

Write a Line: Tests with Answer Templates and String Completion Hints for Self-Learning in a CS1 Course

Oleg Sychev

Volgograd State Technical University

Volgograd, Russia

oasychev@gmail.com

ABSTRACT

One of the important scaffolding tasks in programming learning is writing a line of code performing the necessary action. This allows students to practice skills in a playground with instant feedback before writing more complex programs and increases their proficiency when solving programming problems. However, answers in the form of program code have high variability. Among the possible approaches to grading and providing feedback, we chose template matching. This paper reports the results of using regular-expression-based questions with string completion hints in a CS1 course for 4 years with 497 students. The evaluation results show that Perl-compatible regular expressions provide good precision and recall (more than 99%) when used for questions requiring writing a single line of code while being able to provide string-completion feedback regardless of how wrong the initial student's answer is. After introducing formative quizzes with string-completion hints to the course, the number of questions that teachers and teaching assistants received about questions in the formative quizzes dropped considerably: most of the training question attempts resulted in finding the correct answer without help from the teaching staff. However, some of the students use formative quizzes just to learn correct answers without actually trying to answer the questions.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; • **Applied computing** → **Interactive learning environments**; • **Computing methodologies** → Discrete calculus algorithms.

KEYWORDS

regular expressions, short-answer questions, feedback generation, online learning, introductory programming courses

ACM Reference Format:

Oleg Sychev. 2022. Write a Line: Tests with Answer Templates and String Completion Hints for Self-Learning in a CS1 Course. In *44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '22)*, May 21–29, 2022, Pittsburgh, PA, USA. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3510456.3514159>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE-SEET '22, May 21–29, 2022, Pittsburgh, PA, USA

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9225-9/22/05...\$15.00

<https://doi.org/10.1145/3510456.3514159>

1 INTRODUCTION

Introducing students to the programming domain is a complex process, requiring the development of a significant number of cognitive skills [27]. It spans all levels of revised Bloom's taxonomy of educational objectives [2] from "remember" to "create", having strict constraints of logical correctness on all the levels to get a correct solution. Different e-learning tools may be used to support students while mastering different kinds of learning tasks.

One of these tasks is writing simple statements or headers, learning both syntax and semantics of the programming language [9]. These exercises belong to the application level of Bloom's taxonomy because they require applying syntax rules in typical situations. They lay the foundation for higher-level tasks such as analysis of existing programs and synthesis of new programs. While some of the students can write programs directly after being introduced to a text-based programming language, many others need scaffolding to get used to the syntax before attempting more complex tasks. Automated "write a line of code" assessments give them a convenient playground to acquire and reinforce basic skills. Without good skills in writing single lines of code, students make a lot of mistakes in programming assessments when they are focused on higher-level planning of the algorithm.

E-assessment is a good solution for the exercises requiring writing a single line of code because grading the required amount of questions manually requires too much time. However, simple short-answer question software is poorly suited for this situation because the response in the form of a line of code has high variability; many programming-language features can increase the set of correct answers – from the ability to insert any number of white space characters almost in any place allowing them without changing the answer – to the problem of extraneous (but correctly opened and closed) parentheses in expressions. Without a way to define the set of correct answers using templates or other mechanisms, code-writing short-answer questions become restrictive, giving a lot of false-negative grades and frustrating students.

When code-writing questions are used for formative assessments, it is necessary to provide feedback to the students about their mistakes and the ways to correct them. Otherwise, the students become stuck, requiring intervention from the teacher to understand how to complete their current exercise. Ott [30] supposes using targeted questions and hints to guide students in the process of finding the answer to a problem. Automatically generated feedback allows using formative assessments as homework with a significantly higher amount of completed questions than while utilizing teacher-provided feedback.

Some authors argue for parsing and executing program code to assess it (e.g., [1]), a template-based approach can be useful

for smaller problems involving writing a few lines of code. When the student's task is to write a single line of code, executing it often requires a significant number of enclosing code that must be supplied by the teacher, and testing errors often correspond poorly with the original task.

It is possible to parse students' code and compare resulting syntax trees, but this method also has significant disadvantages. First, there is no single starting symbol of the grammar for parsing a single line of code - the starting symbol depends on the particular question. Sometimes, a line of code cannot be reduced to a single symbol at all. Also, parsing does little to account for variation in the programming language (for example, in the C++ language, there are 6 ways to add 1 to a variable) so the teacher still has to provide all correct solutions. This is especially problematic if the answer contains several parts that can be written differently, as the number of correct answers rises multiplicatively. Template-based systems can solve this problem by letting to describe the variants of each part separately, significantly decreasing the effort required to create a question. Error-correcting parsers are also computationally expensive [31] while pattern matching can be implemented efficiently using, for example, finite automata.

The most important advantage of string templates over parsing is that they allow easy generation of completion hints. While there are works on generating data-driven feedback based on comparing syntax trees [29], they only work for fairly close syntax trees. However, sometimes a single wrong character can lead to a significantly different syntax tree. Also, as we often saw in teaching practice, some poorly-performing students who need hints most give answers that are very far from the expected line of code while error-correcting parsers are only good if there are a few errors. Errors of this kind happen more often in the first stages of learning programming (or when moving from a block-based programming language to a text-based language) where the questions requiring writing a single line of code are used. Novice programmers also rarely think about their programs in the terms that are expressed in the language grammars. So for short answers, comparing string representation of the program code may yield better hints than comparing syntax trees for the courses introducing students to the text-based programming language.

A commonly-used way to specify string-matching templates is regular expressions; they are expressive and relatively compact. Regular expressions were used in automated programming assessment to verify the output of the tested program [21]; they were also used for implementing short-answer questions [8]. However, most question-building tools do not allow combining advanced regular-expression syntax with string completion hints.

In this paper, we describe our experience of introducing formative quizzes with string completion hints into a CS1 course of Volgograd State Technical University, analyzing the results of four years of teaching. The quizzes were created using the Preg question type for Moodle LMS: a short-answer question software using Perl-compatible regular expressions to define correct answers and providing advanced feedback, showing partial matches (i.e., the correct beginning of the student's answer up to the first error) and providing string-completion hints (next correct character and next correct lexeme). The rest of the paper is organized as follows. Section 2 describes the state of the art in template-based assessments

and string completion hints; section 3 provides the background in regular expressions; section 4 describes the developed software; section 5 shows our findings while analyzing our experience followed by discussion in section 6 and conclusion in section 7.

2 RELATED WORK

2.1 Template-based short-answer assessment

Existing systems and approaches use different ways to specify patterns for short answers [7, 34]. Some of those systems calculate grades by performing a word-by-word comparison of weighted words, other systems use specific languages to describe patterns.

WebLAS identifies important segments of correct answers in parsed representations and asks the teacher to confirm each of them and assign weights to them [3]. The teacher is also asked to accept or reject semantically similar alternatives. Each segment of the student's answer is detected using regular expressions. Each segment is graded separately, so it is possible to grade partially-correct answers.

eMax uses a similar approach - it requires the teacher to markup semantic elements [11]. Teachers can also accept or reject synonyms and assign weights to each answer element. The grading approach is combinatorial, i.e. all possible formulations are pattern-matched. The assigned scores can be forwarded for manual review in difficult cases.

FreeText Author requires teachers to describe answers with syntactic-semantic templates for the student answers to be matched against [22]. Such templates are automatically generated from the plain-text representation of the teachers' answers. Through the interface, the teacher can specify mandatory keywords from the correct answers and select synonyms provided by thesauri support. Both acceptable and unacceptable answers can be defined.

IndusMarker uses word- and phrase-level pattern matching to grade student answers [32]. Credit-worthy phrases are defined using an XML-based markup language called the Question Answer Markup Language. Using the "structure editor", the text and number of points can be specified for each phrase.

OpenMark uses its own template language [8]. It can be used for specifying alternative words (e.g., synonyms) or word groups. The system also supports typo detection.

Most of these systems are aimed specifically at grading natural-language answers and are poorly suited for programming-language answers.

2.2 String-completion hints

Providing detailed feedback is an important requirement for short-answer grading systems. When a student makes a mistake, it is not enough to simply tell them that the answer is wrong [5, 18, 20]. They still have to either read their lecture materials to identify their mistakes or wait for the opportunity to consult with their teacher. Falkner et al. [15] suggested that increasing feedback granularity impacts performance positively. Automatic feedback, including the information about possible mistakes and the ways to fix them, significantly increases the efficiency of student efforts, allowing them to understand and correct their mistakes during an online session at a convenient time for them.

There are several well-known automatic feedback generation techniques [23].

- Intelligent Tutoring System model-tracing feedback generation technique that uses comparison of student's steps with buggy production rules [10, 17].
- Constraint-based feedback generation technique that checks student's answer against predefined constraints and generates feedback messages for failed ones [26].
- Natural Language Processing and Machine Learning feedback generation technique that provides feedback using datasets of students' answers [12, 25, 28].

3 BACKGROUND

Regular expressions are a standard method of specifying string patterns [16]. Applied to teaching programming, one of their undoubted advantages is their versatility regarding answer languages. They allow creating complex patterns that cover wide ranges of possible correct answers. Many tutorials and software tools for creating and debugging regular expressions are available [4, 6], but let us explain the very basics for the reader to be able to continue reading on.

Regular expressions, just like any other expressions, consist of operators and operands. The simplest operand is a Latin character: it matches itself. There are also character sets written like `[0-9a-fA-F]` (matches hexadecimal digits), meta-characters (a dot matches any character except line breaks), escape-sequences (`\s` matches whitespaces and `\d` matches decimal digits), and so on.

Operands can be *concatenated* to match a sequence of characters: `\d\s\d` matches any two decimal digits with whitespace between them. They also can be *quantified* to match a sequence of repeated strings: `(\d\s)+` matches a digit followed by a whitespace, repeated any number of times (but at least once). The `*` matches zero or more repetitions of its operand, and the `?` matches 0 or 1 repetitions. Finally, operands can be *alternated*: `ab|cd+` matches either "ab" or "cd", "cdd", "cddd" and so on. Note that concatenation takes precedence over alternation, and quantification takes precedence over concatenation.

Perl-compatible regular expressions (PCRE) have the richest syntax and most powerful features [19]. In addition to the above-mentioned features, they support such powerful features as named subpatterns, backreferences, recursive subpattern calls, look-around assertions, lazy and possessive quantifiers, extended syntax for character classes, and others. In the following sections, regular expressions will mean Perl-compatible regular expressions.

4 PREG QUESTION'S FEATURES AND THEIR USES IN INTRODUCTORY PROGRAMMING COURSES

4.1 Regular Expressions as Templates for Program Strings

4.1.1 Basic Regex Features. The first problem to solve when checking a student's program string is accepting any correct whitespace placement: most of the programming languages allow any number of whitespace characters between any lexemes and require at least

one whitespace between some. We can use repetition of `\s` character class to specify this e.g., `int\s+([_a-zA-Z]\w*)\s*`;

Another useful regular expression feature is alternation. If a student was asked to write a floating-point variable declaration, both "float" and "double" types should be accepted, so our previous example would turn into this: `(float|double)\s+[_a-zA-Z]\w*\s*`. While this can be solved without regular expressions by providing a set of correct answers, their number increases multiplicatively when the answer contains several sections with independent alternatives and quickly becomes impractical.

4.1.2 Advanced Regex Features. One more useful feature of regular expressions is back-referencing. A back-reference works in pair with a subpattern (a parenthesized part of the regular expression) and matches exactly the same string as its corresponding subpattern does. Assume that a student needs to write a "for" loop with an integer counter running from 0 to 9 and an empty body in the C or C++ languages. The counter's name should appear in the answer 3 times unmodified, and this can be achieved by enclosing the variable name in parentheses (thus creating a subpattern) and then referencing it two times later in the expression like this: `for\(\int ([_a-zA-Z]\w*)=0;\1(<=9|<10);\1\++\)\{;` where `\1` is a backreference to the first subpattern (i.e., the expression inside the first set of unescaped parentheses counting opening parentheses from the left). Correct whitespace matching is ignored here for the purpose of readability. This pattern matches student answers like "for (int i=0;i<=9;i++);" and "for (int myVar=0;myVar<10;myVar++){}".

One of the biggest challenges with grading students' answers containing formulas and program code using string templates is matching an arbitrary but correct placement of parentheses around expressions, i.e., "5" and "(((5)))" should be graded the same. Regular expressions using recursive subpattern calls (an exclusive feature of Perl-compatible regular expressions) can solve this problem, though the expressions become a lot harder to understand. Here is a regular expression matching the parenthesized digit 5: `5|(\s*\s*(?: (?-1) |5)\s*\s*)`.

It is clear that recursive regular expressions cannot be used for real-life questions "as is", because usually there are several subexpressions that need to be parenthesised so the whole regular expression becomes less and less readable. To solve this problem, we introduced a set of special regex comments in Preg which look like the following:

- `(?###parens_opt<)body(?###>);`
- `(?###parens_req<)body(?###>).`

These expressions define patterns match everything that their "body" matches, but is enclosed in zero or more(`parens_opt`) or one or more (`parens_req`) parentheses. There is also a number of other predefined patterns for matching brackets, custom parentheses (with any opening/closing symbols), identifiers, etc. With this feature, strings like "5" and "(((5)))" can be matched by a simple and readable regular expression: `(?###parens_opt<)5(?###>).`

In Figure 1 a regular expression from an actual code writing question is shown. The expression is entered in a multi-line text field and is still readable because of using the comment-based patterns. Below is a testing tool for debugging the expression, showing the

range of possible correct answers these expressions match. The tool highlights the matched and unmatched parts of the answers.

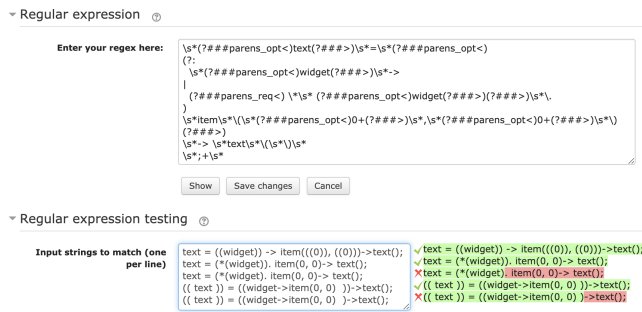


Figure 1: The tool for editing and debugging regular expressions in Preg questions

4.1.3 Partial Matching. Unlike the commonly-used regex libraries, the Preg question’s engine supports finding partial matches, i.e., matches starting at the beginning of the regex and breaking off somewhere in the middle. With this feature, on the higher level Preg highlights correct and incorrect parts of students’ answers in green and red, respectively.

4.2 String Completion Hints

The Preg question type is capable of generating completions of partial matches, leading to full matches. The completion is always chosen so that they will lead the student to the correct answer adding the minimum number of characters. Based on this, Preg can show two hints: the next correct character and the next correct lexeme. Hints may be turned off (for summative quizzes); a penalty score can be specified for each hint usage to discourage students from overusing the hints.

Figure 2 demonstrates how the next correct lexeme hint looks in Preg questions. Here, a student asks to complete the name of the method to call; the hinted part is highlighted yellow. The ellipsis after it shows the student that the answer is not complete after adding it.

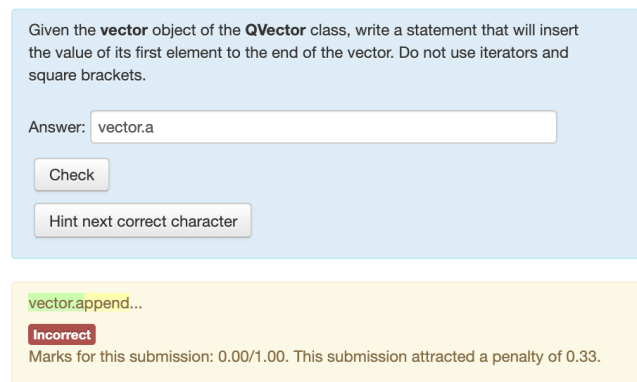


Figure 2: Next correct lexeme hint in a Preg question

5 CASE STUDY

Volgograd State Technical University uses Preg questions in the two-semester CS1 course “Programming basics” for the first- and second-year undergraduate students. The following data was gathered from the first semester of this course (first year, spring semester) unless explicitly state otherwise because the first semester had more academic hours and so more quizzes; the first semester of this course is also focused introducing students to the syntax of the C++ programming language that is better suited for the questions requiring writing a line of code.

5.1 Course Quiz Setup

The first semester of the CS1 course “Programming basics” contains 6 major programming assignments for topics ranging from alternative statements to user-defined data types. Each of them has an associated summative quiz (10 questions, about 15 variants each) that the student must pass in class before attempting the assignment. These quizzes help to make sure that the student knows the topic enough to attempt writing a program. Hints are disabled in summative quizzes, and students cannot re-attempt summative questions once they are graded. The questions in summative quizzes include code-writing Preg questions (from 1 to 5 questions per quiz), multiple-choice questions “find the code lines containing errors”, and short-answer questions “determine the results of code execution”.

Originally, to prepare students for the summative quizzes, for each summative quiz there was created a regular formative quiz, offering a few variants of the difficult questions, and a demonstration quiz with one variant of each question of the summative quiz. All the questions in these formative quizzes were supplied with teacher-defined natural-language feedback, explaining which answer is correct and why. The students actively used the formative quizzes, however, the natural language explanatory feedback was not enough: each lab assignment started from answering students’ questions about the formative quizzes they were stuck at which reduced the time for actual programming. The main problem for the code-writing questions was that the basic Moodle short-answer questions just reported the fact of error; the students had to choose between trying to fix their error by guessing and learning the correct answer.

To enhance the students’ abilities to train during homework, we developed Preg formative quizzes for each code-writing question of each summative quiz with string-completion hints enabled. Students can use formative quizzes as much as they want during homework but have no obligation to attempt them. Each formative quiz usually has about 5 variants of each question. While attempting a formative quiz, the student can correct their answer and re-grade it as many times as they want. They also can use the string-completion hints that the Preg question type provides.

The summative quizzes are graded, but their influence on the course grade is minimal (less than 10% of the course grade). Their main function is being the threshold to cross before attempting more complex tasks. This was done to limit the grade loss caused by attention slips during testing as the students cannot debug their code while attempting summative quizzes. The formative quizzes are provided for training purposes and do not affect the course

Table 1: Quantitative analysis of the regular expressions used in “Programming Basics” course

Parameter	Min	Max	Mean	Stdev
characters in answer	2	109	31.08	19.8
tokens in answer	1	70	14.22	8.75
paths through the expression	1	18	1.58	1.86

grade. All quizzes showed the correct answers and natural-language feedback once the quiz is completed.

The students who fail a summative quiz (by scoring less than 60%) are given time to improve their skills and should re-attempt the quiz until passing it (up to 5 attempts). They can use formative quizzes while preparing for the second and later attempts. As a part of the final exam, students pass the exam quiz containing 18 questions from all 6 summative quizzes, providing a broad overview of their knowledge of the whole course.

After introducing Preg formative quizzes, the main students’ complaint about them was its lack of handling of extra parentheses in expressions which produced false-negative grades. To resolve this problem, we enhanced the existing regular expressions with recursive subpattern calls using the special Preg syntax described in the section 4.1.2. It resolved the problem, significantly lessening the number of false-negative grades.

The results presented below are based on the data collected over four years (2016, 2017, 2019, and 2020) of teaching an introductory programming course in Volgograd State Technical University to the first-year students specializing in “Informatics and Computing” and “Software Engineering” with 479 students in total.

We gathered all the data about the question attempts, including their sequence, students’ responses to each question, and the hints they requested. We also conducted informal interviews with the staff teaching the course.

5.2 Answer templates

The course contains 722 Preg questions using 1160 regular expressions in total. One Preg question contained from 1 to 13 regular expressions (mean 1.5 regular expressions per question) used to test various correct answers. These regular expressions described program codes of varying lengths that can be measured in characters and tokens. In Table 1 we provided measures for the shortest correct answer and the length of the longest correct answers for most of the regular expressions is (theoretically) not limited because of quantifiers “*” and “+” allowing any number of whitespace characters. Another important measure of the regular expression complexity is the number of possible alternative paths through the expressions, showing the variability of code described by them.

The students gave 4952 unique answers that were matched by these regular expressions between 1 to 67 unique answers per regular expression, 4.9 unique answers per expression on average.

Some of these unique answers differed only in whitespace characters. Considering all answers that differ only in whitespaces the same, we found 2107 unique answers, between 1 to 40 answers per regular expression, 2 answers per expression on average. The distribution of the number of unique answers per regular expression

Table 2: Attempts of formative quizzes

Tried to answer questions	Answered correctly	Number of attempts	Percentage
No	No	87	2.8%
Some	No	76	2.5%
Some	Some	239	7.7%
All	Some	738	23.8%
All	All	1960	63.2%

is shown in Fig. 3: the number of unique answers on the X-axis, and the number of regular expressions with this number of unique answers on the Y-axis. The majority of regular expressions had from 1 to 5 unique answers; the number of expressions matching more than 15 unique answers is small. It shows that using regular expressions allowed us to cover a wider range of correct answers than it would be possible using simple open-answer questions with string matching.

As we said before, the main cause of false grades for regular expressions questions were extraneous parentheses. The usage of recursive subpattern calls to catch extraneous parentheses increased recall for the Preg question in the entire course from 0.88 to 0.99 and F-measure from 0.937 to 0.995. The few remaining cases of false-negative grading are solved by teachers individually; sometimes regular expressions were upgraded to match a new solution found by a student.

5.3 Formative Quizzes and Hints

When evaluating Preg formative quizzes’ effect on the students’ performance, it must be taken into account that Preg quizzes were working with only one kind of question - open-answer string questions - which comprised roughly one-third of the summative quizzes. About 43% of the students chose not to use Preg quizzes at all. To understand the next two tables better, it is good to keep in mind the difference between the number of quiz attempts and the number of question attempts: each formative quiz contains from 1 to 5 questions depending on the topic.

Out of 464 students who attempted at least one quiz in the course, 207 (44.6%) chose to not use Preg formative quizzes at all. The remaining 257 students used Preg quizzes for training from 1 to 107 times (mean 9.16, standard deviation 13.98). They made 3100 quiz attempts. Table 2 summarizes the ways the students used Preg formative quizzes. Given that they could improve their answers until answering correctly or giving up, we can consider correct answers as the student being able to complete their attempt (or some questions in it).

It can be seen that more than half of the training attempts ended in answering all questions correctly. In more than three-quarters of attempts, the student answered correctly at least some questions. Only in 5.3% of attempts the student gave up without answering at least one question correctly. Half of them simply used formative quizzes to learn the correct answers, clicking the button to complete the quiz right after it started.

Looking at how the individual questions were attempted, we found 8291 question attempts. The summary is shown in Table 3.

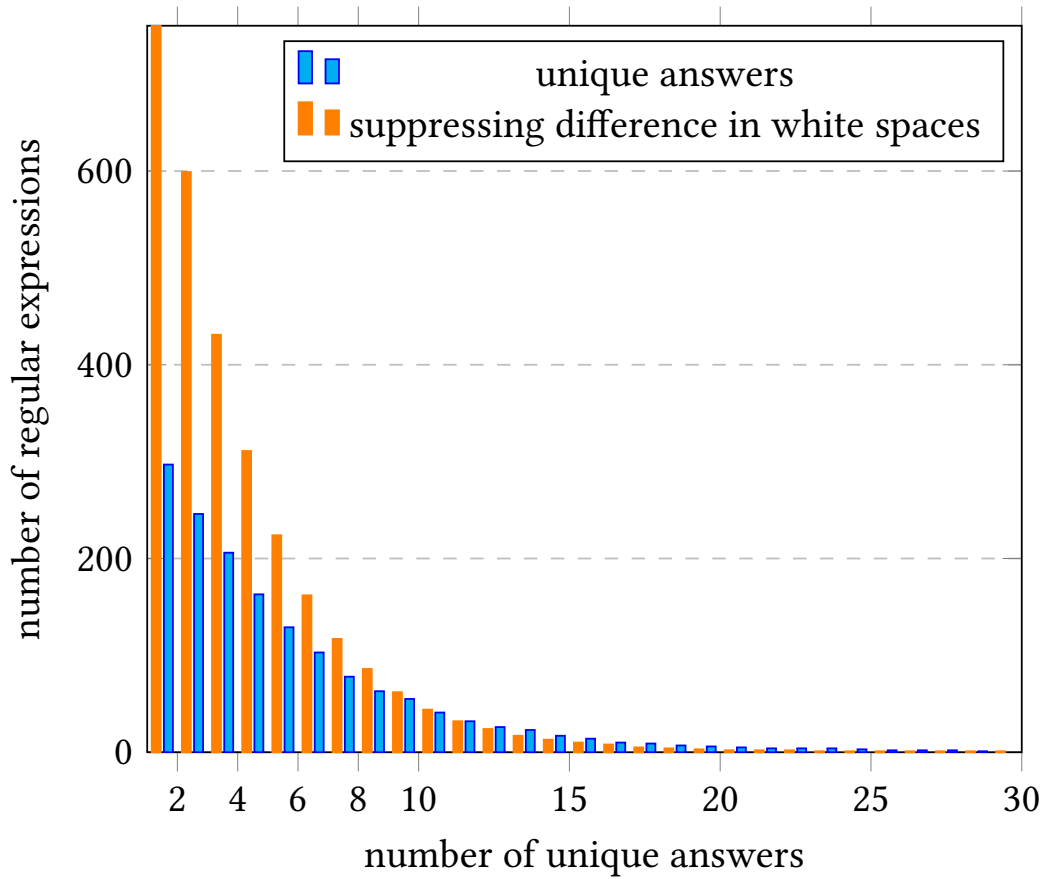


Figure 3: Distribution of regular expressions by the number of unique covered answers

Table 3: Attempts of formative questions

Final answer	Number of attempts	Percentage	Attempts with hints
Absent	581	7%	0
Incorrect	1126	13.6%	304
Correct	6584	79.4%	2000

Note that for individual questions, the number of attempts without even trying to answer should be interpreted differently: it is a valid strategy to answer only some questions during a quiz attempt meant for training if the student is sure in their ability to answer the other questions. The students were able to find correct answers to almost 80% of the questions without support from the teaching staff. About one-third of the completed attempts used hints. In 13.6% of attempts, the student gave up solving a question. In 27% of these attempts, the student used hints but still did not find the correct answer. The percentage of not completed attempts is almost the same for the attempts with hints (13%) and the attempts that did not use hints (15%).

To assess the effect of using Preg quizzes on the overall performance of the students, we studied average exam quiz scores for different groups of students. The exam quiz was not obligatory, and some students were not allowed to take exams because of not completing enough assignments during the semester. We found 272 attempts of the exam quiz. Its results depending on the number of attempts of Preg formative quizzes are shown in Table 4, while depending on the frequency of hint usage are shown in Table 5. Paired, 2-tailed t-test does not show any significant difference in the exam quiz scores according to the number of attempts of Preg formative quizzes. However, the students who used hints sparingly performed significantly better than those who did not use hints at all ($p=0.02$) and those who used hints often ($p=0.05$). It is clear that no significant influence of Preg training on the exam quiz can be found. This may be unsurprising, given that the students had other means of training for the exam quiz and other means to get help. When we consider hint usage, the small group of students who used hints sparingly showed marginally better performance than the students who did not use hints or used them often. However, the small size of this group and the weak effect size require further research to validate this finding. The interviews with the teaching staff indicated that the main advantage of introducing Preg formative quizzes was significantly reducing the number of

Table 4: The exam quiz score by the activity in using formative quizzes during the semester

Student Group	Students	Mean	Stdev
no Preg training	99	0.73	0.16
low Preg training (≤ 7 attempts)	114	0.729	0.16
active Preg training (> 7 attempts)	59	0.76	0.16

Table 5: The exam quiz score by the percent of hints used during the semester

Student Group	Students	Mean	Stdev
did not use hints	142	0.72	0.15
low hint usage ($< 33\%$ of attempts used hints)	30	0.78	0.13
active hint usage ($> 33\%$ of attempts used hints)	100	0.73	0.16

questions about homework formative quizzes they had to answer as the students, with the help of string-completion hints, could resolve many problems on their own.

6 DISCUSSION

While automatic feedback generation for programming exercises was researched extensively (see [24]), the effects of smaller-size tasks like writing a single line of code did not receive the same attention. The reports on the effects of voluntary programming practice in introductory programming courses differ. Spacco et al [33] reported a weak correlation between exam performance and the number of voluntarily completed programming exercises. Estey et al [14] reported a negative correlation between the number of requested hints and exam performance while no relationship between practice time and exam performance. Edwards et al [13] divided students into three groups (no practice, some practice, and full practice) and found that students from the “some practice” groups performed worst in both code-writing exercises and multiple-choice questions in final exams. However, all this research regarded voluntary solving program-writing exercises.

We found that regular expressions are suited well for programming questions with short answers (about one line of code; from 1 to 70 tokens) which are used in introductory programming courses during quizzing. Regular expressions allowed teachers to create questions with up to 18 significantly different answers (i.e., the answers that are different in more than just extra whitespaces and parentheses) per expression which would be time-consuming if using string matching or parse trees matching. However, most of the developed regular expressions had 1 or 2 significantly different answers.

Most of the teachers participating in developing Preg questions found creating a regular expression from a program string simple and straightforward: the best and common practice was writing the program code as it is first, then escaping special characters, then introducing alternative parts, followed by adding the notation for extra parentheses and optional whitespaces. While this process can

be automated, developing a special tool for creating this question bank was found unnecessary. The regular-expression editing, visualizing, and testing interface of Preg questions were adequate to the task of developing expressions for a string of code.

While normal regular expressions often (about 11% of times) produced false-negative grades because of extraneous parentheses in expressions, Perl-compatible regular expressions can handle these using the recursive subpattern calls feature. Practical usage of recursive subpattern calls requires special syntax for typical expressions to keep the resulting expressions human-readable; we developed several patterns using special regular-expressions comments to achieve that. After this, the amount of false-negative grades dropped to 0.8% which can be handled by the teaching staff even in large courses. False-positive grades were extremely rare as a regular expression allows defining a set of correct answers precisely. One regular expression matched 4.9 unique students’ answers on average, up to 67 unique answers matched by one expression.

Analyzing attempts of the formative quizzes for training purposes, we found that when using Preg questions, the students solved at least one question correctly in almost all the attempts and solved all questions correctly in almost two-thirds of the attempts. This means that most of the students used the formative quizzes actively, answering the questions repeatedly until solution if necessary. Only about 5% of attempts were given up without answering correctly a single question. The relatively big number of attempts where not all questions were answered correctly may be not a big issue because sometimes students omitted the questions they knew well, concentrating on the difficult questions while training. A significant number of the students’ first (starting) answers contained major syntax errors (up to omitting or misplacing more than half of the answer’s tokens) that would prevent a question based on error-correcting grammar from providing meaningful feedback, while Preg question could provide enough support to solve the task.

Studying question attempts, we can see that almost 80% of them ended in a correct answer. Students used hints in about each third attempt to answer a question; the relatively low frequency of using hints can partially be attributed to the easiness of some questions. The teachers during interviewing also noted that sometimes very poorly performing students were not even aware that they can use hints. These were mostly the students who either skipped classes often or learned in the second language and had trouble understanding what was going on in the class. This last group, according to the teachers, gained a lot from using hints once they were shown how to use hints personally.

However, there is no significant difference between the percentage of attempts using hints among correctly answered and incorrectly answered attempts. In more than 13% of cases, students gave up answering even having access to the hints that led to the correct answer if used enough times. In 73% of the attempts when students gave up, they did not even try to use hints. This may mean that the teaching staff should spend more time teaching the students how to use hints. Still, the teachers reported a sharp decrease in the number of students’ questions about formative quizzes once the hints were enabled. About 500 training questions per semester were answered correctly using hints; without hints, the students would need to ask their teacher or teaching assistant about solving

the problematic question. This saves the teaching time to spend on actual programming.

We did not find any link between using Preg questions for training and the students' performance in the exam quiz. However, the students had two other kinds of formative quizzes to train (which the teachers found unethical to turn off) and Preg questions accounted only for open-answer questions which comprised about a third of the exam quiz. There seems to be a weak link between using hints sparingly (less than once in every 3 attempts) and performing better in the final quiz. This means that overuse of completion hints is mostly done by poorly performing students; saying simply, repeated clicks on "show me what to type next" does not stimulate learning. So Preg questions can be improved by adding a limiter to the number of hints the student can use. The high frequency of hint usage can serve as a warning sign about learning problems; students who do it require special attention from teaching assistants. The teachers can discourage students from using hints too often by applying grade penalties for their use as the Preg software allows.

The main positive effect of introducing regular-expression questions with string-completion hints was saving teachers' class time that was previously spent on answering students' questions about their homework quizzes. Some teachers noted that foreign students who learned in their second language and so had trouble understanding teachers gained from using Preg questions most as the string-completion hints do not require reading and understanding natural-language texts. It increased the course's inclusivity and saved a lot of teachers' time because explaining problems to these students often take a lot of time. The immediate automatic hints in code-writing quizzes worked well because the tasks are relatively simple and do not require prolonged thought. It was also very valuable during COVID lockdowns.

7 LIMITATIONS AND CONCLUSION

We found that regular expressions matching is a viable way to effectively implement short-answer questions requiring writing a single line of code (up to about 70 tokens). A single regular expression can cover up to 18 different answers. Using regular expression to grade program code requires solving the problem of extraneous parentheses in expressions that is possible using Perl-compatible regular expressions. We used the Preg question-type plug-in for Moodle LMS as a tool implementing the necessary features and able to give hints on completing partially correct students' answers. Formative quizzes with string-completion hints allow students to practice writing lines of program code performing a particular task on their own without being stuck. However, too frequent usage of hints may hinder students' progress and should be discouraged or used as a marker of poorly-performing students needing more attention. This requires further research. The main effect of training with hints seems to be saving teachers' time rather than learning gains in answering the same questions on exams: the students learned the same without consulting their teacher and teaching assistants.

These open-answer questions work only on the application level of Bloom's taxonomy of learning objectives and should be supported by other learning tools and techniques. For the assignments requiring writing more code, it is better to switch from matching

strings to testing the developed code. Summative quizzes, however, stimulate mastering low-level skills before attempting high-level tasks (like solving coding assignments) and ensure a certain level of knowledge before the student attempts complex tasks requiring more attention from the teaching staff.

Further work on the Preg question type will include adding typo detection features: the most common students' complaint was about a typo causing the whole answer to be graded as wrong. Adding more features of Perl-compatible regular expressions (like complex assertions) to the hinting engine will improve its templating ability. The Preg question type is available for the widely-used Moodle LMS under General Public License and can be downloaded from the link is anonymised.

ACKNOWLEDGMENTS

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

REFERENCES

- [1] Kirsti Ala-Mutka. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education* 15 (2005), 102 – 83. <https://doi.org/10.1080/08993400500150747>
- [2] Lorin W. Anderson and David R. Krathwohl (Eds.). 2001. *A Taxonomy for Learning, Teaching, and Assessing. A Revision of Bloom's Taxonomy of Educational Objectives* (2 ed.). Allyn & Bacon, New York.
- [3] Lyle F Bachman, Nathan Carr, Greg Kamei, Mikyung Kim, Michael J Pan, Chris Salvador, and Yasuyo Sawaki. 2002. A Reliable Approach to Automatic Assessment of Short Answer Free Responses. In *COLING '02: Proceedings of the 19th international conference on Computational linguistics* (Taipei, Taiwan) (COLING '02). Association for Computational Linguistics, USA, 1–4. <https://doi.org/10.3115/1071884.1071907>
- [4] Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. 2014. RegViz: Visual Debugging of Regular Expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) (ICSE Companion 2014). Association for Computing Machinery, New York, NY, USA, 504–507. <https://doi.org/10.1145/2591062.2591111>
- [5] Paul Black and Dylan Wiliam. 1998. Assessment and Classroom Learning. *Assessment in Education: Principles, Policy & Practice* 5, 1 (1998), 7–74. <https://doi.org/10.1080/0969595980050102>
- [6] I. Budiselic, S. Sribljic, and M. Popovic. 2007. RegExpert: A Tool for Visualization of Regular Expressions. In *EUROCON 2007 - The International Conference on "Computer as a Tool"*. IEEE, Warsaw, Poland, 2387–2389. <https://doi.org/10.1109/EURCON.2007.4400374>
- [7] Steven Burrows, Iryna Gurevych, and Benno Stein. 2015. The Eras and Trends of Automatic Short Answer Grading. *International Journal of Artificial Intelligence in Education* 25, 1 (01 Mar 2015), 60–117. <https://doi.org/10.1007/s40593-014-0026-8>
- [8] Philip G Butcher and Sally E Jordan. 2010. A comparison of human and computer marking of short free-text student responses. *Computers & Education* 55, 2 (2010), 489–499. <https://doi.org/10.1016/j.compedu.2010.02.012>
- [9] Aparna Chirumamilla and Guttorm Sindre. 2019. E-Assessment in Programming Courses: Towards a Digital Ecosystem Supporting Diverse Needs?. In *Digital Transformation for a Sustainable Society in the 21st Century*, Ilias O. Pappas, Patrick Mikalef, Yogesh K. Dwivedi, Letizia Jaccheri, John Krogstie, and Matti Mäntymäki (Eds.). Springer International Publishing, Cham, 585–596. https://doi.org/10.1007/978-3-030-29374-1_47
- [10] Albert T. Corbett and John R. Anderson. 2001. Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) (CHI '01). Association for Computing Machinery, New York, NY, USA, 245–252. <https://doi.org/10.1145/365024.365111>
- [11] Dezso Sima, Balazs Schmuck, Sandor Szollosi, and Arpad Miklos. 2007. Intelligent short text assessment in eMax. In *AFRICON 2007*. IEEE, Windhoek, 1–7. <https://doi.org/10.1109/AFRCON.2007.4401593>
- [12] Myroslava Dzikovska, Natalie Steinhauser, Elaine Farrow, Johanna Moore, and Gwendolyn Campbell. 2014. BEETLE II: Deep Natural Language Understanding and Automatic Feedback Generation for Intelligent Tutoring in Basic Electricity and Electronics. *International Journal of Artificial Intelligence in Education* 24, 3 (01 Sep 2014), 284–332. <https://doi.org/10.1007/s40593-014-0017-9>
- [13] Stephen H. Edwards, Krishnan P. Murali, and Ayaan M. Kazerouni. 2019. The Relationship Between Voluntary Practice of Short Programming Exercises and

- Exam Performance. In *Proceedings of the ACM Conference on Global Computing Education* (Chengdu, Sichuan, China) (*CompEd '19*). Association for Computing Machinery, New York, NY, USA, 113–119. <https://doi.org/10.1145/3300115.3309525>
- [14] Anthony Estey and Yvonne Coady. 2017. Study Habits, Exam Performance, and Confidence: How Do Workflow Practices and Self-Efficacy Ratings Align?. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) (*ITiCSE '17*). Association for Computing Machinery, New York, NY, USA, 158–163. <https://doi.org/10.1145/3059009.3059056>
- [15] Nickolas Falkner, Rebecca Vivian, David Piper, and Katrina Falkner. 2014. Increasing the Effectiveness of Automated Assessment by Increasing Marking Granularity and Feedback Units. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) (*SIGCSE '14*). Association for Computing Machinery, New York, NY, USA, 9–14. <https://doi.org/10.1145/2538862.2538896>
- [16] Jeffrey E. F. Friedl. 2006. *Mastering Regular Expressions* (3 ed.). O'Reilly, Beijing.
- [17] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. 2012. An Interactive Functional Programming Tutor. In *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/2325296.2325356>
- [18] Graham Gibbs and Claire Simpson. 2005. Conditions under which assessment supports students' learning. *Learning and teaching in higher education* 1 (2005), 3–31. <http://eprints.glos.ac.uk/3609/>
- [19] Jan Goyvaerts and Steven Levithan. 2012. *Regular expressions cookbook*. O'reilly, USA, Sebastopol.
- [20] John Hattie and Helen Timperley. 2007. The Power of Feedback. *Review of Educational Research* 77 (03 2007), 81–112. <https://doi.org/10.3102/003465430298487>
- [21] Colin A. Higgins, Geoffrey Gray, Pavlos Symeonidis, and Athanasios Tsintsifas. 2005. Automated Assessment and Experiences of Teaching Programming. *J. Educ. Resour. Comput.* 5, 3 (Sept. 2005), 5–es. <https://doi.org/10.1145/1163405.1163410>
- [22] Sally Jordan and Tom Mitchell. 2009. e-Assessment for learning? The potential of short-answer free-text questions with tailored feedback. *British Journal of Educational Technology* 40, 2 (2009), 371–385. <https://doi.org/10.1111/j.1467-8535.2008.00928.x>
- [23] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. *Towards a Systematic Review of Automated Feedback Generation for Programming Exercises*. Association for Computing Machinery, New York, NY, USA, 41–46. <https://doi.org/10.1145/2899415.2899422>
- [24] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* 19, 1, Article 3 (Sept. 2018), 43 pages. <https://doi.org/10.1145/3231711>
- [25] H. Lane and Kurt Vanlehn. 2005. Teaching the tacit knowledge of programming to novices with natural language tutoring. *Computer Science Education* 15 (09 2005), 183–201. <https://doi.org/10.1080/08993400500224286>
- [26] Nguyen-Thinh Le and Wolfgang Menzel. 2006. Problem Solving Process Oriented Diagnosis in Logic Programming. In *Proceedings of the 2006 Conference on Learning by Effective Utilization of Technologies: Facilitating Intercultural Understanding*. IOS Press, NLD, 63–70. <https://doi.org/10.5555/1565941.1565955>
- [27] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) (*ITiCSE 2018 Companion*). Association for Computing Machinery, New York, NY, USA, 55–106. <https://doi.org/10.1145/3293881.3295779>
- [28] Cara Macnish. 2010. Java Facilities for Automating Analysis, Feedback and Assessment of Laboratory Work. *Computer Science Education* August 2000 (08 2010), 147–163. [https://doi.org/10.1076/0899-3408\(200008\)10:2;1-C;FT147](https://doi.org/10.1076/0899-3408(200008)10:2;1-C;FT147)
- [29] Victor J. Marin, Maheen Riaz Contractor, and Carlos R. Rivero. 2021. Flexible Program Alignment to Deliver Data-Driven Feedback to Novice Programmers. In *Intelligent Tutoring Systems, Alexandra I. Cristea and Christos Troussas (Eds.)*. Springer International Publishing, Cham, 247–258. https://doi.org/10.1007/978-3-030-80421-3_27
- [30] Claudia Ott, Anthony Robins, and Kerry Shephard. 2016. Translating Principles of Effective Feedback for Students into the CS1 Context. *ACM Trans. Comput. Educ.* 16, 1, Article 1 (Jan. 2016), 27 pages. <https://doi.org/10.1145/2737596>
- [31] Sanguthevar Rajasekaran and Marius Nicolae. 2014. An error correcting parser for context free grammars that takes less than cubic time. arXiv:1406.3405 [cs.DS]
- [32] R. Siddiqi and C. Harrison. 2008. A systematic approach to the automated marking of short-answer questions. In *2008 IEEE International Multitopic Conference*. IEEE, Karachi, 329–332. <https://doi.org/10.1109/INMIC.2008.4777758>
- [33] Jaime Spacco, Paul Denny, Brad Richards, David Babcock, David Hovemeyer, James Moscola, and Robert Duvall. 2015. Analyzing Student Work Patterns Using Programming Exercise Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) (*SIGCSE '15*). Association for Computing Machinery, New York, NY, USA, 18–23. <https://doi.org/10.1145/2676723.2677297>
- [34] Oleg Sychev, Anton Anikin, and Artem Prokudin. 2020. Automatic grading and hinting in open-ended text questions. *Cognitive Systems Research* 59 (2020), 264–272. <https://doi.org/10.1016/j.cogsys.2019.09.025>