Abstract

High-school students specialising in computing fields need to develop the abstraction skills required to understand and create programs. Novices’ difficulties at high-school level, ranging from mastery of the “notional machine” to appreciation of a program’s purpose, have not yet been investigated as extensively as at undergraduate level.

This work explores high-school students’ code comprehension by asking to reason about reversing conditional and iteration constructs. A sample of 205 K11–13 students from different institutions were asked to engage in a set of “reversibility tasklets”. For each code fragment, they need to identify if its computation is reversible and either provide the code to reverse or an example of a value that cannot be reversed. For 4 such items, after extracting the recurrent patterns in students’ answers, we have carried out an analysis within the framework of the SOLO taxonomy. Overall, 74% of answers correctly identified if the code was reversible but only 42% could provide the full explanation/code. The rate of relational answers varies from 51% down to 21%, the poorest performance arising for a small array-processing loop (and although 65% of the subjects had correctly identified the loop as reversible).

The instruction level did not have a strong impact on performance, indicating such tasks are suitable for K11, when the basic flow-control constructs are usually introduced. In particular, the reversibility concept could be a useful pedagogical instrument both to assess and to help develop students’ program comprehension.

Keywords

Program comprehension; Novice programmers; High school; Flow-control constructs; Iteration; SOLO taxonomy

ACM Reference Format:


1 INTRODUCTION

While learning to use basic programming constructs, students manage to write working programs but they appear to lack a more comprehensive grasp of the overall computation carried out by each block of code. In other words, many students do not fully develop the abstraction skills — and perhaps accurate enough mental models — that would allow them to reason about and interact with code. Several studies report problems and misconceptions even for such basic flow-control constructs as conditionals and loops in the context of tertiary education. Kaczmarczyk et al. [13], for instance, identified “a number of misconceptions all related to an inability to properly understand the process of while loop functioning”, and Cherenkova et al. [6] found that “students have significant trouble with conditionals and loops, with loops being particularly challenging”.

Up to now, however, novices’ difficulties with basic programming concepts have received far less attention in high school contexts. Vahrenhold et al. [37], in their recent broad literature review of K–12 computer science research, reported only eight studies addressing a variety of programming topics specifically for the upper secondary level. Our own literature review of ISSEP and WiPSCE conference proceedings, combined with keywords search in the ACM digital library, found no more than a dozen papers about high school learning of flow-control constructs and recursion.

In this paper we address high-school students’ ability to understand the behaviour of small code fragments by looking at how they deal with the concept of reversibility, i.e. the possibility to undo a state transformation in every conceivable situation. A relevant feature of this approach is that, to cope with reversibility, the behaviour of program constructs must be understood as a whole, by analysing the interactions of the constituent parts in different potential execution flows.

A sample of 205 students, attending the last three years of secondary instruction in different institutions that offer a specialisation in computing topics, were asked to engage in a set of reversibility tasklets. Following a short explanation of reversibility in the test sheet, students were asked to decide if each given (tiny) program can be reversed or not, as well as to justify their answer by writing the reversing code (Yes option) or by providing suitable counterexamples (No). Thus, the test covers both code reading — at an abstract level — and code writing abilities. It is also worth remarking that the involved students had no experience at all of similar tasks in class, so ensuring their higher-order thinking skills, instead of simple recall, were put into play in the endeavour. For four of the six items assigned in the test, namely two conditionals and two iteration constructs, we extracted the recurrent patterns in the answers and then we analysed the justifications within the framework of the SOLO taxonomy [1].

In essence, here we will address the following research questions:

RQ1 – Can students grasp the state transformation enacted by simple conditionals and loops in an accurate and comprehensive way?
The complete text of the reversibility task, as well as the recurring patterns and the SOLO classification of students’ justifications will be available from the corresponding author, XX, upon reasonable request.
The statements in the branches of the two conditionals are very simple and straightforward to reverse in isolation. The key insight to decide if the code can be reversed is that there are different computation flows whose outcomes may “overlap”, as shown in Figure 2 for item (b), and in order to find potential overlaps students have to identify border computations. A more explicit overlap occurs in the while loop (c), since all computations starting from $x \geq 0$ result into a final value in the range $[0, 4]$. Hence, such overlap is easier to detect as it does not involve borderline analysis.

Finally, tasklet (d) is more cognitively demanding: students have first to realise that the for loop implements a cumulative sum on a vector; then that the vector’s original state can actually be recovered; finally, they have to write a nontrivial reversing program requiring mastery of the dependencies between the operations carried out at subsequent iterations.

The rationale inspiring the design of the proposed task can be better appreciated in terms of the Block Model (BM) [30], a useful framework to map programming concepts and activities into a matrix representing three program dimensions, namely text, execution behaviour and purpose, as well as four abstraction levels: atomic elements, blocks, relationships between blocks and overall macrostructure, see Figure 3. Typical tracing tasks, for instance, can be achieved by analysing code behaviour at the atomic level (A), as pointed out in [12], whereas “explain in plain English” (EiPE) tasks pertain to the top-rightmost cell which implies the deepest understanding of a program and its purpose. It is also worth observing that while the first two columns are concerned with intrinsic program features, the third one is about extrinsic properties in connection with some application domain.

Now, to begin with, all tasklets (a–d) essentially pertain to the program execution dimension (P: middle column in Figure 3), i.e. with properties intrinsic to the code but requiring a viable mental model of the underlying notional machine. Thus, to know if a program is reversible there is no need to figure out its function. In addition, the analysis of each item can be done at a different (minimal) level of abstraction: tasklet (c) possibly can be achieved by simply tracing the code in a few cases (AP), since the probability of getting “conflicting” outputs is quite high; a reversal of code (a) requires considering the if block as a whole (BP), but can be written even without having actually considered borderline computations; to see that code (b) cannot be reversed, on the other hand, such borderline computations must be identified (RP), what implies that the if branches cannot be analysed independently of each other; finally, a comprehensive grasp of the behaviour of both the given program (MP) and the reversing code is a prerequisite to provide a solution for tasklet (d).

### 3.2 Data collection

We administered the test on reversibility at the end of the school year to a sample of 205 students of the age range 16–19 from (11 classes of) two technical schools, located in two different areas of the country, that offer a specialisation in computing topics. All the students had completed at that time an introduction module of imperative programming, which is scheduled in the third high school year. 101 students were finishing third year (K11), 52 fourth (K12), and 52 their last year of high school (K13). Each collected sheet was then anonymised and digitised prior to analysis.

### 3.3 Data analysis

The reference framework for the core part of our analysis is the SOLO taxonomy [1], an instrument of widespread use to assess code reading and writing tasks, e.g. [21, 33, 38], which is deemed to provide “a means of evaluating cognitive or mental models, to see if the novices are able to make connections between what they have learnt.” [33]. From this perspective, the learners’ achievements are classified in terms of complexity and quality of the interrelationships between parts they are able to deal with.

After measuring the percentages of correct options, the analysis has been carried out as follows. As a first step, a researcher identified and listed the recurrent patterns of code or justification in students’ answers by an inductive content analysis process [23]. Next, a second researcher revised the patterns, possibly adjusting the list by splitting some patterns into more refined ones or merging pairs of essentially equivalent patterns, and assigned a SOLO level to each pattern according to the guidelines summarised in Table 1. During this process, the same researcher checked more than half of the students’ sheets via deductive content analysis (see again [23]), based on the set of identified patterns, as well as classified into
SOLO categories all isolated (non-recurrent) answers which were left unprocessed from the previous steps.

Finally, a third researcher reviewed the categorisation of justification patterns and discussed with the second researcher a few cases where different possible perspectives had emerged. In particular, they eventually agreed that tasklet (c) could be fully achieved at the unistructural SOLO level, so that insights to motivate higher SOLO levels could hardly be found in the students’ answers. In addition, further discussion resolved minor issues and led to the final classification presented in section 4.

It is worth noting that all items except tasklet (c) are in fact relational in that students need to consider non-trivial relationships between parts. Tasklet (c), on the other hand, is unistructural since it makes only sense to consider the loop as a whole: isolating either the condition or the inner assignment would be meaningless. To identify overlaps between final states for the straightforward while in tasklet (c), students could opt for describing it in words, providing two or more counterexamples, e.g. via tracing, or attempting and seeing that any basic code manipulation will fail to retrieve the original value. Any other answer that fails to describe such behaviour is classified as prestructural.

Table 1 does not include the extended-abstract SOLO category, since it does not match with the features of the tasks at hand. It can however be noticed that the proposed tasklets, although simple, require abilities at the relational level that cannot be taken for granted in the high school.

4 RESULTS

In this section we outline the results of our investigation, as well as providing sample answers to better describe the range of SOLO levels and mental models identified in the analysis.

Overall, 74% of the chosen options about code reversibility are correct, but only 42% are also supported by sound justifications, either at SOLO relational level for items (a), (b) and (d), or at the unistructural level for item (c). Table 2 summarises the main figures concerning the selected options as well as the quality of students’ justifications. As it is customary [32], SOLO means are determined by averaging over the weights assigned to each level: 4=relational,

3=multistructural, 2=unistructural, 1=prestructural, and 0=null, i.e. either empty or meaningless justification.

4.1 Analysis of conditional tasks

The first two tasks are relational in that students have to identify if there is an overlap in values resulting from the two update paths (if and else). Tasklet (a) has no overlap, hence is reversible and tasklet (b) has an overlap for borderline values 0 and –1.

Tasklet (a). Most students (more than 90%) rightly conjectured that code (a) is reversible, but only a little more than half of them (55%) were also able to write correct reversing code, while one third provided instead code pertaining to the unistructural SOLO level, usually an instance of pattern Uni1 (28% overall, see Figure 4) that lacks awareness of the interrelationships between statements in the if branches and if condition. The two most “popular” code patterns are shown in Figure 4; other recurrent patterns (one more relational, two multistructural, two more unistructural and two prestructural) are far less frequent.

Tasklet (b). 37.1% of students identify the overlap and clearly describe it as in the following excerpt from an instance of pattern Rel:

“If x’s initial value is either –1 or 1, then the result will be 0…”

As the overlap is small, it is possible to write code that reverses all values except 0 or –1, as shown in Figure 5 for pattern Mul1. Such code reflects a good understanding of reversibility and ability to code. However, it assumes the final value 0 was originally “–1” which is plausible but ignores the other potential original value of “+1”. In comparison, pattern Uni1 does not adjust the the condition of the statement. Although both of them fail for a small number of cases, they show a different level of comprehension.

The third item in Figure 5 (Uni2) is another interesting recurrent pattern that seems to indicate a quasi-syntactic manipulation of the code, where both the condition as well as the operations in the if branches are reversed, resulting into a program that fails to undo the state transformation in almost all the cases.

We speculate that students that answer Yes to both (a) and (b) may be failing to check the relationship between the parts and treating them separately. In fact, more than one third of the students choose the Yes option for both items (a) and (b).
4.2 Analysis of iterative tasks

As mentioned before, the first iterative task (c) is unistructural while task (d) is relational. Thus, we will discuss them separately.

Tasklet (c). Based on the characterisation in section 3.1, tasklet (c) appears to be the least cognitively demanding among the considered ones. And as many as 60% of students were indeed able to identify the (large) overlap relative to the output state.

Examples of the most common acceptable justifications in words are: "We cannot know how many times the loop repeats" (pattern Uni1, 31%); or "The final state is the same, x = 0, for any multiple of 5." (Uni2, 12%). Moreover, 9 students attempted to write the code shown in Figure 6, pattern Uni5, which led them to conclude that the reversal was not feasible.

Most of the 25% prestructural answers reported transformations to the while statement that mimic the transformations performed for items (a) and (b), by editing both the variable update and the condition as shown in pattern Pre1. It appears none of these superficial edits were tested; in fact, 18 students wrote code giving rise to no iterations or infinite loops in a large number of relevant cases (patterns Pre2 and Pre3, the latter being shown in Figure 6). Finally, 15% of the answers were empty.

Tasklet (d). As expected, tasklet (d) turned out to be quite challenging: although 65% of the students had correctly conjectured that the loop is reversible, only one third of them, i.e. 21% overall, managed to write a correct reversing program and about 10% wrote code at loop is reversible, only one third of them, i.e. 21% overall, managed to write a correct reversing program, requiring mastery of the aforementioned dependencies and where the role of x, if used, must be kept consistent. (COD)

As to the last two points, DEP and COD, each original value for index i is now v[i–1]–v[i–1]; as a consequence, the update needs to consider data dependencies and either (i) update the elements from last to first using a downward loop (pattern Rel1 in Figure 7) or (ii) preserve the partial sum at i–1 while proceeding upwards to the next step (pattern Rel2 in Figure 7). Based on the experience of Kumar and Dancik relative to the high school, “down-counting loops are more difficult for computer science students than up-counting loops” [16]. However, the main difficulty is not to write a correct downward loop but to realise the need to reverse the loop direction.

A potential reason why students’ code fails to manage the dependencies between subsequent iteration steps is that a variable, usually x, plays conflicting roles in the loop body. Consider, for instance, the code fragment for the Mul1 pattern (see Figure 7). While we could infer from the use of a downward for the student was aware of such dependencies, x is used in two different roles: in the right-hand side of the first inner assignment it is implicitly meant to hold a partial sum, whereas in left-hand side and in the next statement its value is supposed to be the restored original value of v[i]. Incidentally, in terms of the roles-of-variables pedagogical model [4], the above roles would be gatherer and temporary, respectively.

Students’ answers reveal the following combinations of the above features, each related to a corresponding SOLO category:

- More importantly, students have to be aware of the dependencies between actions at subsequent iterations, e.g. by choosing a smoother processing order. (DEP)
- Without DEP it is not possible to write correct code, but even when students are aware of the dependencies, they still have to write nontrivial reversing code, requiring mastery of the aforementioned dependencies and where the role of x, if used, must be kept consistent. (COD)

As we have identified tasklet (d) as challenging for K11–12 students, we consider students are working at relational level even when they skip the last task. Of the 50 students classified at that level, 27 were fully correct, 13 made a partial attempt relative to
Figure 5: Examples of partially correct code patterns, all alleged to be reversals of the conditional construct (b).

"It is not reversible because if with the original code $x = 15$ the outcome will be $x = 0$, but with the code I propose at the end of the loop $x$’s value will be 5."

Figure 6: Examples of recurrent patterns for tasklet (c), used to reason correctly (Uni5) or alleged to reverse it.

Figure 7: Examples of recurrent code patterns meant to reverse loop (d).
item (d) and 10 did not answer it. Most students work at adjacent levels, with less than 14% working below Uni.

Table 3: Use of downward vs. upward for loop to reverse code (d).

<table>
<thead>
<tr>
<th></th>
<th>correct</th>
<th>incorrect</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>upward loop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>downward loop</td>
<td>4</td>
<td>37</td>
<td>41</td>
</tr>
<tr>
<td>overall</td>
<td>43</td>
<td>65</td>
<td>108</td>
</tr>
</tbody>
</table>

Table 4: Students performance across all four tasks

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rel</td>
<td>(a) to (c) fully correct</td>
<td>24.4%</td>
</tr>
<tr>
<td>R/M</td>
<td>all correct except one being Mul or empty</td>
<td>10.7%</td>
</tr>
<tr>
<td>Mul</td>
<td>all answers are above Uni (excluding c).</td>
<td>12.7%</td>
</tr>
<tr>
<td>M/U</td>
<td>some answers above Uni, none below</td>
<td>8.3%</td>
</tr>
<tr>
<td>Uni</td>
<td>mostly Uni and one empty</td>
<td>15.6%</td>
</tr>
<tr>
<td>Pre</td>
<td>two entries Pre or empty, none above 2</td>
<td>6.8%</td>
</tr>
<tr>
<td>Empty</td>
<td>three of four empty entries</td>
<td>6.8%</td>
</tr>
<tr>
<td>Mixed</td>
<td>Non adjacent levels in non-empty answers</td>
<td>15%</td>
</tr>
</tbody>
</table>

Overall, 15% of students show significant differences in comprehension across the four tasks: for example, they worked at relational level with conditionals, but at prestructural level for one of the iterative tasklets. Only one student failed to see the overlap in tasklet (c), in fact the easiest endeavour, while was relational over other tasklets. Another got tasklet (b) correct but (a) only at unistructural level, while skipping (c) but answering (d) correctly.

4.3 Instruction level impact

So far, the analysis has been concerned with the whole cohort. We will now look at the performance for specific subgroups. As we can see from Figure 8, where are reported the average SOLO means covering tasklets (a), (b) and (d), the instruction level had a limited impact on performance. (Notice that tasklet (c) has not been considered because the relational and multistructural levels do not apply to it.)

A more detailed exploration of each item is provided in Figure 9. We can see there are only minor differences between the K11 and K12 cohorts, but the K13 cohort outperforms the rest by about 20% in tasklets (a–c) and of more than 10% in tasklet (d). It is conceivable that this is due to more coding practice and fluency, which means they can produce better code for tasklets (a) and (d), as well as a better ability to describe their reasoning for tasklets (b) and (c).

For the conditionals (a) and (b) at least half of the class are working at multistructural level and above. This indicates the task can be a good learning exercise at any level because it provides a desirable difficulty [2]: the task is challenging enough for students at relational level to revise and confirm their correct mental models, while students at lower levels who partially understand the task can learn from their mistakes. Similar considerations apply to the while loop (c), that appears to be fully understood by a little more than half of the K11 and K12 cohorts.

As expected, tasklet (d) is challenging for all levels, with only the K13 cohort reaching a 40% of answers at multistructural level or above.

5 DISCUSSION

In this section we will first revisit the three research questions, then discuss other interesting findings and then draw some conclusions and implications for educators.

5.1 Research questions

Based on the results reported in section 4, we will now answer each research question.

RQ1 - Can student grasp the state transformations enacted by simple conditionals and loops in an accurate and comprehensive way? If we focus on the first three items, we can see 70% of K13 students have a good grasp of conditional statements and simple while loops compared to 40–50% of their K11–12 counterparts. Similarly, students that attempted the last item exhibit a good comprehension of the code transformation in terms of describing the final state and its link to the original state. Hence, we can conclude that most students have managed a reasonable grasp of conditional and iterative statements.

RQ2 - When a statement is reversible, are students able to write correct code to recover the original state? The answer to this question needs to be qualified, due to the difficulty of item (d). The fact 20–30% of high school students completed this challenging task indicates their coding skills are above what is expected.

Most of those students used a downward for loop to undo the state transformation of tasklet (d), in spite of the fact that most examples, both from textbooks and teachers, are stereotypically upwards, hence several of them seem to have developed higher-order thinking skills.
On the other hand, there is some concern for the 30% of K11–12 students that made poor attempts to reverse the while loop, and seem to manipulate conditions and update statements without understanding their overall effect. This is best understood looking at patterns Uni5 and Pre1 shown in figure 6. The latter shows students do not reflect after editing the code on the limited scope of their transformation which will only reverse numbers in the range {5, 9}. Thus we can see ability to write code is evenly spread over three ranges:

- **high** (above average) their code is correct and well-thought as explained above for item (d).
- **medium** (average) for the range of students that can reverse conditionals, and their code is mostly correct.
- **low** (below average) for those that lack comprehension of the code they produce, even when is correct, such as in task (a).

There are also a small number of students (7%) that did not attempt to code or reason about at least three tasklets.

**RQ3 - Is student’s performance consistent over all tasks, reflecting their current level of comprehension? Or does their comprehension vary for each task example?**

Most student answers are mapped into the same or adjacent SOLO levels, as described in table 4 reflecting their current code comprehension level. However 15% show marked differences in performance across tasks. In particular, four students that completed task (d) failed to see the overlap in either items (b) or (c), while ten students that show the overlap in item (b), failed to see it in item (c). Thus, their comprehension is patchy. These students are probably representative of the group that would benefit most from code comprehension activities, as they would become aware of the inconsistencies in their solutions.

**Tasklet (d) depth.** The analysis of the answers relative to tasklet (d) was especially instructive because of the richness of its implications. When designing the task, we exhibited a *expert blind spot*: although we new the task to be challenging, we were not fully aware of all the aspects that could potentially emerge by examining the code provided by the students. This can be better appreciated by looking at the features identified in section 4.2 in terms of the block model framework.

As shown diagrammatically in Figure 10, students are required to work at different levels of abstraction in the program execution dimension. At the topmost macrostructure level they have to understand the overall behaviour of code (d) (reading comprehension) and write a reversing program (COD). The connections between
the original value of a vector element and the final value of two subsequent elements (SUB) and the treatment of dependencies (DEP) pertain to the relationships level.

<table>
<thead>
<tr>
<th>Macrostructure</th>
<th>Relationships</th>
<th>Blocks</th>
<th>Atoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>overall comprehension of code (d)</td>
<td></td>
<td>consistency of var roles within iteration step</td>
<td>tracing to test conjectures</td>
</tr>
<tr>
<td>MT</td>
<td>RT</td>
<td>BP</td>
<td>AP</td>
</tr>
<tr>
<td>MP</td>
<td>SUB</td>
<td>PRC</td>
<td>COD</td>
</tr>
<tr>
<td>MF</td>
<td>RP</td>
<td>BF</td>
<td>COD</td>
</tr>
<tr>
<td></td>
<td>RF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: Mapping of the features implied in tasklet (d) into cells of the Block Model.

A tentative outline of a loop processing all the vector elements (PRC) can be drafted by considering the looping block as a whole and neglecting details about dependencies, so working at the block level of abstraction. Similarly, although the roles of variables have their effect on the dependencies, possible inconsistencies can be detected by just looking at the block representing an iteration step. Finally, the possible use of tracing to test conjectures about program execution can be done at the atomic level.

5.3 Implications for instructors

The instruction level appears to have a limited impact on students’ performance, what suggests that reversibility tasks are appropriate for students attending the middle years of high school.

From a pedagogical perspective, designing “low-ceiling” reversibility tasks could be a useful instrument aimed at assessing and fostering students’ mastery of basic program constructs from K to 11 onwards. In particular, similar tasks could help to focus on the need to test program behaviour carefully, and may provide opportunities to examine code at different levels of abstraction and practice with varied success higher-order thinking skills.

Tasklet (d) depth presented a step challenge to students; on the other hand, such complex task allows to expose the links between reading, writing and testing and provides many lessons to be learned. Thus, to use this task as a learning event we should scaffold it, for example by decomposing it into two subtasks: (d1) explain the computation carried out by this small program and reason if is reversible or not; and (d2) write the code that reverse this computation (relative to a different code fragment of similar structure).

More teaching efforts should explicitly introduce methods to approach and analyse a programming task, in particular how to identify suitable test cases in order to check the conjectures on code behaviour.

In addition, instructing students to analyse the code in terms of variable roles could be helpful to let them detect, if not avoid, frequent mistakes, such as that illustrated by item Unit 1 in Figure 7 and commented above.

5.4 Future work and perspectives

This study has presented 4 tasklets to explore code comprehension in a comprehensive way. Other approachable tasklets should be developed exploring additional features of core programming constructs.

Future work will also expand on our implications for educators by (i) exploring ways to scaffold tasklet (d) by decomposing it into more focused subtasks, in particular to test separately code reading comprehension and code writing skills; (ii) testing these and related tasks with different cohorts and (iii) developing and testing guidelines for novice programmers.

6 CONCLUSIONS

In this paper we explored a promising perspective on code comprehension by asking high-school students to reason about reversing conditional and iteration constructs. We analysed students’ performance on four programming tasklets by extracting recurrent patterns in their answers and categorising their code and explanations within the framework of the SOLO taxonomy. We also analysed the structure of the considered tasks in terms of Schulte’s Block Model.

Vahrenhold et al. review of studies in computer science education at K–12 levels [37] concluded by inviting researchers to undertake multi-institutional and multinational projects “in the school sector” similar to those carried out within the academic milieu (e.g. [7, 19, 24]). According to Vahrenhold et al., in spite of “the significantly greater complications involved in the school sector,” we should nonetheless “aspire to similar studies in schools. This is a grand challenge for Computer Science Education Research.”

This paper is a first step in that direction as it covers students from different institutions. The reversibility tasks could represent a useful instrument for multinational projects in this respect, as this study could be replicated in different contexts (and we would be happy to collaborate with educators interested in trying similar investigations).

In the light of our exploration, we think that the reversibility concept could also be a useful pedagogical instrument to assess and develop students’ program comprehension in the classroom.

Acknowledgments

We are very thankful to the teachers who collaborated with us to administer the test to their students.

REFERENCES


