

# Using Software Reasoning to Determine Domain-law Violations and Provide Explanatory Feedback: Expressions Tutor Example

Oleg Sychev<sup>a</sup>, Nikita Penskov<sup>b</sup> and Grigory Terekhov<sup>c</sup>

*Software Engineering Department, Volgograd State Technical University, Lenin Ave, 28, Volgograd, Russian Federation*

**Keywords:** Constraint-based Intelligent Tutor, Software Reasoning, Introductory Programming Course, Expressions.

**Abstract:** Introducing students to a new subject domain involves getting them acquainted with many new concepts. Some of these students need a trial-and-error process to learn these concepts, but it is time-consuming for teachers. An intelligent tutor capable of detecting domain-law violations and providing explanatory feedback can allow training until learning without supervision. This is especially important when teaching software engineering because it requires learning a lot of new concepts and has well-defined laws. Our goal was to develop a tutor capable to explain to the student the cause of their errors: the subject-domain laws that they violated. We present an approach to modeling subject-domain concepts and laws that allows finding correct answers and determining law violations in students' answers. A web-based tool for learning the order of evaluation for programming-language expressions was developed to assess the viability of this approach. The experiments show that Apache Jena and Clingo inference engines work quickly enough to find domain-law violations after each error in middle-sized tasks. The developed tool was evaluated by volunteer undergraduate students and received positive feedback. After the initial evaluation, the tool was used in the learning process; the students' learning gains after using the system were statistically significant.


## 1 INTRODUCTION


Programming is a rapidly growing segment of the labor market, and introductory programming courses are one of the crucial points in programming education because they require learning a significant number of new abstract concepts before the learners start to demonstrate skills. This leads to a high dropout rate and poor satisfaction among the students. A lot of various attempts and strategies are used to support learning in introductory programming courses, ranging from traditional tests (Lajis et al., 2018) to mobile game-based tools (Daungharone et al., 2019).


For some students, simply explaining new concepts and providing examples is enough to comprehend them; these students, typically, can start practicing applying these concepts in higher-level cognitive tasks according to Bloom's taxonomy of educational objectives (Bloom et al., 1956), like analyzing and synthesizing program code. However, some students do not grasp new concepts well, and even a sin-

gle poorly understood or misunderstood concept may pose a significant obstacle for further progress. Some of these misunderstandings can be corrected during discussions with teaching staff and performing simple exercises (e.g. quizzes), but limited class time mostly does not allow the teachers to find and correct all misconceptions for each student. This often results in students trying to solve higher-level tasks using guesses and analogy instead of understanding how their code is supposed to work.

If explaining a concept during a lecture (or textbook reading) failed, the student can learn the concept during assignments, but the ability of the teaching staff to provide feedback is limited so the feedback should be automated. This requires creating learning environments where students can perform simple tasks, exposing the concepts' features and subject-domain laws (or rules), while receiving immediate explanatory feedback if they make a wrong move. Only an automatic tutor can give the poorly-performing students enough experience to learn while experimenting with the subject-domain objects. Another important consideration is limiting the number of new concepts learned at once. Most program visualization educational software shows the program

<sup>a</sup>  <https://orcid.org/0000-0002-7296-2538>

<sup>b</sup>  <https://orcid.org/0000-0002-4443-3399>

<sup>c</sup>  <https://orcid.org/0000-0002-0289-1834>

execution with all its complexity, including call stack, local variables, objects and references, and statement and/or operator order of execution. While this is useful for high-performing students, it can mentally overload the weaker students with a lot of new information at once and prevent learning. Ben-Bassat Levy et al. report that complex program animations overwhelm the weakest students (Levy et al., 2003); Sorva et al. conclude that the success of visual programming simulation exercises is likely to depend on the alignment of GUI interactions with the specific learning goals (Sorva et al., 2013) which is easier to do with several smaller tools, suited for teaching particular topics. According to Kollmansberger, some students found the exercise using his Online Tutoring System too repetitive (Kollmansberger, 2010) - this problem can be solved by using adaptive testing.

The kind of feedback provided to the students during this kind of learning is also important. Simple feedback about the correctness of the answer is not enough to learn without a teacher: students often get stuck during an exercise. A significant number of visual program simulators can provide feedback about the next correct action if the student is wrong. While this always allows completing the exercise, this sort of feedback does not explain why the student was wrong and so is of limited use to foster comprehension: the student can simply perform the hinted correct step without understanding why it is correct. To develop an understanding of the subject-domain concepts, students who gave wrong answers should receive feedback about the subject-domain laws they broke and, possibly, feedback about the next correct move with the explanation of the subject-domain laws that make this step possible. The lack of explanatory feedback is one of the most severe problems in modern question generation as shown in (Kurdi et al., 2020). This problem can be solved by using the constraint-based paradigm where each mistake is tied with breaking a specific constraint. For example, in (Sychev et al., 2020) a similar approach is used to verify a program trace for the given algorithm using Pellet SWRL reasoner.

We hypothesized that an intelligent tutor based on a formal subject subject-domain model, containing an axiomatic definition of subject-domain concepts and laws, will be able to generate both correct answers for simple problems about the studied properties of subject-domain concepts and explanatory feedback about errors the student made, providing a way to train and develop an understanding of subject-domain concepts without teacher's intervention. To provide explanatory feedback, it is necessary to use declarative models, expressing the logic of making decisions

without describing the algorithm to achieve it as imperative models do. This allows the tool to acknowledge all possible correct solutions, but the most important fact is that in a declarative model, wrong solutions will trigger errors in conditions determining if the answer is correct, and the particular failed condition lets the tool determine the kind of error.

However, developing such tools poses a series of problems. Our goals were:

1. building an example formal declarative model of the subject subject-domain, capable of determining semantic mistakes;
2. assessing if modern software reasoners are capable of judging the solutions quickly enough to create a real-time tutor with per-step grading;
3. evaluate if this kind of exercise is of interest for the novice programmers.

We chose determining an evaluation order of the given expression as the experimental task to find the viability of developing intelligent tutors based on declarative formal models of subject-domain laws. This task is well-suited for experimental tutoring applications because many its problems have several correct solutions, so the feedback cannot just say "you are wrong because the correct move in this moment is X" but must explain the reasons why the operator, chosen by the student, cannot be evaluated yet.

Learning programming includes many different topics, each of them containing a lot of concepts, theories, methods, and algorithms (Papadakis et al., 2016). One of these topics is expressions and their evaluation.

While the concepts of expression evaluation order and operator precedence are mostly known to the computer-science students from basic mathematics courses, modern programming languages are more demanding: they have more levels of precedence and introduce new concepts like associativity and sequence points. The correct order of teaching concepts and tasks, while checking single concepts and their mixtures is necessary for good education (Hosseini and Brusilovsky, 2013) that makes developing skills of expression analysis and synthesis an important topic. However, during introductory programming courses, it may not get enough attention in class because of more complex problems like learning to use control-flow statements and debugging techniques.

While understanding the order of expression evaluation seems fairly trivial, when modeled, it proves to be a complex intellectual skill. To answer the basic question "Can the operator X be evaluated at this step?" the student must analyze (at most) 4 branches,

finding whether  $X$ 's left, central, and right operands are evaluated fully and are there any operators with the strict order of evaluating their operands blocking  $X$ 's evaluation. All of these branches require answering from 4 to 7 questions, and the last branch includes two "recursive" questions, implying the ability to solve the same task ahead to identify the operands of the operators with the strict order of evaluation. This makes the order of evaluation of expression an interesting task for developing an intelligent tutor based on the subject-domain model.

## 2 RELATED WORK

Intelligent tutoring systems are widely used nowadays in many subject domains and proved to be helpful and decrease overall time to study course material (Nesbit et al., 2015). There is a lot of intelligent tutoring systems for software engineering education, especially for introductory programming courses as they are often challenging for the students (Crow et al., 2018). They can be categorized by features like the supported programming languages, using open- or closed-answer questions, the kind of tutor (constraint-based or cognitive), the level of questions according to Bloom's taxonomy's, using hand-crafted or automatically generated learning problems, static or dynamic code analysis, the method and language of modeling knowledge, etc.

Cognitive tutors usually give more detailed feedback but have a lot of handwritten rules for checking all possible situations, while constraint-based tutors have more clear rules that can be used in tasks with several correct solutions (Aleven, 2010). According to a constraint-based tools survey (Mitrovic, 2012), expression evaluation order is an ill-defined task in a well-defined subject-domain that is the best choice for the constraint-based approach. In well-known tasks, all errors are generally predictable as students have constrained knowledge about field (Singh et al., 2013). More closed tasks like expression evaluation order can be described without data learning using logical rules for all possible mistakes.

Solving most of the tasks regarding expressions starts from determining the evaluation order. Some pedagogical program visualizers (Virtanen et al., 2005; Bruce-Lockhart and Norvell, 2000) show evaluation of expressions step by step, attracting attention to the evaluation order without disclosing the underlying laws. Interactive visualizers WADEIn (Brusilovsky and Su, 2002), Online Tutoring System (Kollmansberger, 2010), UUhistle (Sorva and Sirkiä, 2010), and others (Donmez and Inceoglu,

2008) ask students for expression evaluation order as a part of more complex exercises. While these tools are useful for integrating everything the students know about program execution, the amount of different tasks the students should do while solving an exercise prevents the students from concentrating on learning a particular topic.

Kumar developed and evaluated tutoring applications called Problets helping teaching programming basics, using a separate tool for each topic (Kumar, 2003). The expression tool iteratively checks the evaluation order of the given expression, checks changing variables values, shows wrong answers, and explains the correct steps. Systems have template-based problems generation and automatic problem solving, but the feedback about evaluation mistakes is scarce, showing only the fact of error. The feedback in the expression Problett tool mostly consists of demonstrating the correct solution of the entire exercise. This requires the student to compare the correct solution with theirs and make a conclusion about why they were wrong. The same approach to feedback is used by Sean Russel (Russell, 2021) in his tool for generating program-tracing questions (including the expression evaluation).

We are attempting to advance the field by developing a tool that provides explanatory feedback not only about the correct way of solving the task but telling the reasons why the student's answer is wrong right after a wrong step is made. This creates a larger search space than simply finding a correct solution, so measuring the program performance and choosing an efficient reasoner is needed.

## 3 DEVELOPED APPLICATION

### 3.1 Architecture

The developed application uses client-server architecture with laws database as shown in Fig. 1. Interaction between its components is processed by HTTP Json REST API. The server component contains API to access positive and negative laws with formulations for backends (software reasoners), decorators for the reasoners, domains that encapsulate exercise business logic, and helper classes for prepare and cache data. The client component provides the user interface.

Interaction processing starts with a client controller that updates a model and specifies the request to the server through a service, the service generates JSON and sends the request. Server helper classes receive and parse the request, call a domain to get laws, statement facts, and solution verb list; call a backend

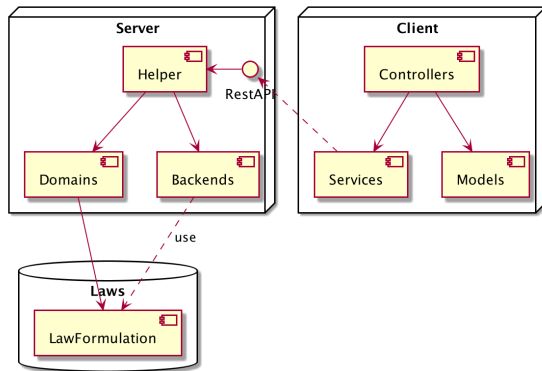


Figure 1: Component diagram.

to solve the task if not found in the cache, and judge the answer. Then the domain is called to produce text descriptions of law violations, and the helper forms a response by extending the request with additional information.

To solve the learning problem, the backend uses a formal model of the subject domain as a set of rules. The problem and the student's answer are represented as an RDF graph. Jena reasoner (McBride, 2002) is used to perform logical inference. After tokenizing the given expression, the backend builds an Abstract Syntax Tree (AST) of expression. Then expression evaluation order graph is built based on AST by adding the information from the operators creating sequence points like binary logical operators or commas. It is represented as a directed acyclic graph (DAG), describing every correct order of evaluation and the reasons for all dependencies that can be used for generating explanatory feedback. The graph is cached and used for grading students' answers step by step, correcting mistakes as they are made.

### 3.2 Usage

The tool accepts programming language expressions in different programming languages (Python and C++). A teacher, after filling a small survey, can create exercises and receive permanent URLs for them to send to the students. The teachers can see simple statistics about the performance of the students on their exercises. The students can also use the tool on their own to solve the expressions of particular interest to them.

The tool asks a student to press operators in the order they are evaluated. If the operator is chosen correctly, it is marked with green and its evaluation order is shown next to it. If a mistake is made (i.e. an operator that cannot be evaluated yet is chosen prematurely), it is marked with red color, and feedback mes-

sages are shown beneath, explaining why the student was wrong (see Fig. 2). The button for each operator shows its position in the expression that makes them identifiable even if there are a few instances of one operator. In many cases, expressions can have several correct orders of evaluation; the tool recognizes all of them as correct.

The tool determines six kinds of mistakes:

1. evaluating an operator with lower precedence first,
2. evaluating two operators with the same precedence and left associativity from right to left,
3. evaluating two operators with same precedence and right associativity from left to right,
4. evaluating a two-token operator (i.e. an operator with two tokens enclosing its operand, like square brackets for array access, function call, or ternary operator) before evaluating the operators inside it,
5. evaluating an operator whose operand is parenthesized before the operator in these parentheses,
6. evaluating an operator belonging to the right operand of an operator with strict operands evaluation order (e.g. logical or, logical and, and comma operators in the C++ programming language) before all of the operators belonging to its left operand. For each mistake, the tool generates a template-based message with the positions of all relevant operators (see Fig. 2).

Consider the situation in Fig. 2 for the C++ programming language. The student started by correctly determining square brackets as the first evaluated operator, but then made a mistake by choosing the "+" operator at position 7 as the second evaluated operator. Two operators block its evaluation for different reasons: "+" at position 5 must be evaluated first because operator "+" is left-associative while operator "\*" at position 9 must be evaluated first because its precedence is higher. The system shows both reasons so that the student can learn from their mistake.

The tool also can show a possible next correct step and explain why it is correct if the student asks for help by clicking a button. A teacher can control the availability of hints for their exercises. The explanation takes into account the two closest unevaluated operators (to the left and the right of the evaluated operator) and describes why the shown operator should be evaluated first. So if the student is stuck, they should not resort to trying all the buttons in order but can see the explanation of the next move.

Fig. 3 shows an example of a hint for the situation where the operator "&" to the left should not be evaluated yet because of its lower precedence while the

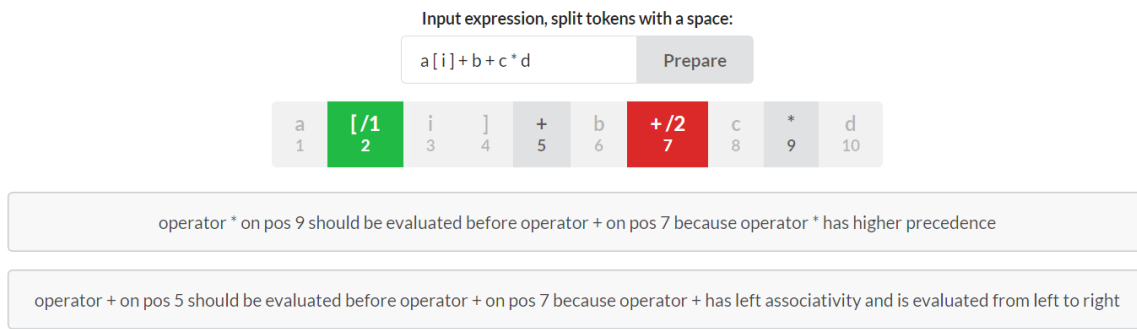


Figure 2: Explanatory feedback after a mistake.

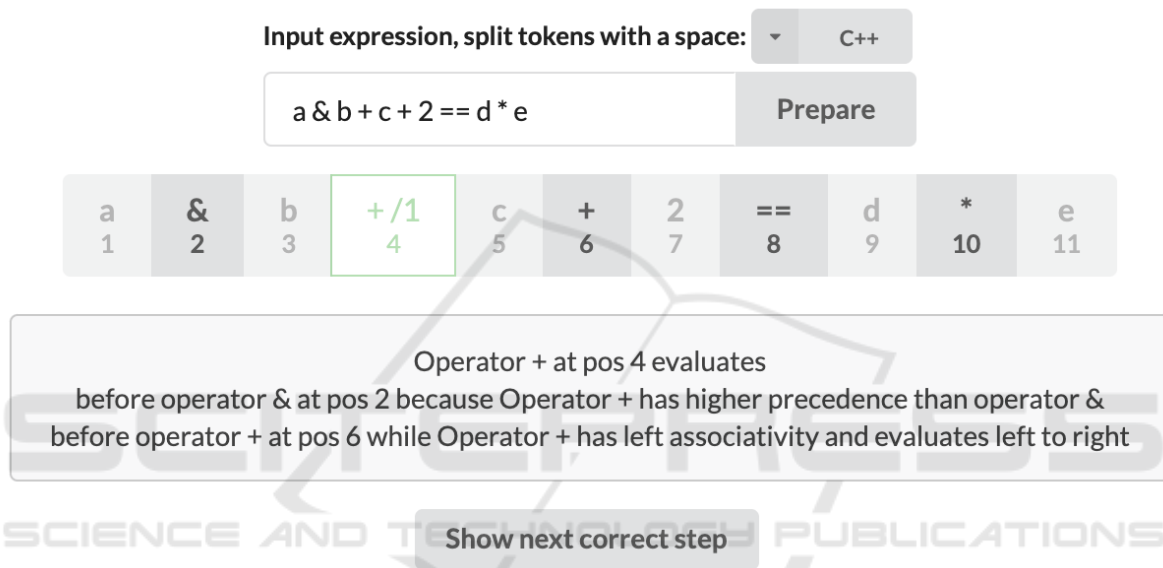


Figure 3: Demonstration of the next correct step with an explanation.

Table 1: Inference wall time for different engines, seconds.

Operands count	Clingo	Apache Jena	Pellet	Prolog	Apache Jena ARQ
4	1.48	2.75	3.98	0.92	3.77
9	1.70	3.67	9.57	1.73	7.92
13	2.35	9.19	21.49	4.64	23.63
17	9.92	9.58	17.08	10.30	13.71
23	23.54	57.25	78.86	30.92	46.71
30	43.37	45.21	344.1	60.03	116.5

operator “+” to the right should not be evaluated yet because the “+” operator is left-associative. Note that the situation before the hint button is pressed has two correct next step variants: the operator “+” at position 4 and the operator “\*” at position 10, because the operands of binary operator “==” can be evaluated at any order.

## 4 EVALUATION AND RESULTS

### 4.1 Reasoners’ Performance

As the developed tool is supposed to judge answers step by step as they are given, it is crucial to use an efficient software reasoner as a backend. We measured the performance of 5 inference engines: Pellet SWRL reasoner, Apache Jena inference engine,

Apache Jena ARQ (scripts using SPARQL Update can “reason” by modifying the knowledge graph), SWI-Prolog (with semweb library), and Answer Set Programming solver Clingo. Most of the time, the reasoning time rose with the task size, but sometimes it decreased instead; so the kind of expression and the reasoner can affect the reasoning time. The experiments (see Table 1) showed that some engines (especially Pellet) are too slow for real-time grading of expressions longer than 4 operators, but the best reasoners - Apache Jena and Clingo - can solve medium-sized tasks quickly enough. We choose Apache Jena inference engine as our backend because it is written in Java and supports multithreading; it also supports RDF natively.

## 4.2 Evaluation by Students

We asked the first-year Computer Science students of Volgograd State Technical University in the middle of the CS1 course to try and evaluate the developed tool; 14 students volunteered. To avoid the prevalence of the best-performing students among the volunteers, the worst-performing student group was selected. They used the tool to determine the order of evaluation of 5 expressions for the C++ programming language:

- $( a + b ) * c + d$  (requires only knowledge of operator precedence),
- $( a + b ) * c * d$  (requires knowledge of operator precedence and associativity),
- $a = b = 0$  (introduces right-associative operator),
- $a < 0 \ \&\& \ b > 0$  (logical AND operator has the strict operand evaluation order),
- $a < 0 ? b = 0 : c = 0$  (ternary conditional operator).

The students were able to complete all the exercises without asking teachers for support. After that, the students filled a short survey. Tab. 2 shows mean and standard deviation for the three five-point Likert scale questions with 1 meaning “Strongly disagree” and 5 meaning “Strongly agree”. Most of the students rated the developed tool as very useful and wanted to use similar tools for the other topics. Most of the students (10 out of 14) made from 1 to 3 errors; only 2 avoided errors entirely. So even the students who already studied this topic in their course learned more about it using the provided explanatory feedback and only 5 simple questions. It shows that even Computer Science students in the middle of the CS1 course have things to learn and practice about the evaluation order

of expressions. Using our tool, it can be done without spending more class time.

Table 2: Students survey.

Question	Mean	Std.dev
Were the explanations provided by the program helpful?	4.28	1.03
Are similar programs useful for learning the basics of programming?	4.71	0.79
Would you like to have similar training programs for more complex subject topics?	4.71	0.79

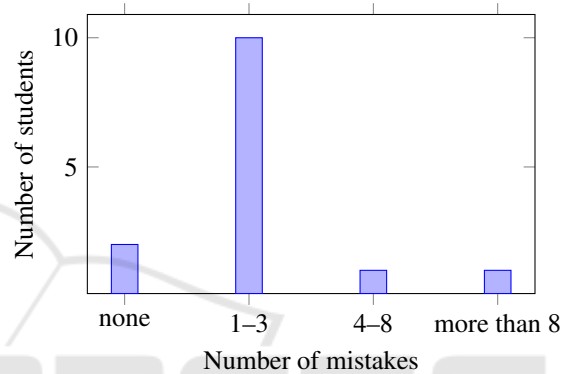


Figure 4: Distribution of students by the number of mistakes they made.

Answering free-text questions, most participants noted that the advantages of the developed system are detailed feedback about mistakes and the overall idea of narrowly focused tools for the specific concepts. Most of the problems were caused by the slow work of the SWRL reasoner (Sirin et al., 2007) used in the first version of the tool and the lack of supplementary materials about studied concepts.

The improved tool was used in the CS0 course of Volgograd State Technical University the next year with 23 expressions combining the studied laws in different ways. The students had the average pre-test score of 56% and learning gains 16% which is statistically significant ( $p < 10^{-11}$ , paired 2-tailed T-test). The students with lower pre-test scores (less than 60%) had better learning gains than the students with high pre-test scores ( $p = 0.001$ ).

## 5 CONCLUSIONS

We developed a tool whose key difference from the previous works (Kumar, 2003; Russell, 2021) is providing explanatory feedback about the violated subject-domain laws after each wrong step. We

showed that developing a formal declarative model of subject-domain laws is possible using different modern rule languages like SWRL, Jena Rules, SPARQL, and Prolog. However, determining the cause of an error requires narrowing the task to closed-answer questions: it is not possible to determine the error cause when an answer can be any number or string. But the closed-answer questions are not necessarily simple; some of them may have a lot of possible answers like determining the order of evaluation of the given expression or building a program trace, so the chances of giving a correct answer by pure guessing can be made negligible.

Another important consideration is that some of the found errors are not atomic. For example, if the student evaluates a multiplication operator before an addition operator, it can be caused by considering operators' associativity before their precedence or by not knowing the relative precedence of these two particular operators. This can be addressed by a teacher by asking the student a set of follow-up questions which can be imitated in the tool as a finite automaton. It may be possible to devise a technique of generating follow-up questions from a formal description of the subject-domain laws, but this requires future research after building more similar models.

One more advantage of the developed formal model is its ability to not just solve the tasks on its own (like many other ITS do), but to determine the set of possible errors for the given task - i.e., the necessary knowledge to solve it. This opens the way to building exercises using a large base of expressions mined from open-source code with automatic classification.

We found that some modern software reasoners (like Apache Jena and Clingo) allow performing inference quickly enough to judge students' answers step-by-step in real-time for middle-size tasks. The time of reasoning grows quickly with the expression growth and may require better strategies like caching the reasoning results after each correct step, advancing the solution, to lower the number of facts that should be reasoned each time. However, the pedagogical effect of solving the tasks with really big expressions (more than 20 operators in a single expression) is questionable: avoiding large, poorly understandable expressions is often preferable. The developed approach can be used for different subject domains if the domain model and domain-specific user interface are provided (Sychev et al., 2021).

The evaluation results show that constraint-based learning tools with explanatory feedback generation are a viable way to learn new concepts during homework without a teacher's intervention. They visualize

the processes that are normally left to students' imagination and explain the rules that were broken by a particular student. The students mostly perceived the tool positively, and the majority of them were able to improve their understanding of expression evaluation. Even the poorest-performing students, making more than 4 errors in 5 expressions, were able to complete the exercises without support from the teachers. However, some of the explanatory messages were complex and may need to be broken down into simpler statements by generating a series of follow-up questions, especially when aiming at younger learners like K-12 students. Further research may include a more rigorous studying of the learning gains using the developed tool (including using the tool as the only means to practice on the topic).

The further development of the developed formal model of expression domain will include enhancing studying expressions to teach determining data types for subexpressions (using different models for statically and dynamically typed languages) and constructing expressions to access modifiable data locations. It can be done by adding more rules on the top of the Abstract Syntax Tree that the developed subject-domain model builds. We are also going to target more programming languages. The tool can be improved by adding the possibility to demonstrate and explain correct steps if the student got stuck and ask a series of follow-up questions.

The tool can be used to teach evaluating expressions during introductory programming courses on different levels of education either as a supporting tool for a classroom activity or on its own during homework. The students can also use it to explore the topic by creating and solving their own exercises. The tool is freely accessible at <https://howitworks.app/expressions>. It supports English and Russian as interface languages and C++ and Python as programming languages. Please contact the development team if you are interested in adding support for your language.

## ACKNOWLEDGEMENTS

The reported study was funded by RFBR, project number 20-07-00764.

## REFERENCES

- Aleven, V. (2010). Rule-based cognitive modeling for intelligent tutoring systems. In *Advances in intelligent tutoring systems*, pages 33–62. Springer.

- Bloom, B. S., Engelhart, M. B., Furst, E. J., Hill, W. H., and Krathwohl, D. R. (1956). *Taxonomy of educational objectives. The classification of educational goals. Handbook 1: Cognitive domain*. Longmans Green, New York.
- Bruce-Lockhart, M. and Norvell, T. (2000). Lifting the hood of the computer: program animation with the teaching machine. In *2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era*, volume 2, pages 831–835 vol.2. IEEE.
- Brusilovsky, P. and Su, H.-D. (2002). Adaptive visualization component of a distributed web-based adaptive educational system. In *International Conference on Intelligent Tutoring Systems*, pages 229–238. Springer.
- Crow, T., Luxton-Reilly, A., and Wuensche, B. (2018). *Intelligent Tutoring Systems for Programming Education: A Systematic Review*, page 53–62. Association for Computing Machinery, New York, NY, USA.
- Daungcharone, K., Panjaburee, P., and Thongkoo, K. (2019). A mobile game-based c programming language learning: results of university students' achievement and motivations. *International Journal of Mobile Learning and Organisation*, 13(2):171–192.
- Donmez, O. and Inceoglu, M. M. (2008). A web based tool for novice programmers: Interaction in use. In Gervasi, O., Murgante, B., Laganà, A., Taniar, D., Mun, Y., and Gavrilova, M. L., editors, *Computational Science and Its Applications – ICCSA 2008*, pages 530–540, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Hosseini, R. and Brusilovsky, P. (2013). Javaparser: A fine-grain concept indexing tool for java problems. In *Workshops Proceedings of AIED 2013*, volume 1009, pages 60–63. University of Pittsburgh, CEUR workshop proceedings.
- Kollmansberger, S. (2010). Helping students build a mental model of computation. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE '10*, page 128–131, New York, NY, USA. Association for Computing Machinery.
- Kumar, A. N. (2003). Learning programming by solving problems. In *Informatics curricula and teaching methods*, pages 29–39. Springer.
- Kurdi, G., Leo, J., Parsia, B., Sattler, U., and Al-Emari, S. (2020). A systematic review of automatic question generation for educational purposes. *International Journal of Artificial Intelligence in Education*, 30(1):121–204.
- Lajis, A., Baharudin, S. A., Ab Kadir, D., Ralim, N. M., Nasir, H. M., and Aziz, N. A. (2018). A review of techniques in automatic programming assessment for practical skill test. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 10(2-5):109–113.
- Levy, R. B.-B., Ben-Ari, M., and Uronen, P. A. (2003). The jeliot 2000 program animation system. *Comput. Educ.*, 40(1):1–15.
- McBride, B. (2002). Jena: a semantic web toolkit. *IEEE Internet Computing*, 6(6):55–59.
- Mitrovic, A. (2012). Fifteen years of constraint-based tutors: what we have achieved and where we are going. *User modeling and user-adapted interaction*, 22(1-2):39–72.
- Nesbit, J. C., Liu, Q. L. A., Liu, Q., and Adesope, O. O. (2015). Work in progress: Intelligent tutoring systems in computer science and software engineering education. *Proceeding 122nd Am. Soc. Eng. Education Ann.*
- Papadakis, S., Kalogiannakis, M., and Zaranis, N. (2016). Developing fundamental programming concepts and computational thinking with scratchjr in preschool education: a case study. *International Journal of Mobile Learning and Organisation*, 10(3):187–202.
- Russell, S. (2021). Automatically generated and graded program tracing quizzes with feedback. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 2, ITiCSE '21*, page 652, New York, NY, USA. Association for Computing Machinery.
- Singh, R., Gulwani, S., and Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 15–26, New York, NY, USA. Association for Computing Machinery.
- Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical owl-dl reasoner. *Web Semant.*, 5(2):51–53.
- Sorva, J., Lönnberg, J., and Malmi, L. (2013). Students' ways of experiencing visual program simulation. *Computer Science Education*, 23(3):207–238.
- Sorva, J. and Sirkiä, T. (2010). Uuhistle: A software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli Calling '10*, page 49–54, New York, NY, USA. Association for Computing Machinery.
- Sychev, O., Denisov, M., and Anikin, A. (2020). Verifying algorithm traces and fault reason determining using ontology reasoning. In Taylor, K. L., Gonçalves, R., Lécué, F., and Yan, J., editors, *Proceedings of the ISWC 2020 Demos and Industry Tracks, Globally online, November 1-6, 2020 (UTC)*, volume 2721 of *CEUR Workshop Proceedings*, pages 49–54. CEUR-WS.org.
- Sychev, O., Denisov, M., and Terekhov, G. (2021). How it works: Algorithms - a tool for developing an understanding of control structures. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 2, ITiCSE '21*, page 621–622, New York, NY, USA. Association for Computing Machinery.
- Virtanen, A., Lahtinen, E., and Järvinen, H.-M. (2005). Vip, a visual interpreter for learning introductory programming with c++. In *Kolin Kolistelut - Koli Calling 2005 Conference on Computer Science Education*, pages 125–130.