2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

An Empirical Study on the Impact of C++ Lambdas and Programmer Experience

Phillip Merlin Uesbeck
University of Nevada,Las Vegas
Department of Computer Science
Las Vegas, Nevada
uesbeck@unlv.nevada.edu

Andreas Stefik
University of Nevada,Las Vegas
Department of Computer Science
Las Vegas, Nevada
andreas.stefik@unlv.edu

Stefan Hanenberg
University of Duisburg-Essen
Institute for Computer Science and Business Information Systems (ICB)
Essen, Germany
stefan.hanenberg@uni-due.de

Jan Pedersen
University of Nevada,Las Vegas
Department of Computer Science
Las Vegas, Nevada
matt.pedersen@unlv.edu

Patrick Daleiden
University of Nevada,Las Vegas
Department of Computer Science
Las Vegas, Nevada
daleiden@unlv.nevada.edu

Lambdas have seen increasing use in mainstream programming languages, notably in Java 8 and C++ 11. While the technical aspects of lambdas are known, we conducted the first randomized controlled trial on the human factors impact of C++ 11 lambdas compared to iterators. Because there has been recent debate on having students or professionals in experiments, we recruited undergraduates across the academic pipeline and professional programmers to evaluate these findings in a broader context.

Results afford some doubt that lambdas benefit developers and show evidence that students are negatively impacted in regard to how quickly they can write correct programs to a test specification and whether they can complete a task. Analysis from log data shows that participants spent more time with compiler errors, and have more errors, when using lambdas as compared to iterators, suggesting difficulty with the syntax chosen for C++. Finally, experienced users were more likely to complete tasks, with or without lambdas, and could do so more quickly, with experience as a factor explaining 45.7% of the variance in our sample in regard to completion time.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Lambda Expressions, Human Factors, C++11

ICSE '16, May 14 - 22, 2016, Austin, TX, USA

ISBN 978-1-4503-3900-1/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884849

1. INTRODUCTION

Modern programming languages are complex abstractions designed to tell the computer what to do. In context, they are used for a wide variety of application domains, from shuttle launches at NASA, to video games, to toys designed to teach students robotics. In the recent past, many general purpose languages designed to be used over these diverse domains have added features from functional programming. As a notable example due to commercial applicability, both Java 8 and C++ 11 have added lambda functions.

While the changes in C++ and Java, amongst other languages, have already occurred and have been deployed worldwide, there appears to be some disagreement in the scientific and development communities as to the impact. It could be that lambdas provide an important new feature that will positively impact people at various experience levels. Or, it could be that such claims are ultimately not supported by evidence.

Given the long history of lambdas, dating back to Alonzo Church in the 30s and more recently by, perhaps especially, the International Conference on Functional Programming (ICFP) community, one might assume the impact on people using them is well studied. As we will discuss, however, we document evidence that the ICFP community has never studied the issue using scientifically replicable techniques like randomized controlled trials (e.g., have a control group, collect evidence, randomize experimental groups, work to make the experiment independently replicable). Specifically, our team read and documented all papers ever published at ICFP and coded them formally by their evidence practices, similarly to previous work [50, 53, 28], finding that only 2.3% contained any information regarding the impact functional programming has on people. Besides this one venue, we were unable to locate any systematic and replicable investigation of the use of lambdas ever, meaning that these languages may have been altered world-wide (causing potentially costly changes to classes, textbooks, and development schedules) without any human factors evidence for or against the design decisions being made.

In this paper, we make two primary contributions. First, we conducted a randomized controlled trial comparing the use of C++

11 lambdas with iterators (our control condition) in a data structure. We tested under these conditions first because in their proposal to add lambda expressions and closures to C++, Willcock et al. [57] set design goals for a lambda syntax stating, "For many uses, lambda expressions and STL-style algorithms compete directly with explicit loops and with explicitly named function objects. Therefore, simple uses of a lambda expression with algorithms must not be significantly more verbose than their equivalent hand-written loops and hand-written function objects. Simple uses must be very simple." Thus, we designed an evaluation on the use of iterators compared to lambdas from this recommendation.

Because it was unclear to us whether the use of lambdas would impact students the same way it impacted professionals, we recruited Freshman, Sophomores, Juniors, and Seniors at the University of Nevada, Las Vegas, in addition to professionals from the field. We did this in part because Höst, Regnell, and Wohlin [20] suggest that differences between students and professionals are minor, as does the more recent work from Salman et al. [42], but we were curious whether these findings would hold under our context. Results showed that 45.7% of the variance is accounted for by differences in experience level, suggesting the differences between students at various levels and professionals, were large in our study. Put simply, experience level impacted how quickly, and whether, a participant could complete our tasks, regardless of the use of iterators or lambdas. Results showed no benefit in regard to time or completion for C++ lambdas at any level, including professionals, and we observed a small, but statistically significant, increase in the number of compiler errors participants had and the percentage of time they spent fixing them. Further, we show that students had difficulty completing tasks using lambdas, but far less so with iterators, whereas all professionals completed all tasks successfully in both groups.

Throughout the rest of this paper, we first discuss arguments for and against lambdas in section 2, then move to related work in section 3. Then in section 4, we describe the hypotheses and the experimental design. Section 5 describes our analyses and results, which we then discuss in section 6. Section 7 describes limitations and future work, after which we conclude.

2. HISTORICAL CONTEXT

The broad goal of this section is to provide the reader with some historical context for why mainstream languages such as Java or C++ have adopted lambdas. We start with reminders on the technical foundation of lambdas and then move to claims made in industry and the literature. We want to be clear here in saying that we harbor no vested interest for or against the idea and have attempted to present the various opinions as neutrally as possible.

The term *lambda expression* or *λ-expression* was coined by Alonzo Church in 1932 as part of the lambda-calculus [4]. In this calculus, used as the foundation of modern programming languages and found in teaching books like those by Pierce [38], a λ-expression (or λ-term) is either a variable, an abstraction, or an application. A variable is comparable to variables or parameters that we find in modern programming languages. An abstraction is something comparable to a function or a procedure: something that can receive parameters in order to be executed. Applications then pass parameters to abstractions in order to execute them.

When the term lambda or lambda expression is used in the context of modern programming languages, it describes the abstraction from the lambda calculus—which is why they are referred to as lambda expressions or (for short) lambdas.[1] The primary difference between a lambda-expression and an ordinary procedure or method is that it does not have a name: lambdas are anonymous. Additionally, lambdas can be defined ad hoc in the code. While programming languages such as Pascal require procedures in separate locations (outside other procedures), lambdas can be defined in bodies of other procedures or methods. Besides these considerations, lambdas can also, depending on the implementation, allow for references to their outer scope (e.g., parameters or local variables) and may be passed as parameters to other functions or methods.

[1]Other words for a similar construct are *blocks* or *closures*.

2.1 Lambdas in Java and C++

Most functional languages are based on λ-expressions. Examples include Lisp and Scheme as well as JavaScript. While object-oriented languages such as Smalltalk had lambda expressions (called *blocks* [13]), more mainstream languages such as Java or C++ did not. However, both languages have similar constructs: in Java, there were anonymous inner classes, while in C++ there were function pointers.

Over time, language designers in both the Java and C++ communities integrated lambdas. To give a sense of the reasoning, Wilcock et al. calls lambda expressions "syntactic sugar for defining function object classes" [57] proposing them to be of practical use. In this sense, a lambda expression syntax design must be "significantly less verbose than defining explicit function objects, not significantly more verbose than explicit loops, and clearer and more intuitive than explicit loops." To our knowledge, the only known studies on language intuitiveness did not appear in the literature until years later (see e.g., [48, 51]). Further, the C++ designers state that "A lambda expression has aspects of a function (it performs an action) and an object (it has state). The primary aim is for lambda expressions to serve as 'actions' for STL algorithms (the way function objects have traditionally been used) and similar callback mechanisms."

To remind the reader of the technical definition, in Java a λ-expression takes the following form:

$$(x_1, x_2, \ldots, x_m) -> \{s_1; s_2; \ldots; s_n;\}$$

where x_i is a parameter, and s_j is a statement and s_n can be a return statement. A concrete example would be:

```
Operation o = (p,q) -> {
        int s = q+p;
        return s-p;
};
int a = 5;
int b = 6;
System.out.println(o.doOperation(a, b));
```

where Operation would be an interface with a method taking two integers as parameters and return an integer. The syntax for C++ λ-expressions is:

$$[\,cl\,]\,(p_1, p_2, \ldots, p_n) -> t\,\{s_1; \ldots; s_n;\}$$

where the *cl* (the capture-list) represents the list of variables of the calling context that are passed to the λ-function. $(p_1, p_2, ..., p_n)$ represents the parameters like any other function. t (the return type of the λ-function) is a regular C++ type, and$s_1; \ldots; s_n;$ is a sequence of C++ statements. A concrete example for this would be:

```
function<int (int, int)> func =
 [] (int p, int q) {
  int s = q+p;
  return s-p;
};
int a = 5;
int b = 6;
cout << func(a,b) << "\n";
```

Given these definitions, it is clear that one of the motivations for integrating lambdas was the application of a map function, as they are known in functional programming. For example, in Scheme, a term such as $(map\ (\lambda\ (x)\ (+\ x\ 1))\ '(1\ 2\ 3\ 4)) = (2\ 3\ 4\ 5)$ applies a function that adds 1 to each element of a list $(1\ 2\ 3\ 4)$. Code with similarities can now be written in Java 8 or C++ 11.

2.2 Arguments in Support of Lambdas

One group that appears in support of the idea of lambdas is Oracle's Java Platform Group. At the JavaOne 2013 Technical Keynote, Mark Reinhold, Oracle's Chief Architect of the Java Platform Group, claimed that lambda expressions are "the single largest upgrade to the programming model ever – larger even than generics." Similarly, Oracle Java Language Architect Brian Goetz followed this by saying "Programming well is about finding the right abstractions. We want the code we write to look like the problem statements it's trying to solve, so we can look at it and immediately know it's correct. Java has always given us good tools for abstracting over data types. I wanted to do better in abstracting over patterns of behavior – that's where lambda comes in... Lambdas are a nicer syntax, but they are also something deeper. It's not just a compiler generating inner classes for you – it uses the invokedynamic feature to get more compact and higher performance code." The duo summarized their position with the claim that "lambda brings 3 weapons to Java – syntax, performance and abstraction... plus parallelism" [2].[2]

Wilcock et al. claim that key language libraries of C++ were not being used as frequently as they could be without lambda expressions because "The lack of a syntactically light-weight way to define simple function objects is a hindrance to the effective use of several generic algorithms in the Standard Library" [57]. Similarly, Samko argues that lambda expressions also allow for better design, stating "Many algorithms in the C++ Standard Library require the user to pass a predicate or any other functional object, and yet there is usually no simple way to construct such a predicate in place. Instead one is required to leave the current scope and declare a class outside of the function, breaking an important rule of declaring names as close as possible to the first use. This shows that lambda functions would add a great [sic] to the expressive power and ease of use of the C++ Standard Library [43]."

In a paper on computer science education courses at Berkeley, Harvey advocates the power of lambda expressions in a discussion of an extension to the Scratch language that enables "procedures that take other procedures as arguments" (i.e. lambda expressions) and stated that "these have proven to be a powerful capability... They are useful, for example, as an alternative to recursion or to looping with index variables when the programmer wants to process all of the elements of a list in a uniform way" and that "by taking key ideas, such as procedures as first class data, from the Scheme language, we can add only a few features to Scratch and still make it powerful enough to support a serious introductory computer science curriculum [18]." A final common supportive argument for lambda expressions seems to be that everyone else does it: "Many programming languages offer support for defining local unnamed functions on-the-fly inside a function or an expression. These languages include Java, with its inner classes; C# 3.0; Python; ECMAScript; and practically all functional programming languages." Also, "anonymous functions within other functions, with access to local variables, are an important feature in many programming languages" [21].[3]

2.3 Arguments Against Lambdas

One plausible argument against lambda expressions is that they are not strictly necessary because alternative language features support the same functionality. Including multiple ways to complete the same task may or may not benefit programmers from a human factors perspective. For example, in discussing lambda expressions in a C++11 FAQ on his webpage, Bjarne Stroustrup states that the "Primary use for a lambda is to specify a simple action to be performed by some function... A lambda expression can access local variables in the scope in which it is used...Some consider this 'really neat!'; others see it as a way to write dangerously obscure code. IMO, both are right [52]." The second implication of this statement is that lambdas might lead to obscure code, seemingly a disagreement with the claims made from the Java One technical keynote that syntax or clarity are achieved through lambdas.

We have also spent some effort reading blogs and other less scholarly opinions on lambdas. This work is generally not peer reviewed, obviously, but in some cases, listening to developers discuss their work is interesting, as some of these individuals have significant practical experience working with various programming technologies. In reading these venues, one theme we see repeatedly is a concern that lambda expressions increase the amount of time it takes for a human to debug. In a blog post called "The Dark Side of Lambda Expressions in Java", Weiss argues that adding features like lambdas to Java may make it less obvious how the source code's structure forms the JVM bytecode, which he claims is the case in the language Scala [56]. The idea here is basically that as the difference between the source code and bytecode grows, it makes it more difficult to trace and understand errors (e.g., long and complex stack traces).

Some developers claim that lambda expressions are confusing. In an article entitled Pros and Cons of Lambda, Ferg attributes this to a misunderstanding of the differences between the concepts of procedures and functions [11]. We do not know whether it is related, but we do find it interesting that, as Meyerovich and Rabkin point out, the adoption rate of functional languages tends to be lower and even when adopted, they are dropped more quickly [32]. Ultimately, not all of these views can be correct, as the claims are contradictory. This leads us to our next section, where we explore the studies that do exist in the literature, focusing especially on those at ICFP.

3. RELATED WORK

In order to find studies related to our research, we first investigated corresponding literature in order to search for experiments we could replicate on lambdas – hence we give a short description of existing literature surveys. Then, we discuss works that are more generally related (i.e. works that study other programming language constructs in experiments). Finally, we introduce works that study language constructs based on historical data. We explicitly do not consider works that study lambda expressions from a technical perspective. For example, while the work is obviously valuable, we do not discuss means to statically type lambda expressions in object-oriented languages or efficiency papers on lambda expressions (see [3] as an example).

[2]Note that in regard to syntax, previous randomized controlled trials have already shown that for novices, Java's syntax is no more usable than one designed randomly from the ASCII table [51]. Other studies on large samples of compiler errors with novices have shown similar problems with Java [8, 7, 1].

[3]The original quotation included citations from the original paper. These have been removed for readability in the current one.

3.1 Literature Surveys

The systematic literature surveys performed by Kitchenham et al. identified a number of empirical studies on different research topics in software engineering [26]. Among them, lambda expressions (or a related topic) are not mentioned. A comparable study later on by Zhang et al. also does not mention lambda expressions or related topics as being in the focus of one or more studies [59]. Finally, another literature study by Kaijanaho with the focus on what empirical studies exist on programming language constructs did not mention that studies about lambda expressions or a related topic could be found in the literature [24].

While the previous surveys tried to find literature independent of the venues where they potentially occur, the study by Stefik et al. studied the papers of the workshops PPIG, PLATEAU and ESP, workshops dedicated to the impact language design has on people. In this work, papers were coded individually and combined with respect to a number of different criteria (e.g., whether it includes a human factors study) [50]. However, no papers regarding lambdas were found at any of these venues.

3.2 Experiments on Programming Language Constructs

Broadening the scope of related works to empirical studies in general (as broadly described in books such as [46] or with the focus on controlled experiments introduced in teaching books such as [58]) reveals that a variety of studies on the human factors impact of programming language constructs exist. We discuss these here.

First, we discuss studies that focus on the usability of programming language constructs in order to achieve a certain task. Examples of such works are the recent experiment series on the impact of static type systems (in comparison to dynamic), which is being studied from a human factors perspective (see [27, 31, 9, 37, 15, 16, 12, 14] among others). While the results of these experiments are not crucial to understand for the present paper, the relevant aspect is how the usability of language constructs is measured: many works study usability by measuring the time until a certain task is completed where the fulfillment of the tasks is checked by corresponding test cases. The interpretation of this measurement is that a faster development time is an indicator of better usability – a way of studying language usability that could already be found in first text books on empirical studies of programming such as the one by Shneiderman [45].

Second, a variety of studies focus on the notation of language constructs. A larger controlled experiment on this issue with the focus on alternative language syntax constructs was done by Stefik and Siebert [51]. Other works that compare different kinds of syntax errors were done by Denny et al. [8, 7] and more recently by Altadmri and Brown [1], amongst others. Such studies indicate that syntax and semantics has a very large impact on students. Studies on professional programmers also show evidence that issues like compiler errors harbor a human factors impact [44], but how they impact these users is, not surprisingly, different. For example, novices have trouble with static typing [1], while other studies generally see a benefit once participants gain some experience [19, 31, 27, 17].

While the previously described studies analyzed certain facets of language constructs by building experiments, other works apply techniques such as repository mining (i.e. the analysis of previously existing or historical data). Examples of such works are the study about the usage of generic types by Parnin et al. who evaluated the use of generic types in the programming language Java [35, 36]. The claim in such works is that use of a certain language construct is an indicator of its impact in practice. These methods can provide significant insight, but one downside is that such works require the availability of the constructs in repository commits, which can only occur after a feature has been deployed. However, after a language feature has seen world-wide deployment, removing it may be difficult, even if it is ultimately found to have a negative impact. Other recent examples of this style of study exist [40, 33].

3.3 ICFP and Programming Language Venues

While the usage of lambda expressions especially in the functional programming community is daily practice, it seems plausible to look into the results to see whether there are indicators (or evidence) that lambda expressions are helpful in any known context or condition. While the related work on literature surveys indicated that no studies exist about lambdas, we (still) want to know whether empirical data exists in the literature about functional programming or about the usability of lambda expressions. In order to investigate thoroughly, we followed a similar approach by Stefik et al., who classified workshop papers according to criteria such as "includes human studies" [50].

The International Conference on Functional Programming (ICFP) started in 1996 and we evaluated all of these papers, through 2014, to try to determine the overall human factors evidence standard. Altogether we found 562 papers in the ICFP proceedings (including short abstracts for keynotes). Among these papers, 13 contained a human-centered study (while 129 papers contained a quantitative empirical study), according to our criteria. The first human-centered studies we identified were published at ICFP in 2003 (see [22, 34]). Having said this, ICFP papers analyzing human factors generally do not follow scientific conventions used in empirical disciplines. For example, one of the approaches used in education is the WWC standard [54], which was established by the U.S. Department of Education's Institute of Education Sciences in order to "assess the quality and findings of existing research." This standard looks at issues such as whether a study had a control group or whether a paper contained data. In this specific case, even amongst the 13 papers found, they generally did not have control groups. The WWC is not perfect and requires statistical expertise to fully understand, but most of the ICFP studies would not meet these well documented evidence standards.

While some of these papers seem to share a common doubt that enough empirical knowledge exists (for example Crestani and Sperber state that "there are surprisingly few constructive investigations of how particular design elements of a programming language can support or hinder a beginner's effort to learn programming [5]"), none of these papers directly refer to the usage of lambda expressions. The only study that seems to be somehow related is one by Pop, who describes in an experience report the application of the programming language Haskell in a Python project. The claim in this paper is that "the most significant problem is the high barrier to entry [39]." Unfortunately, the Pop paper does not give a more detailed description of the described high barrier, and is thus not directly testable. Hence, it is unclear whether or to what extent this can be reduced to the usability of lambdas.[4] While this section predominately discusses ICFP, we found no studies evaluating the human factors impact of lambdas in other programming language venues either (e.g., OOPSLA's entire history). Hence, we conclude again from this additional literature study that no empir-

[4]Taking into account that the application of a whole language is being studied, it is plausible that this high barrier has more something to do with a different semantics of the language, different syntax, different tool support, or other issues.

ical human-centered studies about the usability of lambdas have been conducted.

4. EXPERIMENT

In this section, we discuss the design of our randomized controlled trial on lambda expressions. For the reader unfamiliar with the types of statistical procedures we use for our randomized controlled trials, the standard designs for such experiments from the literature, or other procedures, we recommend reading Vogt [55]. For those unfamiliar with the history of decisions made in regard to our experimental design (e.g., why use randomization? why have a control group?), we recommend reading Kaptchuk [25]. Kaptchuk describes extensively the history of randomized controlled trials and how they developed over time, which is useful in learning about the debates that have already occurred in other fields since the 18th century. Finally, for those unfamiliar with how to evaluate empirical studies in regard to their design and properties, we recommend looking at the WWC standard [54]. As stated above, it provides a thoughtful and objective way to evaluate the design of empirical studies. We designed the experiment here to meet the evidence standards suggested under WWC (e.g., have a control group, use randomization, make the experimental conditions as similar as possible).

4.1 Hypotheses

The aim of this experiment is to find evidence on the impact of lambda expressions in a data structure compared to iteration. To investigate, the following null hypotheses were postulated:

Null hypothesis H0-1: There is no impact on the ability of developers to complete programming tasks using lambda expressions as compared to iterators in C++.

Null hypothesis H0-2: There is no impact on the time it takes developers to correctly complete programming tasks using lambda expressions compared to using iterators in C++.

Null hypothesis H0-3: There is no impact on the number of compiler errors developers have when they complete programming tasks using lambda expressions compared to using iterators in C++.

Null hypothesis H0-4: There is no impact on the percentage of time developers spend in a non-compiling state when using lambda expressions compared to using iterators in C++.

*Null hypothesis H0-*5: Experience level, defined by the position an individual has within an academic pipeline or professional status, has no impact on developer performance under any condition.

As is common in experiments, the null hypothesis will not be rejected if the data between the two groups does not show significant differences. If the null hypothesis cannot be refuted, we would interpret this as insufficient evidence to argue that lambda developers performed better or worse under the context of the experiment. While not stated explicitly, as it is obvious, the interaction between these hypotheses is typically tested by modern statistical packages (e.g., experience may impact lambda/iterators unevenly). The alternative hypotheses can be trivially derived from the null hypotheses.

In terms of variables, we have four dependent variables, 1) whether a task was completed successfully, 2) Time on task to complete programming tasks, 3) the number of compiler errors received by the developer, and 4) the percentage of time developers spend with the program not compiling. We have three independent variables: 1) group (lambda vs. iterator), 2) Tasks (1 - 3) and 3) level of experience. This level of experience variable is described in our model as nominal, with it being either Freshman, Sophomore, Junior, Senior, or Professional. For academic experience level, we confirmed enrollment with the University of Nevada, Las Vegas. For professionals, participants self-reported their place of work and years of experience. For all professionals, we spoke to each participant individually and confirmed that they were full-time programmers in industry.

4.2 Experimental Design

The experiment had two groups, one using lambda expressions and one using iterators (the control). Each participant had to solve four programming tasks, starting with a warmup to get them used to the environment (the warmup was unrelated to lambdas or iterators). All the tasks focused on iteration on a collection using a C++ `vector` object. We chose to test lambdas versus iterators in part because in our analysis of blogs and the literature, developers and scholars consistently claim that lambdas are better in this context and we wanted to determine whether this view would be supported or refuted by evidence. The designs by Wilcock [57] mention a similar idea as a design goal.

4.3 Study Protocol and Participants

The population for this study was recruited from students and professionals. For students, we recruited freshmen, sophomores, juniors and seniors from different classes at the University of Nevada, Las Vegas. We recruited professional developers via Twitter and a mailing list for professional programmers. For students, we confirmed their class schedule and level with the university to get a better understanding of their skills at that point in the academic pipeline, in addition to speaking to them. Specifically, those we are calling "freshman" had taken one C++ class, but had neither studied iterators nor lambdas. For sophomores, while they were enrolled in a data structures course at the time the study was given, they also had not studied iterators or lambdas formally. For Juniors, students were taking one of several computer science courses at that level, including programming languages. By this time, students have been introduced to lambdas and have used iterators. Seniors were recruited from our compilers and software engineering courses and had used iterators and lambdas. For professionals, all were highly experienced programmers and averaged approximately 14 years of self-reported experience.

Like most experiments, we followed a standard protocol, which we will make available to other researchers in a replication packet that includes all tasks and other information required for other scholars to test our work in a different setting. Before the experiment began, participants were randomly assigned to one of two experimental groups and were additionally classified by their confirmed academic level. After instructions, participants were given printouts of sample code they could refer to while solving tasks. Group Lambda got code of a C++ program using lambda expressions and group Iterator received code of the same program written using iterators. They then had time to study the samples before starting the tasks and could refer to these samples later. Participants were then instructed to start the first task by double clicking a file in the folder which is opened on the computer in front of them. After a task was finished, the participants were asked to close the IDE and start the next. In all cases, when we advertised for an experiment, we provided no information on what was under test. As such, participants could not have known the hypothesis, experimental tasks, etc. This procedure is commonly used in randomized controlled trials, as it prevents accidental bias and the good subject effect [55].

Because of ethical considerations in running human experiments, we are typically limited to how much time we can ask participants to spend for lab studies. As such, our study protocol negotiated with the university campus ethics board specifies a maximum time

to work on a task. A previous pilot study on our current experiment indicated that, especially for those with less experience, some participants could not solve the tasks and a timeout was needed. We decided to set the limit to 40 minutes, a number we derived from pilot data. If a participant did not solve the task within 40 minutes, they achieved the maximum time. While this approach was required for important ethical reasons, it has the downside that it may underestimate the time taken for some groups to complete a task. For example, a freshman may be unable to complete a task, ever, if it is at a certain level of complexity, meaning that the timing values are a lower bound if a participant maxed out. After the experiment, the participants were asked to fill out a survey, which provides us with data on the participants' experience level. Those wanting more information on how experience can be measured broadly should see Siegmund et al. [10, 47] or Crk et al. [6].

4.4 Tasks

Each group was asked to solve four tasks, the first of which was a warmup. The warmup task asked the participants to write a loop to iterate through and operate on all the elements of a vector. All participants had seen examples of this type of operation before, even freshman, so the task is neutral across experimental groups.

The other three tasks were different between the two groups and were always presented in the same order. Group Lambda was instructed to use lambda expressions as seen in their code samples and group Iterator used the other. Participants received templates for a program and were asked to fill it in. The programs for tasks two through four contained a method that expected one or more lambda function as parameters or an implementation of an iterator respectively. The participants in group Lambda had to write one or more lambda functions and call the provided method using what they implemented to complete the task. Group Iterator had to use the provided iterator class in a loop to solve the same problem. The tasks were designed to be the same, but solved using different programming constructs.

Specific instructions on how to solve a given task were given in the file the participants had to change. This file was automatically opened when a task was started and contained comments on the task. These comments were the same between conditions and described what to do to solve the task, but not how to solve it. When the participants felt like they successfully solved a given task, they could use a button in the development environment to have the IDE (a derivation of eclipse we designed for the study) run automatic unit tests against their solution. If all tests passed then the task was complete and the participant could move on. If the tests were not all satisfied or the compilation failed, the console of the development environment showed the output of the compiler or the testing framework.

We used the clang C++-compiler and the error output provided is clang's standard. To write the tests, we used the Google Test framework. This framework provides feedback on expected solutions and the result of running a program. Participants could see the source and header files in their programming environment, but not the tests.

In Figure 1, we can see the second task of group Lambda in its solved state. The solution is written between the comments. Participants in this case had to create a lambda expression that is of the type `function<void (item)>` and that added to the already provided `retVal` variable. From there, they called the method in `marketBasket` that expected a lambda function of the same type as an argument, which then iterated over the collection and applied the function on each element. A noteworthy detail is that the participants had to use [&] for their lambda function, which indicates to the compiler that the function captures the variables around it as reference, in contrast to [], which does not capture any variables, and [=], which captures variables by value. The code samples did not show any way to do this other than the [&] so that participants would not be confused and delayed. In other words, our code samples can be considered generous for the lambda group, as we asked participants to use lambdas in only simple ways. This is important, as it means that any impact of lambdas observed is in its simplest use case, meaning that other uses we found in our review may be much harder for developers [56].

The `marketBasket` class of group Lambda task 1 (ignoring the warmup task) can be seen in Figure 2. It contains a method to insert elements into the vector, which is a member of the `marketBasket` class and which also is the collection that is iterated upon. It also contains the method `iterateOverItems(function<void (item)>)`, which is the method the participants in group lambda had to call with their lambda function.

```
#include "task.h"
#include "marketBasket.h"
using namespace std;

/** Please write a program that calculates the sum of the
      prices of the items using the same technique as
     seen in the sample code. Assign the result to retVal
     . **/

float getSum(marketBasket mb) {
   float retVal = 0;
   // Implement solution here
   // ----------
   function<void (item)> func = [&] (item theItem) {
        retVal += theItem.price;
   };
   mb.iterateOverItems(func);
   // ----------
   return retVal;
}
```

Figure 1: The solution for Task 1, group Lambda.

```
#include "marketBasket.h"
using namespace std;

void marketBasket::insert(string itemName, float
     itemPrice) {
   item newItem;
   newItem.name = itemName;
   newItem.price = itemPrice;
   items.push_back(newItem);
}

void marketBasket::iterateOverItems(function<void (item)>
      f) {
   for (vector<item>::size_type i = 0; i < items.size();
        i++) {
      f(items[i]);
   }
}
```

Figure 2: This code was given to participants as part of Task 1, group Lambda.

In Figure 3 you can see the solution to the first task for group Iterator. The solution is, like before, between the comments. Here, a participant had to get the iterator of the `marketBasket` object and then use it in a loop, calling the iterators `next()` method to get the next element and adding the values of each element to the provided `retVal` variable. The iterators `hasNext()` method had to be used as a condition to continue the loop.

The details of group Iterator's `marketBasket` class for task 1 can be seen in Figure 4. The class contains the same insert method as the one in group Lambda's `marketBasket` class, as well as the inner `iterator` class which the participants had to use to iterate over the elements of the collection.

```
#include "task.h"
#include "marketBasket.h"
using namespace std;

/** Please write a program that calculates the sum of the
      prices of the items using the same technique as
     seen in the sample code. Assign the result to retVal
     . **/
float getSum(marketBasket mb) {
    float retVal = 0;
    // Implement solution here
    // ---------
    marketBasket::iterator iter = mb.begin();
    while(iter.hasNext()) {
      retVal += iter.get().price;
      iter.next();
    }
    // ---------
    return retVal;
}
```

Figure 3: The solution for Task 1, group Iterator.

```
#include "marketBasket.h"
using namespace std;

void marketBasket::insert(string itemName, float
     itemPrice) {
   item newItem;
   newItem.name = itemName;
   newItem.price = itemPrice;
   items.push_back(newItem);
}

marketBasket::iterator::iterator(marketBasket *owner) :
     owner(owner), index(0) { }

void marketBasket::iterator::next() { index++; }

bool marketBasket::iterator::hasNext() { return owner->
     items.size() - index > 0; }

item marketBasket::iterator::get() { return owner->items[
     index]; }

marketBasket::iterator marketBasket::begin() { return
     marketBasket::iterator(this); }
```

Figure 4: This code was given to participants as part of Task 1, group Iterator

In task 2, the participants were asked to find the element with the lowest/highest price in the vector which was contained in the `marketBasket` object and return their sum. In the last task, the participants were asked to calculate the sum of the prices of all the items for which the prices are below a certain threshold and return it. Ultimately, these tasks are very similar to the types of situations we have seen argued are positive for lambdas in the literature and we designed them intentionally to be a fair representation of claims made.

4.5 Recording Data

In the experiment, we recorded two things: 1) logging data related to the participant solving the tasks (e.g., compiles, timestamps) and 2) a screencast of the participants while they are trying to solve the tasks. This provided us a record of what happened during the experiment. To make accurate measurements, the experiment uses a modified IDE based on Eclipse.[5] It is slightly modified to work with C++ programs. When a .cpp or .h file is opened, the IDE uses the normal text editor. Syntax highlighting, automatic error checking and method completion are not provided, since they may be slightly different between experimental groups in regard to how they work. The IDE is also generally modified to provide as little support as possible other than what would be expected of a text editor. This was done to try and isolate the language features from other aspects of programming. Previous work has shown that when such experiments are conducted in a more feature rich IDE, they tend to give the same answer [37]. In other words, the literature suggests that such a design, isolating the language feature from the IDE, is reasonable.

The main indicator for effort used in this experiment was development time, as is common practice in a number of experiments (see for example [31, 27, 19, 9]). The measurement of time on each task is started when the participant starts a task and the development environment for the task is started and is stopped when the task is completed successfully. The check if a participant's program satisfies the condition is triggered when the participant presses a button in the IDE, which they learn how to do in the warm-up task. Also, each time the participants use this button, a snapshot of the current state of their code and the output of the compilation and testing processes is recorded. Additional to the IDE's built-in measurements, the actions of the participants are recorded by the software recordmydesktop, which is started when the participant starts a script, initiating the task. Recording stops when the IDE is closed. Finally, all professionals did the experiment remotely, but otherwise used the same IDE, recording tools, and procedures.

5. RESULTS

The experiment had 58 participants. Of these, four data sets were not usable because three of the participants did not follow the experimental protocol and one data set was deleted by accident. Of the usable datasets, 42 participants were students at the University of Nevada, Las Vegas, of which 10 were freshmen, 8 sophomores, 17 juniors, and 7 seniors. The 12 remaining participants were professional programmers. The participants had an average age of 25.89 years (SD = 5.88). The mean programming experience of the participants was 5.59 years (SD = 5.60) and their C++ experience was 2.49 years on average (SD = 2.64). A more detailed breakdown of the ages and programming experience can be found in Table 2. As stated previously, we verified for our study that freshman and sophomores had not yet used iterators or lambdas. Juniors and seniors use these features in their classes and all professionals had used both. We report all results in APA format.

First, before we look explicitly at our hypotheses, we begin with a graph for the percentage of tasks completed by participants at each level of experience, which is shown in Figure 5. As is obvious, younger individuals had considerable trouble with using lambdas. For example, none of the Freshman completed any of the tasks and Sophomores performed little better. Neither group had used iterators or lambdas before the experiment. By the Junior or Senior year, given that these students had been exposed to lambdas by this point in the academic pipeline, performance was not equal, but was closer to parity. Only professionals completed all tasks successfully in both groups. While it is clear that students achieved closer to professional performance as they gained experience (average 4.29 years for seniors) even this length of time practicing C++

[5]http://www.eclipse.org/home/index.php

		T1					T2				T3				Total	
Education	**Group**	**N**	**min**	**max**	**mean**	**SD**	**min**	**max**	**mean**	**SD**	**min**	**max**	**mean**	**SD**	**mean**	**SD**
Freshman	Iterator	6	284	2400	1914.83	859.90	515	2400	1777.00	965.20	1521	2400	2253.50	358.85	1981.78	756.17
	Lambda	4	2400	2400	2400.00	0.00	2400	2400	2400.00	0.00	2400	2400	2400.00	0.00	2400.00	0.00
Sophomore	Iterator	3	1285	2400	1714.67	599.86	287	1032	642.67	373.64	358	2400	1719.33	1178.95	1358.89	872.29
	Lambda	5	1477	2400	2215.40	412.78	487	2400	2017.40	855.52	765	2400	2073.00	731.19	2101.93	646.53
Junior	Iterator	9	346	2400	1227.89	748.77	274	1054	494.00	237.55	155	2400	1189.56	869.43	970.48	735.25
	Lambda	8	1643	2400	2069.50	337.58	283	2400	1396.25	1062.21	187	2400	1108.13	1061.24	1524.63	943.49
Senior	Iterator	3	532	767	675.00	125.53	302	2400	1037.67	1181.07	168	699	449.00	266.85	720.56	660.77
	Lambda	4	2277	2400	2369.25	61.50	307	2400	942.00	979.39	212	2400	826.00	1052.74	1379.08	1049.82
Prof.	Iterator	6	169	308	230.33	50.92	90	424	263.50	106.93	147	268	216.50	47.53	236.78	72.13
	Lambda	6	255	1672	873.83	549.35	193	589	295.67	146.14	99	397	214.67	138.20	461.39	438.04
Total	Iterator	26	169	2400	1151.52	859.76	90	2400	804.81	803.26	147	2400	1186.33	972.40	1047.56	887.29
	Lambda	27	255	2400	1924.19	676.81	193	2400	1348.11	1034.03	99	2400	1237.85	1063.86	1503.38	977.70

Table 1: Descriptive statistics of solving time by group, level and in total

		Age		Programming Exp.		C++ Exp.	
Level of Education	N	Mean	SD	Mean	SD	Mean	SD
Freshman	10	23.60	6.95	2.05	1.55	0.85	0.24
Sophomore	17	23.00	6.57	2.31	1.16	1.69	0.70
Junior	12	24.06	3.73	3.28	1.66	2.44	0.75
Senior	7	27.71	5.50	5.57	3.36	4.29	2.29
Professional	8	31.25	3.70	14.00	5.80	3.42	4.85
total	54	25.89	5.88	5.59	5.60	2.49	2.64

Table 2: Age and experience of participants by level of education

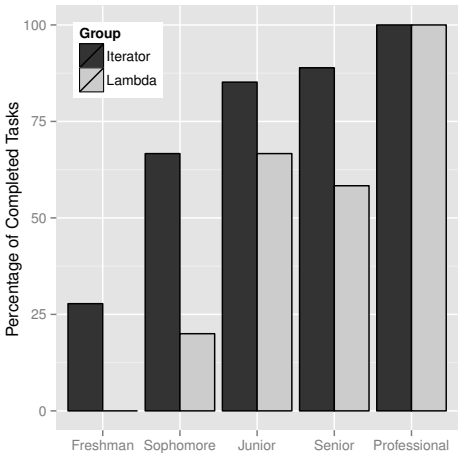

Figure 5: The percentage of tasks completed by participants at each experience level.

was insufficient for them to achieve parity performance with lambdas. While we think it is relatively obvious from the graph alone that this result is significant, we tested this formally with a dependent variable of completed and two independent variables, level of experience, which was significant, $F(4, 152) = 25.414$, $p < .001$, $\eta_p^2 = .401$ (reject H0-5), and group, which was also significant, $F(1, 152) = 16.094$, $p < .001$, $\eta_p^2 = .096$, using an ANOVA (reject H0-1).

Second, we look at the dependent variable *time,* which is the time participants needed to complete the programming tasks. The independent variables for this first test was the *group* the participant was in, which is either iterator or lambda, as well as the *tasks* which are stated as T1 to T3, and their *level of experience*, defined as freshman, sophomore, junior, senior, or professional. There were 27 participants in the lambda group and 26 in the iterator group. The average time it took the iterator group to complete the experiment tasks was (M = 1047.56, *SD* = 887.29) while the average time it took the lambda group was (M = 1,503.38, *SD* = 977.70). Subsampled into level of experience, the freshmen performed worse than all others while having lower times for iterators (*M* = 1,981.78, *SD* = 756.17) than for lambdas (*M* = 2,400, *SD* = 0.00). As stated, none of the freshmen managed to complete the lambda tasks which led to all having the maximum time in this group. This can be seen in Figure 5. The sophomores performed better than the freshmen on iterators (*M* = 1,358.89, *SD* = 872.29) as well as on lambdas (*M* = 2,102.93, *SD* = 646.53). The juniors had (M = 970.48 seconds, *SD* = 735.25) on iterators and (M =1,524.63, *SD* = 943.49) on lambdas. The seniors performed more quickly with (M = 720.56, *SD* = 660.77) and (M = 1,379.08, *SD* = 1049.82) seconds respectively. The professionals performed the fastest on iterators (*M* = 236.78, *SD* = 72.13) and on lambdas (*M* = 461.39, *SD* = 438.04). The graph in Figure 6 shows the mean times of the groups for each task.

To analyze the data, we applied a standard ANCOVA analysis using the statistical package SPSS. To remind the reader what this means, we used SPSS to compute a probability value, using task as a covariate to account for the fact that three tasks were completed in sequence. The ANCOVA then had two fixed factors, group (lambda vs. iterator) and experience (freshman vs. sophomore vs. junior vs. senior vs. professional). The results for the task covariate are $F(2, 159) = 5.985$, $p = .016$, $\eta_p^2 = .038$ which indicates performance on tasks differed and was thus accounted for, which is relatively easy to observe from Figure 6. The impact on group was statistically significant, $F(1, 160) = 20.123$, $p < .001$, $\eta_p^2 = .118$. This means approximately 11.8% of the variation in the experiment seen with individuals was accounted for by differences between iterators and lambdas (Reject H0-2). Experience was also significant, $F(3, 158) = 31.710$, $p < .001$, $\eta_p^2 = .457$, explaining approximately 45.7% of the variance (reject H0-5 under this condition as well). The reader should note that there are always multiple statistical tests that can be used for datasets such as ours. For example, we could have chosen Task as a factor instead of a covariate. We tested many of these alternatives, but found they came to the same broad conclusions. Thus, we chose a conservative statistical model.

Next, we gathered information on the compiler errors that were produced during the experiment and the time it took the participants to fix them (if they could). The compilation errors were extracted from each participant's logs. These snapshots include source code as well as the output of the compilation and testing process, making it possible to analyze the process participants went through when

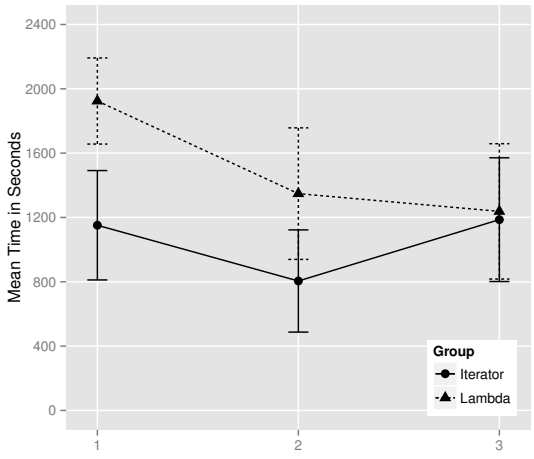

Figure 6: Group times compared

completing the tasks. The amount of time participants spent working on errors was computed as the differences in time between the occurrence of a compilation error and the next successful compilation. These were summed together and then compared to the total time the participants spent on the task.

First we look at the raw number of compiler errors participants had. The lambda group (M = 17.77, SD = 28.58) was larger in the raw compared to the number of errors in the code for the iterator group (M = 8.75, SD= 12.32). In regard to individual tasks, the lambda group has a higher mean in Task 1 (M = 27.89, SD = 27.38), 2 (M = 10.16, SD = 17.16) and 3 (M = 13.55, SD = 36.78) compared to the means of the iterator group's Task 1 (M = 11.23, SD = 13.98), 2 (M = 5.08, SD = 9.15) and 3 (M = 9.78, SD = 12.81). Using an ANOVA, we found a significant difference between the groups, F(1,148) = 5.704, p = .018, η_p^2 = .039, as well as for the tasks, F(2,148) = 4.093, p = .019, η_p^2 = .055. This indicates, that 3.9% of the variance in errors can be explained by the groups the participants were in and 5.5% of the variance can be explained by differences in the tasks. Thus, we can reject H0-3.

Looking at the amount of time used to work on errors, there is a similar pattern. The lambda group's mean percentage time spent on working on fixing errors is (M = 56.37, SD = 35.54), while the iterator group's mean is (M = 44.20, SD = 31.99). The average for Task 1 for the lambda group was (M = 65.05, SD = 34.32) compared to (M = 57.03, SD = 31.34) for the iterator group. For the second task, we observed (M = 45.43, SD = 37.34) and for iterator (M = 26.14, SD = 24.88) of time on fixing errors. The amount of time spent on fixing errors goes up for both the groups compared to their second tasks with an average of (M = 57.75, SD = 33.13) for the lambda group and (M = 48.54, SD = 31.88) for the iterator group. Again using an ANOVA, we found a significant difference between the groups, F(1,148) = 5.178, p = .024, η_p^2 = .035. There was also a significant difference between the tasks, F(2,148) = 8.027, p < .001, η_p^2 = .102. Thus, we can reject H0-4.

6. DISCUSSION

In reviewing the evidence from our study, several considerations come to mind. First, it's important to note that the ANCOVA analysis does indicate a statistically significant negative impact for our main effect in regard to time, but, from looking at the graph, we suspect this is a novelty effect in our tasks that is stronger for the lambda group. Second, note that task 3 had approximately parity performance between the groups with regard to time. There are at least two plausible explanations: either the lambda group "learned" how to use them over time or there is something about task 3 that was easier for the lambda group. After watching the videos and thinking about the analysis, we think the most plausible explanation is that users had to get used to the syntax/semantics in C++, but that they did so rather quickly in regard to time. Further, given differences in ability to complete the tasks, it appears that lambdas are hard to use for the *first several years* of study when learning C++, but that experienced professionals do eventually get the hang of them, even if they do not appear to benefit from them in this context of use.

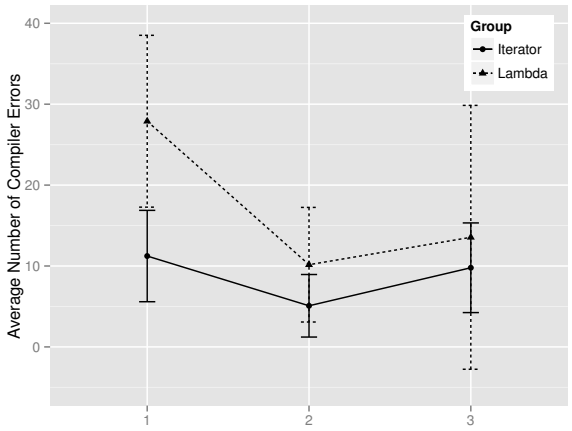

Figure 7: Compiler Errors Compared

To put these results into context with previous work, contrast this study with that of the recent randomized controlled trials on type systems mentioned earlier. First, regarding work on static versus dynamic typing, the key result is that static typing increases productivity over dynamic under a wide variety of conditions [15, 19, 37, 27, 17]. Contrast these controlled trials with that of Stefik and Siebert [51], who showed evidence that static typing had a negative impact on novices. In combination, these studies suggested there was a tradeoff in the impact of static typing based on experience level. This observation was essentially conjecture until recently, with the work of Altadmri and Brown [1], where the authors tracked approximately 37 million compiler errors made by novices. Interestingly, this work confirms independently the conjecture made by the research teams led by Stefik and Hanenberg. Namely, in Figure 6 of the Altadmri and Brown paper, it is clear that after about 9 months, the number of compiler errors experienced due to static typing diminishes with novices. In essence, we now have confirmation that such a tradeoff exists and have a better sense of when the initial negative impact begins to disappear. With lambdas compared to iterators, we found no evidence supporting a similar tradeoff with this particular feature. Namely, undergraduate students at all levels appear to have difficulty with lambdas, but we observed no benefits for professionals.

In regard to build logs, we did observe a larger number of compiler errors in the lambda group and participants spent a larger percentage of their time fixing them. We think the most likely explanation here is that the syntax of C++ lambdas, and the error messages it provides, are not very intuitive. Previous studies have

shown evidence that this is the case with novices [51], and more recent studies have shown evidence that compiler errors can impact professionals [44]. It appears that this study provides a growing body of evidence that syntax and compiler errors can impact people across the pipeline. In other words, having professional status hardly guarantees that a particular syntactic choice in a particular language will be easy to use.

7. LIMITATIONS AND FUTURE WORK

We have conducted what is, to our knowledge, the first randomized controlled trial on the use of lambda expressions. In this study, we have considered developers from a variety of groups, from freshman in college to experienced professionals working full-time. While our results give us a level of skepticism that lambdas provide a time saving benefit in a commonly considered situation seen in the academic literature and amongst developers, we would like to point out several limitations to our study and threats to validity, which we describe in the context of future work.

First, our experiment uses C++ as the language of choice for lambda expressions. We imagine some readers will consider this a limitation to our experiment, despite the fact that all experimenters must ultimately choose tangible and specific scientific conditions under which to test. We want to be clear in saying that we are not claiming generalizability in the broad sense for the same reasons that no research group can, but we are claiming that we observed evidence that C++ syntax and errors may have a negative human factors impact under the conditions of our test. Further, given that our replication packet is trivial to port to other languages, our study simplifies the process of conducting such experiments for the community by providing a first baseline that can be adapted and tested under new conditions. In other words, the reader should keep in mind that no single experiment on non-trivial ideas can tell us the whole truth, which is one reason why the broader programming language wars [49] have been so difficult to evaluate.

Second, we think one important limitation of our work is that lambdas might have different impacts on different kinds of users under different contexts of use. While we tested lambdas against iterators because we saw it so frequently mentioned, it could very well be the case that they are beneficial under other conditions. For example, one obvious case is in graphic user interface programming. Specifically, if neither lambdas, nor anonymous inner classes, are used, then the developer is forced to create a separate class and to embed logic from the user interface in it, which developers might find annoying or frustrating. On the other hand, Weiss [56] argues that lambdas are difficult to debug. It could be the case that under the domain of user interfaces, using them might simplify initial programming, but have unintended side effects (e.g., difficulty of debugging for future developers). This is a significant issue, as it is well known in the literature how time consuming and difficult debugging can be (see e.g., Ko and Myers [29] amongst others for a discussion). The issue is similar with the use of lambdas in concurrent software. The literature shows evidence [41] that various kinds of concurrent programming can make a large difference in regard to productivity, but this does not automatically mean, without evaluating exact experimental data, that lambdas would necessarily help in that context. As Tichy and others have argued, more experimentation is needed [53, 49, 30, 16].

While we carefully designed our tasks to be neutral and to use situations related to our reading of the literature, other tasks should be tested as part of a community effort toward due diligence and replication. Our study does represent a first test, so by definition our replication packet is the first one that exists, but a collection of replication packets designed by independent research teams can provide confidence that no single group can. One possible way forward here given the current data would be to conduct syntax studies using Placebo Languages [51] that have lambda syntax and to compare against the state-of-the-practice in the field. This might help "factor out" syntax and compiler errors in the same way previous work has shown how to do.

As one final consideration, it is often preferable in science to have a working theory that explains why a result occurs, but our study only dances in this direction. We often begin with a hypothesis that may eventually develop into a theory if it becomes supported by enough evidence that the community accepts it as a valid explanation of a phenomenon. Regarding human factors of programming language design, however, we are doubtful that such a scientific theory exists. Our position is supported by the recent work by Kaijanaho on the foundation of evidence in regard to human factors in programming language design [23]. Using a broad-based, objective, scoring technique as his criteria, he found only 22 papers with randomized controlled trials up until 2012 in any major journal or conference in the field. These results are striking in that they bring into doubt not just the human factors evidence at venues like ICFP, but the entire discipline as a whole.

Despite this situation, which Kaijanaho documents with extreme precision and care, we continue to see references to "theories" in the scientific literature, perhaps most commonly, in this context, the well-known Cognitive Dimensions of Notations. In a close analysis of this purported theory, acknowledging only 7 randomized controlled trials existed before its initial publication, and after evaluating the evidence standards of hundreds of papers citing it [50, 49], we harbor significant doubts that it meets the requirements for being called a scientific theory. In order for our community to make substantive progress on the deep challenges of our time, of which the programming language wars is included, we will need theories. However, we are currently in the situation where there is no evidence – by definition, this means there are no theories that are backed by evidence. We encourage movement toward traditional scientific models of research, especially where replication, testing, and evidence gathering are prioritized in peer-review over vague theories or poorly supported explanations.

8. CONCLUSION

This study is part of a larger set of experiments, increasingly showing that the design of programming languages has significant human factors considerations. Specifically, we conducted the first randomized controlled trial on the impact of lambda expressions, namely a comparison of the C++ constructs iterators and lambdas. Evidence from our study shows that those in the lambda group received no benefits in regard to time, compiler errors, or their ability to complete tasks. Further, our results showed that a user's experience level makes a significant impact on whether they can complete programming tasks like ours and also the speed at which the tasks can be completed. Given the popularity of lambdas in the literature, and the fact that design changes to programming languages cause additional costs (e.g., new textbooks, changes to software), we would encourage the broad community to run more tests to find out whether such changes are worth it.

9. ACKNOWLEDGEMENTS

This work was partially funded by grant NSF-CNS-1440878. The opinions, findings and recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funders.

References

[1] ALTADMRI, A., AND BROWN, N. C. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2015), SIGCSE '15, ACM, pp. 522–527.

[2] BENEKE, T. The javaone 2013 technical keynote. https://blogs.oracle.com/javaone/entry/the_javaone_2013_technical_keynote, 2013. [Online; accessed 24-March-2015].

[3] BRACHA, G., AND GRISWOLD, D. Strongtalk: Typechecking smalltalk in a production environment. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 1993), OOPSLA '93, ACM, pp. 215–230.

[4] CHURCH, A. A set of postulates for the foundation of logic. *Annals of mathematics 2*, 33 (1932), 346–366.

[5] CRESTANI, M., AND SPERBER, M. Experience report: Growing programming languages for beginning students. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2010), ICFP '10, ACM, pp. 229–234.

[6] CRK, I., KLUTHE, T., AND STEFIK, A. Understanding programming expertise: An empirical study of phasic brain wave changes. *ACM Trans. Comput.-Hum. Interact. 23*, 1 (Dec. 2015), 2:1–2:29.

[7] DENNY, P., LUXTON-REILLY, A., AND TEMPERO, E. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (New York, NY, USA, 2012), ITiCSE '12, ACM, pp. 75–80.

[8] DENNY, P., LUXTON-REILLY, A., TEMPERO, E., AND HENDRICKX, J. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (New York, NY, USA, 2011), ITiCSE '11, ACM, pp. 208–212.

[9] ENDRIKAT, S., HANENBERG, S., ROBBES, R., AND STEFIK, A. How do api documentation and static typing affect api usability? In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 632–642.

[10] FEIGENSPAN, J., KÄSTNER, C., LIEBIG, J., APEL, S., AND HANENBERG, S. Measuring programming experience. In *Proceedings of the 2012 IEEE 20th International Conference on Program Comprehension* (Washington, DC, USA, 2012), ICPC '12, IEEE Computer Society.

[11] FERG, S. The pros and cons of lambda. http://python.dzone.com/articles/pros-and-cons-lambda, 2012.

[12] FISCHER, L., AND HANENBERG, S. An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio. In *Proceedings of the 11th Symposium on Dynamic Languages* (New York, NY, USA, 2015), DLS 2015, ACM, pp. 154–167.

[13] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[14] HANENBERG, S. Doubts about the positive impact of static type systems on programming tasks in single developer projects - an empirical study. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings* (2010), pp. 300–303.

[15] HANENBERG, S. 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* (New York, NY, 2010), OOPSLA '10, ACM, pp. 22–35.

[16] HANENBERG, S. 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* (New York, NY, 2010), OOPSLA '10, ACM, pp. 933–946.

[17] HANENBERG, S., KLEINSCHMAGER, S., ROBBES, R., TANTER, É., AND STEFIK, A. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering 19*, 5 (2014), 1335–1382.

[18] HARVEY, B., AND MÖNIG, J. Bringing "no ceiling" to scratch: can one language serve kids and computer scientists. *Proc. Constructionism* (2010).

[19] HOPPE, M., AND HANENBERG, S. Do developers benefit from generic types?: an empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013* (2013), ACM, pp. 457–474.

[20] HÖST, M., REGNELL, B., AND WOHLIN, C. Using students as subjects-a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering 5*, 3 (2000), 201–214.

[21] JÄRVI, J., FREEMAN, J., AND CROWL, L. Lambda expressions and closures: Wording for monomorphic lambdas (revision 4). Tech. rep., Tech. Rep, 2008.

[22] JONES, S. P., BLACKWELL, A., AND BURNETT, M. A user-centred approach to functions in excel. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2003), ICFP '03, ACM, pp. 165–176.

[23] KAIJANAHO, A.-J. *Evidence-based programming language design : a philosophical and methodological exploration*. PhD thesis, University of Jyväskylä, 2015.

[24] KAIJANAHO, A.-J. The extent of empirical evidence that could inform evidence-based design of programming languages: A systematic mapping study. *Jyväskylä Licentiate Theses in Computing, University of Jyväskylä* (2015).

[25] KAPTCHUK, T. J. Intentional ignorance: A history of blind assessment and placebo controls in medicine. *Bulletin of the History of Medicine 72*, 3 (1998), 389–433.

[26] KITCHENHAM, B., PEARL BRERETON, O., BUDGEN, D., TURNER, M., BAILEY, J., AND LINKMAN, S. Systematic literature reviews in software engineering - a systematic literature review. *Inf. Softw. Technol. 51*, 1 (Jan. 2009), 7–15.

[27] KLEINSCHMAGER, S., HANENBERG, S., ROBBES, R., TANTER, É., AND STEFIK, A. Do static type systems improve the maintainability of software systems? an empirical study. In *IEEE 20th International Conference on Program Comprehension, Passau, Germany, June 11-13, 2012* (2012), ICPC'12, IEEE Computer Society, pp. 153–162.

[28] KO, A., LATOZA, T., AND BURNETT, M. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* (2013), 1–32.

[29] KO, A. J., AND MYERS, B. A. Finding causes of program output with the java whyline. In *CHI '09: Proceedings of the 27th International Conference on Human Factors in Computing Systems* (New York, NY, USA, 2009), ACM, pp. 1569–1578.

[30] MARKSTRUM, S. Staking claims: a history of programming language design claims and evidence: A positional work in progress. In *Evaluation and Usability of Programming Languages and Tools* (New York, NY, USA, 2010), PLATEAU '10, ACM, pp. 7:1–7:5.

[31] MAYER, C., HANENBERG, S., ROBBES, R., TANTER, É., AND STEFIK, A. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tucson, AZ, USA, October 21-25, 2012* (2012), OOPSLA'12, ACM, pp. 683–702.

[32] MEYEROVICH, L. A., AND RABKIN, A. S. Empirical analysis of programming language adoption. *SIGPLAN Not. 48*, 10 (Oct. 2013), 1–18.

[33] NANZ, S., AND FURIA, C. A. A comparative study of programming languages in rosetta code. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (2015), IEEE Press, pp. 778–788.

[34] PAGE, R. L. Software is discrete mathematics. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2003), ICFP '03, ACM, pp. 79–86.

[35] PARNIN, C., BIRD, C., AND MURPHY-HILL, E. R. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings* (2011), IEEE, pp. 3–12.

[36] PARNIN, C., BIRD, C., AND MURPHY-HILL, E. R. Adoption and use of java generics. *Empirical Software Engineering 18*, 6 (2013), 1047–1089.

[37] PETERSEN, P., HANENBERG, S., AND ROBBES, R. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014* (2014), ACM, pp. 212–222.

[38] PIERCE, B. C. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[39] POP, I. Experience report: Haskell as a reagent: Results and observations on the use of haskell in a python project. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2010), ICFP '10, ACM, pp. 369–374.

[40] RAY, B., POSNETT, D., FILKOV, V., AND DEVANBU, P. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 155–165.

[41] ROSSBACH, C. J., HOFMANN, O. S., AND WITCHEL, E. Is transactional programming actually easier? *SIGPLAN Not. 45*, 5 (Jan. 2010), 47–56.

[42] SALMAN, I., MISIRLI, A. T., AND JURISTO, N. Are students representatives of professionals in software engineering experiments? In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on* (2015), vol. 1, IEEE, pp. 666–676.

[43] SAMKO, V. A proposal to add lambda functions to the c++ standard. Tech. rep., Technical Report–N1958=06-0028, 2006.

[44] SEO, H., SADOWSKI, C., ELBAUM, S., AFTANDILIAN, E., AND BOWDIDGE, R. Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 724–734.

[45] SHNEIDERMAN, B. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, August 1980.

[46] SHULL, F., SINGER, J., AND SJØBERG, D. I. *Guide to Advanced Empirical Software Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[47] SIEGMUND, J., KÄSTNER, C., LIEBIG, J., APEL, S., AND HANENBERG, S. Measuring and modeling programming experience. *Empirical Software Engineering 19*, 5 (2014), 1299–1334.

[48] STEFIK, A., AND GELLENBECK, E. Empirical studies on programming language stimuli. *Software Quality Journal 19*, 1 (2011), 65–99.

[49] STEFIK, A., AND HANENBERG, S. The programming language wars: Questions and responsibilities for the programming language community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (New York, NY, USA, 2014), Onward! 2014, ACM, pp. 283–299.

[50] STEFIK, A., HANENBERG, S., MCKENNEY, M., ANDREWS, A. A., YELLANKI, S. K., AND SIEBERT, S. What is the foundation of evidence of human factors decisions in language design? an empirical study on programming language workshops. In *Proceedings of the 2014 IEEE 20th International Conference on Program Comprehension* (2014), ICPC '14, IEEE Computer Society, pp. 223–231.

[51] STEFIK, A., AND SIEBERT, S. An empirical investigation into programming language syntax. *Trans. Comput. Educ. 13*, 4 (Nov. 2013), 19:1–19:40.

[52] STROUSTRUP, B. C++11 - the new iso c++ standard. http://www.stroustrup.com/C++11FAQ.html#lambda, 2014.

[53] TICHY, W. F. Should computer scientists experiment more? *IEEE Computer 31* (1998), 32–40.

[54] U.S. DEPARTMENT OF EDUCATION INSTITUTE OF EDUCATION SCIENCES. *What Works Clearinghouse Procedures and Standards Handbook*, 2.1 ed. U.S. Department of Education, 2010.

[55] VOGT, W. P. *Quantitative Research Methods for Professionals in Education and Other Fields*, 1st ed. Allyn and Bacon, Columbus, OH, 2006.

[56] WEISS, T. The dark side of lambda expressions in java 8. http://blog.takipi.com/the-dark-side-of-lambda-expressions-in-java-8/, 2014.

[57] WILLCOCK, J., JÄRVI, J., GREGOR, D., STROUSTRUP, B., AND LUMSDAINE, A. Lambda functions and closures for c++. Tech. Rep. N1968=06-0038, ISO/IEC JTC 1, Information technology, Subcommitee SC 22, Programming Language C++, February 2006.

[58] WOHLIN, C., RUNESON, P., HÖST, M., OHLSSON, M., REGNELL, B., AND WESSLÉN, A. *Experimentation in Software Engineering*. Computer Science. Springer, 2012.

[59] ZHANG, H., BABAR, M. A., AND TELL, P. Identifying relevant studies in software engineering. *Inf. Softw. Technol. 53*, 6 (June 2011), 625–637.