ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION

Paper ID #32293

Improving Pedagogy of Computer Programming Through Orthogonal Skill Isolation

Dr. Dov B Kruger, Stevens Institute of Technology (School of Engineering and Science)

Dov Kruger has consulted and taught programming in industry for 20 years, moving back to his alma mater to teach and do research. His primary research interests include secure, high-speed replacements for web programming, and computer-based education tools, particularly assessment.

Dr. Gail P Baxter, Stevens Institute of Technology

Gail P. Baxter is the Co-Director, Center for Innovation in Engineering and Science Education (CIESE) at Stevens Institute of Technology. Baxter leads CIESE research and evaluation efforts and manages a program to support faculty adoption of evidence-based teaching practices in the core courses in the School of Engineering at Stevens. Before joining CIESE, Baxter was a Senior Survey Researcher at Mathematica Policy Research, Inc., Senior Research Scientist at Educational Testing Service, and an Assistant Professor in the Graduate School of Education at the University of Michigan. In addition, she served on National Academy of Sciences Committees on Foundations of Educational and Psychological Assessment and Evaluation of National and State Assessments of Educational Progress. She earned a PhD in Educational Psychology from UC Santa Barbara.

Improving Pedagogy of Computer Programming Through Orthogonal Skill Isolation

keywords: learning programming, pedagogy programming, computational thinking, cognitive overload, formative assessment programming, C++, Java, assembler

Abstract

Computer architecture is a set of rules and methods that describe the functionality, organization, and implementation of computer systems. At the university level, undergraduate students are taught computer architecture so they are better able to: (1) debug their programs and (2) write more efficient programs given an understanding of how the computer works.

Computational thinking skills including decomposition, abstraction, pattern recognition and algorithms are important to beginner programmers even at the university level. Students must be able to construct a logical sequence of operations to accomplish a task. However, when faced with a highly complex language designed for programming efficiency, not ease of learning (such as C++ or assembler), it is also vital to understand the exact semantics of each operation. Because C++ and assembler are so complicated, students often make mistakes largely due to gaps in their knowledge of the language. This paper focuses on strategies for teaching programming in complicated languages, but the pedagogy of less-complicated mainstream languages, such as Java and Python, can benefit from some of the same approaches.

Research has shown that cognitive overload is one of the main barriers to learning programming. Even in a population of highly motivated engineering students, beginners are easily frustrated by the simplest of programs. Without an accurate mental model of how the computer actually works, beginners tend to use trial-and-error strategies, often without success. Further, even when successful, these strategies do not generalize to other more complex programming tasks.

To help students learn a complex programming language, we created a set of exercises for each of four critical programming skills: syntax, semantics of basic operations, contextual knowledge (i.e., fill in missing symbols in an incomplete program snippet), and debugging (identify and correct errors). We call this method orthogonal skill isolation, analogous to the term orthogonality as used in mathematics -- each of the individual skills is independent of the others and can be split into separate exercises, reducing the cognitive overload and frustration for each programming exercise. This does not eliminate the need to write whole programs, just as giving spelling and vocabulary quizzes does not eliminate the need to practice writing essays.

We implemented orthogonal skill isolation in a sophomore computer architecture course in which many students had little or no experience in C++ and/or assembler. After initially poor

results on a midterm given prior to orthogonal skill isolation, dramatic improvements were demonstrated using orthogonal skills exercises to scaffold learning of each skill independently. Retest showed increased scores for all students.

Introduction

In the literature on the pedagogy of computer programming, significant time and attention has been paid to trying to reduce cognitive overload.
Cognitive overload is clearly the main source of beginner frustration and the most important reason that beginners give up on computing.

A great deal of effort has been expended on preparing students to program by teaching them "computational thinking" and by simplifying computer languages, either for pedagogical purposes (BASIC, Logo, Scratch[1]) or out of a conscious desire to increase programmer productivity (Python, Golang[2], Rust[3]). Obviously, if a computer language can be simplified and programmer productivity increases, while not impacting other features, such as runtime efficiency, it is clearly a win. However, it is difficult to design a language that is both simple and produces efficient code, can catch complex errors during compilation, and is not too verbose for programmer productivity. In addition, even if new languages are simply better and replace older ones, the inertia of the market means that existing computer languages dominate for years to decades. Counting C, C++, Java and C# alone, a substantial percentage of the job market is based on programming languages with complicated and obscure semantics and tricky syntax. Programming students today must master some of these languages in order to be viable job candidates.

Programming is a complex cognitive task that can be broken down into four concrete steps, at least at the outset. For larger systems there may be other aspects as well:

- Comprehension of the desired task
- Planning a sequence of steps to accomplish that task
- Writing a program in a particular language to implement the algorithm
- Testing and (usually) debugging the program

In teaching programming, we can show students the programming constructs of a language, such as a loop, but like any language, fluency comes through practice. The obvious way to build competence is to have students write programs, obviously an authentic assessment. This, however, is extremely difficult for beginners, and even the simplest task can take students a great deal of time. In order to improve the efficiency of the learning process, several solutions have been proposed.

Merrenboer tested whether completing partial programs was more efficient than writing programs from scratch [5]. Parsons defined a problem type where the program is cut into blocks,

each of which is correct, and the student sorts the blocks into the optimal order. Solving Parsons problems has the advantage of simplifying syntax since no typing is involved, and it provides scaffolding [6]. Ericson et al. studied Parsons problems and performed experiments comparing Parsons and full programs [7][8]. The claim is that Parsons problems are a more efficient way to learn than either writing full programs, completing programs, or finding errors in code. In this paper, we try to modify this claim because it seems overly broad. The kind of problems where Parsons is equivalent to fill in the missing code are short sequence of logic where the student is being taught the fundamentals of structured programming (i.e loops, conditions, and function calls). Fig. 1 shows a sample Parsons problem for a loop in C++.

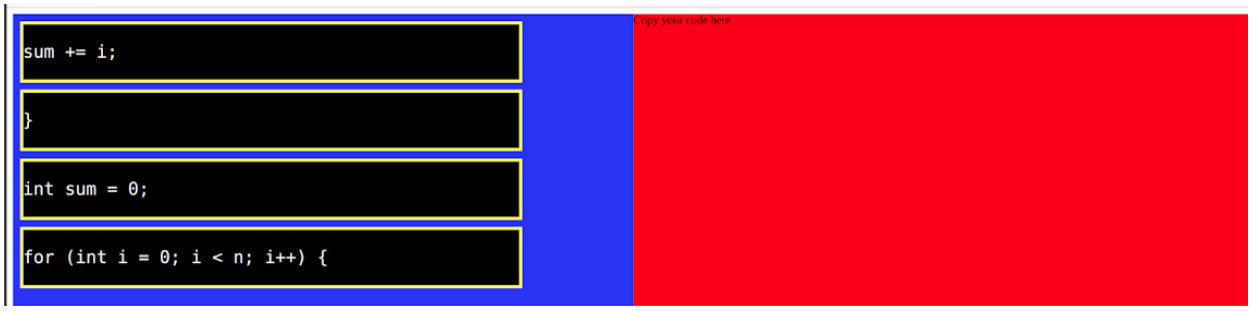

Fig. 1 - Sample Parsons Problem for C++

For the case above of counting from 0 to 9, it is easy enough for students to memorize the correct answer, and thus the puzzle works well. The next level is to provide choices (typically two) for specific features such as edge cases, e.g. ,[determine] what is the correct condition to make a loop execute the correct number of times. This is simply a multiple choice question in a different format, and provides scaffolding by leaking information since both choices show the basic structure of the command. Nonetheless, because Parsons problems can be done quickly, and reduce cognitive overload, they are clearly ideal for teaching beginners the basics. Fig. 2 shows an extension of Parsons with two choices for the same loop in C++ one of which is correct and the other incorrect. The idea this time is for the student to learn how to control exactly how many times a loop executes, and of course to do this for progressively more complex conditions.

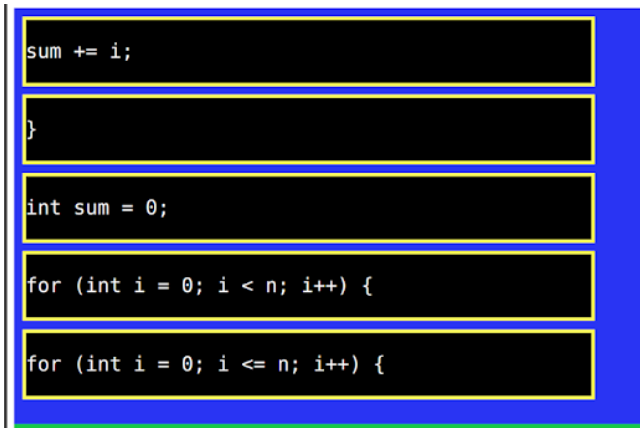

Fig. 2. A Parson Problem with “Distractors”

While the second Parson problem can be used to teach about edge cases, a contextual fill-in-the-blank can be even more effective if the student can handle the increased difficulty because it does not scaffold the basic question of how to compare two values. In English, this kind of question is called CLOZE, and it has long been regarded as the gold standard for measurement of vocabulary skill[8]. From now on, we will refer to context-based fill-in questions as CLOZE in the programming context as well. The following example shows the equivalent CLOZE question to test whether a student knows how to make the loop execute the correct number of times:

```
Complete each code snippet so the loop executes exactly n times

for (int i = 0; i [ < ] n; i++)
  cout << i;

for (int i = 1; i [ <= ] n; i++)
  cout << i;
```

In an undergraduate computer architecture course, students need to learn at a deeper level about many of the underlying computational facts that can cause computation to go wrong, as well as how the computer achieves computation. As long as everything is fine in a high-level language, the student can ignore the underlying implementation. But when errors occur, students need to understand more of the internals because they can be more effective at correcting errors, and/or optimize the program to be more efficient.

For example, any competent programmer must understand overflow. Computation of whole numbers (integers) on computers is unlike pure mathematics in that variables have finite size and therefore have a maximum value beyond which they will not compute the correct answer. This is called overflow, and computation in the C-family of languages can easily overflow, resulting in incorrect computation. This kind of basic fact can easily be taught using

isolated exercises that do nothing else. The following example shows a code snippet asking what the value of the variable is after the computation. This particular problem is scaffolded by informing the student the maximum value is for a short integer rather than requiring them to memorize it.

Given that short has a maximum value of 32767, what is the value of x after the following code snippet?

```
short x = 32766;
x += 3;
// value of x = _____
```

If a student writes -32767 they have demonstrated that they understand overflow, and that they have learned that the range of a signed 16 bit value is -32768..32767. They do not have to memorize the number since we gave the upper bound but they do need to recognize that the lower bound is one larger, since 0 is on the positive side.

There are many other similar problems that are not as vital to a correct program, but important in understanding how to make programs execute more efficiently. Understanding how the computer executes code, and under what circumstances compilers are unable to optimize programs can take students to a deeper understanding. Examples of this include reordering expressions to combine constants, precomputing inverses in floating point and many others. These examples are beyond the scope of this paper, but examining the assembler code that compilers output is an excellent way to understand what is really happening, and how to improve on it.

The next major problem is learning to program at the assembler level, where complexity is higher than C++. In assembler, every instruction has multiple effects. This is intrinsically confusing. Merely lecturing and stating these facts repeatedly is not enough for the majority of students. Writing whole programs is inefficient -- and we have observed that even when students successfully string together sequences of instructions, they may do so without really understanding the precise semantics of each one which leads to problems. The only way to really get students to internalize what is happening is to force them to state exactly what happens in a clear, unequivocal way.

As an example, consider the following instruction in ARM assembler:

```
subs    r0, r2, #3
```

The purpose of the above instruction is to subtract the number 3 from register r2, storing the result in register r0. Since that is the main point of the instruction, students usually can learn that

much. However, there are actually three things happening during the one instruction:

1. Subtraction of the value.
2. The CPU must move on to read the next instruction
3. The CPU compares the result against zero because the operation has an s appended (set flags)

The fact that an instruction has one primary function, and two more ancillary actions is intrinsically difficult. Many students simply miss one or both of the ancillary actions, despite having heard the professor review this multiple times in lecture and successfully completing programming assignments.

The solution is to use contextual fill-in questions to force students to encounter each fact in isolation and demonstrate knowledge. After a few repetitions, students assimilate the rules that apply universally and do not need to be reminded again. For example, the PC (Program Counter) contains the address of the current instruction to be executed. After each instruction is executed on an ARM CPU, the PC moves forward by 4 bytes so that the computer is ready to execute the next instruction. Note that the computer simultaneously computes a new value in register r0, computes the value of the program counter for next time, and because the instruction compares against zero, the Z flag is set to 1 if the answer is zero, 0 otherwise, and the N (negative) flag is set to 1 if the answer is negative, zero otherwise. Fig. 3 shows the instruction and the questions that measure whether students have absorbed this critical information.

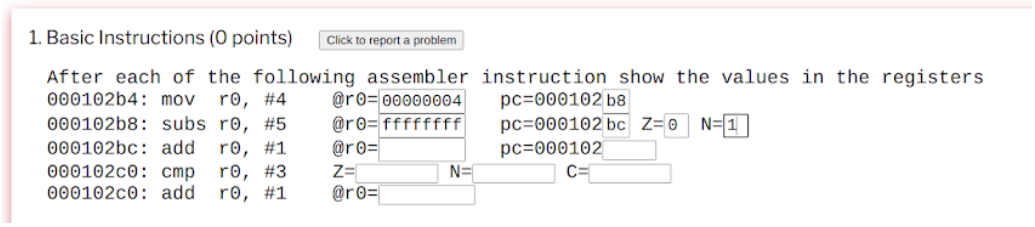

Fig. 3. Formative Assessment for ARM Assembler Instruction Semantics

Once students demonstrate that they understand how the individual instructions work, it is still a big leap to construct whole programs. The problem once again is cognitive overload. So instead, we can use CLOZE questions to scaffold the problem and allow them to write many partial programs quickly before (and after) tackling a full sequence unaided. This means that students must be able to identify which instructions and which primary values are involved, but they do not necessarily have to be able to construct the entire program. This level of question is an important stopping point between the easy introductory questions, and full programming. Fig. 4 shows a scaffolded ARM CLOZE assessment. Note that the comments are part of the scaffolding, showing the students the intended goal of each statement, and documenting the purpose for which each register is being used. Assembler is intrinsically confusing because there are many registers, whose generic names do not in any way indicate their purpose within the

program, so the comments are not only helping the students, but modeling how assembler programs should be commented as part of good programming practice.

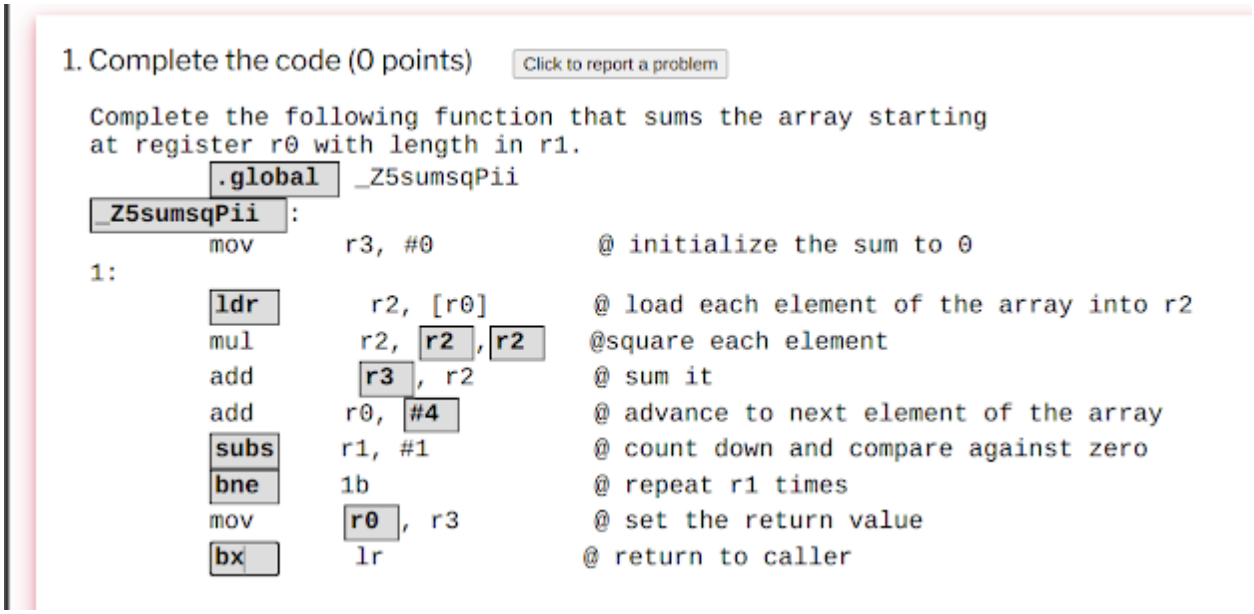

Fig. 4. Formative CLOZE Assessment: ARM Assembler Sum of Squares

Approach or Methodology

The following experiments illustrate some of the cognitive overload issues and how unpacking and tackling each aspect separately can improve students' performance dramatically. In each case, no formal experiment was planned. Situations arose demonstrating that students were experiencing difficulties and the instructor intervened.

In an intensive 5-day, 40-hour industry course for experienced programmers, n=12, in Quebec, French-speaking programmers whose English was nearly perfect were unable to hear corrections made in English while concentrating on a new skill. In three cases, while their attention was focused on a problem in C++, the instructor saw a mistake, and, speaking over their shoulder, told them "you are missing a period". The language processing brain center of these Francophone programmers was occupied and they literally could not hear the correction. After asking for the translation into French, corrections were suggested in their native language 10 out of 10 tries, example: "vous avez perdu le point." Although this was only a small-scale, one-time test, we are confident that the audio feedback channel is very effective in guiding a cognitive skill in a computer language so long as over-the-shoulder instructions are ***given in the native language of the programmer***. A teacher unable to speak the native language of the students must get them to stop thinking about the task, pay attention to the feedback, and then let them go back to it after breaking their concentration, which is vastly less effective. Even learning a few words in their native language can make a huge difference.

Students in a computer architecture course are being taught from a software perspective, learning in detail how the computation works, how to write efficient code, and how compiler translate high-level languages into assembler. The students in this class have already taken an introductory C++ but many have not programmed since the intro course on programming. Students are asked to write simple loops, algorithms with an if statement, and array manipulations. Many had extreme difficulty, showing that either they had never really learned these skills, or that they had learned but forgotten. Students were given small assignments to compute a summation or product in a loop, and a similar problem involving an array. Even though most students successfully completed these assignments, they were unable to perform them quickly and accurately on tests. It is impossible to know with certainty, but we are fairly confident that most of the students did not consciously cheat on the homework. They put in effort, asked for help, wrote programs that worked, yet clearly they were not fluent in C++, and this made it almost impossible to progress to more difficult problems and achieve fluency in assembler where the side effects and number of instructions is far higher. After a midterm in which the class average was low, interventions initiated including semantic short-answer questions to get students to master the individual statements and CLOZE questions to scaffold larger programs. Every student was told they could have a retest if they wanted one, and that if necessary, a third test could be administered, but that in order to qualify for a retest, they would have to answer practice questions, write a CLOZE exam question, and if it had mistakes they would have to fix any bugs. Fig. 5 shows the kind of question created in Canvas. While effective, Canvas assessments are extremely clumsy, and generating a correct question can easily take an hour.

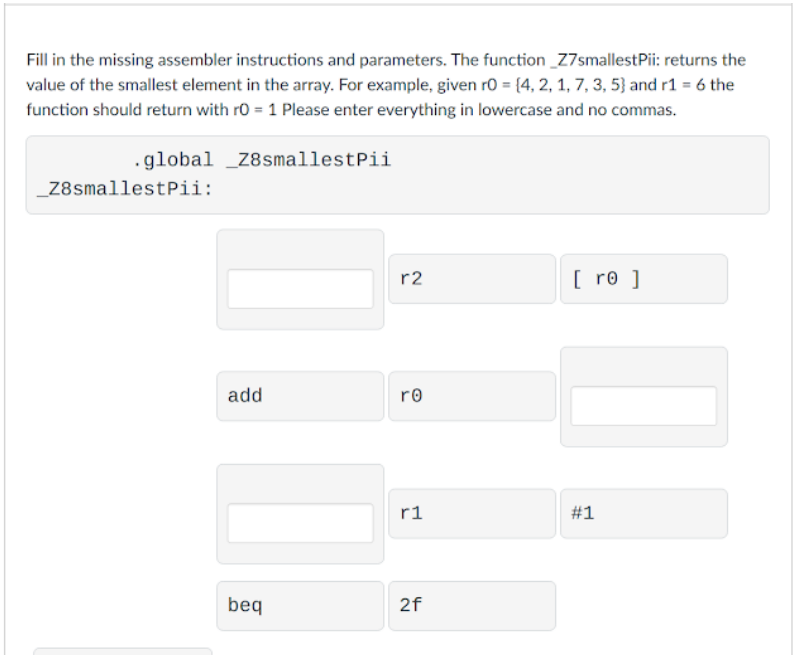

Fig. 5. Canvas-Based CLOZE Question Using Tables

Results and Discussion

In a computer architecture class with 74 students divided into four lab sections most students were able to write simple C++ functions after substantial review and similarly completed simple loops in assembler during lab. Yet they were clearly unable to deliver similar results quickly without help on a midterm. The average on the first midterm was 74. Students were offered an opportunity to retake the midterm after completing semantics fill-in, CLOZE questions, and creating their own problems. Of the 40 out of 74 who opted to do the work, they scored an average of 28.8 points higher. Only one student's second score was lower (s/he was not penalized). Some of the weakest students required two retries to reach a high score, because they had so many gaps in their knowledge that the second pass, while improved, did not eliminate all their mistakes. Only a few students scoring at the very bottom (30%) were unable to climb dramatically, and it became obvious that they were retaking the tests without putting in significant study, hoping to do better without further effort. Even the weakest students who bothered to retake the midterm raised their scores by 10 points.

A significant number of students opted not to retake the midterm. Some, of course, had excellent scores to begin with, but the low score was 58, showing that offering second chances does not mean everyone is willing to put in the effort to achieve a higher score.

This semester, we will be doing more drills before the midterm hoping to raise the average competence to obviate the need for retesting. We will not have a truly comparable class size until next semester, and even then, the effects of remote learning due to COVID may affect results. Still, as we develop a set of online quizzes to help students recall more, we expect to see a large increase in competence and success.

Conclusions and Future Work

This paper has focused on overcoming sources of frustration in programming students and improving outcomes, primarily through formative assessments, carefully scaffolded. We have not yet compared instruction semantics fillin and CLOZE questions directly to Parsons problems but the short answer problems are impressively effective. In the future we hope to directly compare these methods and contrast them for introductory classes and second courses as described above.

We will continue to collect data in a number of classes as assessments are written, and attempt to quantify the effect of scaffolded assessments on learning to program in a number of courses.

One problem administering these kinds of questions is that existing assessment software in learning management systems tends to destroy the format of questions to the point where students find the code incomprehensible. In order to be effective, assessments must be reliable,

accurately reflecting the knowledge and skills they claim to test, and the software must allow fast and efficient generation of as many assessments as needed to give students practice solving new problems, not merely memorizing answers from previous ones. Current tools such as Canvas actively obstruct use of their assessments with hugely time-consuming interfaces, by butchering the format of text with embedded blanks to the point where students do not even recognize the program, and by problems parsing answers with the flexibility demanded by programming (pattern matching).

In the future, we will present an assessment tool that is designed to to generate assessments of the type described above, particularly to take working code that has been tested on a computer and to rapidly turn it into questions with minimal operations so that the process of editing assessments does not introduce bugs. Also important, we hope to generate randomized code problems so that students never have to answer the same question twice while ensuring that the problems are completely accurate while allowing students to continue to learn from formative assessments until mastery without memorizing the answers to limited question pools.

The earlier experiment with non-native speakers suggests that another way to reduce frustration in programmers for whom English is a second language would be to translate the error messages. Ironically in C++ classes instructors regularly have to translate the arcane error messages into ordinary English, so even native speakers are befuddled by the staggeringly poor wording of error messages. Better error reporting is a critical improvement for complex languages, and any new language implementation should consider localized messages. In the absence of improved tools, teaching students how to search for answers to errors is extremely useful in reducing frustration.

In the near future, we expect to collect further data comparing different question types, and measure improvements in student skill using CLOZE and specific debugging problems in a variety of programming classes such as C++ and data structures. This paper suggests that there is not a single silver bullet, but rather a number of scaffolding techniques that together can both reduce student frustration and dramatically improve outcomes in first and second year students.

Bibliography

[1] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. 2010. “The scratch programming language and environment.” ACM Trans. Comput. Educ. 10, 4, Article 16 (November 2010),
[2] R. Pike, “Go at Google: Language Design in the Service of Software Engineering.” Splash 2012 [Online]. Available: https://talks.golang.org/2012/splash.article [Accessed Oct 14, 2020]
[3] J. Blandy, J. Orendorff and L. Tindall. *Programming Rust, 2nd Edition*. Sebastopol, CA: O'Reilly Media, Inc. 2021.
[4] B. Qin, Y. Chen, Z. Yu, L. Song, Y. Zhang. “Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs”. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2020 Pages 763–779
[5] Van Merriënboer JJG, De Croock MBM. “Strategies for Computer-Based Programming Instruction: Program Completion vs. Program Generation.” Journal of Educational Computing Research. 1992;8(3):365-394.
[6] D. Parsons and P. Haden. “Parson’s programming puzzles: a fun and effective learning tool for first programming courses.” In Proceedings of the 8th Australasian Conference on Computing Education-Volume 52. (2006). Australian Computer Society, Inc., 157–163.
[7] B. Ericson. “Evaluating the effectiveness and efficiency of Parsons problems.” ICER ’18, Espoo, Finland, August 13–15, 2018. New York, NY: Association for Computing Machinery, 60-68.
[8]B. Ericson, L. Margulieux,J. Rick. “Solving Parsons Problems Versus Fixing and Writing Code”. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research. Koli, Finland: Associate for Computing Machinery, 20–29.
[9] Taylor, W.L. “CLOZE Procedures. A new tool for Measuring Readability.” *Journalism Quarterly*, Vol 30 issue: 4, pp415-433 (1953).