Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?

Nischal Shrestha
NC State University
Raleigh, North Carolina
nshrest@ncsu.edu

Colton Botta
NC State University
Raleigh, North Carolina
cgbotta@ncsu.edu

Titus Barik
Microsoft
Redmond, Washington
titus.barik@microsoft.com

Chris Parnin
NC State University
Raleigh, North Carolina
cjparnin@ncsu.edu

ABSTRACT

Once a programmer knows one language, they can leverage concepts and knowledge already learned, and easily pick up another programming language. But is that always the case? To understand if programmers have difficulty learning additional programming languages, we conducted an empirical study of Stack Overflow questions across 18 different programming languages. We hypothesized that previous knowledge could potentially interfere with learning a new programming language. From our inspection of 450 Stack Overflow questions, we found 276 instances of interference that occurred due to faulty assumptions originating from knowledge about a different language. To understand why these difficulties occurred, we conducted semi-structured interviews with 16 professional programmers. The interviews revealed that programmers make failed attempts to relate a new programming language with what they already know. Our findings inform design implications for technical authors, toolsmiths, and language designers, such as designing documentation and automated tools that reduce interference, anticipating uncommon language transitions during language design, and welcoming programmers not just into a language, but its entire ecosystem.

CCS CONCEPTS

• **Human-centered computing** → **Empirical studies in HCI**;
• **Software and its engineering** → **Programming teams**.

KEYWORDS

interference theory, learning, program comprehension, programming environments, programming languages

ACM Reference Format:
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 *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3377811.3380352

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380352

PRELUDE

Peter Norvig wrote a guide, "Python for Lisp Programmers" [48], to teach Python from the perspective of Lisp. We interviewed Peter regarding this transition and he described a few challenging aspects of switching to Python, such as how lists are not treated as a linked list and solutions where he previously used macros required rethinking. When asked about the general problem of switching programming languages, he said:

> *Most research is on beginners learning languages. For experts, it's quite different and we don't know that process. We just sort of assume if you're an expert you don't need any help. But I think that's not true! I've only had a couple times when I had to deal with C++ and I always felt like I was lost. It's got all these weird conventions going on. There's no easy way to be an expert at it and I've never found a good answer to that and never felt confident in my C++.*

Peter believes that learning new languages is difficult—even for experts—despite their previous experience working with languages. Is Peter right?

1 INTRODUCTION

Numerous stories on language transitions suggest that even experienced programmers have difficulty learning new languages. For example, a Java programmer who transitioned to Kotlin [68] reports that differences like reversed type notation and how classes in Kotlin are final by default, made the transition less smooth than expected: "if you think that you can learn Kotlin quickly because you already know Java—you are wrong. Kotlin would throw you in the deep end." Similarly, a programmer experienced in C++ who switched to Rust [15] found that Rust's borrow checker, "forces a programmer to think differently." Transitions across radically different languages are especially difficult. For example, a Java programmer switched to Haskell [25] and expressed that "the easy things are often a bit harder to do in Haskell," and another programmer [58] experienced in procedural languages warned that "[lazy evaluation] can be a bit confusing to understand *how* it works in practice especially if you're still thinking like an imperative programmer." Even languages sharing the same runtime can be problematic: "whenever I pick up CoffeeScript, I feel as if most of my understanding of JavaScript suddenly vanishes into thin air." [50] From these stories, one common refrain occurs: previous programming knowledge is sometimes less helpful than expected, and can actively interfere with learning. This seems counterintuitive. Why can previous knowledge actually make learning harder and not easier?

In psychology and neuroscience, studies have shown that confusion can occur when older information interacts with newer information [12, 35, 36, 52, 53, 67]. To illustrate, suppose the bread aisle of your favorite store was recently moved. You may reflexively start walking towards the old location due to *interference*—when previous knowledge disrupts recall of newly learned information. However, if you recently saw that the impossible burger was added to the frozen section (and not a separate health aisle), using knowledge that frozen food can be found in the frozen section is an example of *facilitation* [11]—when previous knowledge helps retrieval of new information. In the same vein, when a Java programmer is learning Kotlin, we postulate that their prior Java knowledge either facilitates or interferes with learning. The knowledge that Java is objected-oriented and uses static typing facilitates their learning as Kotlin shares similar properties. The knowledge that Java classes are not `final` by default interferes with their learning because Kotlin classes are `final` by default. If previous programming knowledge can be framed as a source of interference with new programming language acquisition, interference theory can explain why programming language learning can be difficult for experienced programmers. And when previous programming knowledge isn't relevant, learning can also be difficult because this knowledge doesn't facilitate.

To investigate our hypothesis, we first looked for evidence that programmers could have difficulty learning another language due to interference from their previous knowledge. To this end, we conducted an empirical study examining questions posted on a popular question-and-answer site, Stack Overflow.[1] We analyzed 450 posts for 18 different programming languages and qualitatively coded each post, characterizing posts in terms of whether or not programmers made incorrect assumptions based on their previous programming knowledge. Then, to understand what learning strategies programmers used when learning another language—and why previous knowledge could interfere with this process—we interviewed 16 professional programmers who had recently switched to a new programming language.

We found that:

- Cross-language interference is a problem: 276 (61%) cross-language posts on Stack Overflow contained incorrect assumptions due to interference with previous language knowledge.
- Based on our interviews, professional programmers primarily learned new languages on their own, using an opportunistic strategy that often involved relating the new language to previous language knowledge; however, this results in interference which harms their learning.
- Learning a new language involves breaking down old habits, shifting one's mindset, dealing with little-to-no mapping to previous languages, searching for proper documentation, and retooling in a new environment. All together, these challenges make learning another language difficult.

2 METHODOLOGY

To explore how programmers learn a new language, and understand their potential sources of confusion, we conducted a mixed-methods study through an empirical investigation of Stack Overflow posts across various languages and through semi-structured interviews. We do so through the following research questions:

2.1 Research Questions

- **RQ1: Does cross-language interference occur?** We examined questions programmers had about programming languages on Stack Overflow for evidence of interference with previous programming knowledge.
- **RQ2: How do experienced programmers learn new languages?** To gain a better understanding of why cross-language interference occurs, we interviewed professional programmers on how they learn new languages.
- **RQ3: What do experienced programmers find confusing in new languages?** To examine the ways in which programmers mix a new language with their previous knowledge, we asked programmers about obstacles they faced, and surprises they encountered in their new languages.

2.2 Phase I: Study Design for Stack Overflow

To answer RQ1, we conducted a study using Stack Overflow posts.

Data collection. To gather Stack Overflow questions, we used the SOTorrent [13] data source from the 2019 MSR Mining Challenge. We queried 26 programming languages used previously by Erik [17] and Waren [42] in their investigation of popular language migrations, based on Google search keywords and Github repositories. We gathered Stack Overflow questions for each `<language A, language B>` pair. To keep the analysis tractable [43], we considered only the association between the two languages, and not the direction of the possible interference. We used a stop-rule criteria to cover over 95% of total posts, which resulted in 15 out of the 26 language pairs shown in Table 2. The materials for the study are available online.[2]

Query criteria. We used BigQuery[3] to query the SOTorrent database and used the following filtering criteria to capture potential posts where the programmers are asking questions about a new language (target) coming from a previous language (source):

(1) The question is tagged with both languages, or
(2) The question is tagged with the source language but contains the text of the target language in the title or body, vice-versa.

Analysis. To understand whether or not cross-language interference occurs, we performed a manual inspection of Stack Overflow posts (Table 2). We inspected a random sample of 30 posts for each pair to keep categorization tractable, as done in Barik et al. [14]. We manually excluded posts that did not make any explicit connection between the languages of each pair, sampling another random post to replace it as necessary. Because the inclusion and exclusion criteria can have multiple interpretations, the first two coauthors labelled a random sample of 30 posts. This labelling had 100% agreement between the coauthors, and suggests a clear understanding of how to categorize posts. The two coauthors proceeded

[1] https://www.stackoverflow.com

[2] https://go.ncsu.edu/cross-lang-study

[3] https://cloud.google.com/bigquery/

Table 1: Participants

ID	Exp[1]	Domain	Recent Transition
P1	15	Compilers	C# ⇒ Python ⇒ C++
P2	9	Data Science	Python ⇒ Julia
P3	18	Information Sciences	Python ⇒ PHP
P4	15	Neuroscience	R ⇒ Python
P5	10	Security	C++ ⇒ TypeScript
P6	20	Cloud Services	C# ⇒ TypeScript
P7	6	Cloud Services	C# ⇒ Python
P8	10	Web Platform	C# ⇒ JavaScript
P9	31	Data Science	C# ⇒ JavaScript ⇒ Scala
P10	8	Business Applications	C# ⇒ Rust
P11	12	Web Platform	C# ⇒ Ruby
P12	10	Data Science	Python ⇒ SAS
P13	6	Software Engineering	C++ ⇒ JavaScript
P14	10	Data Science	R ⇒ Python
P15	20	Software Engineering	C# ⇒ Swift
P16	5	Data Science	R ⇒ Python

[1] Years of self-reported programming experience.

to label the rest of the Stack Overflow posts using the following classifications:

- *Correct:* The post makes a connection to a previous programming language with correct assumptions regarding the target language as revealed by the accepted answer, or
- *Incorrect:* The post makes a connection to a previous programming language with incorrect assumptions regarding the target language as revealed by the accepted answer.

Next, we calculated inter-rater reliability (IRR) between the two coauthors (Cohen's κ = 0.89), and obtained "substantial" agreement [40]. We discussed disagreements on whether a post was correct or incorrect: if there was still disagreement, it was reconciled by the first author. Finally, we calculated the percentage of correct and incorrect posts. We used instances of correct and incorrect assumptions as evidence of cross-language interference and facilitation.

2.3 Phase II: Study Design for Interviews with Professional Programmers

To answer RQ2 and RQ3, we conducted semi-structured interviews with professional programmers.

Participants. We used *purposive sampling* [65] to recruit 16 professional programmers who were learning a new programming language within the past 6 months (Table 1); these participants were still early in their learning process and working through their initial stumbling blocks in the new language. The participants (12 male, 4 female, self-reported) were from large software, technology, and data analytics companies with years of programming experience ranging from 5 to 31 years (μ = 12.8, sd = 6.6). There were a total of 14 unique language transitions. Before the interview, participants completed a background questionnaire asking them about their previous languages and an obstacle they have experienced while adapting to the new language.

Protocol. We conducted semi-structured interviews either onsite or remotely, within 60 minute time blocks. Two of the authors conducted and recorded the interviews separately. All sessions were conducted with a single observer and a single programmer. We used the following structure for questions: 1) participant background, 2) first steps, 3) obstacles, 4) learning process, and 5) general strategies. The background information from the questionnaire was used to tailor the questions for the participants. The semi-structured interview format allowed the flexibility to ask questions impromptu and dig deeper into more specific obstacles. The recordings were later transcribed by the first author for analysis.

Analysis. *RQ2: How do experienced programmers learn new languages?* To answer RQ2, we conducted inductive thematic analysis [22] on the interview transcripts over multiple phases: transcribing interviews, generating open codes by labelling notable recurring statements made by the participants, identifying relationships between the codes, and organizing them into meaningful themes.

RQ3: What do experienced programmers find confusing in new languages? To understand how programmers confuse language concepts, we selected themes from our analysis that highlighted interference due to previous programming knowledge.

3 RESULTS

3.1 RQ1: Does cross-language interference occur?

Cross-language interference occurs on Stack Overflow across various language pairs. We found a total of 276 instances of incorrect assumptions (Table 2), which is around 61% of the 450 posts inspected. There were a total of 174 posts with correctly stated assumptions, which is only around 39% of the total posts. It's important to note that this provides evidence of interference occurring but does not imply programmers have incorrect assumptions 61% of the time. The `<Kotlin, Java>` pair had the highest number of posts with incorrect assumptions, which reflects the Java programmer's confusion mentioned in Section 1. The next two pairs, `<C#, Visual Basic>` and `<Scala, Java>`, also contained a high number of incorrect assumptions. However, there were other pairs like `<Python, C++>`, `<Java, C#>`, and `<PHP, Java>`, which had a more even distribution of posts with correct and incorrect assumptions; this suggests easier transitions between the languages. While reviewing the 450 Stack Overflow posts, we encountered instances where programming languages behaved in surprising ways for programmers. We highlight three examples, two of which involved interference between syntax and concepts, and one which involved facilitation—making it easier to use type inference.

Interference: R ⇒ Python[4]

An R programmer is now using Python and its data processing library, Pandas. They are unable to successfully relate their previous knowledge about subsetting, in R, to Python: "I'm seriously confused. Maybe I'm thinking too much in R terms and can't wrap my head around what's going on in Python."

Table 2: Posts by Language Pair

Language Pair[1]	Posts[2]	% Accepted[3]	Correct[4]	
			n	%
<C, C++>	30863	65%	9	30%
<C#, Visual Basic>	11522	62%	8	27%
<Objective-C, Swift>	9416	50%	10	33%
<Python, C++>	6763	51%	15	50%
<Java, C#>	6748	59%	16	53%
<Scala, Java>	6622	55%	8	27%
<PHP, Java>	6152	46%	16	53%
<R, Python>	2824	49%	12	40%
<Kotlin, Java>	2565	53%	6	20%
<Matlab, Python>	2407	53%	11	37%
<Node, PHP>	2077	40%	14	47%
<Ruby, Python>	1314	65%	14	47%
<Perl, Python>	1152	67%	13	43%
<Lua, C++>	1143	63%	12	40%
<Clojure, Java>	1098	68%	10	33%

[1] The pair of programming languages.
[2] Total number of questions where the two languages are tagged or referenced in body.
[3] Percentage of questions that have accepted answers.
[4] Total posts (out of 30) classified as having correct assumptions formed from prior language knowledge.

They present the R expression they want to translate, as well as several attempted translations in Python:

```
# R
data[data$x > value, y] <- 1
# Python
data['y'][data['x'] > value] = 1
```

Several concepts in R interfered, but we will highlight the most significant: Python prevents assignment to copies of dataframes. In this case, the indexing operation `data['y']` returns a copy of the dataframe and setting the value with `[data['y'] > value] = 1` will not work as the R programmer expects. The knowledge that the equivalent R expression will set the value of 1 without any warnings interferes with Python's warning.

Interference: PHP ⇒ JavaScript[5]

A PHP programmer who has switched to programming in JavaScript asks how to store transient information (sessions), such as application state about a user. Typically, PHP uses server-side session variables (`$_SESSION`) for this purpose. While related concepts, such as local storage and browser-based sessions exist, the programmer is warned that sessions cannot be safely and securely stored directly on the client—the programmer's knowledge about server-side sessions leads to a faulty assumption about their applicability in other programming contexts.

Facilitation: Java ⇒ Kotlin[6]

A Java developer is learning Kotlin. They ask if the following Kotlin expression can be simplified:

```
val boundsBuilder: LatLngBounds.Builder =
↪ LatLngBounds.Builder()
```

The developer suspects their declaration is more verbose than it should be, given their knowledge of local variable type inference in Java. They assume the declaration can be simplified:

```
val boundsBuilder = LatLngBounds.Builder()
```

This is an example of facilitation—the accepted answer confirms that the developer can simplify the expression because Kotlin supports type inference, allowing for the explicit type declaration to be removed.

These examples illustrate how previous knowledge of language syntax and concepts interact with knowledge learned in a new language. In some cases, this results in interference, which harms a programmer's ability to grasp new syntax and concepts in the new language. In other cases, this results in facilitation, which helps programmers make meaningful connections to previous languages and helps them learn the new language.

Cross-language interference occurs across various language transitions on Stack Overflow posts. We found that 61% of the 450 posts contained incorrect assumptions about the target language, and only 39% contained correct assumptions.

3.2 RQ2: How do experienced programmers learn new languages?

We present the themes on how experienced programmers learn new languages. A summary of the themes is listed in Table 3.

3.2.1 Programmers learned languages on their own. Programmers who switched teams lacked formal training for the new language and its associated technology stack, leaving learning to themselves. For example, when P1 switched from C# to Python for a new project, there wasn't any training involved and the on-boarding process was, "hey we want to get exposed to the Python world, go get started!" Although some programmers were given training initially on the project, "realistically for learning the new language [they] were pretty much on [their] own" (P7). This forced programmers to watch "language tutorial videos on Pluralsight"[7] (P5) or read online documentation. Some programmers "got initial tips from some folks from the team on what's what" (P6), and when running into complex issues "reached out to the group and said has somebody else hit this before?" (P1).

[4] https://stackoverflow.com/questions/30923882/pandas-logical-indexing-on-a-single-column-of-a-dataframe-to-assign-values

[5] https://stackoverflow.com/questions/47137666/which-variable-is-used-to-store-angular-session-value-same-as-php-session-vari

[6] https://stackoverflow.com/questions/38131655/operator-new-in-kotlin-syntax

[7] https://www.pluralsight.com

Table 3: Learning Strategies and Language Interference Themes

Learning Strategies			
Theme	Description	Representative Examples	Participants[1]
Learning on their own (Section 3.2.1)	Programmers lacked formal training for the new language and its associated technology stack, leaving learning to themselves.	"We didn't have a procedure for people getting up and running." "'I just do everything ad-hoc!" "I got initial tips from some folks from the team on what's what."	P1, P2, P5, P6, P7, P13, P14, P15, P16
Just-in-time learning (Section 3.2.2)	Programmers focused on only learning features as needed.	"There's probably like a content cheat sheet." "I didn't learn typescript step-by-step." "Step one for me is always find and read other people's code."	P1, P2, P3, P5, P9, P14, P15
Relating new language to previous languages (Section 3.2.3)	Programmers tried to map features of the new language to their previous languages.	"I loosely [take] ideas from working in another language." "I would try to find the counterpart of C++ in React." "If you can compare them side by side and find their similarities you're more than halfway there."	P1, P2, P9, P12, P13, P14, P15
Language Interference			
Theme	Description	Representative Examples	Participants[2]
Old habits die hard (Section 3.3.1)	Programmers had to constantly suppress old habits from previous languages.	"I'm typing `a[1]` thinking that it's `a[0]`." "I still type the type first before the variable." "I'm gonna make it an object for this, no don't do that!"	P2, P3, P4, P6, P9, P15
Mindshifts when switching paradigms (Section 3.3.2)	Sometimes programmers wrestled with larger differences that required fundamental shifts in mindsets, or "mindshifts."	"All my assumptions were thrown out the window." "I had to rethink the problem and re-implement it." "There are lots of events and promises all these things makes it really hard to debug."	P2, P5, P6, P9, P10, P13, P15
Little to no mapping with previous languages (Section 3.3.3)	Programmers had a harder time learning the new language when there was little to no mapping of features to previous languages.	"There's a very alien concept in Rust that is the borrow checker." "I've never had a language with traits before." "I did not work with concepts like virtual DOM, shadow DOM before."	P2, P5, P9, P10, P11, P15
Searching for terms and documentation is hard (Section 3.3.4)	Programmers found it difficult to search for information about the language and its associated technologies.	"You don't even know what exists, what to even look for." "Scala is not that common. Some of it required a little deeper digging." "They have their own convention, TypeScript has its own convention, JavaScript has its own convention."	P1, P2, P4, P8, P9, P11, P12
Retooling is a challenging first step (Section 3.3.5)	Programmers faced difficulty retooling themselves in the environment of the new language.	"I was using Visual Studio to debug C# code and now it's gdb to debug C++ code." "In Xcode, build targets aren't 'Universal' in definition like .NET." "The problem is IntelliJ is aimed at the Java developer and I'm using SBT which is from the Scala world."	P1, P2, P9, P12, P15

[1] Participants who used a similar learning strategy.
[2] Participants who experienced the particular language interference theme.

3.2.2 Just-in-time learning is a dominant strategy. To learn new languages, every programmer we interviewed used *just-in-time learning* [21], an opportunistic strategy focused on only learning features as needed. Given time constraints, programmers made use of immediately available resources like online documentation, video tutorials, online searches, and available experts. Traditional resources like programming language books were only used as a reference, since programmers "just don't have time to do that" (P5). Programmers were primarily concerned with completing tasks in a reasonable time and "figuring out how to not burn tons of time on a single problem" (P1). Quicker resources, like cheat sheets, were preferred for language transitions. For example, the first thing P2

did was to make use of cheat sheets [55] to help them transition from Python to Julia. P15 was also a fan of cheat sheets:

> *It seems like if you were going from one framework to another, from one technology stack to another—even if you're not going from A to B, you're just starting off on B—there's probably a content cheat sheet that every dev needs to know.* (P15)

3.2.3 Programmers related the new language to previous languages. To help accelerate the learning process, programmers generally tried to relate the new language to their previous languages. Programmers started by "loosely taking ideas from working in another language" (P14) or looking at existing code because "it's already probably been written and it's out there somewhere or at least something close to it" (P1). While this learning strategy was useful for bootstrapping, some programmers started from scratch. For example, when moving from C# to Ruby, P11 described "trying to be very conservative and mindful and trying not to map anything over, but just treating everything as something brand new." Similarly, P12 explained that they did not try to map things from Python when learning SAS "mostly because the syntax was so new that every time [they] tried to do anything, [they] would have to go and google the syntax." P10 expressed a similar problem when learning about managing memory in Rust after years of using C#: "there wasn't a clean way for me to just get there. I had to go and learn that stuff from scratch." These examples illustrate that programmers typically reuse knowledge—if possible—but sometimes avoid doing so when it's more troublesome.

Programmers use an opportunistic learning strategy, relating syntax and concepts of the new language with their previous language. This offers expediency but causes interference when major differences exist between the two languages.

3.3 RQ3: What do experienced programmers find confusing in new languages?

We present the themes explaining how programmers confuse language concepts. A summary of these themes is listed in Table 3.

3.3.1 Old habits die hard. Programmers had to constantly suppress old habits acquired from previous languages. For example, P3—who was used to Python—had trouble adapting to block delimiters in PHP, where "it's near-impossible to figure out exactly which opening brace you're closing once your HTML/PHP gets to any complexity at all." Similarly, P15 realized that "in Swift, the open curly bracket needs to be on the initial line of the method declaration and if you put it on the next line the method may not execute in an expected fashion." There were minor but frustrating difference like 0 versus 1 indexing for lists in languages such as Python and R. P4 described their frustration in "typing `a[1]` thinking that it's `a[0]`, and then wasting 5 minutes like a complete fool not understanding why nothing makes sense" (P4). Programmers are able to resolve these small differences, but it still causes interference at the onset of learning a new language.

3.3.2 Mindshifts are required when switching paradigms. Some language transitions required fundamental shifts in mindsets, or "mindshifts" [10]. For example, when P2 transitioned from Python to Julia, they were constantly trying to make an object and realizing that "there's no objects, there's only structs!" With Julia, they needed to write more functional code, a shift from the object-oriented programming that they were used to in Python: "it was just needing to shift that and realize I'm never gonna write 'something-dot-something-else' ever or rarely." For P10, they had to completely rethink the problems they would have normally solved in C# because of Rust's unique ownership feature for memory safety:

> *A really fascinating thing about learning Rust was that when I went and started to do these things—things that I would reach for in C# that I knew would work—Rust wouldn't allow it and as a result I had to rethink the problem and re-implement it in a way where the ownership characteristics of that algorithm were very explicit.* (P10)

Another big paradigm shift occurred for P5, P6 and P13—all transitioning from imperative or object-oriented coding to event-driven and asynchronous coding—forcing them to think differently. The programmers had to learn brand new concepts in JavaScript like asynchronous programming or "shadow and virtual DOMs" (P13). P6 described how it was difficult making sense of asynchronous code because "you got a whole bunch of '`async`/`await` mode' working in your mind and you have to convert it." To make matters worse, "the most confusing part is there are a couple of ways to do asynchronous programming, with observables or promises" (P13). For P5, whose background was in C++, the front-end coding in TypeScript was a big challenge because "for the back-end, the code I think is more straightforward. You have the logic and most likely you know single places you'll handle it. It's not like the UI" (P5). Here, the interference issues aren't due to any particular syntax or concept but the way one solves problems in the new language.

3.3.3 Learning a language is difficult when there is little to no mapping with previous languages. Programmers had a harder time learning the new language when there was little to no mapping of features to previous languages. For example, P12 could not make sense of some fundamental programming language features of SAS that were clear in Python, like statements versus method parameters. They could not understand "why some things are statements that affect a procedure, but aren't parameters" and were "still confused about the overall syntax and what is or isn't a statement"—even after having worked in the language for a few weeks. A drastic example was P5, who experienced a big transition from C++ to TypeScript, resulting in *tech shock*: "Everything is different! Not just the programming language—the IDE, source control, everything is different." P13, who underwent a similar language transition, found that concepts were challenging in JavaScript because they "could not equate it back to C++." Due to limited mapping of features to previous languages, programmers could not make full use of facilitation to learn the new language.

In the extreme case, programmers were forced to learn a completely foreign syntax or concept, in particular, when it was an essential built-in feature of the new language. For example, P9 had difficulty learning traits in Scala because they "never had a language

with traits before. Traits have a default implementation and understanding what would be performant and what wouldn't—and when to use what—that was the tricky part." P7 learned that for Python, "the major difference is the multiple inheritance thing, that Python inherits from the C++ world, which supports multiple inheritance. In C# you can't do that." In another case, the difficulty was due to differences in memory management, for example, when P10—who previously used C#—was learning Rust:

> *There's a very alien concept in Rust that is the borrow checker, which is the concept of having the compiler verify more things, and the way it does it is somewhat esoteric. That's very alien, and that's something that I think is really cool but it's also very rough at the moment and so that's kind of something that's been the biggest struggle when trying to learn Rust.* (P10)

Even within the same context, such as data analysis or mobile applications, the lack of mapping caused a lot of confusion. For example, P14, who switched from one data analysis language (R) to another (Python/Pandas), could not find an immediate equivalent for R's `spread` and `gather` functions: "Pandas already had the functionality but it was more hidden using drop level and unstack. These were really hard to understand in Pandas—it was some pretty weird stuff." Similarly, P15, who switched from C# to Swift, was very surprised to learn how the user interface code and its graphical layout view in Xcode were connected: "Knowing that you can't interact with a UI object straight out of the box from the code is very important. Once you draw the referencing outlet connection between View and Controller you can trigger methods and `get/set` properties as you'd expect in the .NET world."

3.3.4 Searching for the right terminology and code examples is difficult. We found that moving to a new programming language presents a *selection barrier* [39], making it difficult to search for information about the language and its associated technologies. Programmers recounted trouble acquiring the vocabulary even before performing the search. For P12, the names for the same structures in Python/Pandas were slightly different than SAS where a "dataframe is data set, a row is an observation, a column is a variable." When they tried to plot with SAS, they "don't know what the name of the `proc` for plotting in SAS is so [they] have to start looking that up first, then find documentation for a couple different ones, then have to figure out how to make them work." On the one hand, "it's the breadth of the libraries that usually get you, you don't even know what exists, what to even look for to see if something is already there" (P1). On the other hand, insufficient search results provided little to no facilitation. For example, P4 had difficulty searching information for a Python library called seaborn—compared to the equivalent R library ggplot—because "it is just less documented. For ggplot, if you google anything, you get like 100 hits, and the top ones are bound to be good due to Google selection of results. With seaborn, you get like 10 hits."

Even when programmers found documentation and code examples, they were either incomplete or lacking in detail. For example, P8 expressed a frustration regarding testing libraries in JavaScript because "they have their own convention, TypeScript has its own convention, JavaScript has its own convention, it is actually mixing everything!" This was especially problematic when conventions found online weren't always the same ones used by the specific team: "There's a lot of conventions around the language. In C++, the styles can change a bunch from team to team" (P1). For some languages, the documentation was either lacking in quality or was completely missing. For example, P2 was frustrated with the Julia documentation because "it was so useless for figuring out the imports." Similarly, P12 expressed that the SAS documentation "only tells you how to copy-paste and run a simple program, leaving you completely mystified as to how the execution and control flow of a SAS program works." This lack of depth can lead to frustrating experiences for programmers when they had better documentation in previous languages, such as P15: "Xcode documentation samples were pretty good enough to where they would run. But the documentation, MSDN, and the available samples for creating Microsoft platform-based applications were tenfold deeper and richer and easier for to use."

3.3.5 Retooling is a necessary and challenging first step. Finally, before programming in the new language, programmers faced difficulty retooling themselves in a new environment. This typically involved adapting to the discrepancies of the new integrated development environment (IDE) for programming in the language. Although programmers were able to adapt to basic features of IDEs (facilitation), there was interference when some aspects of the IDE differed from their previous IDEs. For example, P15 discovered that in Xcode "build targets aren't 'Universal' in definition (like .NET) and when terminologies are shared across platforms but don't implement the same notion, you're lost for days!" Interestingly, for P9 there was interference when they tried building their Scala project in IntelliJ because the IDE attempted to support Scala, but continued presenting dialogs in the previous language:

> *Part of the problem is IntelliJ is aimed at the Java developer and I'm using SBT, which is from the Scala world. And it's sort of importing the SBT into the concepts in the IDE of IntelliJ. So I'm looking at dialogs that are all about Java and which JDK and that doesn't map to what I wrote in the declarative SBT language.* (P9)

Other concerns regarded either a lack of IDE features or learning new features that were distracting. P2 had been "spoiled with Python and PyCharm" and found it very difficult to find proper IDE support for Julia; they just wanted "an IDE that does syntax highlighting and IntelliSense-like autocompletion." P1 found that learning a new feature—like debuggers—effectively halts a programmer's progress on actual tasks and are distracting "because you're learning and debugging at the same time as opposed to just debugging once you're fluent."

However, sometimes the transition to new tools in the language also benefited programmers. For example, P5 found it a lot easier moving from MSBuild (C++) to Gulp (JavaScript), which allowed fast build cycles when developing TypeScript applications. In particular, the DevOps pipeline helped them make progress much quicker:

> *I think right now the build system for us, I think it's better since now we are using DevOps—a pipeline to build the code. It's very easy for us to even schedule the private build and also it's very easy for us to quickly*

> *get new things, check in the code, test it, and even build things on top of it.* (P5)

Programmers confuse a new language's syntax and concepts with previous languages, leading to a number of issues like trying to suppress old habits, wrestling with mapping issues, struggling to find and use proper documentation, retooling and shifting one's mindset for new paradigms.

4 LIMITATIONS

Our mixed-methods approach of investigating Stack Overflow and conducting interviews introduces certain trade-offs and limitations.

The choice of sampling technique in our Stack Overflow analysis has several trade-offs [45]. Because the sampling approach is non-probabilistic, it does not allow for sample-to-population, or *statistical generalization*. Rather, our approach targets diversity (rather than representativeness) in order to identify evidence of interference across many different programming languages.

We used correct and incorrect assumptions as a proxy construct for facilitation and interference. While this approach provides us with a useful, high-level characterization of the Stack Overflow posts, there are potentially additional insights that we could learn had we performed a more intricate qualitative coding technique, such as open coding. The trade-off for doing so is that open coding is significantly more costly to execute. Instead, we conducted semi-structured interviews with experienced programmers to delve deeper into cross-language interference.

The posts we examined on Stack Overflow as well as our interviews do not completely cover the set of all language transitions, as the full permutation space of language transitions is intractable. Our approach attempts to cover language transitions that are most likely to occur in practice. Consequently, there may be some interference issues that our study was not able to identify.

Finally, we acknowledge that qualitative research, however rigorously conducted, involves not only the qualitative data under investigation but also a level of subjectivity and interpretation on the part of the researcher as they frame and synthesize the results of their inquiry. To support interpretive validity, we followed the guidelines set by Carlson [24] and performed a single-event member check with our results. Six participants who replied agreed with our presentation of the results and only wanted minor changes to their quotations. Additionally, we emphasize that interference theory is only one of many possible lenses through which we can organize and present our findings. Other theories, such as *notional machines*, have also been used to identify and explain programming conceptions [18, 29, 30].

5 RELATED WORK

Novice misconceptions. Programmers often have misconceptions while learning new programming languages, but most studies have focused on novices. Swidan et al. [64] proposes "intervention methods to counter those misconceptions as early as possible," but this work is primarily targeted to novices. Similarly, Kaczmarczyk et al. [37] has examined misconceptions and how to measure them for novices. In contrast, the novelty of our work is towards experienced programmers who need to switch languages [44], and requires methods of learning distinct from those designed for novices [32, 38, 63]. Our study investigated switching languages for experienced programmers and took the first steps in examining how knowledge of previous languages interfere when learning another language.

Programming language transitions. There are a few studies on transitions between programming languages. Scholtz and Wiedenbeck [57] studied experienced Pascal or C programmers writing a program in a new language, Icon, and found that they were strongly influenced by their knowledge of what would be appropriate in previous languages. Wu and Anderson [70] conducted a similar study where programmers who had experience in Lisp, Pascal and Prolog wrote solutions to programming problems and found that solutions written in one language facilitated learning in another language. Uesbeck and Stefik [66] studied the effect of using multiple languages in a controlled study, where participants implemented several variations of database queries: some variants involving the same language, while others mixing SQL and Java. While the results were inconclusive, the authors suggest that the methodology could be effective for studying the productivity costs associated with mixing languages. We examined empirical evidence and conducted interviews to understand the transition from one language to the next for various contexts. We also investigated how programmers confuse two different languages using the lens of interference theory [67].

There have been fewer studies on interventions for learning new languages. Bower and McIver [20] explored a new teaching approach called Continual And Explicit Comparison (CAEC) to teach Java, using facilitation, to students who have knowledge of C++. They found that students benefited from the continual comparison of C++ concepts to Java. Shrestha et al. [60] used a similar technique using a tool called Transfer Tutor to teach R from the perspective of Python; programmers who used the tool found the comparisons between the languages useful. These intervention techniques might benefit programmers who learn new languages from the perspective of a known neighboring language, but there are a number studies on larger transitions—for example, from procedural or imperative to object-oriented languages [10, 28, 46, 47]. These studies have shown professional programmers experience greater interference as they have to make fundamental shifts or "mindshifts," which might require further support. In this study, we have uncovered interference issues in the modern context and examined numerous language transitions. We also found other issues that have not been explored like dealing with little to no mapping of language features (Section 3.3.3) and retooling (Section 3.3.5), which have implications for future tools and techniques.

Programming knowledge. Knowledge structures have been proposed for how programmers encode semantic [59] and domain information [23] about a program as well as *prime structures* [41], that include elements of syntax, control-flow and data-flow [51] of the program. These knowledge structures [56] have been formalized and referred to as *programming plans*. Programming plans act like schemas that are first instantiated and then its slots are filled with concrete values as a programmer builds an understanding of the code [62]. Plans may help programmers fill in the "gaps" when trying to understand code.

Gilmore and Green [31] suggested that programming plans may not generalize across different languages, and that plans cannot represent the underlying deep structure of programs. Bellamy and Gilmore [16] examined the protocols generated from experts in different languages as they created programs. Using two different models of programming plans, they found neither model was well supported by protocols; further, different programming language experts generated different types of representations. We believe our results provide further insight as to why plans may not generalize across languages. For example, we found programmers tend to relate a new language to previous languages (Section 3.2.3), which suggests an attempt to reuse previous programming plans as a bootstrapping strategy. However, due to interference issues, the previous plans might either need significant modifications (Section 3.3.3) or be replaced entirely (Section 3.3.2), depending on how closely related the two languages are.

6 DISCUSSION AND DESIGN IMPLICATIONS

Our findings demonstrate that interference is not an isolated phenomenon; indeed, in Stack Overflow, instances of interference are found across all of the programming languages we investigated. Furthermore, in our interviews, participants reported that interference arises routinely as they learn a new language—for example, from having to suppress old habits from previous languages (Section 3.3.1) or having to "rethink the program" (P10) due to a substantially different paradigm (Section 3.3.2 and Section 3.3.3).

As opposed to traditional classroom environments where one learns "step-by-step" (P5), experienced programmers in our study used opportunistic strategies to learn essentially "on [their] own" (P7) or "learning through work" (P13), for example, using online resources or asking teammates (Section 3.2.2) [21]. Unfortunately, these informal approaches to learning sometimes result in an incomplete lens for how the language works, resulting in "unintentional bugs" (P5) and other difficult-to-diagnose problems in the code when something doesn't work as expected.

In the remainder of this section, we present design implications for technical authors, toolsmiths, and programming language designers that can help reduce some of these interference difficulties for programmers.

Implication I—Design documentation that reduces interference and supports knowledge transfer. Programmers in our study desired more accessible resources that leveraged the programming knowledge they already have (Section 3.2.2 and Section 3.2.3). Such resources included "cheat sheets," which present code snippets that map their familiar language to their new language (P2) and relate concepts they already know "from working in another language" (P14), to the new language tutorials, and even resorting to "reading other people's code" (P3, P15) to understand the programming language idioms.

Our findings suggest that resources that teach languages through relating a new language to a known language are more useful and accessible to programmers than resources that present the new programming language in isolation. Several books [26, 34, 49, 72], blogs [5, 7], language documentation [4, 6, 69], and training courses [2, 8] embody this pedagogical strategy.

However, these resources—while useful—are essentially handcrafted through the authors' intuitions about what misconceptions the programmer might have, and not necessarily the ones that programmers actually have. While misconceptions about novice programmers are readily found in the literature [27, 37, 54], misconceptions experienced programmers have are comparatively understudied. Shrestha and Parnin [61] presented three possible instrument designs which can be used for discovering and validating misconceptions when switching languages for experienced programmers. Such research is needed to make learning resources more effective and relevant to experienced programmers.

Implication II—Build automated tools to provide on-demand feedback. Although technical documentation is useful, these resources are decoupled from where the programmer needs the most help—in their program environment as they work (Section 3.2.2 and Section 3.3.4).

Automated tools can help with this. For example, Johnson et al. [33] propose "bespoke" notification tools that provide adaptive feedback to the programmer based on the programmer's prior knowledge of programming languages and concepts. Python 3 adopts this idea of using prior programmer knowledge to assist programmers who come from a Python 2 background, through hard-coded error messages: in Python 2, `print` does not require surrounding parentheses, while in Python 3, `print` is a function and thus must be called like any other function:

```
>>> print "Hello"
  File "<stdin>", li
    print "Hello"
                ^
SyntaxError: Missing parentheses in call to 'print'.
↪  Did you mean print("Hello")?
```

The `SyntaxError` message makes the assumption that this error is due to a misconception (or ingrained behavior) instilled from experience with Python 2. We can repurpose this idea generally to language transitions and help programmers more efficiently resolve error messages that they might otherwise only "eventually figure out" (P1) after spending substantial time and effort.

Implication III—Be intentional about programming language syntax, semantics, and pragmatics. Certain programming languages anticipate that new adopters arrive through common pathways. That is, we expect most new Rust users to come from systems programming languages like C++, and we expect most new TypeScript users to come directly from JavaScript. For these users, intentionally designing language features by considering interference effects can reduce barriers (Section 3.3.2 and Section 3.3.3) to adopting the new programming language.

As an example, a substantial barrier to new Rust users is the borrow checker—a compile-time feature that helps enforce safe memory management [71]—which our own participants described as "a very alien concept" (P10). Even the Rust manual concedes that borrow checking has a costly "learning curve" and that programmers "fight with the borrow checker" because their "mental model of how ownership should work doesn't match the actual rules that Rust implements" [3]. Interference theory also explains

these difficulties: for some programmers, the borrow checker is so unfamiliar as a concept that they have no prior support to *facilitate* learning; and for other programmers, borrow checker concepts at a casual glance seem similar to existing models, such as "resource acquisition is initialization" (RAII), in C++, but ultimately functions differently enough that it *interferes* with their past knowledge.

Intentionally considering these adoption pathways as part of language design can reduce these interference challenges. For instance, the "primary goal of TypeScript is to give a statically typed experience to JavaScript development" and "the intention is that TypeScript provides a smooth transition for JavaScript programmers—well-established JavaScript programming idioms are supported without any major rewriting or annotations" [19]. But providing this smooth transition has a costly consequence: "the TypeScript type system is not statically sound by design."

As the two examples illustrate, designing for interference requires making difficult design trade-offs. But if we want to design programming languages that people actually use, we need to consider how our language design decisions interfere or facilitate with our anticipated programmers' prior knowledge.

Implication IV—Support not only programming languages, but programming language ecosystems. Issues with interference when learning new programming languages are exasperated when new programming languages bring with them new programming language *ecosystems*—that is, "everything is different, not just the programming language" (P5), but the environment in which the programmer builds, edits, debugs and tests their code (for example, *tech shock*, Section 3.3.5).

To address these challenges, React developers provide tool support to welcome programmers into the new ecosystem. Specifically, the `create-react-app` [9] is an integrated toolchain that abstracts away the complexities of third-party library management, live-editing, optimization, and configuration. `create-react-app` allows the user to quickly and easily begin experimenting with the library until the programmer is comfortable enough to *eject* from the `create-react-app` toolchain.

A second method to minimize interference issues from ecosystems is to unify the underlying tooling environment, or at least provide the programmers with a unified tooling experience. From this perspective, we would recommend that toolsmiths and language designers add support for programming languages to well-established integrated development environments, rather than providing custom tool and editing experiences. For instance, the language server protocol (LSP) [1] allows programming language support to be implemented and distributed independently of any given editor or IDE, as long as that IDE implements LSP.

In short, language designers should collaborate with tool designers so that programmers can more easily adopt new programming languages through editing environments that are already familiar to them.

7 CONCLUSION

In this study, we conducted a mixed-methods study to understand what impact previous programming language experience has on programmers. We conducted an empirical study of misconceptions found in Stack Overflow questions across 18 different programming languages and semi-structured interviews with 16 professional programmers. From Stack Overflow, we found 276 instances of interference that occur across multiple languages. We then interviewed programmers who reported various challenges learning a new language—like mixing up the syntax and concepts with their previous programming languages—due to interference. We discussed design implications for technical authors, toolsmiths, and language designers, such as designing documentation and building automated tools that reduce interference, anticipating uncommon language transitions during language design, and welcoming programmers not just into a language, but its entire ecosystem. As Peter suspected, even professional programmers have difficulties with learning programming languages, and we should offer tools and techniques to help them learn more efficiently and effectively.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under Grant Nos. 1559593, 1755762, and 1814798.

REFERENCES

[1] [n.d.]. *Langserver.org*. https://langserver.org/
[2] [n.d.]. *Python for MATLAB Users*. https://www.datacamp.com/courses/python-for-matlab-users
[3] [n.d.]. *References and borrowings*. https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html
[4] [n.d.]. *To Ruby From Python*. https://www.ruby-lang.org/en/documentation/ruby-from-other-languages/to-ruby-from-python/
[5] 2010. *Thinking in Clojure for Java Programmers*. https://www.factual.com/blog/thinking-in-clojure-for-java-programmers-1/
[6] 2014. *Comparison with R / R libraries*. https://pandas.pydata.org/pandas-docs/stable/getting_started/comparison/comparison_with_r.html
[7] 2014. *Rust for C++ programmers*. http://featherweightmusings.blogspot.com/2014/04/rust-for-c-programmers-part-1-hello.html
[8] 2018. *New Course: Python for R Users*. https://www.datacamp.com/community/blog/course-python-r-users
[9] 2019. *Create React App*. https://create-react-app.dev
[10] Deborah J Armstrong and Bill C Hardgrave. 2007. Understanding mindshift learning: the transition to object-oriented development. *MIS Quarterly* (2007), 453–474.
[11] William R Aue, Amy H Criss, and Matthew D Novak. 2017. Evaluating mechanisms of proactive facilitation in cued recall. *Journal of Memory and Language* 94 (2017), 103–118.
[12] David Badre and Anthony D Wagner. 2005. Frontal lobe mechanisms that resolve proactive interference. *Cerebral Cortex* 15, 12 (2005), 2003–2012.
[13] Sebastian Baltes, Christoph Treude, and Stephan Diehl. 2018. SOTorrent: studying the origin, evolution, and usage of Stack Overflow code snippets. *CoRR* abs/1809.02814 (2018). arXiv:1809.02814 http://arxiv.org/abs/1809.02814
[14] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How should compilers explain problems to developers?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. 633–643. https://doi.org/10.1145/3236024.3236040
[15] Nathaniel Barragan. 2018. *My experience with learning Rust*. https://medium.com/@nathanielbarragan/my-experience-with-learning-rust-bbcb6b7c1063
[16] RKE Bellamy and DJ Gilmore. 1990. Programming plans: internal or external structures. *Lines of Thinking: Reflections on the Psychology of Thought* 2 (1990), 59–72.
[17] Erik Bernhardsson. 2017. *The eigenvector of "why we moved from language X to language Y"*. https://erikbern.com/2017/03/15/the-eigenvector-of-why-we-moved-from-language-x-to-language-y.html
[18] Berry, Michael and Kölling, Michael. 2014. The state of play: a notional machine for learning programming. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. 21–26.
[19] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). 257–281.

[20] Matt Bower and Annabelle McIver. 2011. Continual and explicit comparison to promote proactive facilitation during second computer language learning. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. 218–222. https://doi.org/10.1145/1999747.1999809

[21] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1589–1598.

[22] Virginia Braun, Victoria Clarke, Nikki Hayfield, and Gareth Terry. 2019. *Thematic Analysis*. Springer Singapore, Singapore, 843–860. https://doi.org/10.1007/978-981-10-5251-4_103

[23] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543–554.

[24] Julie A Carlson. 2010. Avoiding traps in member checking. *Qualitative Report* 15, 5 (2010), 1102–1113.

[25] Luis P Coelho. 2017. *I tried Haskell for 5 years and here's how it was*. https://metarabbit.wordpress.com/2017/05/02/i-tried-haskell-for-5-years-and-heres-how-it-was/

[26] Michael C Daconta. 1996. *Java for C/C++ Programmers*. Wiley New York.

[27] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. 2012. Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. ACM, 21–26.

[28] Françoise Détienne. 1995. Design strategies and knowledge in object-oriented programming: effects of experience. *Human–Computer Interaction* 10, 2-3 (1995), 129–169.

[29] Benedict Du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.

[30] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies* 14, 3 (1981), 237–249.

[31] D. J. Gilmore and T. R. G. Green. 1988. Programming plans and programming expertise. *The Quarterly Journal of Experimental Psychology Section A* 40, 3 (1988), 423–442. https://doi.org/10.1080/02724988843000005

[32] Philip J Guo. 2013. Online Python Tutor: embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. ACM, 579–584.

[33] Brittany Johnson, Rahul Pandita, Emerson Murphy-Hill, and Sarah Heckman. 2015. Bespoke tools: adapted to the concepts developers know. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 878–881.

[34] Allen Jones. 2002. *C# for Java developers*. AOL Time Warner Book Group.

[35] J. Jonides and D.E. Nee. 2006. Brain mechanisms of proactive interference in working memory. *Neuroscience* 139, 1 (2006), 181—193. https://doi.org/10.1016/j.neuroscience.2005.06.042

[36] John Jonides, Edward E Smith, Christy Marshuetz, Robert A Koeppe, and Patricia A Reuter-Lorenz. 1998. Inhibition in verbal working memory revealed by brain activation. *Proceedings of the National Academy of Sciences* 95, 14 (1998), 8410–8413.

[37] Lisa C Kaczmarczyk, Elizabeth R Petrick, J Philip East, and Geoffrey L Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. ACM, 107–111.

[38] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)* 37, 2 (2005), 83–137.

[39] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[40] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* (1977), 159–174.

[41] Richard C Linger, Harlan D Mills, and Bernard I Witt. 1979. Structured programming: theory and practice. (1979).

[42] Waren Long. 2017. *Analyzing GitHub, how developers change programming languages over time*. https://blog.sourced.tech/post/language_migrations/

[43] Mark Mason. 2010. Sample size and saturation in PhD studies using qualitative interviews. In *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, Vol. 11.

[44] Leo A Meyerovich and Ariel S Rabkin. 2013. Empirical analysis of programming language adoption. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 1–18.

[45] Claus Adolf Moser. 1952. Quota sampling. *Journal of the Royal Statistical Society. Series A (General)* 115, 3 (1952), 411–423.

[46] H James Nelson, Deborah J Armstrong, and Kay M Nelson. 2009. Patterns of transition: the shift from traditional to object-oriented development. *Journal of Management Information Systems* 25, 4 (2009), 271–298.

[47] H James Nelson, Gretchen Irwin, and David E Monarchi. 1997. Journeys up the mountain: different paths to learning object-oriented programming. *Accounting, Management and Information Technologies* 7, 2 (1997), 53–85.

[48] Peter Norvig. 2000. *Python for Lisp Programmers*. https://norvig.com/python-lisp.html

[49] Ajay Ohri. 2017. *Python for R Users: A Data Science Approach*. John Wiley & Sons.

[50] Dino Paskvan. 2014. *Why coffeescript?* https://discuss.atom.io/t/why-coffeescript/131/37

[51] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295–341.

[52] Bradley R Postle and Lauren N Brush. 2004. The neural bases of the effects of item-nonspecific proactive interference in working memory. *Cognitive, Affective, & Behavioral Neuroscience* 4, 3 (2004), 379–392.

[53] Bradley R Postle, Lauren N Brush, and Andrew M Nick. 2004. Prefrontal cortex and the mediation of proactive interference in working memory. *Cognitive, Affective, & Behavioral Neuroscience* 4, 4 (2004), 600–608.

[54] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: a literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1.

[55] QuantEcon. 2017. *MATLAB–Python–Julia cheatsheet*. https://cheatsheets.quantecon.org

[56] Charles Rich. 1981. *Inspection Methods in Programming*. Technical Report TR-604. MIT. http://hdl.handle.net/1721.1/6934

[57] 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

[58] Hari Shankar. 2011. *Why learning Functional Programming and Haskell in particular can be hard*. https://harishankar.org/blog/entry.php/why-learning-functional-programming-and-haskell-in-particular-can-be-hard

[59] Ben Shneiderman and Richard Mayer. 1979. Syntactic/semantic interactions in programmer behavior: a model and experimental results. *Int'l J. Parallel Programming* 8, 3 (1979), 219–238.

[60] Nischal Shrestha, Titus Barik, and Chris Parnin. 2018. It's like Python but: towards supporting transfer of programming language knowledge. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 177–185.

[61] Nischal Shrestha and Chris Parnin. 2019. Instrument designs for validating cross-language behavioral differences. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 205–209.

[62] Elliot Soloway, Kate Ehrlich, and Jeffrey Bonar. 1982. Tapping into tacit programming knowledge. In *Proceedings of the 1982 Conference on Human Factors in Computing Systems (CHI '82)*. 52–57. https://doi.org/10.1145/800049.801754

[63] John Sweller, Paul L Ayres, Slava Kalyuga, and Paul Chandler. 2003. The expertise reversal effect. (2003).

[64] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming misconceptions for school students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 151–159.

[65] Ma Dolores C Tongco. 2007. Purposive sampling as a tool for informant selection. *Ethnobotany Research and Applications* 5 (2007), 147–158.

[66] Phillip Merlin Uesbeck and Andreas Stefik. 2019. A randomized controlled trial on the impact of polyglot programming in a database context. In *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018) (OpenAccess Series in Informatics (OASIcs))*, Titus Barik, Joshua Sunshine, and Sarah Chasins (Eds.), Vol. 67. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:8. https://doi.org/10.4230/OASIcs.PLATEAU.2018.1

[67] Benton J Underwood. 1957. Interference and forgetting. *Psychological Review* 64, 1 (1957), 49.

[68] Bartosz Walacik. 2018. *From Java to Kotlin and Back Again*. https://allegro.tech/2018/05/From-Java-to-Kotlin-and-Back-Again.html

[69] Greg Wilson. 2018. *The Tidynomicon: A Brief Introduction to R for Python Programmers*. https://gvwilson.github.io/tidynomicon/

[70] Quanfeng Wu and John R. Anderson. 1990. *Problem-solving transfer among programming languages*. Technical Report. Carnegie Mellon University.

[71] Anna Zeng and Will Crichton. 2019. Identifying barriers to adoption for Rust through online discourse. In *9th Workshop on Evaluation and Usability of Programming Languages and Tools*. 15.

[72] Nailong Zhang. [n.d.]. *Another Book on Data Science*. https://www.anotherbookondatascience.com