An Empirical Investigation into Programming Language Syntax

ANDREAS STEFIK, University of Nevada, Las Vegas
SUSANNA SIEBERT, The Genome Institute, Washington University School of Medicine

Recent studies in the literature have shown that syntax remains a significant barrier to novice computer science students in the field. While this syntax barrier is known to exist, whether and how it varies across programming languages has not been carefully investigated. For this article, we conducted four empirical studies on programming language syntax as part of a larger analysis into the, so called, programming language wars. We first present two surveys conducted with students on the intuitiveness of syntax, which we used to garner formative clues on what words and symbols might be easy for novices to understand. We followed up with two studies on the accuracy rates of novices using a total of six programming languages: Ruby, Java, Perl, Python, Randomo, and Quorum. Randomo was designed by randomly choosing some keywords from the ASCII table (a metaphorical placebo). To our surprise, we found that languages using a more traditional C-style syntax (both Perl and Java) did not afford accuracy rates significantly higher than a language with randomly generated keywords, but that languages which deviate (Quorum, Python, and Ruby) did. These results, including the specifics of syntax that are particularly problematic for novices, may help teachers of introductory programming courses in choosing appropriate first languages and in helping students to overcome the challenges they face with syntax.

Categories and Subject Descriptors: D.3 [**Programming Languages**]; H.1.2 [**Information Systems**]: User/Machine Systems—*Software psychology*

General Terms: Design, Experimentation, Human Factors, Standardization

Additional Key Words and Phrases: Programming Languages, Novice Programmers, Syntax

ACM Reference Format:
Stefik, A. and Siebert, S. 2013. An empirical investigation into programming language syntax. *ACM Trans. Comput. Educ.* 13, 4, Article 19 (November 2013), 40 pages.
DOI: http://dx.doi.org/10.1145/2534973

1. INTRODUCTION

Despite decades of study, what makes a computer programming language easy to use for people of all skill levels remains elusive. While why is a subject of debate, a considerable number of programming languages are in common use in industry and the classroom. Many of these modern programming languages hold to a particular tradition (e.g., C or Lisp style syntax), and it is common for designers to vary syntax, semantics, or library design. These subtle variations on a syntactic theme are typically chosen by committees or technical experts.

One obvious reason why so many languages exist is that it is difficult to decide *how* to evaluate programming languages, let alone *which people* to evaluate. From our perspective, one community that stands to benefit most from controlled experiments on

This work is supported by the National Science Foundation, under grant CNS-0940521. It expands upon two original works [Stefik and Gellenbeck 2011; Stefik et al. 2011a].
Authors' address: A. Stefik (corresponding author), Computer Science Department, University of Nevada, Las Vegas; email: stefika@gmail.com.

DOI: http://dx.doi.org/10.1145/2534973

programming language design is students. Given the documented attrition rates in introductory programming courses [Beaubouef and Mason 2005], it seems reasonable to assert that novices face significant challenges when initially confronted with a general purpose programming language. Further, since novices must become comfortable with syntax in order to use a general purpose programming language, trying to identify where syntactic barriers exist may benefit instructors teaching various technologies. If the barriers can be identified in a specific enough way (e.g., which tokens, words, or symbols should be altered?) this may also benefit students in the long run as programming languages evolve.

Before we continue, however, we want to acknowledge what should be obvious to most reasonable instructors or computer scientists: in regards to the language wars, syntax is only one piece of an extraordinary puzzle. Many factors impact human performance in programming languages; our focus here is to isolate one feature that might impact students and to study it carefully using empirical methods. With that said, we think there are several compelling reasons why the study of syntax should not be ignored. First, Denny et al. [2011] conducted a recent empirical study showing that programming language syntax in Java remains a major barrier to students. In an experiment involving 330 students, Denny et al. found that students in the top quartile submitted non-compiling source code approximately half the time, whereas those in the bottom quartile submitted non-compiling code 73% of the time. The broad message of Denny et al.'s work is that even excellent students in an introductory programming course experience syntax issues. Later work has shown that syntax errors vary in how difficult they are for novices to solve [Denny et al. 2012].

Second, we have conducted a significant amount of previous work on tools for the blind and visually impaired. In that work, we observed that some programming languages were harder than others to "read" with a screen reader when used by a blind student [Stefik et al. 2011a]. When we considered the issue of syntax more deeply for our unique set of students, we thought it plausible that some languages chose words or symbols that were either unintuitive or seemingly arbitrary (e.g., ++ to increment the value of a variable, use of the word "for" to represent iteration, || to supposedly mean "or"). Given that we also have taught undergraduates in the classroom, we are suspicious that many syntactic choices in traditional C-style syntax (e.g., for loops, conditionals) may be hard for novices to understand or use. As some new languages, like Python, Ruby, or Quorum, have begun to abandon elements of C-style syntax, it seems reasonable to evaluate the impact of these choices on novices.

One might ask, however, whether a study of general purpose programming language syntax is worthwhile at all, given innovations by those developing syntax directed editors [Teitelbaum and Reps 1981], or more modern tools like Alice [Cooper 2010; Pausch 2008], Scratch [Maloney et al. 2010; Resnick et al. 2009], or perhaps end-user programming systems [Ko et al. 2011]. In this regard, it is crucial to recognize that visual tools do help novices initially (although not for the blind), especially in promoting transfer-of-learning from visual systems to text-based programming languages. This has been confirmed independently and using different methodologies with at least the tool ALVIS [Hundhausen et al. 2009] and Alice [Dann et al. 2012]. However, visual tools are not a silver bullet. Garlick and Cankaya compared using Alice in an introductory course to a control group that started with pseudo code, finding that the Alice group had a statistically significant drop in grades [Garlick and Cankaya 2010]; a very different kind of study with a very different outcome from the one discussed by Dann et al. [2012]. Regardless of the ongoing debate, despite decades of study on tools designed to bypass syntax, general-purpose programming languages are still overwhelmingly used in the classroom. Given that our goal as instructors is often

to prepare students for industry, which also overwhelmingly uses general-purpose programming languages, evaluating the impact of syntax on novices seems reasonable.

Given these considerations, we investigated programming language syntax in detail, with an eye toward evaluating competing designs with novice students. We limit our exploration to the following two research questions: (RQ1) Which words and symbols do novices find intuitive (or not intuitive) in a general purpose programming language, and (RQ2) Can novices using programming languages for the first time write simple computer programs more accurately using alternative programming languages? To analyze these broad research questions, we present here four formal empirical studies that we hope will help inform the debate for instructors, students, and language designers.

In the first two empirical studies, we explore how novices rated the intuitiveness of various programming language constructs. To obtain a reasonable sample of novices, we tested both those just beginning their academic career and those with more experience. This first study partially replicates, but greatly expands upon, previous work [Stefik and Gellenbeck 2011]. In a second study, we asked novices to subjectively rate the intuitiveness of larger program constructs in nine programming languages, including C++, Java, Smalltalk, PHP, Perl, Ruby, Go, Python, and Quorum. The purpose of these studies is to provide clues to instructors and language designers as to the kind of words, symbols, and phrases that may or may not make sense to a novice.

We explored our second research question by conducting two additional empirical studies, expanding on previous work [Stefik et al. 2011c]. In all, we had novices program using the languages Ruby, Java, Perl, Python, Randomo, and Quorum. For both works, we included the programming language Randomo, which was generated by starting syntactically with Quorum and then replacing the keywords with randomly chosen values from the ASCII table. Following a long history of using randomized controlled trials in the bio-medical sciences [Kaptchuk 1998], we hypothesize that languages such as Randomo may be useful to other researchers as a form of control group in programming language design studies. Generally, it seems reasonable to expect that novices using commercial programming languages would be afforded significantly higher accuracy rates than those using a language we created with a random number generator.

The overall findings from our experiments show evidence for the following key point: variations in programming language syntax matter for novices. More specifically, our data shows evidence that (1) some syntactic choices made in commercial programming languages are more intuitive than others, (2) variations in syntax influence novice accuracy rates when starting to program, and (3) using randomized controlled trials with placebo, and surveys, we can identify which features of syntax may cause the problems we observe with novices. These observations may be useful to instructors or language designers who need to evaluate which aspects of syntax novices might initially struggle with.

In the course of this article, we will first discuss related work. Then, we will move to the details of our formative surveys. We then discuss our final two experiments on novice accuracy rates, conduct a general discussion, examine broader threats to validity, summarize our overall findings, and conclude.

2. RELATED WORK

As programming itself is such a significant component of what computer scientists do, its study in the literature is only natural and has a rich history. We want to be clear in saying that a complete review of the usability of programming languages, let alone the rich history of argument, is outside the scope of this article. Our goal in this section

is to focus on the broad historical themes we have seen in regards to programming languages, especially their use by novices.

One important research line has been in evaluating the programming practices of novices or trying to make programming seem more natural [Bonar and Soloway 1983; Holmboe 2005; Lister et al. 2004; Lopes et al. 2003; Myers et al. 2008; Schulte and Magenheim 2005; Soloway et al. 1983]. See Kelleher and Pausch [2005] for a comprehensive review of novice programming systems or Ko et al. [2011] for a similar work with end users. With some similarities, the broad psychology of programmers is well studied. Some of the major areas of research have been general comprehension [Pennington 1987b], comprehension in industry relevant conditions (e.g., large scale maintenance [Mayrhauser and Vans 1997]), in regard to issues such as concept retrieval [Cleary et al. 2009], and work on the comprehension of auditory cues for the blind [Stefik 2008]. A number of authors have also established that programming language usage impacts productivity in either industry or the open-source community [Comstock et al. 2007; Delorey et al. 2007]. Other topics are studied by the Psychology of Programmers Interest Group (PPIG).

Besides analysis with novices, visualization has garnered significant interest in the literature [Dann et al. 2012; Hundhausen et al. 2009; Ko and Myers 2009; Garlick and Cankaya 2010]. Other researchers have examined sonification [Bigham et al. 2008; Sánchez and Aguayo 2005; Smith et al. 2004; Stefik et al. 2007; Vickers and Alty 2002], multimedia techniques [Brown and Hershberger 1991; Stefik and Gellenbeck 2009], or programming by voice [Begel and Graham 2004]. Researchers often create tools that surround a programming language, or altogether replace it, and look at the impact on users. Generally, a wide variety of tools (e.g., omniscient debuggers [Ko and Myers 2009; Lewis and Ducassé 2003; Pothier et al. 2007]) have been shown to be helpful to programmers (novice or otherwise) [Myers et al. 2004, 2008].

It is also important to acknowledge that human language is well studied in the area of psychology, for example, in the work of Pinker [1991] or as described in Whitney's textbook [Whitney 1998]. In terms of programming language usability, a wide variety of topics have been studied, such as the use of identifiers [Deißenböck and Pizka 2005], method naming [Høst 2007; Høst and Østvold 2007], coding standards [Binkley et al. 2009], or API designs [Stylos and Myers 2008]. The learnability of programming languages (e.g., Logo [Lukas 1972], Scheme [Findler et al. 2002], Smalltalk [Borning and O'Shea 1987]), the usability of a language [McIver 2001; Pane et al. 2001], or the relationship between programming and natural language [Delorey et al. 2009] have also been investigated.

While the previous few paragraphs provide a bird's-eye view of some of the literature in the area, we want to highlight several articles that we consider to be particularly relevant to the current work. For example, McIver focused her research on the programming language GRAIL in usability studies, making specific predictions about what kind of syntax and semantics would be sensible [McIver 2001]. The low-level thinking on specific constructs used in GRAIL, or in the work of Pane et al. [2001], influenced the analysis techniques used in the Quorum programming language. Further, in terms of the Pane study, while the work here cannot directly confirm or deny the hypotheses presented in that work, the use of study techniques such as Artifact Encoding and Token Accuracy Maps may allow future researchers to re-test such hypotheses in the context of general purpose languages, as opposed to the visual matching tasks these authors used.

Other language designers have also worked toward making languages that are intuitive or easy to use. For example, Holt et al. created the teaching programming languages SP/k [Holt et al. 1977] and Turing [Holt and Cordy 1988]. Smalltalk is also believed to have been adjusted according to usability evidence, however, the literature

discussing this largely just states that evidence was gathered, but does not go into detail about what that evidence was [Borning and O'Shea 1987]. Similarly, in the late 80's and early 90's, many researchers focused on analyzing either the comprehension of languages [Pennington 1987a] or qualitative analysis of a specific language (see Taylor [1990] for one of many examples). For a systematic review of such systems, see recent review articles [Kelleher and Pausch 2005; Ko et al. 2011].

A great deal of evidence on programming language usability can be found in the Psychology of Programmers Interest Group (PPIG) or the abandoned Workshop on Empirical Studies of Programmers (ESP). While a complete history of these two workshops is beyond our scope, much of the work from these workshops focused on understanding programmers. For example, Ramalingam and Wiedenbeck [1997] show an excellent quantitative experiment comparing comprehension with imperative and object-oriented styles. This study shows that language notations do influence novices, and, hypothetically, language designers can exploit some of these findings to improve their languages. In Quorum for example, as in many modern languages, object-orientation is hidden from beginners as optional syntax, taking advantage of this finding.

Much of the work prior to the mid-1990's eventually culminated into the cognitive dimensions framework by Green and Petre, later cited and expanded upon by a wide array of authors [Green and Petre 1996]. This model has been highly influential and has provided a reasonable, experienced, and thorough, set of heuristics. However, while this article held sway for more than a decade, many modern scholars interested in programming language usability have, it seems, been less likely to use or cite it. For example, recent studies have shown quantitative evidence that static typing affords faster programming times for human users [Kleinschmager et al. 2012; Mayer et al. 2012]. Studies like these often have less of a need for heuristics and more of a need for the types of randomized controlled trials seen in the bio-medical sciences, which are better suited for some kinds of very specific research questions (e.g., what is better for humans, static or dynamic typing?).

In regard to our discussion of the literature, we should point out that a number of modern authors have been highly critical of the existing literature in language design, including the cognitive dimensions framework. Perhaps the most consistent complaint seen in the modern programming languages literature is the lack of randomized controlled trials on programming language design. We sum up these positions by looking at the claims made by two authors, Markstrum and Hanenberg. First, Markstrum has looked carefully at the historical literature and his investigation reveals that new syntax or features are simply added to languages, usually with no data regarding human users [Markstrum 2010]. Second, Hanenberg argues that the entire discipline of language usability is based on Faith, Hope, and Love [Hanenberg 2010b]. In other words, both Hanenberg and Markstrum have documented that the language design community often does not use evidence at all; relying nearly exclusively on anecdotes. Evaluating these claims would require a detailed historical treatise and is not in the scope of our work. If validated, however, these are claims that should not be dismissed lightly.

While the discussion of historical work on language usability is interesting, the most recent trend, by far, is that the literature is transforming itself with the use of randomized controlled trials, a tradition our article falls in. While a number of articles could be cited, the broad lesson from carefully controlled experiments is the following: programming language design impacts users [Kleinschmager et al. 2012; Mayer et al. 2012; Ramalingam and Wiedenbeck 1997; Rossbach et al. 2010]. For example, Stylos has conducted a number of studies on alternative API designs and their impact on users [Stylos and Clarke 2007; Stylos and Myers 2008] (e.g., Stylos claims constructors should not require parameters). Evidence in the literature also suggests

that using locks leads to nearly seven times more bugs for novice students compared to transactions in parallel processing in the classroom [Rossbach et al. 2010]; an extraordinary finding. Another emerging trend is to consider the sociology of programming languages, especially in regard to adoption, taken on by Meyerovich and Rabkin [2012].

Finally, a common approach in education is to study language alternatives as they are used in the field (e.g., the classroom). While this is hardly new, one set of recent and interesting examples were conducted by Enbody et al., which studied the use of Python [Enbody and Punch 2010; Enbody et al. 2009]. The broad result showed no observable benefits for using Python in training students for a CS2 C++ course. While finding quantitatively reproducible results from field studies like this is difficult due to the considerable number of potential confounds, we do think that more of this "style" of work could be of significant value to the literature. For example, if curriculum/pedagogy can be partially isolated as an independent variable, variation on languages themselves might be detectable. If language design ultimately is evaluated in the field, however, lessons can be learned from Tew and Guzdial's FCS1 instrument [Tew and Guzdial 2011]. While not designed for testing the impact of language, this instrument may be one of the most rigorously evaluated assessment instruments in the educational computer science literature; the same procedures and techniques used in its creation could be adopted in studying languages.

Overall, the literature on programming language design is truly rich. We have cited a wide variety of articles that we feel have either influenced other researchers or that provide exemplars from various groups (e.g., PPIG). We want to be clear that a complete history of the programming language wars, even if the focus was exclusively on usability, could easily take up a book-length treatise. We strongly encourage readers interested in the topic to look at the rich previous work from many of the conferences and workshops we have mentioned to get a flavor of what exists.

3. STUDIES 1 AND 2: SURVEYS ON PROGRAMMING LANGUAGE SYNTAX

In this section, we present two empirical studies aimed at providing insight into our first research question.

*RQ*1. Which words and symbols do novices find intuitive (or not intuitive) in a general purpose programming language?

The broad idea in both studies is to give participants English explanations of various concepts and to ask them to subjectively rate how "intuitive" a series of word or syntactic choices relates to those concepts are. As the current article expands on a previously published pilot study (see Study 2 in Stefik and Gellenbeck [2011]), we contribute in Studies 1 and 2 in the following ways: we have added (1) a significantly larger number of computer science concepts, (2) all results presented here were conducted on a new sample, and (3) Study 2 presents, for the first time, ratings for larger syntactic constructs in nine programming languages.

3.1. Methodology

As both Studies 1 and 2 hold significant similarities, this section is organized as follows. First, we will describe the population we drew participants from for both studies. Next, we will discuss our materials and tasks broadly, then move to results for each experiment. Finally, we will present a discussion of both studies.

3.1.1. Participants. We solicited 196 participants, summarized in Table I, to take our surveys from a participant pool consisting of students enrolled in several classes at Southern Illinois University Edwardsville after appropriate Institutional Review

Table I.

This table provides an overview of the self-reported experience levels given by participants. #*P* = number of participants self-reporting greater than zero years of experience in a particular category (P = Programmer, NP = Non-Programmer). The languages PHP, Go, and Smalltalk were not included in the survey for study 1.

	Study 1						**Study 2**					
	#P		**programmers**		**non-programmers**		**#P**		**programmers**		**non-programmers**	
Category	*P*	*NP*	μ	σ	μ	σ	*P*	*NP*	μ	σ	μ	σ
Overall	84	0	2.57	2.47	0.00	0.00	93	0	2.38	2.26	0.00	0.00
C++	75	13	1.76	1.87	0.18	0.44	80	7	1.66	1.84	0.12	0.40
Java	31	4	0.46	0.71	0.09	0.48	29	2	0.39	0.70	0.03	0.16
Ruby	3	0	0.08	0.44	0.00	0.00	5	0	0.09	0.41	0.00	0.00
Python	7	0	0.11	0.41	0.00	0.00	6	0	0.08	0.34	0.00	0.00
PHP	—	—	—	—	—	—	11	1	0.28	1.06	0.01	0.12
Perl	2	0	0.03	0.22	0.00	0.00	1	0	0.01	0.05	0.00	0.00
Go	—	—	—	—	—	—	1	0	0.01	0.10	0.00	0.00
Smalltalk	—	—	—	—	—	—	1	0	0.18	1.76	0.00	0.00
FORTRAN	1	0	0.02	0.22	0.00	0.00	0	0	0.00	0.00	0.00	0.00
COBOL	3	1	0.05	0.26	0.01	0.11	3	0	0.03	0.18	0.00	0.00
Matlab	11	4	0.14	0.43	0.06	0.29	9	1	0.09	0.31	0.01	0.12
Basic	24	3	0.48	1.22	0.03	0.16	23	1	0.39	1.20	0.01	0.12
C#	12	0	0.22	0.71	0.00	0.00	12	0	0.17	0.63	0.00	0.00
JavaScript	19	1	0.68	1.74	0.05	0.44	22	0	0.67	1.68	0.00	0.00
HTML	40	8	1.62	2.93	0.23	0.86	45	5	1.56	2.79	0.17	0.78

Board ethics reviews for both Studies 1 and 2. Students came from a variety of courses in the computer science department, including freshman through junior and senior level courses that are taught in a variety of languages (e.g., C++, Java). For study 1, 166 participants decided to take the survey, giving us a high response rate of 84.7%. We classed participants according to their self-reported total years of experience into programmers and non-programmers, where non-programmers were counted as those reporting zero total years of self-reported experience. Given this grouping, study 1 included 84 programmers and 82 non-programmers. Programmers self-reported an average of 2.57 years of programming experience (SD = 2.47). Of the programmers, 11 were female and 73 were male, while of the non-programmers 17 were female and 65 were male. Participants in Study 1 reported an average age of 21.3 years. Two people reported being non-native English speakers.

Of the 196 participants solicited, 166 also chose to take Survey 2. Of these individuals, 93 were classed as programmers and 73 were classed as non-programmers. Of the programmers, 12 were female and 81 were male, whereas for non-programmers, 19 were female and 54 were male. A total of 7 people reported being non-native English speakers. Participants here also reported an average age of 21.3 years. Some of the individuals solicited chose to participate in both Study 1 and 2. As we were concerned this might influence the results, we pilot-tested Study 2 in a previous semester on a different sample. Further, as already stated, we previously published a pilot for Study 1 [Stefik and Gellenbeck 2011]. We found that our results here largely replicate previous work, meaning that participants rate the words/symbols/syntax in approximately the same way regardless of whether they took one survey or two. As such, we will not discuss this issue further.

We also classified participants in Study 2 as programmers if they self-reported any non-zero amount of total programming experience. Whether our measures of experience, self-reporting, or how individuals are classified, were the most appropriate, is not clear, but Feigenspan et al. published results from a survey of programming experience with similarities to ours [Feigenspan et al. 2012]. Our programmers self reported their

average years of programming experience (M = 2.38, SD = 2.26), along with their experience in a number of programming languages (see Table I).

While our non-programmers all reported zero years of total experience, we found that 22 individuals in Study 1 and 13 individuals in Study 2 made contradictory markings. In these cases, individuals marked that they had never programmed before in any language, but some had experience programming with an individual language. As we shall see, we found similarly contradictory markings in study 4, but in that case, interviews indicated that all participants with contradictory markings had never before programmed. In Studies 1 and 2, however, our online survey tool automatically anonymized the data to conform with ethics guidelines, which made such interviews impossible. Other researchers interested in whether or how contradictory markings play a role in intuitiveness measures can investigate the issue by analyzing our replication packet, which is available from the authors.

As a final issue related to our surveys, while versions of our language, Quorum (previously called Hop [Stefik and Gellenbeck 2011]), are freely available online today, we did not directly ask individuals to tell us whether they had used it. At the time, Quorum was not available as a standalone product, so we assumed our participants would not know about it. With that said, while it is unlikely that any of our participants had seen or used an early version of Quorum, in hindsight, we wish we had asked participants anyway. Finally, in all of our results tables, we use the labels *Programmer* and *Non-programmer*, by which we mean those with no total self-reported experience and those with greater than zero self-reported total experience.

3.1.2. Materials and Tasks. Participants used LimeSurvey, an open-source online survey tool, to enter their responses to our questions. After reading the task description, users subjectively rated words or syntax on a scale from 0 (0% intuitive) to 10 (100% intuitive). Figure 2 shows the rating interface for entering the values for one of the loops in Study 2 (Smalltalk, in this case). For Study 1, we focused the concepts we tested on words commonly used in many programming languages (e.g., for, if, static, or concatenation characters; i.e., "+ (Java)" or ". (PHP)"). In the second study, we focused the questions in our survey predominately on loops, `if` statements, or functions, because these features are common across many programming languages. Also in the second study, we allowed users to optionally enter a short explanation of their rating. We did not grade users' qualitative answers, but we did collect them to try and garner some insight into why users responded as they did. As other authors may wish to rephrase our questions, try new words or symbols, or create variations on our procedures, a complete replication packet, including PDFs, raw anonymized data, and the source code for our surveys, is available from the authors on request.

Designing robust, neutral, and clear surveys is not easy and we used a highly iterative process in designing ours. First, we designed and analyzed concept descriptions in small-scale pilots and had them reviewed qualitatively by experts. Second, once these initial reviews were complete, we submitted for peer review, and published a large-scale pilot study on an initial version of the survey [Stefik and Gellenbeck 2011]. After publication, we underwent several additional rounds of expert review as we expanded our survey, adjusting questions based on feedback and pilot data. Finally, after this already extensive approach, we ran several additional small-scale pilot studies to identify potential problems in the design. While we expect some readers, no matter how careful we were with the concepts, will disagree with our descriptions, the final choices made here have been 1) thoroughly vetted by a variety of experts in informal expert reviews, 2) scrutinized by novices and programmers in pilot studies, and 3) analyzed in formal anonymous peer review at an academic journal. Overall, we think the care we took in designing our survey is analogous to the level of care put into recent

educational research tools, such as Tew and Guzdial's excellent FCS1 instrument [Tew and Guzdial 2011]. The FCS1 has a very different purpose than our surveys, and Tew and Guzdial's procedures for statistical validation differ from those we used in pilot studies [Stefik and Gellenbeck 2011], but the level of care in design holds similarities.

3.1.3. Procedure. While our electronic survey was public, we tightly controlled access to prevent cheating or other problems with open online surveys. Participants would contact one member of our team, who would email a token to them. Participants would then complete our survey on a machine and browser of their choosing by entering the token, reading our informed consent form, and answering all of the questions. All data gathered was anonymized internally by LimeSurvey. Once all of our participants had completed the survey, we extracted the data from our online database and conducted statistical analysis in the programming language R.

3.2. Study 1 Results: Words and Symbols

In our first study, the goal was to document how humans subjectively rate the intuitiveness of word and symbol choices representing computer programming concepts. As we stated in previous work, our goal here is to gather formative clues as to what words and symbols might make sense to students. As should be obvious, we are not claiming that our survey somehow "proves" that a word or symbol is better than another; our survey is an investigative tool, not a proof. With that said, it seems reasonable to ask students what they think about language design choices, especially given how esoteric these choices sometimes are.

The survey for Study 1 had approximately six areas of exploration: (1) Types and Operators, (2) Control Flow, (3) Data Structures, (4) Object-Orientated Programming, (5) Input, Output, and Comments, and (6) Aspect-Oriented Programming. A selection of task descriptions for several of the questions asked can be found in Table II. In each subsection, we included a series of potential word or symbol choices regarding these concepts. Many of the words and symbols came from real programming languages, although we also included a number of choices that hypothetically *could* have been considered for these concepts. While any number of concepts could have been chosen, we investigated these six areas because, with perhaps the exception of Aspect-Oriented programming, they are common across programming languages. We decided to include word choices related to Aspect-Oriented programming both out of intellectual curiosity and due to the results of Hanenberg et al. [Endrikat and Hanenberg 2011; Hanenberg et al. 2009].

In terms of our analysis, we now present our results. In each case, the tables present the raw ratings given by novices. In previous work, we supplemented such figures with a rigorous statistical procedure, allowing us to rank the words in the aggregate [Stefik and Gellenbeck 2011]. However, given the size of our new survey in Study 1, repeating all of the questions and ratings would add considerable length to the current article. As such, we provide here the three highest-rated choices for each concept and the three lowest. For those interested in the exact minutia of every word we chose, raw or summarized data is available on request from the authors.

While we have categorized each question presented here to help give a broad overview of the themes of our survey, these themes were not presented to participants. Further, our intent was to understand how participants rated individual words in aggregate, not to detect relationships between our questions in the correlative sense. As such, a traditional factor analysis [Kline 2002] would not reveal the information we are trying to catalog and is not included here. Finally, while we have organized our tables, the questions were given to participants in what is often termed fixed random order (an order that is fixed ahead of time, but chosen through a random process).

Table II. Example Task Descriptions

Task	Concept	Task Description
43: Integer	A variable that holds negative or positive whole values	Rate the following words on how well they express the idea of a variable that holds negative or positive values used for expressing non-fractional amounts, *i.e.*, −5, 15, 0.
30: Assignment Operator	Assigning a value to a computer's memory	Suppose you wanted to write a mathematical expression that represented taking a number, perhaps the number 1024, and putting it into a location in a computer's memory represented by a variable named `x`. Rate each construction according to how well you think it represents assigning a value to a computer's memory.
6: If	Doing something after verifying that something is true	Suppose that something happens after verifying that something is true. For instance, suppose that something will happen after verifying that something named `x` is less than something named `y`. Rate the following on how well they express that the following code will execute after verifying that `x` is less than `y`.
3: Boolean Equals	Checking whether one thing is the same as another thing	Suppose you wanted to check whether something named `x` had the same value as something named `y`. Rate each expression according to how well you think it represents checking whether `x` and `y` have the same value.
32: Array	A variable that references one or more values in a row	Suppose you want to use one variable that holds one or more values in a row. For instance, suppose you have a variable named `x` that holds the values 8, 3, 10, and 6, one after the other. Rate the following words on how well they express that the variable `x` holds multiple values.
23: Generics	Specifying the data type of an element in a larger structure	Suppose a program contains a structure called `list` and you want the data type of the items in `list` to be of the type `dog`. Rate the following based on how well they express that the data type of the elements in `list` is being set to `dog`.
12: Class	Representing something in the computer that exists in the real world	Consider a dog. A dog can bark or run, and it has a height, weight, and color. We might define instructions for creating a representation of a dog in the computer. Rate how well each word represents this concept.
17: Method	A list of instructions that represents a tangible behavior, (e.g., walking at a certain speed)	Rate how well each word describes a list of instructions.
4: Public	A part of a computer program that can be completely accessed by another program	We often want to make certain parts of a computer program completely accessible by other parts. Rate each word on how well it represents making a thing completely accessible.
10: Input	Prompting the user for information	Rate the following words on how well they express that the user is being prompted to provide the program with some information.
29: Try	Attempting to run code that may not run correctly	Some code may or may not run correctly because of problems outside of your control (*e.g.*, the computer cannot find a file). Rate each word on how well it expresses that you are attempting to run a block of code that may not run correctly.
20: Pointcut	Symbolically representing various things in a document for later reference	Suppose you had a Microsoft Word document where, by mistake, you accidentally left the headings as paragraph text, but you meant to make them bold. Pretend that you can use a keyword in a programming language to indicate where all of the headers are so that later you can do something with them all at once (e.g., make them all bold). Rate each of the following words on how well they represent this concept.
22: Before	Automatically executing a list of instructions when certain parts of a program start running	Consider using a program that requires logging in, *i.e.*, an instant messaging client. Whenever you want to access your account, code automatically runs that ensures that you are logged in. Rate each word based on how well it expresses that you want code to automatically run when certain parts of a program start running.

3.2.1. Types and Operators. We asked a number of questions in our survey on concepts related to static types such as integers, floating point values, booleans, or strings. Table III shows the results for these questions. For the integer data type, programmers rated the word `integer` highly, while non-programmers rated the words `number` and `integer` as comparably intuitive. Regarding the concept of a floating point number, both programmers and non-programmers rated the word choices `decimal`, and `number` highly. For boolean types, programmers rated the words `boolean` and `bool` well.

Table III. Word Choice Results for Variable Types. (Word Choices with the Same Average Were Sorted Alphabetically)

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Integer	Non-programmer	number (6.63, 2.68), integer (6.46, 3.38), real (5.91, 3.22)	byte (3.67, 3.09), boolean (3.60, 3.03), cipher (3.11, 2.82)
	Programmer	integer (8.38, 2.42), number (6.20, 2.85), real (5.88, 3.09)	complex (2.76, 3.24), boolean (2.44, 3.11), cipher (2.44, 2.65)
Float	Non-programmer	number (6.70, 3.05), decimal (6.43, 3.31), rational (6.07, 3.25)	char (3.23, 3.22), single (3.21, 3.11), cipher (3.09, 2.98)
	Programmer	decimal (6.81, 3.16), number (6.79, 2.85), float (6.68, 3.45)	character (1.73, 2.60), char (1.64, 2.64), boolean (1.43, 2.17)
Boolean	Non-programmer	condition (5.32, 3.01), logic (5.26, 2.94), binary (5.11, 3.29)	complex (3.33, 2.89), double (3.29, 2.83), mode (3.18, 2.70)
	Programmer	boolean (7.68, 3.11), bool (7.63, 3.08), binary (6.25, 3.33)	complex (2.13, 2.63), real (2.12, 2.71), double (1.83, 2.58)
String	Non-programmer	characters (6.37, 2.76), text (6.15, 3.01), charstring (5.89, 2.90)	complex (4.29, 2.72), boolean (3.73, 3.07), ascii (3.52, 2.91)
	Programmer	string (7.48, 3.07), text (7.23, 2.58), charstring (7.21, 2.76)	byte (3.63, 3.21), alphabetic (3.58, 2.91), boolean (1.83, 2.86)
Enclosing Text	Non-programmer	"my words here" (7.57, 2.89), ``my words here``(7.09, 3.05), [my words here] (6.93, 2.90)	start my words here end (3.60, 3.05), $my words here$ (3.21, 2.91), ?my words here? (3.15, 2.75)
	Programmer	"my words here" (8.35, 2.71), ``my words here``(8.31, 2.55), 'my words here' (7.38, 2.75)	start my words here end (3.26, 2.88), $my words here$ (3.14, 2.57), ?my words here? (2.77, 2.28)

Table IV. Word Choice Results for Variable Manipulation. (Symbol Choices with the Same Average Were Sorted Arbitrarily)

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Assignment Operator	Non-programmer	x = 1024 (6.90, 3.01), x is 1024 (6.32, 3.02), x : 1024 (5.51, 3.01)	x += 1024 (3.52, 2.78), x + 1024 (3.16, 2.89), x / 1024 (3.12, 2.78)
	Programmer	x = 1024 (7.87, 2.93), x is 1024 (5.77, 3.12), x =: 1024 (5.45, 2.69)	x += 1024 (3.21, 3.24), x + 1024 (1.98, 2.30), x / 1024 (1.49, 2.11)
String Concatenation	Non-programmer	fire + fox (7.82, 2.79), fire & fox (6.40, 3.40), fire _ fox (5.77, 3.29)	fire ! fox (2.59, 2.49), fire ? fox (2.51, 2.82), fire $ fox (2.46, 2.61)
	Programmer	fire + fox (8.35, 2.41), fire & fox (6.48, 3.09), fire _ fox (4.71, 3.37)	fire - fox (1.94, 2.68), fire ? fox (1.94, 2.35), fire $ fox (1.67, 2.03)
Cast	Non-programmer	use t as Dog (5.90, 2.78), handle t as Dog (5.82, 2.96), apply Dog to t (5.46, 2.72)	t(Dog) (3.61, 2.75), t + Dog (3.13, 2.77), t^#-dog (2.67, 2.62)
	Programmer	handle t as Dog (6.15, 2.97), use t as Dog (5.99, 2.95), cast(t, Dog) (5.85, 2.64)	(t)Dog (3.63, 2.85), t + Dog (2.50, 2.67), t^#-dog (2.15, 2.33)
Modulus	Non-programmer	x = remainder of 7 / 2 (5.96, 3.01), x = remainder 7 / 2 (5.71, 2.99), x = 7 / 2 remains (5.55, 3.14)	x = 7 2 (3.44, 3.24), x = 2 % 7 (3.12, 3.10), x = 7 . 2 (2.85, 2.79)
	Programmer	x = 7 % 2 (6.86, 3.44), x = 7 mod 2 (6.75, 3.10), x = remainder of 7 / 2 (6.21, 3.03)	x = 7 / 2 (3.08, 3.20), x = 7 2 (2.70, 2.67), x = 7 . 2 (2.08, 2.31)
Increment	Non-programmer	x = x + 1 (7.06, 2.71), x + 1 (5.93, 2.97), raise x (5.49, 2.72)	x -= 1 (3.07, 3.15), x -- (2.82, 2.94), lower x (2.59, 2.78)
	Programmer	x = x + 1 (8.37, 2.36), x ++ (7.44, 2.73), x += 1 (7.14, 3.08)	x -- (1.62, 2.35), x -= 1 (1.44, 2.19), lower x (1.42, 1.95)

Non-programmers, on the other hand, presented little consistency in their ratings. Some of the highest ranked words were `condition`, `logic`, and `binary`.

The last variable type tested was for the string data type. Non-programmers rated the choices `characters`, `text`, and `charstring` highly. Programmers preferred the words `string`, `text`, and `charstring`. As not all programming languages use quotation marks for indicating the beginning and ending of a string, we also tested symbols used for enclosing text (e.g., single quotes, double quotes). Both programmers and non-programmers rated double straight quotes and double slanted quotes well.

Table V. Word Choice Results for Control Structures

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
If	Non-programmer	if (6.89, 2.78), when (6.59, 2.93), require (6.46, 2.63)	case (4.56, 3.02), do (4.52, 3.02), stop (3.72, 2.78)
	Programmer	if (8.37, 2.47), when (7.49, 2.64), whenever (6.86, 2.75)	only (5.30, 2.94), do (4.58, 3.28), stop (3.02, 2.81)
Loops	Non-programmer	repeat (6.88, 2.94), again (6.43, 3.05), loop (6.30, 3.20)	foreach (2.99, 2.97), while (2.37, 2.70), for (2.13, 2.83)
	Programmer	loop (7.88, 2.28), repeat (7.49, 2.70), cycle (6.65, 2.45)	duplicate (4.67, 2.82), foreach (4.35,3.02), echo (4.13, 2.87)

In Table IV, we asked questions about manipulating variables. Overall, both programmers and non-programmers rated the symbol = as the most intuitive choice for assignment (outside the 95% confidence intervals of all others with the exception of the word `is` (for non-programmers)). We also tested the concept of string concatenation and found that both programmers and non-programmers ranked the symbols +, &, and _ highly for this task. For the concept of casting static types, both programmers and non-programmers yielded little consensus.

We also tested the concepts of modulus and incrementing variables. First, when asked about the modulus operator, non-programmers rated `x = remainder of 7 / 2` and `x = remainder 7 / 2` as more intuitive compared to `x = 7 % 2` or `x = 7 mod 2`, which programmers rated as the most intuitive. Additionally, modern programming languages often allow shorthand to phrases like `x = x + 1`, such as `x++` or `++x`. In this case, both groups rated `x = x +1` highest, although the most typical shorthand (`x++`) was fairly close among our sample of programmers. While it may be a coincidence, non-programmers did not rate the `x++` shorthand highly which coincides with the results of Dolado et al. [2003], who documented empirical evidence that side-effect operators such as `x++` cause a statistically signifiant decrease in program comprehension.

3.2.2. Control Flow. Just as the use of static types and variables is common in programming, essentially all major programming languages have some form of program control flow. In this section, we present results in regard to word and symbol choices for conditionals, loops, and operators commonly used in conditional statements (e.g., == vs. =, || vs. *or*). The results for loops and conditionals can be found in Table V. For conditionals both programmers and non-programmers rated the word `if` highly, although several other choices were rated similarly well among non-programmers. For loops, programmers rated the word `loop` and `repeat` highly, while non-programmers rated `repeat` and several other words that may imply repetition highly (e.g., `again`, `loop`). Note that what are perhaps the three most common words for looping in computer science, `for`, `while`, and `foreach`, were rated as the three most unintuitive choices by non-programmers. This result replicates previous work [Stefik and Gellenbeck 2011].

We asked five questions in regard to boolean operators (Table VI). For testing equality, programmers rated the choice `x == y` as the most intuitive, which is not surprising, considering this is commonly used in many programming languages. Non-programmers, however, rated this choice highly, but lower than `x = y` or `x is y`. For the concept of not equal to, Programmers rated the choice `x != y` as the most intuitive, which is also used in many programming languages. Non-programmers rated the choices `x unequal y` and `x not= y` as most intuitive, which were also rated comparatively highly by programmers. When rating the choices for the concept of *boolean and*, both programmers and non-programmers rated the word `and` as the most intuitive, followed by `&` and `&&`. Similarly, the word choice rated as the most intuitive to represent a *boolean or* was the word `or`. When asked to rate word choices for the

Table VI. Word Choice Results for Boolean Comparison Operators

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Boolean Equals	Non-programmer	x = y (8.15, 3.10), x is y (7.87, 2.84), x == y (7.04, 3.29)	x :? y (3.37, 2.88), x ?: (y 3.15, 2.75), x >< y (2.90, 2.72)
	Programmer	x == y (8.64, 2.20), x isEqual:y (7.17, 2.66), x is y (6.81, 3.17)	x <-* y (3.14, 2.84), x <> y (2.12, 2.52), x >< y (2.02, 2.36)
Not Equal To	Non-programmer	x unequal y (6.32, 2.96), x not= y (5.84, 3.16), x =\\= y (4.16, 3.32)	x / y (2.74, 2.91), x == y (2.57, 3.11), x = y (2.43, 3.10)
	Programmer	x != y (7.61, 2.77), x not= y (6.54, 2.87), x unequal y (6.45, 3.12)	x / y (1.92, 2.64), x = y (1.90, 2.78), x == y (1.85, 2.67)
And	Non-programmer	x and y (8.29, 3.09), x & y (8.15, 2.89), x && y (6.61, 3.03)	only x or y (1.98, 2.64), either x or y (1.89, 2.84), x nor y (1.56, 2.55)
	Programmer	x and y (8.85, 2.42), x & y (8.65, 2.38), x && y (7.99, 2.80)	either x or y (1.65, 2.48), only x or y (1.62, 2.45), x nor y (1.46, 2.37)
Or	Non-programmer	x or y (6.28, 3.53), either x or y (5.95, 3.37), x and y (5.41, 3.45)	x v y (3.94, 3.17), x ^ y (3.54, 2.79), x nor y (2.13, 2.86)
	Programmer	x or y (7.60, 3.17), either x or y (6.75, 3.40), x ||y (5.93, 3.73)	x ^ y (3.49, 3.47), x exclusive and y (3.38, 3.54), x nor y (1.67, 2.44)
Xor	Non-programmer	either x or y (6.29, 3.42), x or y (6.24, 3.41), only x or only y (6.20, 3.53)	x and y (3.15, 3.53), x exclusive and y (2.99, 3.00), x nor y (2.91, 3.06)
	Programmer	only x or only y (7.86, 3.21), either x or y (7.20, 3.24), x xor y (6.33, 3.42)	x & y (1.82, 2.70), x && y (1.58, 2.54), x and y (1.46, 2.69)

Table VII. Word Choice Results for Arrays. (Word Choices with the Same Average Were Sorted Alphabetically. Symbol Choices with the Same Average Were Sorted Arbitrarily)

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Array	Non-programmer	value series x (5.60, 2.78), value string x (5.60, 3.06), value arraylist x (5.35, 2.59)	value x (3.79, 2.90), value &x (3.34, 2.77), value @x (3.24, 2.63)
	Programmer	value x[] (6.36, 2.84), value array x (6.21, 2.64), value arraylist x (5.52, 2.67)	value x (3.17, 3.06), value @x (2.96, 2.63), value &x (2.82, 2.73)
Index Operator	Non-programmer	packingList:5 (6.66, 2.75), packingList(5) (6.48, 3.05), packingList[5] (6.34, 3.08)	packingList&5 (3.85, 2.99), packingList!5 (3.68, 2.81), packingList$5 (3.33, 2.70)
	Programmer	packingList[5] (8.40, 1.98), packingList(5) (6.90, 2.79), packingList at 5 (6.58, 2.66)	packingList&5 (3.33, 2.52), packingList$5 (3.32, 2.50), packingList!5 (3.02, 2.33)

programming concept of an *exclusive or* both groups ranked more verbose choices, like `either x or y` and `only x or only y`, as intuitive. Non-programmers also ranked the syntax `x or y` as intuitive, implying that these individuals probably did not fully grasp the concept. This result replicates previous work [Stefik and Gellenbeck 2011].

3.2.3. Data Structures. Table VII shows the results for two questions concerning the programming language concepts of an *array* `x` of type `value` and the index operator used to access one of the elements in an array. For the first, results for non-programmers were inconclusive. Programmers rated the choice `value x[]` as the most intuitive, corresponding to C, which most of our programmers were familiar with. For retrieving the element on position `5` from the array `packingList`, non-programmers also showed no clear preference. Programmers, however, rated the syntax `packingList [5]` well.

As data structures in statically typed programming languages often use generics, we tested these preferences in our study. We differentiated between generics of one data type and generics of multiple data types (Table VIII). For the first, we asked participants to rate syntax for a generic structure `list` with elements of type `dog`. Generally, we find the results on generics to be inconclusive. If an intuitive syntax exists for representing generics, we did not appear to find it.

Table VIII. Word Choice Results for Generics

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Generics	Non-programmer	list[dog] (6.27, 2.55), list(dog) (6.21, 2.56), list{dog} (5.96, 2.55)	list using dog (4.59, 2.81), list-dog (4.44, 2.69), dog list (3.91, 2.83)
	Programmer	list<dog> (6.49, 2.46), list[dog] (6.39, 2.78), list(dog) (5.77, 2.92)	dog list (4.71, 3.22), list->dog (3.99, 2.78), list-dog (3.60, 2.61)
Multiple Type Generics	Non-programmer	books(shelfnumber, title) (6.17, 2.58), books[shelfnumber, title] (6.05, 2.63), books{shelfnumber, title} (5.72, 2.74)	books-shelfnumber, title (4.30, 2.65), books->shelfnumber, title (4.24, 2.70), shelfnumber, title books (4.05, 2.98)
	Programmer	books[shelfnumber, title] (6.67, 2.65), books(shelfnumber, title) (6.51, 2.75), books{shelfnumber, title} (6.25, 2.62)	books->shelfnumber, title (4.14, 2.43), books-shelfnumber, title (3.85, 2.59), shelfnumber, title books (2.98, 2.70)

Table IX. Word Choice Results for Classes and Inheritance. (Word/Symbol Choices with the Same Average Were Sorted Arbitrarily)

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Class	Non-programmer	object (6.21, 2.76), structure (5.87, 2.77), framework (5.84, 2.70)	interface (4.18, 3.03), instance (3.46, 3.04), article (3.35, 2.90)
	Programmer	object (6.95, 2.61), structure (6.76, 2.65), class (6.58, 2.85)	interface (4.06, 2.75), essence (3.85, 2.60), article (2.98, 2.47)
Is a	Non-programmer	truck is a vehicle (6.56, 2.92), truck : vehicle (5.95, 2.65), truck is vehicle (5.88, 3.08)	truck using vehicle (3.57, 2.70), truck overrides vehicle (3.45, 2.75), truck eats vehicle (2.15, 2.69)
	Programmer	truck is a vehicle (7.38, 2.48), truck is vehicle (6.57, 2.60), truck inherits vehicle (5.89, 2.97)	truck imitates vehicle (3.74, 2.57), truck >> vehicle (3.74, 2.93), truck overrides vehicle (3.70, 2.63), truck eats vehicle (1.26, 1.93)
Parent	Non-programmer	source (5.65, 3.00), foundation (5.52, 2.87), parent (5.40, 2.84)	central (4.15, 2.65), remote (4.07, 2.72), uncle (3.95, 2.87)
	Programmer	source (5.88, 3.03), parent (5.77, 3.02), foundation (5.37, 3.09)	prime (3.67, 2.95), uncle (3.17, 2.96), remote (2.96, 2.62)
This	Non-programmer	self (6.52, 3.17), myself (6.37, 3.14), me (6.26, 3.19)	residential (3.79, 2.80), destination (3.66, 3.10), pointer (3.43, 3.01)
	Programmer	self (7.61, 2.51), me (7.01, 2.90), myself (6.88, 2.93)	residential (3.14, 2.68), destination (3.00, 2.98), constant (2.60, 2.92)
Abstract	Non-programmer	blueprint (5.93, 3.04), frame (5.91, 2.70), model (5.65, 3.02)	virtual (3.85, 2.72), blank (3.68, 3.02), void (3.59, 2.97)
	Programmer	prototype (6.42, 2.79), frame (5.95, 2.87), blueprint (5.92, 2.79)	blank (3.70, 2.75), void (3.60, 2.65), empty (3.50, 2.64)

3.2.4. Object-Orientated Programming. Multiple questions in this study covered concepts relevant to object-oriented programming (see Table IX). First, for the concept of a *class*, both programmers and non-programmers rated several words highly, including the word `object` and `structure`. In regard to inheritance, we asked the participants to rate different word choices that describe an *is-a* relationship between two classes: `truck` and `vehicle`. Both groups rated the syntax `truck is a vehicle` as most intuitive. Interestingly, C++'s operator for inheritance, colon, was rated within the statistical margin of error of the English words `is a` by non-programmers. For the concept of a reference to a super class, both programmers and non-programmers rated the words `source`, `parent`, and `foundation` comparably well. Many programming languages use a word such as `this` to represent the current object (e.g., Java). We tested this concept, finding the words `self`, `myself`, and `me` to be rated well overall. There was little consensus for the concept of abstract classes, although ratings for words such as `blueprint`, `frame`, and `prototype` appeared relatively favorable.

Table X shows the results for three questions that cover programming language concepts related to methods. Both groups rated the words `procedure`, `operation`, and `action` as fairly intuitive in describing the concept of a method. In previous work [Stefik and Gellenbeck 2011], we used a different phrasing of the concepts for this

Table X. Word Choice Results for Function Handling

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Method	Non-programmer	procedure (6.90, 2.65), operation (6.54, 2.67), task (6.40, 2.73)	service (4.35, 2.86), enactment (4.07, 2.72), stuff (2.96, 2.56)
	Programmer	procedure (7.29, 2.59), action (6.98, 2.54), operation (6.83, 2.64)	duty (4.08, 2.64), service (3.94, 2.57), stuff (1.98, 2.30)
Return Type	Non-programmer	results in dog (5.80, 2.85), gets dog (5.57, 2.65), produces dog (5.55, 2.85)	elicits dog (3.61, 2.73), begets dog (3.57, 2.64), rejects dog (3.41, 2.80)
	Programmer	returns dog (7.10, 2.41), results in dog (6.65, 2.68), produces dog (5.85, 2.56)	surrenders dog (3.23, 2.75), effects dog (3.13, 2.35), rejects dog (1.98, 2.55)
Return	Non-programmer	provide (6.18, 2.93), report (6.13, 2.77), return (5.96, 2.85)	pass (4.27, 2.82), sub (3.82, 2.58), remit (3.68, 2.77)
	Programmer	return (7.54, 2.45), report (6.57, 2.93), provide (6.55, 2.77)	mention (4.54, 2.58), sub (3.89, 2.52), remit (3.76, 2.51)

Table XI. Word Choice Results for Properties

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Public	Non-programmer	public (8.22, 2.48), open (8.15, 2.76), unrestricted (7.70, 2.77)	hidden (1.45, 1.98), closed (1.44, 2.54), private (0.93, 2.04)
	Programmer	public (9.23, 1.32), unrestricted (8.26, 2.03), open (8.05, 2.52)	restricted (1.24, 2.22), closed (0.86, 1.91), private (0.81, 2.11)
Private	Non-programmer	private (8.48, 2.52), restricted (8.06, 2.66), closed (7.87, 2.38)	public (1.54, 2.49), unprotected (1.45, 2.06), open (1.21, 2.22)
	Programmer	private (9.12, 1.87), restricted (8.62, 1.83), closed (7.87, 2.30)	unprotected (0.90, 1.53), exposed (0.89, 1.55), public (0.85, 1.72)
Protected	Non-programmer	semiprotected (7.13, 2.89), usable (5.12, 2.88), special (4.98, 3.08)	hidden (3.56, 2.69), private (2.93, 2.81), closed (2.45, 2.83)
	Programmer	semiprotected (7.85, 2.58), usable (5.94, 2.74), special (5.79, 3.23)	hidden (3.00, 2.47), private (2.65, 3.09), closed (1.75, 2.49)
Constant	Non-programmer	permanent (8.27, 2.65), constant (8.05, 2.92), const (6.83, 2.87)	integer (3.22, 2.94), variable (2.43, 2.97), writable (2.17, 2.25)
	Programmer	constant (9.19, 1.97), const (8.43, 2.53), permanent (8.23, 2.44)	integer (2.11, 2.43), writable (1.33, 1.87), variable (1.18, 1.96)
Static	Non-programmer	constant (6.74, 2.66), permanent (6.73, 2.59), fixed (6.45, 2.65)	resilient (3.74, 2.65), latent (3.26, 2.77), volatile (2.98, 2.69)
	Programmer	fixed (7.31, 2.42), constant (7.26, 2.86), static (6.62, 2.74)	idle (3.61, 2.82), latent (2.83, 2.62), volatile (2.62, 2.80)

question, although our findings are essentially the same. When describing the data type of the return value, which in our examples is of type `dog`, non-programmers rated a host of syntactical choices comparably. Programmers predictably rated `returns dog` highly, although they rated `results in dog` as comparable. We also asked about obtaining results from functions (returning), for which non-programmers rated a number of choices around six, including `provide`, `report`, and `return`. Programmers rated `return` above all other choices.

In object-oriented programming, it is common to use words such as public and private to control access to objects. Table XI shows the results for questions related to public, private, protected, constant, and static. For the concepts of public and private, a number of words were rated very highly, including the ones most commonly seen in programming languages (e.g., `public` and `private`, respectively). For protected, however, the word choices overall were rated relatively low (around four), except for the choice `semiprotected`. For the concept constant programmers rated the word `constant` as the most intuitive. Non-programmers gave `permanent` a comparable rating. For the last concept, static, we feel the results were generally inconclusive, even for our programmers.

Table XII shows the results for including libraries in a program, calling functions on objects, and the concept of null. First, we asked participants to rate a concept similar to Java's import. While we included the word `include` here, our concept description

Table XII. Word Choice for Other Tested Concepts

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Use	Non-programmer	obtain math_functions (6.23, 2.64), use math_functions (5.99, 2.92), import math_functions (5.85, 2.98)	procure math_functions (4.13, 2.75), purchase math_functions (3.57, 2.70), friend math_functions (3.16, 2.51)
	Programmer	import math_functions (7.00, 2.51), include math_functions (6.77, 2.56), use math_functions (6.56, 2.49)	framework math_functions (4.35, 2.63), friend math_functions (3.81, 2.81), purchase math_functions (2.82, 2.22)
Dot Operator	Non-programmer	Window:close (6.24, 3.08), Window->*close (5.78, 3.17), Window->close (5.63, 3.04)	Window<--close (3.80, 2.80), Window..close (3.78, 2.91), Window@#-close (2.49, 2.75)
	Programmer	Window.close (6.94, 2.90), Window:close (6.44, 2.61), Window->close (6.12, 2.81)	Window--close (3.18, 2.79), Window <--close (2.61, 2.46), Window@#-close (1.61, 1.93)
Null	Non-programmer	undefined (7.12, 3.23), empty (6.37, 2.95), blank (6.29, 2.90)	zero (4.23, 3.53), selected (2.22, 2.58), defined (2.06, 2.70)
	Programmer	undefined (7.57, 2.85), null (7.51, 2.69), empty (7.31, 2.70)	zero (3.30, 3.21), selected (1.90, 2.34), defined (1.74, 2.62)

Table XIII. Word Choice Results for Error Handling

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Try	Non-programmer	check (6.05, 2.63), test (5.84, 2.76), on error (5.35, 2.95)	insure (3.67, 2.53), preserve (3.50, 2.59), protect (3.45, 2.74)
	Programmer	test (5.84, 2.76), check (6.05, 2.63), try (5.34, 2.87)	guard (3.70, 2.81), protect (3.45, 2.74), preserve (3.50, 2.59)
Catch	Non-programmer	error (6.91, 2.98), detect (6.45, 2.73), problem (6.43, 3.10)	except (3.51, 2.53), ordo (3.50, 2.79), main (3.04, 2.74)
	Programmer	error (7.08, 2.54), detect (6.44, 2.86), problem (6.39, 2.88)	except (4.13, 2.75), ordo (3.65, 2.69), main (2.08, 2.11)
Throw	Non-programmer	error (7.11, 2.74), fix (6.46, 2.78), alert (6.43, 2.62)	handler (3.22, 2.47), throw (3.11, 2.65), toggle (3.00, 2.58)
	Programmer	error (7.49, 2.52), alert (7.37, 2.31), fix (6.33, 2.64)	handler (3.76, 2.79), do (3.51, 2.94), toggle (3.38, 2.64)
Finally	Non-programmer	regardless (6.70, 2.92), always (5.95, 3.15), execute (5.85, 2.96)	sometimes (3.21, 2.77), pool (2.74, 2.70), never (2.66, 2.88)
	Programmer	regardless (7.08, 2.73), always (6.33, 3.04), execute (6.21, 2.76)	sometimes (2.60, 2.64), pool (2.21, 2.52), never (1.81, 2.63)

matches closer to how the mechanism in Java works, as opposed to C++. For non-programmers, a number of words were rated at around six, including `obtain` and `use`. Programmers rated `import`, `include`, and `use` well. Second, participants rated symbols that denote calling the function `close` on a class `Window`. Non-programmers rated the syntax `Window:close` as one of few highly rated alternatives, while the dot and the colon fared well with programmers. This somewhat odd result also replicates previous work [Stefik and Gellenbeck 2011]. Lastly, we tested the concept of a null pointer (or reference). The word `undefined` stands out for both groups, although a number of other alternatives were rated comparably.

We also asked participants to rate words related to the concepts of `try`, `catch`, `throw`, and `finally` (Table XIII). For the concept of `try`, both programmers and non-programmers rated the words `check` and `test` well, although overall there was little agreement amongst non-programmers. The words `detect`, `error`, and `problem` appear to stand out among the alternatives for the concept of a catch block, whereas for the concept of throwing an exception, the words that stand out appear to be `error`, `fix`, and `alert`. For the concept of finally, which in some programming languages means that a block is guaranteed to execute, it is interesting to note that the word `finally` was rated poorly by both programmers and non-programmers while the word `regardless` was rated well.

Table XIV. Word Choice Results for I/O

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Input	Non-programmer	input (7.13, 2.84), request (6.80, 2.70), ask (6.13, 3.00)	listen (4.23, 3.30), read (4.20, 2.92), cout (3.44, 3.29)
	Programmer	input (7.35, 2.67), get (6.76, 2.91), request (6.73, 2.63)	take (4.14, 2.63), write (4.07, 2.97), cout (3.19, 3.27)
Output	Non-programmer	display (6.30, 2.79), output (6.15, 2.91), show (5.80, 2.82)	console.write (4.74, 3.10), printf (4.06, 2.75), set (3.71, 2.79)
	Programmer	display (7.33, 2.51), cout (7.11, 3.11), output (7.10, 2.49)	type (3.90, 2.55), read (3.45, 2.92), set (3.17, 2.68)
Say	Non-programmer	vocalize (7.05, 2.57), speak (6.95, 3.20), say (6.90, 2.71)	expose (3.99, 2.90), display (3.79, 2.84), print (3.51, 2.95)
	Programmer	speak (7.50, 2.79), vocalize (7.42, 2.73), say (7.25, 2.36)	print (3.39, 2.99), display (3.20, 2.75), expose (2.87, 2.60)

Table XV. Word Choice Results for Comments. (Word Choices with the Same Average Were Sorted Alphabetically)

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Single Line Comment	Non-programmer	// The next line (5.80 2.79), comment The next line (5.70 2.76), note The next line (5.50 2.80)	secret The next line (3.61, 2.91), REM The next line (3.41, 2.50), quiet The next line (3.37, 2.56)
	Programmer	// The next line (7.77, 2.76), comment The next line (5.00, 3.21), -- The next line (4.93, 2.90)	REM The next line (3.11, 2.86), quiet The next line (2.69, 2.59), secret The next line (2.50, 2.46)
Multi Line Comment	Non-programmer	/* comment text here */ (5.13, 2.91), --- comment text here --- (4.71, 3.05), /** comment text here **/ (4.70, 3.13)	aside comment text here endaside (3.29, 2.73), hidden comment text here hidden (3.29, 3.00), header comment text here footnote (3.27, 2.78), hidden comment text here not hidden (3.26, 2.59)
	Programmer	/* comment text here */ (7.35, 3.05), /** comment text here **/ (6.81, 2.56), <!-- comment text here --> (5.40, 3.19)	conceal comment text here reveal (2.92, 2.60), begin comment text here end (2.87, 2.72), header comment text here footnote (2.35, 2.50)

3.2.5. Input, Output, and Comments. As can be seen from Table XIV, for the concept of input, both programmers and non-programmers rated the word choices `input` and `request` highly. For the concept of output, both groups rated the choices `display`, and `output` well, although there was little consensus overall for non-programmers. As our development team writes a considerable amount of software for the blind and visually impaired community [Stefik et al. 2011a], we were curious what novices and programmers thought an intuitive word choice would be for the concept of outputting text-to-speech. For this concept, both programmers and non-programmers rated `speak`, `vocalize`, and `say` well.

As programming languages often vary substantially in what symbols represent the ideas of single and multiline comments (Table XV), we tested these concepts as well. For making the text `The next line` into a *comment*, non-programmers rated the English words `note` and `comment` highly. Interestingly, however, non-programmers also rated the traditional single line comment used in C++ at approximately the same value (`//`). For *multiline comments*, non-programmers showed no clear preference, while programmers, not surprisingly, rated as intuitive the symbols they were most familiar with (`/* */`).

3.2.6. Aspect-Oriented Programming. While the features common in aspect-oriented programming [Kiczales 1996] are not frequently included in most mainstream programming languages (with notable exceptions), a number of researchers have considered it to be a success (in at least one case, paradoxically [Steimann 2006]), despite recent empirical evidence showing mixed results with its usage in formal studies [Hanenberg

Table XVI. Word Choice Results for Aspect Oriented Programming. (Word Choices with the Same Average Were Sorted Alphabetically)

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Pointcut	Non-programmer	target (6.12, 2.73), location (5.93, 2.78), position (5.39, 2.81)	pointcut (4.01, 2.67), joinpoint (3.83, 2.61), locus (3.73, 3.03)
	Programmer	target (6.33, 2.35), location (6.32, 2.46), position (5.94, 2.70)	joinpoint (3.86, 2.45), pointcut (3.86, 2.66), sample (3.76, 2.59), locus (2.58, 2.53)
Aspect	Non-programmer	situation (5.40, 2.63), location (5.32, 2.76), position (4.87, 2.61)	pointcut (3.88, 2.64), joinpoint (3.76, 2.78), end (3.24, 2.85)
	Programmer	instance (5.38, 2.82), location (5.10, 2.89), target (4.95, 2.80)	aspect (3.95, 2.94), pointcut (3.33, 2.75), end (2.83, 2.72)

Table XVII. Word Choice Results for Method Modifiers

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Before	Non-programmer	advance (5.57, 2.57), initial (5.24 2.57), preempt (5.02 2.95)	earlier (3.71, 2.67), delete (2.88, 3.30), ante (2.77, 2.57)
	Programmer	initial (5.68, 2.99), beginby (5.61, 3.10), before (5.20, 2.87)	independent (2.82, 2.55), ante (2.12, 2.20), delete (1.25, 2.21)
After	Non-programmer	concluding (6.35, 2.59), final (5.82, 2.79), after (5.66, 2.58)	back (3.59, 2.57), static (3.13, 2.68), before (3.02, 2.75)
	Programmer	concluding (6.37, 2.61), after (6.13, 2.49), final (6.06, 2.68)	ultimate (3.14, 2.73), before (2.60, 2.99), static (2.43, 2.51)
Around	Non-programmer	beforeandafter (5.40, 3.11), encircle (5.01, 2.78), firstlast (4.99, 2.71)	before (3.76, 2.68), confine (3.65, 2.80), twice (3.52, 2.59)
	Programmer	beforeandafter (6.05, 2.93), firstlast (5.06, 2.89), encircle (5.02, 2.83)	before (3.05, 2.53), dotwice (3.00, 2.54), twice (2.98, 2.64)

et al. 2009]. Before discussing our results, we should say briefly that aspect-oriented programming is a paradigm for handling so-called "cross-cutting" concerns. Cross-cutting concerns are those that exist across traditional object-oriented boundaries. A common example is logging. For example, if a researcher wanted to log every method on a large system, they have at least two alternatives, namely: (1) write a traditional logger, injecting code into every method to be logged, or (2) write an aspect-oriented logger, where the compiler automatically injects the logging code into a set of methods specified by the programmer.

In terms of our data overall, we hesitate to read into the aspect-oriented results; generally there were minor differences across the board (`target` appeared to do well for describing pointcuts), but we found little terminology that novices rated all that well. We can say, however, that many of the choices made by the aspect-oriented community (e.g., the word `pointcut`) were rated particularly poorly. Whether this was because novices did not understand what we were asking or because they felt no word choice adequately described the concept is unclear.

3.3. Study 2: Surveys on Larger Program Constructs

In Study 1, we investigated the understandability or intuitiveness of common word and symbols choices in programming languages. In our second, we take a slightly different perspective toward answering RQ1, which to remind the reader is: Do novices and programmers subjectively consider all programming language syntax to be equally intuitive (or not intuitive)? In this case, we have gathered syntactical constructs in nine programming languages, including C++, Java, Smalltalk, PHP, Perl, Ruby, Go, Python, and Quorum, again asking non-programmers with no experience and more advanced computer science students (labeled "Programmer" in our tables). We focused our questions on seven common computer science concepts, including loops, strings, if statements, functions, return values, constructors, and inheritance. While our first

```
integer i = 1
repeat 10 times
   i = i + 1
end
```

```
i = 1
while i<=10 do
   i=i + 1
end
```

```
var a[10]int
for i:=range a
{
}
```

```
| i |
i := 1.
10 timesRepeat:
[
    i := i + 1.
].
```

Fig. 1. This is an example of four of the loop examples shown to participants. Reading left to right, the languages tested were Quorum, Ruby, Go, and Smalltalk. The task description for this example is shown in Table XVIII to the right of the heading, "loop."

Table XVIII. Task Descriptions

Task	Task Description
1: Loop	The code in the square boxes is supposed to make any code that may be in the black field execute 10 times. In order for the code in the black field to work, a variable named `i` tracks how often the code has been executed.
2: String	This code is supposed to define a variable named `x` that saves the sentence "Your result is: ". It also should define another variable that stores the numeric value 10. The code should then combine the values of the two variables to form the sentence, "Your result is: 10" and save it to the variable `x`. Some intermediate variables may be used.
3: If	The code in the square boxes is supposed to define a variable called `x` that stores the numeric value 10. It is also supposed to make any code that may be in "Block 1" execute only if `x` is equal to 10. Any code that may be in "Block 2" will be executed whenever `x` smaller than 10. Any code that may be in "Block 3" will be executed otherwise.
4: Function	Suppose we want to specify that there exists a list of instructions that represents a tangible behavior, (e.g., walking at a certain speed). In each example, there is a black field that represents the specific things that must occur when walking, like bending the knees and moving the legs. In this case, each example is supposed to indicate the following behavior: `walk` at a certain `speed`.
5: Return	Suppose we want to specify that there exists a list of instructions that represents a tangible behavior, e.g., to draw a name from a hat. In each example, there is a black field that represents the specific things that must occur when drawing a name, like pulling a piece of paper from the hat and unfolding it. At the end of the list of instructions, the name on the piece of paper needs to be obtained. In this case, each example is supposed to indicate the following behavior: `draw` a name from a hat, which in this case obtains the name `Bill`.
6: Constructor	This code should make a representation of a dog in the computer. The code in the black box represents a list of instructions that is executed when this representation is made. The code should define one representation of a dog, named sam.
7: Inheritance	This code should make a representation of a dog and a mammal in the computer. It should specify that dogs are specific kinds of mammals. The code should specify that mammals have a height and can walk. Similarly, dogs have a species and can bark. It should define a dog named sam, specify its height as 15, its species as Dachshund, and should make sam bark.

study was largely formative descriptive data, in the second we explicitly define the following null hypotheses.

H_{01}. In aggregate, programming languages are rated as equally intuitive.
H_{02}. All programming language constructs are rated as equally intuitive.
H_{03}. Programming experience has no effect on subjective ratings of intuitiveness.

We discuss each of these hypotheses throughout the rest of the narrative.

3.3.1. Methodological Differences with Study 1. A complete list of the syntax we tested would be too long for this article, but like before, a complete replication package with all raw data, questions, surveys, scripts, and other materials are available from the authors on request. To give an idea of the types of constructs we tested, see Figure 1, which gives four looping examples. As in Study 1, for each question, we gave students an English explanation of what the code does and asked raters to mark how intuitively that syntax represented those ideas (see Table XVIII for a complete list). For loop constructs, the description was "The code in the square boxes is supposed to make any code that may be in the black field execute 10 times. In order for the code in the

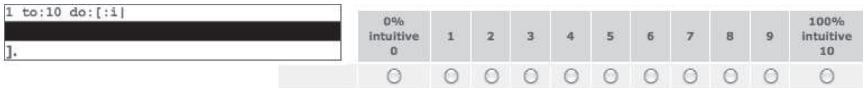


Fig. 2. The online LimeSurvey interface for one of the loop questions.

black field to work, a variable named `i` tracks how often the code has been executed. Please rate each of the following solutions on a scale from 0 (0% intuitive) to 10 (100% intuitive) on how intuitive you find the solution." Note that the descriptions refer to a square box and a black field, shown in Figure 2, but not in Figure 1. This differs slightly from Study 1 in that the previous survey contained only words and symbols as possible responses. Further, as we already described the great care we used in designing our survey, which is very similar with our second experiment, we will not discuss this issue further.

In each case, we took a short sample of code from a given language and asked participants to subjectively rate how intuitively it matches the corresponding concept. We took care to run our examples in a real compiler or interpreter for each language before putting them into the survey and to make each sample do approximately the same thing. For testing the Smalltalk code, the program "VisualWorks" was used. To reduce bias as much as possible, we chose not to give any identifying information about the language in use for each sample and did not indicate that examples all came from a real programming language (although all did). As such, participants who did not already know a particular programming language could not have identified it. In some cases, the language may have included text that could reveal the language to a savvy participant (e.g., statements such as `import java.util.Vector;`). In such cases, we removed words such as "java" and replaced them with the generic phrase "abc." This causes a small minority of the examples to not compile, but prevents potential bias. We pilot-tested this issue carefully on a separate sample, finding that it makes little difference whether the language name is included or not. Finally, the version of Quorum used in the examples was pre-1.0, a version that was never released. Several of the constructs in Quorum were on the drawing board at the time, and we used our data to help us improve the language before moving on to more advanced analysis (in Studies 3 and 4).

It is also important to note that some languages required separate include/import/use statements for some features to work, while others included such features by default. For example, C++ requires `#include` statements in order to use the string class, but Java, and others, do not. We decided to include statements such as these (e.g., include, import, using), because programmers literally have to type them into an editor to make them work.

3.3.2. Results. Before analyzing our data, we first ensured it was approximately normally distributed (e.g., using QQ-plots, histograms) verifying that it did not exhibit unreasonable skew or kurtosis. To test our hypotheses, we analyzed the reported intuitiveness ratings for our experiment in three ways: (1) with respect to the average language intuitiveness across all tasks, (2) the scores of individual language constructs, and (3) in terms of whether the individual was a programmer or non-programmer (see Table XIX). For each task, we tested our hypotheses as a two-factor ANOVA, with partial-eta (η_p^2) values as a variance accounted for measure and Tukey HSD post-hoc tests. Since the results are numerous, we provide only a selection of the ones we found to be the most interesting.

A two-factor ANOVA reveals that languages are rated significantly differently across the board (for all tasks), $F(8, 13428) = 71.071$, $p < 0.001$, $\eta_p^2 = .041$. Similarly, differences between programmers and non-programmers were significant $F(1, 13428) =$

Table XIX. Results for Larger Program Constructs. (Languages with the Same Average Were Sorted Alphabetically)

Task	Group	Top Word/Symbol Choices	Bottom Word/Symbol Choices
Overall	Non-programmer	Quorum (5.53, 2.90), Java (4.99, 2.88), Ruby (4.95, 2.89)	PHP (4.57, 2.80), Smalltalk (4.57, 2.99), Perl (4.09, 2.75)
	Programmer	Java (7.03, 2.77), C++ (6.85, 2.89), Quorum (6.22, 2.85)	Python (5.15, 2.87), Perl (5.09, 2.82), Smalltalk (4.13, 2.68)
1: Loop	Non-programmer	Quorum - repeat times (5.93, 3.02), Smalltalk - timesRepeat (5.40, 2.77), C++/Java - while (5.21, 2.90)	Perl/PHP - for (3.15, 2.93), Go - for (3.07, 2.34), Python - for i in range, specific i (3.01, 2.79)
	Programmer	C++/Java - while (7.81, 2.46), C++/Java - do while (7.57, 2.74), C++/Java - for (7.31, 2.95)	Python - for i in range (4.23, 2.79), Go - for range (3.78, 2.30), Python - for i in range, specific i (2.94, 2.72)
2: String	Non-programmer	Java (5.90, 2.79), Quorum (5.53, 2.95), PHP (5.03, 2.82)	Python (4.51, 2.63), C++ (3.97, 2.80), Go (3.95, 2.45)
	Programmer	Java (6.71, 2.53), Quorum (6.56, 2.46), Python (6.29, 2.71)	PHP (4.61, 2.77), Smalltalk (4.03, 2.36), Go (4.01, 2.55)
3: If	Non-programmer	C++/Java (6.18, 2.83), Quorum (5.85, 2.66), Go (5.58, 2.54)	Perl (4.58, 2.56), PHP (4.58, 2.60), Smalltalk (4.41, 2.65)
	Programmer	C++/Java (8.42, 2.24), Go (6.84, 2.35), Quorum (6.37, 2.70)	Perl (5.67, 2.32), Python (5.52, 2.75), Smalltalk (3.99, 2.28)
4: Function	Non-programmer	Quorum (6.04, 2.92), Smalltalk (5.85, 3.24), Go (5.59, 2.54)	PHP (4.70, 2.60), C++/Java (4.12, 2.77), Perl (3.10, 2.55)
	Programmer	C++/Java (6.70, 2.85), Quorum (6.49, 2.68), Go (6.16, 2.32)	PHP (5.77, 2.59), Smalltalk (4.74, 2.78), Perl (3.66, 2.54)
5: Return	Non-programmer	PHP (5.68, 2.71), Python (5.67, 2.70), Quorum (5.67, 2.99)	Go (5.11, 2.45), Perl (3.79, 2.47), Smalltalk (3.70, 3.09)
	Programmer	PHP (6.94, 2.33), C++ (6.91, 2.75), Java (6.77, 2.61)	Ruby (5.97, 2.59), Perl (4.15, 2.59), Smalltalk (3.03, 2.68)
6: Constructor	Non-programmer	Quorum (4.88, 2.72), PHP (4.77, 2.58), Perl (4.62, 2.62)	Python (4.26, 2.48), Java (4.21, 2.76), Smalltalk (4.18, 3.31)
	Programmer	C++ (5.68, 2.80), Java (5.67, 2.72), Quorum (5.61, 3.18)	Go (4.32, 2.55), Python (3.56, 2.47), Smalltalk (3.33, 3.05)
7: Inheritance	Non-programmer	Ruby (5.88, 2.63), Quorum (5.62, 2.77), PHP (5.60, 2.81)	C++ (5.12, 2.48), Perl (5.05, 2.86), Smalltalk (4.96, 2.82)
	Programmer	C++ (6.52, 2.67), Quorum (6.40, 2.59), Java (6.33, 2.81)	Python (5.09, 2.50), Perl (4.98, 2.55), Smalltalk (3.95, 2.73)

363.565, $p < 0.001$, $\eta_p^2 = .026$, and there was a significant interaction between programmer/non-programmer and language $F(8, 13428) = 28.635$, $p < 0.001$, $\eta_p^2 = .017$. Effects of this size are commonly interpreted as reliable but small. It should be clear to the reader that our first two null hypotheses, H_{01} and H_{02}, should be rejected: some languages, and some constructs in those languages, are subjectively rated by humans as more intuitive than others. Interestingly, we were curious how well our language, Quorum, was rated by non-programmers. Post-hoc Tukey HSD tests reveal that Quorum was rated as statistically significantly more intuitive than Go ($p < 0.001$), C++ ($p < 0.001$), Perl ($p < 0.001$), Python ($p < 0.001$), Ruby ($p < 0.024$), Smalltalk ($p < 0.001$), PHP ($p < 0.001$), and approached significance with Java ($p = .055$). The result holds generally for programmers as well, except that there was no statistical difference between C++, Java, and Quorum, for these users (an example of the interaction effect). We summarize the remainder of our results in Table XX, giving the language, programmer, and interaction results. Obviously, this does not mean that Quorum *is* more intuitive than these languages, but it does mean that novices in our sample certainly *perceived* it to be.

Finally, our results also show that our third null hypothesis (H_{03}) should be rejected. We found that for approximately every year of self-reported experience in C++, users rated C++ examples higher, resulting in the following linear regression equation: $y = 5.37838 + 0.56818x$. In this case, y is the rating for a particular C++ question and

Table XX. A Summary Table Showing the Statistical Results. Overall, Results Indicate That Effects Were Relatively Small ($\eta_p^2 < .1$ in Many Cases), but Were Also Real and Easily Detectable (Literally All Effects Were Significant)

	Language				**Experience**				**Interaction**			
	F	df	p	η_p^2	F	df	p	η_p^2	F	df	p	η_p^2
Aggregate	71.071	8	< 0.001	0.041	363.565	1	< 0.001	0.026	28.635	8	< 0.001	0.017
1: Loop	18.444	26	< 0.001	0.098	347.013	1	< 0.001	0.073	8.083	26	< 0.001	0.045
2: String	15.419	8	< 0.001	0.077	9.366	1	$= 0.002$	0.006	3.420	8	< 0.001	0.018
3: If-else	27.727	8	< 0.001	0.131	65.668	1	< 0.001	0.043	4.306	8	< 0.001	0.023
4: Function	15.107	8	< 0.001	0.076	35.320	1	< 0.001	0.023	7.302	8	< 0.001	0.038
5: Return	27.788	8	< 0.001	0.131	25.531	1	< 0.001	0.017	2.649	8	$= 0.007$	0.014
6: Constructor	6.993	8	< 0.001	0.037	4.126	1	$= 0.042$	0.003	3.583	8	< 0.001	0.019
7: Inheritance	7.445	8	< 0.001	0.039	5.106	1	$= 0.024$	0.003	2.980	8	$= 0.003$	0.016

x is the number of years of self-reported C++ experience. In short, users rated C++ syntax approximately half a point higher for every year of self-reported experience. An omnibus F-test (which we remind the reader is just an ANOVA), reveals that this result is statistically significant, $F(1, 1492) = 143.1$, $p < 0.001, multiple - R^2 = 0.08754$. The adjusted R^2 was 0.08693, close to the $multiple - R^2$ already reported, indicating that these results would have approximately the same effect size at the population level. Our sample generally only had experience with C++, so we did not compare this result with other languages.

3.4. Discussion of Studies 1 and 2

We have documented a corpus of data on human ratings of words, syntax, and larger constructs in programming languages, with the goal of helping instructors and language designers understand the following broad point: novices find many of the choices made by language designers to be unintuitive. While we imagine many computer science instructors already have a "gut feeling" that this is the case, our results help to formalize this notion, in addition to cataloging a wide swath of possible choices. To sum up the results more specifically, our two studies provide evidence for the following claims: (1) while humans vary substantially in their opinions, not all programming languages, nor their corresponding constructs, are rated as equally intuitive, (2) some very common programming language syntax (e.g., for loops), is considered by non-programmers (and sometimes by programmers) to be unintuitive, and (3) as programmers gain experience, they rate familiar syntax higher—by an average of about half a point per year on an 11-point scale.

We recognize that some will claim our survey is a trivial exercise—perhaps believing that human opinions do not matter in language design. While we find this line of reasoning to be rather naive, given our observations while teaching novices in the classroom, we have created a new technique for Experiments 3 and 4, called a Token Accuracy Map (TAM), that provides an estimate of novice accuracy on a per-token basis. While we have not tested all of the words in our sample, words that novices report make sense (e.g., repeat) do seem to plausibly benefit novices when first learning to program. Even if this were not the case, it would not impact our story greatly; from our view, it seems perfectly reasonable to ask students what they think about components of programming languages, so long as we recognize that surveys provide us clues, not proof. The programming language design space is truly daunting and surveys can give us investigative clues regarding many alternatives quickly.

We would, however, like to point out several threats to validity in our first two experiments. First, our novice ratings are based on the assumption that the questions are sensible descriptions of the concepts we are trying to describe and that novices

understood what we were asking. Designing questions such that all readers will agree they were phrased well is difficult, but we have attempted to make our questions clear, neutral, and reasonable. We have already described our process, but we should state again that all surveys suffer from the possibility of accidental or intentional bias. A variety of textbooks devote far more space to these types of issues than we can here (see e.g., Vogt [2006]). Further, it seems plausible, given the lack of consensus in more advanced questions (e.g., exceptions, aspect-oriented programming) that novices, despite our best efforts, may not have grasped our meaning. Detecting whether we phrased the question poorly, whether the concept was understood, or whether novices understood but did not have a consensus view, is not easily determined from our data. Like any survey, varied replications by independent research groups are the easiest way to ensure the correctness and neutrality of the results.

Second, while we have based our surveys around the concept of intuitiveness, we have not defined the term formally. We made this decision intentionally, as our studies were designed in part to help our team garner a more formal understanding of what intuitiveness might mean in the context of programming. While we are hesitant to offer a formal definition, we do think intuitiveness in a programming language may have recognizable properties. For example, individuals do appear to (1) rate as intuitive common words with a well known meaning in English (e.g., repeat, undefined), (2) rate as less intuitive metaphorical names (e.g., throw, catch), and (3) rate as less intuitive symbols where the meaning in programming conflicts with the meaning in English (e.g., the dot operator). However, while these properties seem reasonable given our results, we still struggle with finding a definition of intuitiveness that is completely satisfying. Importantly, intuitiveness is likely dependent on context. In our case, the context was surveys in English of college students attending a university in approximately the middle of the United States. Exploring this issue in other cultures, and other languages, may help give our a community a better understanding of not only what intuitiveness really means, but also how to manipulate it to improve programming languages.

Third, in addition to the questions we asked novices, we suspect that some readers may believe that, given that we are the inventors of the Quorum programming language, that this influenced our results through a form of accidental bias. While accidental bias is difficult to avoid, we should mention that when our surveys were first being piloted, Quorum had not yet been invented. Succinctly, this potential threat reverses causality—we used our surveys to help us investigate what the design of a language could be, not the other way around. For Study 2, some parts of Quorum were still hypothetical, but others had been either designed or implemented. Put another way, we wanted to subject the design of a new and emerging language to peer review from novice students in an effort to make it easier to understand. Of course, surveys do not provide a complete picture, as will be obvious after experiment 4. However, many of our findings here do fit well with common sense. For example, it certainly seems plausible that the highest rated loop in Study 2, Quorum's `repeat 10 times` might be easier to use or understand than the traditional `for(int i = 0; i < 10; i++)`. We now move on to testing assumptions like these more formally.

4. STUDIES 3 AND 4: NOVICE ACCURACY RATES

The evidence presented in Studies 1 and 2 provides data on how humans subjectively rate the intuitiveness of various programming constructs. Our goal with these surveys was to give us clues into the kinds of constructs novices might find easy to understand; techniques which were extremely useful in the formative development of Quorum. In this section, however, we had novices actually try to program, investigating their syntactic accuracy while using a variety of programming languages. This section includes

both an expansion of a small-scale pilot study published previously [Stefik et al. 2011c] and a replication that includes more programming languages on a larger sample. After describing these studies, we give several samples, on a token-by-token level, of the accuracy rates of each language, a presentation that we we call Token Accuracy Maps. Broadly, we are trying to answer the following research question.

*RQ*2. Can novices using programming languages for the first time write simple computer programs more accurately using alternative programming languages?

We think that empirical clues on this research question might provide insight for students using these languages and instructors that ultimately have to teach them by identifying initial syntactic barriers.

Given this broad question, we test here the following null hypotheses,

H_{04}. Novices will have equal accuracy rates while programming, regardless of the programming language used.

H_{05}. All syntactical variations of programming language constructs (e.g., loops, conditionals) will afford equal accuracy rates amongst novices.

The studies were conducted as a repeated-measures between-subjects design with six tasks using an accuracy measurement technique called Artifact Encoding [Stefik et al. 2011a], originally designed for analyzing a talking debugger for blind computer programmers. To grade each experimental task, we used a simplified version of Artifact Encoding [Stefik et al. 2011c].

While previous work discusses the procedure in detail, Artifact Encoding produces a key that can be graded by a computer. These answer keys are metaphorically similar to how an answer key for a class would be constructed; break-down the computer code into components (e.g., did the user define a particular variable correctly?) and score each with a code. In this study, if an answer was correct, we marked that component with a 1. If a particular component was incorrect, we marked it with a 0. Once all components were marked, a total and a percentage was computed for each task. So, in effect, we computed a "percent correct" metric for each task and used these values in our statistical models, but we did so in such a way that we could compute an inter-rater reliability analysis. Previously published forms of artifact encoding designed for testing auditory debuggers for the blind are far more complex [Stefik et al. 2011a].

Participants completed all six tasks using one of the programming languages described in Table XXI. As we will demonstrate, all of the programming languages tested here, including our own (Quorum), have syntactic issues that language designers can choose whether or not to fix. Further, users programming in two of the programming languages (Perl and Java) had accuracy rates so low that we could not statistically differentiate their scores from those using our metaphorical placebo—a language where the keywords were generated randomly from the ASCII table (Randomo). We now briefly describe this idea, known historically in the bio-medical sciences as using a "dummy treatment."

4.1. Dummy Treatments: A Short History

Before we begin discussing our methodology for the last two experiments in our article, we feel our use of Randomo requires some historical context. Kaptchuck describes the development of randomized controlled trials in the bio-medical sciences [Kaptchuk 1998][1]. For example, the earliest known study using a dummy treatment

[1]Technically, Kaptchuk uses the word "sham treatment", although we prefer the term "dummy treatment" used by the National Health Service [House of Commons and Committee 2010]. Kaptchuk's term "sham" was used to describe the deception used in experiments done by Franklin and others testing medical theories

Table XXI. Summary Table of the Programming Languages Included in Each Experiment

Language	Studies	Sample Size	Description
Quorum	3,4	6, 12	A programming language designed using data gathered in Studies 1 and 2. Full documentation of the syntax and semantics is available online at http://quorumlanguage.com/.
Perl	3,4	6, 13	A well-known programming language originally designed by Larry Wall.
Randomo	3,4	6, 12	A programming language based largely on the syntactical structure of Quorum. With the exception of braces, the lexical rule for variable names, and a few operators (e.g., addition, subtraction, multiplication, division), many of the keywords and symbols were chosen randomly from the ASCII table.
Java	4	12	A well-known programming language originally designed by James Gosling at Sun Microsystems.
Ruby	4	11	A well-known programming language originally designed by Yukihiro Matsumoto.
Python	4	12	A well-known programming language originally designed by Guido van Rossum.

was led by Benjamin Franklin (1706-90) as part of a commission to examine the animal magnetism theory espoused by Franz Anton Mesmer (1734–1815). In these initial experiments, participants were blindfolded so they could not detect where the "energy" Mesmer claimed to exist was directed. As is now well known, while these and many other experiments ultimately showed Mesmer's theory to be false, his ideas remained tremendously popular throughout the nineteenth century. Similarly, such dummy treatments were eventually used to discredit drugs, most famously Samuel Hahnemann (1755–1843)'s homeopathic "remedies," which are oddly still practiced by the U.K.'s National Health Service [House of Commons and Committee 2010].

In computer science, we feel the situation in programming language design has unfortunate similarities to that of the medical sciences in the late 18th century. On the one hand, general purpose language designers have a standard of evidence for the efficiency of algorithms, and correctness, that is far beyond what was possible in the 18th century (e.g., asymptotic analysis). On the other, when we examine the syntax and semantics of programming languages, we are highly skeptical that the choices make much sense to humans, especially given our results from Studies 1 and 2. Garnering a better understanding of how humans use programming languages may require a number of research techniques to fully comprehend (e.g., randomized controlled trials with placebo, field studies, classroom interventions, usability studies), but given the widely acknowledged success of randomized controlled trials in other disciplines, it makes sense to consider the impact of this level of rigor.

Given our observations, we conducted a thought experiment: what if we took the keyword choices we ultimately found for Quorum (from Studies 1 and 2) and replaced them with random symbols from the ASCII table? If tests with novices revealed no statistically detectable differences when programming, when compared to a language with randomly chosen keywords, then we should conclude that syntax variations amongst languages may not matter. If we do, however, then such a language might provide a baseline by which we can compare results from many languages. As such, we designed Randomo by replacing keywords from Quorum with random characters, while keeping the structure similar to allow for a comparison (e.g., `if` became `:`). A few symbols were not randomly selected (e.g., addition, subtraction), because we assumed that language designers with a reasonable grasp on reality would not adjust these choices.

Ultimately, our choice is a first attempt. We tried to thread the needle in finding an appropriate choice of Placebo as best as we could. To our knowledge, however, our

in the 18th and 19th centuries [Kaptchuk 1998]. As language designers are generally not trying to deceive, dummy treatment or placebo seems like more reasonable terminology.

experiment is the first to even attempt to use dummy treatments as part of language design, so it may take time for our discipline to find the right balance in such treatments. Our broad point here is that we encourage other education researchers to adopt their own form of dummy treatments and to tackle the programming language problem with the mindset of the medical sciences—randomized controlled trials with placebo. This type of approach may give us a new way forward when investigating the long-standing, and vexing, programming language design problem.

4.2. Methodology

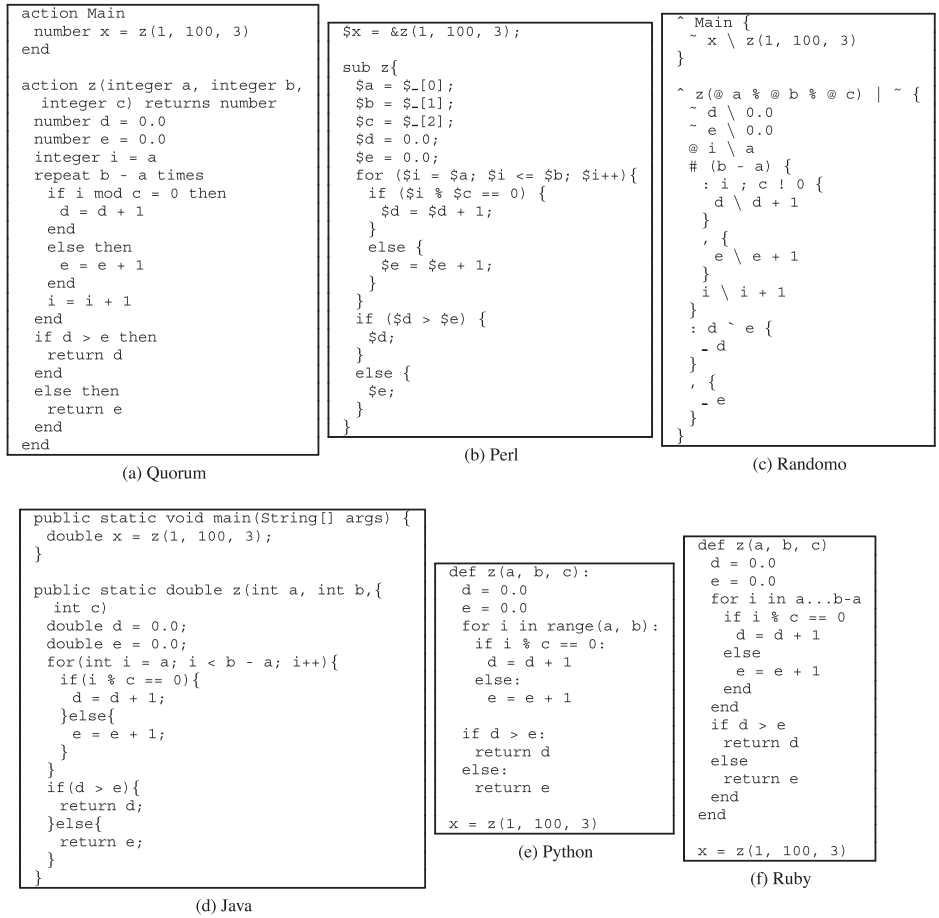
As in the first part of our article, Studies 3 and 4 hold significant similarities. As such, this section is organized as follows. First, we will describe the population we drew participants from for both studies. Next, we will discuss our materials and tasks broadly, then move to results for each experiment. Finally, we will present a discussion of both studies and then a broad discussion overall, including threats to validity.

4.2.1. Participants. For our third study, we solicited 19 participants between the months of April and July of 2011 from non-computer programming classes at Southern Illinois University Edwardsville after appropriate Institutional Review Board ethics reviews. Participants in all groups were equally paid $10 for their participation. Of these individuals, one participant left in the middle of the study. Since this participant did not complete the experiment, that individual's data was subsequently removed. Of the remaining 18 participants, the average age was 21.3 years, with 12 males and 6 females. All reported being native English speakers. The programming languages tested for studies three and four are summarized in Table XXI. In the case of Study 3, six participants were in each group.

For our fourth study, we solicited 73 new participants between the months of September 2011 and June 2012, again from non-computer programming classes at Southern Illinois University Edwardsville. In this case, participants were recruited from a participant pool and were unpaid. Due to a mistake, we accidentally placed 13 individuals into the Perl group and 11 in the Ruby group. All other groups had 12 participants, which adds to 72. The final, 73rd, participant was accidentally given the solution to task 3 instead of the task description, so we did not include that individual's data in our analysis. Three individuals reported being non-native English speakers and one declined to respond to this survey question. The average age for experiment four was 20.1 years, with 48 males and 24 females.

In both experiments, we checked with participants to ensure that they had never programmed a computer. In exit surveys, however, we noticed a handful of students made unusual or confusing markings. For example, one student marked that they had both never programmed and that they were currently taking our senior projects computer science sequence (besides being impossible, we confirmed that this was incorrect). We could have removed these students from the sample, but instead conferred with all students individually and verbally asked each whether they had ever programmed. All confirmed they had not. We further guarded against the effect of experience by randomly assigning all participants to the experimental groups in both experiments.

4.2.2. Procedure and Experimental Walkthrough. When participants began the study, they were first greeted by a proctor and, again, assigned randomly to a group. For those readers unfamiliar with formal controlled experiments, Vogt [2006] describes why issues such as random assignment are important in significant, and easily readable, detail. Participants were then seated at a computer with Windows 7 installed. Cardboard barriers were placed between each individual. These barriers prevented people from looking at other participants' screens or otherwise cheating. For a participant to cheat,



```
action Main
 number x = z(1, 100, 3)
end

action z(integer a, integer b,
  integer c) returns number
 number d = 0.0
 number e = 0.0
 integer i = a
 repeat b - a times
  if i mod c = 0 then
   d = d + 1
  end
  else then
   e = e + 1
  end
  i = i + 1
 end
 if d > e then
  return d
 end
 else then
  return e
 end
end
```

(a) Quorum

```
$x = &z(1, 100, 3);

sub z{
 $a = $_[0];
 $b = $_[1];
 $c = $_[2];
 $d = 0.0;
 $e = 0.0;
 for ($i = $a; $i <= $b; $i++){
  if ($i % $c == 0) {
   $d = $d + 1;
  }
  else {
   $e = $e + 1;
  }
 }
 if ($d > $e) {
  $d;
 }
 else {
  $e;
 }
}
```

(b) Perl

```
^ Main {
 ~ x \ z(1, 100, 3)
}

^ z(@ a % @ b % @ c) | ~ {
 ~ d \ 0.0
 ~ e \ 0.0
 @ i \ a
 # (b - a) {
  : i ; c ! 0 {
   d \ d + 1
  }
  , {
   e \ e + 1
  }
  i \ i + 1
 }
 : d ` e {
  _ d
 }
 , {
  _ e
 }
}
```

(c) Randomo

```
public static void main(String[] args) {
 double x = z(1, 100, 3);
}

public static double z(int a, int b,{
  int c)
 double d = 0.0;
 double e = 0.0;
 for(int i = a; i < b - a; i++){
  if(i % c == 0){
   d = d + 1;
  }else{
   e = e + 1;
  }
 }
 if(d > e){
  return d;
 }else{
  return e;
 }
}
```

(d) Java

```
def z(a, b, c):
 d = 0.0
 e = 0.0
 for i in range(a, b):
  if i % c == 0:
   d = d + 1
  else:
   e = e + 1

 if d > e:
  return d
 else:
  return e

x = z(1, 100, 3)
```

(e) Python

```
def z(a, b, c)
 d = 0.0
 e = 0.0
 for i in a...b-a
  if i % c == 0
   d = d + 1
  else
   e = e + 1
  end
 end
 if d > e
  return d
 else
  return e
 end
end

x = z(1, 100, 3)
```

(f) Ruby

Fig. 3. This code shows one of the code samples provided to participants. The description said the following: This code will count the number of values that are and are not divisible by c and lie between a and b. It then compares the number of values that are and are not divisible by c and makes the greater of them available to the user.

they would have had to physically stand up and walk to a neighbor's computer to do so. As we monitored participants and made reasonable restrictions to prevent cheating, we conclude that each participant worked independently and without assistance from their peers or the Internet.

Each experimental session lasted approximately two hours and followed a standard checklist of procedures. A complete replication package, including all tasks, procedure checklists, task solutions, and proctor scripts is available from the authors on request. Before beginning, participants were read a script describing what they will be doing in the experiment. Once complete, participants were given a code sample worksheet for the particular language group they were in (again as summarized in Table XXI). See Figure 3 for one of the examples on the reference sheet given to participants.

The general idea of our controlled experiment is to give novice users code samples similarly to if a participant was learning to program from home on their own. As such, we intentionally did not teach participants what each line of syntax did in the computer programs. Instead, participants had a standardized set of examples that were

Table XXII.

A table giving the programming concepts highlighted in each task. The amount of time available to complete a task and whether a reference sheet (during) and solution (after) were available for inspection is listed.

Task	Study 3 Times	Study 4 Times	Reference	Concepts Tested
Task 1	7 minutes	6 minutes	Yes	conditional statements, strings, and variables
Task 2	7 minutes	6 minutes	Yes	loops, variables
Task 3	7 minutes	6 minutes	Yes	functions, parameters, return values
Task 4	10 minutes	6 minutes	No	loops, variables
Task 5	10 minutes	6 minutes	No	functions, parameters
Task 6	10 minutes	6 minutes	No	nested conditional statements, variables

identical across groups (with appropriate syntax), which they used to derive the meaning of the computer code on their own. To be clear, we are not arguing that this type of study is exactly like learning to program at home, but we think this serves as a reasonable metaphor. With that said, we think that giving novices examples with some written explanation, and having them derive new solutions, is relatively similar to what students actually do when learning to program. For instance, many U.S. high schools do not even offer computer science courses, so independent study like this (e.g., look at examples and try to write your own), is sometimes the only option students have.

Participants completed a total of six experimental tasks (see Table XXII). In the first three, participants were allowed to reference and use the code samples shown in Figure 3. Seven minutes were allotted to complete each of the first three tasks. Once time was up, participants were given an answer key. This somewhat mimics the idea of finding a working example and then creating a new one on your own. To be clear, these are not typography tasks. Students are not copying code; they are using examples to write new code to a specification that we give them. For the final three tasks, use of the code samples was not allowed, no solutions were given, and ten minutes was allotted for each task. In experiment 3, however, we observed that participants generally did not need the full seven or ten minutes to complete the tasks. As such, in experiment 4, we allotted only six minutes for each task.

4.2.3. Materials and Tasks. For each experimental task, participants were given identical English descriptions of the code they were asked to write (Quorum, Perl, Randomo, Java, Python, or Ruby). For task 1, the description read as follows.

> "Using the code sample given to you, try to write code that defines a variable x that stores real values and is set to 175.3. The code should also define a variable y that stores a string of characters and saves the word false in it. The code should then check whether x is larger than 100. If so, y should save the word true. Otherwise, y should save the words still false. Write your code in the text editor open on the PC in front of you."

The concepts participants had to attempt to program are summarized in Table XXII, all of which focused on the types of core concepts students might see in an introductory course. In other words, we tested features such as loops, conditionals, and function usage, but not more esoteric features such as regular expressions or the use of closures.

We also want to mention that there was a typo in experiment 4's code sample materials. Specifically, we inadvertently left an additional phrase for `i = i + 1` in the code sample document for the language Ruby, inside of the loop. Figure 3 is shown without this typo. This typo did not exist in the answer keys, and as such, we think users in the Ruby group were not at a significant disadvantage. However, to further counteract for this mistake, we did not deduct points from an individual if this line was added to a participant's answer. As such, while the typo may have unintentionally impacted

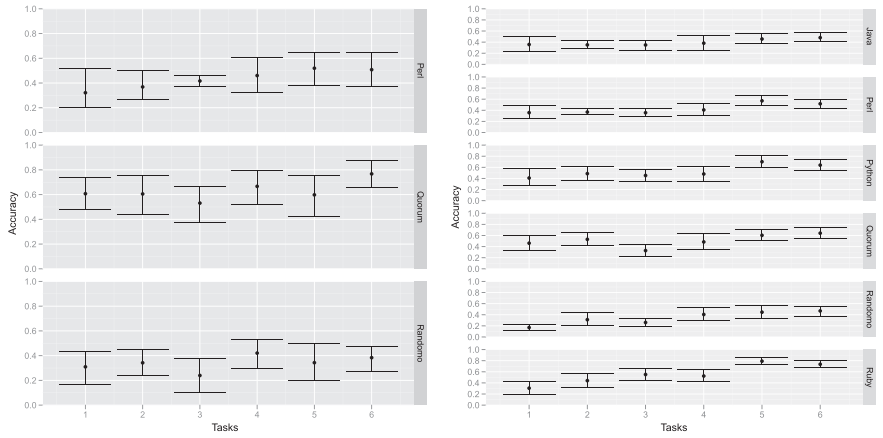


Fig. 4. A summary of the accuracy scores by language and task for experiments 3 (left) and 4 (right). 1.0 means 100% correct, whereas 0.0 means 0% correct.

Ruby users, we consider it unlikely to have made an impact in either the metrics we report or our statistical conclusions.

4.3. Study 3 Results

To ensure our grading could be replicated by other researchers, we first verified that independent raters of the data would give approximately the same result. We did this using a standard inter-rater reliability test called a Kappa analysis (see e.g., Hubert [1977]). Two researchers first trained on data not used in the study and then independently coded approximately 20% of the actual data. A Kappa statistic of 0.80 (raw agreement 91.0%) was found, a result which is typically interpreted as highly reliable. Researchers trained in the grading technique proceeded to code the remaining 80% of the data.

We then proceeded to analyze the data using a standard repeated-measures ANOVA test, with corresponding post-hoc Tukey tests and partial-eta squared values using the statistical package SPSS. For those unfamiliar with this procedure, one must first verify that the assumptions of the statistical test are not violated. To do so, we ran Mauchly's test for sphericity, $\chi^2(14) = 12.071, p = .608$. As the result was non-significant, it implies that the sphericity assumption has not been violated. As such, the standard Greenhouse-Geisser correction was unnecessary for our data.

Next, we conducted a test for within-subjects effects to see if learning played a role in our experiment. Results show that total cross-language averages for task 1 (M = .412, SD = .237) raised slightly by the end of the experiment (M = .552, SD = .224), $F(5, 75) = 3.22, p = .011, \eta_p^2 = .177$). This is not surprising. We would expect participants to improve slightly as they become familiar with either the language or protocol, although it is interesting that scores continued to rise despite the lack of a reference sheet in tasks 4–6. The learning effect interaction with language was non-significant, $F(10, 75) = .735, p = .689, \eta_p^2 = .089$, implying there was no obvious interaction between task order and language. A summary of the accuracy data for each task is shown in Figure 4.

However, the test most critical to our hypothesis is the test of between-subjects effects: did differences in the languages themselves matter? This test shows that differences between the programming languages were both significant and very large, $F(2, 15) = 7.759, p = .005, \eta_p^2 = .508$. Further, post-hoc Tukey HSD tests of

the between-subjects effect indicated a surprising result. While users of Quorum (Average M = .628, SD = .198) were able to program statistically significantly more accurately than users of Perl (Average M = .432, SD = .179), $p = .047$, and users of Randomo (Average M = .341, SD = .173), $p = .004$, Perl users were *not* able to program significantly more accurately than Randomo users, $p = .458$. We found this surprising given that Perl has a relatively C-like syntax, a style that has been copied for decades and is used worldwide in many popular languages. Before running our study, we had assumed that, at least for simple constructs such as loops and conditionals, that any language with C-style syntax would at least afford accuracy rates higher than a placebo. However, as this and the next experiment shows, this was not the case.

4.4. Study 4 Results

In Study four, we conducted exactly the same analysis we did previously, but on a larger sample with six programming languages. First, we again considered the issue of inter-rater reliability, this time with a Kappa score of 0.84 (raw agreement 92.0%). As before, this value indicates that independent raters have high agreement in coding the results. We think it is plausible that Kappa scores went up slightly between experiments 3 and 4 due to minor refinements in our grading protocol.

Unlike the pilot data, results from experiment 4 showed a significant Mauchly's test for sphericity, $\chi^2(14) = 31.454, p = .005$. To account for this, we used the Greenhouse-Geisser correction when reporting our ANOVA results. Given this correction, we conducted a within-subjects test on the main effect of tasks and the task-by-language interaction. Like before, the within-subjects effect was significant, $F(4.20, 276.96) = 29.371, p < .001, \eta_p^2 = .308$), now with a higher partial-eta compared to the pilot (from .177 to .308). Interestingly, the task-by-language interaction was also significant, $F(20.98, 276.96) = 1.64, p = .041, \eta_p^2 = .110$). This latter result might seem contradictory, but the partial-eta scores for both effects are relatively similar, implying the observations were similar in both cases. We think a reasonable interpretation of this result is that all languages experienced some change in scores from beginning to end, as we would expect, and that programming language syntax may have had a small impact on the observed increase in scores over the tasks.

Finally, we again tested the between-subjects effect. In this case, we confirmed that the overall result does replicate on a larger sample—initial accuracy rates of novices did, in fact, vary by programming language, $F(5, 66) = 4.889, p = .001, \eta_p^2 = .270$. In this case, the reader might consider the lower partial-eta to indicate a lack of replication, but this is incorrect. The partial-eta was lower because several of the newly added languages approximately matched each other in performance. In other words, this is what we would expect given additional languages with similar accuracy rates.

In terms of our hypotheses, the between-subjects effects indicates we should reject H_4. In other words, our result here shows clear evidence that syntax does influence initial novice accuracy, a claim we can make with high statistical confidence. However, we want to point out one aspect of our larger experiment that only partially replicated. To do so, we present the overall scores for each language in Figure 4. Notice that the results for the Randomo programming language (our metaphorical placebo) and Perl match very closely to the previous work (near-exact replication), but that Quorum users scored lower overall compared to the pilot (a partial-non-replication). We confirm this observation using post-hoc Tukey tests, which reveal that languages essentially fall into one of two camps: (1) some languages clearly had higher scores than placebo (Quorum (p = .033), Python (p = .011), and Ruby (p = .003)), and (2) for others, there was insufficient evidence to conclude that novices using those languages performed

Table XXIII.

A table showing the cross-task average and standard deviation for each language and the Tukey HSD comparisons. Statistically significant differences are in bold.

			Tukey HSD				
	Mean	**Std. Dev.**	**Python**	**Quorum**	**Perl**	**Java**	**Randomo**
Ruby	.558	.243	0.994	0.940	0.165	**0.044**	**0.003**
Python	.528	.246		0.999	0.412	0.141	**0.011**
Quorum	.508	.232			0.655	0.292	**0.033**
Perl	.429	.189				0.987	0.573
Java	.396	.194					0.922
Randomo	.344	.207					

more accurately than placebo (Java (p = .922), Perl (p = .573)). Additionally, with one exception, no other languages showed significant differences between the others. The exception is that Ruby users had significantly higher accuracy rates than users in Java (p = .044). A table with all Tukey HSD results, mean, and standard deviations overall, is shown in Table XXIII. Our analysis of H_5 requires some context and will be discussed in more detail in the discussion.

4.5. Discussion

We have presented data regarding programming tasks completed by novice computer programmers. We think that a number of points stand out as important with these studies, the most critical of which are as follows: (1) syntax does matter to novices and accuracy rates vary by language, and (2) careful observational data can help us find clues as to which tokens can be altered, modified, added, or removed to help novices use or understand programming languages. An alternative view of point number 2 is that by analyzing languages on a token-by-token basis, instructors may garner clues as to which aspects of a programming language might initially be related to student mistakes.

Experiments 3 and 4 show clear evidence that syntax design is not as trivial as some believe. Both our surveys and accuracy data lead us to the same conclusion—some languages are perceived as easier to understand and some really are easier for novices to use. Interestingly, these results strongly support existing work in the literature, especially by Denny et al., using a completely different research methodology [Denny et al. 2011]. Further, our results also provide support for Denny et al.'s broad conclusion that not all syntax errors are created equal [Denny et al. 2012], a conclusion that resembles our H_{05} (All syntactical variations of programming language constructs (e.g., loops, conditionals) will afford equal accuracy rates amongst novices), which we will now discuss.

To get a better handle on whether we should reject H_{05}, Figure 5 presents the token accuracy map for task 6 with novices using Quorum. On the left of this figure is a set of syntactic choices in Quorum 1.0 (in shaded boxes), followed by two numbers. The number on the left for each shaded box indicates the fraction of participants that correctly placed that token in experiment 3. On the right is the same value for experiment 4. As can be observed, different tokens have wildly different accuracy rates, an observation that holds for all of the languages—H_{05} should very obviously be rejected. For those who would like to see a statistically rigorous test here, the reader should keep in mind that we would only need show two tokens that have different rates. In other words, even a cursory glance of the Token Accuracy Map confirms that this is true. Much more interesting is which tokens were helpful or harmful for novices.

As one of our goals in running these experiments was to find problems in version 1.0 of the Quorum programming language, we were not disappointed. We found two

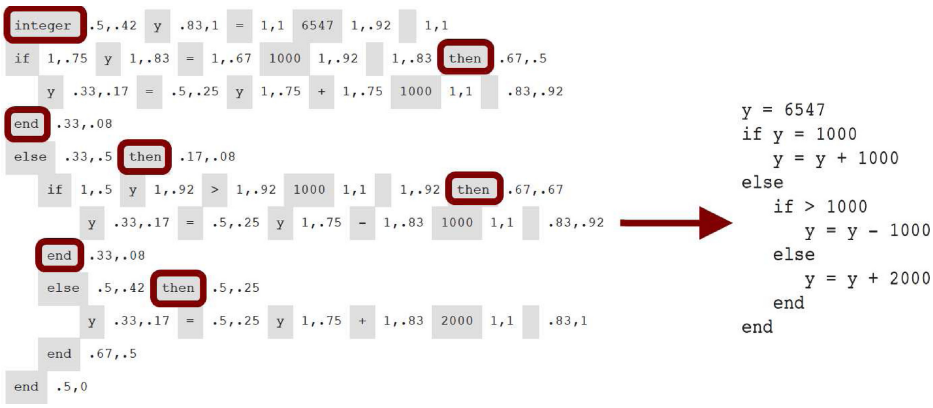


Fig. 5. A summary of the Token Accuracy Map (TAM) for Quorum 1.0, Task 6, with trouble spots in the language circled. The right-hand side shows the modified syntax for Quorum 1.7. For each number next to a highlighted token (e.g., `then`, `integer`), is two numbers. The left number is the proportion of individuals that correctly placed that token in the example in experiment 3, whereas the right number shows experiment 4.

areas of Quorum syntax that our study indicated needed improvement: the (1) design of conditional statements, and (2) the type system. For the first, upon investigating conditional statements in the various languages, it appeared that Ruby was the most successful, we suspect because the syntax was rather terse. In Quorum, we found especially poor accuracy rates for the words `end` and `then`. More explicitly, in experiment 4, only one participant correctly placed the inner `end` tokens. We were surprised by this result, as many academics describe conditionals as "if then" statements. However, the TAM shows only up to 67% accuracy in the best case, with 8% in the `then` after the first `else` for experiment 4 (see Figure 5).

Another result with conditionals we did not expect is the accuracy rate of the double equals sign in conditionals. Our survey data suggested a single equal sign might be more intuitive, so we tried this in Quorum 1.0. While we did this, some on our team (many more than the two authors of this article) considered this choice to be risky. On the one hand, some thought that this would cause issues with novice understanding of assignment statements, while others thought it would help across the board. Our results offer a rather firm conclusion to this question—the single equals sign is a superior choice for novices, a claim we can test by comparing == to our dummy operator in Randomo, the Bang (!). Results from Task 6 (the only task where this was tested), showed a score of 0.67 for = in Quorum (67% of novices used this symbol correctly). When compared to ==, the scores were 0 (Perl), 0 (Java), 0.08 (Python), 0 (Ruby) and 0.08 (Randomo). In other words, the character we randomly chose (!) to represent equality was used correctly by one participant, while the == was used correctly by only 1 out of 48 students in the other four languages. Moreover, 100% of novices used the assignment operator correctly in task 6 for Quorum, where the overloading could have been a problem, but ultimately was not. We understand all too well why many language designers historically made the choice of ==, but the impact on novices is clear.

Our results may also have implications for instructors and students in regards to the use of static type systems, because of the observed difficulty novices have with static type annotations. Some context, however, is critical here, as the usability of type systems has been studied carefully in the literature as of late. For example, the recent literature has largely answered the question of whether static or dynamic typing is beneficial to experienced users. As is now known, under reasonable experimental

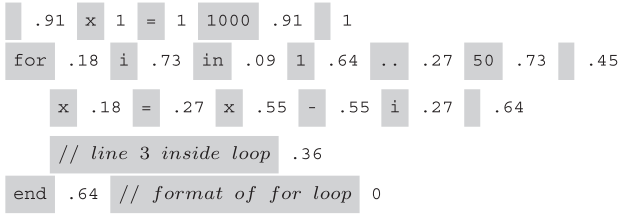


Fig. 6. Token Accuracy Map for Ruby, Task 4. Notice that even with more modern languages like Ruby, novices have significant trouble with `for` loop syntax. In this case, novices consistently missed the words `for`, `in`, and the `..` operator. This follows trends in Quorum, where simply adding extra words to make the syntax more English-like (e.g., in, then) does not necessarily benefit novices.

conditions, static typing helps programmers [Hanenberg 2010a; Kleinschmager et al. 2012; Mayer et al. 2012]. While this result appears to be the growing scientific consensus for more experienced users, novices only placed type annotations approximately half the time in our study, across tasks and languages. From a language design perspective, computer code inside of a method can probably remove type annotations under many conditions while still maintaining static typing. However, for method declarations, there may be an unresolvable trade-off between static and dynamically typed languages. The tradeoff is that novices using type annotations may have slightly greater difficulty. Given time, the available evidence tells us that overcoming this initial barrier really is worth the effort and this leads us to an interesting language design opportunity for Quorum. Specifically, in Quorum 1.7, we have derived a type inference system that removes most type annotations inside of methods, keeps annotations in method declarations, and keeps static type checking itself. More study is needed, but compromises like this may afford ease of use for all.

We suspect that some might believe our results imply dynamic typing is better for novices, but we find this explanation implausible. First, we already know that static typing affords faster programming later in the educational pipeline [Hanenberg 2010a; Kleinschmager et al. 2012; Mayer et al. 2012]. Second, the syntactic benefit of removing the type annotations is relatively small in our studies; a total of only one point in task 6 and slightly more in other tasks. Third, not all dynamically typed languages agree on the syntax and some do a better job than others. For example, the seemingly trivial addition of the $ operator in Perl led to accuracy rates for this character of between 0 and 85%. Other choices in Perl had lower rates still. For example, the `$_[0]` operators for parameters received a score of 0% in study 3 and only 8% in study 4. Languages such as Python and Ruby appear to handle dynamic typing syntax more elegantly by not including an annotation at all. Thus, claiming that dynamic typing is better for novices is incorrect without the proper context.

While we have criticized Quorum and Perl, no language made excellent choices across the board. While a complete analysis of all tasks is beyond the scope of this paper, consider task 4 for the Ruby programming language, Figure 6. While Ruby has made some excellent decisions, especially in regards to its conditional syntax, the `for` construct is not one of the language's bright spots. Novices only correctly used the word `for` approximately 18% of the time and the rest of the syntax for this construct did not do much better. Contrast this result with the loop syntax for Quorum, where novices in experiment 4 used the word `repeat` correctly 58% of the time (85% in experiment 3), an increase of 322%. Succinctly, novices can use statements like `repeat 10 times` (Quorum's syntax) more accurately than traditional C-style looping syntax in Java, Ruby, Python, or others. As one final point on these TAM's, notice that there are occasional

"blank" elements and that these blanks still have accuracy values. These positions are an artifact of the coding system, which allow us to make comparisons between languages. For example, Java requires a semicolon at the end of a line, but Ruby does not. While this means that most students cannot possibly make this error, this point is only given if the user did not make another mistake on a particular line. For example, if a novice left a line blank, or left out part of a line, they do not receive a point. As Denny shows, semicolons are a relatively trivial issue for novices anyway [Denny et al. 2012], which our data supports in a very different way.

5. GENERAL DISCUSSION AND THREATS TO VALIDITY

We have presented two very different views toward analyzing the impact of programming language on novices. In our first two studies, we investigate the types of word choices and symbols that might be easy to understand. In the second two, we analyzed the accuracy rates of novices while completing small programming tasks. In this short section, we discuss our overall results and threats to validity.

Our first two studies were designed to give specific clues on the words and symbols that might make sense to novices. We conducted these because many language designers disagree and instructors could probably use some guidance as to the syntactic tradeoffs in language design with beginners. Thus, our first step was to do the obvious: ask hundreds of novices what they think. Now, the threats here are clear; novices may not have understood the concepts, our descriptions may be accidentally biased, our surveys may have no impact on actual performance in the field, and surveys alone are hardly representative of programming language design itself. On the other hand, while these threats are certainly plausible, they also miss the point—our surveys were actually helpful; Quorum improved significantly because of them and this is confirmed in our accuracy tests. For example, the high novice rating of the word `repeat` prompted us to come up with syntax that used this word. This loop syntax was far more successful than more traditional `for` loops in any language we tested. Similarly, the single equals sign was preferred by novices, prompting us to use it. Ironically, the double equals did so poorly that our randomly chosen operator, bang (!) did just as well (if not better). Using surveys was a calculated gamble, but they provided an important check and balance in our design ideas. In short, academics should be cautious not to overstate the importance of surveys, but they should be equally cautious of cynicism toward a technique that is used widely in scientific literature.

Surveys aside, a number of additional threats to validity come up for our second set of studies. Importantly we must acknowledge that computer programming is a complex task. While our tasks are representative of the types of tasks a novice at the very beginning of their career might be able to accomplish, they are also simple and not representative of what professionals do in the field or perhaps even what students do in a college level course. It remains unclear how syntactic changes impact novices in the field, especially given that novices in the classroom are taught using a variety of pedagogical styles (e.g., scaffolded instruction, studio based learning). Like any experiment, we have tried to balance internal and external validity. As experiments become more tightly controlled, they tend to become less representative of work in practice and our studies are no exception. On the other hand, there is an astonishingly small number of archival quality randomized controlled trials on language design in the academic literature. Our randomized trials appear to be largely replicable, but we think that raising our sample size further, changing our examples, examining students at other points in the academic pipeline, and testing new features (e.g., closures, prefix notation, enum syntax, other loop constructs, switch statements) in a controlled and systematic way could be of great benefit to our work and the literature. Finally, while the use of a placebo programming language is acceptable for a short and carefully

controlled experiment, we believe there are ethical issues with teaching Randomo in the classroom, even if the results would be, admittedly, rather fascinating.

6. SUMMARY AND FUTURE WORK

Analyzing how humans use and interact with computer programming languages is a difficult research problem. Clearly, a great number of factors can influence human performance (e.g., support tools, pedagogical strategies) and different groups likely have different needs (e.g., students, professionals). Our studies here provide what may be the first randomized controlled trials with placebo in this design space, which were developed as part of a long-term investigation into the multiple languages problem. Given the myriad of complex statistics throughout this text, we want to offer a concise take-home message, guided from observations about our data. Overall, our findings include the following.

(1) *Perceived intuitiveness varies across word choices and language constructs*. While programmers become familiar with the somewhat esoteric terminology used in programming languages, non-programmers appear to rate higher those words that are both common in English and literal (e.g., for the concept of iteration, words such as `repeat` instead of `for`).
(2) *Perceived intuitiveness varies between programmers and non-programmers*. When measuring intuitiveness scores of those with self-reported years of C++ experience, individuals rated C++ constructs an average of about half a point higher per year on an 11-point Likert scale. Given this, teachers, users, and designers of programming languages, especially those with significant experience, would be wise to check their assumptions before claiming a design is intuitive.
(3) *Perceived intuitiveness varies across languages as a whole*. When aggregating words and phrases across languages, some languages are considered more intuitive than others, by both programmers and non-programmers. Quorum is perceived as being particularly intuitive.
(4) *Artifact Encoding and Token Accuracy Maps (TAMs) provide us a methodology for measuring novice programming accuracy*. These techniques allow us to gather information related to accuracy both in aggregate (between languages) and in the context of language constructs (within languages).
(5) *Some programming languages may not afford accuracy rates much higher than placebo*. We document evidence that some languages (Ruby, Python, and Quorum) do afford accuracy rates higher than placebo, but found insufficient evidence to determine that this is the case for either Java or Perl.
(6) *Novices may have initial difficulty with static type annotations*. Given that previous work has shown benefits of static typing [Hanenberg 2010a; Kleinschmager et al. 2012; Mayer et al. 2012], this result may imply that type systems only grant benefits once individuals garner experience.
(7) *Novice accuracy varies across language constructs*. While our results are preliminary, Token Accuracy Maps suggest the following.
    (a) Looping constructs that use intuitive word choices afford higher accuracy (e.g., `repeat 10 times`).
    (b) If statements that remove parentheses, braces, and use a single equal sign afford higher accuracy.
    (c) Type annotations and semicolons appear to cause minor deficits in novice accuracy (e.g., `a = 5` has higher accuracy than phrases like `int a = 5;`). Languages with dynamic typing that include a $ symbol (e.g., `$a = 5;`) show slightly lower accuracy rates across the construct.

(d) Quorum is regularly changed to conform to the best available evidence regarding syntactic choices, including the recommendations found in this article. All else being equal, new language designers that want their language to be easy to understand should consider building on Quorum's syntax as a foundation.

While our studies provide new information into how students interpret and use an array of alternative programming language designs, they offer but a glimmer toward understanding the broad multiple languages problem. If the research community is serious about obtaining a deeper understanding of how humans use programming languages, significantly more evidence from a broad community will be required. For example, the major conferences on programming language design make a vast number of academic claims regarding language design, many of which have not been evaluated with human users. For future work, we urge the research community to take a critical eye to the multiple languages problem and the impact it has on our world. Claims made by language designers should not simply be accepted, but subjected to rigorous experiment, formal analysis, and most crucially, with full availability of raw anonymized data and experimental protocols so that the work can be replicated by independent research groups.

7. CONCLUSION

In this article, we conducted four empirical studies. The contribution of this work is a rigorous empirical approach to analyzing the syntax of programming languages. Results show that many aspects of traditional C-style syntax, while it has influenced a generation of programmers, exhibits problems in terms of usability for novices. Alternative syntactic constructs should be considered and tested. In future work, we plan to continue our investigation into programming languages with a family of randomized controlled trials, including further tests of syntactic designs (e.g., exceptions, functions, generics, regular expressions) and tests going beyond syntactic representations (e.g., the use of type systems, parallelism, closures, the design of standard libraries). Our goal is to neutrally investigate the alternatives and to iteratively refine Quorum according to the best available evidence. Thus, we hope that Quorum will serve as a testbed for other researchers to compare against for analyzing alternatives. Finally, barring more, or better, empirical evidence, designers creating new programming languages should consider building on Quorum's syntax as a foundation.

ACKNOWLEDGMENTS

We would like to thank Melissa Stefik, Jeff Wilson, Kim Slattery, David Henry, and Sahana Tambi Sathyanarayana for their help in collecting data for the various experiments. We would also like to thank Melissa Stefik for her work on the Quorum compiler, Jeff Wilson for his on the Quorum debugger, and Ryan Vlazny and Brandon Spencer for their help in developing the Quorum curriculum. In addition, we would like to thank all of the students who were willing to share their time with us in our studies.

REFERENCES

Beaubouef, T. and Mason, J. 2005. Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bull. 37*, 2, 103–106.

Begel, A. and Graham, S. L. 2004. Spoken language support for software development. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'04)*. 271–272.

Bigham, J. P., Aller, M. B., Brudvik, J. T., Leung, J. O., Yazzolino, L. A., and Ladner, R. E. 2008. Inspiring blind high school students to pursue computer science with instant messaging chatbots. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'08)*. 449–453.

Binkley, D., Davis, M., Lawrie, D., and Morrell, C. 2009. To CamelCase or under score. In *Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC'09)*. 158–167.

Bonar, J. and Soloway, E. 1983. Uncovering principles of novice programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'83)*. 10–13.

Borning, A. and O'Shea, T. 1987. Deltatalk: An empirically and aesthetically motivated simplification of the Smalltalk-80 language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'87)*. 1–10.

Brown, M. H. and Hershberger, J. 1991. Color and sound in algorithm animation. In *Proceedings of the IEEE Workshop on Visual Languages (VL'91)*. 10–17.

Cleary, B., Exton, C., Buckley, J., and English, M. 2009. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empir. Softw. Eng. 14*, 93–130.

Comstock, C., Jiang, Z., and Naudé, P. 2007. Strategic software development: Productivity comparisons of general development programs. *World Acad. Sci. Eng. Tech. 34*, 25–30.

Cooper, S. 2010. The design of Alice. *Trans. Comput. Educ. 10*, 4, 15:1–15:16.

Dann, W., Cosgrove, D., Slater, D., Culyba, D., and Cooper, S. 2012. Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. 141–146.

Deißenböck, F. and Pizka, M. 2005. Concise and consistent naming. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*. 97–106.

Delorey, D. P., Knutson, C. D., and Chun, S. 2007. Do programming languages affect productivity? A case study using data from open source projects. In *Proceedings of the 1st International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07)*.

Delorey, D. P., Knutson, C. D., and Davies, M. 2009. Mining programming language vocabularies from source code. In *Proceedings of the Psychology of Programming Interest Group Conference (PPIG 2009)*.

Denny, P., Luxton-Reilly, A., and Tempero, E. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12)*. 75–80.

Denny, P., Luxton-Reilly, A., Tempero, E., and Hendrickx, J. 2011. Understanding the syntax barrier for novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE'11)*. 208–212.

Dolado, J., Harman, M., Otero, M., and Hu, L. 2003. An empirical investigation of the influence of a type of side effects on program comprehension. *Softw. Eng. 29*, 7, 665–670.

Enbody, R. J. and Punch, W. F. 2010. Performance of Python CS1 students in mid-level non-Python CS courses. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)*. 520–523.

Enbody, R. J., Punch, W. F., and McCullen, M. 2009. Python CS1 as preparation for C++ CS2. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE'09)*. 116–120.

Endrikat, S. and Hanenberg, S. 2011. Is aspect-oriented programming a rewarding investment into future code changes? A socio-technical study on development and maintenance time. In *Proceedings of the IEEE 19th International Conference on Program Comprehension (ICPC'11)*. 51–60.

Feigenspan, J., Kästner, C., Liebig, J., Apel, S., and Hanenberg, S. 2012. Measuring programming experience. In *Proceedings of the IEEE 20th International Conference on Program Comprehension (ICPC'12)*.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. 2002. Drscheme: A programming environment for scheme. *J. Funct. Program. 12*, 2, 159–182.

Garlick, R. and Cankaya, E. C. 2010. Using Alice in CS1: A quantitative experiment. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'10)*. 165–168.

Green, T. R. G. and Petre, M. 1996. Usability analysis of visual programming environments: A cognitive dimensions framework. *J. Vis. Lang. Comput. 7*, 2, 131–174.

Hanenberg, S. 2010a. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. 22–35.

Hanenberg, S. 2010b. Faith, hope, and love: An essay on software science's neglect of human factors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. 933–946.

Hanenberg, S., Kleinschmager, S., and Josupeit-Walter, M. 2009. Does aspect-oriented programming increase the development speed for crosscutting code? An empirical study. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09)*. 156–167.

Holmboe, C. 2005. The linguistics of object-oriented design: Implications for teaching. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'05)*. 188–192.

Holt, R. C. and Cordy, J. R. 1988. The Turing programming language. *Comm. ACM 31*, 1410–1423.

Holt, R. C., Wortman, D. B., Barnard, D. T., and Cordy, J. R. 1977. SP/k: A system for teaching computer programming. *Comm. ACM 20*, 301–309.

Høst, E. W. 2007. Understanding programmer language. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA '07)*. 943–944.

Høst, E. W. and Østvold, B. M. 2007. The programmer's lexicon, volume I: The verbs. In *Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07)*. 193–202.

Hubert, L. 1977. Kappa revisited. *Psychol. Bull. 84*, 2, 289–297.

Hundhausen, C. D., Farley, S. F., and Brown, J. L. 2009. Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study. *ACM Trans. Comp. Hum. Interac. 16*, 3, 1–40.

Kaptchuk, T. J. 1998. Intentional ignorance: A history of blind assessment and placebo controls in medicine. *Bull. Hist. Med. 72*, 3, 389–433.

Kelleher, C. and Pausch, R. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv. 37*, 2, 83–137.

Kiczales, G. 1996. Aspect-oriented programming. *ACM Comput. Surv. 28*, 4es, 154.

Kleinschmager, S., Hanenberg, S., Robbes, R., Tanter, E., and Stefik, A. 2012. Do static type systems improve the maintainability of software systems? An empirical study. In *Proceedings of the IEEE 20th International Conference on Program Comprehension (ICPC'12)*.

Kline, P. 2002. *An Easy Guide to Factor Analysis*. New York, NY: Routledge.

Ko, A. J. and Myers, B. A. 2009. Finding causes of program output with the Java whyline. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems (CHI'09)*. 1569–1578.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., and Wiedenbeck, S. 2011. The state of the art in end-user software engineering. *ACM Comput. Surv. 43*, 3, 21:1–21:44.

Lewis, B. and Ducassé, M. 2003. Using events to debug Java programs backwards in time. *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*. 96–97.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., John, H., Lindholm, M., Mccartney, R., Mostroem, J. E., Sander, K., Seppaelae, Simon, B., and Thomas, L. 2004. A multi-national study of reading and tracing skills in novice programmers. In *Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE'04)*. 119–150.

Lopes, C. V., Dourish, P., Lorenz, D. H., and Lieberherr, K. 2003. Beyond AOP: Toward naturalistic programming. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*. 198–207.

Lukas, G. 1972. Uses of the LOGO programming language in undergraduate instruction. In *Proceedings of the ACM Annual Conference - Volume 2 (ACM 72)*. 1130–1136.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. 2010. The scratch programming language and environment. *Trans. Comput. Educ. 10*, 4, 16:1–16:15.

Markstrum, S. 2010. Staking claims: A history of programming language design claims and evidence: A positional work in progress. In *Proceedings of the Conference on Evaluation and Usability of Programming Languages and Tools (PLATEAU'10)*. 7:1–7:5.

Mayer, C., Hanenberg, S., Robbes, R., Tanter, E., and Stefik, A. 2012. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*.

Mayrhauser, A. V. and Vans, A. M. 1997. Program understanding behavior during debugging of large scale software. In *Proceedings of the 7th Workshop on Empirical Studies of Programmers (ESP'97)*. 157–179.

McIver, L. K. 2001. Syntactic and semantic issues in introductory programming education. Ph.D. thesis, Monash University.

Meyerovich, L. A. and Rabkin, A. S. 2012. Socio-PLT: Principles for programming language adoption. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'12)*. 39–54.

Myers, B. A., Pane, J. F., and Ko, A. 2004. Natural programming languages and environments. *Comm. ACM 47*, 9, 47–52.

Myers, B. A., Ko, A. J., Park, S. Y., Stylos, J., Latoza, T. D., and Beaton, J. 2008. More natural end-user software engineering. In *Proceedings of the 4th International Workshop on End-User Software Engineering (WEUSE'08)*. 30–34.

Pane, J. F., Myers, B. A., and Ratanamahatana, C. A. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Hum.-Comp. Stud. 54*, 2, 237–264.

Pausch, R. 2008. Alice: A dying man's passion. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'08)*. 1–1.

Pennington, N. 1987a. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard, E. Soloway, and B. Shneiderman Eds., Greenwood Publishing Group Inc., 100–113.

Pennington, N. 1987b. Stimulus structures and mental representations in expert comprehension of computer programs. *Cogn. Psych. 19*, 295–341.

Pinker, S. 1991. Rules of language. *Sci. 253*, 530–535.

Pothier, G., Tanter, E., and Piquer, J. 2007. Scalable omniscient debugging. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07)*. 535–552.

Ramalingam, V. and Wiedenbeck, S. 1997. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Proceedings of the 7th Workshop on Empirical Studies of Programmers (ESP'97)*. 124–139.

Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. 2009. Scratch: Programming for all. *Comm. ACM 52*, 11, 60–67.

Rossbach, C. J., Hofmann, O. S., and Witchel, E. 2010. Is transactional programming actually easier? *SIGPLAN Not. 45*, 5, 47–56.

Sanchez, J. and Aguayo, F. 2005. Blind learners programming through audio. In *Extended Abstracts on Human Factors in Computing Systems (CHI'05)*. 1769–1772.

Schulte, C. and Magenheim, J. 2005. Novices' expectations and prior knowledge of software development: Results of a study with high school students. In *Proceedings of the 1st International Workshop on Computing Education Research (ICER'05)*. 143–153.

Science House of Commons and Technology Committee. 2010. Evidence check 2: Homeopathy, fourth report of session 2009-10. http://www.publications.parliament.uk/pa/cm200910/cmselect/cmsctech/45/45.pdf.

Smith, A. C., Cook, J. S., Francioni, J. M., Hossain, A., Anwar, M., and Rahman, M. F. 2004. Nonvisual tool for navigating hierarchical structures. In *Proceedings of the10th International ACM SIGACCESS Conference on Computers and Accessibility (SIGACCESS'04)*. 133–139.

Soloway, E., Bonar, J., and Ehrlich, K. 1983. Cognitive strategies and looping constructs: An empirical study. *Comm. ACM 26*, 11, 853–860.

Stefik, A. 2008. On the design of program execution environments for non-sighted computer programmers. Ph.D. thesis, Washington State University.

Stefik, A., Alexander, R., Patterson, R., and Brown, J. 2007. WAD: A feasibility study using the wicked audio debugger. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*.

Stefik, A. and Gellenbeck, E. 2009. Using spoken text to aid debugging: An empirical study. In *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC'09)*. 110–119.

Stefik, A. and Gellenbeck, E. 2011. Empirical studies on programming language stimuli. *Softw. Qual. J. 19*, 1, 65–99.

Stefik, A., Hundhausen, C., and Patterson, R. 2011a. An empirical investigation into the design of auditory cues to enhance computer program comprehension. *Int. J. Hum.-Comp. Stud. 69*, 12, 820–838.

Stefik, A., Hundhausen, C., and Smith, D. 2011b. On the design of an educational infrastructure for the blind and visually impaired in computer science. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (CSE'11)*.

Stefik, A., Siebert, S., Stefik, M., and Slattery, K. 2011c. An empirical comparison of the accuracy rates of novices using the quorum, perl, and randomo programming languages. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'11)*. 3–8.

Steimann, F. 2006. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st Annual ACMSIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. 481–497.

Stylos, J. and Clarke, S. 2007. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th International Conference on Computer Software Education (ICSE'07)*. 529–539.

Stylos, J. and Myers, B. A. 2008. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'08/FSE-16)*. 105–112.

Taylor, J. 1990. Analyzing novices analyzing prolog: What stories do novices tell themselves about prolog? *Instruct. Sci. 19*, 283–309.

Teitelbaum, T. and Reps, T. 1981. The Cornell program synthesizer: A syntax-directed programming environment. *Comm. ACM 24*, 9, 563–573.

Tew, A. E. and Guzdial, M. 2011. The FCS1: A language independent assessment of CS1 knowledge. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11)*. 111–116.

Vickers, P. and Alty, J. L. 2002. When bugs sing. *Interact. Comp. 14*, 6, 793–819.

Vogt, W. P. 2006. *Quantitative Research Methods for Professionals in Education and Other Fields*, 1st Ed. Allyn and Bacon, Columbus, OH.

Whitney, P. 1998. *The Psychology of Language*. Houghton Mifflin Company, Boston, MA.

Received February 2013; revised June 2013; accepted July 2013