International Journal of Computer Science Education in Schools, Oct 2017, Vol. 1, No. 4
ISSN 2513-8359

Novice programmer's misconception of programming reflected on problem-solving plans

Kyungbin Kwon

Indiana University, USA

kwonkyu@indiana.edu

DOI: 10.21585/ijcses.v1i4.19

Abstract

Understanding the students' programming misconceptions is critical in that it identifies the reasons why students make errors in programming and allows instructors to design instructions accordingly. This study investigated the mental models of programming concepts held by pre-service teachers who were novice programmers. In an introductory programming course, students were asked to solve problems that could be solved by utilizing conditional statements. They developed solution plans using pseudo code including a simplified natural language, symbols, diagrams, etc. Sixteen solution plans of three different types of problems were analyzed. As a result, the students' egocentric and insufficient programming concepts were identified in terms of the misuse of variables, redundancy of codes, and weak problem-solving strategies. The results revealed that the students had difficulty designing solution plans that computers could execute correctly. They needed instructional supports to master how to express their solution plans so programs run as intended. Problem driven instructional designs for novice programmers were discussed.

Keywords: computer science education, misconception of programming, novice programmer, computational problem solving

1. Introduction

Computing is becoming a fundamental part of our daily lives, and it has changed our education system as well. Computing is not only developing program codes, but rather solving our problems in scientific and innovative ways. We solve various problems in effective and efficient ways that would not be possible without using or borrowing the power of computers. The problem solving requires scientific activities that include, designing systems, understanding human behaviors, recognizing patterns, generating principles, and so on (Chi, Glaser, & Rees, 1982; Newell & Simon, 1972). These critical, but not easily acquirable activities can be facilitated through "computational thinking" that emphasizes "Ways to Think Like a Computer Scientist" (Wing, 2006, p. 35).

Applying the computational thinking into one's subject area is critical to everyone. In this sense, Wing (2006) suggested that teaching students to think like a computer scientist is not only for computer scientist, but also for everyone. The slogan: "Computer Science for ALL" represents the current trend and need very well. President Obama also addressed the importance and urgency of computer science (CS) education in K-12 as follows: "In the coming years, we should build on that progress, by … offering every student the hands-on computer science and math classes that make them job-ready on day one" (Whitehouse, 2016).

However, the deficiency of CS education in K-12 is getting serious in the United States. Although 9 out of 10 parents surveyed want their child to learn computer science, only 40% of middle and high schools teach computer

programming (Google & Gallup, 2015). Stakeholders in many schools and district administrators do not perceive a high level of demand for CS education as students and parents do (Google & Gallup, 2015). Even when education administrators want to provide CS in K-12, there are not enough qualified teachers available (Google & Gallup, 2016).

The gender differences toward CS make the situation even worse. Usually women show less likelihood to learn CS than men. No conclusive reasons for the phenomenon have been suggested but female students show less confidence on CS and do not perform as good as male students (Rubio, Romero-Zaliz, Mañoso, & de Madrid, 2015). Considering that the majority of pre- and in-service teachers are women (Snyder, de Brey, & Dillow, 2016), the *unconfident* attitude toward CS among females should be considered in offering CS education for pre-service teachers or it will make the deficiency of CS teachers even worse.

The author designed a computer education course for pre-service teachers who were interested in having a computer educator license. Considering their lack of previous programming experience and low-confidence toward computer science that was observed in the classes, he emphasized computational thinking and integrated problem solving activities rather than focusing on program language features. By shifting the focus of programming education from coding to problem solving, he expected to lower the emotional and cognitive barriers (Chang, 2014; Tom, 2015).

Through the teaching practice, he realized the approach was effective. In addition, he identified several common mistakes among students while they made the plans for problem solving, such as misuse of variables, redundant codes, and ineffective problem-solving strategies. The mistakes were barely related to program language features, but associated with their conceptual understanding of problem-solving strategies and programming. Computational thinking emphasizes the importance of problem solving and programming concepts by utilizing computational concepts and practices. Thus, identifying misconceptions observed during problem-solving process will be beneficial to design CS education for pre-service teachers.

This paper aims to investigate the mental models of programming concepts held by pre-service teachers who were novice programmers. The purpose of the investigation is to identify pre-service teachers' knowledge, understanding, and problem-solving techniques adopted during the problem-solving processes. By revealing their cognitive deficit, this study will be able to suggest instructional prescriptions for computer science teacher education.

2. Literature Review

2.1 Evaluation of programming concepts

In order to evaluate a learner's mental model, we need a conceptual framework that represents types of knowledge required in order to complete a certain task. Cognitive scientists generally agree that there are three types of knowledge involved in computer programming: syntactic, conceptual, and strategic knowledge (Bayman & Mayer, 1988; Linn & Dalbey, 1985; McGill & Volet, 1997).

The syntactic knowledge is related to language features such as the grammar and format of statements that are defined differently across program languages. Assuming that you are writing a computer program using a text-based language such as BASIC or Python. You need to know, for example, how to write variables and assign values. You will use an equal sign to assign a value to a variable such as x = 7 in Python. In this case, the value should be located on the right side of the equal sign and the variable on the left side. A syntax error will occur if you reverse the order (e.g., 7 = x) as many novice programmers do (Ma, Ferguson, Roper, & Wood, 2011). Because the syntax rules are very strict compared to natural language, it is a quite challenging task to master them for novice programmers. General programming manuals and traditional CS education have placed the main attention on teaching the syntactic knowledge.

The semantic knowledge is about programming concepts that apply to program languages in general (Shneiderman & Mayer, 1979). Computer programming is a purposeful process to achieve a certain goal or solve a problem, which requires understanding the meaning of codes and their uses. For example, the syntax, x = 7 implies that the program stores the value (7) in the variable (x) and a programmer will use the variable for certain purposes. This is the sematic meaning of the syntax. To write a program that calculates the arithmetic mean of a data set, one will think the followings process: reading the data set, adding each value to calculate the sum of the given numbers,

counting the number of values, and apply a formula to calculate a mean. This process requires design skills that combine language features for a certain act (Linn & Dalbey, 1985). Conceptual knowledge includes semantic understanding, and design skills that are required to write the required code.

In various contexts, writing an effective and efficient program requires strategic knowledge that enables programmers to develop, interpret, evaluate, and debug programs (Bayman & Mayer, 1988). The strategic knowledge allows programmers to apply the syntactic and semantic knowledge to developing programs. Many novice programmers have difficulty designing a program to solve a certain task that requires strategic knowledge (Lahtinen, Ala-Mutka, & Järvinen, 2005). Winslow (1996) argues that "novice programmers know the syntax and semantics of individual statements but they do not know how to combine these features into valid programs" (p. 17).

2.2 Common misconceptions of novice programmers

The main reasons that novice programmers commit errors can be explained by their misconceptions. Pea (1986) identifies the common misconceptions of novice programmers in terms of "hidden mind" representing their faulty assumption that a computer is intelligent enough to guess the programmers' intention. He suggests three types of "bugs": parallelism, intentionality, and egocentrism bugs. Because they can provide a framework to analyze novice programmer's misunderstanding, I will introduce the types here and adopt them to evaluate students' mental model of programming.

Some novice programmers may assume that computers can execute (or even consider) multiple lines of codes in parallel although it runs each code line-by-line from top to bottom (called parallelism bugs). Pea (1986) provides the following example to explain the bug.

AREA = H × W	---- a
H = 10	---- b
W = 5	---- c
PRINT AREA	---- d

Novice programmers may say the answer is 50 in consideration of the assigned values in line b and c. However, computer will not return the result as they expected. People, as novices assume, can *remember* the rule defined in line a and *calculate* the AREA according to the values assigned in line b and c before printing the AREA at line d, *although there is no command* to calculate the AREA after assigning the values and before printing AREA. However, contrary to the novices' expectation or assumption, computers calculate AREA at line a *as instructed* and it uses the values of H and W at that moment. Therefore, the values assigned afterward do not affect the result in line d. Pea (1986) attributes the misconception to novice programmer's incorrect analogical reasoning that programming is similar to the natural language that allows this out-of-order process.

Novices may assume that computers can do what programmers intend (intentionality bugs) as well as mean (egocentrism bugs), but is not explicitly instructed. The intentionality bugs occur when a novice read program codes while considering a solution plan. For example, a student wanted to repeat a set of commands *three times* using a "while loop" that included a counter having an initial value that was zero. The counter was increased by one per iteration. The actual code said "repeat while counter is less than or equal to 3". He thought the loop would stop after three iterations and the counter would be three. His plan was correct and the counter was traced correctly. However, computers would run another iteration because the while loop ran until the condition got false rather than reach 3 unlike the student's intention.

Egocentrism bugs occur when novices omit necessary specifications in an assumption that computers will know what they mean. Pea (1986) explains that novices often use a same name for two different variables because they know the different roles of them while computers have no instruction about it. Egocentrism bugs also occur when novices design program only from their perspective. Mayer and Fay (1987) found that fourth-grade students interpreted the direction of moves and turns based on their orientation reflected on a screen rather than a turtle's orientation in the programming language Logo.

3. Method

3.1 Participants

Twelve students were recruited from an undergraduate course in a large Midwest university in the United States. They were pre-service teachers mostly majoring elementary education (11), senior (8) or junior (4), and female (11). All of them, except one student who had "little" experience, did not have programming experience before taking the course. They were not given compensation for their participation in the study. The study was approved by the University Institutional Review Board (#1701722036).

3.2 Context of Learning

The main purpose of the course was to introduce computational thinking and to teach programming concepts to novices who would possibly teach computer related subjects. In order to emphasize the importance of conceptual understanding of programming, the instructor provided problems that students could solve without a computer, but with paper and pencil. The problems contained conditions that students should consider (See Table 1). When receiving the problems, students were introduced to basic concepts of conditional statements and other required concepts not focused on syntax, but semantic meaning. Students were paired with a peer and asked to solve the problems using pseudo-code that allowed them to develop a solution plan with a simplified natural language, symbols and diagrams if necessary. This approach might reduce the students' cognitive load to consider syntax. Six pairs of students solved problem 1 and five pairs did problem 2 and 3 because a student was absent.

Table 1.Problems presented to the students

	Brief description	Condition	Complexity	Week
Problem 1 (calculator)	Carry out an arithmetic calculation in consideration of the size of two numbers.	Compare two numbers and find which one is bigger or smaller.	Low	3rd week
Problem 2 (T-shirt)	Suggest a color of a T-shirt in consideration of gender and the age of customer.	Consider two conditions: gender (female vs. male) and age (young vs. old)	Middle	5th week
Problem 3 (rock-paper-scissors)	Play rock-paper-scissors with a computer	Compare two selections of a user and computer and decide who wins.	High	8th week

3.3 Identifying misconception

To identify students' misconception of programming, the author analyzed their solution plans. The unit of analysis was a semantic unit including one problem-solving step or one decision-making point. For example, the following solution plan includes five units of analysis.

```
If x > y then                    --- unit 1: decision-making point
a = ((((x*y) + x) - y) / y)      --- unit 2: problem-solving step
else                             --- unit 3: decision-making point
a = ((((x*y) + y) – x) / x)      --- unit 4: problem-solving step
Print (a)                        --- unit 5: problem-solving step
```

4. Results

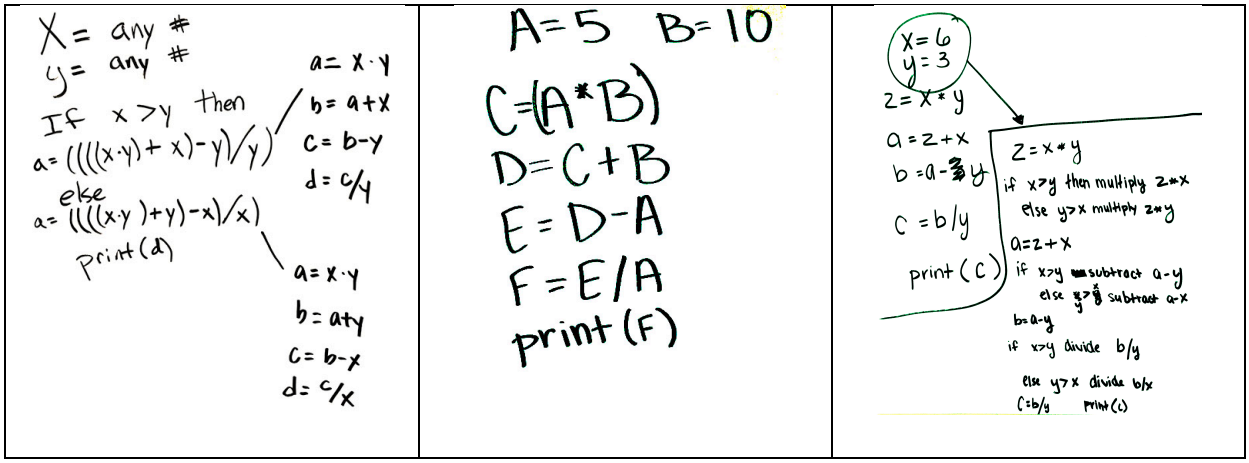
In the following section, students' solution plans of each problem were analyzed to reveal their mental models utilizing the concepts of programming and computational thinking to solve the problem.

4.1 Problem 1: calculator

The first problem was to carry out an arithmetic calculation as the instruction guided: step1) multiply two numbers and save the result, step 2) add the bigger number to the result, step 3) subtract the smaller number from the result, and step 4) divide the result with the smaller number. In this problem, students were informed that a data set had two different numbers. Although there were three chances to decide which number was bigger or smaller from

step 2 through step 4, only one decision-making process was required as shown in Figure 1-a.

To solve the problem, students should 1) use variables to store and calculate values and 2) decide the bigger or smaller number. Based on the cognitive task analysis, students' solution plans were evaluated. First, all the pairs of students used variables explicitly. However, there were differences in how the variables were used; two pairs assigned general numbers to two initial variables such as X = any #, Y = any # (See Figure 1-a); two pairs assigned specific numbers to the initial variables such as A = 5, B = 10 (see Figure 1-b); two pairs assigned specific numbers at first but changed the initial values to general numbers later after figuring out the solution process (see Figure 1-c).



a b c

Figure 1. Solution plans of problem 1

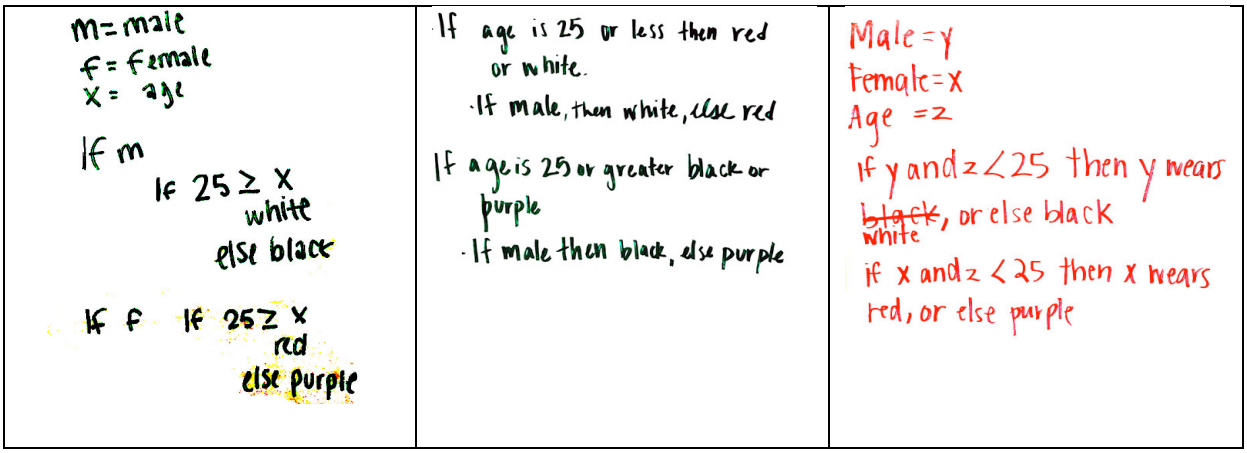
The result suggested that novice programmers liked to use a specific case to understand problem-solving process rather than consider a general solution at first. Although this approach seemed to allow students to understand the problem easily, it misguided them to ignore the important decision-making process. Three pairs who used specific initial values omitted the process and assumed, for example, a computer would know variable B (its value was 10) was bigger than variable A (its value was 5). Therefore, their solution skipped the decision-making process and jumped into the calculation process based on the specific case, which would cause a semantic error in a case when the values were switched. In contrast, the two pairs who did not assign specific values to the initial variables included the process in their solution. One pair understood the process by testing specific values and developed a solution that applied to general cases.

As explained, only three pairs developed conditional statements deciding a bigger or smaller number. Close examination of the statements revealed that there were meaningful differences. One pair repeated the same conditional statement three times whenever the decision was required (see Figure 1-c). The statement also revealed that the pair considered the two conditions (x>y as well as y>x) while using "else", which produced redundancy. In contrast, two pairs used the conditional statement once and carried out the following calculations based on the decision (e.g., Figure 1-a). Surprisingly, one pair condensed four calculations into one step based on the decision while the other listed each procedure sequentially as shown in Figure 1-a. The result suggests that there could be meaningful variations among novices' solutions in terms of the efficiency.

4.2 Problem 2: T-shirt

The second problem was to suggest a color of T-shirt in consideration of a customer's age (older than 25 or not) and gender (female or male). There could be four suggestions according to the two by two conditions. To solve the problem, students were expected to develop a nested conditional statement that contained multiple conditions. In this case, students also needed to use variables. As being revealed in problem 1, students assigned specific values to the variables, such as m = male, f = female, x = age, and used them in the decision making process (see Figure 2-a and Figure 2-c). Two pairs used values in the conditional statement without defining variables explicitly (see Figure 2-b). The misuse or no-use of variables had a negative influence on their solution, as a result. For

example, the statements, "m = male, If m then" was understandable to people, but omitted an important decision-making process that computers needed to tell whether a user was male or female. If the first statement had a variable named "gender" that saved "male" or "female" in it, the second statement could be "If gender is 'male'" or "If gender is 'female'", which instructed computer to carry out the decision-making process.



a b c

Figure 2. Solution plans of problem 2

Regarding the nested conditional statement, four pairs developed a decision-making process properly while committing unnecessary redundancy. One type of redundancy was to identify the result that was not confirmed (Figure 2-b). Examine the following statement. The result in the first line (a) was not necessary at that moment while it was not wrong.

If age is 25 or less then red or white --- a

If male, then white, else red --- b

Another type of redundancy was to specify the other condition while it was confirmed by previous condition. Examine the following statement. (Note students were informed that there were two values in the gender variable for the class practice.) The second statement (d) checked the user's gender unnecessarily which was confirmed by the first conditional statement (c).

If male --- c

…

else female --- d

One pair's solution revealed that students did not properly understand the nested conditional statement structure (Figure 2-c). Examine the following statement.

If y and z<25 then y wears white, or else black --- e

If x and z< 25 then x wears red, or else purple --- f

In this case, students used a logical operator (and) in the first part of conditional statement before the commas. Semantically the statements were valid. However their following underlined statements were not valid because the "else" did not mean the opposite condition but only reverse a part of the logical condition. Here the else actually meant z was greater than or equal to 25 while keeping the gender as it was. The solution plan included ambiguity that computers could not solve with the current instruction.

The analysis of the nested conditional statement revealed that students' solutions, expressed with pseudo-code, omitted necessary specifications based on a false assumption that computers would be able to follow the instructions that were quite ambiguous. On the other hand, students tended to express temporary or alternative results in the statement that were not necessary for computers to carry out the instructions. The results revealed the students'

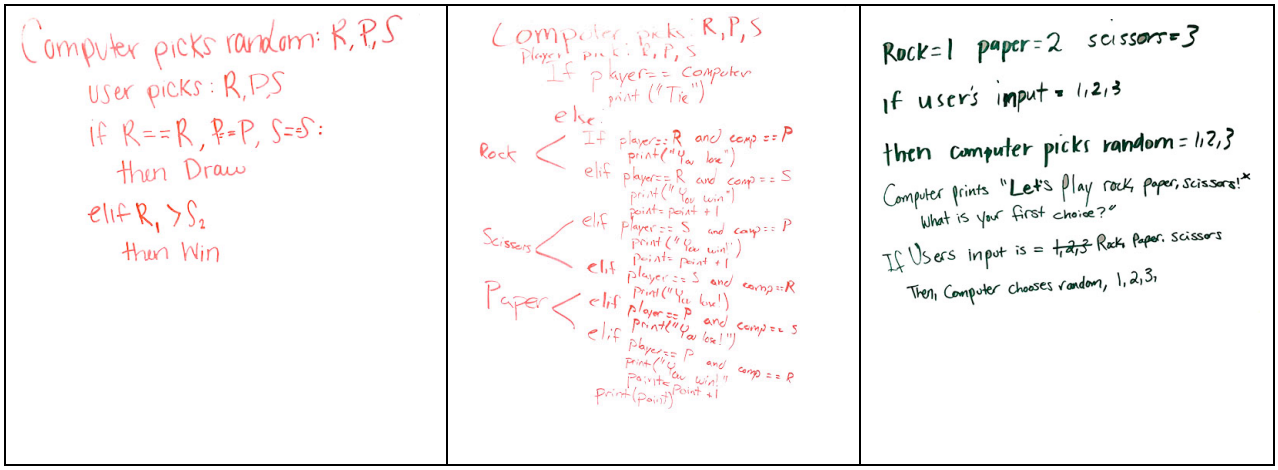
egocentric bugs (Pea, 1986).

4.3 Problem 3: Rock-Paper-Scissors

The third problem was to develop a program that played rock-paper-scissors. A user and a computer selected one of three options (rock, paper, or scissors) independently and the results (win, lose, or draw) would be presented from the user's aspect. To solve the problem, students were expected to 1) design a selection procedure of the options for a user and a computer and 2) consider all possible results according to the selections.

Regarding the first step, all pairs wrote the selection procedure of the two agents properly such as "Computer picks R, P, or S" and "User picks R, P, or S". However, there were two missing elements that were critical to develop a solution. One was to identify variables that would store the selections. The other was to specify that the computer would select an option "randomly". All pairs failed to identify variables, which affected the uncertainty of solutions in the following step. Considering that the problem was presented in the 8th week when students had learned the concept and usage of variables with considerable times of practice, identifying variables while planning a program was a critical bottleneck to master the programming concepts.

Some students had not recognized the concept of random selection. In order to examine whether students could realize the importance of random selection, the instructor mentioned that computer would "randomly" select one option but did not emphasize the concept while presenting the problem. As a result, three pairs did not explicitly mention it while two pairs instructed the random selection such as "Computer picks random: R, P, S". It would be probable that they overlooked its importance and assumed it was enough to instruct, "pick one".



a b c

Figure 3. Solution plans of problem 3

The second step required students to consider all possible combinations of selections that would result in win, lose, or draw. In sum, only one pair considered all nine cases properly (Figure 3-b) and other three pairs did parts of them (e.g., Figure 3-a). One pair could not develop the second step and was excluded from the following analysis (Figure 3-c). All four pairs considered the draw condition first and wrote an instruction for comparisons quite clearly. However, because they did not identify variables properly, there remained ambiguity as below.

If player == computer --- g

If R==R, P==P, S==S --- h

The two statements expressed that the options the user and the computer selected were same. However, it was not clear which values the statement was comparing (g) either which variable represented user's selection (h). One might think this ambiguity could be acceptable especially in the planning phase. However, the following statements raised more serious issues in deciding a winner.

elif $R_1 > S_2$ then Win --- i

if P and S are chosen the print ("S wins") --- j

Students used the mathematical symbol ($R_1 > S_2$) to express that rock won scissors (i). However, the analogical reasoning would cause an issue when developing a program code because it represented different meanings in programming. The analogical reasoning would be useful for students to express their intention, but they did not figure out how computers compare values in that case. Statement i and j also did not specify which variable belonged to the user and this resulted in the ambiguous verdict (S wins), which might cause a semantic error in a program.

5. Discussion

The study revealed the novice programmers' faulty mental models of computational problem solving. In the following sections, the author will elaborate the findings and suggest instructional design to handle the issue.

5.1 Misuse of variables

In all three problems, students consistently demonstrated the misuse or no-use of variables. In a computer program, a variable is declared to store a value that will be referenced and manipulated in the following codes. The fundamental function of variables is to save and update values, which allows a program to be more flexible and adaptable to general cases. The students in this study, however, tended to use variables to store specific values without considering their changes and developed solution plans based on the certain case, for example x = 5, y = 10. This approach allowed them to develop a solution that was valid for the case but not other cases. By using a natural language or a symbolic expression, they were able to initialize (set the initial values of) variables easily. However, this approach misguided them to assume their solution plans would be applied to general cases where the initialization varies. The result suggests that the students did not understand the function of variables and placed more attention on the specific values that the variables stored. This approach affected the quality of solutions in that students assumed computers would be able to evaluate the relation of variables and omitted the evaluation step in their solution plans.

This issue continued in the rock-paper-scissors problem. Students developed conditional statements that compared the selections of a user and a computer such as "if R is equal to R, P is equal to P, S equal to S". This example demonstrated that students paid attention to the values of variables and developed a solution plan based on the values rather than variables so the plan did not include variable names in it. Without informed variables, computers would have no clue which value belonged to which variable. These results reveal that students did not understand the purpose and ways to use of variables. Regarding the issue, Shi, Cui, Zhang, and Sun (2017) suggest that novice programmers encounter difficulty understanding a code and writing a program when they do not figure out the functions of variables.

Students also declared multiple variables according to the number of values such as "m" for male and "f" for female rather than using a variable (e.g., gender) that could store either value. Declaring multiple variables that store values that belong to one category produced unexpected problems. For example, in the T-shirt case, students should determine whether a customer was woman or man using a conditional statement. They assumed that computers would be able to tell the gender if they specified the gender in form of "if m" or "if f". The statement, "if m" implied that there was a decision-making process and the result was male. Although people can guess the process and execute the next step without specific instructions, computers do not have that ability. These results demonstrated how novice programmers committed egocentrism bugs while designing a solution plan (Mayer & Fay, 1987; Pea, 1986).

5.2 Redundancy of inefficient solution

Efficiency is an important issue in developing a program. From the efficiency perspective, committing redundancy can cause errors at worst or add more codes unnecessarily at best. The students' solution plans revealed the two types of redundancy. First, they checked a condition whenever it was required even though the result was consistent (problem 1: calculator). They also developed two conditional statements separately when the cases were mutually exclusive that could be replaced by "else" or "otherwise" (problem 1: calculator and problem 2: T-shirt). This type of redundancy did not cause an error, but increased the length of the codes.

The second type of redundancy had been observed in the nested conditional statements. The students wrote temporary results in a solution although they were not the final ones. If there were two nested conditions,

programmers could narrow down possible cases (temporary results) according to the results of the first condition and finalize the final result based on the second condition. It is reasonable for the students to reduce the cognitive load by specifying the temporary results after evaluating the first condition statement. However, this approach may produce a syntactical error because program languages do not allow the specifications of temporary results. The result suggests that students' mental model of programming can be different from the way a program executes.

5.3 Weak strategic understanding

As the problems got more complex, problem-solving strategies became more necessary. The third problem that required considering combinations of nine cases (3 computer × 3 user selections), revealed that students did not use problem-solving strategies properly. The rock-paper-scissors game involved a random selection in nature which students knew but did not explicitly write while developing solution plans. Here, the author paid more attention to their conceptual consideration of the random selection rather than their syntactic solutions. At the planning phase, students did not need to know what syntactical function they needed to use which would be required when developing a program. The critical issue at the moment was to identify what functions would be necessary to solve the problem. Without considering the requirements, students would encounter a higher cognitive load when developing codes, which might increase the probability of errors or result in problematic codes (Mayer, 1981).

Students' solution plans revealed the discrepancy between human thinking and computer program execution. People easily figure out that rock wins scissors and other combinations as well because they know the rules of the game and can decide the result of each case. However, a computer cannot execute the decision-making process without specific instructions. Identifying the rules that applied to the game was a crucial task in the planning phase. However, some students skipped the process by stating the logic in a simple symbolic expression (e.g., $R_1 > S_2$ that means rock wins scissors) that was not sufficient enough to instruct a computer to execute the decision-making process. To solve the problem, students needed to consider using the concept of logic comparison such as "if a user selected rock AND computer selected scissor". Students who overlooked or skipped this process did not elaborate the rules so that the computer could follow.

Results also revealed that students did not consider all possible cases in their solution plans. This was not related to the lack of syntactic or semantic understanding but problem-solving strategy. As discussed, if students did not consider solution processes or the functions required, they would have higher cognitive load when they develop codes because the missing (or ignored) elements would be an extraneous cognitive load to them at the moment.

The findings suggest that students should be able to (1) identify a solution process as they could solve the problem logically and (2) translate it to the way a computer executed the instructions within the given programming language. The first step required mostly strategic knowledge that was closely related to general problem solving ability (Chi et al., 1982). The second step could be carried out when they were able to think "within the domain of programming" and design plans in a way computers executed (Mayer & Fay, 1987, p. 270).

6. Conclusion and suggestions for instructions

This study identified the novice programmers' egocentric mental models of computational problem solving in terms of the misuse of variables, redundancy of codes, and weak strategic knowledge. Not surprisingly, the mental models were quite different from the ways computers ran a program, which would cause errors. The current study has paid attention to the quality of solution plans that reflected student's programming concepts rather than the specific knowledge of programming languages. Thus, the lack of syntactic and semantic knowledge, one of the most common issues of novice programmers (Brooks, 1990; Soloway, Bonar, & Ehrlich, 1983), was not the concern of this paper. Especially considering the fear of the students (pre-service teachers) regarding programming, the author released the burden by freeing them from writing codes.

Then, why did students encounter the difficulties in designing solutions? Winslow (1996) breaks down program problem solving into four steps: 1) understand the problem, 2) determine how to solve the problem, 3) translate the solution into computer language program, and 4) test and debug the program. The current study answers the question based on the second step: determine how to solve the problem. Winslow (1996) emphasizes that novices have trouble designing solutions that are compatible to computers. The results of the current study demonstrated that the students fully understood the problems and were able to design solutions in their mind. However, they had difficulty expressing the solutions or developing instructions in the way that computers could execute them.

What instructional strategies will be effective for novice programmers to have the expert's mental model that translates solution plans to a compatible program? Experienced programmers, for example, will be able to develop a solution that tests odd and even numbers by checking the remainder of the division of the numbers by two (design a solution plan). They even know program languages have the modulus operation that returns the remainder of one number divided by another (translate the plan to programming). In contrast, novice programmers may spend their cognitive resources to find the rule from their understanding that even numbers can be divided evenly into groups of two (design a solution plan). Thus, novices will have a higher cognitive load in *creating* the plan while experts *retrieve* the plan from their memory (Robins, Rountree, & Rountree, 2003). Considering the lack of knowledge schemas and the higher cognitive load in novices, more specific scaffoldings in problem-solving activities are recommended (M. C. Kim & Hannafin, 2011).

Deek, Kimmel, and McHugh (1998) suggest a "problem driven" instruction that introduces problems to the students and allows them to formulate solution plans. Students are guided to two levels of solution design. First, they visualize or outline the structure of solution plans, which refines the subgoals and relationships among them. Second, they develop algorithmic specifications according to the subgoals. An instructor helps them to map their solutions to program language syntax, where students need most instructor support. This approach resulted in the higher level of competence in both problem solving and program development skills as intended.

Regarding the improvement in logical thinking in problem-solving, B. Kim, Kim, and Kim (2013) introduce a paper-and-pencil programming strategy (PPS) which guides students to illustrate solution plans in various formats such as diagrams, sentences, symbols, tables, and so on. The PPS emphasizes the hands-on activities that support students to "operate" logically so they can solve problems as computers do. Although the authors presented positive results of the approach in improving logical thinking, it is worth noting that novices still need well-structured support to figure out how to translate their algorithmic solutions to programmable ones.

These pedagogical changes are consistent with the argument that novice programmers need strategic problem-solving knowledge beyond syntactical knowledge (Robins et al., 2003). Further research exploring instructional strategies that support novice programmers to translate solution plans to a compatible program is highly necessary.

References

Bayman, P., & Mayer, R. E. (1988). Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology*, 80(3), 291-298. http://dx.doi.org/10.1037/0022-0663.80.3.291

Brooks, R. E. (1990). Categories of programming knowledge and their application. *International Journal of Man-Machine Studies*, 33(3), 241-246. http://dx.doi.org/10.1016/s0020-7373(05)80118-x

Chang, C.-K. (2014). Effects of Using Alice and Scratch in an Introductory Programming Course for Corrective Instruction. *Journal of Educational Computing Research*, 51(2), 185-204. http://dx.doi.org/10.2190/EC.51.2.c

Chi, M. T. H., Glaser, R., & Rees, E. (1982). Expertise in Problem Solving. In R. Sternberg (Ed.), *Advances in the Psychology of Human Intelligence* (pp. 1–75). Hillsdale, NJ: Erlbaum.

Deek, F., Kimmel, H., & McHugh, J. A. (1998). Pedagogical Changes in the Delivery of the First-Course in Computer Science: Problem Solving, Then Programming. *Journal of Engineering Education*, 87(3), 313-320. http://dx.doi.org/10.1002/j.2168-9830.1998.tb00359.x

Google, & Gallup. (2015). *Searching for Computer Science: Access and Barriers in U.S. K-12 Education*. Retrieved from https://goo.gl/oX311J

Google, & Gallup. (2016). *Trends in the State of Computer Science in U.S. K-12 Schools*. Retrieved from http://goo.gl/j291E0

Kim, B., Kim, T., & Kim, J. (2013). Paper-and-Pencil Programming Strategy toward Computational Thinking for Non-Majors: Design Your Solution. *Journal of Educational Computing Research*, 49(4), 437-459.

http://dx.doi.org/10.2190/EC.49.4.b

Kim, M. C., & Hannafin, M. J. (2011). Scaffolding problem solving in technology-enhanced learning environments (TELEs): Bridging research and theory with practice. *Computers & Education*, 56(2), 403-417. http://dx.doi.org/10.1016/j.compedu.2010.08.024

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, 37(3), 14-18. http://dx.doi.org/10.1145/1151954.1067453

Linn, M. C., & Dalbey, J. (1985). Cognitive consequences of Programming Instruction: Instruction, Access, and Ability. *Educational Psychologist*, 20(4), 191-206. http://dx.doi.org/10.1207/s15326985ep2004_4

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2011). Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, 21(1), 57-80. http://dx.doi.org/10.1080/08993408.2011.554722

Mayer, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys*, 13(1), 121-141. http://dx.doi.org/10.1145/356835.356841

Mayer, R. E., & Fay, A. L. (1987). A chain of cognitive changes with learning to program in Logo. *Journal of Educational Psychology*, 79(3), 269-279. http://dx.doi.org/10.1037/0022-0663.79.3.269

McGill, T. J., & Volet, S. E. (1997). A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education,* 29(3), 276-297. http://dx.doi.org/10.1080/08886504.1997.10782199

Newell, A., & Simon, H. A. (1972). *Human problem solving*. Oxford, England: Prentice-Hall.

Pea, R. D. (1986). Language-independent conceptual" bugs" in novice programming. *Journal of Educational Computing Research*, 2(1), 25-36. http://dx.doi.org/10.2190/689T-1R2A-X4W4-29J2

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2), 137-172. http://dx.doi.org/10.1076/csed.13.2.137.14200

Rubio, M. A., Romero-Zaliz, R., Mañoso, C., & de Madrid, A. P. (2015). Closing the gender gap in an introductory programming course. *Computers & Education*, 82, 409-420. http://dx.doi.org/10.1016/j.compedu.2014.12.003

Shi, N., Cui, W., Zhang, P., & Sun, X. (2017). Evaluating the Effectiveness Roles of Variables in the Novice Programmers Learning. *Journal of Educational Computing Research*, 0(0). http://dx.doi.org/10.1177/0735633117707312

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3), 219-238. http://dx.doi.org/10.1007/bf00977789

Snyder, T. D., de Brey, C., & Dillow, S. A. (2016). *Digest of Education Statistics 2015 (NCES 2016-014)*. National Center for Education Statistics, Institute of Education Sciences, U.S. Department of Education. Washington, DC. Retrieved from https://nces.ed.gov/pubs2016/2016014.pdf.

Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: an empirical study. *Communications of the ACM*, 26(11), 853-860. http://dx.doi.org/10.1145/182.358436

Tom, M. (2015). Five C Framework: A Student-Centered Approach for Teaching Programming Courses to Students with Diverse Disciplinary Background. *Journal of Learning Design*, 8(1), 21-37. http://dx.doi.org/10.5204/jld.v8i1.193

Whitehouse. (2016). *Computer Science For All*. Retrieved from

https://obamawhitehouse.archives.gov/blog/2016/01/30/computer-science-all

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33-35. http://dx.doi.org/10.1145/1118178.1118215

Winslow, L. E. (1996). Programming pedagogy - a psychological overview. *ACM SIGCSE Bulletin*, 28(3), 17-22. http://dx.doi.org/10.1145/234867.234872