Investigating Students' Programming Plan Knowledge with Time-Constrained Code Recall Tasks

Ava Heinonen
Aalto University
Espoo, Finland
ava.heinonen@aalto.fi

Reetta Puska
Aalto University
Espoo, Finland
reetta.puska@aalto.fi

Francisco Castro
New York University
New York, USA
francisco.castro@nyu.edu

Juha Sorva
Aalto University
Espoo, Finland
juha.sorva@aalto.fi

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

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

***Abstract*—This full research paper focuses on understanding programming plans – knowledge of recurrent code patterns that accumulate as programmers gain experience. This knowledge enables them to read code not as a sequence of tokens, but in terms of patterns they recognize. As a method for studying the development of plan knowledge, *time-constrained code recall tasks* have been put forward. In such a task, students attempt to memorize and reproduce code they have been briefly shown. However, there is little research on how students' plan knowledge is reflected in their performance on time-constrained code recall. To investigate this, we assigned a code recall task to 27 student programmers: they viewed a code snippet for 15 seconds and attempted to reproduce it from memory. We analyzed which parts of the code they recalled accurately, the order in which they wrote their code, and the code they wrote that diverged from the shown snippet. We found evidence of plan recognition and the use of plan knowledge in reproducing code. Our results suggest the presence of two types of plans: grammatical and task plans. We also see some evidence of reasoning and using higher-level understanding of the code's purpose to reproduce code, indicating code recall could be a process of reconstruction rather than replication. Our findings provide insight into students' plan knowledge, and how it is used in code comprehension and generation. Furthermore, our findings demonstrate that time-constrained code recall tasks can provide insight into students' plan knowledge.**

***Index Terms*—Programming education, Code comprehension, Cognitive modeling**

I. Introduction

A key element of programming expertise is knowledge of recurrent problem types and their generic solutions [15], [24]. In the literature on the psychology of programming and computing education, such patterns are often referred to as *plans*. The importance of plans for reading and writing programs is well established [4], [7], [8], [12], [16], [20].

As mental representations of plan knowledge cannot be observed directly, researchers have used various indirect means to study it. One such method is *code recall tasks* [1], [4], [9], in which people are asked to reconstruct a program from memory or perform some other task that requires them to remember a previously seen program. Plan knowledge enables people to comprehend and memorize code faster and more accurately [3], [4], as one may process programs in terms of a small number of complex but familiar patterns rather than a large number of simple tokens [1], [4], [7], [9].

Recently, Heinonen and Hellas [13] proposed *time-constrained code recall* as a means for monitoring the evolution of students' plan knowledge. In such a task, students are shown a code snippet for an extremely short time and then prompted to replicate the code. Their study found the tasks promising for demonstrating the growth of programming knowledge over time. However, their study was conducted in an online setting with little control over who the participants were; moreover, the study focused only on the end result, i.e., any correct code that the participants eventually succeeded in producing.

We seek to further understand the interplay between plan knowledge and time-constrained recall, focusing on university students. Besides the end result, we look at *intermediate recall* during the code replication process. Moreover, we qualitatively explore the 'recalled' code that diverges from what the students were expected to replicate. Three research questions guide our work:

RQ1 Which parts of the code shown to them do students recall?
RQ2 In which order do students write code when doing a code recall task?
RQ3 What code do students write that is not in the original code?

Our theoretical framework is grounded in cognitive theories related to programming expertise and plan knowledge. Specifically, we build upon Soloway and Ehrlich's conceptualization of programming plans, which describes how experienced programmers organize code into structured patterns or 'plans' facilitating comprehension and recall [20]. This framework suggests that as programmers become more experienced, they transition from seeing code as a series of individual tokens toward perceiving it as collections of familiar, meaningful patterns [14], [21]–[23]. We align our research questions explicitly with this perspective, investigating not only which aspects of code students recall (RQ1), but also the order and process of recall (RQ2), and deviations potentially indicative of reasoning and higher-level comprehension (RQ3).

II. Background

A. Expert-Novice Differences

Expert–novice differences have interested researchers for decades. Early research on chess players famously showed that experts, unlike novices, are able to memorize board configurations after viewing them for very short periods of time [11]. The same has been shown to be true in a wide range of domains [19], including programming. Foundational studies in programming demonstrated that experts memorize and recall code more quickly and accurately than novices due to their ability to structure information into meaningful chunks or patterns, known as programming plans [1], [3], [4]. For example, Adelson [1] and Davies [9], [10] have established that experts quickly identify abstract coding patterns, leading to efficient memory encoding and recall.

These cognitive strategies are linked to experts' deeper, more structured mental representations, in contrast to novices, who often perceive code in fragmented, unstructured ways, which taxes working memory more heavily [5], [10]. The differences in part relate to learning patterns – even experienced programmers perform at novice levels if the code is not presented in a meaningful order [2]–[4].

B. Conceptualizing Programming Plans

Programming plans have been conceptualized across studies. Soloway and Ehrlich [20] describe plans as structured blueprints that guide programmers in comprehending and generating code. This idea aligns closely with Von Mayrhauser and Vans' [22] description of plans as mental models or templates that represent typical code constructs, assisting programmers during software comprehension and maintenance.

Recent reviews of cognitive models in programming [14], [21], [23] further emphasize the dynamic nature of plan knowledge, suggesting that programmers frequently shift between abstract task-level strategies and specific code-level implementation details to effectively solve problems [6], [7].

C. Studying Plan Knowledge

Researchers have employed various indirect measures to investigate programming plans, among which code recall tasks are particularly prominent [1], [3], [4], [9] – this line of research is based on the idea that a person's plan knowledge can be deduced, to some extent, from their recall performance [1], [3], [4].

Code recall tasks require participants to memorize and reproduce code snippets, thereby revealing the structured mental representations underlying programming comprehension [9], [13]. Recently, Heinonen and Hellas [13] introduced time-constrained recall tasks specifically aimed at capturing how plan knowledge evolves, highlighting that the temporal constraints help emphasize cognitive processes involved in initial code comprehension and subsequent recall.

However, prior studies often focused on final recall performance, without exploring the intermediate steps or the processes involved in code reconstruction. Our current study expands upon the approach of Heinonen and Hellas by examining the construction process. Specifically, we analyze the order in which students recall code components and investigate deviations or 'divergent code' generated during recall, thus providing richer insights into the dynamic utilization and elaboration of programming plans.

D. Expanding Perspectives on Code Recall

Previous literature has largely overlooked the cognitive reasoning processes that students employ during code recall. Castro and Fisler [6], [7] emphasize that understanding novice programmers requires exploring how they balance high-level task strategies and low-level code details. They argue that novice programmers exhibit varying levels of abstraction and implementation focus throughout the problem-solving process, suggesting that recall tasks should also capture reasoning processes, not merely rote memorization.

In our study, we address this gap explicitly, analyzing the cognitive reasoning evident in recall tasks, such as participants' use of conceptual knowledge and their logical reasoning about code semantics during recall. This expanded perspective, integrating cognitive psychology insights about memory mutability and reasoning [19], enables a more nuanced understanding of novice programmers' cognition and offers actionable implications for instructional practices in programming education.

III. Methods

A. Participants

To recruit participants, we advertised our study in a programming course at a large European research university. The course, taught by one of the authors, introduces beginner programmers to object-oriented and functional programming using the Scala programming language. The course is mandatory for students of Computer Science, Engineering Physics, and a few related majors; many students from other majors also take it. Our study took place in late 2023, near the end of the 5 ECTS[1] course. The participants received a movie ticket as compensation. Altogether, 27 students participated in the study. The first experiment was a trial run to check that the experimental platform works as it should. The data from the trial run were included in the analysis.

[1]European credits. One ECTS credit is equivalent to roughly 27 hours of study.

Participants filled out a questionnaire to report on their prior programming experience. All participants rated themselves as having at least beginner-level skills in the Scala language. 15 participants indicated that this was their first formal programming course; 17 indicated that they had at least beginner-level knowledge and experience with at least one additional programming language; 16 knew Python. Five participants reported knowing languages other than Scala and Python. Only three students rated themselves as advanced users of any programming language (two of whom claimed advanced knowledge of Scala); no participant claimed to be an expert user of any language. As per the local national guidelines, no ethics review was required, given that the students volunteered to participate, and the study was based on informed consent.

B. Procedure

Students individually participated in one-on-one research sessions outside class hours that were led by a researcher who was not part of the teaching staff. Within these research sessions, participants were shown code within a set amount of time and were then asked to replicate it. In the first part of the sessions, participants completed a practice task to familiarize them with the study platform and code recall tasks. The practice task was very similar to the main code recall task described below but featured a different code snippet to memorize.

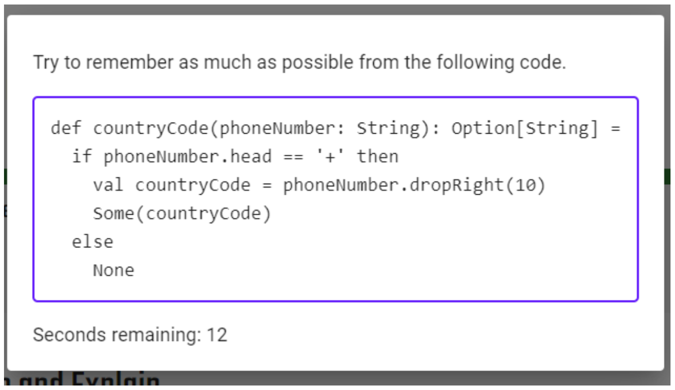
After the practice task, participants began the main task. They viewed the code shown in Figure 1 on-screen for 15 seconds and were then immediately taken to a code editor to replicate the code to the best of their ability. They were instructed to replicate the code verbatim. The editor automatically stored the students' code at each keystroke with timestamps.[2]

C. Materials

The code for the main task defines a function that takes in a phone number (a `String`) and, if the input starts with the plus character, extracts a country code from it (defined here as all the characters in the input string except the last ten). Despite its short length, the code uses various language constructs and standard library methods: there is a function, a parameter, an `if` expression, a local variable, optional values[3], some basic string manipulation with `head`, `dropRight`, and various types of data, and an explicit return type annotation (which is not mandatory, as Scala infers the type if omitted). The code also features the common patterns (plans) of checking input with an `if` expression and returning either `Some` or `None`, depending on whether a result is obtainable.

[2]The analysis presented in this article is based on the data collected via the editor. We also collected verbal data by having students explain their recall process. (Half of them did this as a think-aloud while writing into the editor, and the other half during a post-task interview while viewing a playback of their own editing process.) However, those data remain to be analyzed and are not discussed further here.

[3]Scala's `Option` is comparable to, for example, the `Optional` types in Java and Python. The `Some` class and `None` singleton extend `Option`.


Try to remember as much as possible from the following code.

```
def countryCode(phoneNumber: String): Option[String] =
  if phoneNumber.head == '+' then
    val countryCode = phoneNumber.dropRight(10)
    Some(countryCode)
  else
    None
```

Seconds remaining: 12

Fig. 1: A screenshot from the study platform. The code snippet used for the main code recall task is shown.

The program's domain (phone numbers) does not feature in the course the participants were taking, but the code is structurally similar and conceptually analogous to various functions the students had seen and written. All the specific constructs had been introduced, and most of them had appeared in many programs. However, `Char` literals (in single quotes) had not featured often, the `dropRight` method had been used much less frequently than `drop` (which drops from the left), the `head` method had usually been used on collections other than `String`s, and explicit return types had been omitted in many example functions during the course.

D. Analysis

Our analysis methods were explicitly chosen to answer the research questions as follows: for RQ1, we examined final recall accuracy; for RQ2, we compared midpoint and final recall to explore recall evolution; for RQ3, we investigated what was added erroneously (divergent code) to provide a qualitative perspective on student recall errors. While these methods have some limitations (as discussed below), they align with our goals to capture both accuracy and process in time-constrained code recall.

For RQ1 (*Which parts of the code shown to them do students recall?*), we first applied the Needleman–Wunsch algorithm [18], a sequence alignment method initially developed for bioinformatics. This method aligns two sequences character-by-character, identifying matches and mismatches to determine optimal alignment. We chose the Needleman–Wunsch algorithm because it is sensitive to both syntactic and structural recall (e.g., correct ordering, accurate reproduction of syntax) in short sequences of code. However, we recognize that it equally weights all characters, which may obscure semantic similarities when syntax differs. Alternative representations – such as token-based or AST-based alignments – might better capture structural correctness. Future work could explore these representations to complement our character-level analysis. In part due to this, we supplemented our analysis with qualitative reviews of divergent code (RQ3).

Needleman–Wunsch was used to find the optimal alignment between the program text that each participant wrote and the original program from Figure 1. We then computed, for each character in the original code, the percentage of students who had correctly recalled the character (i.e., the student code had the same character in the same place, when the two programs are optimally aligned). By examining this automated character-by-character analysis, we manually sought to find multi-character fragments that tended to be recalled well or poorly.[4]

For RQ2 (*In which order do students write code when doing a code recall task?*), we considered the relationships between what the students recalled, the order in which they recalled the code, and the constructs and patterns present in the given code. This analysis was similar to the above: we computed character-level recall percentages after aligning the student code with the original. However, for this question, we used not only the student's end product but also two intermediate solutions captured during the student's editing process. One of the intermediate solutions was captured halfway between the timestamp when the student started editing and the timestamp when they finished; the other was captured at the midpoint in terms of keystrokes without considering any timestamps. The differences between the two intermediate solutions were minor, so we report only on the keystroke-based midpoint.

While the selection of the midpoint was arbitrary (half of the keystrokes or time duration), it provided a standardized way to sample an intermediate snapshot of the recall process across participants. We did not base this midpoint on natural pause points or cognitive pauses, which might have reflected more authentic transitions from recall to reasoning. We acknowledge this limitation and suggest that future studies could use clustering or pause-detection approaches to identify more meaningful breakpoints in the recall sequence.

For RQ3 (*What code do students write that is not in the original code?*), we collected the 'divergent' code fragments from the student programs. That is, given an optimally aligned pair of a student program and the original, we collected any student-written code that did not appear in the original program (what might be called 'miss code' in contrast to 'hit code'). We manually grouped these by construct (return type, variable name, etc.). The purpose of this analysis was to qualitatively explore what students incorrectly add during time-constrained recall, both to help explain the statistics for RQ1 and RQ2, and to identify potential reasoning and higher-level comprehension.

IV. Results

A. RQ1: What Was Recalled

Figure 2a shows, for each character in the given code, the percentage of participants who correctly recalled the character.

The original program consisted of six lines of code. The average per-character recall accuracy for each of the six lines was, in order: 82%, 73%, 50%, 41%, 67%, and 72%. We also computed averages for each key pattern and construct; these appear in the middle column of Table I and are summarized below.

[4]Due to the low number of participants with extensive prior knowledge of programming, we do not analyze how prior knowledge affects recall in this article.

TABLE I: Recall accuracy of patterns and constructs in the final code, and at the midway point. The percentages are per-character averages.

Code	Final Acc.	Midw. Acc.
Key terms related to key patterns		
`def ( ) =`	92%	83%
`if then else`	77%	23%
`Option Some None`	70%	34%
Line 1 key constructs		
`phoneNumber`	72%	68%
`String`	80%	75%
`Option`	94%	80%
`String`	69%	48%
Line 2 key constructs		
`.head ==`	79%	20%
`phoneNumber`	64%	25%
`+`	81%	19%
`' '`	11%	4%
Line 3 key constructs		
`val countryCode =`	32%	14%
`phoneNumber`	50%	20%
`dropRight`	51%	13%
`10`	70%	0%
Line 4 key constructs		
`Some`	49%	10%
`countryCode`	24%	8%
Line 6 key constructs		
`None`	68%	13%

1) Recall of Line 1: The first line of code defined a single-parameter function. In Scala, such a function definition follows the form `def` *<name>*`(`*<parameter-name>*`:` *<parameter-type>*`):` *<return-type>* `=`. The unchanging key elements, `def`, `()`, and `=`, were recalled at a high average accuracy of 92%. The parameter `phoneNumber: String` was recalled at a 76% accuracy, with the parameter's name `phoneNumber` recalled somewhat less often (72%) than its type (80%). The function's return type, `Option[String]`, was recalled at a 75% accuracy overall, but within that code fragment, the average for `Option` was 94%, in contrast to only 69% for the type parameter `String`.

2) Recall of Lines 2 to 6: The rest of the lines formed a conditional expression of the form `if` *<condition>* `then` *<expression1>* `else` *<expression2>*. The keywords `if`, `then`, and `else` were recalled at an average accuracy of 77%, with `if` and `then` on Line 2 recalled more often than `else` on Line 5.

The conditional expression on Line 2 checks if the string parameter (`phoneNumber`) starts with the character `+`. The average accuracy of the whole line is 72%. However, the two keywords related to the if expression - `if` and `then` were recalled at an average accuracy of 90%.

The `if`'s conditional consisted of a `head` method call on

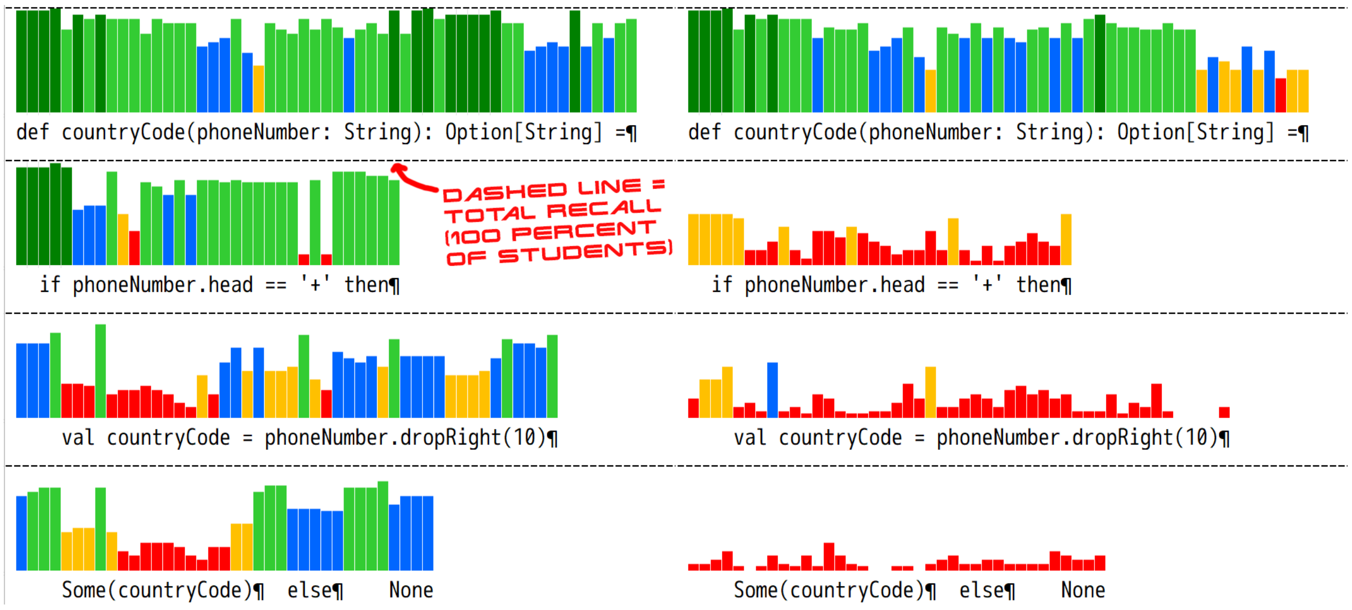


Fig. 2: Recall accuracy as the proportion of students who recalled each character for (a) the final submitted code and (b) halfway through the process. Each bar directly corresponds to the character below it and represents the percentage of students who correctly recalled the character. The character ¶ denotes a newline.

the parameter, and an equals check against a `Char` literal. The `head` call and `==` were recalled at an accuracy of 79%, but the parameter name at only 64%. For the plus character in the literal, accuracy was high (81%), but the rate for the single quotation marks around it was dramatically lower at 11%.

Line 3 dropped the ten last characters from the input and stored the result in a local variable. Here we had the `dropRight` method, the number of characters to be dropped as an `Int` literal, and a new variable `countryCode`. Only 32% of the participants recalled the fragment `val countryCode =`, but the name `phoneNumber` had a higher average accuracy of 50%. For the method name `dropRight`, the average was 51% overall, but the name's beginning (`drop`) was recalled considerably more often than the rest (`Right`): 59% vs. 44%. The literal `10` was recalled by 70% of the participants.

On Line 4, the value of the local variable `countryCode` is wrapped in a `Some` object. The average recall accuracy of the word `Some` was 49%. The average recall accuracy of the word `countryCode` was 24%.

The last two lines form the `else` branch. The `else` keyword was recalled correctly by 58% of the participants, and `None` by 68%.

Across Lines 1, 4, and 6 we also have the key elements of the `Option` `Some`-or-`None` pattern, which had an average recall of 70%.

B. RQ2: Plan Knowledge and Order of Recall

Figure 2b displays, for each character in the snippet, its recall accuracy at the midpoint of the writing process. The third column in Table I shows the average recall accuracy of the key terms related to the patterns and constructs at this midway point.

We also calculated the average recall accuracy for each line of code at the midway point. In order by line number: 74%, 25%, 17%, 8%, 1%, 11%.

The recall accuracy for the program's first characters, `def country`, did not change at all between the midway point and the end, indicating that they were not modified after the midway point. Similarly, our analysis shows that the recall accuracy for some individual characters did not change after the midway point. These characters are `(`, `o` and `t` on the first line of code, and `o` and `n` on the 4th line of code. On Lines 2, 3, 5, and 6, the accuracy of each character changes after the midway point.

Some characters were not recalled by anyone at the midway point: `(10)` on line 3, the `o` in `Some`, `Co` in Code, and the `)` on Line 4.

C. RQ3: Divergent Code

Divergent code refers to code fragments written by the participants that did not actually appear in the code they had been shown. Common types of divergent code were added whitespace (e.g., wider indentation), using different identifiers, and changes in letter case. Also very common (in more than half the programs) was to use double quotation marks (a `String` literal) instead of single ones (a `Char`). Other types of divergent code included the following:

- The return type was `Option[Int]` (7 programs) or `Option[number]` (1). In one program, the `String` input was converted with `toInt`.

- The parameter was an `Int` (5) or `Array[Int]` (1).
- A different `String` method was introduced (5): `take` or `takeRight` instead of `dropRight`, or `contains` instead of `head`.
- The `else` branch *also* yields a `Some` object (3). In one of these programs, the function's return type was also explicitly marked as `Some[String]`.
- Operators and/or brackets used incorrectly (2).
- The literal `9` used instead of `10` (2).
- The local variable defined as a `var` (a non-final variable) rather than a `val` (a final variable) (2).
- A new value assigned to the parameter (instead of introducing a new local variable) (1).
- `val` used where `def` needed (1).
- The `+` character written in the `then` branch, not the conditional.

V. Discussion

A. Plan Knowledge and Plan Recognition

Our results provide evidence of two distinct types of programming plans utilized by novice programmers: *grammatical plans*, representing standard grammatical patterns of programming languages, and *task plans*, representing recurring algorithmic solutions.

1) Grammatical plans: In the code used in this experiment, we can note three grammatical plans that the participants were able to recognize: the function definition, the `Some`-`None` plan, and the `if` control structure. Important words related to these plans were recalled with very high accuracy – in fact, the majority of the participants recalled words such as `def`, `Option`, `if` and `then`.

However, the details of this specific instantiation – such as the names of the method and the parameter, the type of the parameter and the `Some` return value, and the specific condition in the `if` construct – were recalled with lower accuracy. These elements were less accurately remembered than the plan-indicating keywords themselves. This suggests that while the participants may have been able to recognize that a plan such as the `if` construct was present, the details of how it was used in this situation were not necessarily understood in the short 15 seconds that participants had to study the code. In particular, the name of the parameter was recalled with rather low accuracy.

When programming, method and parameter names typically differ between programs, so recalling them accurately would require rote memorization of specific words, which was not the case here. This observation resonates with previous work on novice programmers, highlighting that while novices may grasp general programming structures, the ability to recall context-specific details often develops later.

2) Task plans: Our analysis of the recall accuracy of lines 3 and 4 is most indicative of the use of task plans. These lines implement two tasks, extracting the country code from the string given to the function as a parameter, and returning it wrapped in a `Some` object. The task of returning the value of a calculation from a function or passing it to another function is a common one. It can often be implemented so that the return value from one function is directly passed to another without saving it to a local variable first. This is the case here as well. These tasks could be implemented as one line of code that wraps the return value from the `dropRight` method directly to the `Some` object. In the code shown to the participants, the result of the `dropRight` method was first saved to a local variable, and then the local variable was wrapped into a `Some` object.

We acknowledge that the relatively high frequency of deviation from the presented code, particularly for `dropRight` and `Some` may raise questions about whether students were genuinely unaware of the material. Our interpretation is that these divergences do not reflect a total lack of knowledge, but rather illustrate active recall and reasoning under pressure. Many students substituted functionally plausible constructs (e.g. `take` instead of `dropRight`, or numeric types for strings), suggesting that while they had encountered core constructs, the less emphasis or practice on specific forms (such as `Char` literals or `dropRight`) led to construction rather than rote recall. The fact that many deviated in some way from this portion is thus better explained as evidence of variable familiarity and memory-based reconstruction, not a fundamental failure of instruction.

3) Grammatical and Task plans in light of literature: Overall, in the literature, there are different definitions of plans. Von Mayrhauser et al. described plans as templates, blueprints that represent the typical form or structure of a code construct [22]. These guide the comprehension process by providing the structure into which new information about a program can be slotted [22]. In this view, a plan can represent a code construct, such as a loop or a data structure, that has a generally agreed upon syntax and semantics. Other researchers have classified this type of knowledge of recurring code constructs or concepts, and their implementation, as mental models of programming concepts and knowledge of their syntax [14]. The grammatical plans we identified correspond with this definition of plans.

Others have understood plans as typical action sequences in programming—reoccurring algorithms for solving problems [20]. This view understands that plans are dynamic in nature, as ways to solve a programming problem or achieve some goal in programming [14]. The task plans we identified correspond with this definition of plans.

B. Replication Process as Plan Elaboration

Our results show that the replication process was quite linear. At the midway point, the first line of code was recalled with 74% accuracy, after which the recall accuracy declined. This indicates that participants started from the first line of code. The accuracy of the very first characters of the code also did not change between the midway point and the end, indicating participants did not revisit those after the midway point.

However, our results show evidence of non-linear replication processes as well. On line 3, where the code saves the country code into a variable, the parameter given to the `dropRight` method was not replicated by any of the participants by the midway point. However, other parts of the line are recalled between 13% and 20% on average by the participants indicating that some participants were already working on the code on this line. Participants also had code on lines 4, 5, and 6 recalled by the midway point (between 8 to 13% on average). The parameter given to the `dropRight` method is, by the end, recalled correctly by 70% of the participants. This indicates that participants did go back to add this detail later on.

One explanation for this is that participants focused on the bigger picture first, laying down the main elements of the code first, before going back and adding specific details such as the parameter to the method. This could stem from the recognition of plans and using them to structure the code during the replication process. Already by the midway point, the main keywords of the `if`-pattern were recalled accurately by 23% of the participants. Key terms related to the `Option` pattern, `Option`, `Some`, and `None` are also recalled by some participants at the midway point, with `Some` being recalled by 10% and `None` by 13% of the participants. This indicates these main blocks could have been laid down first before moving on to elaborating on the plans to build the complete solution.

In line of this possible elaboration, the absence of specific characters at the midway point – for instance, the number `10` or the closing parenthesis in `dropRight(10)` – should not be taken at face value as an indicator of lack of understanding. Rather, it highlights a sequencing behavior in recall: students may initially scaffold the broader plan structure before filling in specific parameters or syntactic details. This staged elaboration is consistent with the notion of top-down cognitive processing, where high-level constructs are prioritized early. Thus, the absence of characters mid-process provides insight into recall order and the likely use of memory anchors, not just omission.

Overall, we view this as a manifestation of what Castro and Fisler [7] describe as a *cyclic* pattern of programming where students move between high-level plans and code-level components of programming solutions. Their work, however, used think-aloud descriptions by students alongside observed programming patterns to identify code implementation patterns; we plan to further support our own observations of code recall patterns with student-reported descriptions in future iterations of this work.

C. Reasoning and Task Knowledge in Plan Elaboration

The analysis of divergent code fragments revealed students' active cognitive reasoning processes. Participants substituted semantically plausible constructs (e.g. replacing single quotes with double quotes, or using alternative `String` methods instead of `dropRight`). One of the more common types of divergent code was also defining the type of the `Option` or the parameter as a number instead of a string, which highlights understanding of a phone number as a series of digits instead of exactly remembering the parameter.

Such errors imply that students were not merely recalling memorized code verbatim, but actively reasoning about possible implementations based on their existing knowledge and expectations about the task. This finding resonates with cognitive psychology literature emphasizing the reconstructive and mutable nature of human memory, especially under time constraints [19]. This mutability has been also studied extensively in the context of eyewitness testimony, where individuals often modify or reinterpret memories when recalling them later [17]. Similar to how a witness might revise their memory after hearing new information, students in our study appeared to revise or reconstruct recalled code fragments based on inferred functionality or prior knowledge, rather than solely reproducing what they had seen.

For computing education practice, these findings highlight the importance of explicitly teaching students about different forms of plan knowledge, both grammatical and task-oriented. We see room for instructional interventions that target students' reflective reasoning skills, encouraging learners to critically assess alternative implementations and anticipate typical programming pitfalls.

As concrete examples, educators might benefit from intentionally exposing students to diverse code implementations of common tasks, thereby strengthening their conceptual and syntactical understanding and reducing reliance on rote memorization. Exercises that prompt students to reimplement known patterns in unfamiliar contexts may support the transition from passive recognition to active construction, while plan-based code templates and reflection prompts could help make these mental models more explicit in early instruction.

D. Limitations

This study comes with several limitations. Firstly, the participant sample, while representative of students in an introductory programming course at a large European university, includes varying degrees of prior programming experience. Although all students reported at least beginner-level proficiency in Scala, the potential influence of previous programming exposure – particularly in syntactically similar languages – cannot be fully discounted. Notably, over half of the participants also reported knowledge of Python or other languages. Given that methods like `dropRight` have equivalents in other languages, it is plausible that recall was influenced by transfer from prior experience rather than purely from instruction in Scala. Variability in prior knowledge may have influenced participants' code recall performance, introducing differences unrelated to the plan knowledge targeted by this study. Future work could stratify participants by language background to isolate such effects.

Secondly, our use of only a single code snippet limits the generalizability of our findings. While the selected snippet was carefully chosen for its structural similarity and complexity, the insights derived from this single example might not be applicable to all programming constructs or languages. Future studies would benefit from employing multiple and diverse code snippets to validate the robustness of our observations across varied contexts.

Additionally, our methodological reliance on the Needleman–Wunsch algorithm, which works at the character level, is another limitation. Although effective for assessing recall accuracy, this approach may undervalue semantically equivalent yet syntactically divergent responses, leading to potentially underestimating participants' recall or understanding. For perfect alignment, a student would need to recall not what the program does, but how; one who recalls the syntax but not the functionality would be adjudged to have better recall than another with a functionally equivalent but different program. That is, our numbers do not speak directly about plan knowledge, and our interpretations are subject to debate. Some of the specific numbers that we got are artifacts of how the algorithm works. For example, Figure 2a shows a peak for the `n` in `Option[String]`, because some students wrote `Int` instead of `String`, and `n` is the first character that appears in both.

Lastly, capturing code recall at arbitrary midpoint timestamps may introduce ambiguities in interpreting the recall process. Although the chosen midpoint was consistent in terms of keystroke volume, a more nuanced approach – perhaps capturing recall after natural pauses – might offer deeper insights into the participants' cognitive processes during recall tasks. That is, while our granularity enables fine-grained tracking of recall fidelity, it can also obscure higher-level understanding and lead to potentially misleading interpretations – especially in cases where semantically meaningful units (like parameters or operators) are split across characters. For example, noting that no participant recalled a specific part of code at midway could be interpreted as either a trivial delay or a more meaningful omission, but this cannot be determined from character presence alone.

Despite these limitations, our study provides valuable preliminary insights into how novice programmers leverage plan knowledge under time constraints, highlighting areas for future research and potential instructional strategies in programming education.

VI. Conclusion

In this work, we used time-constrained code recall for studying code recall performance of (n = 27) students recruited from a university-level introductory programming course. Students in the experiment were shown code for 15 seconds, after which they were asked to recall and write it using a code editor.

Overall, our results support the notion that when being shown a piece of code to memorize, students did not only rely on line-by-line reading and rote memorization of the text of the code. Our results show evidence that students used general programming plans (both *grammatical* and *task-related*) in their code recall efforts. However, the evidence for detailed application or understanding of these plans is partial, particularly in lines requiring deeper semantic processing or less familiar syntax. While both types of plans have been discussed in the literature as plan knowledge, and they both represent knowledge of common patterns, our results show differences in how they are used in code recall tasks. Differentiating the two types of plan knowledge may therefore be important for future research.

While the code writing process was overall linear, with the first lines of the code being recalled the most accurately at the midway point, we see some evidence of plan elaboration during the code writing process. We see some evidence that the key terms of the plans are written down by the midway point, while some details of the code such as the number given to a method are not recalled by any participant by the midway point.

Our analysis also shows some evidence that participants used reasoning and demonstrated higher-level understanding of the tasks present in the code to reason what was present in the code they had seen. This indicates the formation of a higher-level understanding of some aspects of the code, and a process of implementing a solution matching this higher-level understanding rather than writing down a memorized set of characters.

Acknowledgements

This work was in part supported by Research Council of Finland grants #356114 and #367787.

References

[1] Beth Adelson. Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, 1981.

[2] Woodrow Barfield. Skilled performance on software as a function of domain expertise and program organization. *Perceptual and Motor Skills*, 1997.

[3] Woodrow Barfield, Woodrow Barfield, and Woodrow Barfield. Expert-novice differences for software: implications for problem-solving and knowledge acquisition. *Behaviour & Information Technology*, 1986.

[4] Allan G. Bateson, Ralph A. Alexander, and Martin D. Murphy. Cognitive processing differences between novice and expert computer programmers. *International Journal of Human-computer Studies International Journal of Man-machine Studies*, 1987.

[5] Allan G Bateson, Ralph A Alexander, and Martin D Murphy. Cognitive processing differences between novice and expert computer programmers. *International Journal of Man-Machine Studies*, 26(6):649–660, 1987.

[6] Francisco Enrique Vicente Castro and Kathi Fisler. Balancing act: A theory on the interactions between high-level task-thinking and low-level implementation-thinking of novice programmers. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, page 295, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Francisco Enrique Vicente Castro and Kathi Fisler. Qualitative analyses of movements between task-level and code-level thinking of novice programmers. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 487–493, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Francisco Enrique Vicente G. Castro. Investigating novice programmers' plan composition strategies. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, page 249–250, New York, NY, USA, 2015. Association for Computing Machinery.

[9] S. P. Davies. The nature and development of programming plans. *International Journal of Human-computer Studies International Journal of Man-machine Studies*, 1990.

[10] Simon P. Davies. Knowledge restructuring and the acquisition of programming expertise. *International Journal of Human-computer Studies International Journal of Man-machine Studies*, 1994.

[11] Adriaan D de Groot. Thought and choice in chess. 1978.

[12] Françoise Détienne and Elliot Soloway. An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies*, 33(3):323–342, 1990.

[13] Ava Heinonen and Arto Hellas. Time-constrained code recall tasks for monitoring the development of programming plans. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 806–812, 2023.

[14] Ava Heinonen, Bettina Lehtelä, Arto Hellas, and Fabian Fagerholm. Synthesizing research on programmers' mental models of programs, tasks and concepts—a systematic literature review. *Information and Software Technology*, page 107300, 2023.

[15] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, et al. Fostering program comprehension in novice programmers-learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, pages 27–52. 2019.

[16] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. Through (tracking) their eyes: Abstraction and complexity in program comprehension. *ACM Transactions on Computing Education (TOCE)*, 22(2):1–33, 2021.

[17] Elizabeth F Loftus and John C Palmer. Reconstruction of automobile destruction: An example of the interaction between language and memory. *Journal of verbal learning and verbal behavior*, 13(5):585–589, 1974.

[18] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[19] Giovanni Sala and Fernand Gobet. Experts' memory superiority for domain-specific random material generalizes across fields of expertise: A meta-analysis. *Memory & Cognition*, 45:183–193, 2017.

[20] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on software engineering*, (5):595–609, 1984.

[21] M-A Storey. Theories, methods and tools in program comprehension: past, present and future. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 181–191. IEEE, 2005.

[22] Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[23] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 40 years of designing code comprehension experiments: A systematic mapping study. *ACM Computing Surveys*, 56(4):1–42, 2023.

[24] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J Ko. A theory of instruction for introductory programming skills. *Computer Science Education*, 29(2-3):205–253, 2019.