future internet

Article

Generic Tasks for Algorithms

Gregor Milicic *, Sina Wetzel and Matthias Ludwig

Institute of Mathematics and Computer Science Education, Goethe University Frankfurt, 60325 Frankfurt, Germany; wetzel@math.uni-frankfurt.de (S.W.); ludwig@math.uni-frankfurt.de (M.L.)
* Correspondence: milicic@math.uni-frankfurt.de

Received: 31 July 2020; Accepted: 1 September 2020; Published: 3 September 2020

Abstract: Due to its links to computer science (CS), teaching computational thinking (CT) often involves the handling of algorithms in activities, such as their implementation or analysis. Although there already exists a wide variety of different tasks for various learning environments in the area of computer science, there is less material available for CT. In this article, we propose so-called Generic Tasks for algorithms inspired by common programming tasks from CS education. Generic Tasks can be seen as a family of tasks with a common underlying structure, format, and aim, and can serve as best-practice examples. They thus bring many advantages, such as facilitating the process of creating new content and supporting asynchronous teaching formats. The Generic Tasks that we propose were evaluated by 14 experts in the field of Science, Technology, Engineering, and Mathematics (STEM) education. Apart from a general estimation in regard to the meaningfulness of the proposed tasks, the experts also rated which and how strongly six core CT skills are addressed by the tasks. We conclude that, even though the experts consider the tasks to be meaningful, not all CT-related skills can be specifically addressed. It is thus important to define additional tasks for CT that are detached from algorithms and programming.

Keywords: computational thinking; generic tasks; algorithms; K–12; problem solving

1. Introduction

In our globalized and digitalized world of today, it is essential to be skilled in computational thinking (CT) [1,2]. Wing defines CT as "the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer—human or machine—can effectively carry out" [3]. In spite of an ongoing discussion regarding the definition of CT, most researchers usually refer to Wing, as she was the one who originally coined the term [4]. The amount of scientific research targeting CT has increased dramatically over the last years [2]. Even though most researchers agree that young people should acquire CT-related skills [5], there is less consent on the appropriate age for learning CT and on how this goal can be reached [6]. The latter can partly be attributed to the unclear role of CT in relation to other skills and subjects: Should CT be taught as a separate subject, should it be part of computer science (CS) as some kind of pre-skill (e.g., [7]), or should it be part of other Science, Technology, Engineering, and Mathematics (STEM) subjects due to its parallels to the more general skills of problem solving and logical thinking? Of course, this question cannot solely be answered on a topical level, but also needs to be addressed in the context of existing school curricula [2]. Integrating new content into K–12 education always poses challenges for administrators and teachers alike, as the already tight schedules often do not allow for additional topics or subjects. Before there will be a clear shift allowing for CT to be a key part of K–12 education, there is a need for conveying this integral skill with little effort for teachers and within the scope of existing subjects.

To understand CT in more detail and thus be able to teach it, the concept must become much more tangible for educators, rather than just providing them with an abstract definition. For this purpose, it can be helpful to divide the broad term into more narrowly defined subskills that can be understood more easily. Within a report for the European Commission on how CT could be integrated into compulsory education [8], different distinctions and notions for these subskills as used by other researchers are described and compared. Analyzing their similarities and differences, the authors Bocconi et al. arrive at a set of six core CT skills: abstraction, decomposition, algorithmic thinking, debugging, automation, and generalization. As there is currently no empirically validated division into CT subskills to our knowledge, we will rely on the distinction and definitions as given by Bocconi et al. The first core skill, abstraction, refers to the ability to reduce the unnecessary detail from a real-world problem—or, more generally, a question—while adhering to the essential information that cannot be neglected. Oftentimes, this also involves finding an appropriate mode of representation for a piece of information. Decomposition encompasses dividing a problem into a set of smaller sub-problems that can be solved more easily. After solving each of these problems independently, the solution to the more general problem can be derived by combining and further developing the solutions to the sub-problems. As such, decomposition is closely related to generalization, which allows one to draw conclusions and solve problems based on priorly solved ones and experiences that were made beforehand. This skill also refers to the recognition of patterns and to their rediscovery in other problems. Algorithmic thinking encapsulates another facet of CT, namely the process of finding and describing steps in a clear manner such that it is possible to arrive from an initial state at a desired target state. This description of steps usually means formulating an algorithm, whether in pseudo-code, a block- or text-based programming language, or even in the form of a diagram. Debugging, as an element of CT, is more than just being able to find a bug in a program, even though this can undeniably be one part of it. However, in its core, debugging also describes the ability to critically scrutinize the outcome of a situation, whether this might be an algorithm, a method, or the process of verifying a hypothesis. The last skill, automation, refers to the act of transferring instructions to a computer in a way that the machine knows how to process. This also includes exploiting all the advantages a machine offers, for example, with regard to the execution of repetitive tasks.

In the last years, more and more materials aimed at testing or fostering CT were developed. For example, Zapata-Cáceres et al. [9] developed a test for CT skills for students in primary schools, and Román-González [10] developed one for students in K7 and K8. In a study aimed at measuring international Information Literacy, Fraillon et al. [1] also developed a test for CT. What many of these task sets and tests have in common is the lack of a distinction between different core CT skills.

In prior works, we have already expressed the desideratum to know more about how different core CT skills can be addressed and fostered; e.g., in a case study [11], we met students with highly evolved skills in the area of algorithmic thinking but very few abilities regarding the skill of abstraction. This mismatch led us to scrutinize how single skills can be purposefully targeted. This paper revolves around two central aspects. First, we will introduce the general concept of so-called Generic Tasks (GTs) that can be used in educational contexts and that can facilitate the creation of new content to teach a particular concept. Apart from the concept in general, we propose GTs that can be derived from almost any algorithm. The second part of the paper revolves around the evaluation of these GTs based on a survey with $n = 14$ experts in the field of STEM education. One main purpose of this study is to analyze in detail which core CT skills are addressed by the defined GTs. We conclude that it is not a trivial endeavor to purposefully target core CT skills apart from algorithmic thinking and debugging when working with algorithms. It is thus important to define tasks for CT that are detached from algorithms and programming, specifically targeting other CT skills as well.

2. Materials and Methods

2.1. The Concept of Generic Tasks

A generic task (GT) can be seen as a blueprint or representative of a whole family of tasks with a common underlying structure, format, and aim. As such, it cannot be given to the students per se, but requires a certain input from the teacher. A GT can be modified without much effort, and thus, it is possible to derive further tasks from it that cater to a specific purpose. A set of GTs may, on the one hand, serve as guidance and best-practice examples for teachers, and, on the other hand, function as blueprint tasks that can help teachers to design further related tasks. An example of GTs in the field of mathematics education for linear functions can be found in [12]. In the context of outdoor mathematics, a ramp, which can be found in many places outside, can serve as a central object for many GTs. Taking the measurements Δy and Δx of a ramp, it is possible to formulate different tasks related to its slope. A task could be posed asking for the corresponding linear function $y = mx + c$, the angle of the slope $\tan \alpha = \frac{\Delta y}{\Delta x}$, or the slope in percent $m = \frac{\Delta y}{\Delta x} \cdot 100$. During the Erasmus+ project MoMaTrE (Mobile Math Trails in Europe, http://momatre.eu/downloads/), a whole catalog with GTs for different mathematical topics was created. In [13] a potential use of Generic Tasks is presented. By defining GTs and linking them to specific topics inside the curriculum, it is possible for teachers to easily create a learning path for their students by focusing on a specific topic while still providing them with a variety of different tasks. As such, instead of defining and analyzing only one single task for a specific topic, the impact of conducting educational research on and with GTs on a more abstract level can be much higher, as a whole family of tasks with a common underlying structure, format, and aim is examined.

Further potential can be identified when the concept of GTs is used in conjunction with modern technology. For each GT, a corresponding sample solution can be defined, as well as additional hints. Both can be made available to the students as asynchronous feedback that does not require the interaction of the teacher. Upon modification and, thus, derivation of a task from a GT, the hints and many aspects of the sample solutions could remain intact from the previously defined ones for the GT. This process could also be done automatically by a program or inside a learning environment. At the university level, such a platform called MATeX already exists for many topics in the field of mathematics for engineers [14]. Using MATeX, it is possible to automatically and randomly generate tasks with corresponding solutions for many typical exercises, such as solving linear systems, optimization problems, and linear regressions.

Apart from the obvious time-saving aspects for the educators, the definition of GTs would also be beneficial from a didactical point of view. Not only could hints and sample solutions be automatically generated and provided, but, depending on the answering format, the input could also be immediately validated. A large number of studies describe the outstanding importance of effective feedback: In their meta-analysis, Hattie and Timperley [15] emphasize the value of feedback in students' learning progress. GTs could also play a crucial role to create a personalized learning process and learning environments of the future by addressing specific competencies on different levels. For instance, GTs can support educators in using internal differentiation by providing tasks in different formats and levels of difficulty. In this context, it would also be desirable to define GTs specifically fostering a certain skill. A whole family of tasks derived from GTs could be used to address a student's individual requirements and cultivate his or her corresponding strengths. As described, many steps of this process could be outsourced to a machine and executed automatically.

Definition of the Generic Tasks for Algorithms

Especially in the field of teaching programming, many attempts have already been made to use technology to simplify and improve the teaching and evaluation process. One attempt is to support the students with asynchronous feedback and evaluate the students' solutions automatically. In a literature review [16], 69 of such tools were analyzed and compared. Most of the tools offer only limited feedback or cannot be customized by educators, making usage of those tools very situational. Similar tools also exist for block-based programming languages, such as Snap [17] or Scratch [18]. iSnap [19], for example, provides data-driven hints without the need for an expert to define the hints beforehand. As a consequence, though, the hints only tell the students what they have to do, but not why, i.e., instead of strategic hints [15] supporting the solution process, explicit hints are given, providing content rather than strategies. Another tool that can support teachers is Dr. Scratch [20], which provides students with feedback regarding the implementation of their Scratch code. However, the evaluation of CT-related skills simply based on the students' input is rather difficult, as the underlying concept seems very narrow. For example, adding a block for a self-defined function without any functionality or adding other additional blocks at all will raise the score in the category of abstraction by two points.

We want to present a different approach of supporting educators who are willing to teach CT. As a first step, we define GTs for algorithms and examine which core CT skills can be addressed by them. Working with algorithms is an important and integral part of CT, as it describes the process to "... develop algorithmic solutions to those problems so that the solutions could be operationalized with a computer" [1]. As described, a requirement for a GT is its independence from the corresponding content, i.e., a GT is not a concrete task that can be assigned to a student, but rather a blueprint requiring input to actually derive a specific task. As such, a GT for algorithms should be usable for a whole class of algorithms. Consequently, the GT could be used for a wide range of different approaches and frameworks, like unplugged activities and visual programming in combination with programmable hardware or other representations, like pseudocode and flow diagrams. Providing an additional, e.g., Python, script can transfer the input for such a GT, an algorithm, into a specific task without any other intervention of the educator. The GTs targeting specific CT skills will thus also be usable in and transferable to other tools and learning environments, making it easier for teachers to integrate them into their lessons and different school subjects [5]. At the same time, using a similar structure of tasks like the proposed GTs would also make empirical results easier to compare with each other across different settings.

The following GTs are based on common types of tasks for programming used in CS education [21–24]. However, by defining GTs on an abstract level, we want to emphasize the possibility of utilizing the tasks in different settings with the aim to teach the students CT as a special way of problem solving, focusing on core skills that are also usable in other situations and contexts with algorithms as the basis. In CS education, on the other hand, the primary focus when similar tasks are used often purely lies on writing code [25], although CT-related skills could be fostered as well [26].

As input for all of the following GTs, the <description> and the <code> are required to derive corresponding tasks. If the algorithm produces a graphical output, an <image> of the expected result can also be given and used, either instead of the textual description or as an additional cue.

Simply giving the students the <description> and, if available, the <image>, as well as asking them to write a corresponding algorithm, results in the implementation task. Asking the students what a program does when being executed based on solely the <code> will be called the analysis task. In computer science education, this type of task is also known as "Explain in Your Own Words" [23] or, simply, "Explaining" [21]. A similar type of task called "Tracing" [21] focuses on the execution of a program line by line [23], and would require the educator to select the parts of the algorithm to be analyzed, making the task definition not independent from the algorithm it is based on and, thus, not transferable into a GT that can be generated automatically. Changing the provided <code> slightly and intentionally inserting errors will result in an <erroneous code>, which, on a semantic level, behaves differently from that which is expected. This type of task is also called "Fixing" [24]. Taking the blocks (or lines, for that matter) of the given <code> apart and putting them in a random order will result in a <code puzzle>. In accordance with CS education, we will call that type of task a Parsons problem [27]. Randomly adding additional blocks to the <code puzzle> will result in a <code puzzle redundant>, called Parsons Problem (Distractor) and, as such, a variant of the original Parsons problem. Studies suggest that the Parsons problem (and its variant) can be an efficient and useful way to teach programming [24,28]. An overview in the form of a graphical representation of the necessary input and the different types of output tasks can be found in Figure 1.

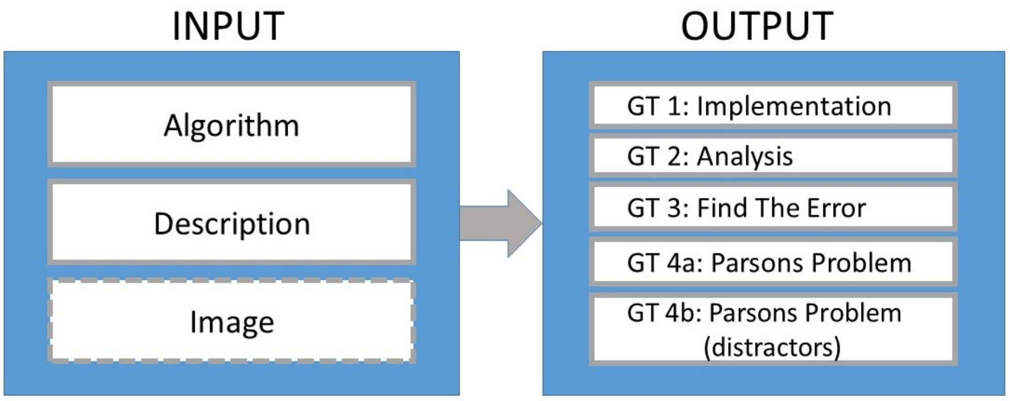


Figure 1. The basic idea for the Generic Tasks (GTs) for algorithms.

Using the components <description>, <code>, and <image> as an input, the four different types of GTs (implementation, analysis, find the error, and Parsons problem) can be defined as follows:

GT 1 Implementation: Create a program for the following description:

<description>

Optional : The result should look like this:

<image>

GT 2 Analysis: What happens when the following program is executed? Describe!

<code>

GT 3 Find the Error: Given is the following program:

<erroneous code>

Originally, the program is supposed to work like this:

<description>

Find and correct the errors so that the program does what it is supposed to do.

GT 4a Parsons Problem: Create a program for the following description:

<description>

Use the following blocks/lines:

<code puzzle>

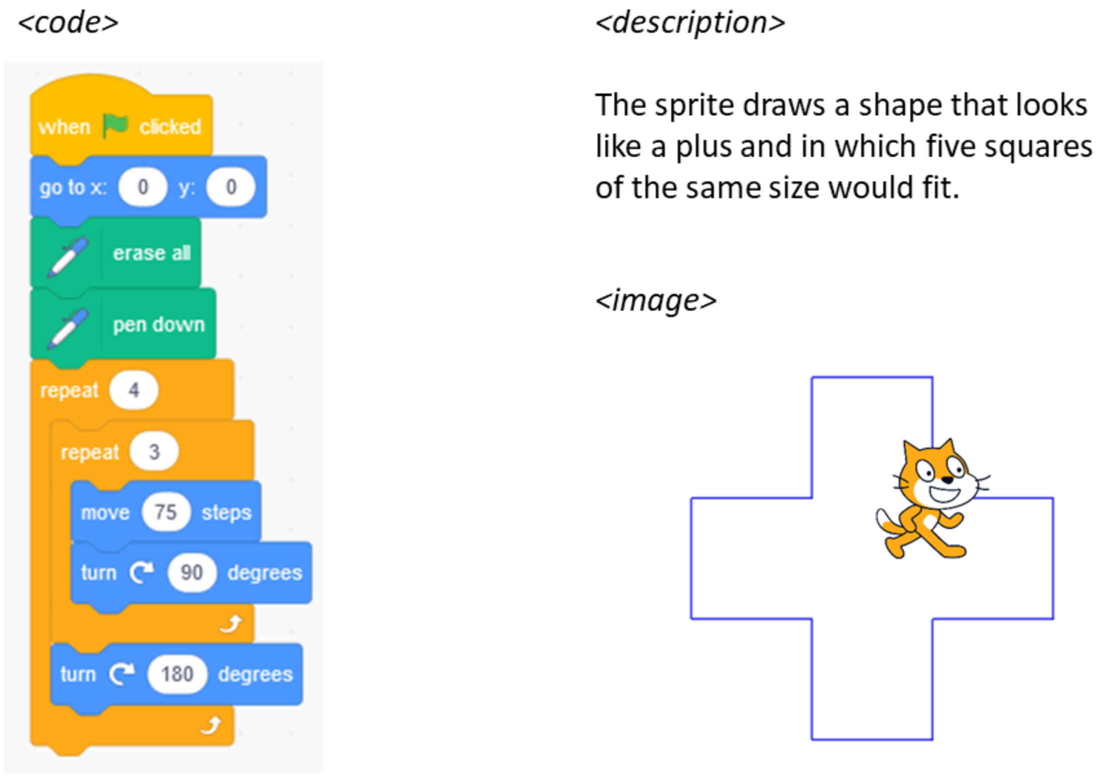
GT 4b Parsons Problem (Distractor): Create a program for the following description:

<description>

Use some of the following blocks/lines:

<code puzzle redundant>

A main pre-requisite for the algorithms to be suitable as input for the GT is that it must be possible to give a clear and precise description of what they do or of the output that they are supposed to generate. Depending on the age and skill level of the target group, the algorithms should also not be too long. Of course, the older and more advanced the target group, the more complex the algorithms can be. One example for a rather short algorithm using a block-based programming language that can serve as input and that also produces a visual output can be seen in Figure 2. As mentioned before, the <image> is only optional as input; however, in some cases, it can make the given <description> much clearer, and can thus prevent the students from misconducting the task. The more automatized the evaluation of a student's answer should be, the more details need to be given in the <description>. For example, if one wants to automatically evaluate a student's answer to the Implementation task for an algorithm like the one in Figure 2, the <description> should also contain measurement information regarding the size of the plus shape. An exemplary set of the corresponding GTs that can be arrived from the algorithm "Plus" from Figure 2 is attached in the Appendix B.



```
when [flag] clicked
go to x: 0 y: 0
erase all
pen down
repeat 4
    repeat 3
        move 75 steps
        turn ↻ 90 degrees
    turn ↻ 180 degrees
```

The sprite draws a shape that looks like a plus and in which five squares of the same size would fit.

Figure 2. One example for possible input.

The approach of defining and using a class of problems and automating the generation and/or evaluation itself is not new. For Parsons problems, a tool called "epplets" already exists [29] for several programming languages (C++, Java, C#), which provides the students with the possibility to solve several Parsons problems. However, the teacher has no influence or control over the given algorithm itself and has to rely on the predefined tasks. In another approach, a JavaScript library is available to include Parsons problems with automated feedback [30], even adapting the difficulty of the problems based on the students' performance [31]. On Runestone [32], an open-source platform for ebooks, educators can define their own Parsons problems and embed them in their ebooks. Using a different approach and conceptual setting like the block model [33], it would also be possible to define other and additional tasks [23]. However, some of those tasks would need more interaction of the educators. In contrast to that, using the proposed GTs facilitates deriving corresponding tasks based on a given algorithm without any further input and for all types of approaches and representations of algorithms.

The defined GTs are not meant to be solved by the students one after another, but rather represent different types of tasks and variations based on an algorithm. There is some evidence pointing towards a hierarchy of programming tasks [22], with Parsons problems (GT 4a and GT 4b) being easier to solve for students than analysis tasks (GT 2). Implementation tasks (GT 1) are considered the most difficult types of problems [21]. The different GTs could thus also be used by educators to address different skill levels of their learners. After generating a corresponding set of tasks based on an algorithm using the defined GTs, more difficult tasks could be assigned to stronger students, whereas weaker students could receive easier tasks. As such, the GTs could also be used as a means to address the heterogeneity of learners. The proposed set of GTs is thus not only to be used for students at a specific age or in a specific grade, as their level of difficulty can be adapted and also depends on the complexity of the input algorithm.

For Snap and Scratch, as two of the most popular block-based programming languages used for programming novices, it would be possible to implement a tool using the input shown in Figure 2 and the defined GTs to automatically generate the tasks presented in Appendix B. A Snap program is stored as an xml file, and a Scratch .sb3 file is just a zip file that has been renamed. The blocks of the program are stored in a JSON data file. It is thus fairly easy to access and manipulate the blocks (GT 3—find the error) or rearrange them (GT 4a and 4b—Parsons problem) for both Snap and Scratch.

Based on the definition of the GTs that we gave above, we raise two main research questions:

RQ1: Which core skills of CT (according to [8]) can be fostered using one of the defined Generic Tasks for algorithms?

RQ2: To which degree are the corresponding skills addressed by the Generic Tasks for algorithms?

RQ1 originates from the desire to create tasks targeting a specific core CT skill. RQ2 builds on RQ1 and aims at answering how effectively the corresponding skills can be targeted and fostered when students work on these tasks.

2.2. Methodology

To validate the proposed GTs, all items were evaluated in an expert's judgment procedure, as proposed, for example, in [9,10]. As the tasks are meant to be used in school and in particular in STEM classes, we invited both in-service STEM teachers and university researchers in the fields of mathematics or CS education who are also certified STEM teachers to serve as experts. As the tasks revolve around algorithms, all invited experts had to have a strong background in programming. A total of $n = 14$ experts from Germany and Austria participated in an online survey in June 2020. Instructions for the experts were given in the survey as well as individually as part of the invitation. In addition to the survey, the experts received an accompanying document containing the definitions of the six core CT skills—abstraction, algorithmic thinking, decomposition, generalization, debugging, and automation, as proposed in [8]. Additionally, the document contained the specific tasks derived from three different algorithms (see Appendix A) using the five proposed GTs. Therefore, the experts had to rate a total of 15 different tasks. All the algorithms were given in the block-based language Scratch. The experts were asked for a general estimation of if they considered the tasks to be meaningful, i.e., whether the students could learn a lot when completing them, on a five-point Likert scale from 1 ("Completely disagree") to 5 ("Completely agree"). They were also asked to rate if and how strongly each task addresses each of the six core CT skills. For the latter, we decided to use a six-point Likert scale from 0 ("Not addressed at all") to 5 ("Very strongly addressed") instead of a five-point Likert scale to avoid providing a middle option so that the experts must at least give a tendency for every skill, whether it is addressed or not.

3. Results and Discussion

The aggregated rating of the experts regarding the meaningfulness of the proposed GTs over all three input algorithms ($m = 3 \cdot 14 = 42$) is shown in Figure 3. Overall, the tasks were received rather positively. For GT 1, GT 2, and GT 3, the vast majority of the experts (73.81%) rather or completely agreed with the hypothesis that the corresponding tasks offer learning opportunities for the students. Noticeably, for the implementation task, almost half of the answers (47.62%) were "Completely agree". GT 1, the implementation task, is thus the most meaningful task to pose, according to the experts. The experts' opinions on the two Parsons problems, GT 4a and GT 4b, are rather similar. In comparison to the other GTs, they consider the Parsons problems less meaningful, as they received more neutral evaluations. This result is rather surprising, as empirical results, e.g., [24,28], suggest that working on Parsons problems can be just as effective for students as when working on tasks similar to GT 1 and GT 3, i.e., writing code fulfilling a specified purpose or finding an error in a given algorithm. A possible explanation could be that, although the concept of a Parsons problem may be known to the experts, the corresponding results regarding its effectiveness may be not.

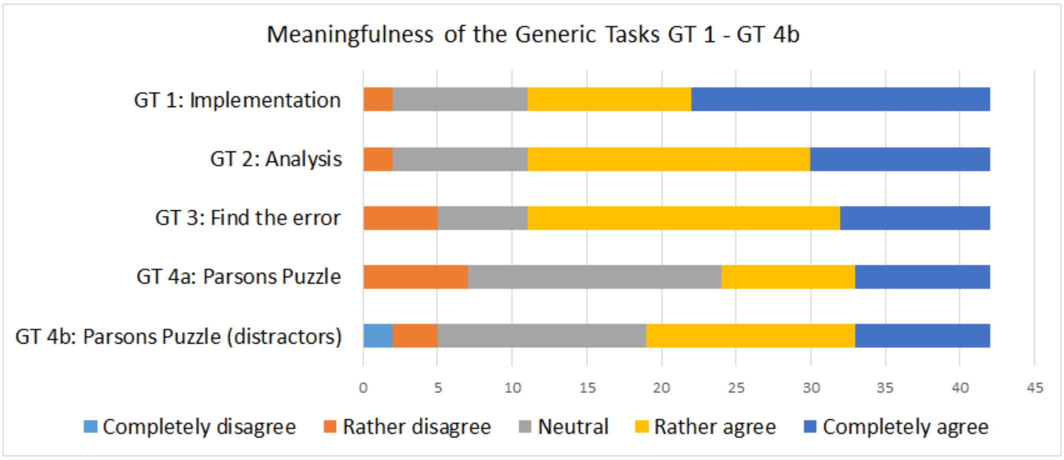


Figure 3. Agreement with the statement "Students can learn a lot when completing this task" for $n = 14$ raters aggregated for the three different algorithms, resulting in $m = 42$ answers per GT.

Performing a calculation of the pair-wise correlations between the three different input algorithms for the raters, estimations regarding the six core CT skills showed that the results all correlate with each other ($r_{12} = 0.69, r_{13} = 0.65, r_{23} = 0.63$). Thus, we decided to analyze the experts' evaluation of the six core CT skills regarding the proposed GTs independently from the input algorithms. We performed a statistical analysis over all $m = 3 \cdot 14 = 42$ answers that were given for each skill and each of the specific tasks belonging to one GT, the results of which can be seen in Figure 4. Several observations can be made. The experts' judgment that GT 1, implementation, is the most meaningful one of the GTs fits together with their assessment that it addresses four of the six core CT skills, namely abstraction (with an arithmetic mean of $M = 3.02$), algorithmic thinking ($M = 4.21$), automation ($M = 3.24$), and generalization ($M = 2.19$), most strongly compared to the other four GTs. This supports the claim that writing code is the most challenging task for students compared to other programming activities, as stated in [21]. Additionally, for four out of the six CT skills, the results of the experts' ratings for GT 1 are the highest over all five GTs. Not surprisingly, as all tasks revolve around the working and learning processes with algorithms, the experts consider the skill of algorithmic thinking to be addressed most strongly by all GTs. Another result that was to be expected is that GT 3, find the error, addresses the skill of debugging the most, with an arithmetic mean of $M = 3.55$. Remarkably, we can report the highest variation of the experts' judgment for the skill of debugging for all five GTs. A possible explanation for this could be that the concept of debugging is mainly known as the activity of finding an error inside a program, rather than it being a CT skill for making assumptions about possible outcomes of algorithms and methods or testing one's own ideas for robustness [8]. As such, the concept of debugging in the context of CT is more general than tracing [23] in CS, which is described as the execution of a program line by line, and could rather be associated with the notion of debugging by some.

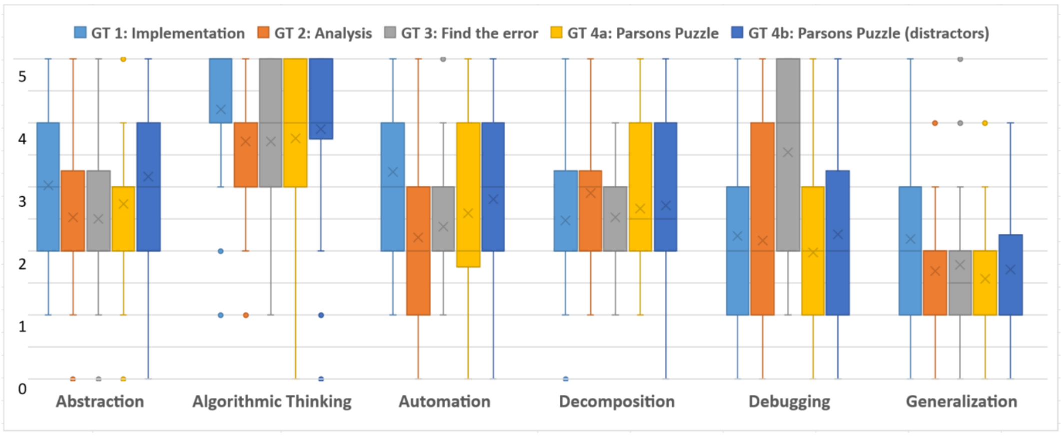


Figure 4. Summary of the experts' evaluations of the GTs regarding the six core computational thinking (CT) skills [8] on a six-point Likert scale from 0 (not addressed at all) to 5 (very strongly addressed). The symbol × marks the arithmetic mean.

We can also observe that the variation of the ratings for most of the skills is relatively high. There are several possible explanations for this. As mentioned in many other works [1,8,25], the notion of CT as a competence and its division into different subskills is still not clearly defined and lacks empirical validation. The raters, although having a strong background in programming, could have not been familiar enough with the core CT skills, apart from algorithmic thinking, to make a proper assessment. The provided definitions of the skills could thus have been not sufficient to support the raters in their judgment. If that were the case, though, it would probably be difficult for them to address these skills in class at all. This, therefore, also hints at the necessity to educate the educators about CT and related skills.

Furthermore, apart from the two skills of algorithmic thinking and debugging, all skills are, on average, rated as being only moderately addressed (mean value of 3) or even less, which implies that the defined GTs are only partially suited to fostering these specific core CT skills. Especially for the skill of generalization, the experts' ratings were quite low, with mean values below 2 for the GTs 2–5. This emphasizes the need for additional tasks and best-practice examples specifically addressing these core CT skills. More advanced programming concepts like functions and objects could help to target skills like decomposition [34] and generalization, but require in return more pre-knowledge from the students [26]. As the defined GTs are based on learning processes with codes and algorithms, which are also the predominant tasks in programming education, we can thus also support the claim that CT is more than just coding [35], requiring more tasks and activities fostering CT skills apart from algorithmic thinking and debugging. In a case study in 2020, we also observed students with a strong interest in CS and highly evolved algorithmic thinking skills only demonstrating basic abstraction skills [11]. This could be a consequence of a strong focus on programming in K–12 computer science education, in which the other CT skills are mostly only moderately addressed, if at all.

As expected by design, GT 4a and GT 4b, which are both variants of Parsons problems without and with distractor blocks, address all six skills on nearly the same level, according to the raters. Both GTs were also deemed similarly meaningful (see Figure 3), with the two bars indicating the distribution of the given answers being very similar. The same holds true for the bars of GT 2 and GT 3, analysis and find the error. Apart from the skill of debugging, all other skills are addressed on a similar level by GT 2 and GT 3, according to the raters. A possible explanation is that with Scratch, we used a block-based language for the declaration of the tasks given to the raters. Without the need to adhere to syntactical standards apart from snapping fitting blocks together, the solution processes for GT 2, analysis, and GT 3, find the error, are very similar. In order to find a potential error (GT 3), the algorithm has to be thoroughly understood, which is also a prerequisite for GT 2 to explain its functionality. As such, GT 3, find the error, can be regarded as the more complex task of the two, as it requires more steps than the analysis task. Our survey would thus support a potential hierarchy of tasks like that proposed in [22], putting GT 2, analysis, and GT 3, find the error, on an intermediate level.

The raters were, as mentioned, not necessarily experts in the area of CT, but were required to be certified STEM teachers with a strong programming background. Their ratings could, therefore, be less accurate, contributing to the variation among the responses. However, this also reflects the status quo in the educational system. If the corresponding educators are uncertain about the skills and how specific tasks are addressing the skills, they can most likely not foster the skills intentionally.

Limitations of the Study

When interpreting the results of this study, one must keep in mind that we asked $n = 14$ experts, which is a rather small sample size. The main reason for not asking more experts was the time that it took to answer all questions thoroughly. Some of the experts reported having needed more than one hour to carefully read the information on the tasks and skills and then complete the questionnaire. Rather than obtaining data from a representative sample, we intended to get a first assessment by the experts regarding the targeted skills and the meaningfulness of the tasks. As these are only estimations based on the definitions we provided the experts with, it will be important, as a next step, to validate the tasks with real students in class. When giving the material to students, we will also assess their pre-knowledge in related areas, such as programming. This will enable us to describe the results in relation to their pre-knowledge, as this will most likely influence how meaningful the tasks are and which skills are addressed.

4. Conclusions

In this article, we defined GTs for algorithms based on common tasks in CS education. Generic Tasks, in general, can be a powerful tool supporting educators in the creation, assignment, and evaluation of tasks for their students. The advantages of GTs are most apparent when they are integrated into an automated learning environment. With the proposed GTs, a teacher can derive five different tasks based on one algorithm. The GTs can thus help to create differentiated tasks and materials within the classroom. For example, as one possible use of the five GTs that we proposed, students could get the assignment to complete some algorithm tasks over the course of one week. To give students the opportunity to choose their own materials and learning paths, points could be assigned to each of the GTs depending on their level of difficulty. The only prerequisite for the students could then be to accumulate a certain amount of points until the end of the week by doing the GTs of their choice for a set of input algorithms given by their teacher.

4.1. Responses to Research Questions

The tasks proposed by us were considered meaningful by the experts in our survey. Interestingly, already in the evaluation of this question, a three-way distinction of the tasks is recognizable. Even though it was to be expected that the GTs 4a and 4b were rated similarly, as they are both variants of Parsons problems, it was more surprising that the analysis task, GT 2, and the find the error task, GT 3, were considered to equally meaningfully target similar skills.

Based on the pair-wise correlations between the raters' estimations, we conclude that the results received regarding the proposed set of GTs do not depend on the algorithms from which they were derived. However, regarding our two research questions, it is difficult to arrive at definite conclusions. As the experts' judgments varied regarding the question of which skills were targeted by the GTs, there are only a few clear conclusions that can be drawn.

One skill that can definitely be fostered by the GTs is algorithmic thinking. GT 1, implementation, seems to be most suitable to foster four out of the six core CT skills, while GT 3, find the error, is best suited to foster debugging. Regarding all other skills, it is unfortunately barely possible to make any clear statements. One reason for this can inherently be attributed to the definition and distinction of the skills. First, the skills might not be disjoint, as an empirical validation is still pending. Second, some of the skills might not be as well known by many educators, as teaching CT is still a rather new idea that has been getting more and more attention and importance only recently [36]. Of course, another reason might be the proposed task design.

4.2. Implications and Suggestions for Future Studies

Based on the high deviation amongst the experts' ratings, two main implications can be elucidated. Firstly, the concepts of CT and its subskills need to be empirically validated in order to distinguish between their different aspects in further research. A clear definition and distinction would also help educators to teach and foster the different aspects of CT. As of now, CT as a concept and its subskills seem to be too vague for educators to successfully target them. This calls, secondly, for more support and dissemination among educators.

Despite the high deviation, our study clearly shows that using algorithms and teaching coding are not enough to convey CT in all its facets. It is thus also important to create tasks for CT that are detached from algorithms and programming. This does not mean, though, that the proposed tasks cannot be used to foster CT. It is just important to keep in mind which skills might be targetable by them and which might be not. Our study thus supports the claim in [35] that CT and programming are not the same thing.

Our future endeavors related to the GTs are threefold. First, we want to create more ideas for GTs targeting other CT skills than algorithmic thinking and debugging. For this purpose, we will need to re-evaluate which the subskills are that CT consists of. Secondly, we need a learning environment in which the GTs are integrated to support educators in teaching CT. A third goal will be to empirically validate the GTs with students.

Author Contributions: Conceptualization, G.M.; Methodology, G.M., S.W., and M.L.; Formal Analysis, G.M. and S.W.; Investigation, G.M. and S.W.; Resources, G.M.; Data Curation, G.M. and S.W.; Writing—Original Draft Preparation, G.M., S.W.; Writing—Review and Editing, G.M., S.W., and M.L.; Visualization, G.M. and S.W.; Supervision, G.M.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding; the APC was funded by Goethe University Frankfurt.

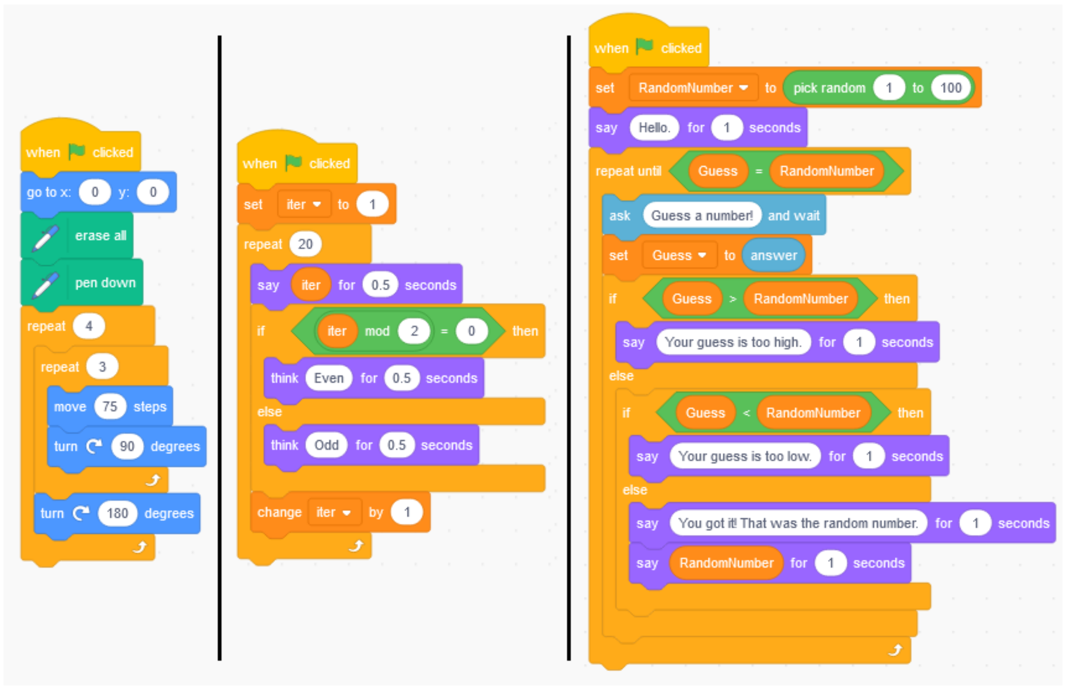
Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CT	Computational Thinking
CS	Computer Science
GT	Generic Task
STEM	Science, Technology, Engineering, and Mathematics

Appendix A. Given Original Examples



```
when [flag] clicked
go to x: 0 y: 0
erase all
pen down
repeat 4
    repeat 3
        move 75 steps
        turn ↻ 90 degrees
    turn ↻ 180 degrees
```

```
when [flag] clicked
set iter to 1
repeat 20
    say iter for 0.5 seconds
    if iter mod 2 = 0 then
        think Even for 0.5 seconds
    else
        think Odd for 0.5 seconds
    change iter by 1
```

```
when [flag] clicked
set RandomNumber to pick random 1 to 100
say Hello. for 1 seconds
repeat until Guess = RandomNumber
    ask Guess a number! and wait
    set Guess to answer
    if Guess > RandomNumber then
        say Your guess is too high. for 1 seconds
    else
        if Guess < RandomNumber then
            say Your guess is too low. for 1 seconds
        else
            say You got it! That was the random number. for 1 seconds
            say RandomNumber for 1 seconds
```

Figure A1. The three original algorithms that the Generic Tasks were derived from (**left**). Plus: The sprite draws a plus sign **(middle)**. Odd–Even: The sprite goes through the numbers from 1 until 20 and thinks about whether the number is odd or even (**right**). Guess a number: The user can guess a number.

Appendix A.1. Plus

<description> The sprite draws a shape that looks like a plus and in which five squares of the same size would fit.
<code> Figure A1 left
URL https://scratch.mit.edu/projects/411733124
<image> Figure A2

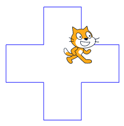


Figure A2. Possible result for the task "Plus".

Appendix A.2. Odd–Even

<description> The sprite goes through the numbers from 1 to 20 and thinks about whether the number is odd or even for each number.
<code> Figure A1 middle
URL https://scratch.mit.edu/projects/411733660.

Appendix A.3. Guess A Number

<description> A random number between 1 and 100 is generated. The user should guess the number. After each input, the program tells the user whether his guess was too high or too low until the random number is guessed correctly.
<code> Figure A1 right
URL https://scratch.mit.edu/projects/411734022.

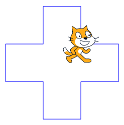
Appendix B. Generic Tasks Based on the Task “Plus”

Using the scheme for the Generic Tasks presented in Section 2.1, for the task “Plus”, as given in Appendix A, it is possible to derive the following tasks.

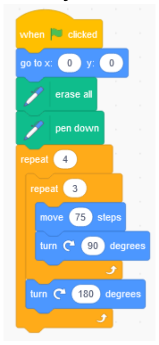
GT 1 Implementation: Create a program for the following description:

> The sprite draws a shape that looks like a plus and that consists of five squares of the same size.

The result of should look like this:

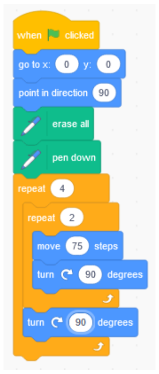


GT 2 Analysis: What happens when the following program is executed? Describe!



```
when 🏳 clicked
go to x: 0 y: 0
erase all
pen down
repeat 4
    repeat 3
        move 75 steps
        turn ⟳ 90 degrees
    turn ⟳ 180 degrees
```

GT 3 Find the Error: Given is the following program:



```
when 🏳 clicked
go to x: 0 y: 0
point in direction 90
erase all
pen down
repeat 4
    repeat 2
        move 75 steps
        turn ⟳ 90 degrees
    turn ⟳ 90 degrees
```

Originally, the program is supposed to work like this:

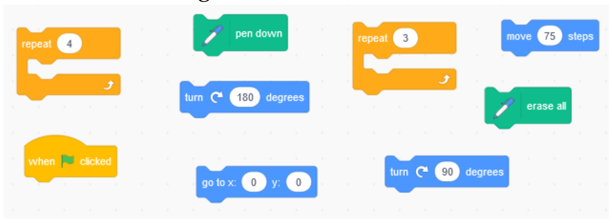
> The sprite draws a shape that looks like a plus and that consists of five squares of the same size.

Find and correct the errors so that the program does what it is supposed to do.

GT 4a Parsons problem: Create a program for the following description:

> The sprite draws a shape that looks like a plus and that consists of five squares of the same size.

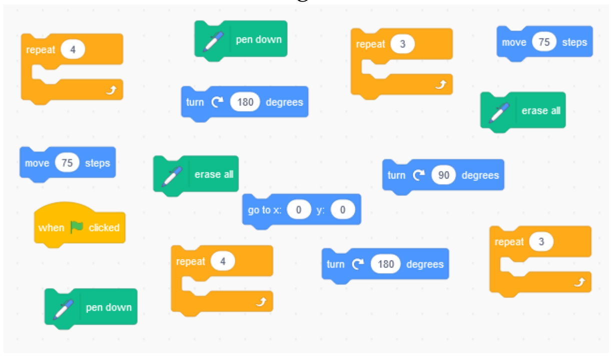
Use the following blocks/lines:



GT 4b Parsons problem (Distractor): Create a program for the following description:

> The sprite draws a shape that looks like a plus and that consists of five squares of the same size.

Use some of the following blocks/lines:



References

1. Fraillon, J.; Ainley, J.; Schulz, W.; Friedman, T.; Duckworth, D. *Preparing for Life in a Digital World: IEA International Computer and Information Literacy Study 2018*; IEA: Amsterdam, The Netherlands, 2019.
2. Pollak, M.; Ebner, M. The Missing Link to Computational Thinking. *Future Internet* **2019**, *11*, 263. [CrossRef]
3. Wing, J. Computational thinking's influence on research and education for all. *Ital. J. Educ. Technol.* **2017**, *25*, 7–14.
4. Wing, J. Computational thinking. *Commun. ACM* **2006**, *49*, 33–35. [CrossRef]
5. Voogt, J.; Fisser, P.; Good, J.; Mishra, P.; Yadav, A. Computational thinking in compulsory education: Towards an agenda for research and practice. *Educ. Inf. Technol.* **2011**, *20*, 715–728. [CrossRef]
6. Hu, C. Computational thinking: What it might mean and what we might do about it. In Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, Darmstadt, Germany, 27–29 June 2011; pp. 223–227.
7. Lu, J.; Fletcher, G. H. Thinking about computational thinking. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education, Chattanooga, TN, USA, 4–7 March 2009; pp. 260--264.
8. Bocconi, S.; Chioccariello, A.; Dettori, G.; Ferrari, A.; Engelhardt, K. *Developing Computational Thinking in Compulsory Education*; JRC Science Hub: Seville, Spain, 2016.
9. Zapata-Cáceres, M.; Martín-Barroso, E.; Román-González, M. Computational Thinking Test for Beginners: Design and Content Validation. In Proceedings of the IEEE Global Engineering Education Conference (EDUCON), Porto, Portugal, 27–30 April 2020; pp. 1905–1914. [CrossRef]
10. Román-González, M. Computational Thinking Test: Design Guidelines and Content Validation. In Proceedings of the EDULEARN15, Barcelona, Spain, 6–8 July 2015; pp. 2436–2444. [CrossRef]
11. Wetzel, S.; Milicic, G.; Ludwig, M. Gifted Students' Use of Computational Thinking Skills Approaching A Graph Problem: A Case Study. In Proceedings of the EduLearn20, Palma de Mallorca, Spain, 6–7 July 2020; pp. 6936–6944.
12. Ludwig, M.; Jablonski, S. MathCityMap-Mit mobilen Mathtrails Mathe draußen entdecken [MathCityMap-Discovering Mathematics Outside with Mobile Mathtrails]. *Mnu J.* **2020**, *1*, 29–36.

13. Barlovits, S.; Baumann-Wehner, M.; Ludwig, M. Curricular Learning with MathCityMap: Creating Theme-Based Math Trails. In Proceedings of the Mathematics Education in the Digital Age, Linz, Austria, 16–18 September 2020.
14. Helfrich-Schkarbanenko, A.; Rapedius, K.; Rutka, V.; Sommer, A. *Mathematische Aufgaben und Lösungen Automatisch Generieren: Effizientes Lehren und Lernen mit MATLAB [Generate Mathematical Tasks and Solutions Automatically: Efficient Teaching and Learning with MATLAB]*; Springer: Berlin, Germany, 2018.
15. Hattie, J.; Timperley, H. The Power of Feedback. *Rev. Educ. Res.* **2007**, *77*, 81–112. [CrossRef]
16. Keuning, H.; Jeuring, J.; Heeren, B. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16), Arequipa, Peru, 9–13 July 2016; pp. 41–46. [CrossRef]
17. Romagosa i Carrasquer, B. The Snap! Programming System. In *Encyclopedia of Education and Information Technologies*; Tatnall, A., Ed.; Springer: Cham, Switzerland, 2019.
18. Resnick, M.; Maloney, J.; Monroy-Hernández, A.; Rusk, N.; Eastmond, E.; Brennan, K.; Millner, A.; Rosenbaum, E.; Silver, J.; Silverman, B.; et al. Scratch: Programming for all. *Commun. ACM* **2009**, 52, 60–67. [CrossRef]
19. Price, T.W.; Dong, Y.; Lipovac, D. ISnap: Towards Intelligent Tutoring in Novice Programming Environments. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17), Seattle, WA, USA, 8–11 March 2017; pp. 483–488. [CrossRef]
20. Moreno-León, J.; Robles, G. Dr. Scratch: A Web Tool to Automatically Evaluate Scratch Projects. In Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE '15), London, UK, 9–11 November 2015; pp. 132–133. [CrossRef]
21. Venables, A.; Tan, G.; Lister, R. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In Proceedings of the Fifth International Workshop on Computing Education Research Workshop, Berkeley CA, USA, 10–11 August 2009; pp. 117–128.
22. Lopez, M.; Whalley, J.; Robbins, P.; Lister, R. Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In Proceedings of the Fourth International Workshop on Computing Education Research, Sydney, Australia, 6–7 September 2008; pp. 101–111.
23. Izu, C.; Schulte, C.; Aggarwal, A.; Cutts, Q.; Duran, R.; Gutica, M.; Heinemann, B.; Kraemer, E.; Lonati, V.; Mirolo, C.; et al. Fostering Program Comprehension in Novice Programmers-Learning Activities and Learning Trajectories. In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '19), Aberdeen, UK, 15–17 July 2019; pp. 27–52. [CrossRef]
24. Ericson, B.J.; Margulieux, L.E.; Rick, J. Solving parsons problems versus fixing and writing code. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17), Koli, Finland, 16–19 November 2017; pp. 20–29. [CrossRef]
25. Moreno-León, J.; Robles, G.; Román-González, M.; Rodríguez, J. Not the same: A text network analysis on computational thinking definitions to study its relationship with computer programming. *Rev. Interuniv. Investig. Technol. Educ.* **2019**, *7*, 26–35. [CrossRef]
26. Hromkovic, J.; Kohn, T.; Komm, D.; Serafini, G. Examples of Algorithmic Thinking in Programming Education. *Olymp. Inform.* **2016**, *10*, 111–124. [CrossRef]
27. Parsons, D. Haden, P. Parsons programming puzzles: A fun and effective learning tool for first programming courses. In Proceedings of the 8th Australasian Conference on Computing Education-Volume 52 (ACE '06), Hobart, Australia, 16–19 January 2006; pp. 157–163.
28. Zhi, R.; Chi, M.; Barnes, T.; Price, T.W. Evaluating the Effectiveness of Parsons Problems for Block-based Programming. In Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19), Toronto, ON, Canada, 12–14 August 2019; pp. 51–59. [CrossRef]
29. Kumar, A.N. Epplets: A Tool for Solving Parsons Puzzles. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18), Baltimore, MD, USA, 21–24 February 2018; pp. 527–532. [CrossRef]
30. Ihantola, P.; Helminen, J.; Karavirta, V. How to study programming on mobile touch devices: Interactive Python code exercises. In Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling '13), Koli, Finland, 14–17 November 2013; pp. 51–58. [CrossRef]

31. Ericson, B.J.; Foley, J.D.; Rick, J. Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. In Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18), Espoo, Finland, 13–15 August 2018; pp. 60–68. [CrossRef]
32. Ericson, B.J.; Miller, B.N. Runestone: A Platform for Free, On-line, and Interactive Ebooks. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), Portland, OR, USA, 11–14 March 2020; pp. 1012–1018. [CrossRef]
33. Schulte, C. Block Model: An educational model of program comprehension as a tool for a scholarly approach to teaching. In Proceedings of the Fourth international Workshop on Computing Education Research (ICER '08), Sydney, Australia, 6–7 September 2008; pp. 149–160. [CrossRef]
34. Rich, K.M.; Binkowski, T.A.; Strickland, C.; Franklin, D. Decomposition: A K-8 Computational Thinking Learning Trajectory. In Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18), Espoo, Finland, 13–15 August 2018; pp. 124–132. [CrossRef]
35. National Research Council. *Report of a Workshop on the Scope and Nature of Computational Thinking*; The National Academies Press: Washington, DC, USA, 2010. [CrossRef]
36. Kafai, Y.; Proctor, C.; Lui, D. From Theory Bias to Theory Dialogue: Embracing Cognitive, Situated, and Critical Framings of Computational Thinking in K-12 CS Education. In Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19), Toronto, ON, Canada, 12–14 August 2019; pp. 101–109. [CrossRef]