An Empirical Study of Experienced Programmers' Acquisition of New Programming Languages

Parastoo Abtahi
Stanford University
Stanford, USA
parastoo@stanford.edu

Griffin Dietz
Stanford University
Stanford, USA
gdietz44@stanford.edu

ABSTRACT

Experienced programmers often need to use online resources to pick up new programming languages. However, we lack a comprehensive understanding of which resources programmers find most valuable and utilize most often. In this paper, we study how experienced programmers learn Rust, a systems programming language with comprehensive documentation, extensive example code, an active online community, and descriptive compiler errors. We develop a task that requires understanding the Rust-specific language concepts of mutability and ownership, in addition to learning Rust syntax. Our results show that users spend 42% of online time viewing example code and that programmers appreciate the Rust Enhanced package's in-line compiler errors, choosing to refresh every 30.6 seconds after first discovering this feature. We did not find any significant correlations between the resources used and the total task time or the learning outcomes. We discuss these results in light of design implications for language developers seeking to create resources to encourage usage and adoption by experienced programmers.

Author Keywords

Programming Languages; Rust, Learning; Computer Science Education.

ACM Classification Keywords

D.3.m [Programming Languages]: Miscellaneous

INTRODUCTION

Once programmers have learned one programming language, they are relatively comfortable picking up additional languages on their own. Many languages share common control flow concepts and some underlying syntax. However, they often have huge variations due to differences garbage collection mechanisms, typing systems, and other principles. In this study, we seek to answer the following questions: (1) What resources do experienced programmers use when learning a new language? and (2) How can developers better design tools that support language learning and adoption?

When learning a new programming language, the Internet is an invaluable tool for programmers. With the help of documentation sites, example code (e.g., Github), question and answer sites (e.g., Stack Overflow), and other online resources, in addition to IDE plugins and compilers errors, experienced programmers are largely able to pick up new languages independently.

Our objective is to inform the design of tools that aim to better support programmers in the process of learning a new programming language. We are primarily interested in what resources users consult and how users leverage Rust's compiler errors and Sublime Text's Rust Enhanced package.

In this paper, we contribute (1) a comprehensive list of resources accessed when learning Rust and metrics of their relative perceived utility, (2) an analysis of the impact of informative compiler errors on language learning, and (3) suggestions for language creators and programming tool developers regarding which resources to dedicate energy toward creating.

RELATED WORK

Programming languages have been studied extensively from a technical point of view, focusing on values such as certainty and efficiency [3]. However, programming languages are an interface between programmers and computation carried out by the program [7], and as such it is important to also study programming languages as a socio-technical system. In this work we focus on the process of learning a new programming language and the resources programmers use during this process.

Language Features

Many factors affect the learning and adoption of programming languages, such as expressivity, correctness, and performance. Prior work has shown that when selecting a programming language, performance does not influence the developers' decision and that developers prioritize expressivity over correctness when selecting a language for a task [6]. In addition to these language features, syntax and notation influence both learnability and adoption of a new programming language. Previous empirical studies have evaluated the effects of syntax, such as the choice of keywords, used in different programming languages [12, 11]. In this work, we study the learnability of the Rust programming language. We focus on both language concepts, such as ownership and mutability, as well as notation and syntax.

Rust

Rust is a systems programming language sponsored by Mozilla Research, which focuses on safety [10]. In this work, we study the process of learning the Rust programming language. We chose Rust for two main reasons. Firstly, it is syntactically similar to C and C++; these are popular programming languages and experienced programmers are likely to be at least moderately familiar with the notation. Moreover, Rust is a relatively new language; therefore, most programmers lack prior experience programming in this language.

In addition to syntax, we focus on two critical concepts in the Rust programming language: mutability and ownership. Rust uses variable bindings, a concept used to bind a value to a name such that it can be used later. As mentioned earlier, safety is one of Rust's priorities; therefore, these bindings are immutable unless specified explicitly. In order to achieve the goal of memory safety, Rust also has a distinct ownership system. Variable bindings in Rust have ownership of what they are bound to, and, as a consequence, when a binding goes out of scope Rust will free the bound resources. Rust ensures that there is exactly one binding for a given value, meaning that passing a variable will also change its owner. In this study, we incorporate both mutability and ownership as concepts that we hope participants will learn during the task.

Rust also provides detailed compiler error messages and offers suggestions for bug fixes. The Rust Enhanced package [9] presents these compiler errors in-line, upon saving the file in Sublime Text 3. In this work, we study if programmers follow the compiler suggestions when debugging and whether or not this impacts how programmers learn the concepts described earlier or the Rust syntax.

Programming Tools

An array of tools have been proposed and designed that aim to improve the programmer's experience when learning a new programming language. These include programming tools that integrate Q&A sites and open-source libraries as part of the IDE [2] and tools that transform the way programmers debug their code by asking "why" and "why not" questions [4, 5]. However, it is unclear what resources programmers utilize when learning a new programming language and which of those they find most useful.

It is critical to design tools that align with the programmers' mental model [1]. Empirical evaluations, such as this work, that observe developers during the completion of a programming task can inform the design of these tools and also uncover surprising findings. For example, Nienaltowski et al. found that when designing compiler errors, more detailed messages do not correlate with a better understanding of errors, but rather that the placement of the information and its structure are more important [8]. Similarly, we believe insight about the process of learning a new programming language can help drive future directions of research in this area.

METHODS

Our study consisted of an exercise in the Rust programming language covering the concepts of mutability and ownership. This section describes our participants, experimental setup, task, procedure, and coding methods.

Participants

A total of 10 experienced programmers (5 male, 4 female, 1 non-binary) were recruited from the Stanford student body. Participants all reported themselves to be proficient in at least 2 programming languages, or expert in at least 1. All participants were at least familiar with one of C or C++ and none of the participants had ever programmed using the Rust programming language.

All participants provided consent to participate and to be screen and audio recorded for the duration of the study. Participants did not receive compensation.

Experimental Setup

The study took place on a 13" MacBook Pro computer in a quiet space. Participants were presented with two side-by-side windows. One window contained a browser, the other contained the Rust starter code (in Sublime Text 3, with the Rust Enhanced Package) with comments detailing task instructions, a text file detailing how to execute the code, and a terminal window in the source file directory (see Figure 1). This computer had both screen and audio recording software running for the duration of the task.

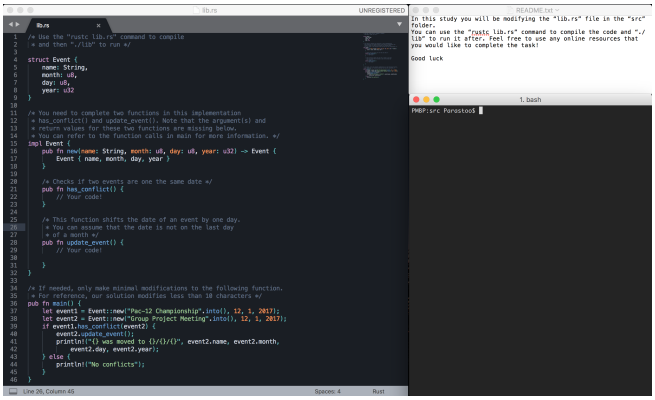

Figure 1. The layout of the different elements during the programming task. Sublime Text 3 on the left with the starter code, TextEdit with the README file on the top right corner, and the terminal for compiling and running the code on the bottom right corner.

Task

For this empirical study, the task had to be simple enough to be completed in less than 40 minutes, but also had to challenge programmers, allowing us to study how they use resources and debug in a language they are unfamiliar with. The task also needed to be unique such that participants would not be able to find solution code online. According to these criteria, we designed a task in which users had to write only a few lines of code, but during which they would encounter both mutability and ownership issues. We iterated on multiple task ideas such as the Collatz Conjecture and a modified version of the Caesar-Cipher, but ultimately decided on a simple struct manipulation task instead.

Users were given starter code (see Figure 2) that declared an Event structure, and their task was to complete the implementation by finishing two functions; one had to check if two events were on the same date and the other needed to shift the date of an event by one day. Note that to simplify the logic, participants were told that they could assume no events would be on the last day of the month. The arguments and the return types for these two functions also needed to be filled in. Participants were told that, if needed, they could make minimal modifications to the main function by adding or changing less than 10 characters.

```rust
struct Event {
    name: String,
    month: u8,
    day: u8,
    year: u32
}

impl Event {
    pub fn new(name: String, month: u8, day: u8,
               year: u32) -> Event {
        Event { name, month, day, year }
    }
    pub fn has_conflict() { }
    pub fn update_event() { }
}

pub fn main() {
    let event1 = Event::new("Event 1".into(),
                            12, 1, 2017);
    let event2 = Event::new("Event 2".into(),
                            12, 1, 2017);
    if event1.has_conflict(event2) {
        event2.update_event();
        println!("{} was moved to {}/{}/{}",
        event2.name, event2.month,
            event2.day, event2.year);
    } else {
        println!("No conflicts");
    }
}
```

Figure 2. The body of the starter code file, without comments, that participants were provided with and were asked to modify for the programming task.

Procedure

Participants were asked to first complete a pre-study survey, collecting demographic information and details of the participant's programming experience. After a brief introduction on how to work with the experimental computer, participants were asked to complete the programming task. Participants were allowed to use any online resource and had 40 minutes to complete the task. Upon finishing, participants completed a short quiz (see Figure 3 and Figure 4) to test their comprehension of mutability and ownership, Rust concepts that they encountered in writing the required code. We also collected qualitative feedback regarding resources participants found most useful and their impression of the Rust Enhanced compiler errors in Sublime. In total the study took roughly one hour to complete.

```rust
fn main() {
    let x = 2;
    x += 1;
    println!("{}", x);
}
```

Figure 3. The code for the first follow-up quiz question, in which participants had to find the missing "mut" declaration.

```rust
struct Point {
    x: i32,
    y: i32
}

impl Point {
    fn add(self, other: &Point) {
        self.x += other.x;
        self.y += other.y;
    }
}

fn main () {
    let mut p1 = Point { x: 2, y: 3};
    let p2 = Point { x: 1, y: 2};
    p1.add(p2);

    println!("P1 x: {}, y: {}", p1.x, p1.y);
    println!("P2 x: {}, y: {}", p2.x, p2.y);
}
```

Figure 4. The code for the second anf third follow-up quiz questions, in which participants had to find the missing "&mut" before the self in the add parameters as well as the missing "" in the "p1.add(p2)" line. declaration.

Coding

For each participant we coded for the following:

1. Google Searches

Participants accessed all resources through Google searches. We chose to separately measure the amount of time spent searching Google for a) syntax help, b) Rust compiler errors, c) Rust concepts, and d) direct lines of code from the starter file. The sum of these four values is equal to the total time spent on Google. We used this value to approximate the amount of time users spend searching for information, rather than consuming or utilizing it.

2. Documentation

Many languages' primary resource is their documentation. While Rust, notably, has several additional reliable resources online, we still tracked how much time each user spent accessing the documentation.

3. Example Code

Rust has a comprehensive example code website (RustByExample), in addition to having readily available open source Github code posted by other users. We hypothesized that experienced programmers might choose to learn a new programming language through these examples rather than documentation or tutorials, so we coded for the amount of time spent on these example code sites.

4. Q&A Sites

Stack Overflow and other online question and answer site are invaluable resources for programmers, both novice and expert. As such, we hypothesized all participants would leverage these sites and kept track of the time spent viewing them.

5. Use of the Compiler
Rust is notable for its descriptive and helpful compiler errors, which are made even more accessible (as in-line errors within Sublime Text) via the Rust Enhanced package. We hypothesized that, once users discovered this feature, they would use it regularly. We recorded how many times, and how often, users accessed these in-line errors by saving within sublime, how many times they compiled the code through Terminal, and how many times they executed the code. We also marked when participants used the compiler error messages to inform a bug fix and when they chose to copy exactly what the compiler messages suggested.

6. Total Task Time
We tracked the overall time spent completing the task to normalize rate of occurrence of the above events across participants.

RESULTS & DISCUSSION

In this section, we summarize the findings of the user study. We first list all online resources utilized by participants and the duration of their usage. We then report the learning outcome regarding the concepts of immutability and ownership describe, as well as the use of compiler errors and their effects on these learning objectives. Finally, we study factors that may have influenced the total task time.

Use of Resources

Combined, participants used four primary resources—Rust Documentation, Stack Overflow, RustByExample, Github—and several secondary resources such as blogs and other Q&A forums, which we have grouped as "Other Example Code" and "Other Q&A Site" respectively.

Participants spent the majority of their online time looking at example code, with RustByExample, Github, and Other Example Code accounting for 43% of time spent on the Internet. Stack Overflow and other Q&A sites account for 22% of online time, with time spent looking at the official Rust documentation clocking in at 9%. In addition, participants on average spent more than a quarter (26%) of their online time searching for the desired resources on Google (see Figure 5).

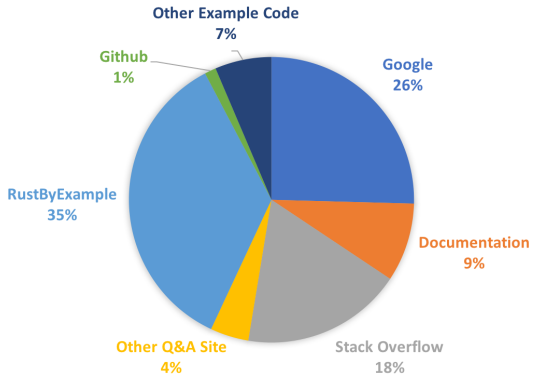

Figure 5. The breakdown of the ratio of time spent on online resources, averaged across all participants.

To reveal how participants were searching for resources, we further divided the Google search time into specific subcategories (see Figure 6). By and large, participants searched for syntax help (57% of searches) and for more information regarding compiler error messages (24%). The remainder of searches were for conceptual principles (15%) or were direct searches of the task starter code to look for online solutions (3%).

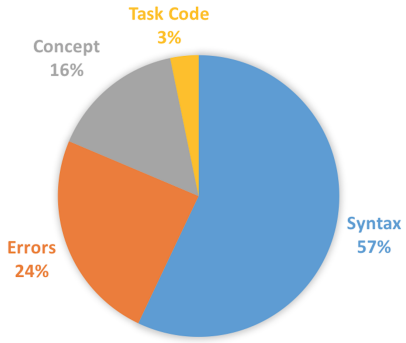

Figure 6. The breakdown of the ratio of time spent on different types of Google searches, averaged across all participants.

Qualitatively, half of the participants reported finding example code to be amongst the most helpful resources, and four participants pointed to Stack Overflow. Interestingly, only one of the participants thought that Rust's official documentation was one of the most useful resources. This breakdown might suggest that language developers should dedicate time to creating thorough example code and to building an online community.

Use of Compiler Errors

Participant behavior indicates that they found the in-line Rust Enhanced compiler errors that showed upon saving to be quite useful. Participants were not told about this feature, and only discovered it, on average, 1058 seconds (17 minutes, 38 seconds) into the task. However, after finding the Rust Enhanced compiler error feature, participants began saving every 30.6 seconds, on average, with an average of 24.8 total saves per person (see Table 1 for a more detailed breakdown).

ID	Time at First Save (s)	Average Time Between Saves (s)	Total Number of Saves
1	1006	22	38
2	1051	68	21
3	1027	11	37
4	1315	32	35
5	1323	9	12
6	302	13	17
7	1142	34	29
8	1090	35	24
9	1016	50	28
10	1310	32	7

Table 1. Participants used the Rust Enhanced save feature very often, but only after first discovering it.

This increased frequency of use suggests that participants found the in-line compiler errors to be a helpful sanity check for their code. Indeed, we observed participants blindly copy or otherwise follow the compiler's suggestions on average 6.9 times throughout the study (min: 1, max: 14).

Our participants also generally expressed positive sentiments towards the Rust Enhanced in-line compiler errors, with seven out of ten participants mentioning they appreciated the feature in the post-study qualitative survey, and three of these participants believed the compiler errors to be the most helpful resource. For example, P10 said: *"I want [in-line compiler errors] for all my programming (maybe it's there and I haven't set it up before). Overall [it's] incredibly useful situating the errors in the actual code (rather than seeing a print out in the terminal of a snippet and needing to find and match that to the code). [It] was so much easier."*

However, some participants were worried they relied on these errors too extensively without fully understanding them, and often wandered down the incorrect path as a result. P2 pointed out the in-line errors were: *"Helpful in general, but also can lead you astray if you really don't know much about a given error."* And P5 expressed: *"[The errors are] full of terminology that is hard to understand for new learners. I need external resources like Google to figure them out. And they are overwhelming."*

Rust Enhanced's in-line compiler errors can be a useful resource to users that are familiar with the underlying concepts causing the relevant errors. However, if the error is related to an unfamiliar idea (e.g. ownership), then the programmer does not know how to interpret or respond to the error.

Learning Outcomes

We did not find any correlation between blindly following the compiler errors and correctness in the post-study quiz. Participants were able to learn the concept of mutability from this programming exercise, with 9 of the 10 participants correctly identifying the source of the bug in part one of the post-study quiz, a missing "mut" in a let statement (see Figure 3).

However, participants were not all able to extend this understanding to mutable references. 6 of the participants recognized that the bug in question 2 required passing an "&mut self" rather than "self"; the other 4 participants understood the need to pass self by reference (&self) but did not see the necessity for mutability (see Figure 4).

Ownership was arguably the most novel paradigm encountered in the study. The post-study quiz required correcting a moved value error by passing a variable by reference rather than by value (see Figure 4). Only half of the participants were able to find and correct this bug.

Task Completion Time

We studied the effect of various factors on the time taken by participants to complete the task. More specifically, we studied the ratio of time spent on online resources over total task time, the ratio of time spent reading documentation over total internet time, the ratio of time spent consulting Q&A sites over total internet time, the ratio of time spent reading example code over total internet time, the ratio of time before the first save in Sublime Text over the total task time, the total number of saves, the average time between saves, and the self-identified expertise level of the programmer. We found no significant correlation between any of these factors and the task completion time.

DESIGN IMPLICATION

In-line Compiler Errors

Given the positive sentiments towards in-line compiler errors and the lowered work-flow overhead resulting from removing the requirement of compiling within the shell, we recommend language developers create packages to make compiler messages accessible in-line. This is consistent with prior work suggesting that the location of error messages and their structure is critical [8].

Compiler Error Linking

The Rust compiler links error messages to the relevant error within the Rust documentation, a page that relies on readers having a conceptual understanding of Rust. However, this linking is not always useful for programmers; as P9 said, *"That is not helpful. At all."* One way to make compiler error messages more helpful to all users would be to link the relevant conceptual information to the error message. This way, when users do not understand an error message, they are able to read about the underlying problem, rather than receive additional unhelpful information. Moreover, we found that participants often copied the compiler message directly into a search engine when the text of the error was unclear. One potential design suggestion for developers of IDEs, specifically, is to search the Internet for the compiler message and provide further clarifications based on the search results inside the programming tool itself.

Resource Variability

Programming languages have different sets of resources available online and these resources are presented at varying levels of quality. These variations might affect the type resources participants utilize when learning a new programming language. For example P10 mentioned during the task that *"a lot of times when I learn new programming languages they don't have one particular guide that has everything, but a lot of what I'm finding is all from the same RustByExample guide."* She followed up by saying that this felt different than how she learns other programming languages which may not have a *"coherent, unified, source of information."* However, it is worth noting that, relative to other participants, P10 did not take advantage of the compiler errors during the debugging process very much, as her first save was 3 minutes and 15 seconds before the completion of the task. Moreover, P10 spent 33.7% of the total task time on RustByExample, which is more than double the average amount of time spent on this site across all participants at 13.8%. Despite these caveats, this observation suggests that the online resources available for a particular language will affect which elements programming tools should incorporate.

Additionally, if the programming tool supports multiple languages, a potential design direction would be to evaluate the

quality and quantity of various resources available for each language. For example if a tool incorporates multiple sources, such as documentation and example code, the quantity of those resources can be determined by web scraping. The quality of the resources can be determined by observing how users utilize the presented information in real-time and prioritizing sources that they found most useful within the tool.

Importance of Example Code

In the Results and Discussion section, we found that participants spent the majority of the task time on RustByExample and looking through various example codes online. When looking at the total time spent on online resources, participants spent on average 42% of that time on RustByExample and other example code, while only 9% of the time was spent reading the documentation. Perhaps, this could suggest that in a programming tool, providing examples to the developer might be more helpful than explicitely defining functions. It should also be noted that RustByExample and other example code website did not contain example code in isolation, but that they often also provided explanations of concepts. Therefore, a combination of example code and very brief explanations might be an effective guide when learning a new programming language.

Notably, examples were especially helpful for learning the notation and syntax. A common question that participants had was trying to find out if "+=" was syntactically allowed in Rust. For example, P10 first wrote "self.day += 1;" in the text editor, then searched online and did not immediately find an example that used this notation, so she modified the code to "self.day = self.day + 1;". Later during the task she came across another piece of example code that had "+=", and she reverted back to that notation. During the initial stages of learning a new language, programming tools, such as Meta [2], could search for the syntax used by the developer in online example codes and warn the user if that particular notation could not be found in the search space.

LIMITATIONS & FUTURE WORK

During the study, participants used Google search to find various resources and to discover relevant information. One limitation of the empirical study conducted is that the resources utilized are affected by the order in which they appear in the Google search result. This may have influenced the online resources used by participants; however, we did not control for this condition as this would also be the case in practice. Another factor that may have biased the use of particular resources is the search history on our computer leftover by the experimenter and previous participants. We noticed this as P7 mentioned *"Oh, interesting! Search history!"* during the task. Future work should consider clearing all browsing history prior to every study session.

To control for the experimental setup and to ensure that the environment is working properly, we conducted the study on our personal computers. As a result, the task completion time was influenced by how familiar participants were with our setup. For example, P3 used keyboard shortcuts on Sublime Text that may have reduced the overall task time; however, other participants may not have had prior experience with this text editor. We also did not specifically recruit Mac users; therefore switching between the window containing the text editor and the one with online resources was likely easier for some participants than others due to familiarity.

Another limitation of this work is that the results may not be directly generalizable to other programming languages. As discussed previously, the quality and quantity of available online resources differs for each programming language. Therefore, the resources used for learning Rust may not apply to learning all programming languages. As discussed in the Design Implications section, however, a high level analysis of the resources used can reveal general trends that can be applied to other languages. Further decoupling the use of resources from the programming language itself is an interesting direction for future work.

Finally, in this work we study the process of learning a new programming language in a controlled environment during a 40-minute task. The findings provide insight as to what resources proficient programmers use and find helpful in their first hour of interaction with a new programming language. It would be valuable to conduct a longitudinal study and to observe how proficient programmers approach learning a new programming language for an unspecified task. The developers' actions can be logged and tracked overtime, to gain insight as to what resources they consult and how they approach fixing bugs outside of a controlled environment.

CONCLUSION

We studied the practices of ten experienced programmers learning to code in Rust, an unfamiliar programming language. By evaluating the resources these participants accessed, their use of in-line compiler errors, and the learning outcomes of the task, we observed that experienced programmers rely primarily on example code and QA sites when searching for information and make use of the in-line compiler errors extensively. Furthermore, while participants were almost universally able to learn the concept of mutability, they struggled to understand ownership. Utilizing these results, we generated four design implications for language developers seeking to create resources to encourage usage and adoption from experienced programmer: in-line compiler errors, compiler error linking, resource variability, and the importance of example code. With the number of programmers ever-growing, we hope this work will contribute to the future development of useful programming tools and online programming resources.

ACKNOWLEDGMENT

We would like to thank the course staff for their assistance throughout the quarter, and especially Will Crichton for his feedback on all aspects of this project. We would also like to thank our participants, who each volunteered an hour of their time to be a part of this study.

WORK DISTRIBUTION

Equal work was performed by both project members.

REFERENCES

1. Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. 2007. Aligning Development Tools with the Way Programmers Think About Code Changes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 567–576. DOI: http://dx.doi.org/10.1145/1240624.1240715

2. Ethan Fast and Michael S. Bernstein. 2016. Meta: Enabling Programming Languages to Learn from the Crowd. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 259–270. DOI: http://dx.doi.org/10.1145/2984511.2984532

3. Andrew Ko. 2016. A Human View of Programming Languages (Keynote). In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH Companion 2016)*. ACM, New York, NY, USA, 2–2. DOI: http://dx.doi.org/10.1145/2984043.2998390

4. Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. ACM, New York, NY, USA, 151–158. DOI: http://dx.doi.org/10.1145/985692.985712

5. Andrew J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 301–310. DOI: http://dx.doi.org/10.1145/1368088.1368130

6. Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical Analysis of Programming Language Adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 1–18. DOI: http://dx.doi.org/10.1145/2509136.2509515

7. Brad A. Myers, Andreas Stefik, Stefan Hanenberg, Antti-Juhani Kaijanaho, Margaret Burnett, Franklyn Turbak, and Philip Wadler. 2016. Usability of Programming Languages: Special Interest Group (SIG) Meeting at CHI 2016. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '16)*. ACM, New York, NY, USA, 1104–1107. DOI: http://dx.doi.org/10.1145/2851581.2886434

8. Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler Error Messages: What Can Help Novices?. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 168–172. DOI:http://dx.doi.org/10.1145/1352135.1352192

9. Daniel Patterson. 2012. Rust Enhanced. (2012). https://github.com/rust-lang/rust-enhanced

10. Rust. 2017. Rust Programming Language. (2017). https://www.rust-lang.org/

11. Andreas Stefik and Ed Gellenbeck. 2011. Empirical studies on programming language stimuli. *Software Quality Journal* 19, 1 (01 Mar 2011), 65–99. DOI: http://dx.doi.org/10.1007/s11219-010-9106-7

12. Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.* 13, 4, Article 19 (Nov. 2013), 40 pages. DOI:http://dx.doi.org/10.1145/2534973