Characterizing the Pedagogical Benefits of Adaptive Feedback for Compilation Errors by Novice Programmers

Umair Z. Ahmed*
umair@comp.nus.edu.sg
National University of Singapore, Singapore

Nisheeth Srivastava
nsrivast@cse.iitk.ac.in
IIT Kanpur, India

Renuka Sindhgatta
renuka.sr@qut.edu.au
Queensland University of Technology, Australia

Amey Karkare
karkare@cse.iitk.ac.in
IIT Kanpur, India

ABSTRACT

Can automated adaptive feedback for correcting erroneous programs help novice programmers learn to code better? In a large-scale experiment, we compare student performance when tutored by human tutors, and when receiving automated adaptive feedback. The automated feedback was designed using one of two well-known instructional principles: (i) presenting the correct solution for the immediate problem, or (ii) presenting generated examples or analogies that guide towards the correct solution. We report empirical results from a large-scale (N = 480, 10,000+ person hour) experiment assessing the efficacy of these automated compilation-error feedback tools. Using the survival analysis on error rates of students measured over seven weeks, we found that automated feedback allows students to resolve errors in their code more efficiently than students receiving manual feedback. However, we also found that this advantage is primarily logistical and not conceptual; the performance benefit seen during lab assignments disappeared during exams wherein feedback of any kind was withdrawn. We further found that the performance advantage of automated feedback over human tutors increases with problem complexity, and that feedback via example and specific repair have distinct, non-overlapping relative advantages for different categories of programming errors. Our results offer a clear and granular delimitation of the pedagogical benefits of automated feedback in teaching programming to novices.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; • **Applied computing** → **Computer-assisted instruction**.

KEYWORDS

Intelligent Tutoring Systems, Introductory Programming, Compilation Errors, Program Repair, Example Generation, User Study

*This work was carried out by the author at IIT Kanpur.

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

ACM ISBN 978-1-4503-7124-7/20/05…$15.00
https://doi.org/10.1145/3377814.3381703

ACM Reference Format:
Umair Z. Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the Pedagogical Benefits of Adaptive Feedback for Compilation Errors by Novice Programmers. In *Software Engineering Education and Training (ICSE-SEET'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377814.3381703

1 INTRODUCTION

CS–1, the Introduction to Programming course, is one of the most popular courses in college with steadily increasing student enrollment [47]. To accommodate teaching programming at this ever-increasing scale, many software tools strive to repair logical errors in students' incorrect code submissions automatically, thus providing feedback by way of repair [15, 21, 36, 40, 43, 44]. These tools learn from the repeated mistakes made by historical student submissions and eliminate manual effort in the current offering, making it possible to scale-up to massive class sizes.

Within CS pedagogy, there is an active interest in aiding students struggling with compile-time errors (also known as compiler or compilation errors). These errors are caught and reported by the compiler before the program executes. They typically occur due to incorrect use of language syntax, or on importing incorrect files/libraries. These are, in fact, the types of errors most likely to be made by novitiates into programming and, therefore, are of prime interest for instruction support for introductory programming courses.

Compiler errors are considered trivial to locate and fix by expert programmers since the compiler reports the line-number in code where the error occurs and a short message explaining the error it encountered [42]. Likely for this very reason, text in compiler error messages in most software tools are targeted towards experienced programmers and appear cryptic to beginners. For example, consider the incorrect code written by a student in Figure 1. The first incorrect code line #5 is missing an asterisk (`*`) operator between two integer expressions; a common mistake made by novice programmers. Clang [25], a popular compiler for C language, reports the cryptic error message of "`called object type int is not a function`" corresponding to the incorrect line, which assumes the knowledge of advanced concepts such as function calls by the programmer to fix the error.

Studies show that students who have just begun to learn to program often struggle with compilation errors [13, 27]. Recognizing the need to support students, there has been significant interest recently in automated compilation error feedback generation, in the form of repair [1, 7, 17, 18, 34], or examples of repair undertaken

by other students [2, 29]. While such automated program feedback efforts are conceptually laudable, it is not yet empirically evident whether they assist students in learning.

To address the limitation in literature surrounding automated feedback systems for learning programming, we conducted a large sample (N=480), randomized and partially controlled experiment, replacing human tutors with automated adaptive feedback systems for the first seven weeks of the introductory programming course at IIT Kanpur, a large public university in India. This user-study data is released in public domain to aid further research[1]. We measured programming performance improvement using multiple metrics longitudinally. We found significant gains in error-resolution during programming assignments as a consequence of this intervention over human-assisted baselines, with the extent of the improvement more significant for difficult errors than easier ones. We also found that the performance improvements seen during feedback tool usage disappeared once the tools were withdrawn during course exams, suggesting that the advantage conferred by automated feedback tools is primarily logistical rather than conceptual. The rest of this paper describes our methods and results in more detail.

2 RELATED WORK

The ultimate goal of our work is to understand how to communicate feedback about compiler errors to students efficiently. One may argue why we do not just map compiler error numbers to simplified or enhanced text messages instead of relying on complicated generative tools as we do. Presumably, it is possible to do such a translation in ways that optimize students' learning. Several studies [5, 6] have indeed proposed methods and guidelines to enhance error message text to suit novice programmers. However, empirical testing has revealed mixed learning outcomes for such approaches. Studies from multiple labs have documented null results from the use of such static text message enhancements [12, 29, 31], while some have claimed positive outcomes [4, 6].

Becker had a control and a test group of 100 students each perform Java-based lab exercises over a four week period [4]. The test group received enhanced text messages explaining compiler errors. Although Becker claims a significant improvement in number of errors per student for a subset of the error types studied, this claim is problematic because the analysis does not control for multiple comparisons, and a simple Bonferroni correction to their reported p-value renders the result non-significant.

Becker, Goslin, & Glanville conducted an experiment where 43 test and 43 control participants tried to debug a 107-line piece of code in which 24 syntax errors had been synthetically introduced [6]. The authors report that the test group exposed to enhanced compiler messages corrected significantly more errors than the control group. The small sample size, and students correcting synthetically introduced errors make it difficult to compare this finding with other studies that report the effect of feedback on self-generated errors.

Nienaltowski et al [29] had 67 students across two separate universities answer a questionnaire, in which they tried to identify the error in nine different pieces of code given enhanced and regular compiler messages. They found no difference between the proportion of correct responses for problems with and without enhanced compiler information.

Denny et al [12] split 83 students into test and control groups that completed coding assignments over two weeks with and without enhanced feedback for compilation errors. They found no significant difference in the number of errors, or the number of attempts needed to reach a successfully compiling code.

The previously mentioned two studies [12, 29] were conducted on small sample size and had a distinct evaluation of student learning. However, a high-sample study from Pettit, Homer & Gee [31] found the same outcome. Pettit et al. conducted *post hoc* analyses from eight semesters' worth of accumulated data (~ 30k code submissions) from an automated assessment tool, with enhanced feedback provided randomly to some students for some problems in four of the semesters. They found that enhanced compiler messages did not reduce the likelihood of successive compiler errors, the occurrence frequency of compiler errors within semesters, or students' time to completion for individual assignments.

Multiple studies have shown that the students find examples of similar errors and corresponding fixes helpful in fixing their errors [30, 41, 45], and also help in problem completion and program understanding [24, 32]. However, in another study [33], the authors noted that showing novices example code confuse them more as they could not dissociate the example code from their own code and were looking for the example code in their own submissions. Thus, on balance, the existing empirical evidence in the literature suggests that transforming compiler messages to suit students or providing them with examples does not reflect in better code correction except in particular circumstances.

Given this state of affairs, it becomes all the more interesting to ask if providing more dynamic feedback using intelligent tutoring systems could yield better outcomes. D'Antoni et al. [9] developed and deployed an Intelligent Tutoring System to provide automated feedback for teaching automata construction to students. Three different types of feedback are provided when student designs an incorrect automaton - binary (correct/incorrect), conceptual hints which propose strategies for fixing it, and counter-example hints which suggests specific string inputs which the students' automaton should or shout not accept. They report that the students prefer and perform better on conceptual and counter-example based feedback over binary feedback (N=377, 1-week session).

Given the commonality of CS1 courses and growing enrollment on Massive Online Open Courses (MOOC), a variety of feedback systems have been published to aid novice programmers at large scale, but without corresponding user studies to measure their effectiveness. CodeOpticon [16] proposes a user-interface that allows tutors to manually intervene in live coding sessions, whenever they observe students making mistakes (N=226, 30 minute session). OverCode [14] semi-automatically clusters the submissions, which the instructors can visualize later to provide personalized feedback per cluster (N=12, 60 minute session). HelpMeOut [19] proposes a social recommender system where erroneous-repaired program examples are suggested from a database of compilation errors (36 person-hour). This database is maintained by the students, requiring manual analysis of the code and error messages to provide feedback. The authors indicate that scaling the system or porting

[1] https://github.com/umairzahmed/seet2020

Figure 1: Prutor [10] deployed for programming assignments, with repair and example feedback tools integrated. On encountering compilation errors, only one of the feedback types (left panel: top or bottom) is shown in *tutor* pane, depending on the students' group assignment. Clang [25], a popular compiler for C language, reports the following errors – Line-5: *called object type 'int' is not a function*, Line-7: *expected ';' in 'for' statement specifier*, Line-8: *expected ')'*.

it to a different programming language would require either additional manual efforts or adoption of AI-based methods along the lines of the tool we have used.

Yi et al. [46] used state-of-the-art automated semantic repair tools, designed for expert programmers, to provide feedback on student assignments. They claim that these tools can generate only partial repairs on the majority of the student assignments, and handing out these partial repairs to students did not seem to help them resolve logical bugs in other students' code (N=263, 15 minute session). It is not immediately apparent that these findings are contradictory to our own, since we provide relevant repair (or example) feedback on students' own code, using tools specialized in fixing compilation errors of introductory programming. Further work is likely needed to characterize the nature of this disparity.

Thus, in summary, while several related automated feedback tools have been proposed in recent literature, a large-scale empirical evaluation of their efficacy has not been possible due to their relative novelty and technical complexity.

3 FEEDBACK STRATEGIES FOR COMPILATION ERRORS

Multiple instructional principles have been evaluated for efficient learning, based on the complexity of the educational content [23]. Research on tutoring systems for programming language indicate significant learning benefits when students get some form of immediate feedback [8]. In the literature on education research, two basic principles for providing instructional feedback have been presented: (i) teaching by correction [26] wherein students learn the correct response framed in the temporal context of the erroneous response they had originally made, and (ii) teaching by examples [38] wherein students learn to solve problems by analogy [35].

In our experimental setup, we have incorporated both types of instructional feedback which can be used to provide immediate feedback and have the ability to scale to a large number of students. In the context of learning to code, *repair* feedback presents the correction of the specific error made by the student. Further, *example* feedback presents the student with correct examples of code with lines similar to the mistake made by student and its desired correction. We are specifically interested in identifying which instructional principle is more effective in helping novice programmers fix compilation errors. We describe how both these feedback strategies are implemented in the tutoring system.

3.1 Learning from correction

Discriminative feedback is provided to students having compilation errors in the form of correct code, known as repair. We used TRACER[2] [1] tool to generate the code repair. Given an erroneous program, TRACER performs the following steps: (i) locates the error lines, (ii) abstracts the variables and literals with their types, (iii) predicts the relevant repair using a deep learning (Recurrent Neural Network) model, and (iv) converts back the repair from abstract representation to concrete code.

TRACER is the current state-of-the-art compilation error repair tool. On the DeepFix [18] test dataset, which consists of ~ 7,000 incorrect student programs with multiple errors, TRACER boasts a

[2]https://github.com/umairzahmed/tracer

repair accuracy of 43.97%. In comparison, DeepFix [18] and RLAssist [17] achieved a repair accuracy of 23.3% and 26.6%, respectively, on the same test dataset. TRACER's feedback is not only accurate, but also relevant to the students - its top recommended fix is the exact same as students' 68% of the times (prior repair tools do not report on relevance metric). Figure 1 lists the top repair generated by TRACER on the incorrect code, in the top-left *tutor* pane.

3.2 Learning from examples

Generative feedback is provided in the form of examples of incorrect-correct code pairs that have the same compilation errors and require the same repairs as the given erroneous code. We used TEGCER[3] [2] tool to generate the example based feedback. Given an erroneous program, TEGCER (i) locates the error lines (ii) abstracts each error line (iii) predicts the class of error-repair which the error line belongs to, and (iv) suggests back the top frequent training examples observed for each predicted class.

To the best of our knowledge, TEGCER is the only tool available in the literature that can automatically generate example based feedback for compilation errors. TEGCER identified 200+ classes of error-repair that capture the mistake made by a student and the desired repair. A repair comprises of insertion or deletion of abstract code tokens, generated by observing compilation errors and repairs made by actual students in previous offerings of the course. It uses a dense neural network to achieve 87% accuracy in predicting a single correct class, from the 200+ error-repair classes. Figure 1 lists the top repair example generated by TEGCER on the incorrect code, in the bottom-left *tutor* pane.

We trained both these tools on the same dataset of 15,000+ incorrect code attempts by 400+ students during 2015-2016-II (spring semester) Introductory to C Programming (CS1) course at IIT Kanpur. Both the tools typically generate feedback within 1 second.

4 EXPERIMENTAL SETUP

We conducted our study as a randomized experiment at IIT Kanpur, during 2017–2018–II (spring-semester) course offering of CS1. The course is offered at this university across two semesters to all first-year undergraduates. One of the learning objectives our tool addresses is the "understanding of programming language syntax and semantics by the example of C language to construct programs and solve problems." A significant component of this course is weekly programming assignments where students attempt four programming questions of varied difficulty every week in the university computer lab. These labs are conducted from 14:00 to 17:15 hours on one particular day of the week. The students are not allowed to discuss among themselves or use the internet to solve these questions. Only hand-written notes are allowed for reference. In the regular running of the course, 40+ tutors, who are CS post-graduate students at the same university, remain physically present during these labs to help guide the undergraduates.

4.1 Online Programming Editor

At IIT Kanpur, the CS-1 course assignments are developed and submitted by students on Prutor [10], a web based Integrated Development Environment (IDE) targeted for education. Prutor supports multiple features, including account management, assignment management, display of grading rubric, and compilation/execution of students programs. Specifically appropriate for our purpose, Prutor maintains a complete record of compilation requests by each student, including a unique (anonymized) ID per student assignment, students' code, compilation errors triggered (if any), and the timestamp.

[3]https://github.com/umairzahmed/tegcer

4.2 Experiment design

The course during 2017–2018–II had 480 students, and they were grouped into two experimental groups randomly using odd-even roll numbering. During the weekly lab conducted from 14:00 to 17:15 hours on any one particular day, 242 of these students received feedback from the repair-hint tool, while the remaining 238 students received feedback from the example-hint tool (the minor imbalance in group size is due to course drop by students after group allotment). Both the experimental groups covered the programming assignment on the same topic for seven weeks, as shown in Table 1. During this 10,000+ person hour practice labs (7 labs * 3 hours * 480 students), our 40+ human tutors were asked not to help students unless they were unable to resolve errors using the automated feedback for more than 5 minutes.

The user-interface of Prutor IDE that the students worked with contained: (i) a main pane wherein students entered their code, (ii) a pane below where standard C compiler errors and warnings were displayed, and (iii) a *tutor* pane on the left that showed feedback only from the feedback mode corresponding to the student's treatment group assignment i.e. either repair hints for the student's code, or correct example codes similar to the student's buggy code.

Whenever a compilation error occurred, either a single fix (per line) or the single best example was shown to students belonging to repair or example groups, respectively. Students who received the example hints could choose to generate the next best examples, up to a maximum of 10 per line, by clicking on a UI button (Figure 1).

4.3 Baseline groups

Data from two immediately prior offerings of the same course at IIT Kanpur was used to assess baseline performance, as shown in Table 2. Students in these two offerings used the same browser-based IDE to complete their programming tasks, assisted by human tutors for debugging their code for both compiler and logical errors. Course offerings remained consistent across our baseline and treatment groups (Table 1). However, while students in both the treatment groups worked on the exact sample lab assignment problems, the baseline group's students worked on problems different from these. The difficulty level of these problems was similar since they adhered to the same learning objectives during the entire offering of the course.

After seven labs (or weeks) of programming practice, students appeared for a *mid-semester lab examination* where they submitted their code on the same browser-based editor. For this exam event, neither feedback-tools nor human tutors provided any help on compilation errors. Students wrote their code under the invigilation of tutors, where discussion with other students or usage of any reference materials was disallowed.

Lab #	Topic
1	Input/Output
2	Conditionals
3	Iterations
4	Nested Iterations
5	Functions
6	Arrays
7	Matrices

Table 1: Weekly labs

Semester	Feedback Type	#Students	Compilations		Time (sec)		#Attempt	
			#Success	#Failure	AVG	STDEV	AVG	STDEV
2016–2017–II	Manual	439	155,135	13,454	85	137	2.11	1.95
2017–2018–I	Manual	453	161,326	15,026	103	155	2.20	2.10
2017–2018–II	Repair	242	99,445	7,306	75	123	1.88	1.69
2017–2018–II	Example	238	97,763	8,624	78	132	1.99	1.86

Table 2: Total # successful and failure compilation-requests made by students during the 7-week labs across various offerings, along with the time taken and the number of attempts to fix the failures. AVG denotes average, and STDEV denotes standard deviation.

5 RESULTS

In this section, we measure the utility of automated feedback tools using existing metrics [19, 31], and then propose two new measurements that provide additional clarity about the nature of value offered by these tools to novice programmers.

5.1 Time-Taken and Repair Attempts

RQ1: *Do students with access to feedback tools resolve errors faster?*

Following the metrics proposed in Pettit et al., we track the time taken to fix programs and the number of repair attempts as key metrics for assessing the efficacy of feedback [31]. From the logs recorded by Prutor, our web-browser based IDE, we can calculate the time taken by students to fix their compilation errors. We track the time elapsed from the first occurrence of a compiler error to the next successful compilation made by the student. Similarly, we also extract the number of attempts it took to fix this error, measured as the number of unsuccessful compilation requests made by the student before finally resolving all compiler errors.

For example, consider a student's code which ran into compilation errors E_1 and E_2, i.e $E_1 \cup E_2$ at 14:10. Further, let us assume that the student made 3 different compilation requests after this, which resulted in errors $E_1 \cup E_2$ at 14:11, E_1 at 14:12, and finally $\varnothing$ (a successful compilation) at 14:13. Then we conclude that the student took 180 seconds and 3 attempts to resolve the compilation error, which occurred at 14:10. In other words, the intermediate compiler errors are accounted for by the #attempts, and their individual time-stamps are ignored.

Table 2 summarizes cohort level statistics for our control and experimental groups[4]. Students in 2016-2017-II and 2017-2018-I offerings of the course took 85 and 103 seconds on average, respectively, to resolve compilation errors with manual help from tutors when they were stuck. In comparison, a significantly lower time, 75 and 78 seconds on average, was required by our experimental group of students (crediting the 2017-2018-II offering of the course) having access to immediate feedback from TRACER [1] and TEGCER [2], respectively. Similarly, the average number of attempts required by our experimental group is significantly lower as compared to that of the baseline semesters.

The cohort level statistics are consistent with the hypothesis that automated feedback tools allow students to fix compilation errors more efficiently - using lesser time and taking fewer attempts. Pairwise t-tests evaluating differences between the test and control cohorts for both time-taken and #attempts are statistically significant at $p < 10^{-5}$ after Bonferroni correction to account for multiple comparisons. While the differences point in the right direction, the size of the effects range from 0.05 to 0.19 for time-taken, and 0.06 to 0.16 for #attempts. Thus, these conventional cohort-level analyses, by virtue of the large sample size of our study, reliably demonstrate a small improvement in resolving compiler errors for automated feedback tools when compared to performance while assisted by human tutors.

[4]We filtered out attempts that took longer than 15 minutes to resolve from our analysis, as these constituted outliers that likely reflected students not spending time on the task or being distracted by some other activity.

5.2 Number of Errors over Time

RQ2: *Do students with access to feedback tools make fewer errors over time?*

Following Hartmann et al., we also use the number of errors made over time as a metric for assessing feedback efficacy [19]. In the top row of Fig 2, we report the number of errors that students make in the 2016-2017-II, 2017-2018-I course offerings (without feedback) and the 2017-2018-II course offerings (with repair and example feedback tools deployed). Each sub-plot focuses on one of the seven weekly labs, held between 14:00 to 17:15 hour (x-axis). The topic of these labs are mentioned in Table 1.

Each point in the seven subplots in the top row of Fig 2 refers to the number of compilation errors (y-axis) which students made for that particular tool/semester, in a 30 minute window at a specific time (x-axis) during lab. These #errors are normalized over #Lines-Of-Code (LOC) written by students in the same window, since the questions and total #students vary across offerings.

For example, consider the Lab-01 of 2016-2017-II course offering, where students made a total of 9, 217 compilation requests between 14:00 to 14:30, out of which 1, 523 programs had compiler errors. The programs that failed to compile had a total of 12, 930 lines of code, which gives us 0.118 #normalized-errors (1523/12930). This is represented by the left-most purple dot in Fig 2, where x-axis=14 and y-axis=11.8. The second purple dot at y-axis=10.5 represents the #normalized-errors for the time-frame 14:15 to 14:45. The rest of the plot proceeds similarly.

From Fig 2, it can be observed that students tend to make more #errors at the beginning of the lab, which steadily decreases until the end of the lab. There is also a natural endogenous decline in errors across labs, where students eventually reach an average of 2-4 errors per 100 LOC, through practice and tutor assistance. Not only does this trend hold in our control group sample during 2017-2018-II, but students seem to be reaching the stable 2-4 errors per 100 LOC state faster, suggesting that our feedback tools are adequately replacing human tutor assistance in previous offerings.

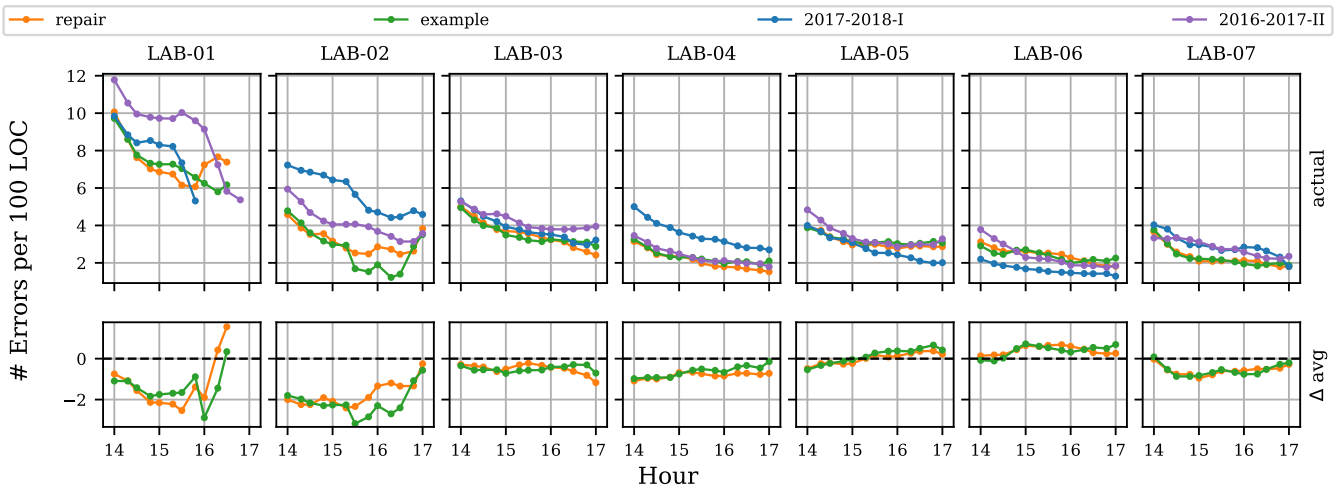


Figure 2: Plot depicting the #errors per 100 Lines of Code (y-axis) that students make over time (x-axis) during weekly lab event scheduled between 14:00 to 17:15 hours. Each point represents a sliding window of 30 minutes, calculated every 15 minutes. The bottom row of plots depict the difference between #errors made by students with access to feedback tools and the average #errors of both previous offerings.

The seven plots in the bottom row of Fig 2 depict the difference between the number of errors made by students with access to feedback tools and the average number of errors of both the previous offerings. For example, for Lab-01 plot at 14:00 hour, the number of normalized-errors per 100 lines of code is 10.0, 9.7, 9.8, and 11.8 for repair, example, 2017-2018-I and 2016-2017-II groups respectively. That is, the average number of normalized-errors for the baseline group is 10.8, with the average difference between repair and baseline being $10.0-10.8 = -0.8$ and $9.7-10.8 = -1.1$ between example and the baseline group. The same is depicted as orange and green point (resp) on the bottom plot for Lab-01, x-value = 14. Generally speaking, negative values of the Y-axis for the bottom row graphs indicate circumstances where the experimental groups appear to have an advantage over the human-tutored control groups in terms of the number of errors manifested per 100 lines of code.

From these plots, a strong protective effect is seen for automated feedback tools at the outset of the course. During Lab-1 and Lab-2, students with feedback tools make an average of 1-2 fewer errors per 100 lines of code as compared to human-tutored controls. This difference is no longer prominent beyond the second week, with automated feedback groups performing statistically identical to the human-tutored controls thereafter.

5.3 Error Survival Probability for Time-Taken

RQ3: *Do students with access to feedback tools have a lesser probability of their errors surviving on spending equivalent time?*

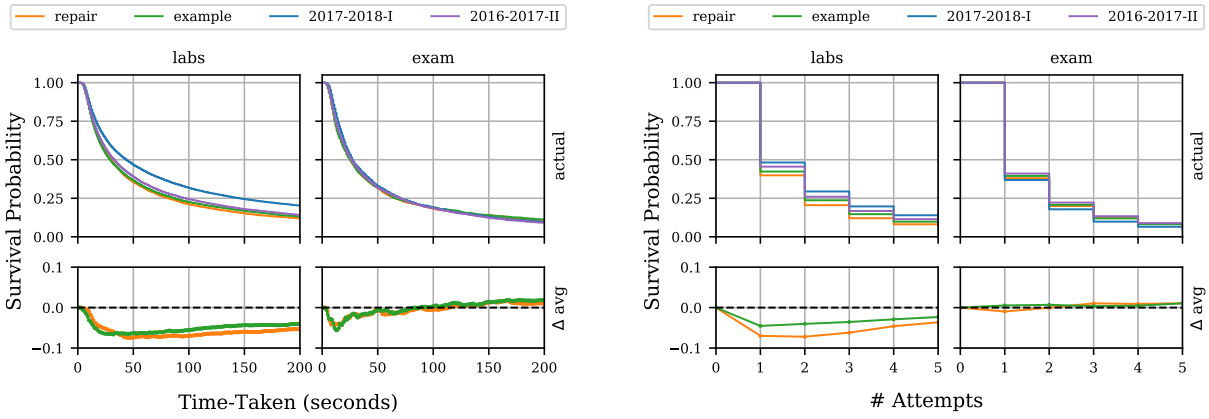
Feedback efficacy can be measured using summary statistics, such as the mean number of errors students make over time [19] and the average time-taken by students to fix errors [46]. In isolation, however, the validity of these metrics faces challenges. In particular, both reductions in errors and time taken are likely to be confounded with problem difficulty. If compiler errors were, for instance, to be caused by inattention, then we'd see fewer errors over time as students get accustomed to programming faster. The time taken to resolve errors may also drop as a consequence of more practice with the interface. Thus, a more granular view of the data, looking at the effect of feedback tools on problems that are both easy and hard to resolve, is needed to remove these potential confounds.

Therefore, complementing summary statistics of cohort-level performance, we use survival analysis for a richer representation and analysis of the tens of thousands of compiler errors that students made in our experiment. We estimate an error survival function, which models the probability of an error surviving beyond a specified time. Survival functions are commonly used in the areas of system reliability, social sciences, and medical research [28].

Survival probability function, denoted S, is formally defined as $S(t) = Pr(T > t)$, where Pr stands for probability, T is a random variable denoting the time of "death" of an event, and t is a specified time. We used the popular *Kaplan-Meier* estimator [22] from *lifelines* [11] Python package to model this survival function for our data.

For example, consider a hypothetical scenario where students make a total of 100 compilation errors during a lab. 30 of these errors are resolved at 25^{th} second, and the remaining 70 errors take exactly 50 seconds to resolve. Then, the survival probability at time 0 second, $S(t = 0) = 1.00$ since all errors persist when the student has spent no time to fix the error. $S(t = 25) = 0.70$ since 30 of the errors got resolved at 25^{th} second, and hence 70 errors survived after 25 seconds. Finally, $S(t = 50) = 0.00$ since, in our small hypothetical dataset, all observed errors got resolved within 50 seconds.

Fig 3a plots the survival function for time taken data for all errors from all students in the experiment, where survival probabilities (y-axis) are shown against time-taken (x-axis) to resolve compiler errors in different control groups. The top-left plot depicts the probability of an error surviving beyond certain time during the 7-week *labs*. The survival probability reaches 0.5 for repair, example,



(a) Error survival probability against Time-Taken

(b) Error survival probability against #Attempts

Figure 3: Error survival-probability (y-axis) plots against (a) time-taken and (b) #attempts (x-axis) to resolve compiler errors, across various control groups. For both figures, the left-side plot depicts that when feedback tools are active during *labs* event, the survival probability of time and #attempts required to resolve compiler errors is slightly lower for repair (orange-line) and example (green-line) feedback tools. When the feedback tools were withdrawn during *exam* event, no side effect was observed.

17-18-I and 16-17-II groups at time 29.5, 28.7, 43.1 and 32.3 seconds respectively. In other words, students of these particular cohorts require at least this amount of time to resolve 50% of the compiler errors. The top-right plot depicts the survival probability of an error during the *exam* event, when no feedback (tool or human) was provided to students.

The bottom row graphs represent the difference in survival probabilities between repair/example (feedback) group and the average of previous semesters (baseline). From the bottom-left plot, it can be observed that the feedback group of students have lower error survival probabilities than the baseline, averaged across all labs. In other words, as compared to the students in the baseline group, the students with access to feedback tools have a lesser probability of their errors surviving after spending an equivalent amount of time to resolve them. To interpret the graphical result concretely, for example, the repair group students have a 7.5% additional chance of resolving a compiler error after having spent 50 seconds on the problem, compared to human tutored students. We had expected *a priori* that feedback using examples would require more time on task for the student to adapt an example feedback, than a pointed repair suggestion. However, this analysis reveals that, for our domain, feedback using examples as time-effective as feedback using specific repair suggestions.

In the bottom-right plot, we observe that all groups have a similar survival probability (y-axis) for any given time-taken (x-axis) during the *exam* event. It is reassuring that within-problem performances of the experiment and control group students are identical, once the feedback tools are removed, indicating that the groups are ability-matched to a considerable degree.

It is revealing, though from a pedagogical perspective perhaps disappointing, that the advantages of automated feedback appear to be primarily logistical. Students who received automated feedback for seven weeks seem to be no better at diagnosing and repairing compilation errors on their own than the human tutored students. The advantage seen during lab events, therefore, does not appear to correspond with an improved understanding of programming. The improvement in performance during labs appears to correspond with effective delivery of repair instructions and suggestions via automated feedback, and not with a conceptual improvement.

5.4 Error Survival Probability for #Attempts

RQ4: *Do students with access to feedback tools have a lesser probability of their errors surviving after an equivalent number of attempts?*

The top-left graph in Fig 3b depicts the survival probabilities (y-axis) against the #compilation-attempts to resolve the error (x-axis) by students from various groups, during the 7-week *lab* event. For the repair, example, 2017-2018-I and 2016-2017-II groups, the survival probability of an error after one compilation attempt is 0.4, 0.42, 0.48, and 0.45 respectively. In other words, the repair (resp example) feedback group has 6.5% (resp 4.5%) higher chance of resolving an error in a single attempt, as compared to the average survival probability of students in the baseline group. Thus, Fig 3b tells a similar story as Fig 3a, with the #attempts at repair per problem as the unit of measurement instead of time-taken. This reinforces the benefit of the tool as we have consilient conclusions from two different metrics. Moreover, as with the time taken analysis, no significant change is observed in survival probability of different groups during the *exam* event, when feedback is disallowed.

5.5 Survival Probability for Specific Error Types

RQ5: *What are some of the interesting errors, where one group performs better over the others?*

Based on the earlier analysis, it is apparent that automated feedback tools do help students resolve compilation errors. We wanted to analyze the specific types of errors further wherein our feedback tools help the most (or least) as opposed to the baseline. Such findings could potentially assist in tailoring feedback type to error category in adaptive tutoring systems. The large sample size and

Error ID	Message	⬚1	⬚2
E_3	Use of undeclared identifier ⬚1	sum	
E_5	Expected identifier or ⬚1	(	
E_6	Extraneous closing brace ⬚1	}	
E_7	Expected ⬚1 in ⬚2 statement	;	for

Table 3: Few of the top frequent compiler errors

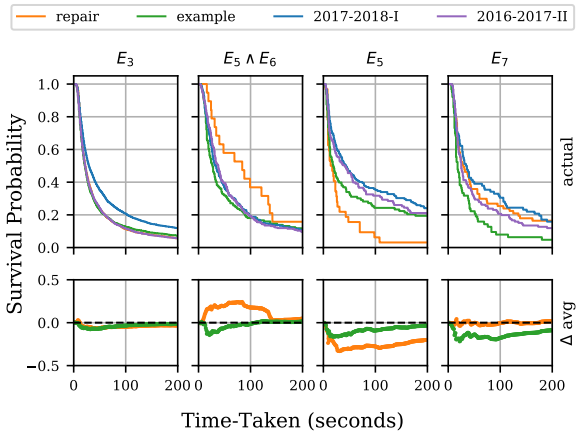


Figure 4: Error survival-probability (y-axis) plots against time-taken to resolve compiler errors (x-axis), for specific errors which display large divergence.

longitudinal design of our study permit us to answer questions about individual error categories with statistical confidence.

To this end, we plot the error survival probability against time-taken for each error-group separately. A program can have one or more compilation errors, a collection of which is called error-group [2]. Table 3 lists few of the top frequently observed compilation errors during our course offering, where the error messages are generalized by replacing program specific tokens with a generic placeholder ⬚. For example, the code in Fig 5a belongs to the compilation error-group $\{E_5 \wedge E_6\}$ since the compiler reported error messages match our generalized errors E_5 and E_6 in Table 3.

Fig 4 shows the survival probability graphs for one the most frequent error-group $\{E_3\}$, and for three error-groups ($\{E_5 \wedge E_6\}$, $\{E_5\}$, $\{E_7\}$) which demonstrate some of the largest divergence between the feedback and baseline groups. The error-group {E_3: *use of undeclared identifier* ⬚ } is typically triggered when students use a variable without declaring it first. The students can resolve this comfortably with or without feedback (for time-taken $t = 50$ seconds, the survival probability is 0.23 for almost all groups).

From Fig 4, students with repair tool feedback show higher survival probability for error-group $\{E_5 \wedge E_6\}$. This error-group is typically encountered when students inadvertently add extra closing brace "}". Fig 5a shows a simple code that triggers these errors, along with the compiler message and feedback from tools. One of the major drawbacks of both the repair and example tool is that they focus only on the compiler reported line-number. Hence, they are unable to pinpoint the correct fix: deletion of spurious "}" on line #5. Instead, both incorrectly suggest deleting the brace on line #8. This seems to affect repair group students adversely.

For the $\{E_5\}$ error-group in Fig 4, both feedback tools demonstrate faster error resolution. At time t=50 seconds, the error survival probability is 0.15 for repair tool as compared with 0.46 average probability for baseline semesters. That is, there is a 31% lower chance for $\{E_5\}$ errors to survive for the repair group. Fig 5b shows a sample code which encounters this error. The code has a stray comma on line #4, but the compiler message seems cryptic while suggesting to either add another variable after the comma or to delete it. The top-2 examples proposed suggest both these scenarios, while the repair tool suggests the deletion of this comma.

$\{E_7\}$ is another interesting error-group, where the example control group demonstrates the lowest survival probability, performing considerably better than the repair group and baseline. Fig 5c lists a simple example code for this error-group, where the student is confused about for-loop syntax. While both the compiler message and the repair feedback suggest the replacement of comma "," with a semi-colon ";" after loop initialization, the example feedback tool suggests this using similar examples, along with a variety of other ways to write a for-loop. The top-2 examples demonstrate the 0-to-(N-1) and 1-to-N iteration. On requesting further examples, the example tool suggests different ways of writing loops with empty-initialization, empty-condition or empty-increment mode, which is perhaps helping students master the for-loop syntax better, as compared to the other groups.

A general hypothesis congruent with these case examples is that feedback by example appears to work better for compiler errors by commission, viz. errors generated because of ignorance of the correct syntax, while feedback by specific repair suggestions appears to be better for compiler errors by omission, viz. situations where the student already knows the correct syntax, but has inadvertently made a mistake. Further research is needed to concretely test this hypothesis both in this specific context and more generally. In its general form, this hypothesis could be of considerable interest to research in natural and artificial language learning, where the difference between learning from positive examples and corrective feedback is extremely salient [37].

5.6 Potential Performance Improvement

RQ6: *What is the improvement provided by feedback tools, across errors of varying hardness?*

The survival analysis of our data quantifies the extent of improvement for the average compilation for the average student at between 5-10% over the controlled baselines. But this average picture glosses over the heterogeneity in problem hardness. Some compilation errors are harder to find and debug than others. It matters because any form of feedback, even a quick self-check, should prove adequate in resolving simple errors, e.g. a missing ';' at the end of a statement. The true test for a feedback mechanism lies in its ability to assist programmers in resolving complex errors.

To view the overall picture of performance improvement across the different error-groups, we developed a new metric based on the Area-Under-Curve (AUC) of error survival probability curves. The basic intuition here was that errors that are intrinsically harder to

```
1 #include<stdio.h>
2 int main(){
3  printf("Hello");
4  }
5  return 0;
6 }
```

		Compiler Message
5	E_5	expected identifier or "("
6	E_6	extra closing brace "}"

	Repair Feedback
6	"Delete this line"

	Example Feedback
6	"Delete this line"
6	// }

(a) Error-Group $\{E_5 \wedge E_6\}$

```
1 #include<stdio.h>
2 int main(){
3  int x,y,;
4
5  return 0;
6 }
```

		Compiler Message
3	E_5	expected identifier or "("

	Repair Feedback
3	int x,y;

	Example Feedback
3	int n;
3	float a, b;

(b) Error-Group $\{E_5\}$

```
1 #include<stdio.h>
2 int main(){
3  for(int j=0, j<10; j++)
4   printf("%d", j);
5  return 0;
6 }
```

		Compiler Message
3	E_7	expected ";" in "for" statement

	Repair Feedback
3	for(int j=0; j<10; j++)

	Example Feedback
3	for(i=0; i<n; i++)
3	for(i=1; i<=n; i++)

(c) Error-Group $\{E_7\}$

Figure 5: Sample programs for specific errors - (a) $\{E_5 \wedge E_6\}$ where feedback tools perform poorly, (b) $\{E_5\}$ where both tools have lower error survival, and (c) $\{E_7\}$ where examples seems to work better. The top repair tool fix prediction, and the top-2 example suggestions are also shown for each example.

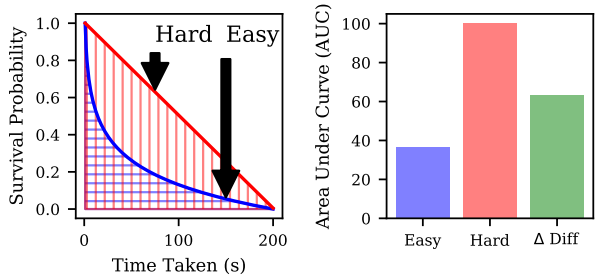


Figure 6: Area Under Curve (AUC) for hypothetical data. The blue curve has lesser error survival probability (easier) compared to the red curve. This performance improvement is quantified as the difference between their AUCs.

resolve would have shallower error survival probability curves[5], and hence have a larger area under this curve. Therefore, we use survival probability AUC as a proxy for problem hardness. Fig 6 shows a hypothetical survival curve and its AUC calculation.

Concretely, let AUC_B represent the average AUC of survival probability plots against time-taken for 2016-2017-II and 2017-2018-I baselines semesters. Let AUC_F represent the best (minimum) AUC of survival curves for repair and example tools when they were deployed during 2017-2018-II semester. Then, the area under the survival curve for baseline AUC_B represents the hardness of resolving this type of error[6]. Also, as we have already seen above, large differences in performance would correspond to large divergences between the survival function curves, with the test group survival probability curve lower than the control group curve if the feedback tools are effective. Hence, we can quantify the difference between the baseline and feedback survival curve area, $AUC_B - AUC_F$ to represent the potential performance improvement of students when feedback tools was deployed. Having defined these two variables, we select the top-100 frequent compiler error-groups. These error-groups occur at a frequency of between 5964 to 35 times on average in the 3 offerings of our course during 7-week *lab* event. Then, we plot survival probability curves (y-axis, 0.0 to 1.0) against time-taken (x-axis. 0s to 200s) for each of these individual compiler error-groups separately.

The relationship between problem hardness and improvement attributable to automated feedback is plotted in Fig 7, with a potential hardness on y-axis against potential performance improvement on x-axis. Positive x-axis indicates that the compiler-error has a lower survival chance for students with feedback-tools, as compared to their previous semester peers. To translate between this graph and our earlier results, consider Fig 4 denoting error survival curve for error-group $\{E_5\}$. The AUC for repair, example, 2017-2018-I and 2016-2017-II is 28.03, 61.78, 82.47 and 74.46 respectively. Then, $AUC_B = (82.47 + 74.46)/2 = 78.46$ and $AUC_F = min(28.03, 61.78) = 28.03$. Hence, error-group $\{E_5\}$ is represented in Fig 7 by a blue-diamond point at y-value= $AUC_B = 78.46$ and x-value = $AUC_B - AUC_F = 50.43$. From Fig 7, we observe that there is performance improvement on the usage of either repair or example feedback tool, across most compiler error-groups. But more importantly, improvements are greater for the harder errors, suggesting that tool usage is more helpful than human tutor help for more complex errors.

This analysis also further clarifies the nature of the logistical advantages provided by the feedback tools. A human tutor has to be signaled, walk to the student, and discuss the compiler error with the student. An automated system faces no such physical limitations. If removing such physical challenges was the primary logistical advantage of feedback tools over human tutors, we would expect no relationship between performance improvement and problem hardness. The fact that we do see such an advantage suggests that the logistical advantage of automated feedback is not entirely physical - feedback tools also have greater facility in presenting feedback for more complex problems.

[5]A shallow survival probability curve implies that the problem is staying alive for more extended periods, thus indicating higher error resolution times.

[6]While we could also use median error resolution time per group to approximate problem hardness, this metric suffers from a confound with practice effects, as described above

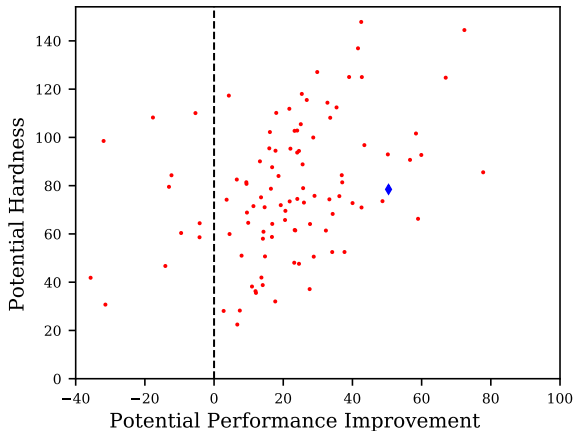


Figure 7: Analyzing potential performance improvement due to feedback tools by 2017-2018-II semester students for time-taken to resolve Top-100 frequent compilation error-groups (*EG*s), against the hardness of the error.

6 THREATS TO VALIDITY

While we have sought to minimize possible confounds in our experimental design, it is appropriate to point out some that we could not control. Most importantly, our human-assisted baseline groups comprised of students from different semesters. While having the control and experimental group within the same semester is desirable, conducting the experiment in real-world lab setting for 7-weeks constrained us from disadvantaging one group of students over the other. Moreover, our control groups of students worked on different programming assignments compared to the experimental groups, while being constrained to the same topic/theme of lab. However, these programming assignments were designed to be of similar difficulty level across semesters, with similar time required to solve them (within 3 hours per lab).

The student cohort from the semester 2017-2018-I seems to, in general, have higher error survival probabilities. This can partially be explained by the fact that they undertake this programming course in their very first semester at the university, as opposed to students from the other control and test groups, who took this course in their second semester. Students in the second semester are expected to have more experience in the computer basics, as well as be better aligned to scholastic expectations in the university.

As we observe in our results, though, the baseline groups are remarkably close to the experimental groups on critical measurements, e.g., survival probability of errors during exams wherein feedback tools had been suspended. Thus, while our study is not a true randomized controlled trial because our control groups were obtained from the immediately previous semesters' offering of the same course, we do not expect this to affect the interpretation of our results substantively.

Our time-taken definition might not always accurately reflect the time taken to resolve compilation error, since the student could be making logical improvements in the same time period. However, this is a random effect across control and baseline groups and hence should not affect our conclusions.

7 CONCLUSION

There exists previous work in literature on developing guidelines and building static tools to improve error messages that novice programmers can easily understand and fix their errors [3, 4, 20, 30, 31, 39, 42]. While this seems to be a better solution, it requires significant manual effort to (re)write *better* error messages. Further, the quality of an error message may not be directly linked to ease of understanding, especially for non-native speakers of the language. We believe such approaches are complementary to the automatic feedback approaches used in this paper, since the enhanced error messages can always be augmented with the automatically generated feedback. Existing user studies on the effectiveness of enhanced feedback are frequently conducted on relatively small study groups [6, 13, 29], or over short periods of time (order of minutes) [6, 33]. Thus, the generalizability of the user studies is difficult to assess.

We set out to conduct a user study impervious to such challenges, to rigorously verify the pedagogical value of automated adaptive feedback in introductory programming. Our randomized controlled experiment involved 10,000+ person-hours (N = 480, 3 hour session per week, over 7 weeks period) of user-study. Additionally, in an improvement over existing studies that frequently report isolated measures of efficacy, we report a comprehensive panel of both conventional and novel metrics to verify the efficacy of automated feedback tools. Our dataset and analysis scripts are released in the public domain for reproducibility, and to support further research *https://github.com/umairzahmed/seet2020*

The large size of our study sample, the randomized nature of our experimental design, and the granular nature of our analysis permit a more nuanced appreciation of the value and limitations of automated feedback tools. Specifically, we find that these tools assist students in fixing compilation errors more rapidly than human tutors, but that this advantage is primarily logistical. In the absence of these tools, students learning to program assisted by these tools are no more effective at repairing compilation errors than human-tutored students. Thus, we conclude that such automated feedback tools cannot, standing alone, be considered effective from a pedagogical perspective. At the same time, these tools have their advantages in the actual task appointed to them - helping students debug their code. As we show in our analyses, these tools prove increasingly superior to human tutors precisely in assisting students in solving rare, complex errors. This is a reasonable finding since the rarity of such complex errors reduces the probability that any individual human tutor will have seen them before. In contrast, the computational resources of the automated tool suffer from no such experiential limitation.

Finally, our results open the tantalizing possibility of designing systems to adapt feedback modalities to individual students' needs rationally. While we have not established this conclusively, our results do suggest that feedback by example is a superior technique when a student's error is caused by ignorance of syntax. In contrast, feedback by repair suggestion is better when errors are inadvertent. Further investigating this hypothesis and finding ways to categorize programming errors into these two categories automatically should lead to the design of such adaptive interface feedback systems.

REFERENCES

[1] Umair Z Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ACM, 78–87.

[2] Umair Z Ahmed, Renuka Sindhgatta, Nisheeth Srivastava, and Amey Karkare. 2019. Targeted Example Generation for Compilation Errors. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 327–338.

[3] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How Should Compilers Explain Problems to Developers?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 633–643. https://doi.org/10.1145/3236024.3236040

[4] Brett A Becker. 2016. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 126–131.

[5] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Unexpected Tokens: A Review of Programming Error Messages and Design Guidelines for the Future. In *ITiCSE '19*. 253–254.

[6] Brett A Becker, Kyle Goslin, and Graham Glanville. 2018. The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, 640–645.

[7] Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. (2016). http://arxiv.org/abs/1603.06129 arXiv:1603.06129 [cs.PL].

[8] Albert T. Corbett and John R. Anderson. 2001. Locus of Feedback Control in Computer-based Tutoring: Impact on Learning Rate, Achievement and Attitudes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) *(CHI '01)*. ACM, New York, NY, USA, 245–252. https://doi.org/10.1145/365024.365111

[9] Loris D'antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. 2015. How can automatic feedback help students construct automata? *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 9.

[10] Rajdeep Das, Umair Z. Ahmed, Amey Karkare, and Sumit Gulwani. 2016. Prutor: A System for Tutoring CS1 and Collecting Student Programs for Analysis. (2016). https://www.cse.iitk.ac.in/users/karkare/prutor/ arXiv:1608.03828 [cs.CY].

[11] Cameron Davidson-Pilon, Jonas Kalderstam, Ben Kuhn, Paul Zivich, Andrew Fiore-Gartland, Luis Moneda, Alex Parij, Kyle Stark, Steven Anton, Lilian Besson, Jona, Harsh Gadgil, Dave Golland, Sean Hussey, Javad Noorbakhsh, Andreas Klintberg, Nick Evans, Matt Braymer-Hayes, Lukasz, Jonathan Séguin, Jeff Rose, Isaac Slavitt, Eric Martin, Eduardo Ochoa, Dylan Albrecht, dhuynh, Denis Zgonjanin, Daniel Chen, Chris Fournier, and André F. Rendeiro. 2018. CamDavidson-Pilon/lifelines: v0.14.6. https://doi.org/10.5281/zenodo.1303381

[12] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 273–278.

[13] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM, 75–80.

[14] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 7.

[15] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 465–480.

[16] Philip J Guo. 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 599–608.

[17] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 930–937.

[18] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning.. In *AAAI*. 1345–1351.

[19] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What Would Other Programmers Do? Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1019–1028.

[20] James J Horning. 1976. What the Compiler Should Tell the User. In *Compiler Construction: An Advanced Course*, G Goos and J Hartmanis (Eds.). Springer-Verlag, Berlin-Heidelberg, 525–548.

[21] Yang Hu, Umair Z Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-factoring based Program Repair applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 388–398.

[22] Edward L Kaplan and Paul Meier. 1958. Nonparametric estimation from incomplete observations. *Journal of the American statistical association* 53, 282 (1958), 457–481.

[23] Kenneth R Koedinger, Albert T Corbett, and Charles Perfetti. 2012. The Knowledge-Learning-Instruction framework: Bridging the science-practice chasm to enhance robust student learning. *Cognitive science* 36, 5 (2012), 757–798.

[24] Sarah K. Kummerfeld and Judy Kay. 2003. The Neglected Battle Fields of Syntax Errors. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20* (Adelaide, Australia) *(ACE '03)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 105–111. http://dl.acm.org/citation.cfm?id=858403.858416

[25] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[26] Richard S Lysakowski and Herbert J Walberg. 1982. Instructional effects of cues, participation, and corrective feedback: A quantitative synthesis. *American Educational Research Journal* 19, 4 (1982), 559–572.

[27] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

[28] Rupert G Miller Jr. 2011. *Survival analysis*. Vol. 66. John Wiley & Sons.

[29] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler error messages: What can help novices?. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 168–172.

[30] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic Grading and Feedback Using Program Repair for Introductory Programming Courses. In *Proceedings of the 22nd ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. ACM, New York, NY, USA, 92–97. https://doi.org/10.1145/3059009.3059026

[31] Raymond S Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 465–470.

[32] Phitchaya Mangpo Phothilimthana and Sumukh Sridhara. 2017. High-Coverage Hint Generation for Massive Courses: Do Automated Hints Help CS1 Students?. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. ACM, New York, NY, USA, 182–187. https://doi.org/10.1145/3059009.3059058

[33] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. ACM, New York, NY, USA, 74–82. https://doi.org/10.1145/3105726.3106169

[34] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_P: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity* (Amsterdam, Netherlands) *(SPLASH Companion 2016)*. ACM, New York, NY, USA, 39–40. https://doi.org/10.1145/2984043.2989222

[35] Alexander Renkl. 2014. Toward an instructionally oriented theory of example-based learning. *Cognitive science* 38, 1 (2014), 1–37.

[36] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 404–415.

[37] Cristina Sanz and Kara Morgan-Short. 2004. Positive evidence versus explicit rule presentation and explicit negative feedback: A computer-assisted study. *Language Learning* 54, 1 (2004), 35–78.

[38] Patrick Shafto, Noah D Goodman, and Thomas L Griffiths. 2014. A rational account of pedagogical reasoning: Teaching by, and learning from, examples. *Cognitive psychology* 71 (2014), 55–89.

[39] Ben Shneiderman. 1982. Designing Computer System Messages. *Commun. ACM* 25, 9 (1982), 610–611. https://doi.org/10.1145/358628.358639

[40] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices* 48, 6 (2013), 15–26.

[41] Emillie Thiselton and Christoph Treude. 2019. Enhancing Python Compiler Error Messages via Stack Overflow. In *Proceedings of the 19th International Symposium on Empirical Software Engineering and Measurement (ESEM '19)*. arXiv:1906.11456. http://arxiv.org/abs/1906.11456

[42] V Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction* 2010 (2010).

[43] Ke Wang, RIshabh Singh, and Zhendong Su. 2017. Data-Driven Feedback Generation for Introductory Programming Exercises. *arXiv preprint arXiv:1711.07148* (2017).

[44] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 481–495.

[45] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. 2012. BlueFix: Using Crowd-sourced Feedback to Support Programming Students in Error Diagnosis and Repair. In *Proceedings of the 11th International Conference on Advances in Web-Based Learning* (Sinaia, Romania) *(ICWL'12)*. Springer-Verlag, Berlin, Heidelberg, 228–239. https://doi.org/10.1007/978-3-642-33642-3_25

[46] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Joint Meeting of the European Software Engineering Conference and the ACM Sigsoft Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM.

[47] Stuart Zweben and Betsy Bizot. 2018. 2017 CRA Taulbee Survey. *Computing Research News* 30, 5 (2018), 1–47.