Programming Versus Natural Language: On the Effect of Context on Typing in CS1

John Edwards
john.edwards@usu.edu
Utah State University
Logan, Utah

Juho Leinonen
juho.leinonen@helsinki.fi
University of Helsinki
Helsinki, Finland

Chetan Birthare
chetan16994@gmail.com
Utah State University
Logan, Utah

Albina Zavgorodniaia
albina.zavgorodniaia@aalto.fi
Aalto University
Espoo, Finland

Arto Hellas
arto.hellas@aalto.fi
Aalto University
Espoo, Finland

ABSTRACT
Analyzing keystroke data from students working on essay and programming tasks, we study to what extent the difference in task context influences performance in typing. Using data from two introductory programming courses offered at two separate institutions, we compare and contrast typing speed between programming and natural language tasks. We observe that students tend to be faster at typing (the same) character pairs when writing natural language text than when learning to write code. We show that students improve on typing character pairs that appear in frequently used words in programming languages, and that typing programming constructs also improves. We find that students are faster at detecting and erasing their mistakes when typing natural language text than when programming. Our results support theories regarding contextual memory, procedural memory, and practice, and have implications for course curriculum and pedagogy design.

CCS CONCEPTS
• Social and professional topics → Computing education.

KEYWORDS
keystroke data, context, digraphs, programming process data

1 INTRODUCTION
The contexts in which we work, study, and live shape our reactions and memory [4, 29, 47]. In the classroom, we know our students, and we often easily remember their names. At the same time, if we happen to meet them in a context outside of the classroom, recalling their names can be difficult. This effect is a result of context-dependent memory [47], which causes us to better recall information in the same context in which it was learned. For example, recalling information is easier underwater than on land if the information was learned underwater [20]; the same effect holds for many situations [47]. In addition to recall, the context influences our actions and behavior. For example, students within a mathematics classroom are more likely to solve a problem using a mathematical approach than students working on the same problem in a social sciences classroom, i.e., “interpretations of how to solve a problem seem to relate closely to [students’] implicit and explicit assumptions about what is a natural mode of proceeding in a certain situation and given a certain type of task” [50].

Most of the research on context-dependent memory has focused on recall, where participants are first presented a set of stimuli, which they are later prompted for. There exists many types of memories, however, one of which – procedural memory – is of particular interest to computer science educators. Procedural memory is a type of unconscious and long-term memory responsible for motor skills such as talking or writing [24, 48]. Such memory is constructed through practice where the activity is repeated over and over until the learned activity becomes automatic [17].

When students learn to program, they work in several overlapping domains, one of which is related to writing syntactically correct code – a key part of being able to program [10, 28]. As mistakes with syntax can lead to significant struggles [1, 6], and as student success is linked with syntax error rates [22] and typing patterns [13, 33, 52], insight into how students type could help when designing new interventions. Understanding student behaviors at the level of procedural memory, or muscle memory, is where our work is situated. We look at this from two different types of tasks, typing computer code and typing natural language, which – we hypothesize – could partially rely on context-dependent memory.

In this work, we study student typing in two contexts: programming in a CS1 course and typing a natural language essay. Further, we analyze the development of student typing as they progress.
through a CS1 course. We report and discuss quantitative measures such as typing speed, frequency of typing errors, and error recovery between the contexts as well as how typing speed and typing errors evolve over the course. To improve the generalizability of our results, we use timestamped keystroke data collected from two separate universities that use different programming languages (Java and Python) and different spoken languages (Finnish and English).

Our overarching research themes in this article are **how do student typing patterns differ between writing natural language and writing computer code and how do those patterns evolve as proficiency in programming increases**. Specific questions include whether students type faster in natural language than in programming, how typing patterns evolve in programming as proficiency increases, and how error detection and recovery differs with context and evolves with time. Answering these questions can contribute to our understanding of context and procedural memory in CS education, as well as lead to development of interventions, pedagogies, and curricula to improve student outcomes and attitudes in programming courses.

Much of the previous research on keystroke data collected from programming courses has focused on specific uses of the keystroke data such as predicting programmer proficiency [13, 33, 52], evaluating curricular innovations [14, 30] and identifying programmers [35]. To our knowledge, however, there exists very little work that uses multiple sources of keystroke data. One of the few is [39], in which the authors sought to determine how the accuracy of identifying programmers based on typing changes across tasks.

This article is organized as follows. In Section 2, we outline the background on which this work builds upon, discussing streams of research on memory, schemas, and typing patterns. In Section 3, we outline the study contexts, data, research questions, and the analysis methodology. Results of our study are presented in Section 4 and further discussed in Section 5. Conclusions and possible directions for future work are presented in Section 6.

## 2 RELATED WORK

In this section we outline related work on context-dependent memory, schemas and problem solving, studies on cognition and programming with a focus on typing patterns, and studies on typing patterns and flow of typing.

### 2.1 Memory and context

Long-term memory can be broadly categorized into two main categories: explicit (declarative) memory and implicit (non-declarative) memory. Explicit memory [53] is responsible for the conscious recollection of information, facts, and experiences, while implicit memory [44] is related to unconsciously used skills such as walking, writing, and the ability to distinguish between categories of items. One of the most important types of implicit memory is procedural memory [24]. It helps one to perform particular tasks without consciously thinking about these activities. Skilled typing is a part of motor procedural memory.

Learning happens in various contexts, and research has shown that it plays a notable role in recall [47]. If information is asked for in the same context it was learned, recall accuracy would be higher than it would be in other contexts [20]. This effect on improving remembering and consequently learning is called context-dependent memory [54]. It is also observable in behavior – context influences actions as it may prime us to behave in particular context-specific ways [50]. At the same time, trying to visualize the desired context may already help recall [47], e.g., by mentally tracing steps to identify missing items.

A number of studies demonstrated dependency of verbal memory performance on such contexts as environment [47], physiological state [15], background music [46], and even chewing a piece of gum [62]. Some of the studies also have shown that perceptual-motor skills depend on such learning contexts as temperature [21], colour [42, 43] and location [42]. The influence of presence or absence of learning contextual stimuli on retrieval of learnt skill/information during the test is called context-dependent retrieval [42]. Such contextual dependencies tend to wane with time as the skill becomes more practised [38, 63].

### 2.2 Schemas and problem solving

Schemas are memory structures that an individual uses to represent categories of information. Schemas and their connections are an integral part of how an individual retrieves and stores information from long-term memory, also guiding patterns of behavior [19]. When learning, new information can be interpreted through existing schemas retrieved to the working memory, or if no appropriate schemas are identified, existing schemas may be adjusted or new schemas may be created [40, 51].

Schemas are an integral part of problem solving, and they guide behavior when faced with a problem [49]. This applies to the domain of programming [5, 7, 41]. Similarly to the general theory [49], it has been suggested that a programmer who writes a solution to a programming problem in a linear manner may have identified and applied an existing schema, while solution to a programming problem for which no existing schema is found is constructed based on schemas on related problems and means-end analysis [5, 41]. Over time, the accumulation and evolution of schemas allows experts to perform familiar tasks fluently, and also, to handle new tasks with less effort [55]. An important part of this process is generalization, where the learner learns to ignore details and to focus on the important areas of the problem [41].

Schema theory applies to computer programming in more than just higher-order tasks like problem solving – it also has application in tasks, like typing, that are largely driven by procedural memory. Schank and Abelson [45] developed the concept of a script – an action-oriented schema for a sequence of actions or events. Scripts imply specific routine actions for completing tasks and involve well-learnt procedural motor and cognitive skills. Schemas involved in skilled typing store procedural rather than declarative information [24].

### 2.3 Cognitive Load

Cognitive load is the amount of work imposed on working memory and it refers to the amount of resources used by working memory while dealing with a task. Three types of cognitive load have emerged from the Sweller’s Cognitive Load Theory (CLT) [49] – **intrinsic**, which is the effort associated with a specific topic, innate difficulty; **extraneous**, which refers to the way information or tasks are presented to a learner and represents processing information
were divided into types based on the types of characters that comprise the digraph: numeric, alphabetic, other and “edge” (where the characters are of different types). Both found that numeric digraphs and edge digraphs are most indicative of future performance. In Thomas et al.’s [52] and Leinonen et al.’s [33] studies, digraphs used in programming assignments were lower compared to when the context was solely programming as “Python” keystrokes, and keystrokes from the essay as “English” keystrokes. There were three sections of the course all taught by the same instructor. Projects and instruction were the same for all three sections. At the beginning of the semester students were given the opportunity to opt into the study according to the university’s IRB protocol, and this paper uses data only from students who opted in. The course was identical for students who chose to participate in the study and those who chose not to.

3.1.2 University B. University of Helsinki is a research-first university in Helsinki, Finland. The data for this study was collected in the Spring of 2017, during which a 14-week introductory programming course was offered online. The introductory programming course is given in Java and it covers the basics of programming, including procedural programming, object-oriented programming and functional programming.

Each week in the course, students work on tens of programming assignments, which are interleaved in the course material. The material is built so that the programming tasks are sequenced to provide scaffolding to students. Whenever a new topic is learned, students are first asked a few quiz-like questions about the topic, after which the students work on a handful of smaller programming assignments. The small programming assignments together build into larger programs. After this, students practice constructing larger programs that use the topics that they have just learned (for additional details on the pedagogy, see [57, 58]).

Programming assignments in the course are completed using Test My Code [59], which is a desktop IDE accompanied with an automated assessment plugin that provides students feedback as they are working on the course assignments. Combined with an automatic assessment server, the plugin also provides functionality for sending assignments for automatic assessment. In addition to the
we measure the four inter-character latencies of Hello. All of our measures are based on keystroke-derived digraphs and (Section 4.4.1) we do not use a threshold. We do not generally measure keystrokes before H considering the amount of time elapsed between typing and the last keystroke. Previous studies (e.g. [9, 35]) did not hold in those cases. We also use linear regression and the Cox-Stuart test in trend analysis.

Table 1: Summary of contexts

<table>
<thead>
<tr>
<th>Attribute</th>
<th>University A</th>
<th>University B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Lectures w/sections</td>
<td>Online, end of course essays on campus</td>
</tr>
<tr>
<td>Language (prog.)</td>
<td>Python</td>
<td>Java</td>
</tr>
<tr>
<td>Language (essay)</td>
<td>English</td>
<td>Finnish</td>
</tr>
<tr>
<td>Participants</td>
<td>254</td>
<td>113</td>
</tr>
<tr>
<td>Environment</td>
<td>Web-based</td>
<td>Desktop</td>
</tr>
</tbody>
</table>

support and assessment, the plugin collects keystroke data from the students’ working process, which allows fine-grained plagiarism detection approaches for the online course and makes it possible to provide more fine-grained feedback on students’ progress.

At the end of the 14-week course, students come to the university campus for an exam, where they work on both essays as well as on programming assignments. The exam covers content from the course, but in addition, students also write about their motivations and aspirations in the essays. Essays were written in an online notepad, which similar to the programming assignments, records keystroke data of students’ writing process.

For the purposes of this study, we use keystroke data from both programming assignments and essays of those students who came to the exam and gave research consent.

The contexts are summarized in Table 1.

3.1.3 Statistical tests. We report $p$ values of all statistical significance tests, of which there are 42. We follow the American Statistical Association’s recommendations to use $p$ values as one piece of evidence of significance, to be used in context [61]. For distribution comparisons we use the Mann-Whitney U test as normality did not hold in those cases. We also use linear regression and the Cox-Stuart test in trend analysis.

3.2 Digraphs

All of our measures are based on keystroke-derived digraphs and similar constructs. A digraph is a pair of consecutive keystrokes. For example, typing Hello yields four digraphs: H→e, e→e, e→l, and l→o. The latency of a digraph is the amount of time elapsed between the two keystrokes. In all of our measurements we consider only digraphs with latencies greater than 50ms. Previous studies on keystroke data use a lower boundary of 10ms (e.g. [9, 35]), but our lower boundary is 50ms as we have observed that some auto-complete events in IDEs take longer than 10ms. For most of our analyses we use an upper threshold of 750ms to filter out disengagement from procedural memory-driven typing [9, 35]. We raise the threshold to 2000ms when measuring digraphs in the context of a programming language construct as multiple digraphs constituting a construct indicates engagement. Analyses with the higher threshold are evolution studies (Section 4.3). For the accuracy study (Section 4.4.1) we do not use a threshold. We do not generally measure the time taken to initiate typing a construct. For example, while we measure the four inter-character latencies of Hello, we don’t consider the amount of time elapsed between typing H and the last key struck before H.

3.3 Research Questions and Contributions

Our research questions for this study are as follows.

RQ1 How does context influence the typing speed of computer source code and written language?

RQ2 How does the typing speed of programming language constructs evolve over time?

RQ3 How does the number of typing mistakes while programming evolve and how does the speed at which students correct mistakes differ between programming and writing natural language?

For RQ1, our hypothesis is that all digraphs will show lower latencies in natural language than in programming, as students are in the unfamiliar environment of computer programming, whereas typing natural language will be far more familiar and practiced. For example, we predict that the p→r digraph will be typed more slowly in Python as compared to English.

While RQ1 focuses on differences in typing code and natural language, RQ2 focuses specifically on typing code and how it evolves. We hypothesize that digraph latencies and whole construct latencies decrease with practice. But, again, context comes into play, as schemas learnt for a particular construct may not be immediately transferred into a different context.

RQ3 deals with typing errors. Our work deals strictly with errors in typing that students catch and correct within a few keystrokes. This is different than analysis of syntax errors at compile time performed by a number of researchers (e.g. [2, 22]). We look at two aspects of typing errors. The first is in the number of errors committed. Our hypothesis is that with increasing familiarity of programming the number of errors in typing familiar constructs decreases. The second aspect is in error recovery time. As measured by the time it takes students to press the delete key after typing an incorrect character our hypothesis is that students are faster at error detection and recovery when typing natural language, and get faster at detection and recovery as their programming skills improve.

We note that we are not directly comparing data between universities, i.e., we do not compare Finnish to English nor do we compare Java to Python. Rather, we use the two university contexts primarily to show generalizability of our results.

Our contributions are as follows: 1) results and analysis that support the theory that procedural memory is affected by context both at a language level (e.g. Java vs. Finnish) and at a word level (e.g. print vs. int), supporting the theory of context-dependent schemas; 2) experimental evidence that practice improves procedural skills in the computer programming context and support for the possibility of measuring cognitive load with keystroke measurements; 3) evidence that practice improves typing error rates as well as time required to detect and correct errors.

4 RESULTS

4.1 Descriptive Statistics

Table 2 shows overall keystroke statistics across the four contexts of the study. In both cases, the number of digraphs in the programming context exceeds that of the natural language context, though, as we will see, the number of natural language digraphs still admits...
statistical testing. The Java context has significantly more digraphs (157000) per student compared to the Python context (19000) due to the larger number of weekly assignments. We also see from Table 2 that, in general, students are faster at typing natural language than computer code, an observation that we will explore.

4.2 Context and Typing Speed

Our first research question (RQ1) is how does context influence the typing speed of computer source code and written language? Our hypothesis that digraphs would have lower latencies (faster) in natural language than in programming is supported by the data shown in Table 2 and Figure 1. As reported in Table 2, on average, English character pairs are typed approximately 49 ms faster than in Python (Mann-Whitney $U = 2.08 \times 10^{12}, p < 0.0001$). Similarly, Finnish character pairs are typed approximately 37 ms faster than Java character pairs (Mann-Whitney $U = 3.10 \times 10^{12}, p < 0.0001$). Digraphs shown in Figure 1a are frequent in both Python and English, and digraphs in Figure 1b are frequent in both Java and Finnish. In the Python/English case, all character pairs are typed faster in English than in the first week of programming, and for eight of the nine, English remains faster in the last week. We also see that speed generally increases from the first to last week, an effect that we will explore in more detail in Section 4.3. Figure 1b shows that the Java/Finnish case is similar: eight of the ten top digraphs are faster in Finnish than Java, with a somewhat general trend of improvement from first to last week, although some digraphs saw the opposite effect (again, see Section 4.3). We controlled for the lack of similarity between programming language and natural language by taking the intersection of the top 40 digraphs in Python/English, and the top 30 digraphs in Java/Finnish. We chose 40 and 30, respectively, to yield about 10 top digraphs in each intersection.

It turns out that language is not the only context to affect typing speed, but that the word being typed also has a strong influence on a student’s proficiency at typing a character pair. In Figure 2 we study digraphs in two common constructs in Python, `print()` and `range()` and compare them to two other contexts: when the digraph is used in another word when programming, and when it is used in any word in English. For each digraph we performed two $t$-tests: “in word” against “not in word” and “in word” against “English”. Cases in which $p$ is small for both tests indicate, with some confidence, a digraph for which typing speed is influenced by word context. We see from the chart that students type $r\rightarrow i$, $i\rightarrow n$, and $n\rightarrow t$ faster in `print()` than not, but students type $n\rightarrow g$

<table>
<thead>
<tr>
<th>context</th>
<th>digraphs/student $\times 10^{-3}$</th>
<th>latencies (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>19 ($\sigma = 11$)</td>
<td>230 ($\sigma = 158$)</td>
</tr>
<tr>
<td>English</td>
<td>1.7 ($\sigma = .95$)</td>
<td>181 ($\sigma = 122$)</td>
</tr>
<tr>
<td>Java</td>
<td>157 ($\sigma = 63$)</td>
<td>216 ($\sigma = 148$)</td>
</tr>
<tr>
<td>Finnish</td>
<td>3.6 ($\sigma = 1.4$)</td>
<td>179 ($\sigma = 121$)</td>
</tr>
</tbody>
</table>

Table 2: Average number of digraphs per student and digraph latency for the four languages with standard deviations. The number of digraphs is given in thousands and latencies are in milliseconds. Digraphs are restricted to those with latencies between 50ms and 750ms.

Figure 1: (a) Box plot of latencies of the top digraphs in both Python and English. The selected digraphs are the intersection of the 40 most common digraphs in Python and the 40 most common digraphs in English. SP is the space character. (b) Average typing speeds of the intersection of top 30 digraphs in Java and Finnish.

**Figure 1**: (a) Box plot of latencies of the top digraphs in both Python and English. The selected digraphs are the intersection of the 40 most common digraphs in Python and the 40 most common digraphs in English. SP is the space character. (b) Average typing speeds of the intersection of top 30 digraphs in Java and Finnish.

more slowly in `range()` than not. For possible explanations see the discussion section.

4.3 Evolution of Typing Speed

Our second research question (RQ2) is how does the typing speed of programming language constructs evolve over time? We study the evolution of typing speed from two perspectives. First, we analyze the evolution of individual digraphs over time and their correspondence to specific programming constructs. Then, we study how the typing speed of complete programming constructs evolves over time.
We report the number of times the construct was typed. Students when measured in all contexts, a spike in latency occurs around the ward trend, indicating improvement in speed of typing. However, results of is all uses in English. Digraphs marked with * have t-test range(). When typed in the context of print(), are largely measuring use in the context of print(). At least 75% of the students were typing this digraph 80 times through the course of the semester. Notably, the downward trend is observed when has a downward trend (see Table 3) as students type it the first 80 course. Consider the digraph p→r. Our hypothesis is that the speed of typing the latency evolution of digraphs shown in Figure 3 and other digraphs. The word is the construct the digraph is used in. n is the number of times the construct was typed. Students in the Java course typed the i→n digraph more times than in Python, so we report with the additional data (400 uses). We report p-values as a measure of confidence in the coefficients.

4.3.1 Digraph evolution. Our hypothesis is that the speed of typing individual pairs of characters improves over time. Our data support the hypothesis, although context of what word is being typed must also be considered. We illustrate this with data from the Python course. Consider the digraph p→r (Figure 3a). The latency of p→r has a downward trend (see Table 3) as students type it the first 80 times through the course of the semester. Notably, the downward trend is observed when p→r is typed in the context of print(). and also when it is typed regardless of context, since, for each use, at least 75% of the students were typing print(), so both curves are largely measuring use in the context of print().

We contrast the progression of p→r with the digraph n→t (Figure 3b). When typed in the context of print(), n→t has a downward trend, indicating improvement in speed of typing. However, when measured in all contexts, a spike in latency occurs around the 20th use. It was at about this time that students started practicing conversion of strings to integers using the int() function – at the first use of n→t, 90% of uses were in print(), while by use 20, print() had dropped to 48%, with the next most common context being int(), accounting for 11% of users. As seen in Figure 3b, this shift causes the mean n→t latency, in the context of any word, to actually increase, meaning students were getting worse at typing n→t. So students appear to be improving as long as the character pair is restricted to a word context. Some readers may note that the latency of the n→t digraph in all contexts does not improve after use 20. This is because, after int() is introduced, other constructs with n→t are introduced shortly thereafter further complicating the latency signal of n→t usages.
We study the average typing speed of the student population when writing keywords, and analyze how the typing speed evolves over the course. For this analysis we allow digraph latencies up to 2000ms to account for initial learning. The lower boundary remains the previously used 50ms.

- When studying the first and last attempts at writing a construct, we study the transition speed between each letter pair, e.g. w→h.
- When studying the evolution of typing over time, we study the overall typing speed of a construct, e.g. while, using the average typing speed of the construct over the population.

We analyze student performance in writing commonly used constructs listed in Table 4. The `System.out.println()` construct, we study the transition speed between each letter pair, e.g. `System.out.println(to);` command that is used for printing.

The time that students took on average to write each construct for the first and for the last time is reported in Table 4, which also includes results from a paired Wilcoxon Signed-Rank test testing for the difference between the first and the last time that each student wrote the construct. We observe that the difference is significant in words not commonly used in Finnish (or English); there is a 24% reduction in time that students took to write `ArrayList`, and a 19% reduction in time that students took to write `sout→TAB`. At the same time, no improvement in typing while is observed. Similarly, in the Python context, `print()` saw a 15% reduction and `range()` a 16% reduction, while `import` didn’t have a reduction. This is presumably because of the parenthesis following `print` and `range`. The reduction in time for Python constructs is impressive considering the study period was much shorter so students had far fewer attempts at typing the constructs than in the Java course. We consider this effect in the discussion section.

Evolution of the typing speed of `ArrayList` and `sout→TAB` is depicted in Figure 4. A downward trend is clearly observable from the figure when writing `ArrayList`, while the trend with `sout→TAB` is more subtle. A Cox-Stuart trend test indicates a downward trend for both `ArrayList` and `sout→TAB` ($p < 0.0001$ and $p = 0.001$ respectively).

Further, we looked into the evolution of the constructs by decomposing them into the digraphs they are built from. Figure 5 shows the average latencies of typing `sout→TAB` for students. We observe that the first keystrokes for a keyword do not evolve significantly, but that the evolution is noticeable in the latter keystrokes for a keyword. Moreover, pressing special keys (e.g. shift for uppercase character) improves noticeably over time. For example, between the first and last use of `sout→TAB`, the time that students take to move from pressing `t` to pressing `TAB` decreases on average by 26%.

4.3.2 Construct evolution. Next, we focus on writing keywords. We study the average typing speed of the student population when writing keywords, and analyze how the typing speed evolves over the course. For this analysis we allow digraph latencies up to 2000ms to account for initial learning. The lower boundary remains the previously used 50ms.

The time that students took on average to write each construct for the first and for the last time is reported in Table 4, which also includes results from a paired Wilcoxon Signed-Rank test testing for the difference between the first and the last time that each student wrote the construct. We observe that the difference is significant in words not commonly used in Finnish (or English); there is a 24% reduction in time that students took to write `ArrayList`, and a 19% reduction in time that students took to write `sout→TAB`. At the same time, no improvement in typing while is observed. Similarly, in the Python context, `print()` saw a 15% reduction and `range()` a 16% reduction, while `import` didn’t have a reduction. This is presumably because of the parenthesis following `print` and `range`. The reduction in time for Python constructs is impressive considering the study period was much shorter so students had far fewer attempts at typing the constructs than in the Java course. We consider this effect in the discussion section.

Evolution of the typing speed of `ArrayList` and `sout→TAB` is depicted in Figure 4. A downward trend is clearly observable from the figure when writing `ArrayList`, while the trend with `sout→TAB` is more subtle. A Cox-Stuart trend test indicates a downward trend for both `ArrayList` and `sout→TAB` ($p < 0.0001$ and $p = 0.001$ respectively).

Further, we looked into the evolution of the constructs by decomposing them into the digraphs they are built from. Figure 5 shows the average latencies of typing `sout→TAB` for students. We observe that the first keystrokes for a keyword do not evolve significantly, but that the evolution is noticeable in the latter keystrokes for a keyword. Moreover, pressing special keys (e.g. shift for uppercase character) improves noticeably over time. For example, between the first and last use of `sout→TAB`, the time that students take to move from pressing `t` to pressing `TAB` decreases on average by 26%.

4.4 Typing Mistakes

Our third research question (RQ3) is how does the number of typing mistakes while programming evolve and how does the speed at which students correct mistakes differ between programming and writing natural language? We measure accuracy of typing a string using a
lookahead approach. For example, to compute the error of typing the string “print” we do the following: when “p” is pressed we begin looking for “r”. If within five keystrokes the student types “r” with no accumulation of characters (i.e. all non-“r” characters were deleted using the delete key) then we proceed and look for “t”. We follow this approach until ‘print’ is typed or until more than five keystrokes are used while looking for a particular character. If we successfully find the string s (s = “print” in our example) then we define the accuracy a to be

\[ a = \frac{|s|}{n-d} \]  

where |s| is the number of characters in token t, n is the total number of keystrokes, and d is the total number of times the delete key was pressed. Note that this method is not effective if a user highlights a portion of the string and deletes or replaces it.

4.4.1 Accuracy in typing code. Our hypothesis is that students would type increasingly error-free code with practice, at least in the context of common constructs. Figure 6 shows that with practice, students type most constructs more accurately. However, it is not consistent across all constructs. Most notable is Python’s `import`, which showed an increase of errors in typing.

4.4.2 Error recovery. Our hypothesis that students are faster at fixing mistakes in natural language than when programming is supported by our results in both the Python/English and Java/Finnish courses as can be seen in Figure 7, which shows the distribution of digraph latencies of the form `!delete→delete`, i.e., a non-delete key followed by the delete key. We omit `delete→delete` digraphs as we observe that students frequently press delete many times in succession to remove large sections of code. During the last week of programming, when students are ostensibly at their peak of typing skill, error recovery time is still slower than when typing in natural language. The Python/English mean drops from 447 to 406 (\(U = 3.23e^8, p < 0.0001, r = 0.10\)) and the Java/Finnish mean drops from 427 to 390 (\(U = 1.13e^8, p < 0.0001, r = 0.12\)). Only digraphs with latencies between 50 and 750ms are considered, so fixing mistakes after a compilation does not play into these results.

Regarding our second hypothesis, that mean latencies would drop with practice, Figure 7 shows that, at least in the Python context, this appears to be the case, although little drop is observed.

In the Python context, the mean latency drops from 465 to 447 (\(U = 1.24e^8, p < 0.0001, r = 0.06\)) from the first week to the last week. There is only weak evidence of such a drop in the Java context, dropping from 430 to 427 (\(U = 3.59e^8, p = 0.027, r = 0.01\)).

5 DISCUSSION

5.1 Context and Typing Speed

While students, in general, tend to type faster in natural language than when programming, this does not hold true for all digraphs, and the word or programming construct in which the digraph is found also plays a role in a student’s ability to type it fluently. A possibility, beyond just whether the word is familiar, is that a digraph’s position in a word or construct has an effect. For example, in Figure 2 we see that in Python’s `print` function, all digraphs but the first one are between 10 and 30ms faster when used in the...
context of print, while p→r shows little difference. Whether this is due to cognitive load when starting to type print or to complexities of other constructs containing pr is unclear, though the latter explanation is assumed to play at least some role since other constructs, notably range (Figure 2), do not reflect this behavior.

Context has previously been shown to have an effect on keystroke based identification accuracies [39] and predicting students’ success based on keystrokes [13]. Comparing our results to those studies, we too found that context plays a part in our analysis. For example, students improve in their typing of constructs more in the Python course even though the period they practice is shorter time-wise. Similar to Peltola et al. [39], our analysis also suffers from having less data from one context (essay writing) than from the other context (programming assignments).

### 5.2 Evolution of Typing Speed

Our results indicate that speed of typing digraphs depends very much on context, both at the language and at the word levels. In Section 4.3.1 we pointed out that the i→n digraph, when restricted to a single word context, is improved on much more in Python than Java. That is, i→n in Python’s print() improves 10 ms over 80 uses (using the regression coefficient from Table 3, −0.119·80) while the same digraph in Java’s String shows no improvement (calculated similarly). One possible explanation is that, since Java has a richer syntax, improvement for Java students is slowed by the larger number of syntactical constructs to learn. Another possible explanation is that, since Python is similar to English (the native spoken language of most students in that course), students are able to borrow schemas from natural language and thus improve more quickly. A third possibility is that the Java students enter the course with more programming literacy than the Python students, thus improving less.

Within the same language, a digraph may show differing improvement and large differences in initial latencies across two constructs. The digraph n→t shows this behavior between the print() and the int() contexts in Python: as can be seen in Figure 3b and Table 3, n→t is typed roughly 20 ms slower in the int() context than in print(), and over the course of 30 uses it improves at different rates between the two contexts. One possible explanation for the differences in latency is the cognitive load (see Section 2.3) associated with converting strings to integers. The intuition is that printing something to the console may be easily understood by a beginning student, while using int() requires students to reason about string types, integer types, conversion between the two, and assigning the result to a variable. Furthermore, it may be possible that the location of n→t, early in int() and late in print(), may play a role, as procedural memory may become more prominent after students gain momentum in typing a construct similar to rollover behavior observed in previous studies [8]. As a result, improvement in procedural memory may be reliant on cognition of the larger-scale task at hand.

As we conjectured earlier, it is not enough to compare digraph latencies across languages, as we have shown that word contexts in the Python language affect performance improvements of a single digraph. This insight helps explain Figure 1, how the improvement in the programming language context from first to last week is mixed – certain difficult constructs may have been taught later in the semester, lowering the performance for digraphs in that construct.

### 5.3 Mistakes and Correcting Them

We have shown that error recovery in natural language is faster than when programming. For students who are novices to programming, writing an assignment may impose a higher cognitive load than writing an essay. In this case, less cognitive resources are left to process sensory information and spot errors. Leyman et al. [34] report the lowering of typing accuracy when cognitive load rises.

We also hypothesized that error recovery would improve with practice, and we presented strong evidence that this is the case, at least in the Python context. The Java context, however, showed only weak evidence of a very small improvement (see Section 4.4.2 and Figure 7). Why the Python context showed improvement while the Java context did not is unclear. It is possible that the fact that Python is similar to English, while Java is not similar to Finnish, plays a role, but further study would be needed to explore this effect. The fact that the effect is very small for both Python and Java is somewhat easier to explain: throughout the semester students are continually introduced to new constructs, and so their typing encounters new challenges. Had the students spent the entire semester practicing just a moderate amount of new syntax then we would expect to see larger gains.

### 5.4 Implications for Practitioners

At a practical level, we have presented evidence that students become more adept at typing code by motor skills practice. Nevertheless, even after the gains shown in this study, students still do not achieve the level of fluency they have in typing natural language. One candidate approach to achieving fluency would be increasing the amount of practice by using exercises directly targeted at syntax, although empirical results are mixed [12, 14, 18, 30]. Another approach would be to include revising the difficulty of the tasks and instructional design since both impose cognitive load which might have affected the typing speed [34, 56].

Our data shows that the speed of writing syntactic constructs varies, and suggests that the speed of writing the constructs is related to the complexity of those constructs – not only in terms of the characters used as suggested by Thomas et al. [52] – but more specifically in the operations that those constructs are related to. This is visible, for example, in the relative typing speed of n→t in the print statement and casting a string to an integer using the int command. In other terms, our results support earlier results that suggest measuring the writing process using keystroke data can be used as a proxy for cognitive load [34]. This could bring additional support to existing approaches for measuring cognitive load (e.g. [37]) and approaches for measuring complexity of computer programs (e.g. [11]).

### 5.5 Implications for Researchers

Our work is beneficial to both theoretical and empirical researchers. We have presented results consistent with context-dependent procedural memory theory, the existence of schemas and schema transfer, and cognitive load theory. Additionally, our results suggest that
context matters when analyzing keystroke data. For example, based on our finding that the digraph latencies of students improved more in the Python context, it is possible that identifying students based on typing [26, 35] works better in some contexts (where less change happens) than others.

There remain many open questions: why exactly do digraph latencies behave differently in different constructs, what caused students in the Java course to improve in their typing more slowly than in the Python course, and similarly, why did Python students improve in error detection and recovery more quickly than Java students? We suggest that a good starting place would be a qualitatively driven enumeration of factors contributing to digraph latency (e.g., position in word, word frequency, digraph prominence in other words, language, etc).

5.6 Threats to Validity
Cognitive load, which depends on familiarity with the material and task difficulty [49], affects typing performance [34]. In this study, we did not control this factor which could potentially explain the difference between typing improvements in the context of Python and Java programming languages. We believe that in future studies controlling for prior experience and knowledge could reveal new correlations and allow tracking the progress more precisely.

We did not consider the location of the keys on the keyboard in the analysis, which could affect the speed of typing different digraphs. Additionally, there are many other factors such as the keyboard layout, whether the person typing is using all fingers in the process or e.g. only their index fingers, whether the person is touch typing, and even the handedness of the person typing, all of which might affect the results, and might have different effects in the different contexts studied here.

Another threat to validity is that we only consider students who continued to the end of the course. This introduces possible selection bias as stronger students are more likely to persevere through to the end. Additionally, the amount of data we have from the natural language contexts (Finnish and English) is less than from the programming contexts (Java and Python), which could affect comparisons between those contexts.

6 CONCLUSIONS
In this article, we presented an analysis of students’ typing in programming language and essay writing tasks. The analysis used data from two separate contexts, one of which uses English as the teaching language and Python as the programming language, while the other uses Finnish as the teaching language and Java as the programming language. Building on theoretical frameworks related to memory from cognitive science and empirical results from computer science education, we empirically showed that context-dependent memory and procedural memory (esp. muscle memory) may contribute to learning programming. As a summary, our research questions and their answers are as follows.

RQ1 How does context influence the typing speed of computer source code and written language? Answer: On average, students are faster at typing when writing essays (in English or in Finnish) than when writing source code (in Python or in Java), even though the keywords of the programming language are in English. Within the programming language, we observed differences in the typing speed of the same character pairs in different programming language constructs. For example, moving from the character ‘n’ to ‘t’ was faster when typing a simple ‘print’ command than when using the ‘int’ command to cast a variable to an integer. We hypothesize that the typing speed of digraphs in different programming language constructs may be related to the relative cognitive load imposed by the semantic meaning of that construct.

RQ2 How does the typing speed of programming language constructs evolve over time? Answer: Students tend to become faster at typing programming constructs over time, although there exists constructs where no difference is observed after tens of uses. Moreover, noticeable improvements can be identified in the use of special keys such as the TAB. This finding is in line with previous research that has suggested that typing data can be used to measure students’ previous programming experience [33]. At the same time, we also observed that the evolution of typing speed should not be studied from digraphs alone, but that the syntactic constructs in which those digraphs take place should be taken into account.

RQ3 How does the number of typing mistakes while programming evolve and how does the speed at which students correct mistakes differ between programming and writing natural language? Answer: Our results show that mistakes with syntax reduce over time, indicating that students become more fluent at writing syntactically correct code over time. When considering the speed with which students correct their mistakes in natural language and programming language, we observe that students are on average faster at fixing their mistakes when writing natural language than when programming. While we observe an improvement in the speed with which students erase mistakes in Python, the speed with which students erase mistakes in Java does not seem to evolve over time.

Our work has implications for practitioners and researchers. For practitioners, our results suggest ways to non-intrusively measure students’ cognitive load and syntactic fluency from keystroke data, giving instructors additional resources to design and evaluate curricula. For researchers, this work opens up new research directions in the use of keystroke data in computing education research and practice, such as modeling the amount of practice that individual students or groups of students need to become proficient with syntactic constructs, which could be then used as input to systems that control and offer additional practice opportunities for students. In addition, studying how students improve in fixing their mistakes in different contexts could shed light to our findings, where we observed little to no improvement with which the students learning Java fixed their mistakes. Such results could lead to further evidence on the applicability of particular languages for novice programmers as well as lead to context-specific interventions that focus on identifying mistakes.

REFERENCES


