Contents lists available at ScienceDirect

International Journal of Child-Computer Interaction

journal homepage: www.elsevier.com/locate/ijcci

Research paper

The causes of difficulty in children's creation of informal programs

Monica Bucciarelli [a,*], Robert Mackiewicz [b], Sangeet S. Khemlani [c], P.N. Johnson-Laird [d,e]

[a] *Dipartimento di Psicologia, Università di Torino, Centro di Logica, Linguaggio, e Cognizione, Università di Torino, Torino, 10124, Italy*
[b] *Department of Psychology, SWPS University of Social Sciences and Humanities, Chodakowska 19/31, 03-815 Warsaw, Poland*
[c] *Navy Center for Applied Research in Artificial Intelligence, Naval Research Lab, Washington, DC 20375, USA*
[d] *Emeritus, Department of Psychology, Princeton University, Princeton, NJ 08540, USA*
[e] *Department of Psychology, New York University, New York, NY 10003, USA*

A R T I C L E I N F O

Article history:
Received 10 July 2021
Received in revised form 12 October 2021
Accepted 8 December 2021
Available online 14 December 2021

Keywords:
Computational thinking
Deduction
Informal programs
Kinematic simulations
Recursion

A B S T R A C T

We present a theory of the causes of difficulty in children's creation of informal programs. Ten-year-old children are able to devise such programs to rearrange the order of the cars in trains on a simple railway track with a single siding. According to the theory, they rely on kinematic mental models that simulate the required sequence of steps, and we devised a computer program, mAbducer, which does so too in creating its own programs for such rearrangements. An experiment showed that a simple measure of the complexity of its programs, based on Kolmogorov complexity, predicts ten-year-olds' difficulty in this task: the measure is the number of words in mAbducer's programs for solving the rearrangement in a minimal number of moves. Complexity, in turn, reflects the structure of the required programs, which need loops of moves to be repeated, and often moves before and after such a loop. Children's errors are predictable in both their location and nature. Our results therefore have implications for the assessment and pedagogy of computational thinking.

1. Introduction

Informal programs underlie people's everyday activities, such as cooking from recipes, assembling kits, and making place settings for a dinner table. Even five-year-old children are able to understand ordered sets of instructions and to create them when, for example, they have to construct a safe place for a little dog to play (Ehsan & Cardella, 2017). Likewise, the following instructions from a toy train set are an example of informal program that children can follow in daily life:

(1) Unpack your train set.
(2) Visually inspect each piece for any obvious manufacturing flaws.
(3) Start to build your railway track with the feeder piece.
(4) Connect pieces of straight track to the feeder piece *until the track has the length you desire.*

As this example illustrates, an *informal program* is a description in natural language of a series of instructions that use an input, such as the pieces of a toy railway track, to produce an output, such as a layout of the track. The example contains a recursive component in the form of a loop as signaled by, "until the track has the length you desire".

Psychologists have investigated the comprehension and formulation of informal programs much less often (fewer than 300 hits on Google scholar) than they have studied formal programming (over 1500 hits on Google scholar). This neglect is surprising, because informal programming is easier but illuminates how people create programs, and so it can help to identify the key abilities on which formal programs depend. These mental processes are the foundation of computational thinking, and its pedagogy and assessment is likely to depend on understanding them.

The aims of the present paper are to elucidate computational thinking, and they are mirrored in its plan. It begins with a theory of how children who know nothing about programming could rely on kinematic models to create informal programs. It argues that the crucial component of programs, and the heart of recursion, is a loop of instructions, and that the resulting structure of a program yields the likely locus and nature of errors in programming. It postulates that the difficulty in creating a program depends on the complexity of its structure, for which it proposes a measure based on Kolmogorov complexity (see Li & Vitányi, 1997). Next, it presents an experiment in which children had to devise informal programs to rearrange the order of cars in toy trains on a railway track. Its results show that a measure of complexity predicts the likelihood that the children erred, that their use of loops correlates with the accuracy of their programs, and that children's errors are systematic rather than haphazard.

* Corresponding author.
E-mail address: monica.bucciarelli@unito.it (M. Bucciarelli).

https://doi.org/10.1016/j.ijcci.2021.100443
2212-8689/© 2021 Elsevier B.V. All rights reserved.

Finally, the paper draws some conclusions from this research about the testing and development of children's computational thinking.

2. Recursion and the theory of informal programming

2.1. The formulation of programs

Computer programming depends on two sorts of reasoning: deduction and abduction. Deduction is the process of inferring what follows from premises, and so it is essential for testing whether a program is correct. Programmers infer from various inputs the consequences of each step in a program, and its components, in order to determine whether they correspond to the required outputs. However, it is impossible to deduce a program from a description of what it is supposed to compute and examples of its required outputs. This conclusion follows at once from the fact that no general deductive procedure can even determine whether or not certain loops of instructions ever halt (see, e.g., Johnson-Laird et al., 2021). Programming calls instead for *abduction*, that is, inferences that go beyond the premises to introduce at least one novel concept, not explicit in either the description of the required computation or its examples. Given an appropriate description of these concepts and the required computation, it is always possible in principle to deduce a program. But, from the standpoint of cognitive science, this maneuver evades a crucial question: how does human thinking abduce new ideas? Computational thinking is therefore founded on deduction and abduction, so that "their solutions can be represented as computational steps and programs" (Aho, 2012, p. 832).

Any computable function has in principle an infinite number of different programs for its computation. Many of them are trivial variants of one another, but a major difference concerns the implementation of recursion. In programming, recursion is often thought of as a highly specialized definition of a function that calls itself, i.e., it is self-referential (see, e.g., Johnson-Laird et al., 2021). The task of devising such functions is difficult, if not impossible, for ten-year-olds (Dicheva & Close, 1996), and understanding how they work is difficult for eleven-year-olds (Kurland & Pea, 1985). The original concept of recursion, however, is in the theory of computability, and it concerns the definition, not of programs, but of the functions that they compute. This concept of recursive functions translates into a much simpler idea than programs that call themselves—the idea of a *loop* of instructions. Primitive recursion, which suffices for all informal programs and nearly all formal programs, is equivalent to a *for*-loop in which instructions are repeated *for* a given number of times. Certain functions, however, cannot be computed in this way. They call for a *while*-loop in which instructions are repeated *while* a given condition holds. These loops can also be used instead of *for*-loops to compute primitive recursive functions (see (Johnson-Laird et al., 2021), for a primer on these matters). The fundamental abduction in computational thinking is therefore to create an appropriate loop of instructions if the computation of a function requires it.

According to the theory of mental models, deduction and abduction in programming rely on kinematic models, which unfold in time in the same temporal order as the sequence of events that they represent (Johnson-Laird, 1983, p. 423). These processes are easier to grasp from a concrete example, and so we will illustrate them using the environment in our experiment (see Khemlani et al., 2013). It consists of a railway track with cars that can move along the track and onto and off the siding. One track runs from left to right (hereafter, left track and right track, respectively), and the siding can be entered only from left track and exited only to left track. The environment can be presented in a computer program, which we use in studies of adults (Khemlani et al., 2013). But, in studies of ten-year-old children, we used the toy railway shown in Fig. 1. Only three moves are allowed in order to rearrange the order of the cars in a train, where n denotes one or more cars:

R n: Move n cars from left track to Right track.
S n: Move n cars from left track to the Siding.
L n: Move n cars from the siding back to the Left track.

The participants moved cars for themselves in becoming familiar with the set up, but they were not allowed to move them when they devised programs for rearrangements.

An infinite number of rearrangements exist in principle. One of them reverses the order of the cars in a train, so that, for instance, their order ABCDEF on left track is rearranged to FEDCBA on right track. A program to do so makes no use of the letters on cars, which are to help participants to keep track of the whereabouts of the cars, but depends solely on the relative positions of cars, e.g., the car at the end of a train in a *reversal*, A, becomes the head of the train in its rearrangement. Alternative proposals of how people comprehend informal programs suggest that the creation of a reversal program might depend on assertions in the "language of the mind" to represent the state of the railway and knowledge of recursion, and then the use of formal rules of inference to infer a solution (see, e.g., Pylyshyn, 2003). However, the model theory postulates instead that individuals envisage the effects of moves using a kinematic mental model. So, to compute the reversal above, they realize that they cannot move A over to right track to be at the head of the rearranged train, because the other cars are in the way. The first step must therefore be to move these other cars onto the siding so that they are out of the way. The effect of the move is shown in this diagram:

A[BCDEF]–

which is similar to the state of the kinematic model that the computer program mAbducer uses (see below): it represents A on the left track, BCDEF on the siding which the square brackets demarcate, and '–' as the empty right track. The rest of the reversal consists in moving car A on the right track:

–[BCDEF]A

and then in moving each car, one at a time, from the siding to left track and then over to the right track, resulting in the required solution:

–[–]FEDCBA

The sequence of moves yielding this solution is as follows, where *N* is the number of cars in the train:

S(N − 1) R1 L1 R1 L1 R1 L1 R1 L1 R1 L1 R1

S(N − 1) refers to moving one less than the number of cars in the train to the Siding, and R1 refers to moving one car to the Right. The preceding sequence describes the required number of repetitions of L1 and R1 moves for a train of six cars, but a program could instead describe them in a loop.

Many programs, such as this one for a reversal, need a loop if they are to apply to trains of any length. Hence, the use of a loop in the formulation of a program calls for a major shift in computational thinking. Not only does the program work for trains of any length, but it also is more parsimonious for a train of a fixed length. When children simulate a rearrangement, and grasp that it calls for a sequence of moves to be repeated more than once, they may be able to describe the sequence in an explicit loop. The larger the number of cars in a rearrangement, the fewer the words in a program using a loop of moves, and so children should be more likely to have the insight to use them. Loops put a greater load on working memory in deducing the consequences of a program (e.g., Bucciarelli et al., 2018), but they may improve the accuracy of programming.

A program for a reversal using a *for*-loop is as follows, where N is the number of cars in the train:

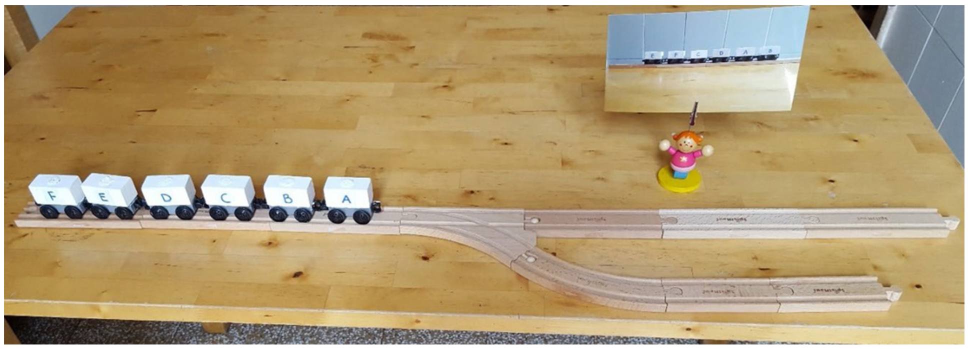

Fig. 1. A toy train with six cars, the track, and the picture of the required rearrangement, used in the experiment.

```
S(N − 1)          – move N-1 cars from left track to the siding.
R1                – move one car from left track to right track.
For (N − 1)       – for N-1 times, move one car from the
times: L1 R1.     siding to left track and then over to right
                  track.
```

Naive programmers who devise a correct *for*-loop may be surprised to learn that to determine its required number of repetitions calls for them, in effect, to solve two simultaneous linear equations. But, a *while*-loop should be easier to devise, because it calls for programmers only to note the conditions in which the loop is entered, and in which it halts. For a reversal, a *while*-loop is repeated as long as there is at least one car on the siding. So, its program is as follows:

```
S(N − 1)
R1
While at least 1 car is on the siding:
    L1
    R1.
```

The discovery of loops is the essential abduction in creating a program for a rearrangement.

As we have mentioned, we devised a computer program, mAbducer, which itself could solve rearrangements, create programs for those depending on a single loop, and deduce their consequences—all for trains with any finite number of cars. Its source code in Common Lisp is available at: http://mentalmodels.princeton.edu/programs/mAbducer-1.0.lisp. It uses kinematic simulations to create its programs using both *for*-loops and *while*-loops, and it also translates the latter into informal English (see Table A.1 in Appendix A for examples of all three sorts of its program for each of the problems in our experiment reported below). It bases its programs on solutions that it discovers to have the smallest number of moves—a task that is feasible for small-scale problems. It uses the same method for all rearrangements including reversals: it searches for loops in sequences of moves that solve a rearrangement, and for the required moves, if any, before and after a loop (for the details, see Johnson-Laird et al., 2021).

Our previous studies showed that adults who know nothing about computer programming can abduce informal programs for rearranging trains of any length, and they can deduce their consequences for a given train (Khemlani et al., 2013). Ten-year-old children have the various abilities needed for these tasks: they can carry out means-ends analyses (e.g., Kuhn, 2013), plan (e.g., Aamodt-Leeper et al., 2001), and make mental simulations (e.g., Ianì et al., 2020). Indeed, when we tested such children, we discovered that they could also devise informal programs and deduce their consequences (e.g., Bucciarelli et al., 2016, 2018). They were therefore the participants in our present experiment.

2.2. The complexity of programs

Our experiment examined the following five sorts of rearrangement.

Input	Output	Our name for the rearrangement
ABCDEF	BADCFE	Swap adjacent pairs
ABCDEF	FEDCBA	Reversal
AABBCC	ABCCBA	Make palindrome
CCBBAA	ABCCBA	Two-loops palindrome
ABCDEF	ACEBDF	Parity sort

Previous studies showed that children and adults can create some recursive programs more easily than others. The differences tended to be systematic, e.g., most children can infer a program for a reversal, but few can infer one for a *parity sort* (Bucciarelli et al., 2016). The difficulty of finding the moves for a rearrangement depends largely on the number of moves it calls for, e.g., a reversal, is quite difficult to solve, because of the number of moves it calls for (see, e.g., Khemlani et al., 2013). In contrast, the principal source of difficulty in abducing a program to make the rearrangement does not depend on the number of moves it elicits. It seems to depend on the complexity of the program. But, how is this factor to be assessed? We explored a number of ways to measure it, including the nature of the working memory required to carry out the program, the number of loops it calls for, and the number of times a rearrangement has to be repeated to get back to the original order of the train (Johnson-Laird et al., 2021). However, the simplest and easiest to assess was a version of Kolmogorov complexity, K-complexity for short. It uses the number of words in a program written in a standard programming language as the measure of the program's complexity (Li & Vitányi, 1997). We used the number of words in mAbducer's translations into informal English of its *while*-loop programs for the five sorts of programs in our experiment (see Table A.1 in Appendix A).

2.3. Errors in informal programs

Errors could occur at any step in the creation of a program. The first step for a programmer is to understand what the program is intended to compute. Some rearrangements, such as reversals, make it obvious. Others are less transparent, such as the parity sort:

ABCDEF $\Rightarrow$ ACEBDF

where the double arrow represents the function—from the initial state of the train to its required final state. It calls for all the cars in odd-numbered positions, ACE, to be behind all the cars in even-numbered positions, BDF. No systematic study, as far as we know, has compared the difficulty of understanding what a program is supposed to compute from the presentation of examples of its computation. A verbal description of the required function is likely to be helpful, but it is also likely to present clues to the required informal program. Hence, in our experiment, we eschewed such descriptions in favor of a presentation of examples of input and required output. And we asked our participants to describe their understanding of the nature of the rearrangement.

The next major step in creating programs for the five rearrangements above is the abduction of loops. People often err. The theory predicts that they are more likely to do so for programs with higher K scores. Their errors could be arbitrary, so that the only way to simulate them would be to make a random move in a program, and to do so with the same probability that an error occurs among the participants. Yet, individuals differ reliably in their ability to create accurate programs (see, e.g., Bucciarelli et al., 2016), and errors may tend to occur at particular places in the formulation of a program. Accurate programs for the five sorts of rearrangement above tend to have three principal stages (see Table A.1 in Appendix A): pre-loop instructions, one or more loops, and post-loop instructions. Some of these stages are not required for certain programs, e.g., *swap adjacent pairs* has only a loop of instructions for its minimal solution. The three stages yield potential places for sources of error (see also Appendix B for their occurrence in programs for particular rearrangements):

1. *The pre-loop moves*. Only two sorts of initial move are possible, S or R, though the number of cars in a move can vary. Likewise, unless the first instruction is an R move, there are three sorts of second instruction. Hence, it should be harder to formulate a correct second instruction than a first instruction. Errors in general are more likely in this stage, because it sets up the environment for a loop.

2. *Repetitions in a loop*. When the instructions in a loop concern the same car, then individuals are more likely to describe the loop correctly than when the instructions concern different cars in the loop. For example, the description of a loop such as L1 R1 in a reversal should be likely to be correct, because the instruction calls for a single car to be moved from the siding to left track and thence to right track. In contrast, the description of an S1 R1 loop in a parity sort (see Table A1) should be less likely to be correct, because its instructions call for one car to be moved to the siding, and then a different car to be moved to right track. To formulate the required loop, the programmer has to change focus back from the siding to left track, and to formulate an instruction concerning a different car. This difference should also occur when a program merely repeats the same sequence of moves more than once instead of formulating an explicit loop.

3. *The post loop moves*. After participants have described a loop or an equivalent sequence of repetitions, they may forget where they are in formulating the program. Hence, they may perseverate with the loop, terminate it too soon, or describe an erroneous post-loop move. Perseveration is likely when at the end of the loop there are still cars on the left track. Otherwise, termination of the program is likely.

We turn now to the experiment that we carried out to test the theory's predictions.

3. The experiment

The experiment called for ten-year-old children to devise their own rearrangement programs using the railway domain. The participants were children at this age because, as we pointed out earlier, they have all the abilities needed to devise informal programs, e.g., they can plan and they can make kinematic simulations (Bucciarelli et al., 2016). But, they are more likely to err than adult programmers, and the experiment aimed to investigate errors. The experiment was designed to test whether K-complexity predicted children's difficulty in devising accurate programs, whether there was any relation between their use of loops and the accuracy of their programs, and whether their systematic errors corroborated the model theory.

3.1. Method

3.1.1. Participants

After their parents had given their informed consent, 28 fifth-grade children (15 females and 13 males; mean age = 10;4 years, sd = 0.3) attending two schools in Turin, Italy, took part in the experiment. The schools did not include courses in programming and, as we confirmed, none of the participants had taken such courses elsewhere. The experiment had the approval of the Ethical Committee of the University of Turin.

3.1.2. Design and materials

The children acted as their own controls and had to formulate programs for the five rearrangements shown above (see Table A.1 in Appendix A for the programs that mAbducer devised). The experiment used a toy train on a wooden track with a siding, and the cars in the train had letters as labels (see Fig. 1). For each rearrangement, the children first devised a program for a train of six cars and then immediately afterwards a program for a train of eight cars. This second program was prefaced with the additional instruction to make a short rule to rearrange the train, which aimed to encourage children to abduce a more parsimonious description, such as a loop of instructions. Each child carried out the five sorts of rearrangement in a different random order.

3.1.3. Procedure

The children were tested individually in a quiet room and in the sole presence of the experimenter. They began by learning the three rules for moving the cars, and how to describe moves using only the number of cars in a move and not the letters on the cars. For each problem in the experiment, the child saw the initial arrangement of the cars on the left track and the required rearrangement of the cars depicted in a photograph behind the right track. We chose to use only examples of the input and output, because a description of the rearrangement could bias the participants to use the same terms in their program, so that they parroted parts of the question back in their programs (see Pane et al., 2001). However, before the children tried to create each program, they were invited to reason about the difference between the initial and final arrangements until they noticed the relevant differences. The children with the assistance of the experimenter, if necessary, noticed the following aspects of the rearrangements:

- the cars are swapped round two by two (*swap adjacent* pairs),
- the cars are reversed in order (*reversal*),
- the first adjacent pair of cars with the same labels, CC, is in the center, then on their sides
  there are the cars of the second pair, then those of the third pair (*make palindrome*),
- the last pair of cars, CC, moves to the center, then on their sides there are the cars of the pair
  that comes after, then those of the last pair (*two-loops palindrome*),
- the cars in even-numbered positions, BDF, move in front of those in odd positions, ACE (*parity sort*).

This procedure and the examples of each rearrangement sufficed for most participants to appear to understand their task.

The experimenter read the instructions to each child, and the key instruction (translated from the Italian) was:

> Try to tell me in words, without moving the cars, how would you form this train [in the photo]. Remember not to use the names of the cars, but tell me how many cars move from one part of the track to another.

Table 1
The percentages of accurate programs that the children in the Experiment (N = 25) devised for the five sorts of rearrangements of trains of six cars, trains of eight cars, and overall, and the K-complexities of correct programs, i.e., the numbers of words in minimal informal programs using while-loops (see Table A.1 in Appendix A).

Sorts of rearrangement	Percentages of accurate programs			K-complexities
	Trains of six cars	Trains of eight cars	Overall	
Swap adjacent pairs	68	72	70	38
Reversal	40	60	50	40
Make palindrome	44	24	34	44
Two-loops palindrome	56	24	40	48
Parity sort	24	28	26	54

The children spoke their descriptions aloud, move by move. When they had finished their description of a program for six cars, the experimenter added two cars to the train on the left track, and said:

> Now, we make this train longer. It is the same train as before, just longer; it has eight cars. And the cars have to be rearranged in the same way as before. Can you make a short rule to rearrange the train? A rule that could rearrange the train even if it was longer than eight cars, a train with any number of cars in it.

We video-recorded the experimental sessions, and two independent judges coded the recordings to make explicit each instruction in the program that a child described. The judges identified correct and incorrect instructions. For a more refined assessment of the accuracy of a program, they also scored the number of correct R moves prior to the first erroneous R move. Readers will recall that an R move—to the right of the track—cannot be undone. Hence, this number provides a finer-grained measure of accuracy than whether or not a program is correct: even erroneous programs can have some correct R moves. The number of correct R moves, however, depends on the number of cars in a train, because a longer train can call for programs to make more R moves than those for a shorter train. We therefore converted the number of correct R moves into percentages, where a correct program has a 100% correct R moves, and a program with an erroneous first R move has a 0% correct R moves. In case an informal program exploits a loop, it can contain the same number of instructions for both six cars and eight cars. If a participant made a redundant adjacent pair of moves, such as R3 R1, instead of a single R4 move, they were still counted as four correct R moves.

The judges also noted the occurrence of any loops of moves, both explicit loops and implicit loops in repetitions of two or more moves, and distinguished three sorts of explicit loops:

- *While*-loops specified the termination condition in advance, e.g., "and so on until no cars are left on the siding".
- *For*-loops specified the number of iterations in advance, though they often did so by using a quantifier such as "all" to refer to the unknown number of cars on a particular part of the track, e.g.: "one by one take all the cars and lead them back [to the left track] and then to the goal".
- *Proto*-loops specify neither the termination condition nor the number of iterations, but indicate that the same move will be repeated, e.g., "and so on", "and we go always like that", and "we move the car from the side to the left then to the goal, and also the last one".

Table A.2 in Appendix A presents typical protocols of children's descriptions of programs.

3.2. Results and discussion

The results of three of the 28 participants were dropped from the analysis, because two of these children failed to formulate any program for at least one rearrangement, and one child's descriptions were too confused for definite transcription. For the remaining 25 participants, the two judges agreed in their coding of the programs on 98% of trials (Cohen's κ =.96, $p < .0001$). They also agreed on 99% of trials about the occurrence of no loops, proto-loops, *for*-loops and *while*-loops in the programs (Cohen's $\kappa = .97$, $p < .0001$). They resolved the discrepancies prior to the statistical analyses.

3.2.1. K-complexity predicts the difficulty of programs

Table 1 presents the percentages of accurate programs. K-complexity predicts the trend in accuracy of the participants' pairs of programs over the five sorts of rearrangement (Page's $L = 1195$, $z = 2.80$, $p < 0.01$), a result that corroborates the model theory's account of complexity. The children differed in their ability to abduce programs (Friedman non-parametric analysis of variance, χ^2 (9) = 35.65, $p < .0001$). The most accurate children abduced nine out of ten correct programs, but one child abduced no correct programs.

Boys tend to encounter computer programming more often than girls. To determine if this explained any gender differences in our data, we compared the performance between them. The boys created 43% correct programs, and the girls created 45% correct programs, and the difference was not reliable (Mann–Whitney test, $z = .08$, $p = .93$). So, the two sexes appear to be equally good in understanding programs (for a review, see Du & Wimmer, 2019).

As the model theory predicts, children used more loops for their programs for trains of eight cars (28% of trials) than for their programs for trains of six cars (20% of trials), whether the programs were right or wrong (Wilcoxon test, $z = 2.14$, $p < .04$, Cliff's δ = .17). This finding may merely reflect the special instruction to find a short rule given to the participants just before they tackled the eight-car rearrangements. It is possible that if the instruction had been given from the outset for the six-car trains, the difference would have disappeared. The difference in accuracy between programs for eight cars (42% accurate) and those for six cars (46% accurate) was not reliable (Wilcoxon test, $z = 0.99$, $p = .32$).

3.2.2. The use of loops correlates with the number of correct r moves

Children often used loops in their descriptions, whether the program was correct or not. The most frequent sort were *for*-loops (92% of loops), but 59% of them were formulated with an explicit number of repetitions based on the length of the train, e.g., "keep repeating these moves for the remaining two couples of cars", and the remaining 41% depended on an explicit quantifier, e.g., "keep repeating until all the trains have moved from the siding"—a description that is akin to a *while*-loop. But, the children hardly ever couched the explicit conditions for a *while*-loop (only 3% of loops), and their descriptions included only a small proportion of proto-loops (5%).

Table 2 presents percentages of correct R moves prior to the first erroneous one (see above) depending on whether or not the children's programs used an explicit loop of any sort. As the Table

Table 2
The percentages of correct R moves in the Experiment (N = 25) as a function of the children's use of explicit loops in their programs for six-car and eight-car trains over the five rearrangements.

Sorts of rearrangement	Percentages of correct R moves for six-car rearrangements		Percentages of correct R moves for eight-car rearrangements		Overall % of correct R moves
	No use of loop	Use of loop	No use of loop	Use of loop	
Swap adjacent pairs	65	83	61	88	75
Reversal	38	73	45	91	58
Make palindrome	62	100	49	50	56
Two-loops palindrome	78	83	57	88	69
Parity sort	55	61	53	63	55
Overall % correct R moves	61	78	52	85	62

Table 3
The distribution of errors over the children's programs in the Experiment (N = 25) as percentages of the main places in programs where the errors occurred, and the balance of error-free programs. Rows in bold highlight the most common location of the different sorts of error.

Junctures	Location &/or sort of error	Percentages
Pre-loop	In first instruction	3
	In second instruction	8
	In subsequent instructions	**15**
Loop	About same car	3
	About different car	**12**
Post-loop	Perseveration	2
	Stopping too soon	2
	Other	**11**
No errors		44

shows, for all ten rearrangements the use of a loop occurred with a higher percentage of correct R moves (Binomial test, $p = 1/2^{10}$). The advantage of using loops was reliable for the participants' programs for both six-car (Wilcoxon test, $z = 2.17$, $p < .04$, Cliff's $\delta = .33$) and eight-car rearrangements (Wilcoxon test, $z = 2.84$, $p < .025$, Cliff's $\delta = .01$); and it did not differ reliably between them (Wilcoxon test, $z = .74$, $p > .4$).

3.2.3. *Systematic errors in children's programs*

Table 3 presents the percentages of the different sorts of errors at different places in the children's programs (see Appendix B for examples of each sort of error at these places). Its broad picture bore out the theory's predictions. Errors in the first instruction (3%) were fewer than those in second instructions (8%; Wilcoxon test, $z = 2.05$, $p < .05$, Cliff's $\delta = 0.21$). The numbers of opportunities for error differ in loops or repetitions of moves and in post-loop sequences. Nevertheless, as predicted, such errors were less likely when they concerned the same cars (3%) than when they concerned different cars (12%), but the difference was not reliable (Wilcoxon test, $z = 1.41$, $p > .15$). In post-loop sequences, as predicted, perseveration occurred for *make palindromes* (13%) more often than for the other two relevant rearrangements (0%; Wilcoxon test, $z = 2.27$, $p = .023$, Cliff's $\delta = 0.29$). Likewise, premature stops occurred more often for *parity sorts* (a mean of 24%) than for *make palindromes* and *two-loops palindromes* (0%), but the difference was marginal (Wilcoxon test, $z = 1.63$, $p = .051$, Cliff's $\delta = 0.16$).

4. General discussion

The model theory postulates that individuals abducing an informal program rely on a kinematic model that simulates the moves needed for the rearrangement. The present study concerned the likely causes of errors in the process for ten-year-old children. The difficulty of a mere solution to a rearrangement of a particular train depends on its number of moves and the total number of cars to be moved. Hence, reversals are the hardest rearrangements to solve of the five in the present study, because they call for more moves of cars (Khemlani et al., 2013). But, the difficulty of abducing a program for a reversal is very different: its abduction depends on the complexity of the program. And so, as Table 1 shows, reversals are among the easiest of programs to abduce. As we explained earlier, the Kolmogorov complexity (K-complexity) of mAbducer's informal programs—the number of words that they contained—is a sensible way to predict the children's difficulty in abducing programs. And it made a reliable rank-order prediction of their difficulty (see Table 1). Because the children dealt with each eight-car program immediately after a six-car program of the same sort, the difference in the number of cars in programs had negligible effects on performance, and as a result this difference did not interact reliably with other factors, such as accuracy over the five sorts of program.

The percentage of correct R moves in a program is a more sensitive measure of accuracy than whether or not the program is correct: an R move cannot be undone, but even erroneous programs are likely to include some correct R moves. This measure showed that programs containing loops yielded a reliably higher percentage of correct R moves than programs containing no loops (see Table 2). It is tempting to construe this relation as causal: the use of a loop in the creation of a program, which is a significant step in computational thinking, leads to greater accuracy in the program. But, of course, the correlation between the two does not establish causation. It is conceivable that some other factor, such as expertise in computational thinking, underlies both the use of loops and accuracy in programming. An experiment of the sort needed to establish a causal relation may not be feasible. It would call for children to use loops in one condition, but not in another, and some way to ensure that they never thought of loops in the second condition.

The errors in children's programs tended to occur at predictable places (see Table 3). Most errors occurred at the start of programs, but others did occur in loops and post-loops. The nature of the errors tended to follow a predictable pattern, but their small numbers led to the predicted differences not always being reliable. We can have some confidence, however, that errors are not haphazard, but tend to reflect the structure of the programs that the children are trying to abduce. The same error was often made by more than one of the children (see Tables S1 in the Supplementary Materials). At most places in a program, there are three possible sorts of move, which each can concern one or more cars. Hence, a particular error with a chance probability of, say, 1 in 5, is not haphazard when, as often happened, more than two children make it.

The development of an informal program depends on abduction. Individuals have to discover the concept of a loop of instructions, which can be of various sorts, and such loops go beyond the mere repetition of a set of moves. They call for an explicit description of what controls the repetition of the loop. So, naive individuals cannot create a program solely from deduction or inductive generalization. The process depends on an analysis of the sequence of moves that a kinematic solution yields, and the creation of a way to describe the sequence that in an ideal case works for trains of any length—a requirement that often necessitates the use of a loop of instructions. There are exceptional rearrangements that do not call for a loop, e.g., switching round the first and last cars in a train can be done without a loop: these six instructions work for a train of any length: S(N − 1), R1, L(N − 2), R(N − 2), L1, R1, where *N* equals the number of cars in a train. Deduction plays a role in programming—prudent individuals test a program by deducing its consequences. So, a deduction from the preceding program applied to the train ABCDEF yields the following sequence: A[BCDEF]–, –[BCDEF]A, BCDE[F]A, –[F]BCDEA, F[–]BCDEA, –[–]FBCDEA. Naive individuals, however, cannot devise a program from deduction alone.

In conclusion, ten-year-old children are able to develop informal programs to compute rearrangements in the order of cars in trains. They simulate solutions to the rearrangements, and often discern the underlying structure of a program that could carry out the same rearrangement on trains of different lengths. What causes them difficulty is, not the number of moves needed in the rearrangement, but the complexity of the program, which depends on its structure—in the case of rearrangements, whether it calls for a pre-loop set of instructions, a loop of more than one instruction, and a set of post-loop instructions. The use of a loop of instructions is crucial for certain programs, e.g., to reverse the order of cars in trains of any length. The required structure of a program also yields important junctures in its creation, and they tend to give rise to predictable errors in its description. Children who are not allowed to move the cars as they try to develop a program often make revealing gestures—they point to cars that are to be moved, and their gestures often indicate the start and end points of moves (Bucciarelli et al., 2016). And, these gestures may support strategies in computational thinking (Ianì, 2019).

Psychologists, as we noted earlier, have rarely investigated the comprehension and formulation of informal programs. But, among the exceptions is a study of the ways that non-programmers, ten-year-olds and adults, formulated solutions to problems representative of common programming tasks (Pane et al., 2001). The results showed that children described looping constructions in a few cases. Their loops used *until* to specify a terminating condition—a *while*-loop, and otherwise they were proto-loops in phrases such as *and so on* or *etc*. The experimental procedure differed from ours, because the experimenter outlined to the participants essential programming techniques and concepts, such as the use of variables, prior to the experiment. Nevertheless, its results corroborated the importance of the structure of the required programs. The investigators pointed out that informal programs differ in style from the then contemporary programming languages. They also suggested that natural languages differ from programming languages, and so informal programmers tend to think in a different way about programs than formal programmers do (see Biermann et al., 1983). Their results, they suggested, could guide designers of future programming languages to make them more natural, and to match the strategies that naive individuals bring to the programming task (Bruckman & Edwards, 1999; Soloway et al., 1989). Likewise, their participants' methods could inform intelligent tutoring systems for teaching programming and for scaffolding its acquisition (Lane & VanLehn, 2005).

In the light of recent developments in machine learning and automatic programming, the need for human programmers is likely to wane (see, e.g., Sharma et al., 2020; but cf. O'Neill & Spector, 2020). A fundamental constraint, however, is that no automatic method can guarantee to find out whether or not a given program goes into an infinite loop (see, e.g., Johnson-Laird et al., 2021). Other recent innovations include the machine translation of informal programs in a natural language into formal programming languages (see, e.g., Sridhar & Sanagavarapu, 2020; and the Codex system, as illustrated at: openaicodexlivedemo). So, although most current programming depends on two intertwined skills: creating a program and formulating it in a given programming language, the crucial task is to devise the algorithm that a program implements. Algorithms existed long before the invention of programmable computers. Their creation depends on an understanding of a set of elementary instructions, and the ability to combine them, especially in loops of operations. Both these skills can be inculcated and tested using the railway computer, which allows individuals to envisage and to master them. Natural languages differ from programming languages, not in computational power (see Johnson-Laird et al., 2021), but chiefly in ambiguity, which in English and most Indo-European languages can be both syntactic and lexical. One consequence is that no program exists for recovering a logical analysis of serious arguments in natural language—a lack that the logician Bar-Hillel described half a century ago as "a scandal of human existence" (Bar-Hillel, 1969, p. 256). Yet, the concepts embodied in high-level programming languages, such as self-referential processes or object-oriented procedures, can all be expressed in everyday English. For skilled programmers, such expressions are messy and often harder to understand than their counterparts in Lisp or Python, but these drawbacks do not affect the potential for natural language as a programming language.

One aspect of programming that our studies of rearrangements have neglected is arithmetic, and the programming of arithmetical functions. Yet, simple modifications to the railway environment provide it in principle with the power of a universal Turing machine that, as far as anyone knows, can compute any computable function. The essential modification is to allow new cars to be inserted into a train, existing cars to be removed from a train, and additions to the track to accommodate longer trains.

4.1. *The pedagogy of programming*

The present study and its precursors showed that individuals differ in their ability to create informal programs. These differences are likely to reflect interactions among motivation, intrinsic ability, and pertinent experience. History shows that ideal learners are autodidacts with access to expert help. The rest of us, however, need explicit teaching. This requirement raises two problems: how can teaching institutions assess the likely competence of naive individuals as programmers? And how can the teaching of programming be improved?

Existing tests of computational ability focus on high school or university students in the context of learning a programming language, and on middle school children in the context of visual programming languages, e.g., Blockly or Scratch (for a review, see Romàn-Gonzàlez et al., 2018). All these tests presuppose that aptitude is a summation of abilities, and so different test items address different components of programming. For example, a test item may concern a problem whose solution depends on the use of a loop, and the participants have to decide which is the correct option among a choice of several (e.g., Romàn-Gonzàlez, 2015). Other methods of assessment include the Computational Thinking Test aimed at students between 10 and 16 years old (see e.g., Romàn-Gonzàlez et al., 2018), the Beginners Computational Thinking Test (BCTt) in its adaption aimed at younger students (Zapata Cáceres et al., 2021), and assessments based on Bebras problems (Lockwood & Mooney, 2018). A typical problem of this latter sort is (see Dagiene & Stupuriene, 2016):

> Beaver is working the crane operator. There are two boxes in the parking place—A and B. At first box A is standing on the 1st podium, box B is on the 2nd podium. There are six commands to operate this crane: 'Down', 'Up', 'Right', 'Left', 'Catch', and 'Release'. They are selected by pressing a command button. Help beaver to switch boxes: box A must be on the 2nd podium, and box B on the 1st.

The scenario is presented with a picture of the objects in their initial spatial arrangement and the six command buttons. The participants have to devise a program for carrying out the swap of boxes A and B. These tests have in common a lack of a valid measure of the theoretical difficulty of different sorts of problem. The railway environment provides such a measure, K-complexity, and it provides a potential test of computational thinking prior to students having learned to code.

For students who are already interested in programming, a variety of systems exist for learning its elements. Computational kits exist for young children, which use blocks or puzzle-pieces to represent code for controlling robots or virtual entities (for a review, see (Yu & Roque, 2019). They allow children to try out sequences of instructions, conditional branches, and even loops. They have an advantage over board games, which also exist as a way to acquire rudiments of programming. Games can be fun, but they tend not to allow for loops of instructions, or for testing the correctness of a program (Scirea & Valente, 2020).

The desirable features of an interactive system that enables users to learn how to develop programs are almost self-evident: they should be fun to use, they should present a large and diverting set of problems that can be ordered in increasing difficulty, and they should help learners to understand how to create programs from combinations of primitive actions, and how to test their adequacy. The railway environment seems feasible as such a system, and for learners with access to computers, mAbducer and its implementation for on-line experiments could be developed into an interactive pedagogical device. Intrinsic motivation to learn is essential, even in computer programming, and self-efficacy matters too (see, e.g., Bjerre & Dohn, 2018). No-one knows what attracts some children to learn to program (or to any other intellectual activity), but girls and boys do differ in the social factors that affect them, e.g., some girls dislike interacting with a computer in the presence of other people (Cooper & Weaver, 2003). Motivation also relates to learner's implicit beliefs that they can develop in ability, which in turn affect how they regulate their actions as they learn (Cornoldi et al., 2003).

The important tasks for future research are twofold. First, it needs to examine children's ability to create informal programs to compute the values of arithmetical functions, such as addition and multiplication: an extension of the railway environment can deal with such functions. These studies may illuminate how children understand numbers and programs that compute new functions from them. Second, future research needs to test whether the railway environment can help budding programmers to understand how to combine existing programs—our present research has been limited to studies of single functions. A special case concerns the mastery of recursive functions for combining existing programs, at first in *for*-loops and *while*-loops, and then in self-referential functions. Formal programming languages, as we have intimated, may be obsolescent. Yet, there is still a need to test the effectiveness of a transfer from informal programming to full-fledged programming in formal languages.

Selection and Participation

The children in the experiment were attending two public primary schools in Turin, Italy. A researcher contacted the managers of the schools and described the experiment, which the Ethical Committee of the University of Turin had already approved. It took the form of a game, and the school managers explained it to the fifth-grade teachers, and invited them to explore the children's interest in taking part in 'a game whose scope was to help a university student to write a report on how children reason'. The teachers gave an informed consent form to the children who were interested, and asked them to give it to their parents to read, and to sign if they approved their children's participation in the study. The form stressed that the parents could withdraw their consent at any time, and that the children could likewise withdraw from the study at any point. Only those children who returned the signed form took part in an experiment. The experiments took place at the schools during class hours, not breaks, in a designated room solely for them. None of the children, in fact, ever withdrew from the study. They enjoyed it.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The Supplementary Materials and the data reported in this paper are archived at the following database: https://osf.io/47pgb/?view_only=d9a204db954542d5a5e63e5d2634f686. The first author thanks Compagnia di San Paolo for support Grant SMaILE N. 2020.1718 [BALM_CSP_AI_21_01]. The second author thanks the Polish National Science Centre for support in Grant 2014/14/M/HS6/00916.

Appendix A

See Tables A.1 and A.2.

Table A.1
mAbducer's *for*-loops and *while*-loops and its English translations for the five rearrangements used in the present studies, and the K-complexities of the while-loops (the number of symbols, including words and parentheses, in the description) and their translations.

Programs with *for*-loops	Programs with *while*-loops	Translations of *while*-loops
1. Swap adjacent pairs	K-complexity: 37 words	K-complexity: 38 words
(Defun Swap(track) (Let* ((len (Length (First track))) (N-of-reps (+ (* 1/2 len) 0))) (Loop For I from 1 to N-of-reps do (setf track (S 1 track)) (setf track (R 1 track)) (setf track (L 1 track)) (setf track (R 1 track))) track))	(Defun Swap (track) (Let ((len (Length (First track)))) (Loop While (> (Length (First track)) 0) do (setf track (S 1 track)) (setf track (R 1 track)) (setf track (L 1 track)) (setf track (R 1 track))) track))	While there are more than zero cars on the left track, move one car to the siding, move one car to the right track, move one car to the left track, move one car to the right track.
2. Reversal abcdef fedbca	K-complexity: 41	K-complexity: 40 words
(Defun Reverse (track) (Let* ((len (Length (First track))) (N-of-reps (+ (* 1 len) −1))) (setf track (S (+ (* 1 len) −1) track)) (setf track (R 1 track)) (Loop For I from 1 to N-of-reps do (setf track (L 1 track)) (setf track (R 1 track))) track))	(Defun Reverse (track) (Let ((len (Length (First track)))) (setf track (S (+ (* 1 len) −1) track)) (setf track (R 1 track)) (Loop While (> (Length (Second track)) 0) do (setf track (L 1 track)) (setf track (R 1 track))) track))	Move one less than the cars to the siding. Move one car to the right track. While there are more than zero cars on the siding, move one car to the left track, move one car to the right track.
3. Make palindrome AaBbCc ABCcba	K-complexity: 45 words	K-complexity: 44 words
(Defun Make-Pal (track) (Let* ((Len (Length (First track))) (N-of-reps (+ (* 1/2 Len) −1))) (setf track (S (+ (* 1 Len) −2) track)) (Loop for I from 1 to N-of-reps do (setf track (R 1 track)) (setf track (L 2 track))) (setf track (R (+ (* 1/2 Len) 1) track)) track))	(Defun Make-Pal (track) (Let ((Len (Length (First track)))) (setf track (S (+ (* 1 Len) −2) track)) (Loop While (> (Length (SECOND track)) 0) do (setf track (R 1 track)) (setf track (L 2 track))) (setf track (R (+ (* 1/2 Len) 1) track)) track))	Move two less than the cars to the siding. While there are more than zero cars on the siding, move one car to the right track, move two cars to the left track. Move one more than half the cars to the right track.
4. Two-loops palindrome	K-complexity: 40 words	K-complexity: 48 words
(defun two-loops (track) (Let* ((len (Length (First track))) (N-of-reps (* 1/2 len))) (Loop for I from 1 to N-of-reps do (setf track (S 1 track)) (setf track (R 1 track))) (Loop for I from 1 to N-of-reps do (setf track (L 1 track)) (setf track (R 1 track))) track))	(Defun Two-loops (track) (Loop While (> (Length (First track)) 1) do (setf track (S 1 track)) (setf track (R 1 track))) (Loop While (> (Length (SECOND track)) 0) do (setf track (L 1 track)) (setf track (R 1 track))) track)	While there are more than zero cars on the left track, move one car to the siding, move one car to the right track. While there are more than zero cars on the siding, move one car to the left track, move one car to the right track.
5. Parity sort	K-complexity: 50	K-complexity: 54 words
(Defun Parity (track) (Let* ((Len (Length (First track))) (N-of-reps (+ (* 1/2 Len) −1))) (setf track (R 1 track)) (Loop for I from 1 to N-of-reps do (setf track (S 1 track)) (setf track (R 1 track))) (setf track (L (+ (* 1/2 Len) −1) track)) (setf track (R (+ (* 1/2 Len) 0) track)) track))	(Defun Parity (track) (Let ((Len (Length (First track)))) (setf track (R 1 track)) (Loop While (> (Length (First track)) 1) do (setf track (S 1 track)) (setf track (R 1 track))) (setf track (L (+ (* 1/2 Len) −1) track)) (setf track (R (+ (* 1/2 Len) 0) track)) track))	Move one car to right track. While there are more than one car on left track move one car to siding, move one car to right track. Move one less than half the number of cars in the train to left track. Move half the number of cars in the train to right track.

Table A.2
Translation from the Italian of some children's protocols containing loops of actions for correct solutions. The table presents what the children said, and the corresponding moves that their remarks implied. It also identifies loops of moves.

Swap adjacent pairs (S3): HGFEDCBA has to be rearranged as GHEFCDAB	
'I move two cars from start to siding.'	HGFEDC[BA]–
'I move a car from siding to arrival.'	HGFEDC[A]B
'I move another car from siding to arrival.'	' HGFEDC[–]AB
'And this rule repeats for all the cars.' The assertion is a for-loop.	–[–]GHEFCDAB
Reversal (S6): HGFEDCBA has to be rearranged as ABCDEFGH	
'I send seven cars from start to siding.'	H[GFEDCBA]–
'Then I send one car from start to arrival.'	–[GFEDCBA]H
'I move one car from siding to arrival.'	–[FEDCBA]GH
'And this rule holds also for the other six cars.' The assertion is a for-loop.	–[–]ABCDEFGH
Make palindrome (S11): AABBCC has to be rearranged as ABCCBA	
'I move five cars from start to siding.'	A[ABBCC]–
'I move one car from siding to arrival.'	A[BBCC]A
'This move is repeated one time, that I move one car from siding to arrival.' The assertion is a for-loop.	A[BCC]BA
'Then I move one car from siding to start.'	AB[CC]BA
'I move two cars from siding to arrival.'	AB[–]CCBA
'I move two cars from start to arrival.'	–[–]ABCCBA
Two-loops palindrome (S23): DDCCBBAA has to be rearranged as ABCDDCBA	
'One car goes from start to arrival.'	DDCCBBA[–]A
'Then one car goes from start to siding.'	DDCCBB[A]A
'Then one car goes from start to arrival.'	DDCCB[A]BA
'And we repeat the same move for another couple.' The assertion is a for-loop.	DDCC[BA]BA
	DDC[BA]CBA
'Then we move two cars from start to arrival.'	DD[CBA]CBA
'Then, we move one car from siding to arrival for three times' . The assertion is a for-loop.	–[CBA]DDCBA
	–[–]ABCCBA
Parity sort (S26): HGFEDCBA has to be rearranged as HFDBGECA	
'I move one car from start to arrival.'	HGFEDCB[–]A
'I move one car from start to siding.'	HGFEDC[B]A
'Same thing for three times.' The assertion is a for-loop.	–[HFDB]GECA
Then I take the four cars from the siding and move them to the arrival.'	–[–]HFDBGECA

Appendix B

See Table B.1.

Table B.1
Typical configurations and errors in correspondence of the three main junctures of the programs. Errors in loops are indicated by a continuous line.

The first move in programs		
Example of error on the first move		
(S17) Swap adjacent pairs: FEDCBA has to be rearranged as EFCDAB		
'I move one car to arrival.'		FEDCB[–]A
Example of error on the second move		
(S16) Two-loops palindrome: CCBBAA has to be rearranged as ABCCBA		
'I move one car to arrival'		CCBBA[–]A
'I move one car to arrival'		CCBB[–]AA
Repetitions in the loop		
Example of error in same cars loop (L1 R1)		
(S28) Reversal: FEDCBA has to be rearranged as ABCDEF		
'I move five cars to siding'	S5	F[EDCBA]–
'I move one car from start to arrival'	R1	–[EDCBA]F
'I move one car from siding to arrival'	L1 R1	–[DCBA]EF
'I move five cars from siding to arrival'	L5 R5 (in place of L1 R1)	-impossible-
Example of error in different cars loop (R1 S1)		
(S17) Parity sort: FEDCBA has to be rearranged as FDBECA		
'I move one car from start to arrival. I move one car from start to siding' .	R1 S1	FEDC[B]A37
'I move one car from start to arrival. I move one car from start to arrival' .	R1 R1 (in place of R1 S1)	FE[B]DCA
The moves in post loop sequences		
Example of perseveration error (when there are cars at start after the loop)		
Make palindrome: AABBCCDD has to be rearranged as ABCDDCBA		
(S4) 'I move six cars from start to siding.'	S6	AA[BBCCDD]–
'I move one car to arrival and one car to start'	R1 L1	AB[BCCDD]A
'I move one car to arrival and one car to start'	R1 L1	AB[CCDD]BA
'I move one car to arrival and one car to start'	R1 L1 (in place of L2)	AC[CDD]BBA
Perseveration leads to errors		
Two loops palindrome: DDCCBBAA has to be rearranged as ABCDDCBA		
S(17) 'I move one car from start to arrival. I move one car from start to siding	R1 S1	DDCCBB[A]A
'I move one car from start to siding. I move one car from start to arrival'	R1 S1	DDCC[BA]BA
'I move one car from start to siding. I move one car from start to arrival'	R1 S1	DD[CBA]CBA
'I move one car from start to siding. I move one car from start to arrival'	R1 S1	–[DCBA]DCBA
Perseveration leads just to redundant moves, not to error		
Example of stop error (when there are no cars at start after the loop)		
Parity sort: HGFEDCBA has to be rearranged as HFDBGECA		
(S11) 'I move one car from start to arrival. I move one car from start to siding' .	R1 S1	–[HFDB]GECA
'These moves repeat for all the other cars.' (end)	R1 S1 3 times	HGFEDC[B]A38 - not completed

References

Aamodt-Leeper, G., Creswell, C., McGurk, R., & Skuse, D. H. (2001). Individual differences in cognitive planning on the tower of hanoi task: Neuropsychological maturity or measurement error? *Journal of Child Psychology & Psychiatry*, *42*, 551–556. http://dx.doi.org/10.1111/1469-7610.00749.

Aho, A. V. (2012). Computation and computational thinking. *The Computer Journal*, *55*, 832–835. http://dx.doi.org/10.1093/comjnl/bxs074[org/10.1093/comjnl/bxs074].

Bar-Hillel, Y. (1969). Colloquium on the role of formal languages. In *Foundations of language. Vol. 5* (pp. 256–284).

Biermann, A. W., Ballard, B. W., & Sigmon, A. H. (1983). An experimental study of natural language programming. *International Journal of Man-Machine Studies*, *18*, 71–87. http://dx.doi.org/10.1016/S0020-7373(83)80005-4.

Bjerre, A., & Dohn, N. B. (2018). Guided tinkering as a design for learning programming. In N. B. Dohn (Ed.), *Designing for learning in a networked world* (1st ed.). (pp. 177–196). London: Routledge, (Routledge Research in Education).

Bruckman, A., & Edwards, E. (1999). Should we leverage natural-language knowledge? An analysis or user errors in a natural-language-style programming language. In *Proceedings of the conference on human factors in computing systems* (pp. 207–214). Pittsburgh, PA: ACM Press.

Bucciarelli, M., Mackiewicz, R., Khemlani, S. S., & Johnson-Laird, P. N. (2016). Children's creation of programs: Simulations and gestures. *Journal of Cognitive Psychology*, *28*(3), 297–318. http://dx.doi.org/10.1080/20445911.2015.1134541.

Bucciarelli, M., Mackiewicz, R., Khemlani, S. S., & Johnson-Laird, P. N. (2018). Simulation in children's conscious recursive reasoning. *Memory & Cognition*, *46*, 1302–1314. http://dx.doi.org/10.3758/s13421-018-0838-0.

Cooper, J., & Weaver, K. D. (2003). *Gender and computers: understanding the digital divide*. Psychology Press.

Cornoldi, C., De Beni, R., & Fioritto, M. C. (2003). The assessment of self-regulation in college students with and without academic difficulties. In *Advances in learning and behavioral disabilities. Vol. 16* (pp. 231–242). Bingley: Emerald Group Publishing Limited, http://dx.doi.org/10.1016/S0735-004X(03)16009-0.

Dagiene, V., & Stupuriene, G. (2016). Informatics concepts and computational thinking in K-12 education: A Lithuanian perspective. *Journal of Information Processing*, *24*, 732–739, http://dx.doi.org/10.2197/ipsjjip.24.732.

Dicheva, D., & Close, J. (1996). Mental models of recursion. *Journal of Educational Computing Research*, *14*(1), 1–23. http://dx.doi.org/10.2190/AGG9-A5UD-DEK0-80EN.

Du, J., & Wimmer, H. (2019). Hour of code: A study of gender differences in computing. *Information Systems Education Journal*, *17*, 91–100.

Ehsan, H., & Cardella, M. E. (2017). Capturing the computational thinking of families with young children in out-of-school environments. In *Proceedings of the 2017 American society for engineering education annual conference and exposition. Vol. 12. Columbus, Ohio*. http://dx.doi.org/10.18260/1-2--28010.

Ianì, F. (2019). Embodied memories: reviewing the role of the body in memory processes. *Psychonomic Bulletin & Review*, *26*, 1747–1766. http://dx.doi.org/10.3758/s13423-019-01674-x.

Ianì, F., Limata, T., Bucciarelli, M., & Mazzoni, G. (2020). Children's kinematic false memories. *Journal of Cognitive Psychology*, *32*, 479–493. http://dx.doi.org/10.1080/20445911.2020.1796686.

Johnson-Laird, P. N. (1983). *Mental models: towards a cognitive science of language, inference, and consciousness*. Cambridge, MA: Cambridge University Press, Harvard University Press.

Johnson-Laird, P. N., Bucciarelli, M., Mackiewicz, & Khemlani, S. S. (2021). Recursion in programs, thought, and language. *Psychonomic Bulletin & Review*, http://dx.doi.org/10.3758/s13423-021-01977-y, in press.

Khemlani, S. S., Mackiewicz, R., Bucciarelli, M., & Johnson-Laird, P. N. (2013). Kinematic mental simulations in abduction and deduction. In *Proceedings of the national academy of sciences of the united states of America. Vol. 110* (pp. 16766–16771). http://dx.doi.org/10.1080/20445911.2015.1134541.

Kuhn, D. (2013). Reasoning. In P. D. Zelazo (Ed.), *The oxford handbook of developmental psychology* (pp. 744–764). Oxford: Oxford University Press.

Kurland, D. M., & Pea, R. D. (1985). Children's mental models of recursive logo programs. *Journal of Educational Computing Research*, *1*, 235–243. http://dx.doi.org/10.2190/JV9Y-5PD0-MX22-9J4Y.

Lane, H. C., & VanLehn, K. (2005). Teaching the tacit knowledge of programming to novices with natural language tutoring. *Computer Science Education*, *15*, 183–201. http://dx.doi.org/10.1080/08993400500224286.

Li, M., & Vitányi, P. (1997). *An introduction to kolmogorov complexity and its applications* (2nd ed.). New York: Springer.

Lockwood, J., & Mooney, A. (2018). Developing computational thinking test using bebras problems. CC-TEL/TACKLE@EC-TEL.

O'Neill, M., & Spector, L. (2020). Automatic programming: The open issue? *Genetic Programming and Evolvable Machines*, *21*(1), 251–262.

Pane, J. F., Ratanamahatana, C. A., & Myers, B. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, *54*, 237–264. http://dx.doi.org/10.1006/ijhc.2000.0410.

Pylyshyn, Z. (2003). Return of the mental image: are there really pictures in the brain? Trends. *Cognitive Science*, *7*, 113–118.

Romàn-Gonzàlez, M. (2015). Computational thinking test: design guidelines and content validation. In *7th Annual international conference on education and new learning technologies*, IATED, Barcelona, Spain. (pp. 2436–2444).

Romàn-Gonzàlez, M., Pérez-Gonzàles, J.-C., Moreno-León, J., & Robles, G. (2018). Can computational talent be detected? Predictive validity of the computational thinking test. *International Journal of Child-Computer Interaction*, *18*, 47–58. http://dx.doi.org/10.1016/j.ijcci.2018.06.004[org/10.1016/j.ijcci.2018.06.004].

Scirea, M., & Valente, A. (2020). Boardgames and computational thinking: How to identify games with potential to support CT in the classroom. In *FDG '20*, *International conference on the foundations of digital games* (p. 8). Bugibba, Malta, New York, NY, U.S.A.: ACM, http://dx.doi.org/10.1145/3402942.3409616, September (2020) 15-18.

Sharma, N., Chawla, V., & Ram, N. (2020). Comparison of machine learning programs for the automatic programming of computer numerical control machine. *International Journal of Data and Network Science*, *4*(1), 1–14.

Soloway, E., Bonar, J., & Ehrlich, K. (1989). Cognitive strategies and looping constructs: an empirical study. In E. Soloway, & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 191–207). Hillsdale, NJ: Lawrence Erlbaum Associates.

Sridhar, S., & Sanagavarapu, S. (2020). A Compiler-based approach for natural language to code conversion. In *2020 3rd International conference on computer and informatics engineering* (pp. 1–6).

Yu, J., & Roque, R. (2019). A review of computational toys and kits for young children. *International Journal of Child-Computer Interaction*, *21*, 17–36. http://dx.doi.org/10.1016/j.ijcci.2019.04.001.

Zapata Cáceres, M., Martín-Barroso, E., & Román-González, M. (2021). BCTt: Beginners computational thinking test. In *Proceedings of the raspberry pi foundation research seminar series*, *Understanding computing education. Vol. 1*. UK: Raspberry Pi Press.