Mental simulation and the construction of informal algorithms

Sangeet Khemlani[1] and Phil Johnson-Laird[2]
sunny.khemlani.ctr@nrl.navy.mil, phil@princeton.edu
[1]Naval Research Laboratory, Washington, DC 20375 USA
[2]Princeton University, Princeton NJ 08540 USA

Abstract

We describe two studies that show that when individuals who are not programmers create algorithms, they rely on mental simulations. Our studies concerned a railway domain in which carriages are rearranged – a simple environment but equivalent in computational power to a Turing machine. Participants successfully solved rearrangement problems (Experiment 1), and created algorithms to solve them (Experiment 2) and their performance corroborated the use of simulation. The participants tended to use loops and to prefer while-loops even though they are of greater computational power than for-loops. Their ability to create algorithms for abstract problems improved when they first had to create algorithms for more concrete problems. We devised a computer program that creates its own algorithms for rearrangement problems. It generates Lisp functions that operate on lists and creates descriptions of them in everyday language. The complexity of the resulting algorithms predicts participants' difficulty in devising them.

Keywords: algorithms, computer programming, creativity, deduction, problem-solving, reasoning

Introduction

A long controversy about human thinking is whether it depends on logic (Rips, 1994), probabilities (Oakford & Chater, 2007; Tenenbaum & Griffiths, 2001), or mental simulations (Craik, 1943; Johnson-Laird, 1983; Hegarty, 2004). Many inferences such as syllogistic deductions can be explained by mechanisms that depend on any of the three approaches (see Khemlani & Johnson-Laird, 2012, for a review). Indeed, few inferential tasks unequivocally depend on one approach. Computer programming may be such a task: it is readily explained by appealing to mental simulation (Bornat, Dehnadi, & Simon, 2008; Caspersen, Bennedsen, & Larsen, 2007; Kurland & Pea, 1985). To debug faulty code, programmers have to mentally simulate the algorithm to discover the situations in which the computer failed to produce the expected output. It is less apparent how mental rules of logic or probabilities could be used develop algorithms. Logic can be used to deduce the consequences of a program, but the creation of a program goes beyond logic (cf. Gulwani, 2010; Kitzelmann, Schmidt, Mühlpfordt, & Wysotzki, 2002). Probabilities hardly enter into the process, because computer programs are deterministic, and the language of the probability calculus is ill equipped to operate over the structures of programs. Mental simulation is therefore an appropriate framework with which to characterize the ability to create algorithms, and researchers can benefit from studying the simulations programmers use to solve tasks (Holt, Boehm-Davis, & Schultz, 1987).

Expert programming depends on more than just mental simulation, however. Programmers often have specialized knowledge of programming languages, of relevant software platforms and tools, and about computer science in general (Boehm-Davis & Ross, 1992). For that reason, many studies have tested the ability of novice programmers to write computer programs (see, e.g., Anderson, Pirolli, & Farrell, 1988). Few have investigated how those without any background in programming try to create algorithms. Miller (1974) pioneered such studies. He examined the way college students unfamiliar with computers wrote instructions for others to follow, and found that they tended *not* to use loops in their instructions, even though they could understand them (Miller, 1981). More recently, Pane and his colleagues carried out a study in which they presented non-programmers with static descriptions of an agent moving in a popular video game, PacMan, and the participants had to summarize how agents moved in general. They again preferred not to make use of loops, but when they did, they appeared to rely on while-loops (Pane et al., 2001).

Despite these results, there exists no psychological theory of how non-programmers construct algorithms. To develop such a theory, and to study algorithmic creativity in non-programmers, we designed a novel problem-solving task environment in which reasoners have to sort the order of a list in various ways. We introduce the environment below, and then explain how individuals build kinematic mental models to construct algorithms for their solutions. We then describe two experiments that show that reasoners intuitively understand the environment (Experiment 1) and that they can mentally create algorithms for the problems in the environment (Experiment 2).

Rearrangement problems and the railway environment

We studied how individuals who have never learned computer programming create algorithms in everyday language. For problems that they readily understood, we used the railway environment shown in Figure 1. The environment consists of a railway track and a siding. It is an analog of a finite-state device with two stacks – the left track (a) holds the input and also acts as a stack, the siding (b) acts as another stack, and the right track (c) holds the final output. Participants' task is to move the cars from the left track to the right track into a specific order. Cars can move only from the siding to and from the left track, and from left track to right track. Multiple cars can be moved at once, i.e., any move of a selected car applies to all cars in front of it. For example, in Figure 1, if you moved the E car

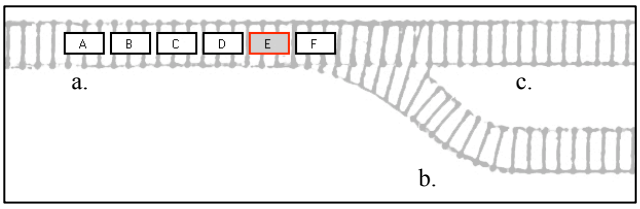

Figure 1. The railway domain with an example of an initial configuration in which a set of cars is on the left side (a) of the track, the siding (b) can hold one or more cars while other cars are moved to the right side of the track (c).

to the right track, then the F car would move along in front of it. To restrict the environment to a single stack, cars could move from the siding only to the output on the right track. In summary, only three sorts of move are possible in the railway environment:

R: one or more cars moved from left track to right track.
S: one or more cars moved from left track to siding.
L: one or more cars moved from the siding to left track.

One constraint is that cars can be neither removed nor added to trains in our rearrangement problems – if they could be, then the railway environment would be equivalent to a universal Turing machine power.

Experiment 1 below investigated all 24 possible re-arrangements of four cars, and examined whether the participants perseverated, i.e., made one or more unnecessary moves. They can use a simple variant of "means-ends" analysis in which they work backwards from the required goal, invoking operations relevant to reducing the difference between the current state and the goal (e.g., Newell & Simon, 1972; Newell, 1990). For rearrangement problems, they need only envisage each successive car in the goal. Suppose, for instance, they have to re-arrange the order ABCD into ACBD. The starting state is: ABCD[ ], where the square brackets denote the contents of the siding, which is empty at the start. Their immediate goal is to get D to the far end of the right track: [ ] . . . D. So, they move D from left to right track: ABC[ ]D. The next partial goal is to get B to the right track, and so they need to move C out of the way onto the siding: AB[C]D. Now, they can move B to the right: A[C]BD. They move C off the stack: AC[ ]BD. The next move is intriguing. They should move both A and C together from left to right track. But, if reasoners perseverate, they may move only C to the right track. Their solution won't be minimal, because they then have to make a separate move of A to right track.

We investigated how reasoners solve single instances of such problems, but our primary goal was to understand the processes and representations non-programmers use to create algorithms. In the following section, we explain how kinematic mental models can be used to construct algorithms, and illustrate the predictions that the model-based theory makes.

A model-based theory of algorithmic creativity

How do naïve individuals create informal algorithms? We hypothesize that individuals simulate solutions to problems, where a simulation consists of a sequence of kinematic mental models representing states of the world, real or imaginary, and the sequence itself represents a logical or temporal order of the states (Johnson-Laird, 1983, Ch. 15). Reasoners use such simulations to carry out three separate steps to create an algorithm: 1) they solve at least two different instances of a rearrangement problem using a kinematic sequence of moves; 2) they scan the kinematic sequences to abduce a pattern; 3) they translate the pattern into a verbal description. We address the three steps in turn.

Step 1: Problem-solving as simulation. The first step is to solve two different instances of a rearrangement problem. Otherwise, re-arrangements are ambiguous. At any point in the simulation, only a single move is made, and so to reverse, say, four carriages, reasoners can begin by envisaging the transformation from the start state:

ABCDEF[ ] → [ ] . . . A

This partial goal calls for a move of five cars onto the siding, A[BCDEF], so A can be moved to right track, [BCDEF]A. The next partial goal is to get B to right track, and so it should be moved to left track, B[CDEF]A, and over to right track, [CDEF]BA. A repeated loop of these two operations moves each car in turn off the siding and to right track, and solves the problem.

Two variables should affect performance in the solution of rearrangement problems: the number of moves and the number of their operands. Obviously, the greater the number of moves, the more difficult a problem should be – the only sort of theory that would not make this prediction would be one that made no appeal to simulation. A more subtle prediction concerns the number of operands. In a reversal problem, such as the one above, each move after the first has an operand of one car. We can contrast this case with the solution of a palindrome problem, such as:

ABCCBA[ ] → [ ]AABBCC

There are three cars, BCC, on the left that match the goal, but they are blocked, and so to solve the problem, the blocking cars are moved onto the siding: ABCC[BA]. The three cars on the left are moved to the right: A[BA]BCC. One car on the siding matches the goal, and so it is moved to the left: AB[A]BCC. Two cars on the left match the goal, and so they are moved to the right: [A]ABBCC. The car on the stack matches the goal, and so it is moved to the left and then over to the right, and the problem is solved. Its minimal solution required a total of 10 cars to be moved in 6 moves. This solution has a mean number of operands per move greater than that for the reversal problems, and so the theory predicts that the palindrome problems should be more difficult than reversal problems of the same number of

moves. And individuals may make an unnecessary move in their solution of the problem, i.e., they may fail to solve the problem parsimoniously. Number of operands has a family resemblance to "relational complexity", which concerns the number of arguments in a relation, and which affects problem difficulty (Halford, Wilson, & Phillips, 1998). However, the number of operands concerns, not the number of arguments of an operator, but whether the value of a single argument is one or more entities.

Step 2: Pattern abstraction and abduction. The second step in creating an algorithm is to recover the structure of the solutions – the loop they contain, and any operations before or after it. Consider the moves to reverse trains of four and five cars, respectively:

(S3 R1 L1 R1 L1 R1 L1 R1)
(S4 R1 L1 R1 L1 R1 L1 R1 L1 R1)

where 'S3' means *move three cars from left track to the Siding*, 'R1' means *move one car from left track to Right track*, and 'L1' means *move one car from the siding to Left track*. The loop of operations is (R1 L1). But, how many times should it be iterated? There are two ways to find the answer. The simpler is to observe the conditions in the simulation when the loop ceases, respectively:

D[ ]CBA
E[ ]DCBA

In both cases, the siding is empty, and so this condition determines that a while-loop should continue until the siding is empty. The alternative answer depends on computing the number of times that a for-loop should be executed, and it calls for the solution of a pair of simultaneous linear equations to obtain the values of *a* and *b* in:

number-of-iterations = a * train-length + b.

Step 3: Conversion to natural language. The third and final step is to map the structure of the solution into a description. A *general* algorithm for reversing the order of cars applies to trains of any length. Hence, it needs to describe a loop of moves. When reasoners convert the algorithm to a natural language description, their responses should yield the condition in which the loop stops (an indication that they've constructed while-loop) or else reflect the number of times for which the loop should be executed (an indication that they've constructed a for-loop). The solution of simultaneous equations calls for more than just simulation, whereas the halting conditions of a loop can be observed in a simulation, and so the theory predicts that correct responses should tend to use while-loops more often than for-loops.

We have implemented all three steps in a computer program that discovers and outputs algorithms to solve any re-arrangement problem that depends on a single loop. It outputs a for-loop, a while-loop, and a translation of the while-loop into informal English (see Appendix). Each of these algorithms solves any instance of the relevant class of rearrangements.

Experiment 1 tested whether solutions to rearrangements depend on the number of moves and the number of operands. Experiment 2 tested whether reasoners use simulation to construct algorithms, and therefore formulate while loops, and whether the theory predicts the relative difficulty of different sorts of problem.

Experiment 1

Experiment 1 tested the effects of number of moves and number of operands on the solution of simple rearrangement problems in the railway environment. The problems were simple and called for the rearrangement of only four cars. Hence, our interest was in whether the participants could solve the problems without making redundant moves. The participants had to solve all the 24 possible rearrangements of trains containing four cars. Their minimal solutions call for various numbers of moves (1, 4, 5, 6, 7, or 8), and as a consequence the theory predicts an increasing trend in redundant moves for these problems. The total numbers of operands in minimal solutions was (4, 6, 8, 10, or 12), and as a consequence there should be an increasing trend in redundant moves. Because these two variables are only partially correlated, we were able to examine their effects independently (see Table 1 below).

Method

Participants. Twenty undergraduate students at Princeton University served as participants, and none had had any prior training in logic or computer science.

Design and procedure. Participants acted as their own controls and carried out all 24 problems, which were presented in a different random order to each of them. When they had completed the experiment, they carried out two of the problems again, but they had to think aloud as they did so. They were tested individually, and carried out the experiment on a PC running LispWorks 4.4. They interacted with the system using the mouse and the keyboard of the computer. They were shown a three-minute instructional video that guided them through the elements of the railway environment, and that presented the instructions. The key instruction stated that they should try to solve each problem with as few moves as possible.

Results and discussion

Non-programmers were able to solve rearrangement problems with ease: they produced very few incorrect solutions. Table 1 presents the participants' mean numbers of moves to solve the problems depending on the minimum number of moves and the total number of operands. We dropped the two extreme problems from the statistical

# of moves in a minimal solution	Total number of operands (cars) moved in minimal solutions					Mean # of actual moves
	4	6	8	10	12	
1	1.0					1.0
4		4.3	4.7	4.6		4.5
5		5.5	5.2			5.4
6			6.5	6.6		6.6
7			7.9			7.9
8			8.3	8.5	8.6	8.4
Mean # of actual moves	1.0	4.9	6.5	6.9	8.6	

Table 1. The mean numbers of moves in Experiment 1 in rearrangement problems as a function of the total number of moves in their minimal solutions and the total number of operands (cars) to be moved.

analysis so that they would not bias the results, i.e., the problem that required only one move to solution, and the problem that had a total of 12 operands. Given that the participants solved the problems, it is hardly surprising that the mean number of the participants' moves increased with the minimal number of moves required to solve a problem (Page's trend test, $L = 1809.5$, $z = 8.47$, $p < .0001$). But, the results also showed that their mean number of moves also increased with the number of operands (Page's trend test, $L = 276$, $z = 5.69$, $p < .0001$). In other words, the participants tended to fail to find minimal solutions, and as the mean number of operands increased so the number of their moves increased, independently of the total number of moves in a minimal solution. (For brevity, we spare readers the latency results, but their patterns corroborated both of these effects.) There was a reliable tendency for the participants to make redundant moves. Every participant made at least one redundant move (Binomial, $p = .5^{20}$).

In summary, the experiment shows that naive individuals can solve simple rearrangements. It corroborated the prediction that the number of moves affected the difficulty of the problem, and thereby supported simulation-based accounts. Likewise, it corroborated the prediction unique to the model-based theory that the number of operands should affect the difficulty of a problem. The following experiment tested whether non-programmers could formulate general solutions for rearrangement problems.

Experiment 2

In Experiment 2, the participants had to formulate algorithms to solve three sorts of rearrangement: *reversals*, such as ABCDEFGH becomes HGFEDCBA; *palindromes*, such as ABCDDCBA becomes AABBCCDD; and *parity sorts*, such as ABCDEFGH becomes ACEGBDFH. Participants had to construct the algorithms in their mind's eye with no access to the railway environment. They were familiar with the environment, because they had just solved five practice problems on it, but these problems were simple rearrangements that differed from the problems in the experiment proper. They were then shown the inputs and outputs for each of the problems, and they had to write down algorithms for solving them. They did so for *fixed-length* problems in which trains of eight cars had to be rearranged, and *indefinite-length* problems in which trains of any number of cars had to be rearranged. The fixed-length problems should be easier than indefinite-length problems, because only the former can be solved without loops. Likewise, complexity and number of operands predict a trend in difficulty over the three sorts of general rearrangements: reversals should be easier than palindromes, which in turn should be easier than parity sorts. The latter should be the hardest to solve because they call for an extra operation in their algorithm (see the Appendix).

Method

Design and materials. The participants acted as their own controls and carried out six problems: the three sorts of rearrangement as both fixed-length problems of eight cars and indefinite-length problems of any number of cars. The session began with five practice problems akin to those in Experiment 1, which the participants merely had to solve by interacting with the railway system. These problems were unrelated to the experimental problems: each of them had a train of 6 cars, and a solution depending on 8 moves. The experiment proper followed, and the participants' task was to type out a procedure that would solve each problem, but they were not allowed to interact with the railway environment. They carried out two blocks of trials, one of the definite problems and one of the indefinite problems, presented in a counterbalanced order to two groups of participants. The order of the three sorts of rearrangement was randomized for each participant within the blocks. For the indefinite-length problems, the participants were told that a car containing an ellipsis stood in place for any number of cars that had the same pattern.

Participants and procedure. Twenty students from the same population as before took part in the experiment. They watched an instructional video and were told how to interpret the car containing an ellipsis. They then solved the practice problems using the same procedure as before. In the experiment proper, the participants were told to write a description of a procedure for solving each of the experimental problems as efficiently as possible. They were free to use their own words in any way that they wanted, but they no longer were allowed to manipulate the cars in the railway environment.

Results and discussion

Two independent raters scored the correctness of the algorithms and whether they contained a while-loop, a for loop, or no loop whatsoever (see Appendix for examples of correct responses). Inter-rater reliability was high for judgments of correctness (Cohen's $\kappa = .82$) and the sorts of loops that participants devised ($\kappa = .73$). A third independent rater resolved the disagreements. Performance with the fixed-length problems was at ceiling (90% correct)

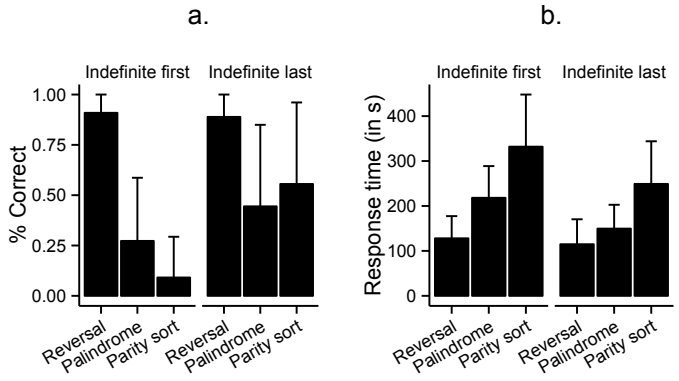

Figure 2. The percentages of correct algorithms (panel a) and the response times in s (panel b) for the indefinite-length problems as a function of the sort of rearrangement, and whether they occurred in the first or second block of trials.

and much better than the indefinite-length problems (52% correct; Wilcoxon test, $z = 3.5$, $p = .0004$; Cliff's $\delta = .64$). Figure 2 accordingly shows only the performance for the indefinite-length problems, and the Appendix provides examples of participants' correct algorithms. The three sorts of rearrangement yielded the predicted trend in accuracy and in the time to respond (see Appendix; Page's trend tests, $zs > 3.08$, $ps < .002$). Likewise, the participants used many more while-loops (74% of correct solutions) than for-loops (26% of correct solutions) for indefinite-length problems. The use of while-loops correlated with accuracy ($r = .32$, $p < .0005$), whereas the use of for-loops did not ($r = .14$, $p = .10$). The differences in ability were striking: the best participant created a correct algorithm for every problem, whereas the worst did so for only a third of the fixed-length problems and for none of the indefinite-length problems.

General Discussion

The ability to create algorithms might seem to be a case of competence in pure mathematics with little relation to everyday life. Problems in rearranging cars in toy trains may similarly seem remote from the exigencies of daily life. However, algorithmic thinking is regularly called for, e.g., in laying place settings on a table, in determining kinship relations, in following a recipe or a set of instructions. Other sorts of algorithmic thinking are needed to determine the consequences of knitting patterns, instructions for kits, maintenance manuals, and, above all, algorithms in computer programs.

Algorithmic thinking is easier when you can manipulate an external environment and solve a problem using only partial means-ends analysis, i.e., you can use the railway environment and solve a rearrangement of the cars in a train, one car at a time (Experiment 1). But suppose that your task is to devise an algorithm for the general problem of sorting cars in this way – so that cars in odd-numbered positions precede cars in even-numbered positions. The algorithm for this task is not obvious. According to the present theory, the way that you carry it out is to make another simulation so that you can figure out what is going on. You should then notice that there is a loop of two operations (move one car to the right, and then one car onto the siding) that has to be repeated while more than two cars remain on the left track. It follows that while-loops should occur more often than for-loops in putative algorithms, because it is easier to envisage halting conditions for while-loops from simulations than to use them to compute the number of iterations for a for-loop. The difficulty of the task also depends on the Kolmogorov complexity of the program, as indexed in the number of its instructions (in Lisp or in everyday language), and on the number of operands (Experiment 2).

Computer scientists often complain about the lack of any valid test of the likely ability of naive individuals as computer programmers (e.g., Bornat, Dehnadi, & Simon, 2008). The rearrangement problems in our experiments may provide the basis for such a test. At the very least, we now know that individuals differ reliably in their ability both to solve problems in the railway domain (Experiment 1), and to formulate informal algorithms for their solutions (Experiment 2). The question remains as to whether such tasks are reliable predictors of ability. Mathematicians, logicians, and computer programmers, learn to reason about the repeated loops of operations that are needed in recursive functions. Previous studies have examined how novice programmers cope with such reasoning in trying to specify algorithms in a programming language (see, e.g., Anderson & Jeffries, 1985). Our studies have shown that naive individuals with no training in computer programming are able to make simulation-based deductions, to solve rearrangement problems, and even to abduce informal algorithms for their general solution.

The evidence we have reported corroborated the theory based on mental models. To the best of our knowledge, no other theory of naïve algorithmic creativity exists. But, a theory could be developed in principle from an axiomatization of the domain in first-order logic (see, e.g., McCarthy & Hayes, 1969; McCarthy, 1986; Rips, 1994). A typical axiom would capture the effects of a move, e.g.:

> For any x, y, if x is a car & y is a train & z is a train & y is on right track & z is on left track & x is at the front of y & R 1 is carried out then x is at back of z & not (x is at front of y).

No one has proposed such an account, and so it is not yet possible to pit it against the model-based theory. But, we cannot rule it out, and remark only that the approach runs into difficulties. Our participants' think-aloud protocols raise problems for it, because they report moving cars around in a mental simulation of the railway environment. Likewise, their reliance on simulations predicts their use of while-loops in algorithms, because simulations yield the halting conditions for while-loops more readily than the number of iterations for for-loops. These results seem difficult, if not impossible, to explain without recourse to the use of mental simulations.

Acknowledgements

This research was supported by a National Science Foundation Graduate Research Fellowship to SSK and by NSF Grant No. SES 0844851 to PJL to study deductive and probabilistic reasoning. We are grateful to Monica Bucciarelli, Sam Glucksberg, Adele Goldberg, Geoffrey Goodwin, Louis Lee, David Lobina, Max Lotstein, Robert Mackiewicz, Paula Rubio, and Carlos Santamaria, for their helpful comments and criticisms.

References

Anderson, J.R., & Jeffries, R. (1985). Novice Lisp Errors: Undetected losses of information from working memory. *Human-Computer Interaction*, *1*, 107-131.

Anderson, J. R., Pirolli, P., & Farrell, R. (1988). Learning to program recursive functions. In M. Chi, R. Glaser, & M. Farr (Eds.), *The nature of expertise* (pp. 153-183). Hillsdale, NJ: Erlbaum.

Boehm-Davis, D., & Ross, L. (1992). Program design methodologies and the software development process. *International Journal of Man-Machine Studies, 36,* 1-19.

Bornat, R., Dehnadi, S., & Simon (2008). Mental models, consistency and programming aptitude. *Proceedings of the Tenth conference on Australasian Computing Education Conference*, *10*, 53–61.

Caspersen, M.E., Bennedsen, J., & Larsen, K.D. (2007). Mental models and programming aptitude. *ACM SIGCSE Bulletin, 39*.

Corballis, M. (2011). *The recursive mind*. Princeton: Princeton University Press.

Craik, K. (1943). *The Nature of Explanation.* Cambridge, UK: Cambridge University Press.

Gulwani, S. (2010). Dimensions in program synthesis. In Proceedings of the 12th International ACM SIGPLAN Conference. Hagenberg, Austria.

Halford, G.S., Wilson, W.H., & Phillips, S. (1998). Processing capacity defined by relational complexity: Implications for comparative, developmental, and cognitive psychology. *Behavioral and Brain Sciences*, *21*, 803-865.

Hegarty, M. (2004). Mechanical reasoning as mental simulation. *Trends in Cognitive Sciences*, *8*, 280-285.

Holt, R.W., Boehm-Davis, D., & Schultz, A. (1987). Mental representations of programs for students and professional programmers. In *Empirical Studies of Programmers: Second Workshop* (pp. 33-46). Ablex Publishing Corp.

Johnson-Laird, P.N. (1983). *Mental models*. Cambridge: Cambridge University Press.

Khemlani, S., & Johnson-Laird, P.N. (2012). Theories of the syllogism: A meta-analysis. *Psychological Bulletin*, *138*.

Kitzelmann, E., Schmidt, U., Mühlpfordt, M., & Wysotzki, F. (2002). Inductive synthesis of functional programs. In Calmet, J., Benhamou, B., et al. (Eds.) *Artificial Intelligence, Automated Reasoning, and Symbolic Computation*. New York: Springer.

Kurland, D. M., & Pea, R. D. (1985). Children's mental models of recursive LOGO programs. *Journal of Educational Computing Research*, *1*, 235-244.

Miller, L. (1974). Programming by non-programmers. *International Journal of Man-Machine Studies*, *6*, 237-260.

Miller, L. (1981). National language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, *20*, 184-215.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Newell, A., & Simon, H.A. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice-Hall.

Oaksford, M., & Chater, N. (2007). *Bayesian rationality*. Oxford: Oxford University Press.

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.

Rips, L.J. (1994). *The psychology of proof.* Cambridge, MA: MIT Press.

Tenenbaum, J. B., & Griffiths, T. L. (2001). Generalization, similarity, and Bayesian inference. *Behavioral and Brain Sciences, 24*.

Appendix. Natural language solutions (as outputted by the computer program for abducing them) to three sorts of general problem: reversals, palindromes, and parity sorts, and examples of correct algorithms created by participants; and the percentage of participants' algorithms that correctly solved the given problems in Experiment 2.

Problem	Automatically generated algorithms	Examples of correct algorithms	% Correct
Reversal	1 Move one less than the cars to siding. 2 While there are > zero cars on siding 3 ...move one car to right track 4 ...move one car to left track. 5 Move one car to right track.	"i'll move everything in the side track. then i'll move each letter back onto the left track and then to the right track." (Participant 14)	90%
Palindrome	1 Move one less than half the cars to siding. 2 While there are > two cars on left track 3 ...move two cars to right track 4 ...move one car to left track. 5 Move two cars to right track	"step1: cut the train into half, move the right half to siding step2: for both half trains on the left and siding track, move a pair of carts of the same letter to the right. Continue doing so until all the carts are on the right track." (Participant 1)	68%
Parity sort	1 While there are > two cars on left track 2 ...move one car to right track 3 ...move one car to siding. 4 Move one car to right track. 5 Move one less than half the cars to left track 6 Move half the cars to right track	"Move the rightmost car to the right track, and move the next car to the side track. Continue alternating between right track and side track until the left track is empty. Then move all cars from the side track to the left track, and then to the right track." (Participant 7)	55%