Yang, J. C. et al. (Eds.) (2018). Proceedings of the 26th International Conference on Computers in Education. Philippines: Asia-Pacific Society for Computers in Education

Generation of Personalized Abstract and Real-World Programming Exercises

Thomas James TIAM-LEE & Kaoru SUMI
Future University Hakodate, Japan
g3117002@fun.ac.jp

Abstract: We present an approach for automatically generating abstract and real-world coding exercises with adjustable difficulty. Using our approach, we can produce exercises based on abstract computational operations which are useful for learning the syntax of a programming language, as well as exercises based on real-life contexts which are useful for learning abstraction and logic formulation. We present the details of our approach as well as an initial evaluation of the exercises by students and teachers of programming. This work can pave the way for the development of intelligent programming tutors with adaptive and personalized feedback that can display content based on the state of the student.

Keywords: Programming, education, content generation

1. Introduction and Related Studies

Computer programming is becoming a core competence in our society. Governments around the world, including the USA, EU and Japan are already making efforts to integrate computer science into their education system (Digital Promise, 2017; Balanskat and Engelhardt, 2015 & MEXT, 2016). With the growing popularity of programming, there has also been a surge in the number of tools and resources aimed at supporting people who are learning how to code. Online web tutorials like Codecademy (https://www.codecademy.com) and Treehouse (https://teamtreehouse.com), aim to bring programming to a wide audience. Visual programming languages like Scratch (Maloney et al., 2010) and App Inventor (Wolber, 2011) help teach programming concepts to children. Intelligent programming tutors like Ask-Elle (Gerdes, Heeren & Jeuring, 2017) and Java Sensei (Cabada et al., 2015) teach coding while providing students customized feedback based on their performance.

Personalized feedback is at the heart of many intelligent programming tutors. Previous studies have shown that personalized feedback can significantly increase student performance and satisfaction in learning (Gallien & Oomen-Early, 2008). Recently, education techniques such as flipped learning have also risen in popularity. In this approach, students acquire knowledge at home instead of in the classroom (FL Network, 2014). This underscores the need for teachers and curriculum designers to provide tools that can attend to specific needs of individual students. Previous intelligent programming tutors have sought to provide different types of adaptive and personalized feedback. Ask-Elle provides personalized hints based on a prediction of the student's intention in writing code. Java Sensei gives empathetic responses to the student's affective state to help drive motivation. In our previous paper, we developed an intelligent programming tutor that detects and responds to student confusion by giving hints and adjusting the difficulty of the problems (Tiam-Lee & Sumi, 2018).

However, there is little work done on the personalization of coding exercises. In the work of Wakatani & Maeda (2016), tracing and debugging exercises are generated with code templates in the PHP language. In the work of Prados et al (2005), variations of manually-written exercises are generated by using parameterized questions. In the work of Hsiao, Brusilovsky & Sosnovsky (2009), object-oriented tracing questions are generated from parameterized code. Although these works present approaches to automatically generate exercises, they either focus on non-coding skills in programming (tracing and debugging) or do not offer enough flexibility to generate coding exercises that vary in terms of difficulty, theme, or solution structure.

The type of coding exercise we are concerned with is one in which the student must write code to accomplish a specified task. For example, the student may be required to write code to get the average of two numbers. To simplify the complexities of handling user input and output, we use the format of completing a function, but in practice this approach can be applied to work with exercises that require input and output as well.

Previous works on exercise generation mostly deal with tracing and debugging exercises. In the computer science-specific learning taxonomy proposed by Fuller et al. (2007), these types of exercises only cover theoretical competency skills such as remembering, understanding, and analyzing. Students who wish to attain higher competencies must also learn how to apply and create concepts. These skills are covered by such exercises as writing code to solve problems. In this paper, we present an approach for generating personalized exercises of this type without using parameterized questions. We discuss the generation of two types of exercises that differ by their level of abstraction. First, we discuss exercises based on abstract programming computations, which test students' ability to translate computer operations into coding syntax. Then, we also discuss exercises based on real-world contexts, in which students must understand which operation to use to solve the problem.

2. Generation of Personalized Programming Exercises

In this section, we discuss an approach for automatically generating programming exercises. First, we discuss the representation of a programming exercises. Then, we discuss how we automatically generate exercises using this representation.

2.1 Exercise Definition and Representation

We represent an exercise as a set of nodes arranged in a flowchart-like structure. This flowchart represents the sequence of operations in a solution for the exercise. Like a typical programming flowchart, nodes can either be an operation node (rectangle) representing a single operation, condition node (diamond) representing a true or false condition that branches into two paths, or a return node (circle) representing the output of the exercise. Two examples of this are shown in Figure 1.

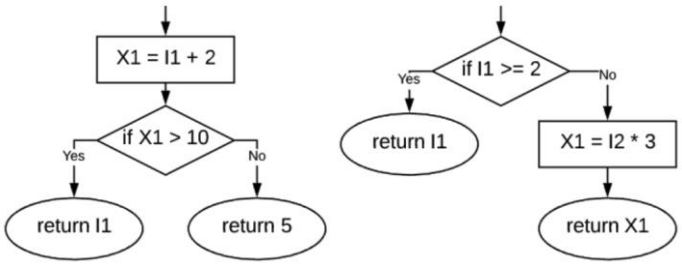


Figure 1. Examples of the exercise representation

2.2 Personalization

In this paper, we mainly focus on adjusting difficulty as a point for personalization in the generated exercises. For the purposes of this study, we define as the number of operations needed to solve the problem. In the examples above, the exercises have a difficulty of 2 (we do not count return nodes). Although the number of operations is not the only factor for an exercise's difficulty or learning value, this is the simplest form of metric that we start with for this purpose. Other points of personalization, such as the types of computations involved are also possible.

Using our approach, exercises with varying difficulty can be generated based on the student's ability or affective state. For example, a student who is having difficulty may be given easier exercises, while a student who is performing well may be given more difficult exercises. A student who is confused may be given exercises with similar operations. Sensing the student ability is not covered in this paper, as we focus on the generation aspect alone. In the succeeding sections,

we discuss our approach for generating exercises based on abstract computational operations and real-world computational operations in which the difficulty could be adjusted dynamically.

2.3 Generating Exercises Based on Abstract Computational Operations

An exercise can be procedurally generated as follows. First, the structure of the exercise is generated by the following algorithm. First, an operation or condition node is set as the head of the exercise. The head of the exercise is the first operation to be performed. Then, additional nodes are continuously added until the structure contains c nodes, where c is the difficulty of the exercise. Finally, all paths of the structure are terminated with a return node. The difficulty of the exercise could be controlled by changing the value of c.

Once the structure has been generated, the parameters for each node are assigned. We define the depth of a node N as the number of steps needed to reach node N from the head node. Next, we define C_{list} to be a list of critical nodes. We define a critical node as a node whose operation affects the return value of the function. For the exercise to be ideal, we want all nodes to be critical (otherwise, some operations will be irrelevant to the output).

Initially, we add all return nodes and condition nodes to C_{list}. Return nodes directly affect the output by their definition, and condition nodes affect which branch to take, affecting which return node will be reached. Next, we set a counter variable $i = 1$. We loop through the remaining operation nodes in descending depth. For each operation node N, we create a unique variable X_i, assign it as the left-hand side of the assignment in N, and then increment i. Then, we select a random node C from C_{list} that satisfies the following conditions: C has at least one operand that is not yet defined, and C is reachable from N. If no nodes satisfy the above conditions, the generation is treated as a failure. Otherwise, the variable X_i is assigned as one of the operands of C. Since C is a critical node and X_i now affects its operation, it follows that N is now a critical node as well, so we add it to C_{list}. Optionally, X_i may be assigned to more nodes from C_{list}. This process is repeated until all nodes are in C_{list}.

For each node that has no operands yet, we attach at least one variable I_1, I_2, I_3, ... to it as an operand. These variables represent input parameters of the exercise function. Finally, we fill all remaining operands slots that have not been assigned yet with constants. Figure 2 shows the process.

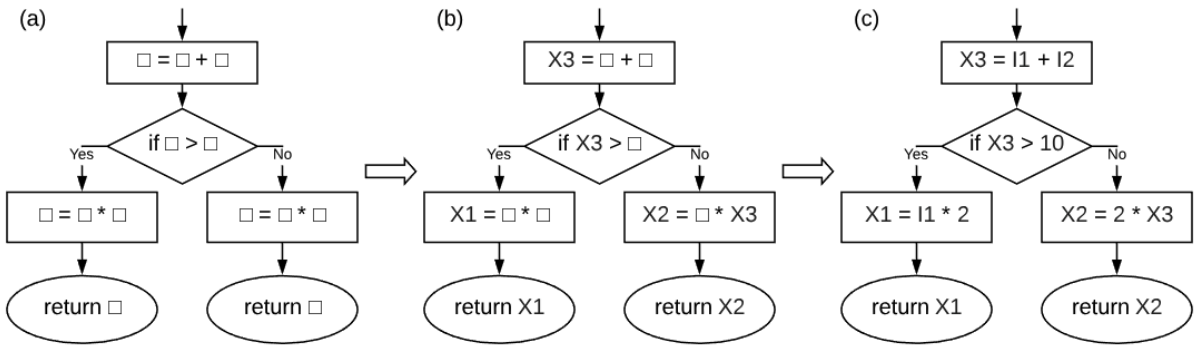


Figure 2. Exercise generation process. In (a), the exercise structure is generated with $c = 4$. In (b), all nodes are ensured to be relevant to the output. In (c), the remaining parameters are assigned

The exercise text can be generated from a mapping of each node configuration to natural language text. Table 1 shows some of these mappings. We use a recursive approach in generating the text, starting from the head node. To get the problem text for a node, we get the corresponding text for that node, replacing **{1}** and **{2}** with the operands and **{O}** with the left-hand side variable in the case of an operation node. We also replace **[A]** and **[B]** recursively.

Table 1

English Text for Selected Node Configurations

Node	English Text
Operation Node (+)	Get the total of **{1}** and **{2}**. Store the result in **{O}**. **[A]**
Operation Node (-)	Subtract **{2}** from **{1}**. Store the result in **{O}**. **[A]**
Condition Node (>)	If **{1}** is greater than **{2}**, **[A]**. Otherwise, **[B]**
Return Node	Return **{1}**.

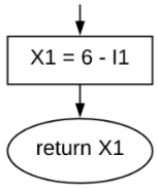
Further modifications are made to improve the clarity of the generated text. First, we remove "store the result in {O}" for operations wherein the result is used only in the immediately succeeding operation, and simply refer to the variable as "the result". Second, use separate labels for nested conditional statements to avoid ambiguity with the "if... otherwise" semantic. Table 2 shows some exercises generated using the above approach.

Table 2

Examples of Exercise Generated with Abstract Computational Operations with Difficulty = 1 and Difficulty = 2

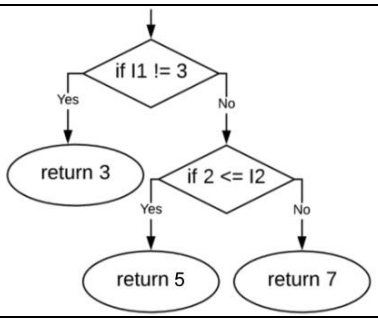
[Difficulty: 1] Complete the function. Perform 6 minus I1. Return the result.

```
int func(int I1) {

}
```



[Difficulty: 2] Complete the function. If I1 and -3 have different values, return 3. Otherwise do the instructions in A.
A: If 2 is less than or equal to I2, return 5. Otherwise return 7.

```
int func(int I1, int I2) {

}
```



2.4 Generating Exercises Based on Real-World Computations

In this section, we discuss how exercises based on real-world computations can be generated. To do this, we perform planning using two phases of backward state-space search on a domain commonsense knowledge base. The knowledge base contains a collection of assertions about the real world, as well as actions that can cause effects on the state of the world. Assertions are represented as predicate logic. Actions are divided into story actions and computational actions. Story actions are actions that provide context to the computation, while computational actions are actions that directly perform some computation on the data. Table 3 shows some examples of story actions and computational actions.

Table 3

Examples of Story Actions and Computational Actions

Story Actions	Computational Actions
Action: introduce_hobby(person p, hobby h)	**Action:** convert_meters_to_feet(distance_value d, meters m, feet f)
Precondition: ¬ known_hobby(p)	**Story Req.:** intent_to_convert(d, m, f)
Postcondition: known_hobby(p) ^ hobby_of(p, h)	**Pre:** in_unit(d, m) ^ ¬ known_in_unit(d, f)

Text: The hobby of [p] is [h].	**Post:** ┐ in_unit(d, m) ^ in_unit(d, f) ^ known_in_unit(d, f) **Text:** Complete the function that converts [d] to feet.
Action: state_intent_to_buy(person p, hobby h, item i) **Precondition:** hobby_of(p, h) ^ used_in(i, h) ^ sellable(i) **Postcondition:** intent_to_buy(p, i) **Text:** [p] wants to buy a new [i].	**Action:** compute_area_of_square(item i, square sq, side s, distance_unit d, area a) **Story Req.:** intent_to_compute_area(i) **Pre:** property_of(i, s) ^ property_of(i, a) ^ shape_of(i, sq) ^ introduced(s) ^ in_unit(s, d) ^ ┐ known(a) **Post:** known(a) ^ in_unit_squared(a, d) **Text:** Complete the function that computes for the area of [i] in [d]2.

Each computational action has a story requirement, which is a set of assertions that need to be true in the story for the system to be able to use this as the target computation of the problem. Furthermore, each computational action is also mapped to an exercise representation (discussed in Section 2.1) that is used for generating the exercise solution.

The first phase of the planning aims to determine the computations for the exercises as well as the inputs required for the computations. First, the algorithm selects a random computational action that will serve as the main requirement of the problem. To create multi-step problems, regression may be performed on this action against one or more computational actions. In Figure 3, the main computation was to compute the area of a square in feet. This required that the length of the side of was given in feet. After regression on another action (convert meters to feet), the requirement changed such that the side needed to be given in meters. The union of these requirements along with the story requirement, which is for an intention to compute the area of an object to exist, serves as the goal state of the next phase of the planning.

Next, the second phase of the planning takes the goal state derived from the previous step, and then attempts to build a story context behind the computation by using a backward state space search on the set of story actions in the domain. Backward state space search works backward by starting on the goal state and attempting to reduce it by performing regression against other action. As an example of this, in Figure 4 the goal is that Julia is not hungry. The planner reduces this goal by performing regression on the "eat" action, matching the parameters "Julia" and "ramen". After this, the new goal is for Julia to be hungry and for ramen to be edible. Backward state space search attempts to find a sequence of actions until the goal state is reduced to a state that is true in the initial state of the world.

Finally, the text can be generated by concatenating the templates associated with each action in the sequence. Table 4 shows some examples of generated exercises with real-world computations.

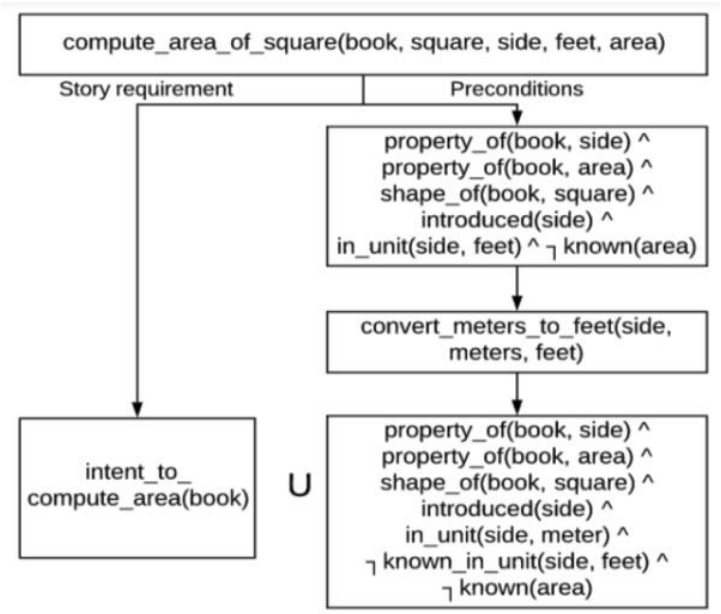


Figure 3. Determining the computation of the exercise through regression

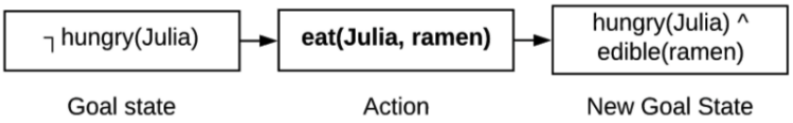


Figure 4. Example regression of a story goal

3. Initial Evaluation and Discussion

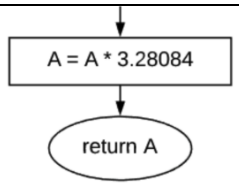
We sought initial opinions from students and teachers of programming with the purpose of determining the viability of the generated exercises to be used for learning programming. In this pre-evaluation, we do not evaluate the personalization aspect of the exercises yet. Instead, we focus only on the quality of the generated exercises in terms of usage in programming education.

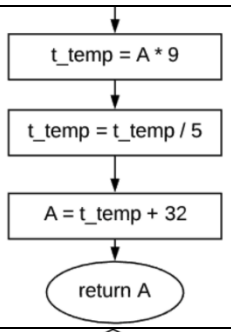
3.1 Student Evaluation

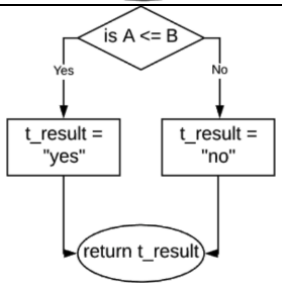
For the student evaluation, the participants are 13 students in a Japanese university. Each student solved 4 abstract level exercises and 4 real-world level exercises randomly generated by the system. The students were asked to rate each exercise on two criteria on a 5-point Likert scale: (1) how easy it is to understand the exercises (5 – very easy, 1 – very difficult) and (2) how engaging it is to answer the exercises (5 – very engaging, 1 – not engaging at all). They can also qualitative feedback on each exercise. Table 5 shows the results.

Table 4

Examples of Exercise Generated with Real-World Computations of Varying Difficulties

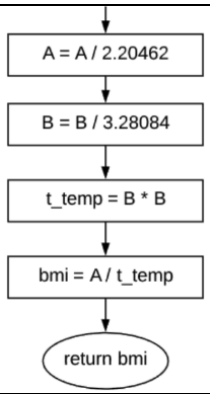
[Difficulty: 1] Neil likes traveling very much. Neil is planning to travel to France for a vacation. France uses the imperial system for measurement. Neil is A meters tall. Neil should convert the height of Neil to feet. Complete the function that converts the height of Neil to feet. 1 meter is 3.28084 feet.
`double func(double A) {`
 	
`}`
	

[Difficulty: 3] It is a rainy day. Thomas walked to the restaurant. Thomas gets sick because of the rain. The body temperature of Thomas is A degrees Celsius. Thomas wants to know the body temperature of Thomas in degrees Fahrenheit. Complete the function that converts body temperature to Fahrenheit. To convert Celsius to Fahrenheit, multiply it by 9/5 and then add 32.
`double func(double A) {`
 	
`}`
	

[Difficulty: 3] Cloud is a doctor. Cloud wants to buy a new stethoscope for work. One stethoscope costs A yen. Cloud has a total of B yen. Complete the function which should return yes if Cloud has enough money to buy the stethoscope, or no otherwise.
`String func(double A, double B) {`
 	
`}`
	

[Difficulty: 4] Thomas is overweight. Thomas wants to get more fit. Thomas
weighs A pounds. The height of Thomas is B feet. Complete the function that computes the body mass index (BMI) of Thomas. To compute the BMI, divide the weight in kilograms by the height in meters squared. 1 meter is 3.28084 feet. 1 kilogram is 2.20462 pounds.

```
double func(double A, double B) {

}
```



On average, the score for "how easy it is to understand" is 4.08 for the abstract level questions and 3.75 for the real-world level questions. A common feedback cited on real-world level questions is some student's unfamiliarity with some concepts such as "body mass index" and "feet". On average, the score for "how engaging it is" is 3.52 for the abstract level questions and 3.75 for the real-world level questions. Several students cited the repetitiveness of the exercises in the abstract level as a reason for lack of engagement.

Table 5

Results of Student Evaluation

		Abstract	**Real-World**
How easy to understand is it?	Avg.	4.08	3.67
	Std. Dev.	1.57	1.42
How engaging is it?	Avg.	3.52	3.75
	Std. Dev.	1.44	1.37

In general, these results show that the exercises can be engaging and easy to understand for the students, but considerations must be made with certain factors such as the wording of the exercises and the student's prior familiarity of the concepts used.

3.2 Teacher Evaluation

We also sought the qualitative feedback of 7 programming teachers who are handling university programming courses. 4 of the teachers were from Japan while 3 of the teachers were from the Philippines. We showed 10 exercises generated by our approach to each teacher, 5 on the abstract level, and 5 on the real-world level. We then asked them to fill up a qualitative survey about the exercises.

Most of the teachers have stated that the exercises on the abstract level are usable for teaching programming (5), but several have also pointed out that the exercises are too simple and straightforward (6) so it is appropriate for beginners (3). Some teachers stated that the exercises do not require students to analyze the problem on a higher level (2). Some teachers stated that they perceive the exercises on the real-world level to be of higher quality (3). Reasons cited are because they are relevant to real world concepts (3) and they not straightforward and thus require the student to do a higher level of analysis (1). However, some teachers stated that the types of computations in the examples given to them are simple and limited for practical use (2). Almost all the teachers have stated that the exercises generated in the abstract level are easy to understand (6). One of the teachers is concerned that the lack of an intention in the context of a real world may cause the exercises to be difficult to understand (1).

For the exercises generated in the real-world level, several have stated that the real-world context of the problems can allow students to imagine and understand them better (3), but there are also concerns that the irrelevant story components that are not necessary for the problem could cause distractions in understanding (2). A common concern, however, is that the grammar and wording of

the sentences sound unnatural (3) at times and could affect understanding. For example, character names are repeated instead of using pronouns.

The teachers perceived the exercises in the abstract level as a way for teaching students the syntax of the programming languages such as remember the syntax of an if-else statement or remembering how to write arithmetic expressions (5), while the exercises generated in the real-world are more appropriate for logic formulation and/or mapping real-world relationships to programming statements (4), although the types of computations might need to be diversified for it to become very useful. The two different levels of abstraction can potentially be used to target different aspects of coding skill.

3.3 Discussion

The results of the evaluation show that while it is possible to generate programming exercises that are understandable and engaging, it is also important to consider the right time to present the exercises. Some students perceived the exercises as less engaging because they are too easy, suggesting that it does not match their skill level. On the hand, the lack of familiarity with some concepts such as "body mass index" was often cited as a reason for lack of engagement.

This shows that the preconceptions and understanding of the student must be considered in deciding the kinds of exercises to be given. This is also consistent with the teachers' feedback that certain types of exercises are suitable only for certain skills of programming. This highlights the need for a system that can adapt to the states of the student, such as skill level or understanding of different concepts.

While students and teachers perceived the contextualization of the exercises into real-world settings as engaging in general, it is unclear as to what extent story elements should be used for programming exercises. Some teachers have stated that they do not find the story elements of the real-world exercises to be necessary apart from the main computation (e.g., convert meters to feet), while others have stated these elements are helpful in contextualizing the problem. More research must be done to evaluate the pedagogical value of these elements. Nevertheless, generation of exercises with a story has the potential for certain types of adaptive feedback such as personalization, which can be helpful in learning (Bates and Weist, 2004).

4. Conclusion

In this paper, we discussed an approach for generating programming exercises that use abstract and real-world computations. Exercise on the abstract level target students' ability to remember the syntax of the programming language, while exercises on the real-world level target students' ability to formulate programming logic based on the relationships of real-world entities. Considerations in generating programming exercises include the wording of the problems, the familiarity of the students with the concepts presented, and the diversity of the types of computations that can be generated.

Future work can further improve the generation process by incorporating natural language processing techniques to increase the quality of the sentence structures generated by the exercise. Ways on increasing the knowledge base can also be explored. There is also a need to assess the pedagogical value of the exercises in the context of learning.

References

Balanskat, A., & Engelhardt, K. (2015). Computer programming and coding: Priorities, school curricula and initiatives across Europe, European Schoolnet.

Bates, E. T., & Wiest, L. R. (2004). Impact of personalization of mathematical word problems on student performance. The Mathematics Educator, *14*(2).

Cabada, R. Z., Estrada, M. L. B., Hernández, F. G., & Bustillos, R. O. (2015, July). An affective learning environment for java. In Advanced Learning Technologies (ICALT), 2015 IEEE 15th International Conference on (pp. 350-354). IEEE.

Digital Promise (2017). Computational Thinking for a Computational World. Report.
FL Network (2014). The four pillars of FLIP.
Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D et al. (2007). Developing a computer science-specific learning taxonomy. ACM SIGCSE Bulletin, *39*(4), 152-170.
Gallien, T., & Oomen-Early, J. (2008). Personalized versus collective instructor feedback in the online courseroom: Does type of feedback affect student satisfaction, academic performance and perceived connectedness with the instructor? International Journal on E-learning, *7*(3), 463-476.
Gerdes, A., Heeren, B., Jeuring, J., & van Binsbergen, L. T. (2017). Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. International Journal of Artificial Intelligence in Education, *27*(1), 65-100.
Hsiao, I. H., Sosnovsky, S., & Brusilovsky, P. (2009, September). Adaptive navigation support for parameterized questions in object-oriented programming. In European Conference on Technology Enhanced Learning (pp. 88-98). Springer, Berlin, Heidelberg.
Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. ACM Transactions on Computing Education (TOCE), *10*(4), 16.
Ministry of Education, Culture, Sports, Science and Technology (MEXT). Programming Education at Elementary School Level. Online Report, 2016.
Prados, F., Boada, I., Soler, J., & Poch, J. (2005). Automatic generation and correction of technical exercises. In International conference on engineering and computer education: Icece (Vol. 5).
Wakatani, A., & Maeda, T. (2016, August). Evaluation of software education using auto-generated exercises. In Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES), 2016 IEEE Intl Conference on (pp. 732-735). IEEE.
Wolber, D. (2011, March). App inventor and real-world motivation. In Proceedings of the 42nd ACM technical symposium on Computer science education (pp. 601-606). ACM.