Factors Affecting Compilable State at Each Keystroke in CS1

Steven Scott
Department of Computer Science
Utah State University
Logan, Utah, United States
stevenderonscott@gmail.com

Arto Hellas
Department of Computer Science
Aalto University
Espoo, Finland
arto.hellas@aalto.fi

Juho Leinonen
Department of Computer Science
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

John Edwards
Department of Computer Science
Utah State University
Logan, Utah, United States
john.edwards@usu.edu

***Abstract*—**
In this paper, we analyze keystroke log data from two introductory programming courses from two distinct contexts to investigate the proportion of events that compile, how this relates to contextual factors, the progression of programs, and academic outcomes. We find that, as students write their programs, frequency of compile and run events increases as does the proportion of events that compile. We also find a spike in the number of compile and run events as a program nears completion, that the proportion of events that compile varies by assignment, length of program, and programming context, that real-time IDE error diagnostics lead to higher proportion of events that are in compilable state, and that a student's awareness of their compilable state is correlated with exam score while the amount of time they spend in an uncompilable state is not. Among the practical implications of our work are the fact that researchers cannot rely on frequency of compilation remaining constant through an assignment and a call to researchers and practitioners to design pedagogies that enhance student awareness of their compilable state.

***Index Terms*—keystroke analysis, keystroke data, programming process data, predicting performance, educational data mining**

I. Introduction

Learning to program is a path paved with errors and learning how to handle them. In the context of computing education research, compilation errors manifested through the work of novice programmers have been under scrutiny for years [13], [31], [8], [26]. The prevalence of compilation errors have been studied across multiple contexts [8], [2], looking both into often-occurring errors as well as into how the types of recurring compilation errors evolve over time [10], [2], [3]. To help learners, researchers have sought to make compilation error messages more informative [12], [5], as well as used compilation errors to identify learners in need of help [24], [47], [6], [9].

A stream of research of particular interest to us, which has become more popular during the last decades, is the analysis of the programming process of a learner [23]. Programming process data can be collected at multiple granularities [23], [42]. Submission data collected by automated assessment systems are perhaps the most common, while keystroke data collected from the programming environment as the programmer types are less common. Keystroke data can then be used to both reconstruct and analyze the programming process [21], [38].

In this paper, we contribute to the understanding of novice programming processes through the analysis of *compilable state*, which refers to whether code compiles or not over the course of completing programming assignments. Using data from two contexts with facilities for keystroke data collection, we analyze compilable state with respect to the programming language, IDE support, progress in writing a program, assignments, and students. Our research questions for the present work are as follows.

RQ1 *What factors affect compilable state?*
RQ2 *How do measures based on compilable state correlate with academic outcomes?*

II. Background

A. Programming errors

Learning to program is, in part, about learning the notation of the programming language and its syntax [15], [27] and thus, it is not surprising that analysis of programming error messages have received plenty of attention within computing education research [7]. Studies have highlighted that a lot of time is spent on figuring out "trivial mechanics" [37], that solving common errors takes a similar amount of time for high-performing and low-performing students [13], and that some errors can take a considerable amount of time to fix when compared to others [3], [13].

Context plays an important role in programmig errors [7]. For example, Denny et al. [14] observed that the majority of programs submitted for assessment from an online drill and

practice system had syntax errors, but Vihavainen et al. [41] noticed that the vast majority of submissions sent for assessment from an IDE did compile. Similarly, an ITiCSE working group [23] focused on source code analytics conducted a replication study of [39], observing that one possible reason for differences in the replication outcomes was that the new dataset came from a context where students were provided template code that did not initially compile, which was not the case for the original study [23]. Naturally, the programming language may also relate to the errors that students face during programming; as an example, Stefik and Siebert [40] observed that languages closer to the English language could be more intuitive to learners. Consequently, it is meaningful to consider also the native language of the learners, as done by Reestman and Dorn [35], who studied BlueJ data and observed small but statistically significant differences in error distributions between country and language groups.

B. Programming process and performance

Analyzing sequential source code state data collected from students' programming process, researchers have developed a variety of metrics to quantify the programming process (and performance). In 2006, Jadud proposed a metric called the Error Quotient (EQ), which quantified the prevalence of errors in subsequent compilations and consequently students' ability to identify and fix them [24]. Watson et al. [47] later expanded on Jadud's approach and introduced a new metric called Watwin score that included the time that students took to fix errors as an additional feature. Carter et al. [9] proposed a different approach called the Normalized Programming State Model (NPSM) that tapped into more fine-grained data collected from an IDE, also considering errors from the perspective of whether students had used debug functionality of the IDE when fixing errors and looking into the prevalence of runtime exceptions in addition to compilation errors. Following on the work of Jadud, Becker [6] looked in more detail into how errors repeated over the programming process, effectively partially quantifying whether students had learned to avoid errors. All of the approaches have been linked with student performance in the respective analyses [24], [47], [9], [5].

The aforementioned approaches, with the exception of the work by Becker, could be seen to quantify the programming process and the construction of the respective metrics through a state machine tuned to the respective contexts. Researchers have also identified challenges in the reproducibility of the results using data collected from other contexts [33], [1], [36]. As an example, Petersen et al. [33] discussed the need to fine-tune parameters of EQ for it to match a particular context, finding that the efficacy of Error Quotient as a metric for performance was rather context-dependent, while Richards and Hunt outlined challenges with applying NPSM to data collected from the BlueJ environment [36]. The granularity of data used for constructing such models also relates to the performance; as an example, Ahadi et al. [1] looked into both EQ and Watwin score constructed based on IDE action data (running, testing, or submitting programs) and pause data (event pairs with pauses longer than ten seconds), and found differences in the scores and their correlation with course outcomes; Ahadi et al. also proposed and evaluated using programming process data to build machine learning models to predict course performance [1].

C. Fine-grained programming process data

A stream of research into the programming process that has become more popular during the last decade has focused on collection of fine-grained data from the programming process [23]. This often constitutes collecting each individual keystroke from the programming process, providing researchers a more fine-grained view to how the program is constructed. Such data can provide insight into e.g. how novices learn to write simple statements [41] and into the programming process [4], [38], as well as into the frequency of compilation errors over the programming process [42].

D. Research gap

While there are plenty of studies that look into syntax errors and errors in the programming process, as well as a handful of studies that have looked into the analysis of keystroke data, there is a gap in prior research in combining these two streams. We observe the need to explore the quantification of the programming process into a continuous state that describes the code either as compiling or non-compiling in order to gain more insight into the struggles of novice programmers.

III. Methodology

A. Context and data

For the present study, we used three datasets from two contexts.

1) Python: We used the two public Python keystroke log datasets published by Edwards [16], [17]. The 2019 dataset contains keystrokes from 505 participants and 5 assignments in a CS1 course and has 5 million unique events. Data was collected using a very basic IDE called *Phanon* [18] that did not have syntax error underlining. The 2021 dataset has 44 participants and 8 assignments in a CS1 course and has 1 million unique events. It was collected using the *PyPhanon* plugin [19] to *PyCharm*. The *PyCharm* IDE has error underlining. Additionally, these datasets include students' high school GPAs, highest ACT score, grade on each assignment, and scores on exams. Assignments were designed to teach foundational programming concepts like variables, control structures, functions, objects, class, etc. Exams are composed of multiple choice, true/false, and fill-in-the-blank questions that are automatically graded using our learning management system. Many questions are general computer science questions (e.g. *What stores more data, 1Mb or 1Gb?* but the majority of questions show Python code and ask questions like "what is output?" and "which is the missing line of code?" Students in this context were undergraduates at a research university in the United States, taking their first computer science course at the university. They may have past programming experience from high school courses or personal projects, but this course

is likely the first programming experience for many of them. Computer science majors, which made up ˜30 % of students in the 2021 cohort, are likely freshmen because CS1 is their initial major-specific course.

2) Java: Java keystroke data was collected from a free open online introductory programming course offered by a Finnish research-first university. The course is offered as an online e-book written in English that contains interleaved theory, worked examples, and programming exercises (for additional details on the course pedagogy, see e.g. [44], [43]). The course covers the basics of programming using Java, with an initial focus on variables and basic control structures, followed by basics of object-oriented programming. In order to complete programming exercises, course participants install an environment which they use to work on exercises locally. The environment collects programming process data (including keystrokes), downloads course exercises for participants, and provides the capabilities for running, testing, and submitting exercises, which are assessed using an automated assessment system (for details, see [45]). The course has no end of course exam and participants are not eligible to receive university credits for completing the course. A course certificate is available upon completion, however. For the present study, we use data from 304 participants who consented to their data being used for research purposes (a total of 11 million events). The data comes from the first four parts of the seven-part course. The workload for the first four parts is approximately 3 ECTS[1].

In total, this analysis uses 17 million keystroke events from 853 participants.

B. Terminology

The following terms are used throughout the remainder of this paper.

- Event - an event is any recorded action taken by the student. Events include keystrokes, pasting code into the program, running the program, and submitting the assignment.
- Compilable state - an event or sequence of events in the programming process where the code compiles with no errors (see below for more discussion on how this is computed).
- Recovery - A recovery begins with an event that switches the program from a compilable state to an uncompilable state. The recovery continues until the first event the returns the program to a compilable state. It is normal for students to have various recoveries while programming, often without realizing they have introduced or resolved errors.
- Known Recovery - A known recovery begins when the student attempts to run their program (Python) or presses the compile button (Java) and the code has an error preventing it from compiling. The known recovery ends when an event returns the program to a compilable state. The student thus knows that an error was present in their program, even if they may be unaware of the exact moment when they resolve the error.

[1]European Credit Transfer System. One ECTS corresponds to approximately 25 to 30 hours of work, while 3 ECTS would amount to 75-90 hours of work.

C. Analysis

For the present work, we define *compilable state* as code that compiles with no errors. The compilable state of a program at a particular event was determined by re-creating the file based on the keystroke data and then using programming-language-specific functions to determine if the program would compile immediately after that event was applied. This compilable state is distinct from run events, which are events where the student attempts to run their program, causing a compile to occur and potentially logging either a compile-time or run-time error. Both run events and compilable state are discussed below.

Compilable state is affected by context differences. In Python, only syntax errors are caught at compilation time, meaning that many errors do not impact compilable state. In Java, the compiler catches more errors, including type checking, undefined variables, and undefined arguments in method calls, which can all impact compilable state. For the present analysis, in both contexts, we reconstructed the evolution of students' source code for the programming exercises, and compiled the source code at each keystroke event.

To answer RQ1, *What factors affect compilable state?*, we visually and statistically study keystroke data collected from both contexts, scoping our work on contrasting compilable state with five context- and programming process-specific factors. To answer RQ2, *How do measures based on compilable state correlate with academic outcomes?*, we look into the proportion of compilable state events in the programming process as well as into recoveries from error states and contrast them with students' exam scores, high school GPA, and highest ACT score, collected for one of the Python datasets.

In the analysis and in reporting, we use the term `compile` and `run` interchangeably to indicate an event where, in Python, the student runs the code and in Java the student clicks a button labeled "run" that compiles and runs the code. While these events are initiated as runs, a compile occurs in both the Python and the Java contexts.

D. Statistical tests

We report p values of all statistical significance tests. Our statistical tests are two-tailed. We follow the American Statistical Association's recommendations to use p values as one piece of evidence of significance to be used in context rather than using an alpha threshold [46].

IV. Results

A. Factors that affect compilable rate

Our first research question RQ1 is: *What factors affect compilable state?* We discuss a number of specific factors: contextual differences (including programming language and

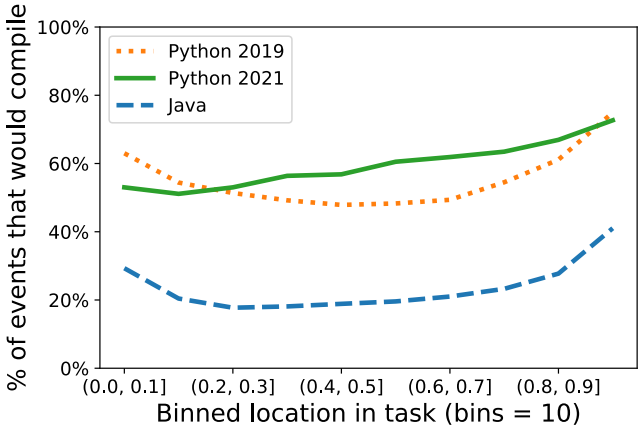


Fig. 1: Change in the percentage of events in a compilable state over the course of tasks. The offset between Java and the other lines is a result of programming language and other context differences. Despite this offset, the 2019 Python data and the Java data follow a remarkably similar trend line. The Python 2021 IDE provided continuous highlighting of syntax errors, unlike the Java or 2019 datasets. All 3 datasets show an increase in compilable state at the end of the programming process.

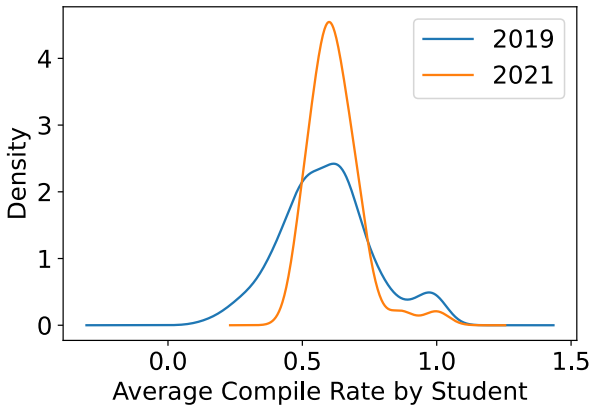


Fig. 2: Distribution of students' average compilable rate by year. The 2019 students did not have underlined error hints whereas the 2021 students did.

whether the IDE offers underlined error hints); how far along a student is in the process of writing a program, solution length, and the type of assignment. We leave other factors, such as instructional methods, student demographics, time of day, time remaining before due date, and IDE complexity, to future work.

1) Context: programming language: Overall, when comparing Python and Java, we observe that Python code is more frequently in a compilable state (56.4% of all events) than Java code (23.7% of all events). We see in Figure 1 that students' Python code is in a compilable state far more often than Java code. However, since all Java data in these datasets was collected in an open-course context in Finland and all Python data was collected in a CS1 course in the United States, the magnitude of this impact cannot be separated from the impact of these other contextual factors.

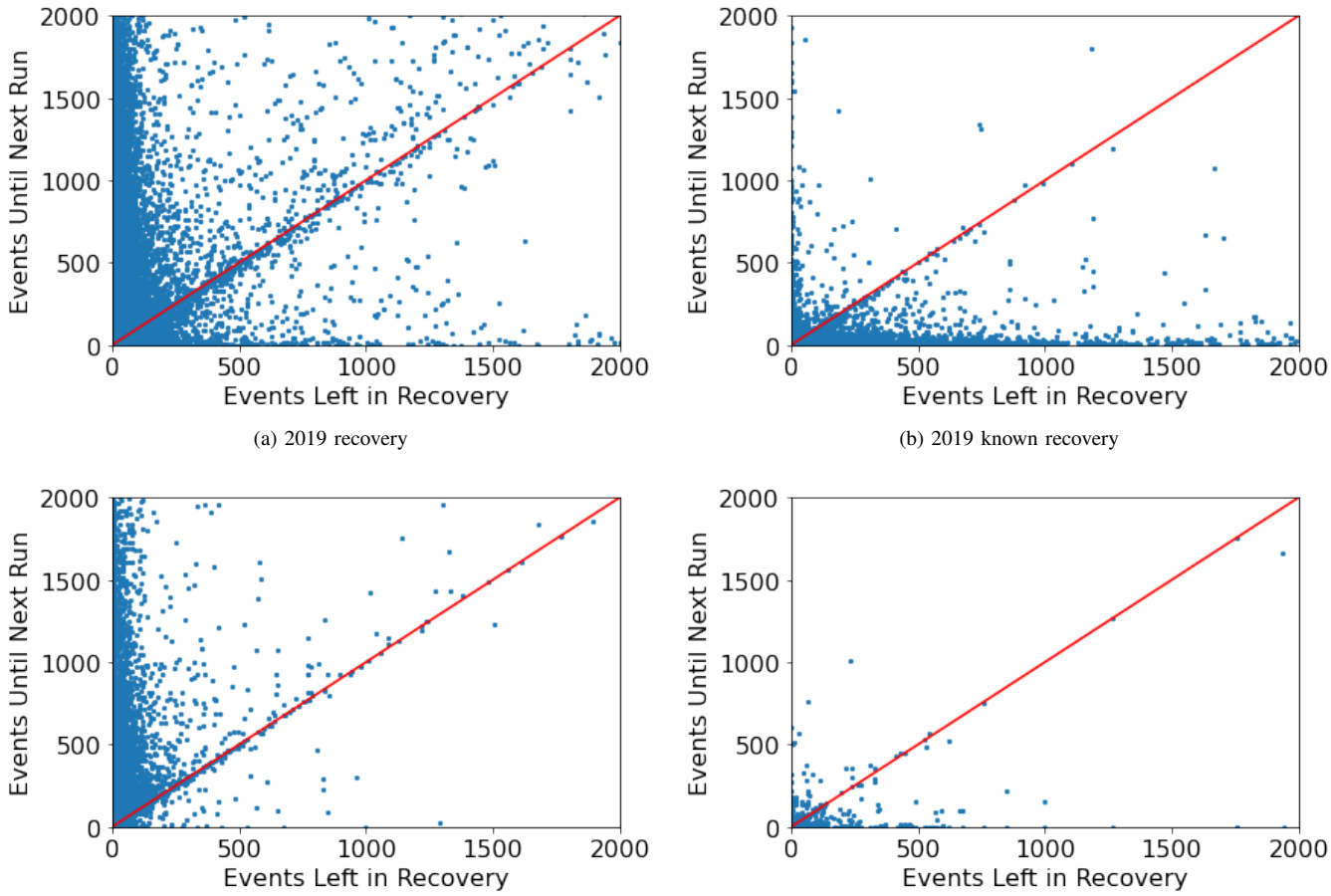
2) Context: IDE error annotations: We analyzed whether using an IDE that underlines errors affects the percentage of events in a compilable state by comparing data before and after an IDE change in the Python data. Moving from an IDE with no highlighting of compilation errors (2019 dataset) to one that continuously highlighted compilation errors (2021 dataset) increased the percentage of events in a compilable state from 58% to 62% (Figure 2). A one-sided t-test for these two averages yielded a t-statistic of -1.444 and a p-value of 0.0746, indicating a possibility that the improvement stemmed from the IDE change.

We also look at the effect of context on recoveries. We define a *recovery* as the series of events starting with an event that moves the code into an uncompilable state and ending with an event that restores the code to a compilable state. Recovery length is the number of events in the recovery. Informally, the recovery length is how long it takes the student to get back into a compilable state. Recoveries happen frequently: when a student is typing a line of code the incomplete line will likely be syntactically incorrect until it is completed. Students may or may not be aware whether their code is compilable at any given moment. We define a *known recovery* as a recovery that begins with a compile that fails. In this case, students are aware that they are in an uncompilable state. It turns out that whether the recovery is known or not makes a big difference in students' behavior.

To analyze student behavior in recovery, we look at the "recovery ratio", which we define as the number of events until the next compile divided by the number of events to the end of the recovery. See Figure 3. A recovery with a low ratio (a dot that appears below the $y = x$ line in the figure) means that the student compiled before the recovery was complete, resulting in a failed compile. A recovery with a ratio on or just above 1 (a dot on or just above the line) means that the student compiled just after the error was fixed. And a recovery ratio above 1 (above the line) means that the student continued programming after fixing the error without stopping to compile.

In Figure 4, we see the distribution of percentage of recoveries per student for which students compiled before returning to a compilable state. In recoveries, students who used an IDE with error underlining (2021 context) compiled before recovering less often ($\mu = 1.3\%$) than students without error underlining (2019 context, $\mu = 4.8\%$), with strong significance ($t = -8.79, p = 2.46^{-17}$).

3) Progression in writing a program: Figure 1 shows the percentage of events in compilable state as students progress towards completion of a programming assignment, divided into ten event bins. From Figure 1, we observe that, in the 2019 Python and Java data, students generally start out their



(a) 2019 recovery

(b) 2019 known recovery

(c) 2021 recovery

(d) 2021 known recovery

Fig. 3: Recovery ratio and known recovery ratio. Each point on the charts represents a recovery. A recovery is the sequence of events to resolve a compilation error in a student's program. A known recovery begins when a student attempts to run or compile their program and ends when the program returns to a compilable state. For instance, if a student introduced a syntax error, typed a few more lines of code, attempted unsuccessfully to compile their program, and then spent several more events fixing the error, the recovery would start with the introduction of the error, the known recovery would start with the attempted compilation, and both would end simultaneously. The x axis is the recovery length and the y axis is the number of events until the next compile. The diagonal line is at $y = x$. Dots below the line indicate recoveries during which the student attempted to compile/run their program before the error was fixed. Java data is similar to 2021 data.

assignment with code that is compilable moderately often. Then as they progress through their work, their code is compilable less and less often, until it reaches a minimum between 1/5 and 2/5 the way to completion. The code then gradually becomes more compilable until the max is reached near submission. This visibly differs for the 2021 Python data with continuous error highlighting, where there is a more stable positive progression throughout.

When looking only into events when students run their code, illustrated in Figure 5 over the progression in a program, we see a trend of increasing compiles towards the end culminating in a spike in compile frequency in the last tenth of events. It also seems that Java students compile their code less often in the middle of the process when compared to Python students.

4) Solution length: The length of an assignment submission, measured by the number of events, appears to also be related to the percentage of compilable events in a solution. The percentage of compilable events and solution length are correlated in the Python context ($r = -0.520, p = 2.11^{-283}$) and in the Java context ($r = -0.169, p = 1.27^{-254}$). Here, approximately 27% of the variance in percentage of compilable events is explained by number of events for Python, while the corresponding number for Java is 2.9%.

5) Variation by assignment: Recall that we define a *recovery* as a series of events between an event bringing the code into an uncompilable state and an event restoring the code to a compilable state. Figure 6 shows the distribution of recoveries taking more than 25 events per every 10,000 events grouped by assignment, showing that the recovery lengths can

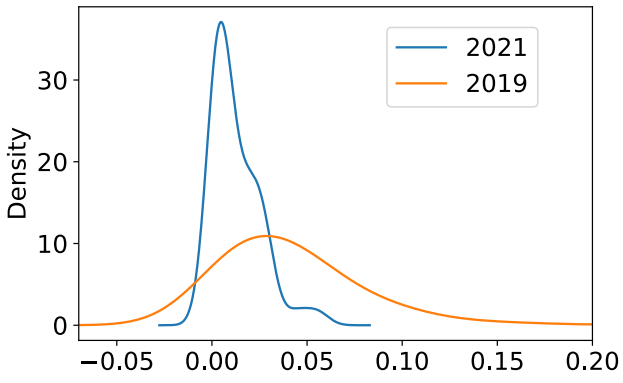


Fig. 4: Distribution of the percentage of recoveries per student for which students ran the code before the error was fixed.

be very different for different assignments. Note that the 2019 assignment 4 had more long recoveries per 10,000 events than the other assignments for both fall and spring semesters. This may indicate that assignment was unusually complex.

B. Correlation with academic outcomes

Our second research question RQ2 is: *How do measures based on compilable state correlate with academic outcomes?*

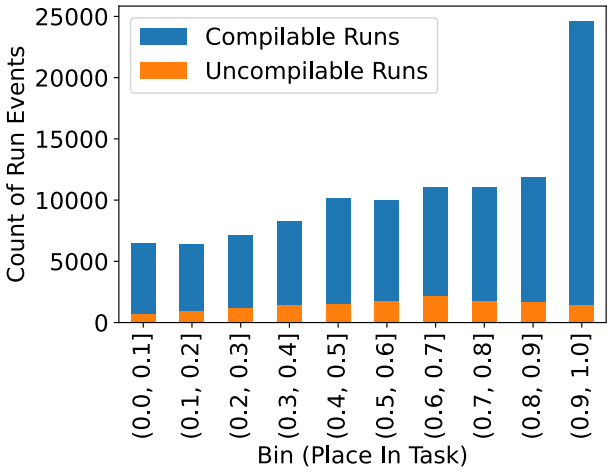
To test this question we used our 2021 Python data, as it is the only dataset used in this paper that has exam score, GPA, and ACT score. We failed to find a statistically significant correlation between the compilable rate and students' exams scores ($r = 0.0347, p = 0.43$), high school GPA ($r = -0.0912, p = 0.61$), or highest ACT score ($r = 0.248, p = 0.16$). The lack of significance here is interesting: students who kept their assignments in a compilable state throughout the programming process did not do significantly better on exams than those who did not.

While compilable rate was not correlated with outcomes, recovery ratios are correlated. For each student, we calculated the percentage of their recovery ratios that were above 1. For example, if a student's percentage is 50%, then during half of their recoveries, the student compiled at least once before fixing the error. This percentage of recovery ratio for each student is weakly correlated with average exam score ($r = 0.142, p = 0.0010$) using the 2019 and 2021 Python datasets, both of which have exam score data.

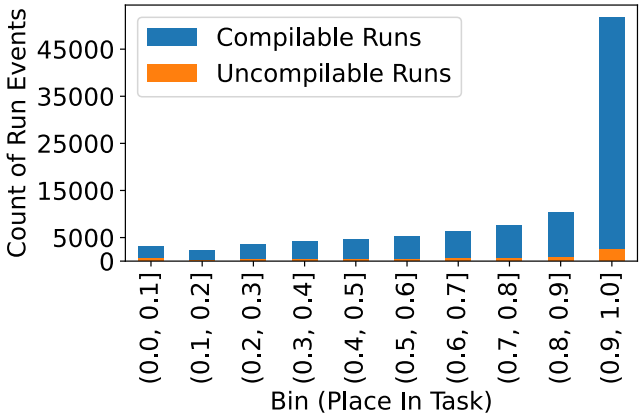
V. Discussion

A. Contextual differences

Our data shows a number of contextual differences, both between the Java and Python context, but also between the Python context with and without continuously available error feedback from the IDE. Perhaps the most notable difference between the contexts was the substantial difference in the proportion of events that compile, where Python programs were more than twice as likely to be in a compilable state than Java programs. We see two key explanations to this:



(a) Python



(b) Java

Fig. 5: Distribution of run events during the course of program development. Failed run events are in orange; successful are in blue. The frequency of run events increases dramatically near the end of writing the program, with a higher percentage of successful compiles in both programming contexts.

(1) differences in compilers and (2) differences in verbosity. In general, Python compilers are more forgiving than Java, with many errors, such as type checking errors, visible only at runtime. Since our Python data does not show whether a run was successful or not, our statistics only indicate whether the Python code had a syntax error or not. Thus, with more errors occurring at compile time, the Java code should have higher error rates than Python.

Regarding verbosity, Java is also a more verbose language, with students required to type relatively large numbers of characters when compared to Python. It is important to note, however, that the difference between the Python and Java event compilability percentages appears to be roughly linear as students write their programs (c.f. Figure 1). That is, an approximately 30 percentage point difference in the rate of

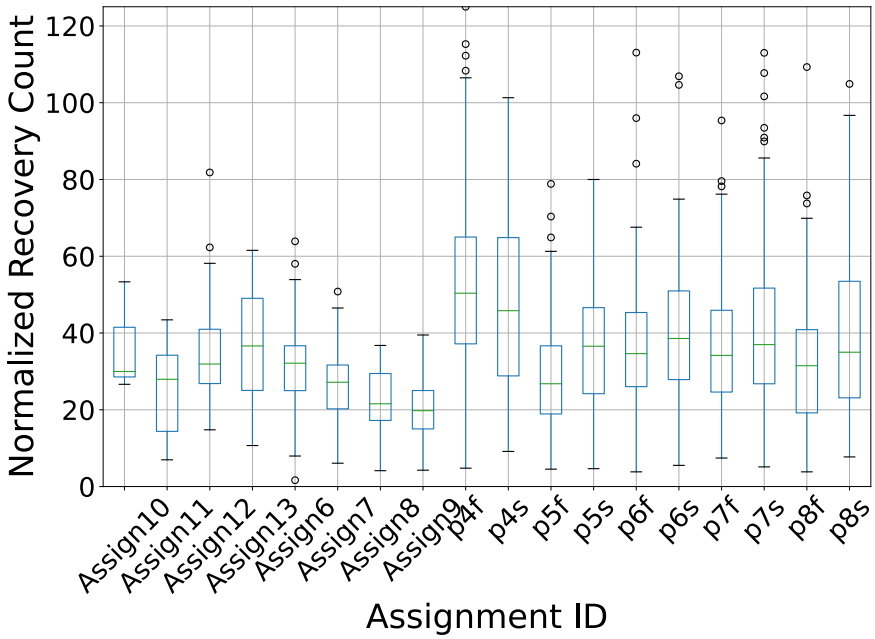


Fig. 6: Distribution of recoveries taking more than 25 events divided per 10,000 events in the assignment per student. Java distributions are similarly varied but not displayed. Assignment names beginning with 'Assign' are from the 2021 cohort, assignment names beginning with 'p' are from the 2019 cohort, with 'f' and 's' indicating whether the results are from the fall or spring semester respectively. Some outliers are omitted from the visualization.

compilability separates Python and Java at the beginning, at the minimum, and leading up to submission. We also observed that Java students were less likely to run their programs in the middle when compared to Python students (c.f. Figure 5).

When considering the effect of the programming environment highlighting compilation errors, we observed that events collected in the Python environment with error highlighting compiled 62% of the time, while events compiled in the Python environment with no error highlighting compiled 58% of the time. These results strengthen the observation of [25], who suggested that students recovered from errors quicker in an environment that continuously compiled the code and provided a highlight of lines that did not compile. Visual inspection of Figure 1 also suggested that the error highlighting helped students maintain their code in a compilable state, as there was no noticeable drop in the process similar to what was observed with the other datasets.

From the recovery data shown in Figures 3 and 4 we see that IDE error highlighting is doing its job – students are more likely to successfully fix their errors before compiling when IDEs highlight errors. It even enables students to continue programming with confidence after fixing an error without being burdened with taking time to recompile.

B. Looking into the programming process

When looking into the programming process, we observed two key characteristics present in both contexts. First, as shown in Figure 1, we observed that the programming process included a dip in compilable rate. Second, as shown in Figure 5, we observed that the rate of running the programs was not constant throughout the process and that students were inclined to run their programs more towards reaching a solution.

When considering prior studies that have focused on analysis of submissions (e.g. [39]), which could be seen as a form of running the code when no local testing possibilities are available, our results indicate that such analyses might be missing significant amounts of information from the process. This concern has been highlighted also in the past, when comparing keystroke data and data collected with other granularities [42], [23].

Differences in data granularity (and contexts) could also outline some insight into the challenges of generalizability of methods used for assessing students' performance based on programming process (c.f. [24], [47]), as discussed e.g. in [33], [1]. When considering the programming process as a continuum of keystroke events, it is only natural that some of the events compile and some of them do not compile. Methods looking into the process at multiple granularities, perhaps building on repeated error density [6], should be looked into to quantify what the students are doing and how they are faring with what they are trying to do.

With more open datasets such as [16], [17], there is a possibility to reinvigorate research into how novices build programs (c.f. [4]), where classic studies have observed behaviors such as stoppers, movers, and tinkerers [32], as well as outlined

how programs are constructed [11].

C. Towards higher compilation rate

When looking into the programming process, the question that naturally emerges is what students are trying to achieve in the process of writing a computer program. Are they writing the entire program and then debugging it or are they gradually building features into a program and debugging each feature? The data appears to support the idea that students are tending to write more features at the beginning of the code-writing process and testing (and potentially fixing) them en masse. This possibility is supported by Figure 5, which outlines the proportion of run events throughout the programming process. As students write their programs they compile and run their code more frequently, culminating in a spike in run frequency in the last tenth of events.

While the above could represent a successful process for some students, prior research has over and over again discussed the challenges related to the syntax of programming languages [15], [27], [3], [13], [7]. We see clear room for improvement in the proportion of events that compile, and call for researchers to look into this aspect. We envision one potential help in the form of an environment with more specific indicators of compilable state, which in turn could highlight the importance of trying to maintain code in a compilable state for students. Drawing inspiration from [34], who used semaphors in a programming environment to indicate that students should run their programs every now and then, we see that researchers could both visualize the compilation rate over time to students, as well as create an intervention similar to [34]. In general, leading to higher compilation rate could help students run their programs more often, which could help them monitor whether they are proceeding in a favorable direction.

D. Compilable state and academic outcomes

While we call for helping students in the programming process, we acknowledge that in the present study we observed that a student's compilable rate is not necessarily directly correlated with a student's exam scores, high school GPA, or ACT scores. The implication is that asking students to remain in a compilable state as much as possible may not actually lead to improved course outcomes. Indeed, recovery ratio, a measure of a student's awareness of their compilable state, is correlated with exam score. Perhaps the answer is not that students should compile more often but that they should increase their ability to independently detect errors. This may run counter to common wisdom, suggesting pedagogies that teach students to detect errors without a compiler and, potentially, without the error underlining that has shown itself so useful in this very paper. As our measures were relatively simple, we see potential of adapting prior methods used for quantifying the programming process and performance (c.f. [24], [47], [9], [6]) to the granularity of keystroke data.

E. Limitations of work

This work has several limitations, which we discuss next. The contexts are considerably different beyond the difference in programming language; the Python dataset came from a US University classroom context, while the Java dataset came from an open online course. There are likely differences in e.g. pedagogy, assignments, support, etc. These difference certainly could have an impact on student programming behavior, impacting compilable state, recoveries, etc.

The programming language characteristics of Python and Java also limit the comparability of their respective compilability rates. For instance, since Java is a statically-typed language, its compiler will raise type errors, whereas dynamically-typed Python's compile() function will not.

Regarding the participants, there is a selection bias in that students and participants allowed their use of data for research. Additionally, we do not know the prior programming background of the participants, which would likely effect how they fare in programming courses; as the Java course was an open online course, it is possible that it is more likely attended by those already somewhat familiar with programming, which would make our observations about the differences in compilation rate somewhat more alarming.

While the 2019 and 2021 cohorts of students in the US context are more similar to each other than the open course students, there are some limitations to their comparability. While both cohorts were taking the same CS1 class targeting the same learning objectives, their assignments were different problem sets and the course instructors were different. Additionally, the COVID-19 pandemic could have changed enrollment patterns for college students between the two student cohorts. Any of these factors could have impacted student programming behavior. Of these, we explore the impact of syntax highlighting and differing assignments in our analysis. We also acknowledge that the 2021 dataset is much smaller than the 2019 dataset.

VI. Conclusions

In the present work, we explored keystroke-level programming process data collected from two universities, one with Python data and one with Java. Because our tools, analysis code, and much of our data is open[2] [16], [17], re-analysis and replication studies [23] should be straightforward. Compiling each event in the keystroke data, we looked into factors contributing to compilable state and how the compilable state relates to academic achievement. To summarize, our research questions and their answers are as follows.

(RQ1) *What factors affect compilable state?* We observed that (1) Python programs were in a compilable state more frequently than Java programs; (2) IDE support for highlighting syntax errors may increase overall proportion of compilable events; (3) programs are less likely to be in a compilable state in the middle of the programming process and reach a higher

[2]Data processing, analysis, and visualization are available at https://github.com/stevescott32/compilable-state

proportion of compilable events towards the end of the programming process; (4) students are considerably more likely to run their programs at the end of the programming process than early on in the programming process; (5) number of events in the programming process correlates negatively with proportion of compilable events, in both contexts; and (6) recovering from a compilation error differs between assignments.

(RQ2) *How do measures based on compilable state correlate with academic outcomes?* We found no strong evidence supporting that the proportion of compilable events would be linked with students' exam scores, high school GPA, or their highest ACT score, but found that student behavior while recovering from errors correlates with exam score.

Overall, our results provide more evidence of contextual factors present in many programming courses, which in turn can help interpret study outcomes and consider the generalizability of results from one context to another. Our results also provide further support for collecting and analyzing fine-grained process data for understanding how programs are constructed [42]. In particular, our data showed a large proportion of run events towards the end of the process of solving an assignment, which would result in lack of insight especially into the early construction process and possible struggles, if one would rely on data collected only from the end of the process. We also call for researchers to develop methods for analyzing and understanding the programming process and struggles from fine-grained data, similar to [24], [47], [9], [6]. Finally, we call for researchers to develop and measure means to help students maintain their code in a compilable state, and highlight a possible inspiration for that work from the development of programming environments with support for emphasizing the need to run programs [34].

As a part of our future work, we are looking into language independent approaches for analyzing programming process data. Prior research has shown promise in using graph- and tree-like structures such as control flow graphs [30], parse trees [42], abstract syntax trees [20], and control flow abstract syntax trees [22] for analyzing and representing student programs. Using such structures for program representation, streams of individual events could be grouped together and linked to certain changes in the structure, which in turn can allow us building a stronger understanding of the types of structures that students struggle to work with.

VII. Data Availability

The Python datasets used in the study are openly available [16], [17]. The Java dataset is not openly available as programmers can be identified from typing data [29] and consent for sharing the data in a non-anonymized format has not been granted. We are, however, exploring options for releasing the data in an anonymized format (combining [28] with AST transformations).

References

[1] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*, pages 121–130. ACM, 2015.

[2] Alireza Ahadi, Raymond Lister, Shahil Lal, and Arto Hellas. Learning programming, syntax errors and institution-specific factors. In *Proceedings of the 20th Australasian computing education conference*, pages 90–96, 2018.

[3] Amjad Altadmri and Neil CC Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM technical symposium on computer science education*, pages 522–527, 2015.

[4] Aivar Annamaa, Annika Hansalu, and Eno Tonisson. Automatic analysis of students' solving process in programming exercises. In *IFIP TC3 Working Conference "A New Culture of Learning: Computing and next Generations*, pages 42–47, 2015.

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

[6] Brett A Becker. A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 296–301, 2016.

[7] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the working group reports on innovation and technology in computer science education*, pages 177–210, 2019.

[8] Neil CC Brown and Amjad Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the tenth annual conference on International computing education research*, pages 43–50, 2014.

[9] Adam S Carter, Christopher D Hundhausen, and Olusola Adesope. The normalized programming state model: Predicting student performance in computing courses based on programming behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, pages 141–150, 2015.

[10] Simon Caton, Seán Russell, and Brett A Becker. What fails once, fails again: Common repeated errors in introductory programming automated assessments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pages 955–961, 2022.

[11] Simon P Davies. Characterizing the program design activity: Neither strictly top-down nor globally opportunistic. *Behaviour & Information Technology*, 10(3):173–190, 1991.

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

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

[14] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 208–212, 2011.

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

[16] John Edwards. 2019 CS1 Keystroke Data, 2022. Harvard Dataverse. https://doi.org/10.7910/DVN/6BPCXN.

[17] John Edwards. 2021 CS1 Keystroke Data, 2022. Harvard Dataverse. https://doi.org/10.7910/DVN/BVOF7S.

[18] John Edwards, Joseph Ditton, Dragan Trninic, Hillary Swanson, Shelsey Sullivan, and Chad Mano. Syntax exercises in CS1. pages 216–226, 2020.

[19] John Edwards, Kaden Hart, and Christopher Warren. A practical model of student engagement while programming. In *Proceedings of the 2022 ACM SIGCSE technical symposium on computer science education*, 2022.

[20] Paul Freeman, Ian Watson, and Paul Denny. Inferring student coding goals using abstract syntax trees. In *International Conference on Case-Based Reasoning*, pages 139–153. Springer, 2016.

[21] Kenny Heinonen, Kasper Hirvikoski, Matti Luukkainen, and Arto Vihavainen. Using codebrowser to seek differences between novice programmers. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 229–234, 2014.

[22] David Hovemeyer, Arto Hellas, Andrew Petersen, and Jaime Spacco. Control-flow-only abstract syntax trees for analyzing students' programming progress. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 63–72, 2016.

[23] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, et al. Educational data mining and learning analytics in programming: Literature review and case studies. *Proceedings of the 2015 ITiCSE on Working Group Reports*, pages 41–63, 2015.

[24] Matthew C Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84. ACM, 2006.

[25] Ioannis Karvelas and Brett A Becker. Sympathy for the (novice) developer: Programming activity when compilation mechanism varies. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pages 962–968, 2022.

[26] Sarah K Kummerfeld and Judy Kay. The neglected battle fields of syntax errors. In *Proceedings of the fifth Australasian conference on Computing education-Volume 20*, pages 105–111. Citeseer, 2003.

[27] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *Acm sigcse bulletin*, 37(3):14–18, 2005.

[28] Juho Leinonen, Petri Ihantola, and Arto Hellas. Preventing keystroke based identification in open data sets. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, pages 101–109, 2017.

[29] Krista Longi, Juho Leinonen, Henrik Nygren, Joni Salmi, Arto Klami, and Arto Vihavainen. Identification of programmers from typing patterns. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 60–67. ACM, 2015.

[30] Andrew Luxton-Reilly, Paul Denny, Diana Kirk, Ewan Tempero, and Se-Young Yu. On the differences between correct student solutions. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 177–182, 2013.

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

[32] David N Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1):37–55, 1986.

[33] Andrew Petersen, Jaime Spacco, and Arto Vihavainen. An exploration of error quotient in multiple contexts. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 77–86. ACM, 2015.

[34] Danijel Radošević, Tihomir Orehovački, and Alen Lovrenčić. New approaches and tools in teaching programming. In *Radošević, D., Orehovački, T., Lovrenčić, A:" New Approaches and Tools in Teaching Programming", Central European Conference on Information and Intelligent Systems, CECIIS*, 2009.

[35] Kyle Reestman and Brian Dorn. Native language's effect on java compiler errors. In *Proceedings of the 2019 ACM conference on international computing education research*, pages 249–257, 2019.

[36] Brad Richards and Ayse Hunt. Investigating the applicability of the normalized programming state model to bluej programmers. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, pages 1–10, 2018.

[37] Anthony Robins, Patricia Haden, and Sandy Garner. Problem distributions in a cs1 course. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 165–173, 2006.

[38] Raj Shrestha, Juho Leinonen, Arto Hellas, Petri Ihantola, and John Edwards. Codeprocess charts: Visualizing the process of writing code. In *Australasian Computing Education Conference*, pages 46–55, 2022.

[39] Jaime Spacco, Paul Denny, Brad Richards, David Babcock, David Hovemeyer, James Moscola, and Robert Duvall. Analyzing student work patterns using programming exercise data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 18–23, 2015.

[40] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):1–40, 2013.

[41] Arto Vihavainen, Juha Helminen, and Petri Ihantola. How novices tackle their first lines of code in an ide: Analysis of programming session traces. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, pages 109–116, 2014.

[42] Arto Vihavainen, Matti Luukkainen, and Petri Ihantola. Analysis of source code snapshot granularity levels. In *Proceedings of the 15th Annual Conference on Information technology education*, pages 21–26. ACM, 2014.

[43] Arto Vihavainen, Matti Luukkainen, and Jaakko Kurhila. Multi-faceted support for mooc in programming. In *Proceedings of the 13th annual conference on Information technology education*, pages 171–176, 2012.

[44] Arto Vihavainen, Matti Paksula, and Matti Luukkainen. Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 93–98, 2011.

[45] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 117–122. ACM, 2013.

[46] Ronald L Wasserstein and Nicole A Lazar. The asa statement on p-values: context, process, and purpose, 2016.

[47] Christopher Watson, Frederick WB Li, and Jamie L Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*, pages 319–323. IEEE, 2013.