

Article

Mass Generation of Programming Learning Problems from Public Code Repositories

Oleg Sychev *  and Dmitry Shashkov 

Software Engineering Department, Volgograd State Technical University, Lenin Ave. 28,
400005 Volgograd, Russia; brookit18@gmail.com

* Correspondence: oasychev@gmail.com

Abstract: We present an automatic approach for generating learning problems for teaching introductory programming in different programming languages. The current implementation allows input and output in the three most popular programming languages for teaching introductory programming courses: C++ , Java, and Python. The generator stores learning problems using the “meaning tree”, a language-independent representation of a syntax tree. During this study, we generated a bank of 1,428,899 learning problems focused on the order of expression evaluation. They were generated in about 16 h. The learning problems were classified for further use with the used concepts, possible domain-rule violations, and required skills; they covered a wide range of difficulties and topics. The problems were validated by automatically solving them in an intelligent tutoring system that recorded the actual skills used and violations made. The generated problems were favorably assessed by 10 experts: teachers and teaching assistants in introductory programming courses. They noted that the problems are ready for use without further manual improvement and that the classification system is flexible enough to receive problems with desirable properties. The proposed approach combines the advantages of different state-of-the-art methods. It combines the diversity of learning problems generated by restricted randomization and large language models with full correctness and a natural look of template-based problems, which makes it a good fit for large-scale learning problem generation.



Academic Editors: Maria Nisheva-Pavlova, Galia Angelova and Moulay A. Akhloufi

Received: 4 February 2025

Revised: 22 February 2025

Accepted: 25 February 2025

Published: 28 February 2025

Citation: Sychev, O.; Shashkov, D. Mass Generation of Programming Learning Problems from Public Code Repositories. *Big Data Cogn. Comput.* **2025**, *9*, 57. <https://doi.org/10.3390/bdcc9030057>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: learning problem generation; introductory programming learning; artificial intelligence; programming languages; expressions

1. Introduction

Various programming languages are used in introductory programming courses to teach future computer science professionals. According to the results of different studies, Java, Python, and C++ remain popular languages for teaching beginner programmers [1,2]. Teachers choose a programming language according to their features. For example, Python is valued for relative easiness and readability, Java is valued for emphasizing the object-oriented approach, and learning C++ allows users to foster an understanding of managing memory and resource usage [3].

During introductory programming courses, the student needs to develop their understanding of the basic concepts of programming, for example, evaluation of expressions, control flow structures, variable scope, and so on. That requires a lot of training that involves solving relatively simple learning problems that demonstrate each concept while receiving feedback on the correctness of their actions. The large sizes of the student cohorts and the lack of teacher time prevent individual work with a teacher at that stage.

An intelligent tutoring system (ITS) can be used to facilitate solving that kind of learning problem [4,5]. ITSs are designed to imitate intelligent behavior of human teachers in selecting learning problems to solve, verifying solutions, providing feedback, and demonstrating worked examples [6]. An ITS can adapt to a particular student; it can take into account the actual level of their knowledge, provide suitable programming learning problems for developing skills and reducing knowledge gaps, and provide personalized feedback according to the student's actions. This is especially effective during the initial stages of programming learning, when basic skills are developed because generating meaningful feedback is much easier for learning problems that require the development of basic skills. The use of ITS can reduce the workload of the teaching staff, improve the quality of learning by allowing students to solve simple learning problems until mastery, and allow the teachers to concentrate on more creative tasks when working with students.

However, that requires a large and diverse problem bank. The problem bank should contain a large number of learning problems; this allows the verification of students' knowledge of various concepts and rules of the subject domain. Individual training until mastery for groups of students requires different learning problems: from simple problems requiring only one rule to solve to complex problems requiring in-depth knowledge of the subject domain and combining all the relevant knowledge. A particular student may need several learning problems with a particular configuration of required skills to master them. Also, when students are learning in groups, it is good to give each student a unique set of learning problems to avoid sharing solutions among the students. That requires creating a large, diversified bank of learning problems that contains many learning problems for each combination of taught skills and each difficulty level.

Most of the programming-related ITSs are devoted to a single programming language [7–9], but some support several languages (for example, [10]). For example, PyKinetic is intended only for learning Python and has seven difficulty levels, where for each problem there are several test cases and one predefined hint [11]. Some ITSs use problem banks with a limited set of learning problems, manually divided into difficulty levels. That limits their usage in large classes and during training to mastery and makes adding support for new programming languages very costly.

Developing problem banks for multi-language ITSs is particularly challenging because, while many features of programming languages are common, each language has unique features that require learning [12–14]. Typically, this problem is solved by developing a problem bank for every language separately, but that significantly increases the workload for authoring learning problems. The alternative approach is converting learning problems of the same type between programming languages with separate classifications of learning problems by required skills and complexity. Most of the differences between programming languages are syntactic; something that is carried out in one of the languages can be achieved in others because they all can be reduced to a Turing machine.

Creating large and diversified problem banks requires automatizing learning-problem generation. In this study, we leverage the previously described techniques of generating learning problems for programming ITSs from open source code to generate a big, diversified learning-problem bank for an ITS teaching determining the order of evaluation of a programming-language expression. The ITS supports the C++ (standard C++ 11 and above), Java (version 11 and above), and Python (version 3.8 and above) programming languages, and the learning-problem generator should support conversion of expressions between those languages to increase the number of available learning problems for every language.

The contributions of this article are as follows:

- it describes a method for learning problem generation in the programming domain that uses existing open-source code to minimize human labor and allows conversion of learning problems between programming languages and generating multi-language problem banks;
- it presents a large learning-problem bank for the ITS CompPrehension on teaching programming-language expressions generated using the proposed method;
- it describes validation procedures necessary for generating large problem banks that cannot be validated manually, including simulation of different user behavior and gathering data on the results of their grading in the ITS core;
- it provides the results of selective evaluation of generated learning problems by expert teachers, who were given a way to select any kind of learning problems according to the classification by the generator.

The rest of the article is organized as follows: Section 2 describes related work on learning-problem generation. Section 3 describes the proposed method of generating learning problems in the programming domain based on open-source code. Section 4 describes the generated problem bank, its validation, and evaluation. Section 5 discusses our findings. Section 6 lists threats to the study's validity. Section 7 provides conclusions.

2. Related Work

One of the main questions in automatizing the generation of learning problems is the amount of manual work required to use the generated problems. It dramatically affects the scalability of the generator: the less manual work per generated problem is required, the more usable learning problems you can generate. Ideally, the amount of manual work must be zero.

There are two stages at which manual labor can be used during learning-problem generation: preparation and verification. Template-based methods use manual labor in the preparation stage: the human author creates problem templates, which are then used to generate many learning problems. This allows us to precisely control the kinds of problems the problem bank receives, but also limits their diversity; that method puts the complex, creative work on the human author. This method is also vulnerable to errors introduced by the human template author. A viable alternative is generating as many diverse problem sets as possible and controlling the kinds of used problems at the problem selection level instead, which allows for the use of the kinds of problems that were not envisaged by the human problem author.

Manual verification is usually used in methods using random generation or machine-learning models (including LLMs). This is caused by the unreliability of the methods: some of the generated learning problems are wrong and cannot be used. Even with a high percentage of usable problems, it requires manual verification of the entire generated bank to weed out wrong learning problems, which significantly limits the scalability of the problem-generation process. Methods that guarantee the correctness of all generated problems allow for large-scale problem generation.

One of the approaches to problem generation is the manual creation of problem templates, which are then used to create a variety of problem instances. Sadigh et al. [15] use a template-based approach to problem generation. A learning problem is formalized during the generation process to produce the basis of the template, which is then mutated to generate new problems. Okonkwo and Ade-Ibijola et al. use context-free grammars to build templates for synthesizing short Python programs with loops [16] and expressions [17] to teach programming. A context-free grammar is used to formally describe the syntax of nested loop problems. The product rules allow teachers to define templates for generating

nested loops of different depths, including recursive structures. However, the reported implementation of this synthesis method only supports the Python programming language.

Gulwani et al. [18] offer an example-based generation of learning problems. To solve a problem, the Satisfiability Modulo Theory is actively used for effective exploration of the program search space and Version Space Algebra for manipulation and concise representation of the problem data. This method allows the generation of bit-vector algorithms, spreadsheet macros that solve a specific set of problems, and mathematical terms.

There is a problem-generation approach based on using Answer Set Programming (ASP). ASP uses a model based on “if-then” production rules; ASP programs consist of a set of facts and rules that describe problems and their constraints; its most powerful feature for learning-problem generation is using variables that receive random values. ASP has been used by Polozov et al. to generate word math problems [19], and by O’Rourke et al. to generate problems for the intelligent algebra tutor [20]. The disadvantage of those problems is that they look artificial. That can be acceptable in algebra but not acceptable in much more complex expressions of programming languages.

Some approaches involve defining sets of constraints that a generated problem must satisfy. In the SQL Tutor developed by Martin and Mitrovic [9], the construction of an SQL statement for a problem begins with one or more constraints, which expand as the statement is formed. This approach requires human involvement and cannot be considered fully automatic. Martin and Mitrovich report that it took 3 h of human effort to create 200 problems, which does not allow problem generation to be scaled to large problem banks.

A growing trend in learning problem generation is the usage of Large Language Models (LLM). Unfortunately, those models have an unpredictable tendency to hallucinate, which significantly affects the accuracy of their results [21]. Therefore, when using an LLM to generate learning problems, human evaluation is necessary to verify the correctness of the generated problems, which significantly affects the performance of the generator.

For example, Austin et al. [22] describe a method for generating program code using a single language model with different parameter sizes. To confirm the quality of the work, benchmarks were used that included code synthesis tasks using basic programming skills and mathematical problems in the datasets. However, the authors note that the model has low performance, and it is often not possible to achieve correct problem generation at the first attempt. Similar problems arise with the Codex model described by Chen [23]: unit testing results of the dataset revealed frequent issues in the generated code, with the pass rate for most tested parameters below 50%. AlphaCode, a model designed to generate code for competitive programming tasks, has similar results [24]. So using LLMs to generate learning problems in programming-related tasks is usually not a good idea.

Xue et al. [25] describe a more complex approach to LLM-based code generation, using an approach that iteratively improves the code, exploiting the strengths of different programming languages and reducing the number of bugs. The code is first generated by an LLM in the main programming language (Python was used in the experiment), then, in case of failure, in other languages (Java & C++ claimed), and then translated back to the main language using the same LLM. The code obtained at each iteration was tested against test cases. Reported accuracy reaches 96% on one of the two benchmarks and increases the baseline by 18%. No human intervention is required; the process is fully automatic. However, in that study, multi-language generation is only used to improve the implementation in the main language.

The literature on using LLMs to generate learning problems in well-defined domains like programming is in its infancy, but there are recent works on using LLMs to conduct one step of the method proposed in this article: translating program code from one programming language to another. Those solutions are not reliable and tend to produce errors. For example, Pan et al. 2024 [26] conducted a big study on bugs introduced when translating code between programming languages by using LLMs; they note a high error rate. The most common errors are incorrect translation of operators, which is critical for the learning problems this study considers, and the usage of syntax of the source programming language in the translated code. LLMs can include in the translated code logic that was not present in the source code, or use a library API of the target language that does not match the behavior of the source language. The study's authors conclude that LLMs currently lack the capability to translate the code of real-world projects effectively. They also note that solutions based on translation without LLMs are deterministic and predictable, which supports the choice made during this study. There are studies on creating an LLM-based code conversion solution where the authors try to reduce errors introduced by an LLM. In a study by Eniser et al. [27], a solution was created for translating code to Rust from other programming languages, which uses zero-shot prompting and a strategy of differential fuzzing and feedback strategies to fix errors in the translated program code. However, their approach, in conjunction with various LLM models, gave the maximum success rate of approximately 50%. In another project, InterTrans, the authors were able to achieve computational accuracy from 87% to 95% [28]. InterTrans performs several intermediate code transformations into different languages, which form the tree of translation paths. The system searches for the best translation among these paths. This solution is computationally expensive, particularly as the number of intermediate code translations decreases, resulting in a decline in translation quality. The research results show that the current precision of code translation by LLM models is too low to be used for generating learning problems. Researchers find that wrong or low-quality learning problems can contribute to student dissatisfaction, anxiety, and emotional exhaustion. Even though faculty support can mitigate some of those negative effects, the problem quality is a critical factor in student satisfaction [29]. This emphasizes the need for quality and precision in generating learning problems.

There are approaches that use templates and constraints. Amruth Kumar in his study [30] recommends using the template generation of learning problems to determine the order of expression evaluation. Together with the problem formulation data, "learning objects" are created, which act as metadata for searching problems and adapting them to the student.

While many researchers use different approaches to generate learning problems for programming assignments, they usually were only oriented on one programming language and often required human work either at the beginning (creating problem templates) or in the end (verifying generated problems), which limited the method scalability and the amount of generated problems. In our research, we aim to close this gap and propose a method of generating learning problems spanning several programming languages without human participation that should dramatically increase the output of the problem generator.

One of the effective approaches combining the natural look of the learning problems generated by the template-based method with the diversity and robustness of random learning-problem generation is to mine existing data for problem templates [31]. In the programming domain, this can be done by mining open-source software repositories. That method was successfully used to generate expression evaluation learning problems in the C++ language [32]. However, not every programming language has a large number of open-source repositories with code suitable for learning-problem generation. So, a large-scale

problem generator for an ITS supporting multiple languages should be able to perform the conversion of generated problems from one programming language to the others. It should also support problem classification to determine the knowledge and skills that are needed to solve each problem because there is no human author of a template to provide that information. This study is devoted to solving these problems.

3. Learning Problem Generation

3.1. Method

The method of learning-problem generation that we propose is based on the method of generating expression-related learning problems described in [32]. However, that method supported problem generation only in the source programming language, which is unsuitable for multi-language tutoring systems because there may be little code base for generating learning problems in some programming languages, especially when it comes to specific kinds of problems requiring code with rarely-used properties. So we improved that method to allow for conversion of the program code that the learning problem is based on.

We propose the following method of generating learning problems in the programming domain from source code:

1. Find and download source code in one of the supported programming languages from open-source repositories.
2. Parse the source code and collect suitable pieces of code for generating learning problems.
3. Convert those pieces of code into language-independent syntax tree representation called “meaning tree”.
4. For each supported programming language do the following steps:
 - (a) Perform extended tokenization of the language-independent expression tree for the programming language to calculate metadata.
 - (b) Use the tree to generate run-time information that affects the problem solution in the given language. This allows the creation of multiple problems from the same templates in the form of syntax trees.
 - (c) Calculate metadata necessary to find and use the learning problem according to the pedagogical needs by using the converted tree and extended tokens.
5. Create a learning problem from the meaning tree and related metadata for problem variants in different programming languages;
6. Store the generated problems in a problem bank for later use. At this stage, filtering is used to avoid saving too many similar learning problems or too many similar steps in one problem.

Algorithm 1 shows the proposed method in the formal form.

That method was used to generate learning problems for the ITS teaching programming-language expressions. So the necessary pieces of code, in this case, were expressions, and the run-time information generated at stage 4 (b) includes the values of operands of the operators like logical AND and logical OR that do not always calculate all their operands.

Algorithm 1 Learning Problem Generation**Require:** S : source code repository**Require:** L : set of supported languages**Ensure:** P : problem set

```

1:  $C \leftarrow \text{Download}(S)$  ▷ Download source code from the repository
2:  $C_{AST} \leftarrow \text{Parse}(C)$  ▷ Parse the source code into AST
3:  $E \leftarrow \text{FindComplexExpressions}(C_{AST})$  ▷ Find expressions with more than two operators
4: for each  $e \in E$  do ▷ Iterate over complex expressions
5:    $M \leftarrow \text{ConvertToMeaningTree}(e)$  ▷ Convert expression into a Meaning Tree
6:   for each  $L \in \mathcal{L}$  do ▷ Iterate supported languages
7:      $T_L \leftarrow \text{Tokenize}(M, L)$  ▷ Tokenize the Meaning Tree for language  $L$ 
8:      $D_L \leftarrow \text{ComputeMetadata}(M, T_L)$  ▷ Compute metadata for  $L$ 
9:      $R_L \leftarrow \text{GenerateRuntimeInfo}(M, L, D_L)$  ▷ Generate runtime data for  $L$ 
10:     $R_L \leftarrow \text{FilterRuntimeData}(R_L)$  ▷ Filter redundant runtime data
11:     $T_L \leftarrow \{T_L + R \mid R \in R_L\}$  ▷ Generate multiple instances with runtime data
12:  end for
13:   $P_e \leftarrow \text{GenerateProblems}(M, \{T_L, D_L\}_{L \in \mathcal{L}})$  ▷ Generate learning problems
14:   $\text{Store}(P_e)$  ▷ Store problems in the problem bank
15: end for

```

The generated learning problems are supplied with metadata that is used to determine the conditions of their pedagogical usage. The chief categories of metadata are as follows:

1. the number of steps to solve the problem: how many operators must be pressed in a certain order for the student to successfully solve the problem;
2. the difficulty coefficient: a value between 0 and 1, which summarizes its relative difficulty compared to other problems according to the method described in [32];
3. the domain concepts used (e.g. types of operators) and necessary to understand it (see Table 1);
4. the skills a student must master to solve the problem (see Table 2);
5. the typical errors (violations) that a student can make when solving the problem (see Table 3).

Table 1. Concepts distribution in the generated bank

Concept	Problem Count
Common	
Assignment operator	302,201
Assignment expression operator in Python ¹	145,901
Arithmetic operators	476,156
Binary operators	
Add (+)	214,337
Subtract	134,737
Multiply	103,577
Divide	41,776
Unary Plus	282
Unary Minus	8145
Floor Division ²	922
Prefix Increment	1240
Prefix Decrement	579
Postfix Increment	7506
Postfix Decrement	347

Table 1. Cont.

Concept	Problem Count
Augmented assignments	
Add (+=)	15,819
Subtract	2272
Division	128
Bitwise And	2945
Bitwise Or	5674
Left bitwise shift	85
Right bitwise shift	55
Logical operators	
Logical NOT Operator	78,466
Logical AND Operator	103,310
Logical OR Operator	78,857
Comparisons	
Equal	182,256
Not Equal	81,843
Less than	43,746
Greater Than	35,416
Less or Equal	14,529
Greater or Equal	22,844
Bitwise operators	
NOT operator	11,551
AND operator	78,267
OR operator	40,647
XOR operator	5841
Right Shift Bitwise Operator	35,863
Left Shift Bitwise Operator	41,772
Other	
Member Access Operator	883,215
Subscript Operator	414,095
Indirection operator	57,630
Function call	771,602
Ternary Conditional Operator	47,643
Cast operator	172,364
Size-of operator	28,500
Comma operator	140,703

¹ Assignment expression operator in Python (:=). ² These floor division cases were identified without inferring the types of operands. They were either taken from Python programs or contained explicit type cast.

In many programming languages, the process of expression evaluation is not static. There are operators that choose whether some of their operands will be evaluated based on the values of other operands. For example, the ternary conditional operator evaluates only two of three operands: the condition is evaluated first, and then, depending on its value, one of the two remaining operands is evaluated. Another common example can be seen in logical operators: the and operator evaluates its right operand only if the value of the left operand is true, and the or operator evaluates its right operand only if the left operand is false. This affects the correct answer, the number of steps to solve the problem, and necessary skills and possible errors because it can “turn off” some parts of the expression.

Table 2. Skills distribution in the generated bank.

Skill	Problem Count
Student can determine if the operator has a central operand between its tokens	1,393,101
Student can determine whether the central operand of the current operator has been evaluated	1,062,830
Student can determine if the operator has left and right operands	1,151,385
Student can determine if there is a competing (unevaluated) operator to the left or to the right of the given operator	1,378,257
Student can determine that the selected operator is enclosed in an unevaluated operator	731,481
Student can determine the order of evaluation based on the operators enclosed by parentheses	306,375
Student can determine the order of evaluation based on the operators' precedence	1,114,576
Student can determine the order of evaluation based on the operator associativity for unary operators	949,923
Student can determine the order of evaluation based on the operator associativity for binary operators	929,213
Student can determine the presence of operators with the strict order of evaluating their operands	990,798
Student can determine whether the currently selected operator has a strict order of evaluation of operands	437,952
Student knows which operand is evaluated first	205,590
Student can determine whether the first operand of a strict-order operator is fully evaluated or not	205,590
Student can determine the strict order operators in which all operands are evaluated ¹	456,252
Student can determine whether some operands of the strict-order operator are not evaluated	362,116
Student can determine that in a programming language, the central operands of operators will have a strict order of evaluation ²	492,538
Student can determine that the current operator is not placed inside an operator that can have more than one central operand	198,217
Student understands that a function has only one argument.	308,082
Student can determine whether the previous central operands of an enclosing operator are evaluated	190,436

¹ The comma operator in C++ is a strict order operator, but it always calculates both its operands, while other strict-order operators can omit the evaluation of the right operand. ² In some languages, such as Java, the specification guarantees a strict order of evaluation of arguments when calling a procedure.

Table 3. Violations distribution in the generated bank.

Violation	Problem Count
The student made a mistake by choosing the left operator with a lower precedence	790,579
The student made a mistake by choosing the right operator with a lower precedence	890,271
The student pressed "Everything is evaluated" when there were still operators waiting to be evaluated	1,428,750
The student made a mistake regarding the effect of parentheses on the order of evaluation	1,041,322
The student tried to evaluate an enclosing operator without evaluating all the enclosed operators first	731,481
The student tried to evaluate first the left of two subsequent right-associative operators	97,808
The student tried to evaluate first the right of two subsequent left-associative operators	1,201,334
The student did not take into account the strict order of operand evaluation of some operators	395,208

The data about values of the control operands are not present in the source code; therefore, the learning-problem generator generates different sets of values for them, creating 2^N learning problems from the same template expression, where N is the number of control operands. Then the metadata for the resulting set of problems are calculated, and the problems are sorted according to the required skills and possible errors. To avoid storing a large number of similar problems, only one problem with each unique set of skills and possible errors is saved to the problem bank.

Some of the expressions used for generating learning problems can be very large: crawling open-source repositories, we found expressions with more than 100 operators. Doing too many similar steps is impractical for training; it becomes routine work without learning anything new. It was decided to avoid storing learning problems that have more than 4 steps requiring the same set of skills to solve. This allows some big learning problems that are various; it requires many different skills to solve but blocks even relatively small problems that are repetitive.

Storing learning problems for a multiple-language ITS also required careful consideration. By simply copying every learning problem for each language, we will enlarge the database significantly with little gain. The technique of using the Meaning Tree, described in the next section, allowed us not just to convert learning problems from one language to another, but also to store them in a universal form that should only be carried out once. However, the metadata (required skills, possible violations) of the same expression can vary in different languages, so it is not possible to store the generated problem in a single record either. It was decided to store the universal Meaning Tree, which is the biggest component of the learning problem, in a single record and transform it to the required language on the fly when the learning problem is used; however, the problem metadata are stored separately for each language. This allows us to combine the flexibility of problem search with storing as little data as possible.

3.2. Meaning Tree

To facilitate the development of multi-language systems for generating learning problems, we have developed a solution called a meaning tree, which can represent code in a supported programming language in the form of an abstract syntax tree that can store parts of the program independently of the syntax rules of a particular programming language, but relies only on the semantics of the stored fragment. The abstract syntax tree formed by the meaning tree can be translated into a syntactically correct string in any supported programming language that supports the used programming features. For this study, we developed the engine for building meaning trees from expressions in the C++, Java, and Python languages and generating new strings in the same three languages.

Figure 1 shows the general idea of the meaning tree as a universal intermediary representation of expressions in different programming languages. It takes expressions in different programming languages as input and can produce output in all the languages the expression's features are supported in. The process is shown in more detail in Figure 2. An expression in the C++ language is first parsed, then converted into the meaning tree where the operators are presented in their abstract form. As a result, the generated expressions in all the languages have the pointer expressions of array element access converted to the square brackets syntax (if desirable for learning, the pointer access operator can remain in the C++ version); the Python version has changed parentheses to reflect the difference in the operator precedence between the languages. The system recognized a typical pattern for compound comparisons and correctly generated them for the Python language. So it produces expressions on all the target languages without losing the semantics of the original expression, which can be used to generate learning problems.

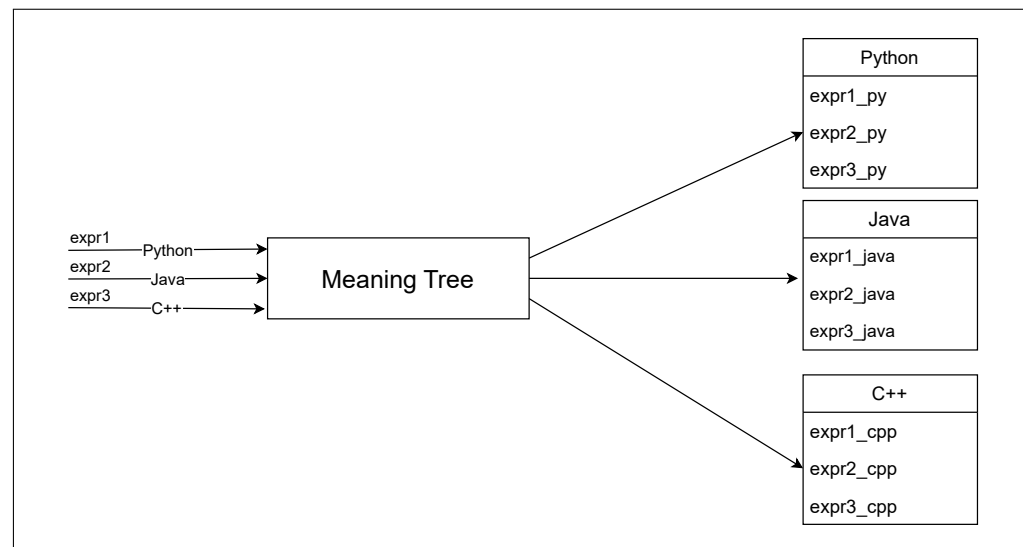


Figure 1. The concept of meaning tree.

Many programming languages have unique features [12] that cannot be easily translated to many other languages; for example, the C++ language has a feature of direct working with memory through pointers and the comma operator; Python has the matrix multiplication operator and so on. So it is important to map possible features of each language and store the features used in each meaning tree to determine if the conversion to the desired target language is possible. Table 4 lists features that are supported in the Meaning Tree software and their support in different programming languages. An expression is converted if it does not contain any feature that is unsupported in the target language. The list of features can be easily extended when adding support for more programming languages to the system.

Table 4. Meaning Tree features and their support by programming language viewers of the Meaning Tree Library

Feature	C++	Python	Java	Feature	C++	Python	Java
Basic Arithmetic Operators	+	+	+	Subscript Operator ([])	+	+	+
Basic Comparison Operators	+	+	+	Array item access with pointer arithmetic	+	$\pm^1$	$\pm$
Compound Comparison	+	+	+	Pointer Member Access	+	$\pm$	$\pm$
Floor Division	+	+	+	Pointer Indirection Operator	+	–	–
Exponentiation (**)	$\pm^1$	+	$\pm$	Pointer Address-of Operator	+	–	–
Logical Operators	+	+	+	Assigning a value to a pointer	+	$\pm$	$\pm$
Bitwise Operators	+	+	+	Object Member Access	+	+	+
Assignment (=)	+	$+^3$	+	Function Call	+	+	+
Augmented Assignments (+=)	+	$+^3$	+	Method Call	+	+	+
Increment/Decrement Operators	+	$+^3$	+	New Object Expression	+	+	+
Comma Operator	+	–	–	New Array Expression	+	+	+
Matrix Multiplication (@)	–	+	–	Delete Dynamic Object Expression	+	–	–
Membership Test Operator (in)	–	+	–	Size-of Operator	+	–	–
Identifiers	+	+	+	Type Cast	+	+	+
Integer, Float, Boolean Literals	+	+	+	Reference Equality Operator	$\pm$	+	+
String Literals	+	+	+	Instance Checking Operator	$\pm$	+	+
Interpolated String Literals	–	+	+	Ternary Conditional Operator	+	+	+
Collection Literals	+	+	+	Three-way Comparison Operator (<=>) ²	+	–	–
Collection Comprehensions	–	+	–	Lambda Expressions	completely unsupported		
Collection Slice Expression	–	+	–	Yield Expression			
				Await Expression			

“+” means: the feature is supported. “–” means: the feature is not supported. ¹ “ $\pm$ ” means: this language does not support the feature natively and a workaround is used when converting from another language. ² Available since C++20. ³ Python expression assignment (“walrus” operator) may be used.

Different programming languages can give different orders of precedence for the same operators. So converting expressions sometimes requires adjusting parentheses. The meaning tree representation stores the exact order of expression evaluation, so it does not need parentheses; instead, the parentheses are generated when converting the meaning tree to a string in a particular language. You can see an example in Figure 2. The Python representation of the input expression requires additional parentheses around $z == k$ because the bitwise operator $\&$ in Python has precedence over the equality operator.

Let us consider some examples of features of the supported programming languages and conversions between them. For example, the Python programming language supports the compound comparison feature—special representations of comparisons in the mathematical style that is not supported by other languages. For example, the following Python expression:

$$a < b \text{ and } b < d \text{ and } m > d \quad (1)$$

can be converted to

$$a < b < m < d, \quad (2)$$

which makes it shorter and easier to read, which makes compound comparisons popular. Our meaning tree representation supports transformations of these comparisons in both directions, which is not trivial. This transformation is achieved by constructing a directed acyclic graph (DAG) for each binary comparison operand, with the edge direction reflecting the kind of comparison, and separating weakly connected graph components from it to convert it into a chain. An example of this transformation is shown in Figure 3. The reverse transformation from Python is achieved by adding a special common node representing several binary comparisons, which are combined by the operator *and*.

Another feature of Python is that it considers assignment a statement, unlike Java and C++, in which assignment is an operator. Python 3.8 introduced the assignment operator `:=` (also called the “Walrus operator”), which cannot be used as a separate statement. So when converting Python assignment statements to other languages, the assignment statement can be converted to the assignment operator. When converting expressions from the C++ and Java languages to Python, the meaning tree converts the root assignment as a statement while other assignments become Walrus operators (see Table 5).

Another example of a language-specific feature is working with pointers in the C and C++ languages. Java and Python languages do not directly support pointers, but objects and arrays are stored and passed by reference. Therefore, it is important to ensure the correctness of the conversion of pointers from the C++ code so that the semantics of the expression are not lost. Although pointers cannot be converted directly to Java or Python, many typical usages of them can. For example, the C++ language allows access to array items using pointer arithmetic, which can be converted into the indexing operator (see Figure 2). Passing the address of an object to a function can be converted from the C++ language as passing the reference, memory allocation as the “new” operator, and field (method) access through a pointer as a regular member access in the Java and Python languages.

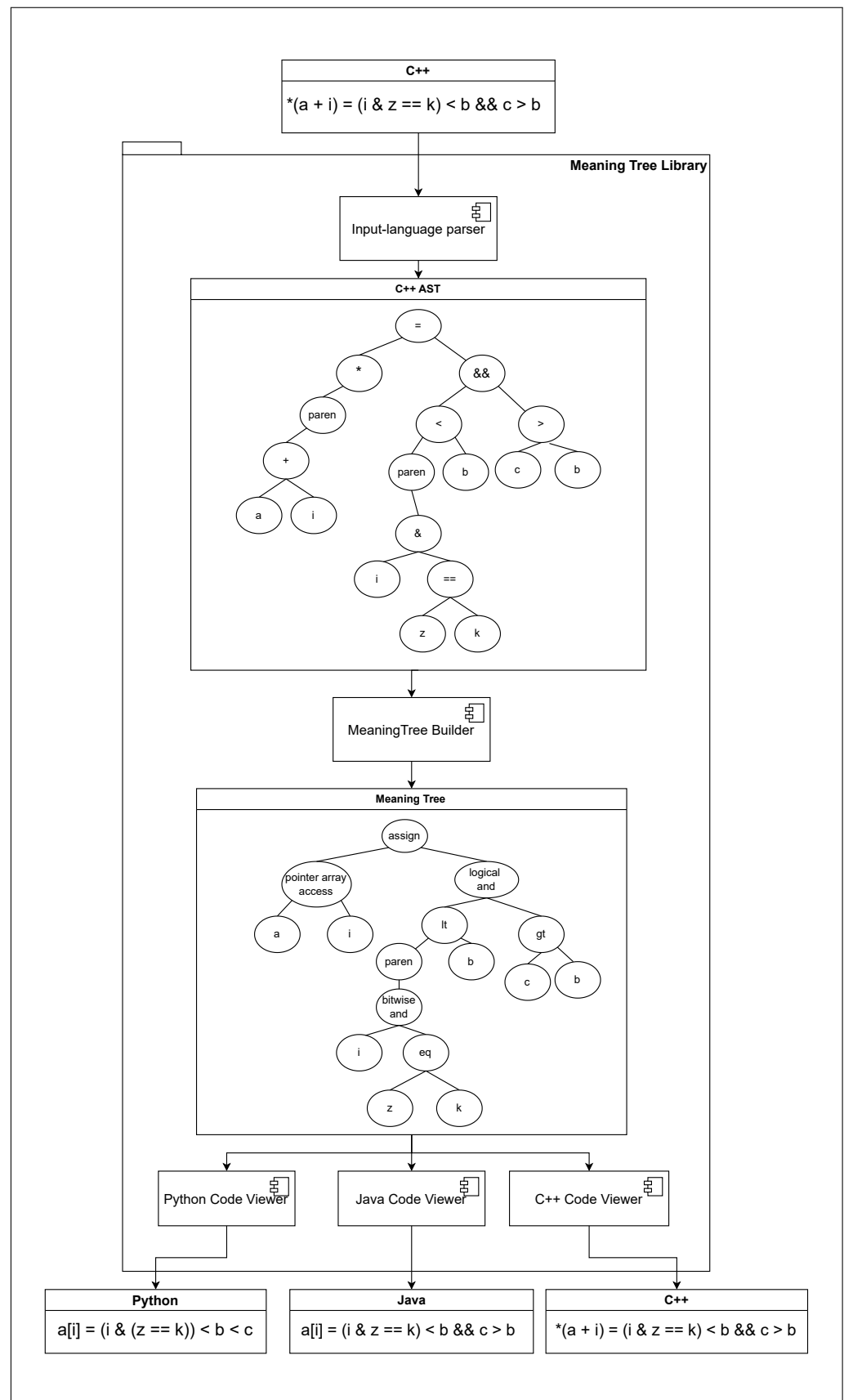


Figure 2. Example of Meaning Tree conversion process.

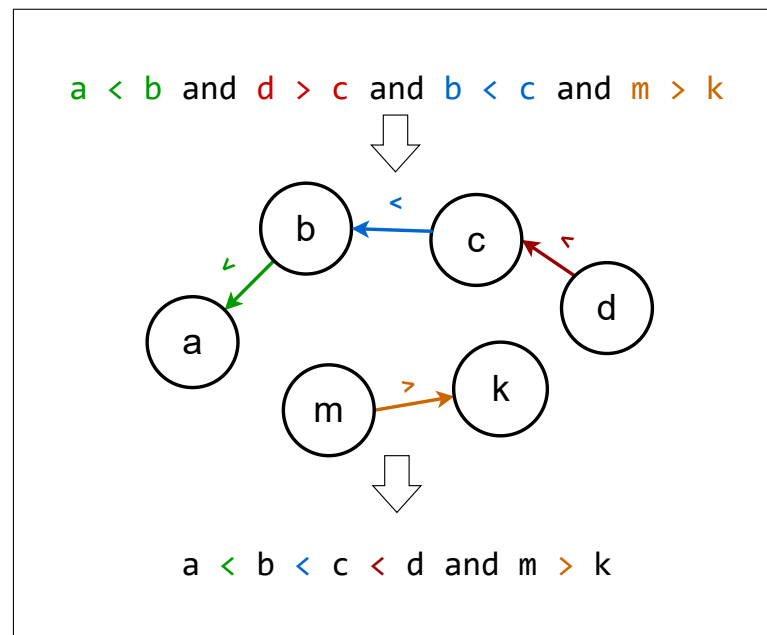


Figure 3. Example of transformation of a Python expression with compound comparison using Directed Acyclic Graph.

Some of the non-trivial conversions are shown in Table 5. For example, unlike Python, the Java and C++ languages do not have a built-in operator for exponentiating numbers. This operator is replaced by a call to the relevant function of the standard library. Floor division can be implemented by converting the result of the division operator to the integer type. Since Python supports list literals and dictionaries, it is necessary to represent them in other languages; this feature can also be implemented using standard libraries.

Table 5. Examples of some non-trivial conversions.

Feature	Python	Java	C++
Exponentiation	<code>a ** b</code>	<code>Math.pow(a, b)</code>	<code>pow(a, b)</code>
Floor Division	<code>a // b</code>	<code>(int)(a / b)</code>	<code>(int)a // b</code>
Assignment	<code>a = 5</code> <code>func(a := 5)</code>	<code>a = 5;</code> <code>func(a = 5)</code>	<code>a = 5;</code> <code>func(a = 5)</code>
Reference Equality Comparison	<code>a is b</code>	<code>a == b</code> (both a and b have reference types)	<code>&a == &b</code>
Instance Checking Expression	<code>isinstance(a, int)</code>	<code>a instanceof Integer</code>	<code>dynamic_cast<int>(a)</code> <code>!= nullptr</code>
Dynamic Collection (List)	<code>[1, 3, 4, 5]</code>	<code>new ArrayList<>(List.of(1, 3, 4, 5) ¹)</code>	<code>{1, 3, 4, 5};</code> (needs casting to <code>std::vector</code>)
Map (Dictionary)	<code>{"a": b, "x": y}</code>	<code>new TreeMap<String, Object>() {put("a", b); put("x", y);}</code>	<code>{{"a", b}, {"x", y}};</code> (needs casting to <code>std::map</code>)

¹ "List.of": supported since Java 9.

Some of the language features are not supported by the meaning tree representation because they are not interesting for the currently generated learning problems, and their conversion between programming languages is problematic. Pointer operations in the C and C++ languages cannot generally be expressed in other languages that do not support pointers, but some of their usages (e.g., dynamic arrays and storing objects by references) can. Examples of unsupported features are Lambda expressions, list comprehensions, and function pointers: even if they are present in each of the supported languages, their correct mutual mapping is beyond the scope of this study.

The developed meaning tree model allows us to increase the number of generated learning problems by converting expressions from different languages so that a diverse

set of learning problems for each learning situation in every language can be generated, making problem banks for each language more balanced.

3.3. Implementation

The system that was developed for generating learning problems consists of three modules: the learning problem generator itself, the problem bank to store the generated problems and relevant metadata, and the Meaning Tree library that is used to convert pieces of code between the supported programming languages. The overall problem generation is performed according to this diagram in Figure 4.

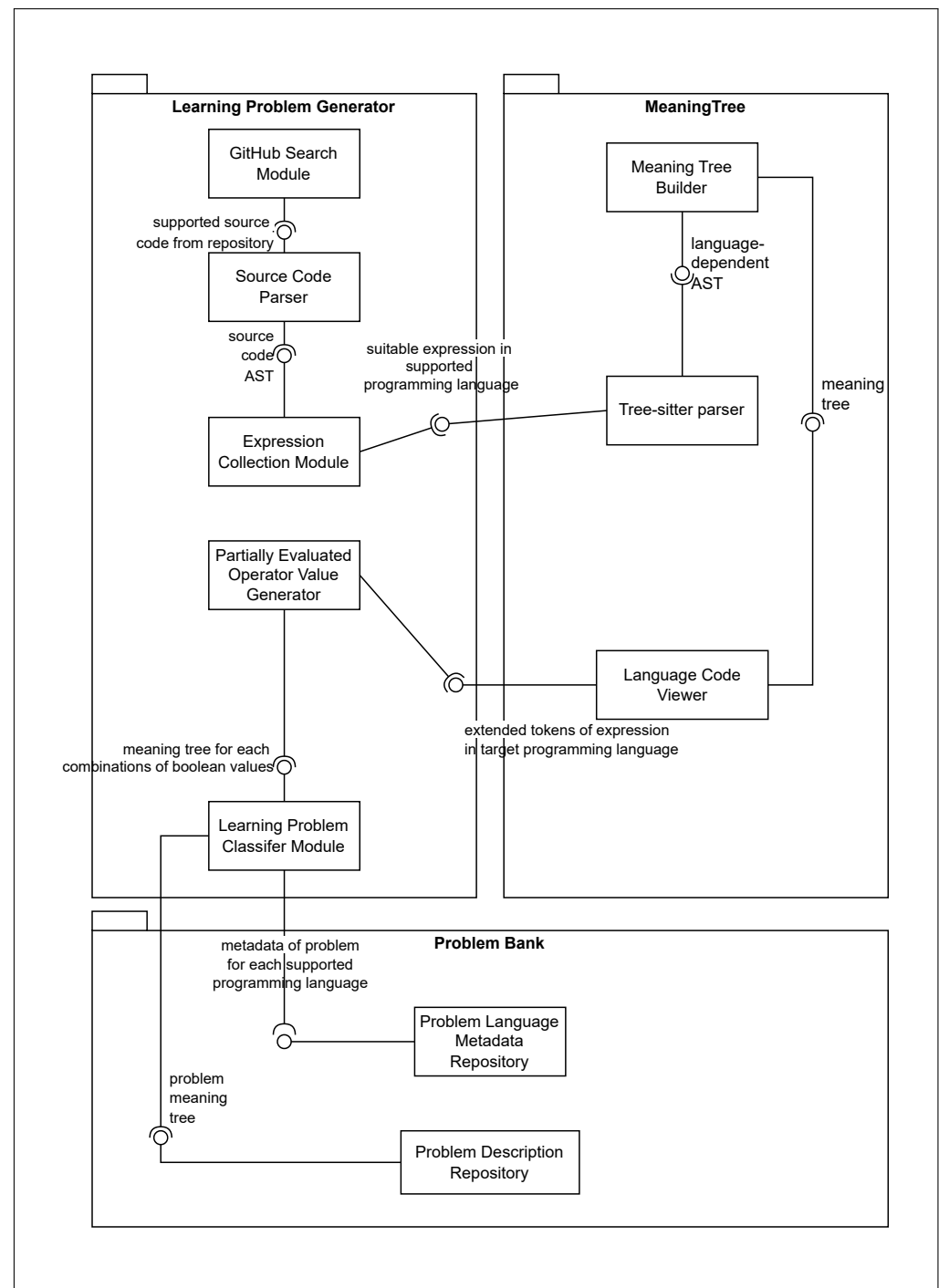


Figure 4. Component Diagram of the Learning Problem Generator.

The learning problem generation begins with downloading and parsing source code described in [32] with adjustments necessary to support different input languages. The first step is to analyze popular GitHub repositories in Python, Java, and C/C++ using the GitHub Search API. Popular repositories are selected as follows: the first 1000 repositories are selected based on the number of “stars” in GitHub, which characterizes their popularity, and the following repositories are selected based on the last update in the repository. The number of repositories was limited based on the number of problems needed for the problem bank. The repository data is downloaded to the local machine using the GitHub Search module.

After the repository is downloaded, the parser runs on each source code file. The programming language and grammar used are determined by the file extension. To standardize parsing of different programming languages, we used the Tree-sitter incremental parser, which supports a large number of programming languages [33]. The input program code is first fed to Tree Sitter, configured with a grammar of the given programming language; it creates a language-specific AST. Then subtrees are extracted from that AST according to the kind of learning problems we need to generate—for this study, those were expressions. Extracted subtrees are passed to the Meaning Tree library if all the language features used in that expression are supported. If the analyzed file contains syntax errors, its correct content will be analyzed because Tree-sitter can recover from syntax errors, and the relevant expressions will be processed. A depth-first traversal of the AST obtained from the Tree-sitter is performed to create language-independent nodes of the meaning tree. The resulting nodes preserve the meaning of the parsed expression and the order of its evaluation.

Meaning Tree is a modular library and command-line utility that can be easily integrated into other projects requiring the conversion of simple parts of the program between programming languages. The library provides the following interfaces that should be implemented to support new programming languages:

- language parser: translates the code in a supported programming language to a meaning tree (language-independent AST);
- language code viewer: generates code in the supported programming language from a meaning tree;
- tokenizer: generates a stream of tokens in the supported programming language from a meaning tree.

Developers can implement these interfaces for a programming language they need and thus expand the support of programming languages in the Meaning Tree library. For this study, we developed modules for three programming languages that are most often used in introductory programming courses: Java, C++, and Python. The Meaning Tree Builder module uses an implementation of individual languages to build a language-independent meaning tree.

The meaning tree is serialized in the RDF Turtle 1.1 format, which, after parsing the repositories, will be used in the generator and stored in the problem bank. Language code viewer modules are used to translate the meaning tree to each supporting language and create language-dependent data that can be used to precisely classify the learning problem and show the problem formulation to the tutor users. In this step, all expressions are deserialized and tokenized to each supported language.

Static data are not enough to determine the order of evaluation of some expressions. So in the next step, run-time values are generated for the necessary operators to determine which parts of the expression are evaluated if necessary for the given expression. This allows for the creation of several learning problems from the same template (expression). Each of the generated problems is classified according to the used concepts, subject-domain laws, and required skills, which can be different because parts of the expression may be

omitted. For each problem template, only one problem from each set of problems with the same concepts, laws, and skills is saved; that allows avoiding having too similar learning problems. Finally, the metadata for each problem are stored in the metadata repository in the problem bank, and the relevant meaning tree with attributed run-time values is stored in the problem repository. The meaning tree (or language-dependent syntax tree) takes a lot more disk space than the metadata, so it is stored once for each source expression; the representation in the specific language is generated on the fly when the online tutor requests a learning problem.

4. Evaluation

4.1. Sample

We performed a massive generation of learning problems concerning the order of evaluation of operators of programming-language expressions. The generated learning problems can be used in the ITSs CompPrehension [34] and HowItWorks. Figures 5 and 6 show examples of the generated learning problems solved in the user interface.

The screenshot shows a user interface for solving a programming problem. At the top, a light blue box contains the instruction: "Press the operators in the expression in the order they are evaluated". Below this, a code expression is displayed with operators underlined and numbered 1 through 29. The expression is: `argc = 5 + copy + replace + (user || pass ? 1 : 0) + (user != null) + (pass != null)`. Below the expression, a text box says "everything is evaluated". Underneath, a list of 10 steps shows the order of operator evaluation, with the position and the operator being evaluated. The steps are: 1. operator = at pos 1, 2. operator + at pos 23, 3. operator + at pos 7, 4. operator + at pos 17, 5. operator != at pos 26, 6. operator != at pos 20, 7. operator + at pos 5, 8. operator || at pos 10, 9. operator + at pos 3, 10. operator = at pos 12. The final step shows the result: false. At the bottom, a green box says "Solved".

Figure 5. Example of solving a problem with Java/C++ expression in the ITS. The underlined symbols are expression operators that the learner must press in the order of their evaluations. The numbers above are the positions of the expression operators, and the numbers below prefixed with the # sign are the order of evaluation of the operators that has already been pressed by the learner.

The learning-problem generation process ran unsupervised for about 16 h; 1360 open-source projects were used, which resulted in generating 1,428,750 learning problems. Out of them, 34.47% of learning problems were generated in the C++ language, 32.84% in the Java language, and 32.69% in the Python language.

Table 6 shows descriptive statistics of the generated problem bank. Learning-problem difficulty was calculated using the method described in [32]. You can see that the mean difficulty, which was normalized to the range from 0 to 1, is rather high, but the mean number of solution steps, required skills, and possible subject-domain rules violations is near the middle of relevant ranges. Overall, the generated bank contains learning problems with a wide range of difficulties, used concepts, and required skills. However, the mean concept count is relatively low, which shows that most of the generated learning problems use similar operators. Given the overall number of learning problems in the generated bank, this is insignificant because there are many problems with a high skill count in the bank.

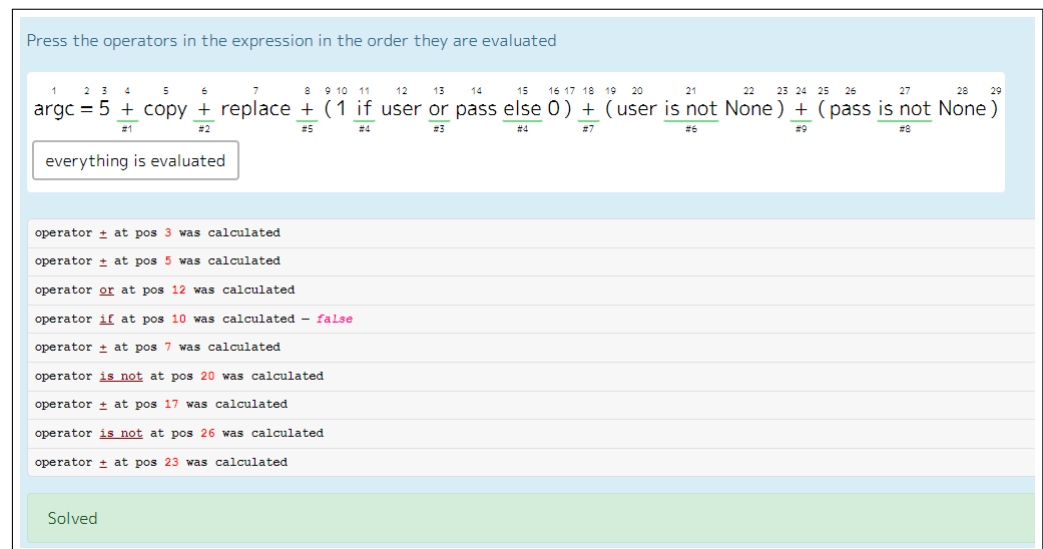


Figure 6. Example of solving a problem with Python expression in the ITS. The underlines and numbers play the same role as in the previous figure.

Table 6. Descriptive statistics on the generated problems.

Property	Minimum	Mean	Standard Deviation	Maximum
Difficulty	0.22	0.68	0.16	1.00
Solution steps	3	6.04	1.78	17
Skill count	3	10.95	2.82	21
Concept count	1	3.82	1.42	16
Violation count	2	7.75	1.85	14

Table 1 shows the number of learning problems that contain different kinds of operators. Most of the operators are present in tens of thousands (or more) of learning problems, except rare operators like assignments augmented by division and bitwise shifts, unary plus (that does not do anything in 2 of the 3 source languages), and decrement operators.

Table 3 shows the amount of generated learning problems, where different kinds of domain-rules violations are possible. Table 2 shows the number of learning problems that require each skill developed in the tutor in the order of evaluation. The lowest number of learning problems were generated for checking the operators with right associativity (which are few), but almost 98 thousand problems is more than enough to conduct learning in any reasonable group of students. Regarding the necessary skills, which give finer granularity than violations because they take into account different causes why the answer was correct, the only small number is for the skill regarding a strict-order operator that always evaluates all its operands. There is only one such operator—the comma operator in the C++ language, which is used very rarely.

We also gathered the data on the distribution of the number of concepts, violations, and skills in learning problems; you can see them in Figures 7–9. The unusual distribution of problems with different numbers of skills can be explained by the fact that some skills are often present together, so problems requiring very low numbers of skills are few. Problem-solving is a complex activity that cannot be covered by 1–2 basic skills per problem.

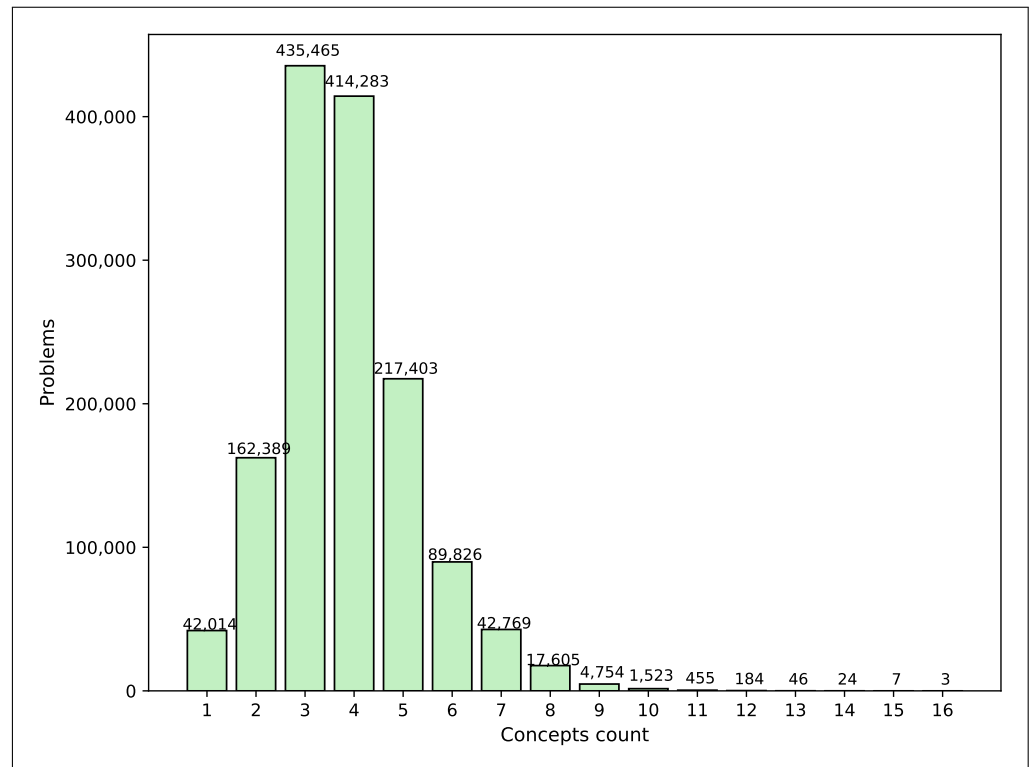


Figure 7. Distribution of learning problems per number of concepts.

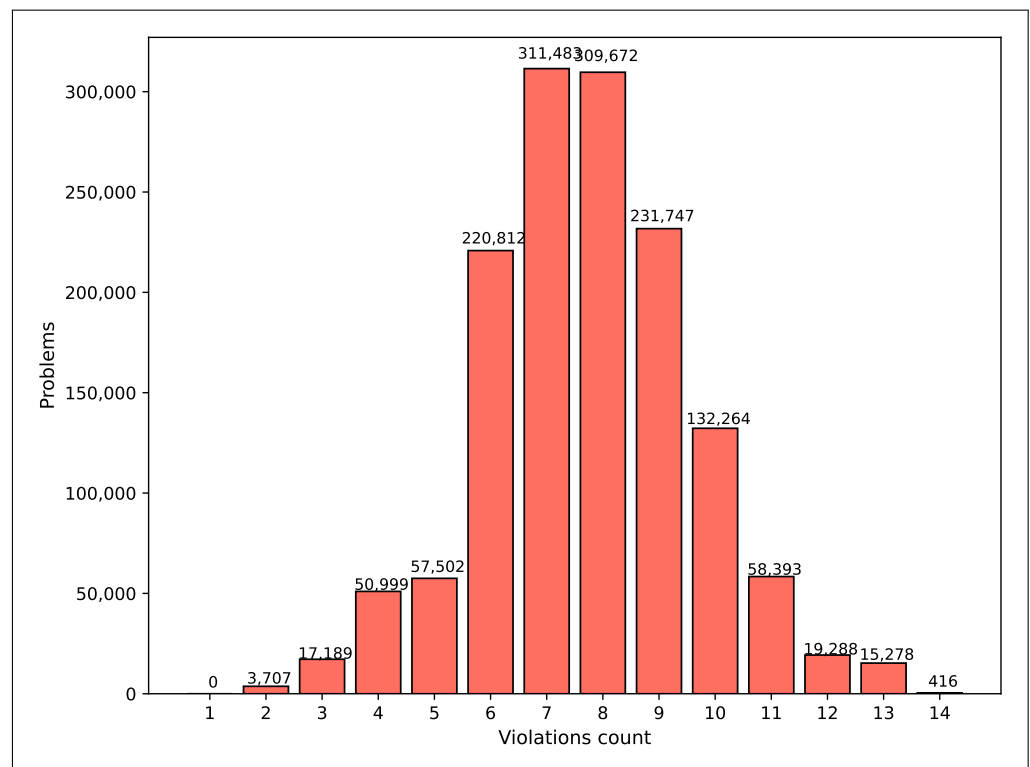


Figure 8. Distribution of learning problems per violations of subject-domain rules.

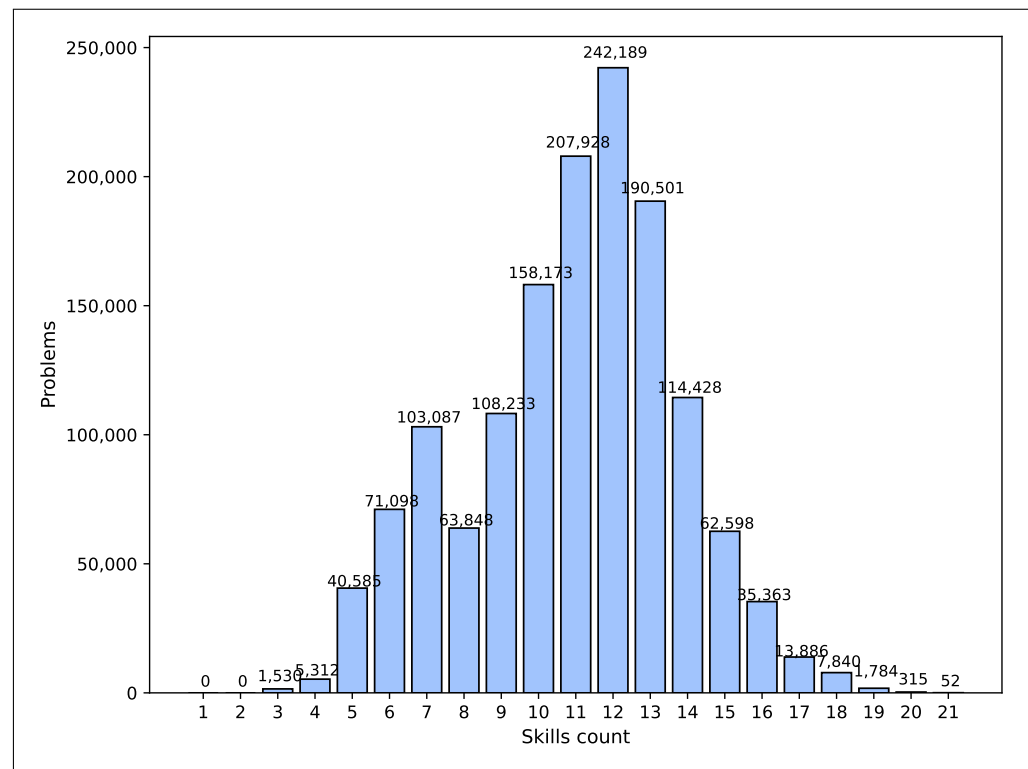


Figure 9. Distribution of learning problems per number of required skills.

4.2. Testing and Validation

Big data software places particular emphasis on testing software and validating the data because it cannot be done manually, and researchers must rely on the results of automatic procedures.

The chief new piece of software developed for this research was the Meaning Tree library, which performs the complex task of converting parts of the program from one programming language to another. Currently, there are many machine-learning-based solutions for this task, but their accuracy remains insufficient for the mass generation of learning problems that can be used without further human inspection [25]. So we developed a set of 417 unit tests for the Meaning Tree library, covering the operators used in all the supported languages. Each test contains an expression in a certain programming language as a text string. It is converted to the two other supported languages using the tested library and then compared with the expected output. 158 unit tests were also reversed to verify that the conversion was symmetric. This testing allowed us to ensure the correctness of the conversion in the Meaning Tree library.

Another important validation problem was ensuring accurate classification of the generated learning problems. This cannot be made solely through unit tests because covering all the different situations possible in the learning problems by manually written (or verified) tests is practically impossible. Instead, automated testing was used.

We wrote program code that imitates the user's actions in the ITS to model all the possible solution paths and gather information on the violations of subject-domain rules that happened and the skills used by the ITS core to solve the problem. Both correct and incorrect paths were explored, with the ITS limitations that two chained errors in one solution are impossible because erroneous steps are not added to the solution but trigger immediate feedback. That validation allowed us to verify the classification system and to be sure that there were no bugs in the problem classification of the big dataset. The validation process of 1.4+ million learning problems took about 2.5 h.

4.3. Evaluation by Teachers

We asked 10 experts, who were teachers and teaching assistants in introductory programming courses, to evaluate the quality of the generated problems in teaching introductory programming courses. They were given the ability to request learning problems by setting the programming language with the tags subsystem, problem difficulty, and desired (target), allowed, and denied concepts, skills, and domain-rule violations, which were called “laws” for better usability (see Figure 10). Then they could try the found learning problems to evaluate how well their requests were satisfied. The experts could try as many combinations as they wanted. Then they had to complete a short survey, with most of the questions using a Likert scale from 1 to 5. They were also given a free text field to write their thoughts on the generated learning problems.

Tags

☐ Java
☐ C++
☒ Python

Stage

Stage #1 Questions in bank: 28536 (28525)

Number of questions

10

Question difficulty

0.85

Concepts

Denied Allowed Target Arithmetic operators

Denied Allowed Target Array and collection literals

Denied Allowed Target Comparison operators

Denied Allowed Target Increment and decrement

Violations

Denied Allowed Target Left-associative binary operators

Denied Allowed Target Left-associative binary operators

Denied Allowed Target Right-associative binary operators (assignment)

Skills

Denied Allowed Target Understand that multiple central arguments (function arguments, array elements) are evaluated strictly from left to right in the current programming language

Denied Allowed Target Distinguish between binary and unary operators

Question examples

not_o2_=_pcb_find(o,_start,_None))_or_not_o2.values[0].isnum_python1496109156_v

(not_demo_compatibility_and_not_mbf_features_and_not_prboom_comp_python1930411961_v

gb.debug_active=_not_gb.debug_disable_and_(gb.debug_stopped_or__python-2124895434_v

(info.attrs[WGDEVICE_A_LISTEN_PORT])_or_info.attrs[WGDEVICE_A_FWM_python460128949_v

Add stage

Figure 10. Overview of problem-selection settings.

Most of the questions were formulated so that answer 5 was positive, but two of them (Questions 7 and 8) were inverted to ensure the consistency of the obtained data. Question 5 had a neutral scale; its goal was to study which kind of classification of learning problems the domain experts prefer: violations of rules of the subject domain or skills. The survey results are presented in Table 7.

Table 7. Survey of experts.

#	Survey Question	Boundaries on 5-Point Scale	Avg.	Std. Dev.
1	Do the learning problems presented help develop skills in determining the order of evaluation of operators in an expression?	(1) Don't contribute at all (5) Contribute to the full extent	4.5	0.5
2	To what extent do changes in the difficulty of the problems correspond to changes in the difficulty settings?	(1) Does not match (5) Matches exactly	4.1	0.94
3	Is the learning problem classification used flexible enough to allow you to select learning problems with any desired properties?	(1) Not flexible at all (5) Very flexible	4.2	0.98
4	Do the Skills and Violations list all the necessary options for the topic "The Order of Evaluation of Expressions"?	(1) Many are missing (5) For all occasions	4.6	0.49
5	What do you think is preferable for selecting learning problems: Violations or Skills?	(1) Violations are much more convenient (5) Skills are much more convenient	3.7	1.1
6	To what extent do the concept settings allow you to avoid receiving learning problems with operators that students are not required to know?	(1) Never (5) Always	3.7	1.0
7	How much do the problems generated by the program differ in appearance from those compiled by a human?	(1) Indistinguishable (5) Very different	2.6	1.11
8	How often did you see learning problems that use expressions that are atypical for the requested programming language — the learning problem that you would like to avoid?	(1) Very rarely (5) Very often	2.0	1.0
9	Are the problems ready to be given to students?	(1) Require manual selection (5) Fully suitable	4.1	1.14

The experts evaluated the generated bank positively. They strongly agreed that the generated learning problems help to develop skills in determining the order of expression evaluation and that the provided classification lists all the necessary options to select learning problems with any desired properties. They agreed with many other statements, and the average ratings of the two "inverted" questions were below 3. The "lowest" ratings were received in the questions about how different the generated problems were from human-authored ones (with the experts slightly favoring the "indistinguishable" options) and the concept classification that was intended to prevent students from receiving expressions with operators they did not know, which may indicate that the list of concepts needs improvement. Between violations and skills as methods of classifying learning problems, more experts chose skills than violations, but the result is close to the average, so it is better to keep both classifications to let particular teachers use the methods they like best.

In the free-text responses, the experts noted that the system of skills was too complex and required a more thorough explanation, including warning teachers about the skill setting combinations that are impossible to satisfy (that is, if any learning problem that requires Skill 1 also requires Skill 2, it is not desirable to make Skill 1 target and make Skill 2 denied). Their opinion about the estimation of the difficulty of problems was different:

some experts found the problems with difficulty below 0.5 too similar to each other, while other experts found the problems with difficulty above 0.5 too similar to each other.

5. Discussion

During this study, a bank consisting of more than 1 million learning problems related to the order of expression evaluation from open-source projects was generated. The generation of such a vast number of problems was made possible by the implementation of Meaning Tree, a library that can translate pieces of code from one programming language to another. This allowed us both to use more repositories of open-source code as input for generating learning problems and to generate more problems from each repository. This let us overcome the limitations of existing works on learning-problem generation. Unlike other researchers, we did not rely on teacher-created templates that limit the diversity of generated learning problems, nor did we rely on random generation, which often produces artificial-looking code. Using parts of real, human-authored program code allowed us to both maintain a high diversity of generated problems and make them look natural.

Other researchers also stressed the need to classify learning problems during their generation. Kumar [30] classified the generated learning problems according to the 40 “learning objectives” he defined. They describe possible errors in understanding the semantics and syntax of a programming language, which is similar to the “violations” used in this study. However, we do not limit ourselves to possible errors but also suggest that teachers classify problems according to the skills required to solve the problem, which takes into account different ways of being right, not only different ways of being wrong as Kumar’s learning objectives do. So the findings of Kumar were improved. In the CompPrehension ITS [34], the verification of learning problems is based on the formal description of the verification process in the form of a Thought Process Tree or Graph [35] and can be defined objectively according to the subject-domain rules, instead of being selected subjectively by a teacher. This is one of the new features of our approach compared to the approaches of other researchers.

Okonkwo et al. [16] and Ade-Ibijola [17] used a problem-generation method based on template modification based on context-free grammars. The Okonkwo study generated 120,330 problems, and the Ade-Ibijola study generated 500,000 problems, which is smaller but comparable to our results. However, the learning problems generated by their approach look artificial (especially variable names) and have low diversity because they are based on the randomization of a few templates. Our approach generates more diverse learning problems because it is based on human-authored code, which is an improvement over the works of other researchers. To assess the diversity of the generated bank, we calculated the number of unique kinds of problems in the generated bank. The learning problem was considered unique if it had 122,202 unique combinations of used concepts, possible domain-rule violations, and required skills. This is a very high diversity that is practically impossible to achieve by template-based methods because it requires human authoring of more than 120 thousand templates.

In other studies that contained information about the number of learning problems generated, it did not exceed 1000. For example, Kumar [30] noted that only 50 learning problems were evaluated during the study. Martin and Mitrovic [9] generated less than 200 learning problems. They noted that generating that amount of problems required 3 h of human time. The generation process was not fully automated, which did not allow them to scale it to the level of generating large banks of learning problems as the method we propose does.

The amount of the learning problems generated during this study can look very high and impractical. However, several moments must be taken into account when analyzing it. First, the bank is generated in three programming languages that are most often used for teaching introductory programming courses; the amount of learning problems generated in each language is less than 500,000. Second, intelligent tutoring systems can use fine-grained measures of learning problems, taking into account used concepts (that are useful for filtering the problems containing operators that the learners are not expected to know), domain-rule violations, and required skills (needed to measure the learners' mastery and adapt the sequence of problems to each learner's situation). Analyzing the generated bank, 122,202 unique combinations of concepts, violations, and skills, which gives us on average only 11.76 learning problems per combination. That shows that the learning bank is diverse, not just big. Also, as we are presenting a new method of learning problem generation, we were interested in evaluating its scalability, which required generating a bigger learning problem bank that may be necessary in particular situations.

The size of the problem bank also allowed us to include some rare kinds of problems. For example, to verify the skill if the learner can determine the existence of a central operand, the expression must contain at least one operator without a central operand. There are only 35,649 of such problems in the bank of 1.4 million, which comprises about 2.5% of the bank. Another example is the skill to determine if the competing operator to the left or to the right of the given operator exists; the competing operators are almost always absent for the leftmost and rightmost operators in the expression. The only exception is the expression that starts from a prefix operator (that do not have a left operand) and ends with a postfix operator or square brackets (that does not have a right operand) because the previously used skill is enough for them. Our generator created 50,493 learning problems of that rare kind, which comprises 3.5% of the problems in the bank. Note that while the number of problems that do not require a single skill is relatively high, we should take into account the problem diversity concerning other required skills and used programming languages, which severely limits the number of problems with a particular skill combination. In smaller banks, the number of those problems for each language might be too small, which can cause repetitions of the same problems for the same group of learners, or, worse, for the same learner.

The generated bank of learning problems can be filtered to create smaller banks for particular courses, limiting learning problems by the programming language, studied topics, and so on. The amount of generated problems allows us to implement the idea "never repeat the same problem in the same year" that prevents such forms of cheating as remembering a solution to the problem the learner encountered before or sharing solutions among the learners. It can also be used in Massive Open Online Courses (MOOCs) that can attract large cohorts of students. For example, Burd et al. report that about 3500 students attended the MIT MOOC on introductory programming in just two semesters in 2016–2017 year [36]. Similarly, Duran et al. write about 3900 students participating in a Finnish MOOC on introductory programming [37]. Courses with thousands of students that can freely communicate online require larger banks of learning problems both to avoid cheating and to individualize the learning process, giving everyone a set of problems adapted to their unique situation.

6. Threats to Validity

Automatic generation of big datasets is the kind of research that cannot be fully verified manually because of the huge amount of effort that is required. So studies like that have to rely on manual validation of a randomly selected subset of problems and automatic validation of the entire dataset by two independent algorithms with the analysis of every case of differences. In our case, we implemented the algorithm for learning-problem classification and used the ITS code to validate it. An ITS based on thought process graphs can report the exact kind of domain rule violations and skills used for that validation. We analyzed all possible sequence actions of learners for each learning problem and compared the violations that happened and skills used to the learning-problem classification. This proves that the classification on violations and skills was performed correctly. Still, it is possible that some errors might have remained unnoticed, which poses a threat to the study's validity. For example, the problem difficulty was calculated using the approximate function developed according to teachers' difficulty scores as described in [32] for a smaller bank of similar problems. Further work can improve the precision of difficulty estimation on large problems banks, like the one presented here.

Another threat to the study's validity comes from the relatively small evaluation performed by human experts. It is better to evaluate the properties of the generated learning problems with students during actual learning, but this can only happen at certain times of the year when the students study the relevant topic and the number of participants is limited. Full evaluation with students for a problem bank of this size may take several years. It will be one of the focuses of our further studies. Still, by employing experts from different universities, we partially mitigated that threat; the experts used different programming languages in their courses, which contributed to gathering diverse opinions.

The survey of experts used the Likert scale that can be subject to biases. The survey was balanced by including "inverted" questions (where the higher rate means a worse opinion) to avoid acquiescence bias; the results show that the answers of experts were free from the acquiescence bias.

7. Conclusions

We presented an enhanced method of generating learning problems for the computer science domain that takes into account differences between programming languages using the language-independent representation of program elements that we called the "meaning tree". The method allows fully automatic generation of learning problems that can be classified and given to learners without the need for further verification or improvement by humans, which makes the method scalable to the big data field of study.

Using it, we performed large-scale learning-problem generation by generating a problem bank of 1,428,899 learning problems in the field of the order of evaluation of expressions in three programming languages that are most used in introductory programming courses: C++, Java, and Python. It took only about 16 h. The problems were classified according to the used concepts (kinds of operators), required skills, violations of subject-domain rules that are possible during problem-solving, difficulty, and the number of steps required to complete the problem. To the best of our knowledge, this is the first learning-problem generation of that scale for multi-language intelligent tutoring systems (ITS). The generated learning problems can be used in the ITSs: HowItWorks and CompPrehension. The learning problems were diverse: from small, easy learning problems requiring only 3 skills to solve to complex learning problems with 17 solution steps requiring 19 different skills to solve. Unlike some randomization methods that have been used for large-scale learning problem generation, the problems generated by our method look natural and human-written.

The results of our research are made open to the scientific community. The generated dataset was too big to attach it to the article; it is available at the public folder: <https://drive.google.com/drive/folders/1BBC7gzaFfSpEaE2DAAhunSDjVsWiATl2> (accessed on 27 February 2025). The Meaning Tree library developed during this study is available at https://github.com/bardoor/meaning_tree/tree/expr-stable (accessed on 27 February 2025). The learning-problem generator is available as a part of the CompPrehension project: <https://github.com/CompPrehension/CompPrehension/tree/shashkov/meaning-tree-integration> (accessed on 27 February 2025).

Generating such large banks of learning problems requires automatic validation techniques to avoid errors in problem generation and verification. To validate the problem bank, we tried every possible path for solving each problem and created sets of factual skills used and subject-domain rule violations that occurred, which were compared with the results of our problem classifier to ensure its accuracy. The learning problem bank proved to be fully valid.

The surveyed experts (teachers and teaching assistants in introductory programming courses of two universities) agreed that the generated problems allow the development of topic-related skills and are ready to be given to students without further filtration or changes. The classification system was deemed flexible enough, but it requires more documentation and fine-tuning of the user interface to avoid creating contradictory problem requests that cannot be satisfied.

While the method of learning-problem generation from open-source code requires significantly more effort to develop the generator for each kind of learning problem than large language models (LLMs), the resulting learning problems are generated precisely and ready to use. That allows for generating large problem banks that cannot be filtered manually.

The results presented here close the gap in literature concerning the automatic generation of learning problems in the programming domain spanning several programming languages so that it can be used by different educational organizations in introductory programming courses. The generated problems do not require any selection or modification by humans and can be directly given to the learners. This advances the field of learning problem generation. The results can be used not only in the programming domain but also in other domains by using digital twins of subject-domain entities instead of open-source code.

Our further work includes evaluating the generated learning problems with students of introductory programming courses and expanding the Meaning Tree library to support control-flow statements, functions, and other common elements of programming languages.

Author Contributions: Conceptualization, O.S.; methodology, O.S.; software, D.S.; validation, O.S. and D.S.; writing—original draft preparation, D.S.; writing—review and editing, O.S.; visualization, D.S.; supervision, O.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Ethical review and approval is not required, because this study involved the analysis of data being based on voluntary participation and having been properly anonymized. The research presents no risk of harm to subjects.

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: The dataset of generated learning problems is available from authors; the article contains the link to download the dataset.

Acknowledgments: The authors thank Mikhail Denisov for his advice on working the manuscript and Artem Prokudin for his help in running the software.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
ASP	Answer Set Programming
AST	Abstract Syntax Tree
DAG	Directed Acyclic Graph
ITS	Intelligent Tutoring System
LLM	Large Language Model
MOOC	Massive Open Online Course
RDF	Resource Description Framework
SQL	Structured Query Language

References

1. Ben Arfa Rabai, L.; Cohen, B.; Mili, A. Programming Language Use in US Academia and Industry. *Inform. Educ.* **2015**, *14*, 143–160. [\[CrossRef\]](#)
2. Siegfried, R.M.; Herbert-Berger, K.G.; Leune, K.; Siegfried, J.P. Trends of Commonly Used Programming Languages in CS1 And CS2 Learning. In Proceedings of the 2021 16th International Conference on Computer Science & Education (ICCSE), Lancaster, UK, 17–21 August 2021; pp. 407–412. [\[CrossRef\]](#)
3. Ali, S.; Qayyum, S. A Pragmatic Comparison of Four Different Programming Languages. *ScienceOpen Prepr.* **2021**. [\[CrossRef\]](#)
4. Figueiredo, J.; García-Peñalvo, F.J. Intelligent Tutoring Systems approach to Introductory Programming Courses. In Proceedings of the Eighth International Conference on Technological Ecosystems for Enhancing Multiculturality, Salamanca, Spain, 21–23 October 2020; TEEM’20, pp. 34–39. [\[CrossRef\]](#)
5. Oli, P.; Banjade, R.; Lekshmi Narayanan, A.B.; Brusilovsky, P.; Rus, V. Exploring The Effectiveness of Reading vs. Tutoring For Enhancing Code Comprehension For Novices. In Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, Avila, Spain, 8–12 April 2024; SAC ’24, pp. 38–47. [\[CrossRef\]](#)
6. Nwana, H. Intelligent tutoring systems: An overview. *Artif. Intell. Rev.* **1990**, *4*, 251–277. [\[CrossRef\]](#)
7. Abu Naser, S.S. Developing an Intelligent Tutoring System For Students Learning To Program in C++. *Inf. Technol. J.* **2008**, *7*, 1055–1060. [\[CrossRef\]](#)
8. Naser, S.S.A. JEE-Tutor: An Intelligent Tutoring System For Java Expressions Evaluation. *Inf. Technol. J.* **2008**, *7*, 528–532. [\[CrossRef\]](#)
9. Martin, B.; Mitrovic, A. Automatic Problem Generation in Constraint-Based Tutors. In *Proceedings of the Intelligent Tutoring Systems*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 388–398. [\[CrossRef\]](#)
10. Kumar, A.N. Using problelets for problem-solving exercises in introductory C++/Java/C# courses. In Proceedings of the 2013 IEEE Frontiers in Education Conference (FIE), Oklahoma City, OK, USA, 23–26 October 2013; pp. 9–10. [\[CrossRef\]](#)
11. Fabic, G.V.F.; Mitrovic, A.; Neshatian, K. Adaptive Problem Selection in a Mobile Python Tutor. In Proceedings of the Adjunct Publication of the 26th Conference on User Modeling, Adaptation and Personalization, ACM, Singapore, 8–11 July 2018; UMAP ’18, pp. 269–274. [\[CrossRef\]](#)
12. Khoirom, M.S.; Sonia, M.; Laikhuram, B.; Laishram, J.; Singh, T.D. Comparative Analysis of Python and Java for Beginners. *Int. Res. J. Eng. Technol.* **2020**, *7*, 4384–4407.
13. Farooq, M.S.; zaman Khan, T. Comparative Analysis of Widely use Object-Oriented Languages. *arXiv* **2023**, arXiv:2306.01819. [\[CrossRef\]](#)
14. Lekkila, E.; Christopoulos, A.; Laakso, M.J. A Data-Driven Approach to Compare the Syntactic Difficulty of Programming Languages. *J. Inf. Syst. Educ.* **2023**, *34*, 84–93.
15. Sadigh, D.; Seshia, S.A.; Gupta, M. Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems. In Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education, ACM, Tampere, Finland, 12 October 2012; ESWEEK’12, pp. 1–8. [\[CrossRef\]](#)
16. Chinedu, O.; Ade-Ibijola, A. Synthesis of nested loop exercises for practice in introductory programming. *Egypt. Inform. J.* **2023**, *24*, 191–203. [\[CrossRef\]](#)

17. Ade-Ibijola, A. Syntactic Generation of Practice Novice Programs in Python. In *Proceedings of the ICT Education*; Springer International Publishing: Berlin/Heidelberg, Germany, 2018; pp. 158–172.
18. Gulwani, S. Synthesis from Examples: Interaction Models and Algorithms. In *Proceedings of the 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timisoara, Romania, 26–29 September 2012; pp. 8–14. [\[CrossRef\]](#)
19. Polozov, O.; O'Rourke, E.; Smith, A.M.; Zettlemoyer, L.; Gulwani, S.; Popovic, Z. Personalized mathematical word problem generation. In *Proceedings of the 24th International Conference on Artificial Intelligence*, Buenos Aires, Argentina, 25–31 July 2015; AAAI Press: Washington, DC, USA, 2015; IJCAI'15, pp. 381–388.
20. O'Rourke, E.; Butler, E.; Díaz Tolentino, A.; Popović, Z. Automatic Generation of Problems and Explanations for an Intelligent Algebra Tutor. In *Proceedings of the Artificial Intelligence in Education*; Springer International Publishing: Berlin/Heidelberg, Germany, 2019; pp. 383–395.
21. Shuster, K.; Poff, S.; Chen, M.; Kiela, D.; Weston, J. Retrieval Augmentation Reduces Hallucination in Conversation. *arXiv* **2021**, arXiv:2104.07567.
22. Austin, J.; Odena, A.; Nye, M.I.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.J.; Terry, M.; Le, Q.V.; et al. Program Synthesis with Large Language Models. *arXiv* **2021**, arXiv:2108.07732.
23. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374.
24. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A.D.; et al. Competition-level code generation with AlphaCode. *Science* **2022**, *378*, 1092–1097. [\[CrossRef\]](#) [\[PubMed\]](#)
25. Xue, T.; Li, X.; Azim, T.; Smirnov, R.; Yu, J.; Sadrieh, A.; Pahlavan, B. Multi-Programming Language Ensemble for Code Generation in Large Language Model. *arXiv* **2024**, arXiv:2409.04114. [\[CrossRef\]](#)
26. Pan, R.; Ibrahimzada, A.R.; Krishna, R.; Sankar, D.; Wassi, L.P.; Merler, M.; Sobolev, B.; Pavuluri, R.; Sinha, S.; Jabbarvand, R. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon, Portugal, 14–20 April 2024; ICSE '24. [\[CrossRef\]](#)
27. Eniser, H.F.; Zhang, H.; David, C.; Wang, M.; Christakis, M.; Paulsen, B.; Dodds, J.; Kroening, D. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. *arXiv* **2024**, arXiv:2405.11514.
28. Macedo, M.; Tian, Y.; Nie, P.; Cogo, F.R.; Adams, B. InterTrans: Leveraging Transitive Intermediate Translations to Enhance LLM-based Code Translation. *arXiv* **2024**, arXiv:2411.01063.
29. Fila, M.J.; Eatough, E.M. Extending knowledge of illegitimate tasks: Student satisfaction, anxiety, and emotional exhaustion. *Stress Health* **2018**, *34*, 152–162. [\[CrossRef\]](#) [\[PubMed\]](#)
30. Kumar, A. Rule-based adaptive problem generation in programming tutors and its evaluation. In *Proceedings of the Workshop on Adaptive Systems for Web-Based Education: Tools and Reusability*, 12th International Conference on Artificial Intelligence in Education (AI-ED 2005), Amsterdam, The Netherlands, 18–22 July 2005; pp. 35–44.
31. Sychev, O. From Question Generation to Problem Mining and Classification. In *Proceedings of the International Conference on Advanced Learning Technologies*, ICAIT 2022, Bucharest, Romania, 1–4 July 2022; pp. 304–305. [\[CrossRef\]](#)
32. Sychev, O.; Penskoy, N.; Prokudin, A. Generating Expression Evaluation Learning Problems from Existing Program Code. In *Proceedings of the 2022 International Conference on Advanced Learning Technologies (ICAIT)*, Bucharest, Romania, 1–4 July 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 183–187. [\[CrossRef\]](#)
33. Clem, T.; Thomson, P. Static Analysis at GitHub: An experience report. *Queue* **2021**, *19*, 42–67. [\[CrossRef\]](#)
34. Sychev, O.; Penskoy, N.; Anikin, A.; Denisov, M.; Prokudin, A. Improving Comprehension: Intelligent Tutoring System Explaining the Domain Rules When Students Break Them. *Educ. Sci.* **2021**, *11*, 719. [\[CrossRef\]](#)
35. Sychev, O. Educational models for cognition: Methodology of modeling intellectual skills for intelligent tutoring systems. *Cogn. Syst. Res.* **2024**, *87*, 101261. [\[CrossRef\]](#)
36. Burd, H.; Bell, A.; Hemberg, E.; O'Reilly, U.M. Analyzing Pre-Existing Knowledge and Performance in a Programming MOOC. In *Proceedings of the Seventh ACM Conference on Learning @ Scale*, Virtual Event, USA, 12–14 August 2020; L@S '20, pp. 281–284.
37. Duran, R.; Haaranen, L.; Hellas, A. Gender Differences in Introductory Programming: Comparing MOOCs and Local Courses. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, Portland, OR, USA, 11–14 March 2020; SIGCSE '20, pp. 692–698. [\[CrossRef\]](#)

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.