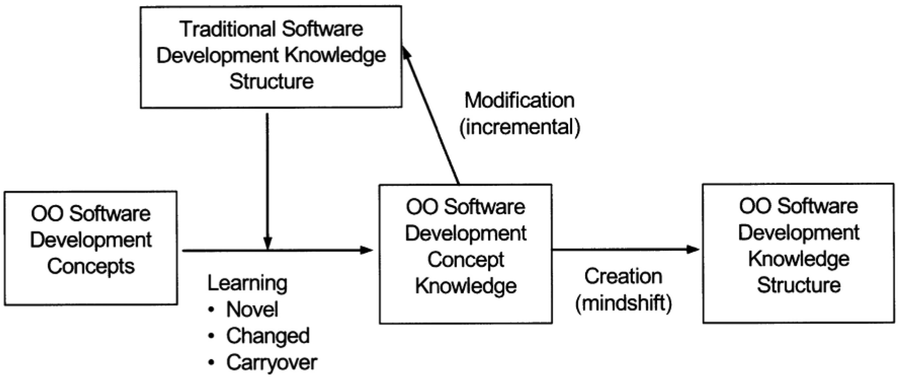

Traditional Software Development Knowledge Structure
Modification (incremental)
OO Software Development Concepts
Learning
• Novel
• Changed
• Carryover
OO Software Development Concept Knowledge
Creation (mindshift)
OO Software Development Knowledge Structure

Fig. 2. Mindshift learning theory, as presented by Armstrong and Hardgrave [5, p. 459].

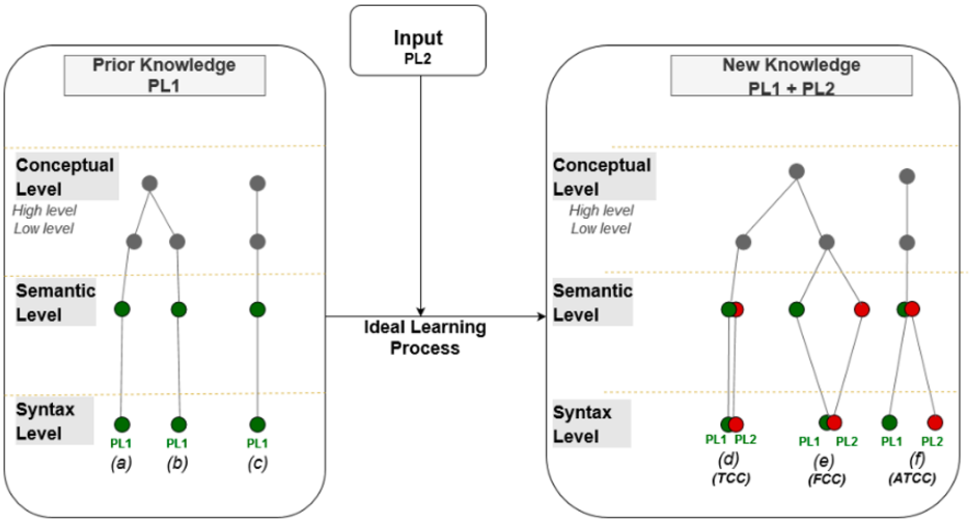


Fig. 3. Tshukudu and Cutts' model of programming language transfer [116, p. 230].

object-oriented languages [5, 6, 78]. They found the high levels of transfer for carryover and novel concepts and lower levels of transfer for changed concepts.

2.3.3 Tshukudu and Cutts. Tshukudu and Cutts' model [116], builds on program comprehension models from computing education research [85, 105, 106, 119] and semantic transfer from research on natural language learning [49, 50]. Their model of transfer involves knowledge of a programming language at three levels: syntactic, semantic, and conceptual. When applying knowledge from a prior programming language to the new programming language, Tshukudu and Cutts propose three potential outcomes (shown in Figure 3): (1) the syntax and semantics can align across languages, called a **true carryover construct** (**TCC**); (2) the semantics may differ but the syntax is similar, called a **false carryover construct** (**FCC**); or (3) the semantics may be shared but the syntax differs, called an **abstract true carryover** (**ATC**). In validating this model, Tshukudu and Cutts ran a series of studies of learners moving from one language to another and found the most transfer occurred with TCCs. Transfer was more difficult with both FCCs and ATC [114, 116].

2.4 Synthesizing the Theoretical Work on Transfer

The theoretical work on transfer converges in several ways. First, it is clear that similarity promotes transfer while dissimilarity impedes transfer. The CS-specific work generally identified several levels of transfer that must be considered when teaching and learning programming languages: syntax, concept (i.e., looping), and algorithm or paradigm.

There are a few predictions that emerge from synthesizing the theoretical work on transfer in both cognitive psychology and computing education. First, transfer of knowledge from one programming language to another cannot be assumed. In particular, raw coding skill in one language likely does not transfer to a new language that uses a different paradigm. For example, students' experience with drag-and-drop programming in Scratch will have limited transfer to text-based programming in an **integrated development environment** (**IDE**). Second, the transfer comes more easily when two programming languages are similar both in their surface features (i.e., syntax) and their semantic structure, particularly for novices (i.e., moving from Scratch to Snap! or Javascript to Java). However, surface similarity can also induce negative transfer and impede learning when the semantic structure is different. Third, experienced programmers are more able to recognize conceptual similarities between languages and programming paradigms compared to novices, particularly if the surface features are different.

3 EMPIRICAL STUDIES OF TRANSFER WHEN LEARNING TO PROGRAM

Having reviewed the theoretical literature on transfer, we now shift focus to empirical studies of transfer when learning to program. The question of transfer has long been a focus in computer science education research (e.g., [109]). Within this literature, the specific question of transfer between programming languages has a similarly long history given the centrality of programming in the field of computer science (e.g., [102, 129]. Early work on this question sought to understand expert/novice learning differences by focusing on how programmers who were familiar with one professional language went about learning a second language (e.g., [104]). There is a renewed focus on the question of transfer between programming languages due to the emergence of introductory languages and programming environments designed for novices that are distinct from professional programming languages (e.g., block-based programming) and the rise of CS instruction for younger learners (e.g., [42, 126]). We begin this section with a review of recent investigations of transfer from block-based to text-based languages before looking at research investigating transfer between two text-based programming languages.

3.1 Transfer from Block-based to Text-based Programming

Driven by the success of platforms such as Scratch and libraries like Blockly, block-based programming has increasingly become the way that novices are introduced to the practice of programming and field of computer science more broadly [10, 93, 121]. This can be seen in the rapidly growing ecosystem of block-based environments and the growing number of computer science curricula designed for them [26, 65, 80]. Research on the use of block-based environments in introductory contexts has found that these tools do support learning on their own [31, 53, 74] as well as in comparison to text-based languages [64, 90, 100, 123, 125]. In reviewing work focused on the transition from block-based to text-based programming, a number of studies have been conducted across a series of block-based tools.

Given Scratch's prominence in the block-based programming space, it is not surprising that it has been used as a source block-based language in transfer research. For example, Armoni and colleagues followed a group of students who had taken Scratch programming courses in prior years as they moved on to a high school programming course taught in a text-based programming language (either Java or C#) [4]. By looking at how students with prior Scratch experience performed relative to their classmates who had no prior block-based experience, they sought to make claims about the transfer of programming knowledge from Scratch to text-based languages. The authors found relatively little quantitative difference in performance on assessments between those who had prior Scratch experience and those who did not, but did find differences in their qualitative analysis related to motivation and self-efficacy. Additionally, the authors identified programming patterns from Scratch present in text-based programs of students who had prior Scratch experience, suggesting some transfer did occur [4]. A similar approach was used with younger learners comparing students with prior Scratch experience to those without across two different schools, finding similar positive results for those with the prior Scratch experience [41]. Another example is the work of Grover and colleagues [43], who used preparation for future learning approach [14] to help scaffold learners in moving from Scratch to text-based programming languages and found significant positive gains for students on text-based programming questions.

Whereas Scratch was initially focused on learning in informal contexts, the Alice programming environment has had a more explicit focus on classroom learning. Textbooks have been written for Alice [20, 21]. This has resulted in numerous studies of students transitioning from Alice to text-based languages with varying levels of success. Studies following undergraduate students moving from an introductory course taught in Alice to a follow-on programming course taught in a text-based language have reported students not transferring the knowledge gained in Alice to the subsequent language [83] or documented struggles in moving from Alice to another language within a single course [89]. Other research found students performed better when learning pseudocode prior to a text-based language compared to Alice [33]. At the same time, other studies have reported students self-reporting that Alice helped them in their subsequent courses [18] and have documented successful transfer from Alice to text-based languages. For example, Dann and colleagues report evidence of positive transfer from Alice to Java when the transition was accompanied with pedagogical strategies to help learners make the transition[22]. We will return to this work later when we discuss strategies for supporting transfer (Section 4).

A third programming environment used for studying transfer from block-based to text-based programming is Pencil Code [9]. One interesting feature of Pencil Code is that it supports both block-based and text-based authoring, providing two different interfaces for the same underlying programming language, thus, it becomes possible to set up comparative studies where students use only the block-based or text-based versions of the environment. A strength of this approach is that it controls for language and environmental factors–the runtime environment and underlying programming language remain the same and only the modality (blocks vs. text) changes. This is the study design used by Weintrop and Wilensky, who had one set of students go through an introductory curriculum in a blocks-only version of Pencil code and the second group of students uses a text-only version of Pencil Code before both groups transitioned to Java [126]. Students in the block-based condition scored higher on a programming assessment after the introductory portion of the study [125], but there was no difference in performance after transitioning to text-based programming [126].

3.2 Transfer from One Text-based Language to Another

While the question of transfer from block-based to text-based languages is a relatively recent area of study, there is a long history of research on transfer between text-based languages. This work largely falls into two categories. The first is focused on studying transfer between languages with largely similar semantics but differing syntax (e.g., moving from Python to Java). The second looks at the transfer between differing programming paradigms. The term "paradigm" is meant to classify similar groups of languages based on common behaviors and features, such as imperative, procedural, object-oriented, functional, and logic programming, so transfer research across paradigms may follow students as they move from procedural to object-oriented programming.

Early work on transfer between programming languages by Scholtz and colleagues found that learning a second programming language was easier than learning a first language but that interpretation of the second language was largely shaped by knowledge of the initial language [102, 104]. There are also expert-novice differences in the way people approached solving problems in the new language. Researchers found novices tend to use bottom-up approaches to solving problems in new languages [128], while experts tend to use a top-down approach, revising their plans as their familiarity with the new language grows [103]. Other researchers found a failure to productively transfer knowledge to a second language. For example, Walker and Schach identify instances where learners attempted to use knowledge of their first language (Pascal) to write programs in a second language (Ada), often attempting to use Pascal (or Pascal-similar) constructs unsuccessfully [120]. In a more recent study comparing Java to Scheme, the researchers concluded: "that upper-level students do not readily transfer knowledge gained in one language to another, even when that transfer is raised during lectures" [29, p. 128].

Researchers have also investigated the question of transfer between programming languages when moving from one programming paradigm to another (e.g., [6, 78, 101]). Unlike differences in languages within the same paradigm (i.e., transitioning from C++ to Java), a shift in paradigm is perceived as more significant given the role paradigms play in shaping the approach to and design of programs (i.e., transitioning from Lisp to Java) [28, 57, 86]. A number of studies have focused on experienced programmers learning object-oriented programming, finding that prior experience with non-object-oriented language was a barrier to learning to program in the new paradigm [78, 79, 101].

3.3 Summary of Empirical Research on Transfer When Learning to Program

To date, the literature on learning a second programming language has developed a rather muddy picture of knowledge transfer and it is unclear how well the predictions from Section 2.4 hold up under empirical tests. There are documented instances of positive transfer (e.g., [22, 41, 43, 76]) as well as a lack of transfer (e.g., [83, 89, 108, 126]). These studies have collectively examined a wide array of transfer situations, with participants in different age groups and varying degrees of prior programming proficiency, using a large number of different programming language pairs, with varying degrees of support for students in transferring knowledge. It is not clear if the range of different findings is an artifact of this variation. A CS-specific theory of transfer could improve our understanding of the pattern of results in these empirical studies by more precisely specifying what content we would expect to transfer in what contexts, enabling the design of experiments to empirically test these hypotheses. We will return to this idea in the discussion section.

4 STRATEGIES THAT IMPROVE TRANSFER DURING LEARNING

Theories of analogical transfer posit that students who understand the conceptual structure of problems will be more likely to apply that structure to novel situations and contexts [24, 35]. How does one become expert-like? How do we learn to "see" the underlying relationships in domains, such that we can recognize those relationships in new stimuli and settings? What instructional strategies will promote the understanding of conceptual structure? In this section, we review research-based strategies for highlighting conceptual structure and helping students understand what concepts transfer (see Table 1). First, we provide an overview of strategies to promote comparison and identification of the key relationships and highlight studies that have used these strategies in computing education. We conclude with recommendations for practitioners on how to develop broader systems that will help students successfully navigate multi-lingual CS pathways.

Table 1. Strategies to Promote Comparison and Improve Analogical Transfer

Category	Specific pedagogical strategy
General strategies for using comparison	Present two or more problems simultaneously. Structure the comparison process by: –prompting students to identify what's similar and different –providing explicit feedback to support accurate relational inferences
	Compare structurally similar problems to promote generalization and structurally dissimilar problems to promote discrimination
Using perceptual cues	Use progressive alignment by gradually fading the perceptual similarity of compared problems as the student acquires expertise
	Highlight relationally corresponding parts by: –Gesturing between the corresponding parts –Matching the colors of the corresponding parts –Spatially arranging compared problems so that their corresponding parts are most obvious (i.e., in direct alignment)

4.1 Using Comparison to Improve Transfer

Prompting students to compare contexts can draw attention to the common relationships. For example, in the ray-tumor problem described in Section 2.1, typical hints and scaffolds (e.g., showing a diagram of the convergence schema) failed to improve students' knowledge transfer. However, providing two problems with the same schema and prompting students to compare how they are similar led to increased success on the transfer problem. Comparison is thought to engage students in structure mapping, helping them notice commonalities in the relational structure as well as important differences that are connected to the structure [36, 59]. A meta-analysis of 57 experiments assessing comparison-based learning vs other instructional strategies (e.g., sequential presentation, traditional instruction) concluded that comparison leads to better learning outcomes at a medium effect size ($d = 0.50$) [1].

It is important to structure the comparison process, as guided instruction is more effective than pure discovery in promoting learning and transfer [47, 70]. Perkins and Solomon [88] developed "bridging" and "hugging" as strategies to encourage comparison and highlight the connections between different contexts. "Bridging" is a strategy where teachers "build a bridge" from the initial context to the next context, explicitly linking the two as a means to help learners build connections while "hugging" highlights the similarity between two contexts, making it easier for transfer to occur. These strategies have been successfully employed to support programming language transfer between a number of languages, including from Alice to Java [22], Alice to Python [112], and MakeCode to Python [76]. Without proper scaffolding, the comparison may be ineffective, especially for novices in a domain [47, 96]. In the next sections, we discuss effective ways of structuring comparison to promote transfer as predicted by theory and provide examples of these pedagogical techniques from empirical computing education research.

4.1.1 *Present Two or More Problems Simultaneously.*

Comparison is taxing on students' attention and memory. Keeping the exemplars available visually allows students to compare problems simultaneously, lessening the demands on memory and facilitating the Structure Mapping process. For example, presenting two worked examples of math problems simultaneously results in better understanding than presenting the same problems sequentially (e.g., [71, 96]).

Grover [42] and Dorling and White [25] used this strategy in their computing education research. Grover's computing curriculum shows learners analogous representations of the same program, often shown in language-agnostic pseudocode, block-based code, and text-based code. Presenting these forms side by side is intended to help learners "draw analogies between different formalisms to foster deep and abstract understanding of fundamental concepts and structures of algorithms" [42, p. 260]. Dorling and White [25] used side-by-side comparison as a pedagogical approach to help students identify differences and similarities between programming languages as a means to support productive transfer. Specifically, they designed a sequence of programming activities that moved learners from unplugged programming activities to block-based activities with Scratch, to text-based activities with Python. All three programming activities ask the students to solve the same type of problem (drawing geometrical shapes). By holding constant the programming challenges, and thus the conceptual and algorithmic parts of the programming task, students can focus on differences in the language syntax and programming environment. This approach of shifting languages while holding the programming task relatively constant led the authors to conclude "the practice of using graphical languages in conjunction with, (in effect using a graphical tool as a form of pseudo coding), not in place of, text-based programming languages, can improve the confidence, independence, and resilience of pupils when learning to program using a text-based language" [25, p. 196].

4.1.2 Compare Structurally Similar Problems to Promote Generalization and Structurally Dissimilar Ones to Promote Discrimination. The type of comparison can affect what is learned. Expert knowledge is characterized not only by the ability to see structural similarities but also by the ability to differentiate between structural dissimilarities, particularly instances where surface similarity is high. As described above, analogical comparisons are useful for supporting generalization to instances with a common structure. However, this could also lead to overgeneralization in cases where the structure does not apply. Such negative transfer is common when there are problems that share surface similarities, but that have different underlying structures. In these cases, it is helpful to provide contrasting examples and prompt students to point out how they are different. It is particularly useful if the cases are "near misses", varying minimally except for critical structural aspects that differentiate them [36, 75].

Tshukudu and Jensen [117] used this technique and found that explicit instruction was particularly useful at addressing errors related to FCCs, where semantics differ but the syntax is similar. Specifically, the researchers asked students to take two short tests, one in Java and one in Python. The researchers then analyzed student responses and identified errors that could be attributed to FCCs. The researchers then spent 25 minutes of class time reviewing the test results, specifically focusing on these errors and pointing out differences and similarities between the two languages. In a follow-up test after the intervention, students made significantly fewer mistakes related to FCCs.

4.2 Using Perceptual Cues to Improve Transfer

Perceptual features can be irrelevant and distracting to transfer, whereas attending to the relational structure is key for achieving expert-like reasoning. This may lead to the conclusion that perception is irrelevant to the learning of complex domains. However, there is good evidence that perceptual and higher-level reasoning processes closely interact during the course of learning. Some have even argued that expert knowledge is inherently perceptual (e.g., [8, 54]).

As an illustration, most people with basic algebraic knowledge would claim fluency with the order of operations. Despite this, spacing equations in ways that are consistent (e.g., $5 + 2{\times}7$) or inconsistent (e.g., $5{+}2 \times 7$) with the order of operations has been shown to facilitate and impede the ability to reason on such problems [63]. In addition, when asked to generate equations from a word problem, people proficient in algebra often space equations in ways consistent with the order of operations [62]. These findings suggest that conceptual representations may have a basis in perception even in what appear to be relatively abstract domains.

The notion that perceptual processes interact with learning also has implications for how to best support learning to promote transfer. Earlier, our discussion focused on how novices gravitate towards perceptual features, and that this can lead to the over contextualization of knowledge such that it impedes transfer. At the same time, similarity in perceptual features can be used as a scaffold for understanding abstract concepts and attending to critical relationships.

4.2.1 Use Progressive Alignment. Novices' attention to surface features of problems can be used to facilitate deeper processing. Kotovsky and Gentner [56] found that first presenting children with close analogical comparisons (those that share both surface and structural similarities) helped them to identify structural patterns in more difficult far analogical comparisons (those that share structural but not surface similarities) presented later, whereas presenting only far comparisons resulted in poor performance throughout. Analogies with common surface features can be used as an initial hook to support students in engaging in deeper, structural comparison, later on, a process coined "progressive alignment" [38, 56].

4.2.2 Highlight Relationally Corresponding Parts. Using surface cues to highlight corresponding parts is particularly effective when problems are presented simultaneously [48]. Researchers have found a variety of effective strategies for doing this, including using gestures or similar colors to indicate corresponding parts [72, 94] or arranging examples such that the corresponding parts are directly aligned in space (the math worked examples placed side by side, chemical formulas placed top to bottom, etc. [69]).

4.3 Programming Environments for Supporting Transfer

It is clear from the theoretical and empirical research that transitioning students from one programming language to another requires careful planning in order to maximize the positive transfer of prior knowledge and minimize negative transfer. The previous section discussed ways of designing lessons to support transfer between languages. Researchers have also developed a number of programming environments to support transfer which implement the above strategies to varying degrees. An early attempt was Fix and Wiedenbeck's ADAPT environment, which used artificial intelligence to help learners familiar with the C or Pascal programming languages learn the Ada programming language [30]. To do so, ADAPT provides suggestions of potential commands and templates of common programming actions to scaffold the learner.

Researchers have also developed multiple programming environments that combine features of block-based and text-based programming. For example, Pencil Code [9] and Tiled Grace [46] enable block-based and text-based programming in a single interface. Other environments bring block-based features into text-based IDEs, thus situating block-based programming in the context of conventional text-based editors [13, 58], or otherwise blending aspects of block-based and text-based environments into a single hybrid interface [124]. Frame-based editing is a notable approach in this space [61]. It retains the block-based characteristic of preventing syntax errors through scaffolded and context-aware inputs, but commands are input via the keyboard and the resulting program has a text-based appearance.

These types of hybrid environments create an intermediate step between block-based environments and text-based IDEs, but may not actively encourage students to engage in Structure Mapping or otherwise leverage insights from theory. Shrestha and colleagues' Transfer Tutor does. The Transfer Tutor helped users learn a new language (in this case R) by presenting the equivalent implementation of programs in a language familiar to the user (Python). The Transfer Tutor stepped students through examining predetermined code snippets, using highlighting and tooltips to draw students' attention to similarities and differences between the languages. However, students could not write or execute the code [107].

4.4 Recommendations for Future Development

As the research on supporting transfer acknowledges, "the nature and extent of instruction needed to enable cross-language transfer in upper-level students is an interesting open question" [29, p. 218]. We encourage teachers and curriculum developers to purposefully create lessons to facilitate transitions from one language to the next using the strategies described above and to consider how multiple strategies can be layered and/or further supported by students' programming environment.

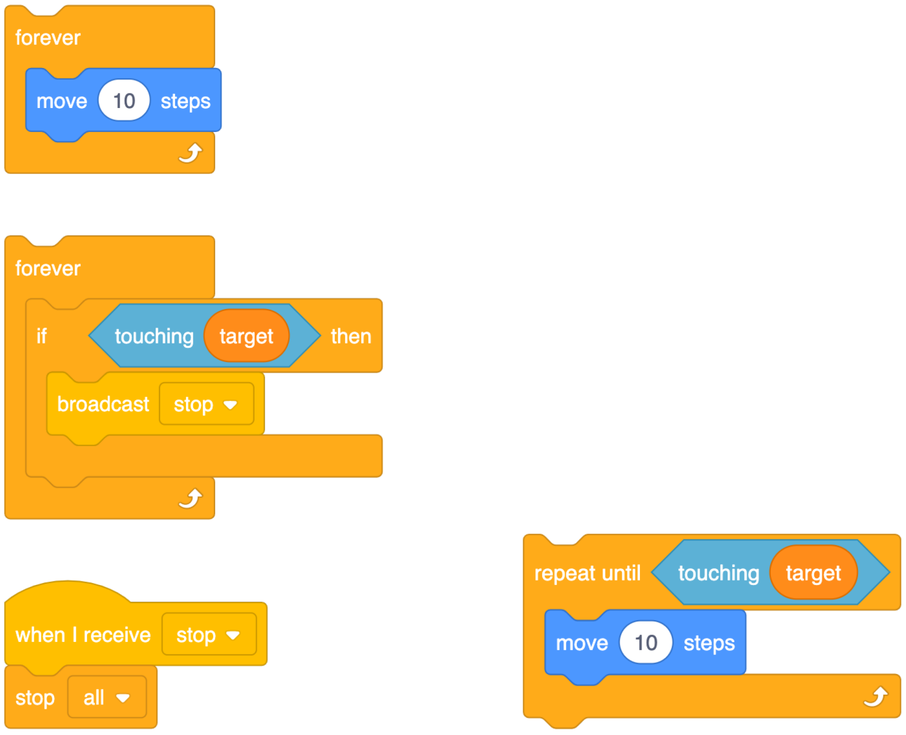
For example, one lesson could ask students to compare a programming solution written in both languages. Display both solutions side-by-side and ask students to identify and discuss similarities and differences. As students discuss the code, support students in Structure Mapping by using color-coding, arrows, or other cues to connect analogous sections of code. This type of exercise could be used to help students understand similarities and differences in syntax as well as broader programming strategies, especially if they are implemented in a dynamic programming environment that updates the cues in response to student actions. For example, Moors and colleagues found that students who learned Scratch often developed a habit of "extreme fine-grained programming" [77]. Students who program in this style take advantage of Scratch's parallelism and the forever loop to break up simple tasks into even smaller sub-tasks and avoid the use of control structures (Figure 4(a)). This style of programming does not transfer to text-based languages but a more straightforward implementation (Figure 4(b)) would. In a similar vein, Repenning and Basawapatna discuss how the different affordances of the block-based languages Scratch and AgentCube led students to implement different approaches to programming an hourglass simulation [92]. Explicitly discussing different ways of accomplishing the same task in different programming languages will improve students' understanding of the structure of both languages.

We also encourage primary and secondary school and district-level administrators to consider issues of transfer when planning multi-grade computer science course pathways. Teaching too many programming languages in too short of a time can lead to a fractured experience where students re-learn the same concepts year after year instead of deepening their understanding of core CS concepts over time [52]. We recommend minimizing the number of new programming languages students encounter, particularly in their first few years of computer science instruction. When students do transition to a new programming language, we recommend developing curriculum pacing guides that include lessons designed to promote the transfer of core CS knowledge from students' prior programming language to the new one.

5 DISCUSSION

5.1 The Role of Theory in Computing Education Research

We see great potential in drawing on findings from the field of cognitive science in general, and research on analogical transfer specifically, as a generative activity to advance our understanding of how students learn to program. The framework presented by Malmi and colleagues on the four roles that theory can play in computing education research [67] can help us think through what form this may take. The first role Malmi et al. discuss is of using theory as a way to discuss results. Given the significant amount of empirically-grounded research on students' transfer (or failure to transfer) (e.g., [22, 43, 125], there is an opportunity to incorporate theoretical constructs from cognitive science, such as analogical reasoning or structure vs. surface similarity when discussing the data. The second role of theory, to predict results, has similar potential. When designing a



(a) Extreme fine-grained programming. (b) More standard implementation.

Fig. 4. Examples of different programming strategies from Moors et al. [77, p. 61–62].

new programming language, environment, or curriculum, cognitive science can be used to shape expectations that can then be empirically tested. The third role of theory is to inform pedagogy. As discussed above, there is research on pedagogical approaches to support students learning a second programming language, some of which is grounded in theory (e.g., [117]), and there are clear examples from cognitive science that can further advance the approaches used in computing education research (e.g., [37]). Finally, we see great potential in theories from cognitive science related to learning and transfer as a data analysis framework, especially given the computing education field's focus on classroom-based research. While some researchers are already using theory to explain empirical results (e.g., [115]), the larger body of analogical transfer literature presented above can provide a framework for reanalysis or meta-analysis of previous studies.

5.2 Towards CS-specific Theories of Transfer

Kölling and colleagues [60] identified 13 challenges when going from block-based to text-based languages. These challenges range from memorizing syntax and commands in text-based languages to the organization (or lack thereof) of block-based commands and interpreting error messages in text-based environments. Integrating work of this nature with psychological theories of analogical transfer can lead to the development of CS-specific theories of transfer. A CS-specific theory could move beyond general descriptions of how and when the transfer occurs to make more specific predictions about what knowledge and skills will transfer easily between programming language pairs. CS education researchers could draw from Barnett and Ceci's [7] taxonomy of transfer distance which we described in Section 2.2. A CS-specific version of this taxonomy could include contextual dimensions such as programming modality (i.e., blocks vs. text), programming paradigm, or learning context (i.e., formal education vs. informal learning environments) and describe how different dimensions intersect to create situations where knowledge transfers more or less easily. For example, due to the asymmetric nature of similarity [55, 81, 118], CS-specific theories may predict that certain skills will transfer well from one language to another (e.g., block-based to text-based), but transfer poorly in the reverse direction (e.g., text-based to block-based).

CS-specific theories could also make predictions about when and how to scaffold transfer of specific content knowledge. For example, a CS-specific theory might predict that students learning Scratch, which uses "repeat [number]" loops that iterate a pre-specified number of times, and then Java, which uses for loops with an initializing statement and an incrementer/decrementer, would benefit from explicit instruction on the similarities between the looping structures as well as looping errors and strategies that are less easily implemented in Scratch (i.e., off-by-one errors or alternatives to incrementing/decrementing by one). Work in developing and testing CS-specific theories of transfer has begun (e.g., [115]), but much work remains in developing pedagogy, curricula, and tools explicitly based on these theories. Further, CS-specific theories can also inform the design and implementation of multi-lingual course pathways by informing the selection of languages and the timing of language transitions.

CS-specific theories of transfer would also inform the development of valid and reliable assessments. Creating effective multi-lingual assessments requires assessment developers to understand which skills and concepts transfer across languages and which are unique to a given language and apply these understandings to design assessment items, scoring models, and score interpretations. Research on pseudocode-based assessments like the Advanced Placement CS Principles exam or the FCS1/SCS1 has found that the pseudocode is not truly language-neutral or language-independent [44]. The way programs are presented (block-based or text-based) impacts student performance even when the assessment is in pseudocode [122]. Students whose primary programming language was more similar to the pseudocode tended to perform better on the assessment. In addition, students' error patterns on the pseudocode-based assessment differed depending on the language they were taught [82, 113, 123]. In other words, students' assessment performance was dependent on how easily they were able to transfer knowledge from the programming language they were taught to the pseudocode. Ideally, scoring models for pseudocode-based assessments would account for this variation in difficulty; a well-developed CS-specific theory of transfer would facilitate the development of such models.

6 CONCLUSION

In this article, we focused on the prior and potentially future links between research on transfer in cognitive psychology and research on programming instruction in computing education. We reviewed two large bodies of literature with significant, but relatively untapped, synergy. Emerging theories on knowledge transfer between programming languages are strongly grounded in empirical findings but there is room for cognitive science to provide explanatory mechanisms for the observed behavior. For example, drawing on the distinction of surface vs. structural similarity as a means to explain successful and unsuccessful transfer as it relates to syntactic (i.e., surface) and semantic (i.e., structural) similarities. In presenting reviews of these two literatures side-by-side, our hope is to help the computing education research field see such opportunities for fruitful cross-pollination of ideas based on contemporary cognitive science work on the analogical transfer.

We conclude with some more general thoughts about the utility of applying theory to practice. Applying theory to practice is an iterative, rather than a linear, process. Creating instructional materials is an act of design or engineering–context and constraints both matter greatly. There are likely multiple ways to instantiate any given theoretical recommendation into an actual lesson.

Even theories supported by decades of research can turn out to be under-specified in important ways when researchers try to translate those recommendations into instructional materials that fit a specific context. Teaching and learning go far beyond what happens in a student's brain–it is a complex cultural activity. Creating theory-informed instructional materials–and then empirically testing which products were most effective–is a necessary feedback mechanism for refining theories of learning [110, 111].

REFERENCES

[1] Louis Alfieri, Timothy J. Nokes-Malach, and Christian D. Schunn. 2013. Learning through case comparisons: A meta-analytic review. *Educational Psychologist* 48, 2 (2013), 87–113.

[2] John R. Anderson. 1993. *Rules of the Mind.* Lawrence Erlbaum Associates, Hillsdale, NJ.

[3] John R. Anderson and Christian D. Schunn. 2000. Implications of the ACT-R learning theory: No magic bullets. In *Proceedings of the Advances in Instructional Psychology: Educational Design and Cognitive Science.* R. Glaser (Ed.). Lawrence Erlbaum Associates, 1–33.

[4] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari. 2015. From scratch to "Real" programming. *ACM Transactions on Computing Education* 14, 4 (2015), 25:1–15. Retrieved from http://dl.acm.org/citation.cfm?id=2677087.

[5] Armstrong and Hardgrave. 2007. Understanding mindshift learning: The transition to object-oriented development. *MIS Quarterly* 31, 3 (2007), 453. DOI: https://doi.org/10.2307/25148803

[6] Deb Armstrong and H. James Nelson. 2000. Knowledge transfer between languages and paradigms. In *Proceedings of the Americas Conference on Information Systems.* 8.

[7] Susan M. Barnett and Stephen J. Ceci. 2002. When and where do we apply what we learn?: A taxonomy for far transfer. *Psychological Bulletin* 128, 4 (2002), 612.

[8] Lawrence W. Barsalou. 1999. Perceptual symbol systems. *Behavioral and Brain Sciences* 22, 4 (1999), 577–660.

[9] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens. 2015. Pencil code: Block code for a text world. In *Proceedings of the 14th International Conference on Interaction Design and Children.* ACM, New York, NY, 445–448. DOI: https://doi.org/10.1145/2771839.2771875

[10] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable programming: Blocks and beyond. *Communications of the ACM* 60, 6 (2017), 72–80. DOI: https://doi.org/10.1145/3015455

[11] Elizabeth L. Bjork and Robert A. Bjork. 2011. Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. *Psychology and the Real World: Essays Illustrating Fundamental Contributions to Society* 2 (2011), 59–68.

[12] Alan F. Blackwell, Marian Petre, and Luke Church. 2019. Fifty years of the psychology of programming. *International Journal of Human-Computer Studies* 131 (2019), 52–63.

[13] Jeremiah Blanchard, Chistina Gardner-McCune, and Lisa Anthony. 2019. Amphibian: Dual-modality representation in integrated development environments. In *Proceedings of the 2019 IEEE Blocks and Beyond Workshop.* IEEE, Memphis, TN, 83–85. DOI: https://doi.org/10.1109/BB48857.2019.8941213

[14] John D. Bransford and Daniel L. Schwartz. 1999. Rethinking Transfer: A Simple Proposal with Multiple Implications. *Review of Research in Education* 24, 1 (Jan. 1999), 61–100. https://doi.org/10.2307/1167267

[15] John T. Bruer. 1997. Education and the brain: A bridge too far. *Educational Researcher* 26, 8 (1997), 4–16.

[16] William G. Chase and Herbert A. Simon. 1973. Perception in chess. *Cognitive Psychology* 4, 1 (1973), 55–81.

[17] Michelene T. H. Chi, Paul J. Feltovich, and Robert Glaser. 1981. Categorization and representation of physics problems by experts and novices. *Cognitive Science* 5, 2 (1981), 121–152.

[18] Daniel C. Cliburn. 2008. Student opinions of Alice in CS1. In *Proceedings of the 38th Annual Frontiers in Education Conference.* IEEE, T3B–1.

[19] Jennfer G. Cromley, Steven M. Weisberg, Ting Dai, Nora S. Newcombe, Christian D. Schunn, Christine Massey, and F. Joseph Merlino. 2016. Improving middle school science learning using diagrammatic reasoning. *Science Education* 100, 6 (2016), 1184–1213. DOI: https://doi.org/10.1002/sce.21241

[20] W. Dann, S. Cooper, and B Ericson. 2009. *Exploring Wonderland: Java Programming Using Alice and Media Computation.* Prentice Hall Press.

[21] Wanda Dann, Stephen Cooper, and Randy Pausch. 2006. *Learning to Program with Alice.* Prentice-Hall, Inc.

[22] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper. 2012. Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education.* ACM, 141–146.

[23] Jodi L. Davenport, Yvonne S. Kao, Bryan J. Matlen, and Steven A. Schneider. 2020. Cognition research in practice: Engineering and evaluating a middle school math curriculum. *The Journal of Experimental Education* 88, 4 (2020), 516–535. DOI: https://doi.org/10.1080/00220973.2019.1619067

[24] Samuel B. Day and Robert L. Goldstone. 2012. The import of knowledge export: Connecting findings and theories of transfer of learning. *Educational Psychologist* 47, 3 (2012), 153–176.

[25] Mark Dorling and Dave White. 2015. Scratch: A way to logo and python. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education.* ACM, Kansas City Missouri, 191–196. DOI: https://doi.org/10.1145/2676723.2677256

[26] C. Duncan, T. Bell, and S. Tanimoto. 2014. Should your 8-year-old learn coding?. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education.* ACM, New York, NY, 60–69. DOI: https://doi.org/10.1145/2670757.2670774

[27] John Dunlosky, Katherine A. Rawson, Elizabeth J. Marsh, Mitchell J. Nathan, and Daniel T. Willingham. 2013. Improving students' learning with effective learning techniques. *Psychological Science in the Public Interest* 14, 1 (2013), 4–58. DOI: https://doi.org/10.1177/1529100612453266

[28] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing plan-composition studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education.* 6.

[29] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. 2017. Assessing and teaching scope, mutation, and aliasing in upper-level undergraduates. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education.* ACM, Seattle Washington, 213–218. DOI: https://doi.org/10.1145/3017680.3017777

[30] V. Fix and S. WIEDENBECK. 1996. An intelligent tool to aid students in learning second and subsequent programming languages. *Computers & Education* 27, 2 (1996), 71–83. DOI: https://doi.org/10.1016/0360-1315(96)00022-X

[31] D. Franklin, G. Skifstad, R. Rolock, I. Mehrotra, V. Ding, A. Hansen, D. Weintrop, and D. Harlow. 2017. Using upper-elementary student performance to understand conceptual sequencing in a blocks-based curriculum. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education.* ACM, New York, NY, 231–236. DOI: https://doi.org/10.1145/3017680.3017760

[32] Laura Fries, Ji Y. Son, Karen B. Givvin, and James W. Stigler. 2021. Practicing connections: A framework to guide instructional design for developing understanding in complex domains. *Educational Psychology Review* 33, 2 (2021), 739–762.

[33] Ryan Garlick and Ebru Celikel Cankaya. 2010. Using alice in CS1: A quantitative experiment. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education.* ACM, 165–168.

[34] Dedre Gentner. 1983. Structure-mapping: A theoretical framework for analogy. *Cognitive Science* 7, 2 (1983), 155–170.

[35] Dedre Gentner. 2010. Bootstrapping the mind: Analogical processes and symbol systems. *Cognitive Science* 34, 5 (2010), 752–775.

[36] Dedre Gentner, Susan C. Levine, Raedy Ping, Ashley Isaia, Sonica Dhillon, Claire Bradley, and Garrett Honke. 2016. Rapid learning in a children's museum via analogical comparison. *Cognitive Science* 40, 1 (2016), 224–240.

[37] Dedre Gentner, Jeffrey Loewenstein, and Leigh Thompson. 2003. Learning and transfer: A general role for analogical encoding. *Journal of Educational Psychology* 95, 2 (2003), 393.

[38] Dedre Gentner, Mary Jo Rattermann, Arthur Markman, and Laura Kotovsky. 1995. Two forces in the development of relational similarity. In *Developing Cognitive Competence: New Approaches to Pocess Mdeling.* Erlbaum, 263–313.

[39] Mary L. Gick and Keith J. Holyoak. 1983. Schema Induction and Analogical Transfer. *Cognitive Psychology* 15, 1 (1983), 1–38.

[40] Robert L. Goldstone and Samuel B. Day. 2012. Introduction to "new conceptualizations of transfer of learning". *Educational Psychologist* 47, 3 (2012), 149–152.

[41] Marcos J. Gomez, Marco Moresi, and Luciana Benotti. 2019. Text-based programming in elementary school: A comparative study of programming abilities in children with and without block-based experience. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education.* ACM, Aberdeen Scotland UK, 402–408. DOI: https://doi.org/10.1145/3304221.3319734

[42] Shuchi Grover. 2021. Teaching and assessing for transfer from block-to-text programming in middle school computer science. *Transfer of Learning: Progressive Perspectives for Mathematics Education and Related Fields.* Springer Nature.

[43] Shuchi Grover, Roy Pea, and Stephen Cooper. 2015. Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education* 25, 2 (2015), 199–237. DOI: https://doi.org/10.1080/08993408.2015.1033142

[44] Mark Guzdial. 2019. We should stop saying language independent. We don't know how to do that. *Blog @ The Communications of the ACM* (2019). Retrieved from https://cacm.acm.org/blogs/blog-cacm/238782-we-should-stop-saying-language-independent-we-dont-know-how-to-do-that/fulltext.

[45] Erin J. Higgins, Amanda M. Dettmer, and Elizabeth R. Albro. 2019. Looking back to move forward: A retrospective examination of research at the intersection of cognitive science and education and what it means for the future. *Journal of Cognition and Development* 20, 2 (2019), 278–297. DOI: https://doi.org/10.1080/15248372.2019.1565537

[46] Michael Homer and James Noble. 2014. Combining tiled and textual views of code. In *Proceedings of the IEEE Working Conference on Software Visualisation.* IEEE, Victoria, BC, 1–10. DOI: https://doi.org/10.1109/VISSOFT.2014.11

[47] Benjamin D. Jee and Florencia K. Anggoro. 2019. Relational scaffolding enhances children's understanding of scientific models. *Psychological Science* 30, 9 (2019), 1287–1302.

[48] Benjamin D. Jee, David H. Uttal, Dedre Gentner, Cathy Manduca, Thomas F. Shipley, and Bradley Sageman. 2013. Finding faults: Analogical comparison supports spatial concept learning in geoscience. *Cognitive Processing* 14, 2 (2013), 175–187.

[49] N. Jiang. 2000. Lexical representation and development in a second language. *Applied Linguistics* 21, 1 (2000), 47–77. DOI: https://doi.org/10.1093/applin/21.1.47

[50] Nan Jiang. 2004. Semantic transfer and its implications for vocabulary teaching in a second language. *The Modern Language Journal* 88, 3 (2004), 416–432. DOI: https://doi.org/10.1111/j.0026-7902.2004.00238.x

[51] Jennifer A. Kaminski, Vladimir M. Sloutsky, and Andrew F. Heckler. 2013. The Cost of Concreteness: The Effect of Nonessential Information on Analogical Transfer. *Journal of Experimental Psychology: Applied* 19, 1 (2013), 14–29.

[52] Yvonne Kao, Irene Nolan, and Andrew Rothman. 2020. Project scoring for program evaluation and teacher professional development. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, New York, NY, 1133–1138. DOI: https://doi.org/10.1145/3328778.3366959

[53] C. Kelleher, R. Pausch, and S. Kiesler. 2007. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1455–1464.

[54] Philip J. Kellman and Christine M. Massey. 2013. Perceptual learning, cognition, and expertise. In *Proceedings of the Psychology of Learning and Motivation*. Elsevier, 117–165.

[55] Alex Koch, Hans Alves, Tobias Krüger, and Christian Unkelbach. 2016. A general valence asymmetry in similarity: Good is more alike than bad. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 42, 8 (2016), 1171.

[56] Laura Kotovsky and Dedre Gentner. 1996. Comparison and categorization in the development of relational similarity. *Child Development* 67, 6 (1996), 2797–2822.

[57] Shriram Krishnamurthi and Kathi Fisler. 2019. Programming paradigms and beyond. In *Proceedings of the Cambridge Handbook of Computing Education Research*. Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University, 377–413. DOI: https://doi.org/10.1017/9781108654555.014

[58] D. Krpan, S. Mladenovic, and G. Zaharija. 2017. Mediated transfer from visual to high-level programming language. In *Proceedings of the 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics*. IEEE, Opatija, Croatia, 800–805. DOI: https://doi.org/10.23919/MIPRO.2017.7973531

[59] Kenneth J. Kurtz, Chun-Hui Miao, and Dedre Gentner. 2001. Learning by analogical bootstrapping. *The Journal of the Learning Sciences* 10, 4 (2001), 417–446.

[60] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. 2015. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*. ACM, New York, NY, 29–38. DOI: https://doi.org/10.1145/2818314.2818331

[61] M. Kölling, N. C. C. Brown, and A. Altadmri. 2017. Frame-based editing. *Journal of Visual Languages and Sentient Systems* 3 (2017), 40–67. DOI: https://doi.org/10.18293/VLSS2017

[62] David Landy and Robert L. Goldstone. 2007. Formal notations are diagrams: Evidence from a production task. *Memory & Cognition* 35, 8 (2007), 2033–2040.

[63] David Landy and Robert L. Goldstone. 2007. How abstract is symbolic thought? *Journal of Experimental Psychology: Learning, Memory, and Cognition* 33, 4 (2007), 720.

[64] C. M. Lewis. 2010. How programming environment shapes perception, learning and goals: Logo vs. Scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. New York, NY, 346–350.

[65] Yuhan Lin and David Weintrop. 2021. The Landscape of Block-based Programming: Characteristics of block-based environments and how they support the transition to text-based programming. *Journal of Computer Languages* 67 (2021), 101075.

[66] Meryl Reis Louis and Robert I. Sutton. 1991. Switching cognitive gears: From habits of mind to active thinking. *Human Relations* 44, 1 (1991), 55–76. DOI: https://doi.org/10.1177/001872679104400104

[67] Lauri Malmi, Judy Sheard, Päivi Kinnunen, Simon, and Jane Sinclair. 2019. Computing education theories: What are they and how are they used?. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. Association for Computing Machinery, New York, NY, 187–197. DOI: https://doi.org/10.1145/3291279.3339409

[68] Lauren Margulieux, Brian Dorn, and Kristin Searle. 2019. Learning sciences for computing education. In *Proceedings of the Cambridge Handbook of Computing Education Research*. Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University, Cambridge, UK, 208–230.

[69] Bryan J. Matlen, Dedre Gentner, and Steven L. Franconeri. 2020. Spatial alignment facilitates visual comparison. *Journal of Experimental Psychology: Human Perception and Performance* 46, 5 (2020), 443.

[70] Bryan J. Matlen and David Klahr. 2013. Sequential effects of high and low instructional guidance on children's acquisition of experimentation skills: Is it all in the timing? *Instructional Science* 41, 3 (2013), 621–634.

[71] Bryan J. Matlen, Lindsey E. Richland, Ellen C. Klostermann, and Emily Lyons. 2018. Impact and prevalence of diagrammatic supports in mathematics classrooms. In *Proceedings of the International Conference on Theory and Application of Diagrams*. Springer, 148–163.

[72] Bryan J. Matlen, Stella Vosniadou, Benjamin Jee, and Maria Ptouchkina. 2011. Enhancing the comprehension of science text through visual analogies. In *Proceedings of the Annual Meeting of the Cognitive Science Society*.

[73] Nicole M. McNeil, Caroline Byrd Hornburg, Heather Brletic-Shipley, and Julia M. Matthews. 2019. Improving children's understanding of mathematical equivalence via an intervention that goes beyond nontraditional arithmetic practice. *Journal of Educational Psychology* 111, 6 (2019), 1023–1044. DOI: https://doi.org/10.1037/edu0000337

[74] O. Meerbaum-Salant, M. Armoni, and M. M. Ben-Ari. 2010. Learning computer science concepts with Scratch. In *Proceedings of the 6th International Workshop on Computing Education Research*. 69–76.

[75] Norma Ming. 2015. Analogies vs. Contrasts: A comparison of their learning benefits. In *Proceedings of the 2nd International Conference on Analogy*. NBU, Sofia, Bulgaria, 338–347.

[76] Monika Mladenović, Žana Žanko, and Andrina Granić. 2021. Mediated transfer: From text to blocks and back. *International Journal of Child-Computer Interaction* 29 (2021), 100279. DOI: https://doi.org/10.1016/j.ijcci.2021.100279

[77] Luke Moors, Andrew Luxton-Reilly, and Paul Denny. 2018. Transitioning from block-based to text-based programming languages. In *Proceedings of the 2018 International Conference on Learning and Teaching in Computing and Engineering*. IEEE, Auckland, New Zealand, 57–64. DOI: https://doi.org/10.1109/LaTICE.2018.000-5

[78] H. James Nelson, Deborah J. Armstrong, and Mehdi Ghods. 2002. Old dogs and new tricks. *Communications of the ACM* 45, 10 (2002), 132–137. DOI: https://doi.org/10.1145/570907.570910

[79] H. J. Nelson, G. Irwin, and D. Monarchi. 1997. Journeys up the mountain: Different paths to learning object-oriented programming. *Accounting, Management and Information Technologies* 7, 2 (1997), 53–85. DOI: https://doi.org/10.1016/S0959-8022(96)00024-0

[80] Mark Noone and Aidan Mooney. 2018. Visual and textual programming languages: A systematic review of the literature. *Journal of Computers in Education* 5, 2 (2018), 149–174. DOI: https://doi.org/10.1007/s40692-018-0101-5

[81] Andrew Ortony, Richard J. Vondruska, Mark A. Foss, and Lawrence E. Jones. 1985. Salience, similes, and the asymmetry of similarity. *Journal of Memory and Language* 24, 5 (1985), 569–594.

[82] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, validation, and use of a language independent CS1 knowledge assessment. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. Association for Computing Machinery, New York, NY, 93–101. DOI: https://doi.org/10.1145/2960310.2960316

[83] D. Parsons and P. Haden. 2007. Programming osmosis: Knowledge transfer from imperative to visual programming environments. In *Procedings of the 20th Annual NACCQ Conference*. S. Mann and N. Bridgeman (Eds.). Hamilton, New Zealand, 209–215.

[84] Harold Pashler, Patrice M. Bain, Brian A. Bottge, A. Graesser, Kenneth Koedinger, Mark McDaniel, and Janet Metcalfe. 2007. *Organizing Instruction and Study to Improve Student Learning (NCER 2007-2004)*. Technical Report. National Center for Education Research, Institute of Education Sciences, U.S. Department of Education., Washington, D. C. Retrieved from https://ies.ed.gov/ncee/wwc/PracticeGuide/1.

[85] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295–341. DOI: https://doi.org/10.1016/0010-0285(87)90007-7

[86] N. Pennington, A. Y. Lee, and B. Rehder. 1995. Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human-Computer Interaction* 10, 2 (1995), 171–226.

[87] David Perkins. 2010. *Making Learning Whole: How Seven Principles of Teaching Can Transform Education*. John Wiley & Sons.

[88] David N. Perkins and Gavriel Salomon. 1988. Teaching for transfer. *Educational Leadership* 46, 1 (1988), 22–32.

[89] K. Powers, S. Ecott, and L. M. Hirshfield. 2007. Through the looking glass: Teaching CS0 with Alice. *ACM SIGCSE Bulletin* 39, 1 (2007), 213–217.

[90] Thomas W. Price and Tiffany Barnes. 2015. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the 11th Annual International Conference on International Computing Education Research*. ACM , 91–99. DOI: https://doi.org/10.1145/2787622.2787712

[91] Stephen K. Reed, George W. Ernst, and Ranan Banerji. 1974. The role of analogy in transfer between similar problem states. *Cognitive Psychology* 6, 3 (1974), 436–450.

[92] Alexander Repenning and Ashok Basawapatna. 2021. Smacking Screws with Hammers: Experiencing Affordances of Block-Based Programming through the Hourglass Challenge. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, New York, NY, 267–273. DOI: https://doi.org/10.1145/3408877.3432444

[93] M. Resnick, Brian Silverman, Yasmin Kafai, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, and Jay Silver. 2009. Scratch: Programming for all. *Communications of the ACM* 52, 11 (2009), 60. Retrieved from http://portal.acm.org.turing.library.northwestern.edu/citation.cfm?id=1592761.1592779.

[94] Lindsey E. Richland and Ian M. McDonough. 2010. Learning by analogy: Discriminating between potential analogs. *Contemporary Educational Psychology* 35, 1 (2010), 28–43.

[95] Frank E. Ritter, Farnaz Tehranchi, and Jacob D. Oury. 2018. ACT-R: A cognitive architecture for modeling cognition. *Wiley Interdisciplinary Reviews: Cognitive Science* 10, 3 (2018), e1488. https://doi.org/10.1002/wcs.1488

[96] Bethany Rittle-Johnson, Jon R. Star, and Kelley Durkin. 2009. The importance of prior knowledge when comparing examples: Influences on conceptual and procedural knowledge of equation solving. *Journal of Educational Psychology* 101, 4 (2009), 836.

[97] Anthony V. Robins, Lauren Margulieux, and Briana B. Morrison. 2019. Cognitive sciences for computing education. In *Proceedings of the Cambridge Handbook of Computing Education Research.* Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Cambridge, UK, 231–275.

[98] Doug Rohrer, Robert F. Dedrick, and Sandra Stershic. 2015. Interleaved practice improves mathematics learning. *Journal of Educational Psychology* 107, 3 (2015), 900.

[99] Benjamin M. Rottman, Dedre Gentner, and Micah B. Goldwater. 2012. Causal systems categories: Differences in novice and expert categorization of causal phenomena. *Cognitive Science* 36, 5 (2012), 919–932.

[100] D. Saito, H. Washizaki, and Y. Fukazawa. 2016. Analysis of the learning effects between text-based and visual-based beginner programming environments. In *Proceedings of the 2016 IEEE 8th International Conference on Engineering Education.* 208–213. DOI: https://doi.org/10.1109/ICEED.2016.7856073

[101] Igor Moreno Santos, Matthias Hauswirth, and Nathaniel Nystrom. 2019. Experiences in bridging from functional to object-oriented programming. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E - SPLASH-E 2019.* ACM, Athens, Greece, 36–40. DOI: https://doi.org/10.1145/3358711.3361628

[102] Jean Scholtz and Susan Wiedenbeck. 1990. Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction* 2, 1 (1990), 51–72. Retrieved from http://www.tandfonline.com/doi/abs/10.1080/10447319009525970.

[103] J. Scholtz and S. Wiedenbeck. 1991. Learning a new programming language: A model of the planning process. In *Proceedings of the 24th Annual Hawaii International Conference on System Sciences.* 3–12. DOI: https://doi.org/10.1109/HICSS.1991.183956

[104] Jean Scholtz and Susan Wiedenbeck. 1993. Using unfamiliar programming languages: The effects on expertise. *Interacting with Computers* 5, 1 (1993), 13–30. DOI: https://doi.org/10.1016/0953-5438(93)90023-M

[105] Carsten Schulte. 2008. Block Model: An educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceeding of the 4th International Workshop on Computing Education Research - ICER'08.* ACM, Sydney, Australia, 149–160. DOI: https://doi.org/10.1145/1404520.1404535

[106] Ben Shneiderman and Richard Mayer. 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences* 8, 3 (1979), 219–238. DOI: https://doi.org/10.1007/BF00977789

[107] Nischal Shrestha, Titus Barik, and Chris Parnin. 2018. It's like python but: Towards supporting transfer of programming language knowledge. In *Proceeding of the 2018 IEEE Symposium on Visual Languages and Human-Centric Computing.* IEEE, Lisbon, 177–185. DOI: https://doi.org/10.1109/VLHCC.2018.8506508

[108] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. 2020. Here we go again: Why is it difficult for developers to learn another programming language?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* ACM, Seoul South Korea, 691–701. DOI: https://doi.org/10.1145/3377811.3380352

[109] Elliot Soloway and James C. Spohrer. 1988. *Studying the Novice Programmer.* Taylor & Francis Group.

[110] James W. Stigler and Karen B. Givven. 2017. Online learning as a wind tunnel for improving teaching. In *Proceeding of the Improvement Science in Evaluation: Methods and Uses. New Directions for Evaluation*, C. A. Christie, M. Inkelas, and S. Lemire (Eds.), Vol. 153. 79–91. Retrieved from https://uclatall.com/.

[111] James W. Stigler, Ji Y. Son, Karen B. Givvin, Adam B. Blake, Laura Fries, Stacy T. Shaw, and Mary C. Tucker. 2020. The better book approach for education research and development. *Teachers College Record* 122 (2020), 32 pages. Retrieved from https://uclatall.com/.

[112] Nour Tabet, Huda Gedawy, Hanan Alshikhabobakr, and Saquib Razak. 2016. From alice to python. introducing text-based programming in middle schools. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE'16.* ACM, Arequipa, Peru, 124–129. DOI: https://doi.org/10.1145/2899415.2899462

[113] Allison E. Tew. 2010. *Assessing Fundamental Introductory Computing Concept Knowledge in a Language Independent Manner.* Ph.D. Dissertation. Atlanta, GA. Retrieved from http://hdl.handle.net/1853/37090.

[114] Ethel Tshukudu and Quintin Cutts. 2020. Semantic transfer in programming languages: Exploratory study of relative novices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education.* ACM, Trondheim Norway, 307–313. DOI: https://doi.org/10.1145/3341525.3387406

[115] Ethel Tshukudu and Quintin Cutts. 2020. Semantic transfer in programming languages: Exploratory study of relative novices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education.* Association for Computing Machinery, New York, NY, 307–313. DOI: https://doi.org/10.1145/3341525.3387406

[116] Ethel Tshukudu and Quintin Cutts. 2020. Understanding conceptual transfer for students learning new programming languages. In *Proceedings of the 2020 ACM Conference on International Computing Education Research.* 227–237.

[117] Ethel Tshukudu and Siri Annethe Moe Jensen. 2020. The role of explicit instruction on students learning their second programming language. In *Proceedings of the United Kingdom & Ireland Computing Education Research Conference.* ACM, Glasgow United Kingdom, 10–16. DOI: https://doi.org/10.1145/3416465.3416475

[118] Amos Tversky. 1977. Features of similarity. *Psychological Review* 84, 4 (1977), 327.

[119] A. Von Mayrhauser and A. M. Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55. DOI: https://doi.org/10.1109/2.402076

[120] Karen P. Walker and Stephen R. Schach. 1996. Obstacles to learning a second programming language: An empirical study. *Computer Science Education* 7, 1 (1996), 1–20. DOI: https://doi.org/10.1080/0899340960070101

[121] David Weintrop. 2019. Block-based programming in computer science education. *Communications of the ACM* 62, 8 (2019), 22–25. DOI: https://doi.org/10.1145/3341221

[122] David Weintrop, Heather Killen, and Baker E. Franke. 2018. *Blocks or Text? How Programming Language Modality Makes a Difference in Assessing Underrepresented Populations.* International Society of the Learning Sciences, Inc.[ISLS].

[123] D. Weintrop and U. Wilensky. 2015. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *Proceedings of the 11th Annual International Conference on International Computing Education Research.* ACM, New York, NY, 101–110. DOI: https://doi.org/10.1145/2787622.2787721

[124] D. Weintrop and U. Wilensky. 2017. Between a block and a typeface: Designing and evaluating hybrid programming environments. In *Proceedings of the 2017 Conference on Interaction Design and Children.* ACM, New York, NY, 183–192. DOI: https://doi.org/10.1145/3078072.3079715

[125] David Weintrop and Uri Wilensky. 2017. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education* 18, 1 (2017), 3. DOI: https://doi.org/10.1145/3089799

[126] David Weintrop and Uri Wilensky. 2019. Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers & Education* 142 (2019), 103646. DOI: https://doi.org/10.1016/j.compedu.2019.103646

[127] Mark Weiser and Joan Shertz. 1983. Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies* 19, 4 (1983), 391–398.

[128] Susan Wiedenbeck. 1993. An analysis of novice programmers learning a second language. In *Proceeding of the Empirical Studies of Programmers: 5th Workshop: Papers Presented at the 5th Workshop on Empirical Studies of Programmers, December 3-5, 1993, Palo Alto, CA.* Intellect Books, 187.

[129] Quanfeng Wu and John R. Anderson. 1990. *Problem-solving Transfer among Programming Languages.* Technical Report. Carnegie Mellon University. Retrieved from https://apps.dtic.mil/sti/citations/ADA225798.

Received 14 January 2021; revised 16 November 2021; accepted 17 December 2021