https://doi.org/10.53656/math2026-2-5-ect

Educational Technologies
Образователни технологии

ENHANCING COMPUTATIONAL THINKING AND CODE COMPREHENSION THROUGH ADVANCED PARSONS PUZZLES

Ivo Damyanov, Martin Madzhov
South-West University, Blagoevgrad (Bulgaria)

Abstract. Parsons puzzles are simplified code-construction tasks in which learners construct programs by selecting and rearranging predefined program segments. As a well-established pedagogical tool in programming education, they stimulate computational thinking and a deeper understanding of code by focusing on the structure and logic of the code rather than on syntax. Although many implementations of Parsons puzzles exist today, most of them offer only standard functionality and are locked to a single language. To overcome these limitations and expand the puzzle variability, this paper presents an improved architecture for Parsons puzzles, introducing new features designed to improve applicability across courses and languages. Several key additions have been integrated into the presented Parsons puzzles application: (1) Flexible support for languages with dissimilar syntax, (2) Multi-line puzzle blocks, (3) An improved version for handling nested constructions, (4) The introduction of the concept of mini-blocks for constructing more complex puzzles in realistic scenarios, (5) Support for "fill-in-the-blank" puzzles.

Keywords: Parsons puzzles; computational thinking; code comprehension; educational tools

1. Introduction

Computational thinking (Cansu & Cansu, 2019) is a fundamental skill that goes beyond its connection to computer science. It is a key element in computer science education and includes skills such as logical reasoning, decomposing problems, recognizing patterns for abstraction, and algorithmic thinking. These skills enable learners to formulate and solve complex problems systematically by breaking them down into simple and manageable parts, identifying patterns, and developing step-by-step solutions. Naturally, this makes computational thinking important for mastering programming. Equally important is *code comprehension* – the ability to read, interpret,

and understand existing code – which is fundamental to debugging, maintaining code over time, and working in a team. As stated by Letovsky and Soloway (1986), "the goal of program understanding is to recover the intentions behind the code." Gilmore (1991) explicitly distinguishes the process of understanding from the state of understanding: "The former involves the mobilization of cognitive resources and processes in some particular configuration with the goal of constructing some mental representation of the program code. It is this mental representation of the code that is the comprehension state."

For those starting out in programming, mastering these skills can be challenging due to the specific cognitive load caused by the syntax of the language(s) used and the logic in traditional code creation exercises. On the other hand, challenges related to disinterest in programming education are particularly relevant in the contemporary educational environment. Disinterest can manifest itself through "help abuse" (Aleven et al., 2006), when students use automated support in programming environments by clicking sequentially on hints to get the ready-made solution, instead of analyzing the logic and building their own understanding. Even more problematic is the excessive reliance on large language models (LLMs) such as ChatGPT (https://chatgpt.com/) or Claude (https://claude.ai/) to completely solve the tasks at hand. LLMs can easily solve the tasks in introductory programming courses without requiring students to delve into the assignments and strive to learn the solutions (Hou et al., 2024).

Among the established didactic approaches used in computer science education, the combination of structured thinking with the gradual development of programming skills occupies a special place. Parsons problems can be naturally combined with and complement various didactic approaches. Introduced by Parsons and Haden (2006), these puzzles provide learners with shuffled fragments of code that must be arranged in the correct order to form a valid program. By providing syntactically correct fragments, Parsons puzzles allow learners to focus primarily on the logical structure and meaning of the code. Existing research highlights their effectiveness in improving code comprehension and promoting computational thinking (Ericson et al., 2017), making them valuable in introductory programming

courses. Parsons puzzles also offer a promising solution for overcoming disinterest or obtaining a quick solution through "help abuse". Unlike traditional programming exercises, which can be easily solved with the help of an LLM, Parsons puzzles require manipulation of puzzle blocks, which cannot lead to the correct solution without understanding the logical structure and connections between the code fragments. This makes the direct transfer of solutions from LLM significantly more difficult and encourages the true assimilation of computational thinking. In addition, the interactive nature of the puzzles and the instant feedback when arranging the fragments keep students' attention and reduce the likelihood of distraction by external activities. There are various ideas in the scientific community for analyzing real-time interaction to identify genuine engagement versus superficial manipulation (see Gobert et al., 2015), which can be transformed and applied to Parsons puzzles.

This article discusses how Parsons puzzles can be improved to significantly increase the variability that can be achieved within a single puzzle, as well as how to adapt the puzzles themselves to make them more flexible for use with different languages that have significant differences in their syntax. In addition, it helps develop understanding and skills for building branches in execution and embedded structures of actions – something that is fundamental to imperative programming languages. In the context of contemporary educational challenges related to the rapid development of technology, effective programming teaching methods are becoming increasingly important. Existing artificial intelligence-based solutions for assisting in the creation of program code highlight the need for new skills. Among them, the ability to critically read and interpret code generated by an external source stands out – a role that is increasingly being performed by AI. Instead of writing code from scratch, the focus is shifting to understanding, verifying, and correcting already proposed solutions. In this context, Parsons puzzles offer a suitable environment for developing structural thinking skills and recognizing logical dependencies in a program fragment (Denny et al., 2008). By working with predefined code blocks, learners are encouraged to delve into the key passages that determine the correctness of the algorithm. Adding blocks that are similar in meaning but

incorrect further enhances analytical thinking and the ability to spot critical details. This builds resilience against superficial acceptance of AI-generated code. In addition to their cognitive value, Parsons puzzles, constructed in this way, also have a meta-pedagogical dimension: they create an environment that shifts the focus from passive "copy and paste" behaviour – often observed when working with AI-assisted tools – to active participation and critical thinking. Skills such as tracking logical errors, analyzing edge cases, and checking invariants are developed precisely through this process of purposefully building correct code in the presence of distracting elements. As a result, learners do not simply use AI as a tool, but become competent mediators and validators of machine-generated knowledge. This transformation is particularly significant in the contemporary context of computer science education, which is seeing a trend towards deconstructing traditional syntax learning and an increasing emphasis on mastering abstract algorithmic principles. The ability to interact effectively and critically with automated code generation systems will be crucial for future computer science professionals, where human expertise will be expressed not so much in routine code writing as in its understanding, adaptation, and ethical evaluation.

2. Parsons puzzles and related theories

Computational thinking, as popularized by Jeannette Wing (2006), is a cognitive approach to formulating and solving problems so that solutions can be effectively implemented by information systems. It includes conceptual skills such as *decomposition* – breaking problems down into smaller parts, *pattern recognition* – identifying similarities, patterns or structures, *abstraction* – focusing on essential details while ignoring non-essential ones, and *algorithm construction* – constructing solutions step by step. These components are an essential part of programming and closely correspond to Parsons puzzles, in which learners work with pre-divided code, identify structural dependencies, and build correct algorithmic solutions by arranging the fragments.

Within computational thinking, there are a number of pedagogical frameworks with wide application, such as *Use-Modify-Create* (Lee et al., 2011) – focused on gradual transition to creativity, *U2MC* or

Understand/Use-Modify-Create, with even greater emphasis on understanding before modification, *TIPP-SEE* (Salac et al., 2020) – focused on structured analysis and interactive code modification, and *PRIMM* (Predict-Run-Investigate-Modify-Make) – focused on systematic development of programming skills. All of these frameworks share a common philosophy of gradually building programming skills through structured code skeletons and gradually increasing the independence of learners. They use scaffolding to help learners move from passive perception to active code creation, thereby developing deeper understanding and confidence in programming, and in each of these frameworks, Parsons puzzles can be effectively applied as a complementary tool.

On the other hand, empirical research by Denny et al. (2008) shows that as a testing tool, Parsons puzzles allow students to demonstrate what they know – something that is not possible with coding tasks because they find them difficult and are unable to even begin to solve a coding task from scratch.

Research conducted by Smith et al. (2024) shows through qualitative and quantitative methods that distractors in Parsons puzzles encourage students to focus on the details in the finished code. This is particularly important in formative learning contexts, where the goal is to improve skills and deepen understanding. Without distracting elements, students may solve problems mechanically without deeply analyzing the code, which limits their learning.

Parsons puzzles, as a supplement to traditional teaching methods, lead to a significantly better understanding of program constructs and greater confidence in solving programming problems. They are based on a number of theories and studies such as: *developments related to cognitive load* – Parsons puzzles reduce cognitive load because they require completion rather than the creation of code from scratch, and eliminate the need to remember syntax, *working examples* – showing solutions from experts aids learning, especially for beginners, *self-efficacy* – Parsons problems can boost students' confidence by giving them a chance to succeed early on, as well as *metacognition and self-regulation* – Parsons puzzles can support this process through structured reasoning and self-observation (Ericson et al., 2022).

3. Advanced Parsons puzzles application

Parsons puzzles are most often available in two versions – one-dimensional (1D) and two-dimensional (2D). In 1D puzzles, the lines of code are shuffled up and must be rearranged in the correct order. Variations may include distractors (unnecessary or incorrect fragments) to increase the difficulty. In 2D puzzles, in addition to the order, students must also determine the nesting of individual constructs (e.g., the body of the loop or conditional constructs). Solutions for the two-dimensional layout, by creating indents in the code, have been proposed for Python programs in (Ihantola & Karavirta, 2011). Python syntax is free from the use of brackets for block formatting. Training beginners in programming using ready-made code constructs, even with existing distractors, reduces external cognitive load, allowing learners to focus on understanding the code. However, there is still room for improvement in their functionality, especially for multilingual contexts and complex code structures, which is the focus of our new application.

To illustrate the variations in functionality available in the public domain for Parsons puzzles, we use the Feature-Oriented Domain Analysis toolkit (Kang et al., 1990) in Figure 1. The new features offered by the present solution (multiline blocks, mini-blocks for slots, attachment to slots and fill-in-the-blanks task) are also included.

Our research shows that most Parsons puzzle tools are locked to a single language, although they can be reused for languages with similar syntax. However, there are no Parsons puzzle applications that are equally suitable for languages such as Python, SQL, and C++, for example. Variants such as JS-Parsons (https://js-parsons.github.io/) and Codio (https://codio.github.io/parsons-puzzle-ui/) are presented as language-independent but ultimately they are Python-centric. There are some variants of Parsons puzzles for learning Regular Expressions (Wu et al., 2023), SQL (Wu & Ericson, 2024) and C (Sheng et al., 2023) but they are not developed as complete, publicly available platforms.

With the present implementation, we aim for flexible support of languages that have different syntaxes, by first improving the 2D puzzle variant. The automatic generation of the compound code blocks by handling

nesting and mini-blocks is not mentioned in the existing literature, and we consider it as a contribution. This is an unexpected detail, as one would assume that standard tools would have evolved to include such features.

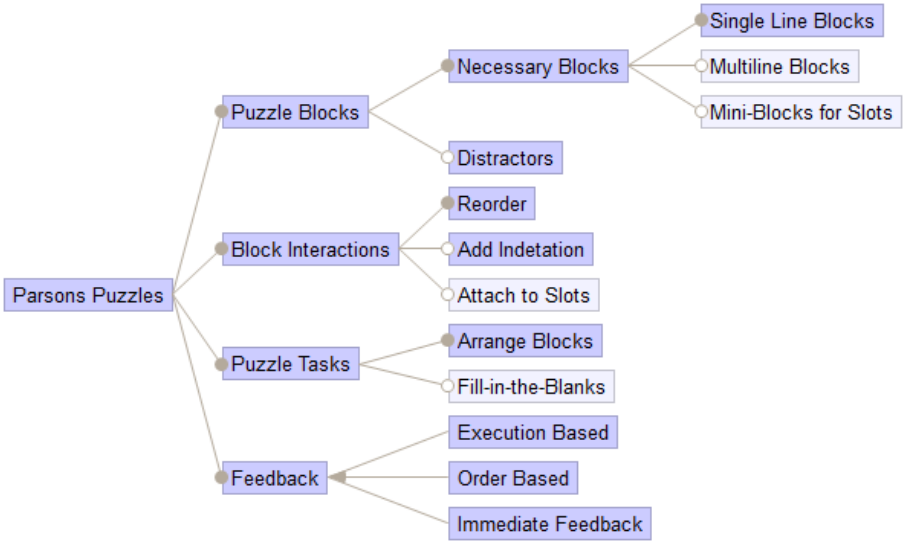

Figure 1. Feature diagram of Parsons puzzles available in the public domain

Our new Parsons puzzle app (available for download at https://github.com/damianovswu/Parsons-Puzzles-Toolkit) introduces new key features designed to increase its variability and usefulness in programming education and code comprehension, as well as opening it up to different application scenarios.

3.1. Flexible multilingual support

Most tools are oriented towards a single language, ignoring syntactic variations. This makes them extremely unsuitable for larger-scale use in different courses and situations. One of the main problems that existing puzzle implementations face with languages that have syntax similar to C, C++, C#, Java, or JavaScript – more precisely, bracket-based languages – is that the curly brackets { and } increase the "noise" in the puzzle. Puzzles for programs (or just functions) with deeper levels of nesting appear with

multiple blocks containing only brackets. Moreover, for this type of language, the presence of bracket-only blocks renders 2D manipulations meaningless. On the other hand, if we talk about two-dimensional puzzles (consider, for example, Python and T-SQL), indentation matters for Python but is irrelevant for T-SQL, which again limits the use of such a system to a single language in most cases. Multilingual support in a single platform is definitely an advantage, facilitating knowledge transfer between languages and allowing learners to identify common concepts and paradigms regardless of syntactic differences. Introductory courses often start with the use of pseudocode, which requires additional flexibility on the part of such a platform in terms of syntactic specifics. Such flexibility is particularly valuable because multilingual programming is becoming the norm in today's software industry.

3.2. Automatic surrounding of nested blocks

To achieve flexible support for languages with different syntaxes, our solution offers a two-dimensional puzzle variant, in which nesting is implemented with automatically added blocks in the "ordered" variant of the puzzle. In languages such as Python, indentation defines compound code blocks – a concept that confuses beginners who are transitioning to languages with clear delimiters (e.g., curly braces in C++). Our application automatically generates enclosing blocks (i.e., blocks with { and }, see Figure 2) for indented code in puzzles for C-family languages, but not for Python, where indentation alone is sufficient.

This feature visually connects indentation with block structure, helping learners understand scope and nesting in different languages. For example, an if statement puzzle in Python may look like a flat sequence, while its equivalent in C++ includes automatically generated brackets that clarify how the structure is preserved.

Automatic indentation (see Figure 3) helps develop consistent coding practices, which is an important aspect of professional programming. Novice programmers using languages with C-like syntax often make the mistake of omitting explicit block declarations for loop bodies or for the if and else parts of conditional statements. When the code is later edited without

forming a compound statement (i.e., without explicitly enclosing it in brackets), this can lead to hidden defects that are difficult to detect.

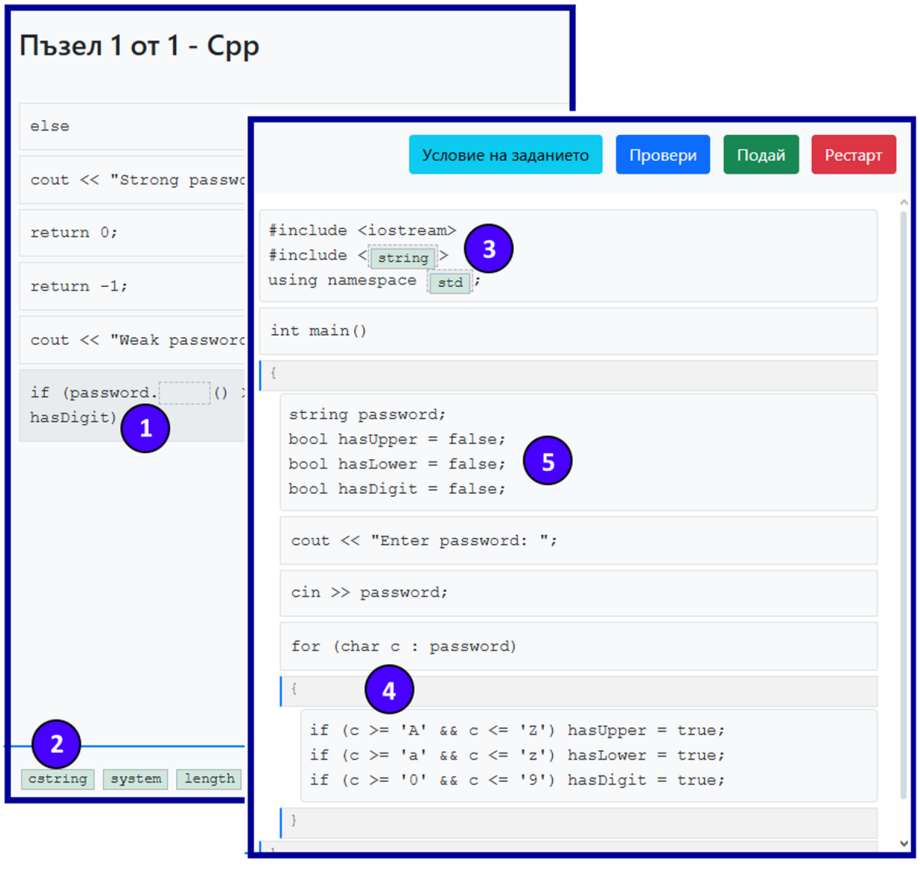

Figure 2. User interface elements for solving a C++ Parsons puzzle. (1) Available puzzle blocks with distractors, (2) Available mini-blocks for slot completion with distractors, (3) Slots for placement, (4) Auto-generated braces when indentation is applied, (5) Multiline puzzle block

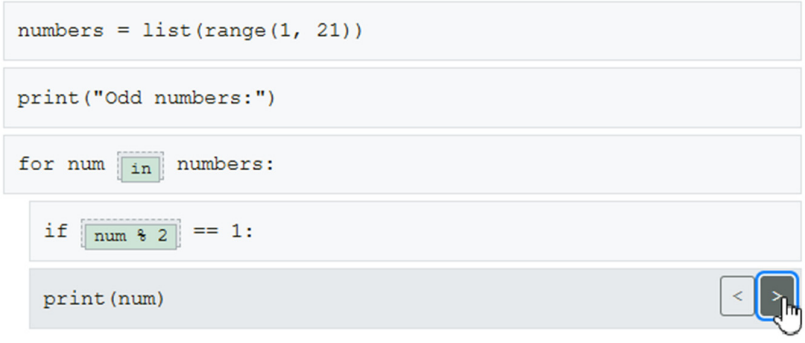

Figure 3. Solving a Parsons puzzle. The indentation button appears on hover.

3.3. Slots and mini-blocks

Wu and Ericson (2024) propose variants defined as micro-Parsons problems or horizontally arranged code blocks – a format useful for puzzle versions involving regular expressions and SQL queries. While the regular expressions variant is often sufficient to cover task variations, SQL-based tasks are frequently reduced to simple select statements. Therefore, we believe that instead of relying solely on this approach, the introduction of slots within code blocks – and mini-blocks that can be inserted into them – significantly increase the range of tasks that can be implemented as Parsons puzzles. Moreover, mini-blocks substantially increase the logical complexity of puzzles not only for general-purpose programming languages but also beyond them. For example, in a puzzle containing a for loop block, the loop condition can be represented as a slot into which learners insert a mini-block such as $i < 10$, or $i <= 10$. This level of granularity allows educators to focus on specific concepts (e.g., loop termination conditions, function parameter types, etc.) without overwhelming learners with the need to construct an entire program. Mini-blocks also help adjust difficulty by varying the complexity of the embedded fragments.

In addition, the tool preserves the textual nature of code while adding visual embedding elements, supporting the transition from block-based to text-based programming. Whereas environments like Scratch fully abstract away syntax through visual blocks, this Parsons puzzle format adapts the

syntax dynamically. This approach also offers flexibility and a unified platform for both imperative (e.g., C or Java) and declarative (e.g., SQL) languages, providing an intuitive transition from block-based languages (such as Scratch) to text-based ones through the gradual removal of visual cues.

To create slots in the puzzle blocks, the instructor must only add the slot identifier enclosed in brackets marked with **§** (see Figure 4). The instructor interface will request the entry of mini-blocks for each of the defined slots – correct and distractors, if any.

3.4. Multiline puzzle blocks

One of the things that limits the creation of puzzles is the variability in the code that leads to the same correct solution. For example, if we have a sequence of blocks with variable definitions, their position is limited only by the place of their first use. This variability often makes it difficult for puzzle authors, and they resort to writing the variable definitions in a single line to form a code block.

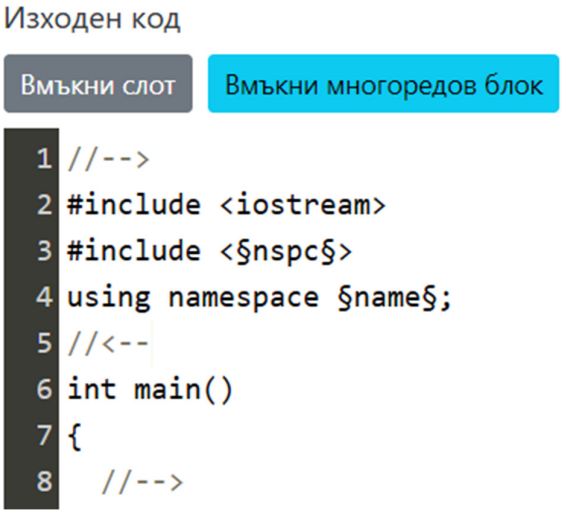

Figure 4. Creating slots and multiline puzzle blocks: using special symbols for slot tagging and language-specific comments for multiline blocks

This limitation does not allow trainees to develop good style. Another situation is, for example, in SQL puzzles, where complex queries involve multiple columns, which are often impossible to describe as a single-line code block. Therefore, to reduce the difficulty for the puzzle creators themselves, we introduce multi-line code blocks. From an implementation point of view, they are marked with specific comments (according to the syntax of the language in which the puzzle is written), which minimizes the manipulation required for their creation by puzzle authors (See Figure 4).

3.5. Single-block puzzles

Single-block puzzles (Figure 5) are a special variant that makes it easy to solve "fill-in-the-blank" type tasks. When loading a single-block puzzle, only the "Arranged puzzle" panel is displayed on the screen, with the only code block preloaded into it. The instructor is free to create single-block puzzles using multi-line code blocks or single-line code blocks to build the appropriate presentation.

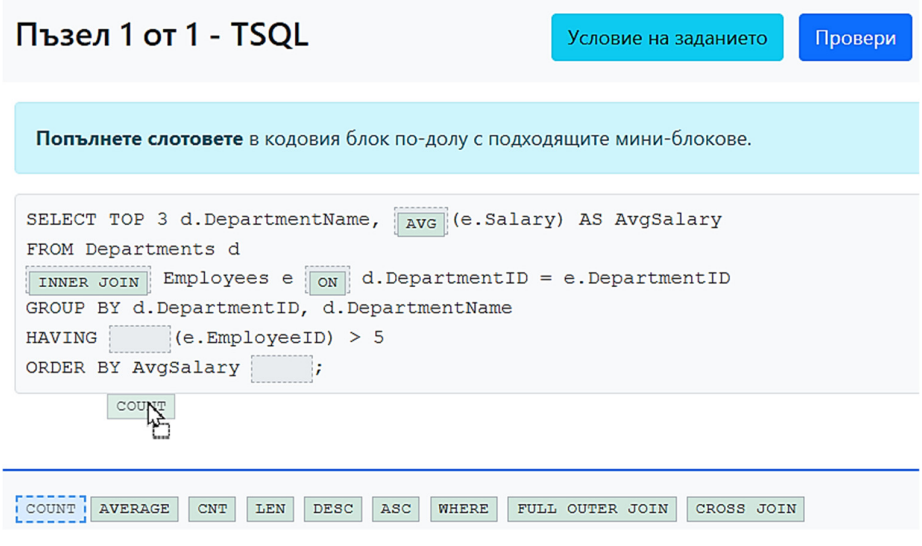

Figure 5. Solving a single-block Parsons problem

Single-block puzzles can be easily transformed for use in foreign language teaching exercises, as they structurally avoid the language specifics characteristic of programming languages.

4. Discussion

The newly introduced features increase the potential of Parsons puzzles in terms of multidisciplinary integration and cognitive support. How do these new features fit in with J. Wing's computational thinking model described above?

Decomposition: Slots and mini-blocks allow students to focus on small logic fragments, multi-line blocks allow breaking down code into logical units, making the problem more approachable.

Pattern recognition: In the context of algorithmic thinking, pattern recognition refers to the ability to identify recurring structural or logical motifs across different tasks – for example, iterative processes, conditional branches, or recursive dependencies – and to leverage these regularities to construct efficient and generalizable solutions. Programming through puzzles supports this skill by presenting learners with a series of tasks that share underlying algorithmic structures. The visual and modular nature of puzzle blocks makes these repeated patterns immediately perceptible, helping students notice similarities and transfer solution strategies from one problem to another.

Abstraction: Slots and mini-blocks allow students to focus on the core logic or expressions without worrying about templates or syntax confusion. Multi-line blocks can encapsulate logic such as an entire if-else block or SQL clause, hiding internal details when necessary. Also, the fill-in-the-blank puzzles in a single block train the student to recognize the key missing component that completes the logic, ignoring surrounding noise.

Algorithm construction: Sequencing blocks by dragging and dropping naturally models the construction of algorithms. Support for nested blocks allows students to build correct control flow (e.g., loops inside conditional constructs).

Support for multiple languages means that students can learn how similar algorithms are constructed in different syntaxes, which reinforces universal algorithmic thinking. This makes the puzzles a tool for application in a wide

range of disciplines, including interdisciplinary courses. This functionality also responds to the growing trend of integrating programming into non-programming disciplines (data science, bioinformatics, computational linguistics), where students often need to learn several languages simultaneously.

The automatic generation of nested constructs and introduced slots implements a cognitive scaffold. A visual scaffold is provided that reinforces the focus on decomposition (isolating nested blocks) and algorithmic thinking (ordering nested logical elements). On the other hand, increasing the complexity of mini-blocks (for slots) can achieve a fading support effect as an effective strategy for developing autonomy (Collins et al., 1989).

Despite their considerable potential, there are also challenges. Careful gradation and clear instructions tailored to the target audience are necessary. A careful balance must be struck between support and encouragement of autonomy, in line with educational goals and the level of the learners. Careful design and validation of the technical implementation are required. Effective integration of Parsons puzzles into the educational process requires appropriate methodological training of teachers and the development of skills for optimal calibration of complexity.

5. Conclusion

Parsons puzzles have proven useful in teaching computational thinking and code comprehension, offering an approach that minimizes the syntax-related costs of programming education. The features of our new application – flexible multilingual support, automatic nesting, slots and mini-blocks, and multi-line code blocks – build on this foundation, providing theoretical improvements in learning. These innovations promise greater flexibility, visual clarity, and targeted learning that meets the diverse needs and requirements of curricula. The theoretical analysis of the pedagogical implications of the presented functionalities provides a basis for empirical validation and further development of the application. Although empirical confirmation is beyond the scope of this article, the conceptual benefits offer compelling arguments for further development and future work aimed at conducting a series of pedagogical experiments, testing the effectiveness of

the multiline puzzle blocks, mini-blocks, and applying the puzzles themselves in different academic courses.

REFERENCES

Aleven, V., McLaren, B., Roll, I. & Koedinger, K. (2006). Toward meta-cognitive tutoring: A model of help seeking with a Cognitive Tutor. *International Journal of Artificial Intelligence in Education*, *16*(2), 101 – 128.

Cansu, F.K. & Cansu, S.K. (2019). An overview of computational thinking. *International Journal of Computer Science Education in Schools*, *3*(1), 17 – 30. https://doi.org/10.21585/ijcses.v3i1.53.

Collins, A., Brown, J. S., & Newman, S. E. (1989). Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In: L. B. Resnick (Ed.), *Knowing, learning, and instruction: Essays in honor of Robert Glaser*. Lawrence Erlbaum Associates, Inc., 453 – 494.

Denny, P., Luxton-Reilly, A. & Simon, B. (2008). Evaluating a new exam question: Parsons problems. *Proceedings of the fourth international workshop on computing education research*, 113 – 124. https://doi.org/10.1145/1404520.1404532

Ericson, B.J., Denny, P., Prather, J., Duran, R., Hellas, A., Leinonen, J., Miller, C.S., Morrison, B.B., Pearce, J.L. & Rodger, S.H. (2022). Parsons problems and beyond: Systematic literature review and empirical study designs. *Proceedings of the 2022 working group reports on innovation and technology in Computer Science education*, 191 – 234. https://doi.org/10.1145/3571785.3574127

Ericson, B.J., Margulieux, L.E. & Rick, J. (2017). Solving Parsons problems versus fixing and writing code. *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, 20 – 29. https://doi.org/10.1145/3141880.3141895.

Gilmore, D.J. (1991). Models of debugging. *Acta Psychologica*, *78*(1 – 3), 151 – 172. https://doi.org/10.1016/0001-6918(91)90009-O

Gobert, J. D., Baker, R. S. & Wixon, M. B. (2015). Operationalizing and detecting disengagement within online science microworlds. *Educational Psychologist*, *50*(1), 43 – 57. https://doi.org/10.1080/00461520.2014.999919

Hou, X., Wu, Z., Wang, X. & Ericson, B. J. (2024). Codetailor: LLM-powered personalized Parsons puzzles for engaging support while learning programming. *Proceedings of the Eleventh ACM Conference on Learning@ Scale*, 51 – 62. https://doi.org/10.1145/3657604.3662032

Ihantola, P. & Karavirta, V. (2011). Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education. Innovations in Practice*, *10*, 119.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E. & Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study (No. CMUSEI90TR21).

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J. & Werner, L. (2011). Computational thinking for youth in practice. *ACM Inroads*, *2*(1), 32 – 37. https://doi.org/10.1145/1929887.1929902

Letovsky, S. & Soloway, E. (1986). Delocalized plans and program comprehension. *IEEE Software*, *3*(3), 41. https://doi.org/10.1109/MS.1986.233414

Parsons, D. & Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. *Proceedings of the 8th Australasian Conference on Computing Education*, *52*, 157 – 163.

Salac, J., Thomas, C., Butler, C., Sanchez, A. & Franklin, D. (2020). TIPP&SEE: a learning strategy to guide students through use-modify Scratch activities. *Proceedings of the 51st ACM technical symposium on computer science education*, 79 – 85. https://doi.org/10.1145/3328778.3366821

Sheng, Y., Li, B., Wu, Z., Zhong, P. & Duan, G. (2023). A C Language Learning Platform Based on Parsons Problems. In: Hong, W. & Weng, Y. (eds) *Computer Science and Education. ICCSE 2022.*

Communications in Computer and Information Science, *1812*. Springer, Singapore. 541 – 552. https://doi.org/10.1007/978-981-99-2446-2_49.

Smith, D. H., Poulsen, S., Emeka, C., Wu, Z., Haynes-Magyar, C. & Zilles, C. (2024). Distractors Make You Pay Attention: Investigating the Learning Outcomes of Including Distractor Blocks in Parsons Problems. *Proceedings of the 2024 ACM Conference on International Computing Education Research*, *1*, 177 – 191. https://doi.org/10.1145/3632620.3671114

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

Wu, Z., Ericson, B.J. & Brooks, C. (2023). Using Micro Parsons Problems to Scaffold the Learning of Regular Expressions. *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education*, *1*, 457 – 463. https://doi.org/10.1145/3587102.3588853

Wu, Z. & Ericson, B.J. (2024). SQL Puzzles: Evaluating Micro Parsons Problems with Different Feedbacks as Practice for Novices. *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 1 – 15. https://doi.org/10.1145/3613904.3641910

✉ **Dr. Ivo Damyanov, Assist. Prof.**
ORCID iD: 0000-0001-6795-7630
Faculty of Mathematics and Natural Sciences,
South-West University,
Blagoevgrad, Bulgaria
E-mail: damianov@swu.bg

✉ **Martin Madzhov, Assist. Prof.**
ORCID iD: 0009-0001-4546-7910
Faculty of Mathematics and Natural Sciences,
South-West University,
Blagoevgrad, Bulgaria
E-mail: madzhov@swu.bg