# What Happens When Students Switch (Functional) Languages (Experience Report)

KUANG-CHEN LU, Brown University, United States of America
SHRIRAM KRISHNAMURTHI, Brown University, United States of America
KATHI FISLER, Brown University, United States of America
ETHEL TSHUKUDU, University of Botswana, Botswana

When novice programming students already know one programming language and have to learn another, what issues do they run into? We specifically focus on one or both languages being functional, varying along two axes: syntax and semantics. We report on problems, especially persistent ones. This work can be of immediate value to educators and also sets up avenues for future research.

CCS Concepts: • **Social and professional topics** → **Computing education**; • **Software and its engineering** → **General programming languages**.

Additional Key Words and Phrases: Programming language learning, Language transfer, Pyret, Python, Racket

**ACM Reference Format:**
Kuang-Chen Lu, Shriram Krishnamurthi, Kathi Fisler, and Ethel Tshukudu. 2023. What Happens When Students Switch (Functional) Languages (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 215 (August 2023), 17 pages. https://doi.org/10.1145/3607857

## 1 INTRODUCTION

A great deal of work has gone into teaching students their first programming language. Less work has gone into what happens when students learn a subsequent one. While prior knowledge can help, psychology tells us that because of phenomena like proactive interference, what we have previously learned can sometimes also *hinder* our ability to learn new material. Many programmers surely have direct personal experience with this.

Though some prior research has examined subsequent-language learning in students (Section 2), this is the first work we know to systematically study this question where functional programming languages are involved. In addition, we are inspired by the works of Tshukudu and collaborators (discussed in detail in Section 2), who studied the consequences of syntactic and semantic similarity. We therefore set up two studies, one with syntactic difference and semantic similarity, and the other, vice versa.

A pure research experiment (such as a lab study) could invent languages that vary in precisely these ways. However, in an educational setting, we need comprehensive materials (such as IDEs, libraries, documentation, and textbooks) that an experimental language is unlikely to provide. As a

Authors' addresses: Kuang-Chen Lu, Department of Computer Science, Brown University, Providence, RI, United States of America, kuang-chen_lu@brown.edu; Shriram Krishnamurthi, Department of Computer Science, Brown University, Providence, RI, United States of America, shriram@brown.edu; Kathi Fisler, Department of Computer Science, Brown University, Providence, RI, United States of America, kathryn_fisler@brown.edu; Ethel Tshukudu, University of Botswana, Gaborone, Botswana, tshukudue@ub.ac.bw.




2475-1421/2023/8-ART215
https://doi.org/10.1145/3607857

compromise, we therefore use Pyret, which was expressly inspired [The Pyret Crew [n. d.]] by the syntax of Python but the semantics of mostly-functional languages like Racket and the ML family:

**syntactically different, semantically similar:** One study examined how college students exposed to Racket viewed Pyret (Section 4). There are almost no syntactic similarities between the Lispy parenthetical syntax of Racket and the infix syntax of Pyret, yet the semantics are almost identical: most Racket programs can trivially be transliterated to Pyret [Fisler et al. 2023]. We will refer to this as the *Racket-Pyret* study.

**syntactically similar, semantically different:** In another study, with differences in population and course details, college students transitioned from Pyret to Python (Section 5). This keeps the syntax fairly familiar but adds imperativity and state. We will refer to this as the *Pyret-Python* study.

Our work centers around the challenges of learning a new programming language. We identify features that do and don't cause students problems. In particular, we observe that the Racket-Pyret transition is quite seamless, but moving to Python introduces significant difficulties.

## 2 RELATED WORK

There was notable interest in the question of second (or subsequent) language learning in the 1980s and 1990s. Walker and Schach [1996] studied students who knew procedural languages transitioning to Ada. They noticed that when asked to solve programming problems in Ada, students preferred to use familiar syntactic constructs, such as `for`-loops, over new ones (in this case, the loop-exit-end constructs of Ada). Scholtz and Wiedenbeck [1990, 1991, 1992, 1993]; Wiedenbeck and Scholtz [1996] did a series of work on language transfer between procedural and object-oriented languages. Roughly speaking, their work emphasizes program construction and plans (program-construction knowledge that is irrelevant to syntactic details). They argued that plans are transferable between programming languages, that people tended to stick to their known plans, and that the success of transfer depends on how well the known plans aligned with the new languages. Kessler and Anderson [1986] found that novice programmers trained in iteration are more successful transitioning to recursion than vice versa. Wu and Anderson [1990] found that transfer in programming "was largely localized in writing the first drafts of programs".

These works differ from ours in two important ways. First, most of them did not focus on functional programming (though Wu and Anderson [1990] did use Lisp). Second, most have students write programs rather than read them. Arguably, if students cannot even read a program, it is unclear whether they can write them; at any rate, the link between program writing and reading is not clear in a functional setting (Fowler et al. [2022]; Lopez et al. [2008] suggest that the two are associated for small Python and Java programs, respectively). By instead focusing on reading, we can avoid some confounding factors: e.g., students writing programs "in the wild" may get help from teaching assistants, while in a lab study, they may feel nervous with a new language.

A series of papers by Tshukudu and co-authors [Tshukudu 2019; Tshukudu and Cutts 2020a,b; Tshukudu et al. 2021a,b; Tshukudu and Jensen 2020] and Tshukudu's dissertation [Tshukudu 2022] focused on the syntactic versus semantic split we mentioned in Section 1. They reported that a familiar syntax facilitates language transfer if the semantics is also similar, but hurts if the semantics is different. There are multiple notable differences between their work and ours:

(1) They focused entirely on procedural and object-oriented programming in Java, Python, C, and C++. The sets of features we consider—such as higher-order functions and implicit returns—cannot or do not arise in their settings.
(2) The underlying imperative semantics of their languages are quite similar. We focus on deeper semantic differences, such as functional versus imperative.

(3) Similarly, the syntactic differences between their languages are arguably much smaller than the difference between Pyret and the Lispy syntax of Racket (which seems to provoke a visceral reaction in many programmers).

For all these reasons, our work complements theirs and provides a richer sense of the comparison space. Indeed, we arrive at different conclusions (Section 6) than they do.

Santos et al. [2019] summarized their observations of students learning Java in a second course after a first course in Racket. When we compare their experience to ours, we find different sets of challenges. For instance, their students had difficulty going from Racket's **`struct`** to Java's **`class`**; our students had little difficulty going from Racket's **`struct`** to Pyret's algebraic datatypes. Their students also had trouble going from Racket's **`cond`** expressions to Java's **`if`**, while ours did not going from Pyret's multi-armed conditionals to Python's **`if`**. Interestingly, ours struggled with **`return`**, which they do not mention. Even more surprisingly, their paper does not discuss state. There are also two notable methodological differences to our work. First, they vary both the syntax and semantics significantly. Second, their work appears entirely anecdotal, and it is unclear what process they used to arrive at their findings.

Some other works, unlike ours, use observational studies rather than experimental ones. Denny et al. [2022] investigated reflections by students transitioning from MATLAB to C and reported that syntactic differences, the new (static) type checking, and new error messages were the most challenging. Shrestha et al. [2020] inspected hundreds of StackOverflow questions (involving 18 programming languages), interviewed 16 programmers, and concluded that prior languages commonly interfere with learning a new one. Bose et al. [2022] investigated StackOverflow posts about Julia made by people transitioning from various programming backgrounds to Julia and found that those new Julia programmers commonly "seek information on how they can accomplish tasks in Julia that they were previously able to do before with the language they are transitioning from", and that the new type systems and the new package system also impose many challenges.

Extensive work [Espinal et al. 2022; Franklin et al. 2016; Kölling et al. 2015; Moors et al. 2018; Powers et al. 2007; Weintrop 2019; Weintrop et al. 2018; Weintrop and Wilensky 2015, 2019] on transferring from (visual) blocked-based to textual languages has identified many challenges. Most of those challenges seem specific to block-based user interfaces, such as maintaining the integrity of expressions, memorizing available syntactic constructs, and a strong tendency to solve programming problems in a top-down approach. Most of these issues tie to the specific nature of blocks, which is outside our scope.

Readers may also find two surveys useful. Kao et al. [2022] reviews computing education findings on programming language transfer and the relevant literature from cognitive science. Tshukudu's dissertation [Tshukudu 2022] summarizes prior work on language transfer more broadly.

## 3 COMMON STUDY CONTEXT

Both studies were conducted in computer science courses at a selective private university in the USA. They were embedded in courses and part of the course pedagogy. We discuss the study designs and student populations in more detail alongside each study.

While Python and Racket are very large languages, students saw only limited aspects of them in the respective courses. In Racket, they only saw the restricted, functional subsets presented by language levels [Felleisen et al. 2018; Krishnamurthi 2001]. Although Pyret has stateful operators, these were not shown (and accidental uses of state produce errors). In Python, students saw traditional imperative programming, rather than more functional aspects like list comprehensions. Our studies focused only on parts of these languages that were germane to the surrounding courses.

```
fun g(l, n):
  cases (List) l:
    | empty => 0
    | link(f, r) =>
      (f * n) + g(r, n * 10)
  end
end

g([list: 2, 3, 5], 1)
g([list: 7, 8], 1)
```

```
(define (h m x)
  (if (empty? m)
      0
      (+ (* (first m) x)
         (h (rest m) (* x 10)))))

(h '(4 2 6) 1)
(h '(5 3) 1)
```

Fig. 1. Racket-Pyret: Survey 1: Program Pair (this example is **CaseList**)

*The study instruments are far too long to include in the paper. To aid reproducibility and interpretation, they are provided in full in the appendices. The paper contains excerpts so the reader can get a sense of the nature of the questions.*

## 4 STUDY 1: RACKET TO PYRET

### 4.1 Context for This Study

This study was part of an accelerated introductory course that students can place into. Placement involves four weeks of self-study (with questions answered on a help forum) in *How to Design Programs* [Felleisen et al. 2018], which uses Racket. Students covered chapters about basic data, lists, recursion on lists, and higher-order functions, submitting several graded programming exercises.

This course is one of multiple routes into computer science, and students could take a less demanding course if they wished. Therefore, some students did not complete the placement process. This leads to some self-selection: students uncomfortable with Racket or programming functionally may simply have dropped out of the class (and hence also the study).

The course had 70 students. Almost all (98%) had prior programming experience, typically in Java/C#/C++ (91%) or Python (76%). 87% were "not familiar at all" with Pyret, 10% "slightly familiar", and only one student "moderately familiar" (presumably because they looked ahead to see what the course would use). This programming background is typical for students taking advanced US secondary school computing courses. The demographic—largely male and White or Asian—mirrors this population. The background is likely to be a decent proxy for students who have had 1–2 years of computing at the secondary or tertiary level.

### 4.2 Study Design

After students had completed the Racket assignments, we gave them Survey 1, overviewed in Table 1. (Full programs are in Appendix A; an example is in Figure 1.) Students first predicted the output of Pyret programs (but Pyret was presented not by name but as a "hypothetical language" to make it unlikely students would just run the programs), then ones in Racket. Students were then given isomorphic *pairs* of Pyret-Racket programs (as in Figure 1) to compare, inspired by the large body of literature on contrasting cases [Gibson and Gibson 1955; Schwartz et al. 2011], which says that people often provide deeper analyses when asked to compare multiple related objects.

Table 1. Racket-Pyret: Survey 1: Overview

| | Question | Answer Format |
|---|---|---|
| Pyret/Racket only | What is the (first/second/third) result? *When the program produces multiple outputs, there is one question for each result.* | • I don't know.<br>• It's an error.<br>• It is NOT an error. (Please specify the expected result below.) |
| | (Pyret-only) How confident are you in your answer? | *Only shown if they did not answer "I don't know"*: (1-5) Not at all confident - Very confident |
| | (Pyret-only) If there are parts of this program that you find unclear, please describe them. | Text response |
| Pair comparison | In what ways do these programs seem similar and in what ways different? | Text response |
| | Do you find one of these programs clearer than the other? If so, why? | Text response |
| | You are welcome to comment on any other experiences (programming in other languages, other subjects, etc.) that influence how you understand these programs. | Text response |
| Background | What class year are you? | *Elided*: Various demographics relevant to the institution |
| | Do you have any programming background? (check all that apply) | Multiple-select: Web; Block languages; Python; Java, C#, and C++; Matlab, Stata, or R; Other |
| | How familiar are you with Pyret? | Likert: (1-5) Not familiar at all - Extremely familiar |

Survey 1 (shown in full in Appendix A) covered the following concepts in both languages. These were chosen to cover the main concepts students were already exposed to or might encounter. (**Names** link to the program pairs presented in the appendix.)

- variable naming (**Fun**, **Caret**, and **Hyphen**)
- variable shadowing (**ImpShadow** and **ExpShadow**)
- defining a variable twice in a block (**DefTwice**)
- referencing a variable before its definition (**EarlyUse**)
- multi-armed conditionals (**BasicCond** and **BadCond**)
- conditionals in Racket versus pattern-matching in Pyret (**CaseList**, **SymStr**, and **CaseStruct**)
- accessing list contents using selectors (Racket) versus fields (Pyret) (**FirstRest** and **DotFirst**)
- user-defined structures (Racket) versus algebraic data types (Pyret) (**Struct1**, **Struct2**, **CaseStruct**, and **DotField**)
- using selectors with higher-order functions (**DotField**)

- built-in testing constructs (**Check**)
- local variable definitions (**LocalDef**)

After Survey 1 was done, we wanted to address any misconceptions students had. There has long been a debate about whether the best way to fix misconceptions is to not talk about them (and thereby avoid reinforcing them) or to confront them directly. The research on refutation texts [Posner et al. 1982] shows that it is better to address them explicitly.

Following this principle, after students completed Survey 1, we provided a refutation text to correct any misconceptions they may have had. This document, shown in Appendix B, presents the program pairs they had just seen, points out common misunderstandings we had seen of the programs, and explains why the misunderstandings are wrong. The document's design is based on guidance given in a recent study [Weingartner and Masnick 2019], which shows the effectiveness of refutation texts in physics. The document also explains that the "hypothetical language" is Pyret, and tells them how they can experiment with it.

Two weeks later, we administered Survey 2 (Appendix C). The only programs on this survey were variants (e.g., altering details to avoid recall effects) of those *Pyret* programs that students did especially poorly on in Survey 1. Students were again asked to predict program outputs and to rate their confidence; the goal was to see whether the refutation text had helped and whether students had retained its information. Students did not have any lectures between the two rounds.

### 4.3 Results & Discussion

Our goal was to study how well students can understand a new language, Pyret, relative to their recent experience with Racket. Briefly: students had *almost no difficulty* with this transition. (Appendix D provides a detailed analysis.) As our Pyret-Python study yielded more interesting results, we report only on concepts that proved challenging in the Racket study.

Three challenging programs reflect syntactic oddities that may not generalize to other languages:

**variable naming** In Lispy syntax, variable names can contain carets: e.g., `c^d`. This is illegal in traditional infix syntaxes; in particular, `^` has a special meaning in Pyret (akin to the `|>` operator in OCaml). Students had difficulty with this, either assuming the name was valid or assuming `^` meant exponentiation (revealing their prior programming experience). In contrast, students did not struggle with Pyret's support for kebab-cased variable names (e.g., `my-account`). While these are illegal in many common languages (including Java and Python), students had used them extensively in Racket. This concept was not repeated in Survey 2.

**shadowing** Pyret is a lexically-scoped, block-structured language. However, its designers felt that implicit shadowing sometimes leads to bugs, so shadowing requires the use of a **`shadow`** keyword at the binding site. Most students could not intuit this in Survey 1; their understanding improved dramatically (18% → 90%) in Survey 2, presumably based on the refutation text.

**field access** To access, say, the `c` field from a list of structures that contain it, one might want to write `map(.c, ...)`. This "naked" accessor is a syntax error in Pyret; it must be preceded by a reference. Student performance on this improved only somewhat (13% → 44%) in Survey 2.

Of more interest are situations that might generalize to several languages and hence might benefit from clarification in manuals and instruction:

*Multi-arm conditionals.* In both Racket and Pyret, multi-armed conditionals are evaluated sequentially, first-to-last, and falling through produces an error. **BadCond** includes programs where multiple predicates match and none do. While students generally correctly predicted the behavior, they did so with low confidence and also had several predictions that were incorrect but reasonable.

*Variable redefinition.* **DefTwice** defines a variable twice in a block, which is invalid in both languages. Students' comments suggested that they were uncertain about whether redefinition would mutate the variable or cause an error. In Racket, 54% thought it will mutate, while 43% thought it would error (with 3% unsure). In Pyret, however, 87% predicted mutation and only 10% thought it would error (with 3% other answers). We conjecture this was caused by Pyret's syntactic similarity to prior imperative programming they had experienced, though we did not find a statistically clear influence (Appendix D).

In Survey 2, where students only predicted Pyret programs, the percentage who chose error climbed to 67%. Disappointingly, 28% *still* believed it would produce the answer dictated by mutation (with 4% other answers). This suggests a persistent misconception. Fortunately, this triggers a *syntax* error in Pyret, so students cannot *program* with this misunderstanding. Languages where redefinition silently takes one or the other value can likely lead to wrong programs that are very difficult for students to debug.

*Pattern-matching.* Pyret (unlike Racket) uses simple pattern-matching for algebraic datatypes. **CaseList** (Fig. 1) shows the first context in which students encounter it: the **cases** keyword starts a pattern-match, while `empty` and `link` are the two list constructors. While four students were unclear on the meaning of **cases**, most (26 out of 30) students who answered the any-parts-unclear question were confused about `link`. Students' comments suggested that they attempted to (correctly) read the **cases** expression as a conditional, but the `link` part as a *function call*. Some were then confused by the presence of multiple names after `link` (both `f` and `r`, neither of which is the list, `l`). Despite this, however, about two-thirds of students (67% and 66% for the first and second outputs respectively) were able to intuit the intended semantics and correctly predict the answers for the programs.

Conventional wisdom assumes the intuitiveness of pattern-matching. This example suggests that this may need to be more closely evaluated. One possible cause for confusion here is that, in their prior Racket experience, students decomposed lists using conditionals with explicit predicates (like `empty?`) applied to the data to distinguish the variants. They may have tried to apply a similar interpretation here, even though it does not quite fit (due to the variables that would be unbound by that interpretation). Interestingly, a later program (**CaseStruct**) used pattern-matching over a datatype *defined* in the same program. Students predicted better (both outputs have a 93% correct rate), and with slightly more confidence, perhaps because they saw the datatype definition.

## 5 STUDY 2: PYRET TO PYTHON

### 5.1 Context for This Study

Unlike in the previous study, this one was run in an introductory course designed for students with no prior experience; it has a reputation of being gentle in pace. The course followed a data-centric viewpoint [Krishnamurthi and Fisler 2020]: it started with functional programming over images and tables, then covered algebraic datatypes, including lists and trees. This part was done in Pyret. Two-thirds of the way into the 3-month course, instruction switched to Python, covering programming with state, dictionaries, and data analysis with Pandas.

170 of the 186 who finished the course had submitted background surveys at the start of the semester. Only 75 of the 170 reported having any prior programming experience. Forty of these had used Python (others had varied background from Scratch through Java and C++). Among those with prior Python experience, 10 reported being self-taught while 17 took an Advanced Placement (AP in the USA), International Baccalaureate, or other college-level class. The others varied from school coding clubs to other (pre-AP) classes in middle school or high school. Approximately 45% were female-presenting and about 15–20% appeared to be US racial minorities.

### 5.2 Broad Study Parameters

As in the previous study, we used reading tasks to assess students' expectations of program behavior in a new language (here, Python). We conducted this study in four phases. The first three phases were administered within the course's twice-weekly drills on lecture content. Drills were graded on participation rather than correctness, yet they counted towards the final grade, so most students did them.

Our initial set of programs was inspired by two sources: questions students asked on the course help forum during previous iterations of the same course, and other differences between the languages that we knew students would encounter given the course lectures and assignments (the labels **P1.n** and **P2.n** reference Phases 1 and 2, respectively, along with the problem number):

(1) Python introduces mutable lists (**P2.12**)
(2) Python doesn't check "type" annotations (**P1.6**, **P1.7**, **P2.5**, and **P2.6**)
(3) Python introduces `return` (**P1.1**, **P1.3**, and **P2.1**)
(4) Python introduces `print` (**P1.2** and **P2.2**)
(5) Python allows redefinition of variables (**P1.10** and **P2.8**)
(6) whitespace indentation matters in Python (**P1.4**, **P1.5**, **P2.3**, and **P2.4**)
(7) `if` is an expression in Pyret and statement in Python (**P1.8**, **P1.9**, and **P2.7**)
(8) Python overloads operators like `*` (**P1.6**, **P1.7**, **P2.5**, and **P2.6**)
(9) Python has scoping vagaries [Guth 2013; Politz et al. 2013], like lifting bindings (**P1.9**)
(10) Python introduces `for`-loops, which can be confusing when combined with `return` and `print` (**P2.9**, **P2.10**, and **P2.11**)

*The full programs (Table 5) and other parts of Phases 1 & 2 are provided in Appendix E. We provide representative excerpts in the paper.*

### 5.3 Phase 1

Before the course transitioned to Python, the professor informed students that they wanted to gather data to guide subsequent lecture design. They were asked to predict the behavior of Python programs prior to learning the language in class. Questions were presented in the form shown in Figure 2, where students were given multiple choices for what appears in the space indicated by ??????. (This format, with a REPL interaction, was chosen intentionally to make it possible to cover both printed output and returned values.)

```
>>> x = 2
>>> def get_x():
...       return x
...
>>> get_x()
2
>>> x = 3
??????
>>> get_x()
??????
>>>
```

Fig. 2. Pyret-Python: Phase 1 (this example is **P1.10**)

In the first phase, students were given 10 programs. They could choose multiple answers, but if they did so, they were asked to explain why they were uncertain. 171 students submitted responses. Most (84% and 81%) students correctly answered **P1.3** (about basic use of `return`) and **P1.9** (about lifting definitions within `if`). More than half generally predicted the effect of redefinition (Figure 2) correctly: 68% predicted that redefinition produces nothing; 59% predicted the consequence of the variable updating. About half (54%) answered both parts correctly. However, for the other 7 programs, fewer than 25% of students predicted the correct answer.

Students with some Python background were more likely to give correct answers to almost all programs. (See Appendix G for statistical analysis.) However, even those students did poorly on the 7 difficult questions: no more than 30% answered any one of those correctly.

The concept of redefinition also appears in the Racket-Pyret study (**DefTwice**). We compare the two studies on this aspect at the end of the next subsection.

### 5.4 Phase 2

Between Phase 1 and Phase 2, students were given three lectures on how to translate Pyret programs they had written into Python. The lectures were informed by the above problems. The second phase used the same format as the first, this time with 12 programs. Eight were variants of the programs on which students performed poorly in the first phase. The other four covered `for`-loops, variable mutation, and list mutation. `for`-loops had been covered in lecture, as had dataclasses and mutable fields; the course had not yet covered mutation to lists or variables.

*Results.* 148 students submitted responses. Three observations reflected cases that we expected students to improve on with practice:

- Students strongly (71% students in Phase 1 and 57% in Phase 2) expected type annotations to be checked, i.e., a program that violates an annotation should fail with an error. Pyret can be run with or without static type-checking; either way, at least the top-level of an annotation is enforced (if only dynamically). Thus, students may have expected to see the similar syntax in Python have similar behavior. Python, however, does not check annotations!
- When a variable is seemingly[1] introduced inside an `if` and used after the `if`-block, most (80%) students expected it to be defined at use. This is true in Python, though not in Pyret. This could, however, reflect a dynamic scoping misconception [Fisler et al. 2017].
- Students improved (**P1.10→P2.8**) their understanding of redefinition: more students (68% → 81%) predicted that variable redefinition is valid; and more students (59% → 75%) predicted that redefinition changes the value of variables referred to by a previously defined function. Note that whether redefinition is predicted to be valid varies in student populations and in languages. In Survey 1 of Racket-Pyret, 54% students predicted that redefinition is valid in Racket, and 87% students predicted so in Pyret (**DefTwice**). The Racket-Pyret programs do not, however, include functions.

Other issues became the subjects of programs in the remaining phases, as described below.

### 5.5 Phase 3

Given that students had failed to improve significantly on many common questions between Phases 1 and 2, we designed the third phase to use a different question format: each question gave a potential answer that the program would *not* produce and asked students to "explain the problem as best as you can, as if you were helping your friend debug their code" (see Figure 3 for an example). This phase tested three programs on which students had performed poorly in Phase 1 and Phase 2 (see Figure 4). Between Phases 2 and 3, students had four more lectures (covering mutable variables, mutable lists, and aliasing), and a homework assignment on Python programming.

*Results.* The nature of the prompt meant we could analyze student responses along two dimensions: whether their explanation was correct and whether any fix they proposed (which was often given though not required) would actually address the problem.

We randomly sampled 30 of the 137 responses for detailed analysis. Two authors iteratively arrived at codes [Richards and Hemphill 2018] with perfect agreement, reflecting the above dimensions, shown in Appendix F. Afterwards, one of these authors eyeballed the remaining 107 responses to confirm that we had covered the set of answers.

[1] Python lifts the variable introduction to the top of the function block, so it is actually defined for the entire function.

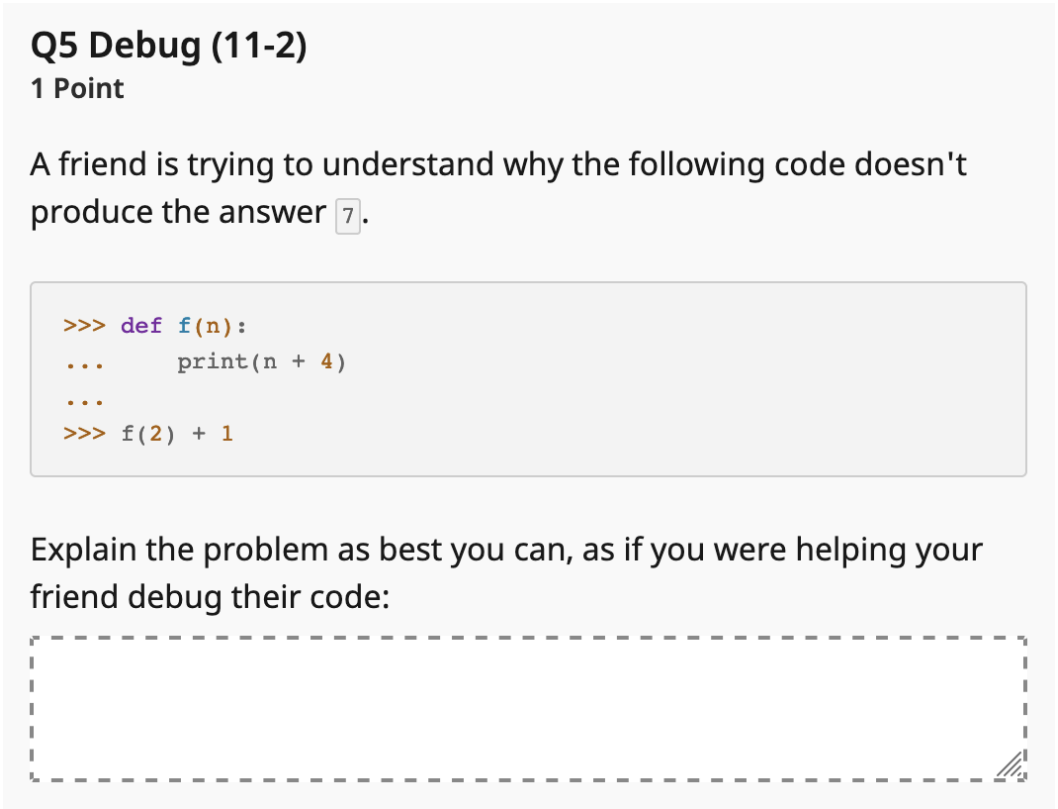


Fig. 3. Pyret-Python: Phase 3: Question (this example is program (A) from Figure 4)

(A) desired answer: 7

```
>>> def f(n):
...     print(n + 4)
...
>>> f(2) + 1
```

(B) desired answer: 5

```
>>> def g(x):
...     return
...     x - 10
...
>>> g(15)
```

(C) desired answer: 24

```
>>> def f(n, x):
...     v = n + x
... return v + v
>>> f(7, 5)
```

Fig. 4. Pyret-Python: Phase 3: Programs

We discuss the three programs (Figure 4) individually:

**(A)** The problem here is confusing **`print`** with **`return`**. Most (about 86%) students correctly observed that a return is needed in the function. Only about 43% students provided enough detail in their explanations for us to infer their mental models of **`print`**. Unfortunately, most of these mental models are at least subtly wrong. Only about 7% correctly observed that **`print`** only displays an answer and does not return a value, and because `f` lacks a **`return`**, it will return `None`. About 20% claimed that the function will return the value of the **`print`**, which is `None`. Note that this is subtly incorrect, and could cause confusion in other contexts.

(A)

```
a = 2 + 3
return a
```

(B)

```
def f(a, b):
    s = a + b
f(2, 3)
print(s)
```

(C)

```
def word_double(word: str):
    result = ''
    for letter in word:
        result = result + letter + letter
        return result
print(word_double('exam'))
```

Fig. 5. Pyret-Python: Final Exam: Relevant Questions

About 7% were much farther off, claiming that the problem was that `f` would return the value of **`print`**, which was a *string*, which could not be added to a number. About 10% effectively believed **`print`** would terminate the program! Students who provided incorrect explanations suggested a long tail of other misconceptions.

**(B)** Almost all (90%) students correctly surmised that the **`return`** on its own would return `None`, which is correct; they also suggested the correct repair. However, this masks a misconception: about 7% incorrectly believed that `x - 10` would evaluate, but its value would be ignored.

**(C)** This program produced a wide range of answers that are hard to group accurately. About 10% said they were unsure why the program doesn't work as desired. Superficially, 57% correctly noted that the program has an indentation error. However, there were (also) numerous explanations that suggested a variety of misconceptions. Some believed the function ended at the assignment to `v`, and that the top-level **`return`** is syntactically valid. Some believed the function would return `12`; some an error because a function can't "return" a *definition*. (It is noteworthy that a corresponding Pyret function would give a syntax error because the function has no body expression.) Many students believed the problem is that the call to `f` should be surrounded by a **`print`**. Some also believed the **`return`** should be replaced with **`print`**. Most surprising of all, some students believed a problem is that `v` cannot be used twice, revealing an odd "linearity" misconception not previously seen in the literature. (Perhaps these are budding Rust programmers?)

### 5.6 Phase 4

Phase 4 was part of the final exam. Between Phase 3 and the exam, students had six additional lectures (on dictionaries and Pandas). They also completed a larger programming project (again, using dictionaries and Pandas) prior to taking the final.

Final exam time limitations meant we could not exercise all the preceding topics. We therefore tested three programs that further investigate the most persistent issues we observed from the earlier phases; the exam programs are in Figure 5. The exam asked students to provide exactly one result for each program (free-form, not multiple choice), providing either an answer, "nothing" (for `None`), or an error (with a brief description of the problem).

*Results.* We now discuss each of the final exam questions in turn:

**(A)** This program is erroneous: the top-level cannot **`return`**. We added this new program because some wrong answers in earlier phases suggested that many students hadn't realized that **`return`** is associated with functions. Indeed, 53% responded with `5` and 26% with "nothing". Only 23% said it would produce an error. 85% of them identified the correct problem.

**(B)** We added this new program because prior research [Fisler et al. 2017] as well as comments on multiple earlier programs suggested the possibility of a dynamic scope misconception. Indeed, 19% of students thought this would produce 5. 74% correctly identified that this program has an error, of whom 85% identified the right problem (that `s` is not in scope). There were also a handful of arguably fanciful semantic interpretations such as: **`print`** would work if `s` had been returned. This confusion could be caused by the sloppy use of words and incorrectly suggestive syntax: e.g., when we write **`return`** `s`, Python returns the *value* of `s`, not the *variable*.

**(C)** This program will run without error, but double only the first letter ("ee"). Only 17% correctly predicted this output. Arguably, the name of the function suggests this is incorrect: that the whole *word* should be "doubled". By that interpretation, the function arguably contains a "premature" **`return`**. Indeed, 33% expected the value that would be returned if the **`return`** were out-dented by one level, "eexxaamm", while 7% expected "examexam". Many students expected this program to error: because a **`for`**-loop cannot iterate over a string, `letter` is undefined, `+` does not work over strings, etc. Most interestingly, 8% thought it would produce the error that the **`return`** was incorrectly indented! One reason might be that they were accustomed to only certain patterns of where **`return`** occurs, and expected a checker for those. More intriguingly, students may have noticed the mismatch between the function's name and its code. While this may seem fanciful, Roy Pea has documented such phenomena, which he calls "superbugs" [Pea 1986]: "the idea that there is a *hidden mind* somewhere in the programming language that has intelligent, interpretive powers" (emphasis in original).

Overall, we see that even after a month of Python education and practice, students have difficulty with certain rudimentary aspects of Python, particularly surrounding **`return`**. Students seem to expect automatic **`return`**s (as in Pyret) where Python expects them to write one explicitly. This questions whether the approach taken by most functional languages—to always return the last value—may be a more predictable and less error-prone design.

In addition, we see enormous confusion between *returning* and *printing*. As we noted, some students even think that **`print`** prints and then *halts* the program, though many more think it would both print *and* return a value. (Incidentally, the latter is what Pyret's `print` function—which students were not shown—does, which also makes it useful for ad hoc debugging.) In many traditional imperative curricula, functions by novices rarely return values, which may sidestep this. However, this of course leads to non-compositional code. In contrast, most functional curricula sidestep this problem in the other way, by focusing on returning and composing values rather than printing them. Given that modern programming libraries are full of value-returning functions as well as higher-order functions, this may well be a much better foundation.

Inasmuch as a language wants to support both printing and returning, it seems important that these outputs be noticeably distinct. The DrRacket IDE, for instance, uses different colors for printed output and returned values. No study has examined whether the difference is salient to users, but it would appear to be a step in the right direction. Pyret uses a slightly different design: because printing is often used as a debugging device, beyond `print`, it provides a special syntactic construct, **`spy`**. The output from this is presented graphically, with a reference back to the source location, and "spied" values are printed using the value-renderer, enabling user interaction (e.g., expanding and collapsing values, changing numeric representations, etc.). Again, we are not aware of formal studies on this user interface, but it suggests another design for avoiding this confusion.

## 6 DISCUSSION & CONCLUSION

*Swapping Language Order.* It would be interesting to consider swapping language order. However, pedagogically, we felt it would not make sense for us to put Python before Pyret:

(1) Pyret offers lightweight support for images, tables, etc., while Python has libraries needed for large-scale data processing. Therefore, there is a natural transition from "pleasant language designed for smaller data" to "industrial language that handles large data". Going in the other direction made less sense for our use.
(2) We feel (and the data don't contradict this) that starting functionally is simpler, especially for data processing.
(3) If we start with Python, that gives a significant leg up (at least in their perception) to students with Python experience. This can cause those students to show off in class, students without that background to drop out, etc. (The impact would likely be disproportionately on underrepresented students.) Pyret, whatever its other virtues, at least offers a more level playing field.

*Controlled Experiments.* This paper is an experience report, not a report on controlled experiment. There are real difficulties to performing them:

(1) Those controls are not easy to obtain in our setting. For instance, it is impossible to meaningfully A/B test in a class with one section (since there would be too much communication between the populations).
    We did not administer language pre-tests because doing so would have caused some students to think they needed to know that language at the start, and less confident students would have dropped the classes (and these would disproportionately be from underrepresented backgrounds). This is especially true for the Pyret-Python study, where students are aware that Python is sometimes taught in secondary schools, and may therefore have felt they could only take the course if they already knew some of it. As educators, we followed a Hippocratic principle, sacrificing the knowledge gained from a pre-/post-test in return for reaching the most students.
(2) The alternative is a "lab study", where no subject's education/career is on the line, and we can control everything. Doing this is much harder than it might appear. First, getting participants to attend studies of more than 30-45 minutes is difficult. However, very little learning can happen in that time. Second, it is difficult to get participants to return to follow-up sessions. Without it, we can't measure long-term learning, nor can we design a reactive study (design new phases based on the outcome of earlier phases). For these and other reasons, a controlled version of our Pyret-Python study would be extremely difficult (and would lack ecological validity).

Rather, we believe that our study designs, instruments, and initial findings are still very useful for people to create variants for their settings. Over time we will hopefully build a modern body of knowledge on this topic.

*Relating to Transfer.* Our Racket-Pyret study stands in interesting contrast to Tshukudu [2022], who argues that there should be little or no transfer when the syntax is different. We have refrained from making strong statements about *transfer* because students' prior knowledge could be a factor. In either case, we believe there is a need for a richer framework for talking about these phenomena. In particular, irrespective of our "expert" judgment on similarity and difference, if students *perceive* languages to be similar, irrespective of syntax, they can construct analogies between them and carry ideas across using these analogies, in the style suggested by Gentner [1983] and others.

In both studies, most new concepts are unsurprisingly challenging for students. In the Racket-Pyret study, the challenging new concepts tend to be somewhat obscure issues that would not arise often. In contrast, some new concepts like pattern-matching over algebraic data types do *not* seem particularly challenging. It is, of course, possible that students are using contextual clues to make sense of the program as a whole. Nevertheless, these suffice to make accurate predictions.

The Pyret-Python study involves many new concepts in Python: statements, explicit returning, printing, and mutation. Almost all these concepts are challenging. In our simple examples, variable redefinition and some lifting of variable definitions are not challenging, but if we were to use these features in more and different ways, students may not perform as well; prior research documents difficulties with aliasing and mutation even for more advanced students [Fisler et al. 2017].

*Prior Student Experience.* One important difference between our two studies is that many of the Racket-Pyret students had prior programming experience. It is possible that that experience is what made the transition to Pyret easy. (Indeed, they may have viewed Pyret through a Pythonic lens and thus understood it easily.) With our student populations, it was difficult to control for this. However, other institutions that use similar languages (e.g., Racket combined with a more traditional infix language, whether Pyret or OCaml or the like) would be in a good position to study something closer to our Pyret-Python condition.

More broadly, any attempt to view post-secondary school students as a "tabula rasa" will increasingly fail in many educational systems. Students around the world are being exposed to computing in schools from an early age. Indeed, our project originally set out to measure *transfer* effects from one language to the other. This is much easier to do when students have no prior programming. However, students in various places now have a little or even a lot of programming background before tertiary education, typically of an imperative nature in Scratch [Maloney et al. 2010; Resnick et al. 2009] or Python. We thus have to take into account the sum of all prior influences.

*Conclusion.* Overall, we believe our work adds useful new data to the literature on learning new languages. First, it introduces these ideas to functional programming educators, providing some preliminary methods about how we might go about investigating the issues that arise. Second, it provides concrete data about pairs of potentially-interesting languages. Third, it seemingly contradicts prior work, which suggests that far more study is necessary. Finally, it provides evidence for difficulties in Python that are not present in more functional languages.

## ACKNOWLEDGMENTS

We are extremely grateful to our reviewers for their close reading and many questions and suggestions. We thank the US NSF for support under Grant No.: 2227863. The authors gratefully acknowledge Schloss Dagstuhl for Seminar 22302, through which the second and third authors first met the fourth. The idea for this paper came about over conversations during breaks at an outside table at the Schloss.

## REFERENCES

Dibyendu Brinto Bose, Gerald C Gannod, Akond Rahman, and Kaitlyn Cottrell. 2022. What Questions Do Developers Ask about Julia?. In *Proceedings of the 2022 ACM Southeast Conference (ACM SE '22)*. Association for Computing Machinery, New York, NY, USA, 224–228. https://doi.org/10.1145/3476883.3520205

Paul Denny, Brett A. Becker, Nigel Bosch, James Prather, Brent Reeves, and Jacqueline Whalley. 2022. Novice Reflections During the Transition to a New Programming Language. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1 (SIGCSE 2022, Vol. 1)*. Association for Computing Machinery, New York, NY, USA, 948–954. https://doi.org/10.1145/3478431.3499314

Alejandro Espinal, Camilo Vieira, and Valeria Guerrero-Bequis. 2022. Student Ability and Difficulties with Transfer from a Block-Based Programming Language into Other Programming Languages: A Case Study in Colombia. *Computer Science Education* (2022), 1–33. https://doi.org/10.1080/08993408.2022.2079867

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing* (2 ed.). MIT Press. https://htdp.org/

Kathi Fisler, Shriram Krishnamurthi, Benjamin S. Lerner, and Joe Gibbs Politz. 2023. *A Data-Centric Introduction to Computing* (2023-02-21 ed.). https://dcic-world.org/

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 (SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 213–218. https://doi.org/10.1145/3017680.3017777

Max Fowler, David H. Smith IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. 2022. Reevaluating the Relationship between Explaining, Tracing, and Writing Skills in CS1 in a Replication Study. *Computer Science Education* 32, 3 (July 2022), 355–383. https://doi.org/10.1080/08993408.2022.2079866

Diana Franklin, Charlotte Hill, Hilary A. Dwyer, Alexandria K. Hansen, Ashley Iveland, and Danielle B. Harlow. 2016. Initialization in Scratch: Seeking Knowledge Transfer. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 217–222. https://doi.org/10.1145/2839509.2844569

Dedre Gentner. 1983. Structure-Mapping: A Theoretical Framework for Analogy. *Cognitive Science* 7, 2 (1983), 155–170. https://doi.org/10.1016/S0364-0213(83)80009-3

James J. Gibson and Eleanor J. Gibson. 1955. Perceptual Learning: Differentiation or Enrichment? *Psychological Review* 62, 1 (1955), 32–41. https://doi.org/10.1037/h0048826

Dwight Guth. 2013. *A Formal Semantics of Python 3.3*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign. https://hdl.handle.net/2142/45275

Yvonne Kao, Bryan Matlen, and David Weintrop. 2022. From One Language to the Next: Applications of Analogical Transfer for Programming Education. *ACM Transactions on Computing Education* 22, 4 (2022), 1–21. https://doi.org/10.1145/3487051

Claudius M. Kessler and John R. Anderson. 1986. Learning Flow of Control: Recursive and Iterative Procedures. *Human–Computer Interaction* 2, 2 (June 1986), 135–166. https://doi.org/10.1207/s15327051hci0202_2

Michael Kölling, Neil CC 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 (WiPSCE '15)*. Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/2818314.2818331

Shriram Krishnamurthi. 2001. *Linguistic Reuse*. Ph. D. Dissertation. Rice University, United States – Texas. https://scholarship.rice.edu/handle/1911/17993

Shriram Krishnamurthi and Kathi Fisler. 2020. Data-Centricity: A Challenge and Opportunity for Computing Education. *Commun. ACM* 63, 8 (July 2020), 24–26. https://doi.org/10.1145/3408056

Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/1404520.1404531

John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15. https://doi.org/10.1145/1868358.1868363

Luke Moors, Andrew Luxton-Reilly, and Paul Denny. 2018. Transitioning from Block-Based to Text-Based Programming Languages. In *2018 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. IEEE, 57–64. https://doi.org/10.1109/LaTICE.2018.000-5

Roy D. Pea. 1986. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of educational computing research* 2, 1 (1986), 25–36. https://doi.org/10.2190/689T-1R2A-X4W4-29J2

Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 217–232. https://doi.org/10.1145/2509136.2509536

George J. Posner, Kenneth A. Strike, Peter W. Hewson, and William A. Gertzog. 1982. Accommodation of a Scientific Conception: Toward a Theory of Conceptual Change. *Science Education* 66, 2 (1982), 211–227. https://doi.org/10.1002/sce.3730660207

Kris Powers, Stacey Ecott, and Leanne M. Hirshfield. 2007. Through the Looking Glass: Teaching CS0 with Alice. *ACM SIGCSE Bulletin* 39, 1 (March 2007), 213–217. https://doi.org/10.1145/1227504.1227386

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, and Brian Silverman. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (2009), 60–67. https://doi.org/10.1145/1592761.1592779

K. Andrew R. Richards and Michael A. Hemphill. 2018. A Practical Guide to Collaborative Qualitative Data Analysis. *Journal of Teaching in Physical education* 37, 2 (2018), 225–231. https://doi.org/10.1123/jtpe.2017-0084

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)*. Association for Computing Machinery, New York, NY, USA, 36–40. https://doi.org/10.1145/3358711.3361628

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. https://doi.org/10.1080/10447319009525970

Jean Scholtz and Susan Wiedenbeck. 1991. Learning a New Programming Language: A Model of the Planning Process. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, Vol. 2. IEEE, 3–12. https://doi.org/10.1109/HICSS.1991.183956

Jean Scholtz and Susan Wiedenbeck. 1992. An Analysis of Novice Programmers Leaming a Second Language. In *Empirical Studies of Programmers: Fifth Workshop (PPIG 1992)*. 187–205. https://ppig.org/papers/1992-ppig-4th-scholtz/

Jean Scholtz and Susan Wiedenbeck. 1993. Using Unfamiliar Programming Languages: The Effects on Expertise. *Interacting with Computers* 5, 1 (March 1993), 13–30. https://doi.org/10.1016/0953-5438(93)90023-M

Daniel L. Schwartz, Catherine C. Chase, Marily A. Oppezzo, and Doris B. Chin. 2011. Practicing versus Inventing with Contrasting Cases: The Effects of Telling First on Learning and Transfer. *Journal of educational psychology* 103, 4 (2011), 759. https://doi.org/10.1037/a0025140

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 (ICSE '20)*. Association for Computing Machinery, 691–701. https://doi.org/10.1145/3377811.3380352

The Pyret Crew. [n. d.]. The Pyret Programming Language. http://pyret.org

Ethel Tshukudu. 2019. Towards a Model of Conceptual Transfer for Students Learning New Programming Languages. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. Association for Computing Machinery, New York, NY, USA, 355–356. https://doi.org/10.1145/3291279.3339437

Ethel Tshukudu. 2022. *Understanding Conceptual Transfer in Students Learning a New Programming Language*. Ph. D. Dissertation. University of Glasgow. https://theses.gla.ac.uk/82984/

Ethel Tshukudu and Quintin Cutts. 2020a. 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 (ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 307–313. https://doi.org/10.1145/3341525.3387406

Ethel Tshukudu and Quintin Cutts. 2020b. Understanding Conceptual Transfer for Students Learning New Programming Languages. In *Proceedings of the 2020 ACM Conference on International Computing Education Research (ICER '20)*. Association for Computing Machinery, New York, NY, USA, 227–237. https://doi.org/10.1145/3372782.3406270

Ethel Tshukudu, Quintin Cutts, and Mary Ellen Foster. 2021a. Evaluating a Pedagogy for Improving Conceptual Transfer and Understanding in a Second Programming Language Learning Context. In *Proceedings of the 21st Koli Calling International Conference on Computing Education Research (Koli Calling '21)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3488042.3488050

Ethel Tshukudu, Quintin Cutts, Olivier Goletti, Alaaeddin Swidan, and Felienne Hermans. 2021b. Teachers' Views and Experiences on Teaching Second and Subsequent Programming Languages. In *Proceedings of the 17th ACM Conference on International Computing Education Research (ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 294–305. https://doi.org/10.1145/3446871.3469752

Ethel Tshukudu and Siri Annethe Moe Jensen. 2020. The Role of Explicit Instruction on Students Learning Their Second Programming Language. In *United Kingdom & Ireland Computing Education Research Conference. (UKICER '20)*. Association for Computing Machinery, New York, NY, USA, 10–16. https://doi.org/10.1145/3416465.3416475

Karen P. Walker and Stephen R. Schach. 1996. Obstacles to Learning a Second Programming Language: An Empirical Study. *Computer Science Education* 7, 1 (Jan. 1996), 1–20. https://doi.org/10.1080/0899340960070101

Kristin M. Weingartner and Amy M. Masnick. 2019. Refutation Texts: Implying the Refutation of a Scientific Misconception Can Facilitate Knowledge Revision. *Contemporary Educational Psychology* 58 (July 2019), 138–148. https://doi.org/10.1016/j.cedpsych.2019.03.004

David Weintrop. 2019. Block-Based Programming in Computer Science Education. *Commun. ACM* 62, 8 (July 2019), 22–25. https://doi.org/10.1145/3341221

David Weintrop, Alexandria K. Hansen, Danielle B. Harlow, and Diana Franklin. 2018. Starting from Scratch: Outcomes of Early Computer Science Learning Experiences and Implications for What Comes Next. *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Aug. 2018), 142–150. https://doi.org/10.1145/3230977.3230988

David Weintrop and Uri Wilensky. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*. Association for Computing Machinery, New York, NY, USA, 101–110. https://doi.org/10.1145/2787622.2787721

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. https://doi.org/10.1016/j.compedu.2019.103646

Susan Wiedenbeck and Jean Scholtz. 1996. Adaptation of Programming Plans in Transfer between Programming Languages: A Developmental Approach. In *Empirical Studies of Programmers: Sixth Workshop*. Ablex Norwood, NJ, 233–253.

Quanfeng Wu and John R. Anderson. 1990. *Problem-Solving Transfer Among Programming Languages*. Technical Report. Carnegie Mellon University. https://apps.dtic.mil/sti/citations/ADA225798

Received 2023-03-01; accepted 2023-06-27