Separating Algorithmic Thinking and Programming

Maurice Chandoo[1]

Abstract: We describe an approach to teaching algorithmic thinking and programming and the first experiences that we made with it in practice. The idea is to present computational problems as a certain kind of game that the learner can play in order for them to develop a concrete idea of what constitutes an algorithm. The purpose of this is to emphasize that one can think of algorithms independently of a particular programming language. For the programming part a mini language called machine programs and a method to construct such programs from traces is described.

Keywords: programming education; computational thinking; notional machines; execution traces

1 Introduction

The ubiquity of computers and their immense computing power in our age has contributed to the perception that programming is a generally useful ability. Consequently, it has become commonplace to teach students in other fields than CS how to program. In contrast to CS students, however, they usually only have one course which does not only need to convey the fundamentals of programming but also a minimal understanding of algorithmic thinking.

We describe an approach to teaching algorithmic thinking and programming that can be implemented as a short learning unit. It emphasizes the separation of algorithm design and implementation: design the algorithm first without thinking about a programming language and then implement it in a principled manner using its execution traces. The purpose of our approach is to diverge from the language-centric perception of programming that is often implicitly conveyed ("what commands do I need to write in this programming language to solve the given task") to a more algorithmically-oriented view: "what do I need to compute and how can this be done with the given primitives". Additionally, it should convey the message that designing and communicating algorithms can be done independently of a programming language [La18]. Due to the formal nature of our approach we believe that it might be particularly suitable for mathematically inclined students.

The general idea for teaching algorithmic thinking in our approach is to present computational problems as games to the learner. The learner can test and demonstrate his or her winning strategy (algorithm) by playing the game. We describe a general mechanism to translate computational problems into games. The basis for this translation is a formal definition

[1] FernUniversität in Hagen, Lehrgebiet für Softwaretechnik und Theorie der Programmierung, Universitätsstraße 11, 58084 Hagen, Deutschland maurice.chandoo@fernuni-hagen.de

cc by sa

of *model of computation* from automata theory described in [Sc67]. For example, Turing machines, push-down automata and finite state machines are examples of models of computation in this sense. The definition does not only describe concepts from theoretical computer science but can also express more playful settings such as Kara the ladybug [RNH00] or technical ones such as a simple CPU.

Models of computation are a subclass of notional machines. A notional machine is described as “the general properties of the machine that one is learning to control” by du Boulay who introduced the concept [dB86]. Sorva has given an extensive survey on the relevance of notional machines in programming education and different research threads in this area [So13]. In particular, forming a mental representation of a notional machine is one of the difficulties that novice programmers face. We aim to minimize this difficulty by starting with a very simple model of computation and then generalizing it to a more complex one.

The idea of teaching algorithmic thinking by letting students play games is fairly natural and has been explored and implemented before. For example, a way of using games for specific computational problems as means to understand and develop algorithms in groups has been described by Futschek and Moschitz [FM10]. By algorithmic thinking we mean a set of abilities related to constructing and understanding algorithms as defined in [Fu06]. We see it as a subset of the broader, more popular term computational thinking [Wi06].

In general, we feel that the complexity of translating a mental representation of an algorithm into a program seems to be underestimated. A learner might know an algorithm for a given problem but does not know how to translate it into a given programming language other than by trial and error. In our approach the learner is taught how a formal representation of an algorithm can be constructed from its traces. The traces can be either written manually or generated from playing the game that is used to present the computational problem.

In [Ch19b] we present a programming method that deals with implementing complex algorithms. The method presented here is a simplified version thereof, which is adapted to the context of models of computations and which does not require prior programming experience. Similar approaches are described in [HLR19] and [DvK10]. All have in common that one starts with executing an algorithm by hand for a concrete input and then generalizing the steps to build a program incrementally. In these trace-based approaches, the user must already possess a mental representation of the algorithm that they want to implement. A related approach called direct-manipulation programming is tested in [ADF19]. The importance of traces and tracing activities in early programming courses has been affirmed in [HJ13].

The paper is structured as follows. The first part contains a formal description of models of computation and how computational problems can be framed as what we call machine-computer game. Moreover, a sequence of training tasks which build on each other is given. The second part describes machine programs—a formalism used to describe algorithms—and how to construct them from traces. Finally, we briefly report our experiences with this

approach in an experimental run with two 16-year-old high school students and what next steps are planned for evaluation and further development of this approach.

2 Models of Computation

A model of computation can be seen as an abstract description of a machine: it consists of buttons (operations) and indicator lamps (predicates), and it resides in a particular state (machine state). The indicator lamps (partially) describe its current state. When a button is pressed the machine changes its state depending only on its current state and which button has been pressed. For example, a Turing machine with tape alphabet $\{0, 1, \square\}$ has 5 buttons and 3 indicator lamps. For each character x from the alphabet it has an indicator lamp which is on iff the current cell contains x and a button which writes x to the current cell. It also has buttons to move the head one cell to the left and right. Its state consists of the tape contents and the position of the head.

Formally, a model of computation consists of a set of machine states S, a set of operations (total functions from S to S) and a set of predicates (total functions from S to $\{0, 1\}$). A counter machine with k registers can hold a non-negative integer in each register; a machine state is a sequence of k such numbers ($S = \mathbb{N}_0^k$). Each register can be incremented or decremented by one (decrementing a register that contains 0 has no effect). This means the machine has $2k$ operations. For each register it has a predicate which holds iff that register contains 0.

A simple task on a counter machine with 3 registers is to copy the number of register 1 to register 2. More specifically, the machine is initialized with an unknown state and a human computer has to operate the machine such that register 1 and 2 contain the value that register 1 had in the beginning. This can be presented as a game to the learner as shown in Figure 1. The difficulty is that the indicator lamps (predicates) only reveal partial information about the machine state, i.e. whether a register contains 0 or not.

A trivial strategy for the computer is to determine the number in register 1 by counting (decrement until 0) and then set register 1 and 2 to this value by repeatedly applying the '+1'-operations. This strategy can be generalized to solve arbitrary tasks: determine the initial machine state, solve the problem without the machine, enter the result into the machine. The issue is that no algorithm is executed on the machine itself. To prevent the learner from resorting to this strategy an additional requirement must be made that it should be possible to teach their strategy to a person who cannot count (e.g. a preschool child). More generally, the computer is only allowed to remember a constant amount of information independent of the input.

The learner can train the first steps of algorithmic thinking by trying to find algorithms for simple models of computation and tasks. If the learner consistently wins the game without resorting to the trivial winning strategy described above then this indicates that they found a

Copy

Goal: the final value of register 1 and 2 should be the initial value of register 1

Register 1	Register 2	Register 3
> 0	= 0	> 0
+1	+1	+1
-1	-1	-1
End	Restart	

Fig. 1: Machine-computer game interface for the task copy on a 3-counter machine

correct algorithm. The feedback for the learner is relatively quick (provided the inputs are not too large) and easy to obtain since there is no need to convert the mental representation of the algorithm into a program or an informal description; by playing the game they can check whether their algorithm produces the desired output for a given input.

Any computational problem which can be framed as the task of reaching a particular target machine state from an unknown initial machine state in some model of computation can be presented as machine-computer game. For example, in a gummy bear factory there are two containers: one with sugar and one with food dye. Beneath these two containers is a mixer. A gummy bear consists of three pieces of sugar and two pieces of food dye. There is a button which drops a piece of sugar in the mixer and another one which drops a piece of food dye in the mixer, provided the respective container is not empty. There are also two indicator lamps which show whether the sugar or food dye container is empty. The task is to put as many piece of sugar and food dye as possible into the mixer in the correct ratio (3:2) to produce gummy bears. For example, if there were 60 pieces of sugar and 45 pieces of food dye in the beginning then there should be 60 pieces of sugar and 40 pieces of food dye in the mixer in the end. The challenge is that one does not know how many pieces of sugar and food dye are in the containers in the beginning. Moreover, the trivial strategy involving counting cannot be applied here (because it is not possible to revert the operations of dropping sugar or food dye in the mixer) thus making it a good introductory task to set the learner on the right path.

A stack machine with k registers over an alphabet Σ can hold a string over Σ (including the empty string ε) in each register; a machine state is a sequence of k such strings. For each register it has an operation to remove the last character or append a given character from Σ; removing the last character from an empty string has no effect. For each register and each

character x from Σ it has a predicate which holds iff x is the last character of the string in that register.

In the following, sequences of training tasks for the counter and stack machine are given. We write Ri and Ri' for the initial and final value of the i-th register, respectively. For a string x let $|x|$ denote its length. The right column contains tasks for counter machines and the left one for stack machines over the alphabet $\{A, B, C\}$. The numbers in parentheses after the task names indicate the available number of registers.

- Reverse (2): R2′ = the reverse of R1
- Move (3): R2′ = R1
- Copy (3): R1′ = R2′ = R1
- Concat (3): R3′ = R1 concatenated with R2
- Repeat (4): R3′ = R1 concatenated with itself |R2| times
- Substr (4): R4′ = substring of R1 starting from the $(|R2|+1)$-th character with length |R3|
- Binary (4): R2′ = x times A where x is the value of the binary number in R1 (A=1, B=0, last character is LSB)

- Move (2): R2′ = R1
- Copy (3): R1′ = R2′ = R1
- Add (3): R3′ = $R1 + R2$
- Mult (4): R3′ = $R1 \cdot R2$
- Div (4): $R1 \cdot R3' + R4' = R2$
- Parity (1): R1′ = R1 mod 2
- Prime (6): R2′ = 1 if R1 is prime, 0 otherwise

For the task Substr the initial machine state must be chosen such that it describes a valid substring, i.e. $|R2| + |R3| \leq |R1|$. The game for these tasks can be played at [Ch19a].

We recommend starting with a task like the gummy bear factory that has a concrete scenario before introducing the abstract counter machine. The task Parity is interesting because one cannot infer what the computer does simply by observing them play the game. It could be either that they count the number and determine its parity (illegal) or they keep track of the parity. This becomes visible when the algorithm has to be formalized. The task Prime is a bit more difficult and can be given as optional task. The stack machine can be introduced as a generalization of the counter machine: a counter machine is a stack machine with unary alphabet.

3 Machine Programs

The complexity of general-purpose programming languages can make it difficult for a learner to grasp the basics of programming as they struggle with language-specific details. Mini languages such as Karel the Robot or the drawing turtle try to solve these difficulties

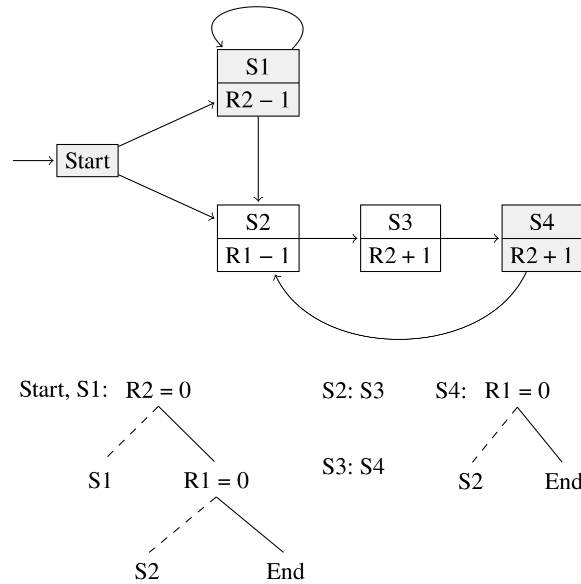

Fig. 2: Machine program for a counter machine computing $R2' = 2 \cdot R1$

by reducing the language to a bare minimum and are suggested for programming courses for non-CS students by Brusilovsky et al [BSS04].

Machine programs constitute a mini language and are a generalized form of finite state automata for arbitrary models of computation. A machine program consists of a set of (program) states of which each is associated with an operation that is executed when that state is reached and of a special state *Start*. After a state has been visited, the program needs to decide with what state to continue or whether to halt. This decision process is encoded by a binary decision tree (BDT). Each inner node of such a tree is a predicate. If the predicate holds, the right (non-dashed) subtree is taken, otherwise the left (dashed line). The leaves are program states or *End* to indicate end of execution.

Figure 2 shows such a program and Table 1 is the trace it produces for the input $(2, 1)$. An empty cell in a register column indicates that the value has not changed. For instance, the first row begins with the start state and register contents $(2, 1)$. Then the BDT of Start is checked. The predicate at the root node is $R2 = 0$ and therefore the program must check whether this holds. Since R2 contains 1 this is false and thus the left (dashed) branch is taken, which leads to the leave with S1. Therefore the program continues execution with the program state S1 in the second row.

While Figure 2 also shows the control flow graph (CFG), the set of states along with their

Tab. 1: Execution trace of program from Fig. 2

Program State	Operation	R1	R2	Predicate Sequence
Start		2	1	R2 = 0 ⇢ S1
S1	R2 − 1		0	R2 = 0 → R1 = 0 ⇢ S2
S2	R1 − 1	1		S3
S3	R2 + 1		1	S4
S4	R2 + 1		2	R1 = 0 ⇢ S2
S2	R1 − 1	0		S3
S3	R2 + 1		3	S4
S4	R2 + 1		4	R1 = 0 → End
End				

operations and BDTs are already a complete description of the program. A gray state indicates that the program's execution can end there.

While it is possible to describe algorithms for models of computation as a sequence of instructions (see [Sc67]) or with conventional control structures, we opted for this particular representation due to the tight correspondence between machine programs and traces which is the basis for the programming method described below.

A fundamental principle of programming is to construct complex programs from simpler ones using an abstraction mechanism such as functions. Subprogram calls are one such mechanism that can be easily integrated into machine programs. Suppose P is a machine program for the model of computation M. P can be treated as if it were an operation of M because it takes a machine state as input and returns one (assuming it always halts). Thus, if one writes a new program P' for M, a program state s of P' can be associated with P meaning: execute P when s is visited and then continue with the BDT of s. There should not be any cycles in the subprogram calls such as P' calling P and vice versa.

Programming Method. The last column in Table 1 describes the path that was taken in the BDT of the program state in that row. For example, in the second row the machine state is $(2, 0)$ after executing the operation R2 − 1. Thus, in the BDT of S1 the predicate R2 = 0 is checked first (true) and then R1 = 0 (false), which leads to S2. This trace table can be used to derive a partial version of the program in Figure 2. The program states and their operations can be directly read from the table. The BDT of a program state can be reconstructed by combining the paths from the predicate sequences. For example, for S4 we combine the paths 'R1 = 0 ⇢ S2' and 'R1 = 0 → End'. Whenever an inner node in a reconstructed BDT has only one child, we add $\emptyset$ as other child which represents a program state with undefined behavior. Applying this procedure leads to the partial program shown in Figure 3. The program state $\emptyset$ has *End* as BDT and an arbitrary operation can be assigned

to it (represented by $*$ in the CFG). Further traces can be added and incorporated until the program is complete (no more undefined behavior).

This method has a consistency guarantee. Suppose that we want to reconstruct a program P from some of its traces T. Let P' be the partial program derived from T as described above. Then P' will either behave identically to P or terminate in the undefined state $\emptyset$. In our example this means the partial program in Figure 3 behaves identically to the complete one for all inputs (x, y) such that $x \geq 1$ and $y = 1$.

The previous example only served to show the correspondence between machine programs and their traces. But since the goal is to construct a program from scratch this begs the question of how to obtain trace tables without a program? Playing the machine-computer game yields a trace containing the operations and machine states. The learner then has to fill in the column for the program states manually. Whenever two rows have different operations they cannot be assigned the same program state. The converse is not necessarily true, i.e. in Table 1 the fourth and fifth row have the same operation but different program states. The predicate sequences can also be filled in manually but this seems to be too complicated in practice. Alternatively, the machine-computer game can be slightly modified to yield these sequences as well. The predicates are hidden from the computer. In order to see one the computer has to ask the machine. The order in which the computer asks to see the

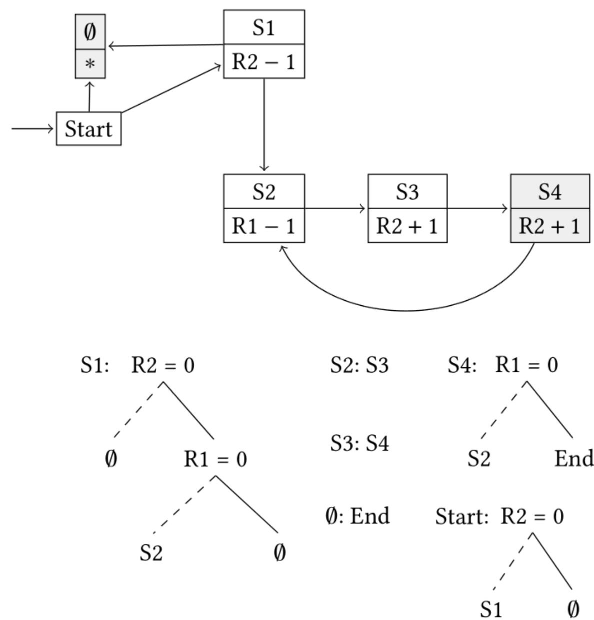

Fig. 3: Partial program constructed from Tab. 1

predicates is the predicate sequence. After an operation is executed all predicates are hidden from the computer again. The key challenge for the learner is to understand under what circumstances two rows correspond to the same program state. After obtaining such a table the rest of this procedure can be automated.

A less rigid way of applying this method is to leave out the predicate sequences. In this case the traces are only used to construct the CFG since the sequence of program states in a trace corresponds to a path through the CFG. Then the BDTs are developed separately knowing which states must occur as leaves from the CFG. Manually constructing BDTs makes it easier to exploit repeating subtrees in them. A hybrid approach could be to partially determine a BDT using a few predicate sequences and then complete it manually.

4 First Experiences

We carried out a trial run with two 10th graders (Gymnasialschüler, both age 16) over 5 days for 3 hours per day. Neither of them had previous experience with programming and both performed very well in math as attested by their math teacher. Moreover, the trial run was held during school vacation without any kind of reward except a certificate of participation implying both had a high intrinsic motivation. We prepared a collection of worksheets based on the tasks described in Section 2, which they had to solve by themselves. We only engaged if there was a conceptual misunderstanding, the description of a problem was unclear or to verify solutions, otherwise both of them worked independently on the worksheets.

The 1st worksheet presented the gummy bear factory problem and another similar one. To describe their solution for the tasks they were given trace tables with columns for the operation and machine state only. The first row in these tables was already given and they had to fill out the other rows to demonstrate how their algorithm would proceed on that input. The 2nd worksheet contained a description of counter machines and the tasks for them (except Div and Prime). Again, they were given trace tables to demonstrate their solutions. Additionally, they were also given the option to play the machine-compute game for the tasks. The 3rd worksheet contained a description of machine programs along with the program from Figure 2 and the trace from Table 1. They had to fill out two trace tables with predetermined inputs for this program to see whether they understood the semantics. The other problem was to reconstruct a machine program from a given trace. The purpose here was to make them aware of the connection between traces and machine programs. The last problem was to implement the algorithms that they found for the previous worksheet as machine programs. The 4th and 5th worksheet described the stack machine and its tasks. The first problem was to find algorithms and demonstrate them using trace tables again. The second problem was to implement them. For the task Binary it contained an explanation of how a binary number can be converted by summing the appropriate powers of two.

At the end of the second day both had already found and implemented algorithms for all tasks up to Mult. Both were able to find algorithms for all of the tasks without help. The

only exception was the task BINARY for which one student didn't find an algorithm but was able to develop one with the help of an example trace that we gave them. Before the workshop ended one student wrote correct machine programs for all tasks except CONCAT and BINARY and the other one for all tasks except REPEAT. The programs that they had to write were quite large. For example, both their programs for SUBSTR had 27 program states and over 60 nodes in the BDTs.

They were given laptops and an online interpreter to write machine programs (see [Ch19a]). At first, they tried to implement their algorithms ad hoc, i.e. directly typing something into the interpreter. While this worked for the small programs up to ADD, they both struggled to implement their algorithms for MULT and the first tasks on the stack machine. Whenever they thought that they fixed a problem in their program by making some modification, they seemed to introduce another one. After realizing this, they switched to using our programming method and wrote their programs using pen and paper. When they copied their finished programs, the programs were either immediately correct or had a minor bug which they were able to quickly identify using the traces produced by the interpreter and resolved it quickly. We consider this to be a positive indicator for our programming method's utility.

The students did not use the column for the predicate sequences when they applied the mehtod. Instead, they only filled out the program state column and used this information to derive a partial CFG, which then was used as basis to construct the BDTs. It seems that a less rigid way of applying the method was more intuitive and useful to them. Additionally, the machine programs for the tasks on the stack machine contained many recurring subtrees in the BDTs; constructing those using predicate sequences would be needlessly repetitive. Since they were constructing the BDTs separately, they were able to recognize these common patterns and exploit them.

For the algorithmic part, the concept with the machine-computer game and writing traces to demonstrate their algorithms worked well in practice. Even before introducing machine programs both seemed to have grasped what constitutes an algorithm in the context of a model of computation. Alternatively, prompting them to play the machine-computer game for a particular task to check whether they have found a correct algorithm also worked well.

5 Outlook & Discussion

From our experiences with the trial run it seems that our approach and the difficulty of the tasks are suitable for a university-level programming course for non-CS students or maybe as an optional intro course for CS students who haven't started their first semester yet (Vorbereitungsveranstaltung). The learning unit outlined here will be implemented and evaluated in a CS course that is part of a preparatory college (Studienkolleg). Moreover, the worksheets and software are being refined to make them publicly available.

A more sophisticated variant of the programming method presented here that can be used to implement complex algorithms has been tried out in an advanced programming course

for CS students and has been met with positive feedback from the students. Our goal is to extend the contents of the concept presented here such that it connects to this advanced method in which algorithms are expressed in a similar manner to machine programs.

Lastly, we want to briefly address the question of evaluating the effectiveness of a programming method. Programming in practice requires various skills such as a good understanding of the programming language in use and the ability to select adequate frameworks and libraries to accomplish a given task. One constant, inevitable aspect of programming, though, is the ability to formalize a given algorithm in some formal language. This ability can be tested in the context of models of computations (MoC) as follows.

The test subject gets a description of a MoC and an algorithm for it along with some example traces. The task is to implement this algorithm in a programming language of their choice such that the implementation is consistent with the example traces. It should be verified that the test subject has correctly understood the algorithm. For example, in an OOPL a MoC could be modeled as a class C whose methods represent the operations and predicates. Whenever an operation is called the operation name and the machine state after applying that operation is printed. Then, implementing the given algorithm means to write a program that only uses a single variable whose type is C such that the traces it produces are identical to the given example traces.

This type of evaluation measures the ability to formalize a given algorithm in a precise sense in a way that prevents the test subject from implementing some 'similar' algorithm that might be easier to implement. This differs from the common testing method where an algorithmic problem is given and the task is to write a program which solves it. Often, efficient algorithms are more challenging to implement than their naive counterparts. Therefore the ability to formalize complex algorithms is desirable but difficult to assess with the common testing method.

Bibliography

[ADF19] Adam, Michel; Daoud, Moncef; Frison, Patrice: Direct Manipulation versus Text-based Programming: An experiment report. In: Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, Scotland, UK, July 15-17, 2019. pp. 353–359, 2019.

[BSS04] Brusilovsky, Peter; Shcherbinina, Olena; Sosnovsky, Sergey A.: Mini-languages for non-computer science majors: what are the benefits? Interact. Technol. Smart Educ., 1(1):21–28, 2004.

[Ch19a] Chandoo, Maurice: , https://upsl.uber.space/aws19/info.txt, 2019.

[Ch19b] Chandoo, Maurice: A Systematic Approach to Programming. 2019.

[dB86] du Boulay, Benedict: Some Difficulties of Learning to Program. J. Educational Computing Research, 2(1), 1986.

[DvK10] Desel, Jörg; von Klenze, Leo: AMSEL - ein Lernsystem zum Algorithmenentwurf. In (Kerres, Michael; Ojstersek, Nadine; Schroeder, Ulrik; Hoppe, Ulrich, eds): DeLFI 2010 - 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V., 12.-15. September 2010, Universität Duisburg-Essen. volume P-169 of LNI. GI, pp. 33–44, 2010.

[FM10] Futschek, Gerald; Moschitz, Julia: Developing Algorithmic Thinking by Inventing and Playing Algorithms. In: Constructionism, Paris. 2010.

[Fu06] Futschek, Gerald: Algorithmic Thinking: The Key for Understanding Computer Science. In: Informatics Education - The Bridge between Using and Understanding Computers, International Conference in Informatics in Secondary Schools - Evolution and Perspectives, ISSEP 2006, Vilnius, Lithuania, November 7-11, 2006, Proceedings. pp. 159–168, 2006.

[HJ13] Hertz, Matthew; Jump, Maria: Trace-based teaching in early programming courses. In: The 44th ACM Technical Symposium on Computer Science Education, SIGCSE '13, Denver, CO, USA, March 6-9, 2013. pp. 561–566, 2013.

[HLR19] Hilton, Andrew D.; Lipp, Genevieve M.; Rodger, Susan H.: Translation from Problem to Code in Seven Steps. In: Proceedings of the ACM Conference on Global Computing Education. CompEd '19, ACM, New York, NY, USA, pp. 78–84, 2019.

[La18] Lamport, Leslie: If You're Not Writing a Program, Don't Use a Programming Language. Bulletin of the EATCS, 125, 2018.

[RNH00] Reichert, Raimond; Nievergelt, Jürg; Hartmann, Werner: Ein spielerischer Einstieg in die Programmierung mit Java, Kara to Java - erste Schritte beim Programmieren. Inform. Spektrum, 23(5):309–315, 2000.

[Sc67] Scott, Dana S.: Some Definitional Suggestions for Automata Theory. J. Comput. Syst. Sci., 1(2):187–212, 1967.

[So13] Sorva, Juha: Notional machines and introductory programming education. TOCE, 13(2):8:1–8:31, 2013.

[Wi06] Wing, Jeannette: Computational Thinking. Communications of the ACM, 49:33–35, 03 2006.