Psychonomic Bulletin & Review
https://doi.org/10.3758/s13423-021-01977-y

THEORETICAL REVIEW

Recursion in programs, thought, and language

P. N. Johnson-Laird[1,2] · Monica Bucciarelli[3,4] · Robert Mackiewicz[5] · Sangeet S. Khemlani[6]

Accepted: 25 June 2021

Abstract

This article presents a theory of recursion in thinking and language. In the logic of computability, a function maps one or more sets to another, and it can have a recursive definition that is *semi*-circular, i.e., referring in part to the function itself. Any function that is computable – and many are not – can be computed in an infinite number of distinct programs. Some of these programs are *semi*-circular too, but they needn't be, because repeated loops of instructions can compute any recursive function. Our theory aims to explain how naive individuals devise informal programs in natural language, and is itself implemented in a computer program that creates programs. Participants in our experiments spontaneously simulate loops of instructions in kinematic mental models. They rely on such loops to compute recursive functions for rearranging the order of cars in trains on a track with a siding. Kolmogorov complexity predicts the relative difficulty of abducing such programs – for easy rearrangements, such as reversing the order of the cars, to difficult ones, such as splitting a train in two and interleaving the two resulting halves (equivalent to a faro shuffle). This rearrangement uses both the siding and part of the track as working memories, shuffling cars between them, and so it relies on the power of a linear-bounded computer. Linguistic evidence implies that this power is more than necessary to compose the meanings of sentences in natural language from those of their grammatical constituents.

Keywords Computational power · Recursion · Informal programs · Grammar · Mental models · Working memory

> "General rules for programming have been discovered. Most of them have been used in Kansas City freight yards for some time" – Lehmer (1949, p.142)

The concept of recursion is important but often misunderstood. It can frighten the uninitiated, and the initiated sometimes increase its mystery. They write that recursion is a form of computation that embeds elements, or structures, within those of the same kind. That's true, but only part of the truth, because recursion can be simpler in many of its uses. And so we aim to demystify the concept and get to its core.

An intimate relation exists between recursion and programs, where a program is a finite set of instructions for carrying out a computation that ends with an output. Formal programs are written in a language that can be transformed into a set of instructions that a computer can execute. Informal programs are in natural language and commonplace – in recipes, knitting patterns, and instructions for assembling kits. The competence to create and to comprehend them underlies the ability to learn how to write computer programs. Our experiments tested naive individuals, where "naive" means only that they knew nothing about programming, and the tasks were to devise informal programs or to deduce their consequences. They formulated them in natural language, and its basis is said to be recursion (see, e.g., Berwick & Chomsky, 2016).

Public Significance Statement This article has three principal goals: To describe recursive functions from the theory of computability, to explain how naive individuals use kinematic mental models to create informal programs to compute these functions, and to show that the computational power they need is similar to the power needed for using natural language.

✉ P. N. Johnson-Laird
phil@princeton.edu

[1] Department of Psychology, Princeton University, Princeton, NJ 08544, USA

[2] Department of Psychology, New York University, New York, NY, USA

[3] Department of Psychology, Università di Torino, Turin, Italy

[4] Centro di Logica, Linguaggio, e Cognizione, Università di Torino, Turin, Italy

[5] Department of Psychology, SWPS University of Social Sciences and Humanities, Warsaw, Poland

[6] Navy Center for Applied Research in Artificial Intelligence, Naval Research Lab, Washington, DC, USA

Published online: 15 December 2021

Recursion means different things in different disciplines. It even means different things in the same discipline. Biologists use the term to refer to self-referential formulas, for example, for simulating the growth of plants (Prusinkiewicz & Lindenmayer, 1990, p. 15). Psychologists use it to refer to thinking about thinking, mental time travel, and children's theory of mind (Corballis, 2011). They use it to refer to self-similar fractal shapes, which individuals can learn from just a few examples (Lake & Piantadosi, 2020), and to intentional acts, where part of such an act is the knowledge that it is intentional (Johnson-Laird, 2006, Ch. 4; Vicari & Adenzato, 2014). To semanticists, recursion refers to certain concepts in natural language. The concept of *owning*, for example, requires a recursive definition: the transfer of ownership transfers the right to transfer ownership (Miller & Johnson-Laird, 1976, p. 560). Likewise, the wholly circular definition of an optimist as "anyone who believes that optimists exist" seems plausible. Two of us believed George W. Bush when he said on TV that he was an optimist, and so we're optimists. Now, you believe that optimists exist, and so you're an optimist too. Soon, everyone is an optimist – even avowed pessimists – and so the definition goes too far (Johnson-Laird, 2006, Ch. 11). It can be hard to grasp the consequences of recursion – a point to which we return below in the context of reasoning.

Linguists use "recursion" to refer to grammatical rules, such as: noun-phrase → noun-phrase and noun-phrase. The arrow specifies that a noun-phrase can consist of a conjunction of two noun-phrases formed with "and," so a grammatical constituent contains embedded within it constituents of the same sort (Pinker & Jackendoff, 2005). Linguists have also argued that recursion is the central component of natural language in which an operation of *merge* constructs complex structures out of concepts (e.g., Berwick & Chomsky, 2016; Hauser et al., 2002). One recent study showed that adults and 3- to 5-year-old children can generate self-embedded sequences of symbols more often than chance, and after additional exposure to examples, monkeys can too (Ferrigno et al., 2020). Computer programmers often use "recursive" to refer to a program for computing a function that contains at least one clause that is circular and one that is not. We refer to such programs as *semi*-circular: complete circles are vicious because as in our definition of "optimist" they go on and on forever, but *semi*-circular programs can be virtuous if they contain at least one other clause that ensures that they halt.

What, if anything, do all these different uses of "recursion" have in common? The answer is that they derive from the theory of recursive functions (aka the theory of computability, see Rogers, 1967). It is the branch of logic that presaged the invention of the programmable digital computer. This basis, as we illustrate, is not just self-embedding. Our article accordingly aims to answer three main questions:

1. What is recursion?
2. How could naive individuals create informal programs in natural language to compute recursive functions?
3. How much computational power does informal programming need, and how does it relate to the power needed for natural language?

Previous studies have addressed these questions in part, but they have not led to a general psychological theory of recursive thinking. And questions about recursion in natural language have led to controversy. The present article aims to answer the three questions and to yield a corroborated theory of recursive thinking and the computational power of the human brain on which it depends.

The article begins with an unusual computer that is simple enough for our studies of informal programming, whose participants include children, but that is powerful enough to illustrate the computation of recursive functions. This computer consists of a railway with one train, a straight track from left to right, and a siding. It serves as a computer that participants control. The article then uses this computer to illustrate the recursive definitions of functions, programs to compute their values, and the power that computers need to carry out these programs. Next, it presents a theory of informal programming based on kinematic mental models that simulate sequences of events, such as the movements of cars on the railway. This theory is implemented in a computer program, mAbducer, that can create its own programs for the railway computer, both formal and informal. The program is written in Common Lisp, and its source code is at: https://www.modeltheory.org/models/. To corroborate the theory, the article reviews evidence from studies of naive adults and children. It also considers linguistic evidence for the computational power needed to understand natural language. Finally, it assesses potential shortcomings in our studies and draws psychological conclusions about recursive thinking.

The railway computer

An informal program is a description in natural language of how to carry out a particular computation. We devised a simple computer to study whether our participants could create informal programs. As the epigraph to this paper intimates, it is a railway, but it can be transformed into the most powerful computer, a universal Turing machine (see, e.g., Davis, 1958). Figure 1 shows how the railway appears on a computer screen. On its left track is a train of six cars, which can each move in either direction, and which push any cars in front of them. A human user controls the computer, and can command, say, that car b in Fig. 1 moves to the siding – it pushes car a onto the siding too. Figure 2 shows an actual toy railway of the same sort, which children used in our studies of their informal

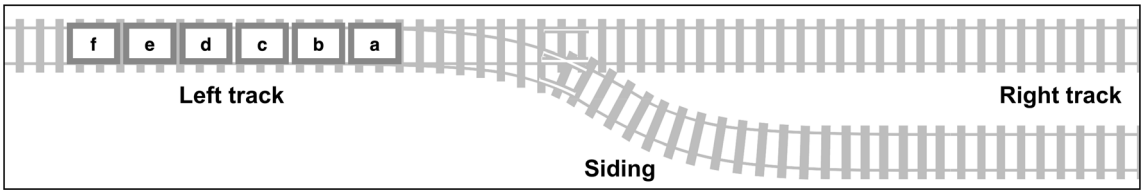

Fig. 1 The railway track as a computer in an initial state: A train, whose cars are labeled with the letters f–a is on left track. Each car can move, pushing any cars in front of it, from left track to the siding, from left track to right track, and from the siding back to left track. Hence, the siding can be used as an intermediate place to store cars to rearrange their order as they arrive on right track

programming. The railway is similar to one that Knuth (1997, p. 240) described, but ours can compute functions that his railway cannot.

The tasks we investigated concern rearrangements of the order of the cars in a train. At the start, the train is on the left track (as in Figs. 1 and 2) and the goal is to rearrange its cars into a new order on the right track. Only three sorts of move are allowed:

R: move one or more cars from left track to **R**ight track.
S: move one or more cars from left track to **S**iding,
L: move one or more cars from siding back to **L**eft track.

Once cars get to right track they must remain there, because no move allows them to return to left track. Likewise, cars can move from the siding only to left track, because no single move takes a car from the siding to right track. A move of two cars to the siding is described as: S2, where "S" denotes a move to siding and "2" denotes the number of cars in the move. In rearrangements, the three sorts of move make no reference to the labels on cars, which are there only to help individuals to remember the locations of cars.

Our empirical studies concern rearrangements that are simple enough for children to understand. They are permutations of the order of cars in a train (Bóna, 2012, p. 1), which depend solely on the positions of cars in the train. They apply to any finite number of cars. Some, however, may be constrained, say, to any odd number of cars, such as those concerning palindromes with a central car. The mathematical properties of rearrangements applying to any number of items (cars) have seldom been studied (Miklós Bóna, personal communication, 2 May 2016; cf. Bóna, 2019). There are infinitely many different rearrangements – just as there are infinitely many sentences in natural language – and they differ in how much power a computer needs to carry them out, and power, as we explain, concerns the nature of the computer's memory for the intermediate results of computations. The railway computer has enough power to cope with any rearrangement.

The fundamentals of computation

Functions

We can now explain recursion, and we begin with a fundamental concept. A *function* at its simplest is a mapping from one set to another, and for any member of the input set, the independent variable, there is at most one member of the output set, the dependent variable. Some functions are more general and have an input of several independent variables from different sets, for example, the function that determines how much income tax you should pay. Indeed, the members of sets, and therefore functions of them, can refer to entities of any sort: numbers, words, cars, trains, railways, and even functions themselves. A relation such as *equals* refers to members of two sets, and it can be treated as a function from the two sets to an output of *true* or *false*, depending on whether or not it holds between its two inputs. One of the railway's basic functions is L: it takes as input the number of cars to be moved and the current state of the railway, and its output is the resulting state of the railway. This function seems easy to understand. But, what happens, say, if L has an input of one

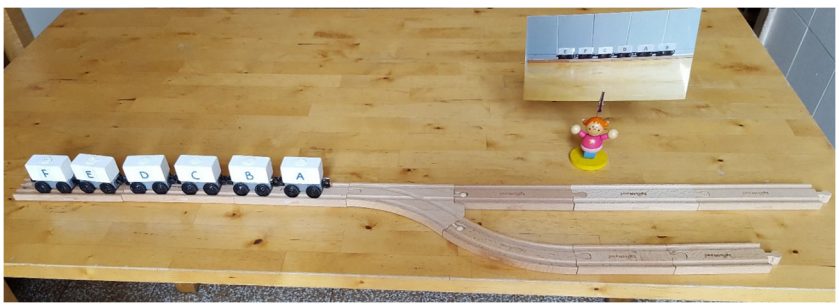

Fig. 2 A toy railway in its initial state, with a train of six cars, *f–a,* and a picture above the doll's head of a required rearrangement of the cars, used in the experiments with children

car and the railway that has no cars on the siding? Functions can be partial and have no output value for certain inputs, for example, division by zero. But, for the railway, it is convenient for L in this case to map to an output of an unchanged railway. We reiterate that the main constraint on functions is to deliver at most a single output for a given input, and that despite having inputs and outputs, functions are descriptions of a mapping: they don't do anything.

A rearrangement is a function too. It maps a train of cars in a particular order on left track to a new order on right track. Here are instances of the same rearrangement for two trains of different lengths, i.e., numbers of cars, and readers might try to understand what it does. The arrows show the mappings from input to output, and we have used a bold font for the cars in the first half of the train to help readers to see what is going on:

abcdef ⇒ **a**d**b**e**c**f

abcdefgh ⇒ **a**e**b**f**c**g**d**h

In fact, the function interleaves the cars in the second half of the train within those in the first half of the train. If you have spent any time in casinos, you may know that the same interleaving of a deck of cars is known as a faro shuffle (or riffle shuffle), with its origin in the eponymous card game. It is a rearrangement with some striking mathematical properties that relate, for example, to Fourier analysis (see Diaconis et al., 1983). Later, we show that it also occurs in the interpretation of English and in the grammars of other languages.

When you infer the description of a rearrangement from a couple of examples of its mappings, as above, you assume that the same rearrangement applies to longer trains, such as ones containing 10 or 100 cars. Your assumption is an inductive inference. And like all inductions, it has no justification: it could be that once a train is longer than eight cars, the function has a different sort of output, for example, it reverses the order of cars. Even though it lacks any justification, induction is not just a philosophical puzzle, but also a useful and habitual way of thinking (Ramsey, 1990/1926, p. 91). Your assumption that a function holds over longer trains than those in its examples is one that is built into our program, mAbducer, which creates formal and informal programs according to our theory of the process.

Functions are descriptions that don't do anything, and some functions cannot be computed. Most important is the one that Gödel (1967/1931) proved to be not computable. Its description is simple: for any arithmetical assertion, determine whether or not it can be proved in a consistent formalization of arithmetic. Hence, there are true assertions in arithmetic that cannot be proved. It is thus crucial not to confuse a function with a program that computes its values. Indeed, most functions cannot be computed, and for any function that is computable, there are infinitely many distinct programs that can compute it (Rogers, 1967). Quantum computers, if they can be made to work, will carry out computations much faster than current computers, and thereby make certain procedures tractable, such as those underlying cryptographic systems. But, they will not transform the uncomputable into the computable (Nielsen & Chuang, 2000).

Recursive functions

A set of basic functions can be used to define all others. So, what are these basic functions? A standard answer, which goes back to Gödel and others (Adams, 2011), relies on the fact that any function whatsoever, such as the reversal function for a railway train, can be treated as an arithmetical one. You can assign anything – an entity such as a railway car, a function, even a program – a unique number. And, in the railway domain, the prime factors of a judicious choice of such a number refer to a basic function, the number of its operands, and an encoding of the current state of the track. This fact about numbers is at the heart of what makes computation possible: it can all be done with numbers. Otherwise, the set of basic functions would depend on the particular domain. And we would need a different sort of computer for each domain that was not numerical. In the standard theory of recursive functions, there are three sorts of basic function that suffice to define any other function (see, e.g., Davis, 1958, Ch. 3): the *constant* function that takes any single positive integer as its input but always outputs 0; the *successor* function that adds 1 to any positive integer; and a set of *identity* functions each of which output one particular input from a set of several, for example, given three separate inputs of integers, one such function outputs the first of them, another outputs the second of them, and another outputs the third of them. The set of basic functions is not so important as how they can be combined to define a new function. Indeed, in an alternative account there is only one basic function, which takes three inputs, and suffices for the definition of any computable function (Melzak, 1961). So, what matters are the different ways in which the basic functions are put together to define new functions. And it is among these different ways that recursion first appears.

The theory of recursive functions uses three ways to assemble existing functions to define new functions. The first way is the *composition* of functions. For instance, we can define a function that maps the railway with at least one car on its siding into a railway with at least one car on right track and one less car on the siding. We define this function as a move of one car to right track (R) applied to a move of one car to left track (L): R is composed with L. We could encode the railway, cars, and its three basic functions, in arithmetical terms, but for simplicity we forego this exercise. The second way to assemble existing functions is *primitive recursion*. It uses at least two clauses to define a function: one states the value of the function for an input of zero, and the other states its value

for an input of (n + 1) in terms of its value for n. For instance, a primitive recursive definition of the railway's reversal function is as follows, where n refers to the number of cars in a train:

```
For a train with zero cars, output a train with zero cars, and
for a train of (n + 1) cars, put the n+1th car in front of the
reversal of n cars.
```

Mathematicians frame such functions for the purpose of proofs. We show how this definition works in a moment when we describe the computation of the function.

Most computable functions have primitive recursive definitions, and in the early 1900s mathematicians supposed that all of them had. But, Ackermann (1967/1928) discovered a computable function that cannot be defined using primitive recursion. It maps two integers into a new integer. Yet, with only small increases in the magnitudes of the two input integers, the new integer increases in magnitude at a rate faster than exponential. The definition of this function, and others of the same sort, calls for the third way to assemble existing functions into a new function: *minimization*, which is the most powerful form of recursion. It plays no part in rearrangements, and so we relegate its explanation to Appendix 1.

Programs

Programs describe the instructions for computing the values of functions. They need a computer to execute them. Programmers sometimes treat the word "function" as referring both to a function and to the program for computing its values. The confusion increases with their similar use of the word "recursion," which has contributed to the mystery associated with the word. Primitive recursion, as we remarked, is a way to define functions, but programmers refer to a "recursive function" as a program for computing such a function: it has a description that is circular in part, referring to itself – it is *semi*-circular. Some programming languages, such as early versions of Basic, do not allow programs to be described in this *semi*-circular way. These languages allow other ways to describe the computation. The instructions are numbered in a list, and to create a loop an instruction refers back to the number of an earlier instruction to elicit its execution (see also the programs for Turing machines, in Appendix 2).

Any recursive function can be computed using the repetition of a loop of instructions, and there are two sorts of loop (Rogers, 1967). In one sort, the loop is repeated *for* a stated number of times, i.e., it is a *for*-loop. But, this sort of loop won't work for functions that can be defined only in terms of minimization. Their computation calls for a loop that is repeated *while* a given condition holds, i.e., a *while*-loop. *While*-loops also work for programs to compute primitive recursive functions. We give examples of these two sorts of loop below. Meanwhile, a difference between them is telling. When a *for*-loop is entered, it states the number of times the loop of instructions repeats. When a *while*-loop is entered, there may be no way to compute how many times it will repeat before it outputs a result. Indeed, it may never yield an output: it may not be computable. When programmers write a *while*-loop that never ends, they have to interrupt its execution manually.

To illustrate the different ways of computing functions, we introduce simple diagrams to represent states of the railway computer. The following diagram, for instance:

```
-[bc]a
```

shows that there are no cars on left track, cars b and c are on the siding, which the square brackets demarcate, and car a is on right track. Cars enter the siding from left track and exit it to left track. We now illustrate the difference between *semi*-circular, *while*-loop, and *for*-loop programs.

Each of the three sorts of program is informal and reverses the order of cars in a train, and we adopt the convention of naming the program in the first line of its definition and stating its input variables in parentheses. The *semi*-circular program is as follows:

```
Reverse (track)
If left track is empty and the siding is empty output track,
otherwise if there is more than one car on left track,
    move all but one car to the siding,
    move one car to right track, and reverse track.
otherwise move one car to left track and then to right track,
    and reverse track.
```

We have italicized the self-referential calls that elicit the program itself. Given a train of three cars, abc, on left track: abc[-]-, the program acts as follows:

1. First, it calls its second instruction, which calls the function itself again to the updated track: `-[ bc] a`
2. This time the program calls its third instruction, which calls the function itself to the updated track: `-[ c] ba`
3. This time the program calls its third instruction again, which calls the function itself to the updated track: `-[ -] cba`
4. This time the program calls its first instruction, which yields the output of its preceding step (3), which is the output of its preceding step (2), which is the output of its first step (1), and so the final output is: `- [ -] cba`

The program reverses the order of a train of any length.

A *while*-loop program depends on the conditions that must hold for the loop to repeat. The following *while*-loop program for reversal is the one that the mAbducer program creates in Lisp and then translates into informal English (to which we have inserted parenthetical phrases for clarity):

```
Reverse-while-loop (track)
Move one less than (the number of) the cars (in the train) 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.
Output the track.
```

Unlike the preceding *semi*-circular program, each step in this one yields an immediate output. From the initial state of the track: abc[-] -, the first two instructions yield:

```
a[bc]   -[bc]a
```

The computation now enters the loop and carries out its two instructions:

```
b[c]a    -[c]ba
```

It repeats the loop again:

```
c[-]a     -[-]cba
```

It now halts because there are no cars on the siding, and outputs the final state of the track a program using a *for*-loop for a reversal repeats the loop for a stated number of times, which depends on the number of cars in the train. The mAbducer program creates such a program in Lisp (see Appendix 3), and we have translated it into informal English:

```
Reverse-for-loop (track)
Move all but one car to the siding.
Move one car to right track.
For one less than the number of cars in the train, repeat the loop:
   Move one car to left track.
   Move one car to right track.
              Output the track.
```

The effect of carrying out this program is identical to that of the *while*-loop above.

All three sorts of program can compute the same values for any primitive recursive function. They work in different ways, but otherwise are equivalent. In fact, no limit exists on the number of different ways to write a program to compute any computable function (Rogers, 1967). If any of our participants had devised a *semi*-circular program, we would have suspected that they must have had some experience in programming. But, none of them ever formulated a *semi*-circular program.

In summary, a function is a mapping that tells you *what* has to be computed if possible. A program (aka an algorithm or effective procedure) tells you *how* to compute a function: infinitely many different programs can compute the same function. And a computer is what carries out the program. A way to distinguish them is to remember, first, that many functions cannot have a program that computes them; and, second, that a program is a description geared to a particular sort of computer, human or machine, and so nothing happens until a computer executes the program for computing the function. Functions, programs, and computers, are therefore the core of computation. Marr (1982) made them familiar to cognitive scientists in his analysis of theories of vision at the computational level (function), at the algorithmic level (program), and at the implementation level (brain as computer). One consequence for cognitive science is that theories existing only at the "computational level" leave much to be explained, for example, the Gestalt, Piagetian, and Vygotskian theories of thinking (see Johnson-Laird, 1983, Ch. 1). They are equivalent to descriptions of functions, and so they may not be computable. As a psychologist once asked one of us, isn't there a simple way to tell whether a theory is computable? Of course you can infer that certain theories are computable, but there can be no general method to tell you – computability is a function (mapping any theory to a binary value of whether it is computable or not), and it is not computable. As Marr also warned cognitive scientists, accounts that exist only at the algorithmic level may lack a clear theory at the computational level. He had in mind accounts of stereopsis, but his worry anticipates the difficulty of understanding what function deep neural networks learn to compute.

Computers, their power, and the Chomsky hierarchy

A program is a description written in a language that, directly or indirectly, controls a computer. But, the computer must have sufficient power to carry out the computation. Your computer has more power than ours if yours can execute all the programs that ours can, plus some additional programs. What constrains computational power is memory – in particular, the nature of the computer's "working" memory for intermediate results. It is therefore sensible to ask how much power do humans need for thinking and for using natural language. The standard analysis of computational power is known as the Chomsky hierarchy, because its origin is in his analysis of grammars (Chomsky, 1959). At the lowest level of power are computers that can cope with only a finite number of input-output pairs, for exmple, a vending machine. They need no working memory for intermediate results. One level up in the hierarchy are *finite-state* computers. They have no working memory, but instead enter into only a finite number of different states. Yet, they can produce infinitely many different outputs. Certain rearrangements can be carried out using

the railway as a finite-state computer, such as a program that swaps around the orders of adjacent pairs of cars in a train. It rearranges abcdef into badcfe (see Appendix 3). It can cope with trains of any length, and so it has a countable infinity of different outputs. As it swaps the orders of adjacent pairs of cars, it yields this sort of sequence:

```
abcd[-] abc[d] ab[d]c abd[-]c ab[-]dc a[b]dc -[b]adc b[-]adc  -[-]badc
```

Only a single car ever has to be moved onto and off the siding during the execution of the program. So, no matter how long a train is, human users of the railway computer have only to remember where they are in the sequence of moves: S1 R1 L1 R1, which they repeat until left track is empty.

When a computation calls for more than a fixed number of states, it needs a working memory, and such a memory can take the form of a *stack*. It works like the siding on the railway except that programmers tend to think of a stack as vertical. Access to the stack, like the siding, is at one end only, and in principle it has to accommodate a train of any length. Once an item is removed from a stack, it goes at once to the output. So, for the siding to behave as a stack, when cars leave the siding for left track, they must then go at once to right track. When the siding is used in this way, the railway computer is up one level in the hierarchy of power: it is a *push-down* computer – it has this name, because when an item is put on top of a vertical stack, it pushes down the items beneath it. This sort of computer can carry out the computations for a reversal of the order of cars in a train of any length – a computation beyond the power of a finite-state computer. For a reversal, the siding needs to accommodate one less than the number of cars in a train. The computation treats the siding as a stack, because as soon as it removes a car from the siding, it moves the car via left track to right track.

A push-down computer can match parentheses in algebraic expressions – a task to which we return in the section on language, and it can parse sentences using other *context-free* grammars (for a proof, see Chomsky, 1959). A grammar is a function, and its rules are context-free provided that they specify the constituents of a phrase without regard to the grammatical context in which the phrase occurs. Context-free grammars can handle both most of the structure of programming languages (Aho & Ullman, 1972, p. 138) and the fragment of natural language used in the informal programs that mAbducer creates.

A computer with a stack accommodating an unbounded number of items cannot compute certain programs. For instance, it cannot carry out a faro shuffle, which we described earlier. The proof is simple, and it is shown in the moves of the cars on and off the siding during a solution to a faro shuffle using a minimal number of moves. (We explain presently how we know the sequence is minimal.) Suppose that the initial track is: hgfedcba[-]. The following sequence uses the minimal number of moves to carry out the faro shuffle, where we have shown in bold those cars you should pay attention to:

```
-hgfedcb[-]a hgfe[dcb]a hgf[dcb]ea hgfdcb[-]ea hgfdc[-]bea hgf[dc]bea
hg[dc]fbea hgdc[-]fbea hgd[-]cfbea hg[d]cfbea h[d]gcfbea hd[-]gcfbea-
[-]hdgcfbea
```

If you check the movements of d, you will see that it enters the siding on the second move above along with c and b, exits with them two moves later, but like c it does not go at once to the output on right track. Instead, it later moves back onto the siding, off again, and so on, before exiting after its third visit. Such movements are impossible in a push-down computer, because as soon as an item leaves its stack, it goes to the output. A subtlety of the railway is therefore that its left track can act as a stack too: left track and siding shuffle cars to and fro in order to carry out a faro rearrangement. However, the number of cars that they have to hold is a linear function of the length of the train and, in particular, a proportion of its length. Because a rearrangement has no effect on the number of cars in a train, if each of the three parts of the track can accommodate the train, no need ever occurs to lengthen the track. This use of the railway therefore has the power of a *linear-bounded* computer, i.e., it needs access to a working memory that is a linear function of the length of the input. Such a computer can cope with parsing *context-sensitive* grammars with rules that can take into account the grammatical context of a constituent. For example, an auxiliary verb, such as *have*, in a verb phrase should be singular for a grammatical subject that is a singular, and a grammar can handle this dependency using a context-sensitive rule (e.g., Chomsky, 1957, p. 39).

A computer with two unbounded stacks has the power of a universal Turing machine. (Hopcroft & Ullman, 1979, Sec. 6.6). No sort of computer, as far as anyone knows, has any greater power. Likewise, an unbounded version of the railway is also at the top of the Chomsky hierarchy: its conversion into a Turing machine is straightforward (see Appendix 2).

In summary, minimization is the most powerful sort of recursion, but it is beyond the ability of naive individuals to compute except for tiny inputs (see Appendix 1). Primitive recursion defines a computable function in a *semi*-circular referential way. *Semi*-circular programs and *while*-loops can compute minimizations and primitive recursions. *For*-loops can compute primitive recursions. A linear-bounded computer can carry out programs that compute any primitive recursive rearrangement, including those for a faro shuffle, a reversal, and a swap of adjacent cars. To determine whether naive individuals can create informal programs for these primitive recursive functions, we have identified the power required for the corresponding mental processes. Because informal programs are described in natural language, we need to identify the power that its use calls for. The topic is highly controversial. Just about every level in the Chomsky hierarchy – from finite-state computers up to universal Turing machines

– has had its defenders, and finer subdivisions exist within the hierarchy (e.g., Jäger & Rogers, 2012). We return to this controversy after we shown how naive individuals are able to create informal programs to compute primitive recursive functions.

A theory of how naive individuals create informal programs

An open question at the start of our research was whether naive individuals could create programs for any primitive recursive functions. Nevertheless, we formulated a theory of how in principle they should be able to do so. This section presents that theory. It postulates that individuals carry out three main mental processes in order to program a rearrangement of trains of any length. These processes should:

1. Solve examples of the required rearrangement on the railway in order to understand what the program is supposed to do.
2. Abduce a program to compute the rearrangement.
3. Deduce the consequences of the program to test whether it is correct.

The mental processes underlying these tasks differ, and we implemented all three of them in mAbducer, which automatically creates programs in Lisp for solving rearrangements and translates those based on *while*-loops into informal English. We now describe how it carries out each of the three tasks, and we interject an account of the complexity of programs, which should predict the difficulty both of abducing them and of making deductions from them.

Problem solving: The discovery of solutions to rearrangements

The first step in the development of a program is to solve examples of the rearrangement, that is, to discover the required sequence of moves – the fewer, the better. These moves should get the train on left track to the goal of its rearrangement on right track. The search for a solution can be carried out on the railway itself, but the present theory allows that individuals can also simulate moves using a kinematic mental model. Models are iconic in that they correspond insofar as possible to the structure of what they represent, and so kinematic models unfold in time in order to represent movements in the environment. We therefore refer to the present account as the "model" theory.

The *problem-solving* component of the theory explains how people find solutions to rearrangements. It postulates that they start by using trial and error, but, as previous studies of reverse engineering and of other sorts of problem solving have shown, they soon acquire both local and global constraints. These constraints reduce the number of alternatives in a depth-first search for a solution, which human solvers almost always adopt (Lee et al., 2008; Lee & Johnson-Laird, 2013a, 2013b). The theory therefore postulates a search in which if the constraints allow more than one move, an arbitrary choice is made from them. The search uses a variant of the "means-ends" analysis often invoked as a general method for solving problems (e.g., Newell, 1990). We refer to this variant as a *partial* means-ends analysis. The standard analysis is to envisage the required goal and to work backwards from it, invoking actions relevant to reducing the difference between the goal and the current state of the solution. In the case of rearrangements, individuals do not need to envisage the goal in its entirety, but merely each of its successive parts.

Consider this example of a "parity sort," which is the converse of a faro shuffle:

`abcdef[-]- ⇒ -[-]acebdf`

The rearrangement moves all the cars in odd-numbered positions in front of all the cars in even-numbered positions. The match between the position of f in the initial arrangement and in the target rearrangement, elicits an obvious move:

`R1: abcde[-]f`

There are only three sorts of move (R, S, L), but trial and error soon leads to an explosion of possibilities. Even if we discount the number of cars in a move, there are several thousand possible sequences of 8 moves. But, constraints reduce the number, for example, L is pointless without a preceding S. And human reasoners soon learn the tactic of moving cars onto the siding to enable cars behind them to move to right track. Hence, in the present example, the next part of the goal is to get d to right track, but car e is in the way, so it must move to the siding, and then d can move to the right:

`S1: abcd[e]f` `R1: abc[e]df`

In general, the theory postulates that in an ideal performance people proceed according to the following constraints:

> While the rearrangement still has a part to be solved, they consider, first, whether more cars on left track can be moved to right track to satisfy part of the goal than cars on the siding can be moved via left track to do so. They make the appropriate move. In case neither of the preceding sorts of move can be made, if a car or cars on left track satisfy part of the goal but are blocked from moving, they move the blocking cars to the siding, and then make the move to right track.
> Otherwise, they move as many cars from siding to left track that can satisfy part of the goal, and then move the largest subset of them to right track.

In this way, reasoners can solve the rearrangement by decomposing it into moves of one or more cars into their correct positions. To solve the parity sort of six cars above, the procedure yields this sequence of moves:

```
R1 S1 R1 S1 R1 L2 R3
```

For a parity sort of eight cars, it yields this solution:

```
R1 S1 R1 S1 R1 S1 R1 L3 R4
```

Given the aim that programs should yield solutions calling for a minimal number of moves, it is reasonable to wonder whether there might be more parsimonious solutions. The mAbducer program as an exercise in artificial intelligence also finds solutions that make minimal numbers of moves. It constructs all possible sequences of moves breadth first to solve a rearrangement. It finds all feasible continuations for each current sequence of moves, and adds them to create a set of new sequences. When a sequence can go no further or solves the rearrangement, it is removed from the list, and the final output is the set of successful sequences. Some rearrangements have more than one minimal solution. This AI method is computationally intractable, but it yields minimal solutions for each of the rearrangements in our experimental studies (see Appendix 3). And it is how we knew that the sequence in the earlier section for a faro shuffle was minimal.

The model theory predicts that the main difficulty in solving a rearrangement depends on the number of moves in a minimal solution. It also predicts that perseveration is likely to lead to solutions of more than a minimal number of moves. For example, in searching for a solution to the parity sort, if individuals find the correct penultimate move of all the cars on the siding to left track, they should have a tendency to perseverate and to move them on to right track. It is not a mistake, but it leaves behind one car on left track, which was already there, and which then has to be moved by itself to right track. A parsimonious solution is instead to move all the cars on left track to right track, not only those that have just been moved from the siding.

Programming: The abduction of programs

To create an informal program calls for individuals to describe how to solve a rearrangement in a sequence of instructions. The description is in a natural language so that a user can carry it out on the railway computer. Programming is easy to confuse with the preceding task of problem solving, but the difference between them becomes clear if the program has to carry out a rearrangement for trains of any length, i.e., of any number of cars. It is one thing to find a sequence of moves that solves a parity sort of abcdef, but quite another to create an informal program for a parity sort of trains of any length. The program cannot be deduced either from a description of the function such as: *all the cars in odd-numbered positions are put in front of all the cars in even-numbered positions*, or from examples of its mappings. And it cannot be an inductive generalization from them, either. Its construction is akin to the creation of explanation or plan of how to get from one arrangement to another. It depends on a process of abduction.

A common view of abduction is that a reasoner seeks a hypothesis from which an observation follows, so that the hypothesis implies the observation (Peirce, 1955; for a survey, see, e.g., Douven, 2017). In our theory, abduction is an induction that introduces at least one new concept that is not part of the observations from which it starts, and that it uses in part to explain these observations, i.e., it implies them. Reasoners can observe the sequence of moves that solve a particular rearrangement, but to create a program that applies to trains of any length, they are forced to introduce an idea that is not explicit in these observations – the idea of a *for*-loop or of a *while*-loop. The creation of a program is therefore an exemplary case of abduction, because it introduces a novel idea, a loop, and it formulates a program from which the initial solutions follow. We now describe the model theory's account of the process.

The abduction starts with the moves in the solution to the rearrangement. The solutions above of the two instances of the parity sort are:

```
abcdef ⇒ acebdf:      R1 S1 R1 S1 R1 L2 R3

abcdefgh ⇒ acegbdfh: R1 S1 R1 S1 R1 S1 R1 L3 R4
```

The first step is to search in these simulations for a loop of more than one instruction: a loop of a single instruction merely increases the number of its operands, for example, R1 becomes R4. The sequences above have two equivalent loop structures: either the loop starts at once with repeats of R1 S1, and later calls for a single R1 prior to the two final moves, or else after an initial R1, the loop repeats S1 R1 prior to the two final moves. Granted, say, the latter loop structure, individuals can start their description of an informal program with:

```
Move one car from left to right track.
```

The simulations show that the pair of moves in the loop, S1 R1, for six cars has two repetitions, and for eight cars has three repetitions. Individuals can then infer a *for*-loop:

```
For a number of times equal to half the length of the train minus one,
repeat:
      move one car to the siding,
      move one car from left to right track.
```

They can likewise infer the final two moves:

```
Move the number of cars equal to half the length of the train minus one to
left track.
Move the number of cars equal to half the length of the train from left to
right track.
```

Human perception makes loops seem obvious. But, they are not obvious for a program, and so mAbducer starts with

the maximum possible repeated sequence of moves in a solution, which is half the length of the solution, and checks whether it is repeated. If not, it looks for a repetition of one move less in length, and so on down, until it looks for a pair of repeated moves. Hence, in the parity sort it detects the repetition of S1 R1 after the initial move of R1. It also detects the other loop structure. To determine the number of repetitions of the *for-loop*, mAbducer solves two simultaneous linear equations of the following sort, where n is the number of cars in the train, and a and b are the unknowns:

Number of repetitions of loop = (na) + b.

So, the equations for the parity sort above are:

$$2 = 6a + b$$
$$3 = 8a + b$$

and the solution is a = 1/2 and b = -1. Hence, the *for*-loop is repeated for ½n – 1 times. The same procedure also determines the number of cars in any moves before or after the loop that depend on the number of cars in the train. Those individuals who formulate a correct *for*-loop carry out an equivalent procedure, though they may be surprised to learn that in effect they are solving two simultaneous linear equations.

An alternative way for people to program the parity sort is with a *while*-loop. Individuals simulate solutions to two trains of different lengths, and from the state of the track on entering and on exiting the loop they infer the condition governing its repetition, so that when this condition no longer holds, the program exits from the loop. In their simulation of the solution of a parity sort for a train with six cars, they observe that after an initial move, the loop starts in this situation:

```
abcde[-]f
```

and continues while there is more than one car on left track until this situation:

```
a[ce]bdf
```

Whereupon, it halts. The crux is that the loop continues while there is more than one car on left track. Their program starts with the initial move that precedes the loop. mAbducer simulates this procedure to abduce the following program (see Appendix 3):

```
Move one car to right track.
While there is more than one car on left track
   move one car to siding,
   move one car to right track.
```

Naive individuals have a simple procedure for the two final moves, which they could also use after the *for*-loop:

```
Move all the cars on the siding to left track.
Move all the cars on left track to right track.
```

The use of quantifiers, such as "all the cars," calls for an implementation of the predicate calculus (see, e.g., Khemlani & Johnson-Laird, 2021), which is outside the scope of mAbducer, and so it describes moves in terms of the length of the train. Table 6 in Appendix 3 presents all mAbducer's programs for the rearrangements in our studies – those with a *for*-loop, a *while*-loop, and its automatic translation of the latter into informal English.

In programs calling for the power of linear-bounded computers, the required loop is *dynamic*, i.e., the number of cars in at least one move in the loop changes with each repetition. For example, the faro shuffle has a loop of four moves, R S R L, but the numbers of cars for S and L depend on the initial length of the train and on how many repetitions of the loop have already occurred. The loop to shuffle eight cars is repeated three times and followed by a single R2 move:

```
R1 S3 R1 L3
R1 S2 R1 L2
R1 S1 R1 L1
R2
```

The loop is dynamic because S and L move three cars in the first iteration, two cars in the second iteration, and 1 car in the third and final iteration. These values depend on the solution of three simultaneous linear equations based on the total number of repetitions of the loop and the number of the current repetition. The difficulty of solving a rearrangement depends on its minimal number of moves. In contrast, the model theory predicts that the difficulty of an abduction depends on the complexity of the required program, and so we now consider how best to assess this variable.

The complexity of programs

A potential criterion for the complexity of a program is the computational power needed to execute it. So, for instance, a program that swaps the order of the two cars in each adjacent pair requires only a finite-state computer, a program that does a reversal requires a push-down computer, and a program that does a faro shuffle requires a linear-bounded computer (see Appendix 3). Another aspect of power is tractability, that is, whether processing time and demands on memory are a linear, a polynomial, or a still greater function of the length of the input (see, e.g., Garey & Johnson, 1979). However, both computational power and tractability are too crude as indices, because they cannot distinguish the difficulty of different programs within their categories. And, if working memory can handle a kinematic model that shuffles cars between siding and left track at least up to a certain number, then the difference between rearrangements calling for this linear-bounded memory and those calling for only a push-down memory may not be reflected in human performance.

All rearrangements are primitive recursive – the class includes those that require only a finite-state computer – and so a *for*-loop suffices for them, though human reasoners should prefer *while*-loops, because they do not call for calculating the number of times a loop should repeat. Hence, another pertinent factor is the number of loops in a program. Working memory is needed for programs based on one loop, but some rearrangements call for more than one loop (see Appendix 3).

If a reversal is carried out on a train, and then carried out on its result, the outcome is the train in its original order. Reversals are "reversible" to use Piaget's term for an aspect of children's operations on the world (Inhelder & Piaget, 1958). However, as mAbducer shows, to return to the original order of a train of 52 cars, a faro shuffle has to be carried out eight times – a stratagem often used by magicians and card sharps (Diaconis et al., 1983). Different rearrangements take different numbers of repetitions to get back to the original order of items, and this number might index the relative complexity of a program. In fact, it doesn't. Given a train of eight cars, for instance, a palindrome rearrangement (see Appendix 3) takes four repetitions to return the train to its original order whereas the faro shuffle takes only three.

So far, our potential keys to the complexity of a program are computational power, tractability, number of loops, and number of repeated rearrangements needed to return a train to its original order. We considered weighting them in a single measure, but a more plausible criterion seemed to be the number of instructions that a minimal program contains. This factor changes from one programming language to another. But, a single index that might encapsulate the critical factor is Kolmogorov complexity (e.g., Kolmogorov, 1965), which we refer to as *K-complexity*. It is the number of symbols in a standard language that are needed to describe the simplest possible program for computing a given function. Beyond a certain size, however, it is impossible to prove that a program is minimal (Chaitin, 1998). Nonetheless, mAbducer finds minimal solutions to rearrangements of a moderate size, and the programs it develops – even if they are not minimal – are simple enough for K-complexity to be a feasible measure of their complexity. Table 1 presents, in rank order of their intuitive complexity, the eight main programs in our experiments: it states their numbers of moves before, during, and after loops, and the numbers of their operands, i.e., cars. It also shows that the intuitive complexity matches the rank-order of K-complexity as shown in the number of words in mAbducer's Lisp programs using *while*-loops and in its English translations of them. The one discrepancy occurs with the palindrome that calls for two separate loops of instructions (the two-loop palindrome), which has a much smaller K-complexity for the Lisp function than for its translation into English. If human working memory is equivalent in power to a linear-bounded computer, the factors in Table 1 should also govern the difficulty of the abduction of programs. K-complexity should then predict the difficulty of abducing programs, and the difficulty of deducing their consequences.

Deducing the consequences of programs

The original model theory was developed to explain human deductions. One of its key principles is that models are iconic insofar as possible, that is, their structure corresponds to the structure of what they represent (Johnson-Laird, 1983, p. 419), and so kinematic models unfold in time as do the events they represent whether real or imaginary (ibid., p. 423). Some tokens in models are symbolic rather than iconic, and they relate to procedures that deal with abstract concepts, such as negation (Khemlani et al., 2014). The theory also distinguishes between implicit models that underlie intuitive inferences and their explicit counterparts that underlie deliberate inferences (e.g., Johnson-Laird, 1983. Ch. 6). Such a "dual process" account of reasoning is due to the late Peter Wason (Manktelow, 2021). Many others later developed the idea (e.g., Evans, 2008; Kahneman, 2011). Intuitions may play a role in thinking about programs, but deductions from them call for deliberations. Likewise, although some deductions concern probabilities (e.g., Khemlani et al., 2015), they do not appear to enter into inferences testing deterministic programs (cf. Oaksford & Chater, 2020).

Deductions from programs call for a kinematic process that simulates the effects of a sequence of moves. Hegarty and her colleagues pioneered the empirical study of such models of systems of pulleys and cogs, and discovered that individuals could animate only one cog or pulley at a time (e.g., Hegarty et al., 2013; Lowrie et al., 2019). This constraint corresponds to simulations of the railway, because each move affects only one or more adjacent cars that move together. The deductive process identifies a move from its description in the program, carries it out on the current model of the railway to yield its next state, and then moves to the next instruction in the program. If the program is incorrect, the simulation may halt too soon, deliver an erroneous rearrangement, or fail to halt. But, the deductive consequences of a program can only show that it is correct if it is tested with all possible inputs. So, correctness for programs for rearrangements calls for an inductive proof.

As an example of a deduction, consider the informal program for a parity sort from the preceding section:

Table 1 The rank order of the intuitive complexity of eight programs for rearrangements based on their number of moves (and of their operands, i.e., cars) prior to the loop (pre-loop), in the loop, and after the loop (post-loop), where n is the number of cars in a train, and their K-complexity (Kolmogorov complexity) as defined in the number of words in mAbducer's *while*-loops and in its English translations of them

Rearrangement	Pre-loop number of moves (and of number of cars in them)	Loop number of moves (and of number of cars in them)	Post-loop number of moves (and of number of cars in them)	K-complexity: number of words in Lisp *while*-programs (and their English versions)
1. Swap adjacents abcdef ⇒ badcfe	0	4 (1)	0	37(38)
2. Reversal abcdef ⇒ fedcba	1 (n - 1) 1 (1)	2 (1)	0	41(40)
3. Palindrome abcdef ⇒ afbecd	1 (½n - 1) 1 (2)	1(1) 1(2)	0	41(42)
4. Center palindrome abcde ⇒ aebdc	1 (½n - ½) 1(1)	1 (1) 1 (2)	0	41(42)
5. Make palindrome abcdef ⇒ acefdb	1 (n - 2)	1 (1) 1 (2)	1 (½n + 1)	45(44)
6. Two loop palindrome abcdef ⇒ cbafed	0	2 (1) in loop 1 2 (1) in loop 2	0	40(48)
7. Parity sort abcdef ⇒ acebdf	1 (1)	2 (1)	1 (½n - 1) 1 (½n)	50(54)
8. Faro shuffle abcdef ⇒ adbecf	0	2 (1) 2 (½n - 1)	1 (2)	68(79)

Note: The programs for two-loop palindromes were hand-crafted, because mAbducer deals only with programs containing a single loop

```
Move one car to right track.

While there is more than one car on left track

   move one car to siding,

   move one car to right track.

Move all the cars on the siding to left track.

Move all the cars on left track to right track.
```

Individuals simulate the initial state of the railway, and then the consequences of the first move:

```
abcdef[-]-   abcde[-]f
```

They now simulate the *while*-loop, which they repeat until there is only one car on left track:

```
abcd[e]f    abc[e]df

ab[ce]df    a[ce]bdf
```

They then carry out the two final moves:

```
ace[-]bdf   -[-]acebdf
```

So, the final rearrangement is a parity sort: acebdf, and the program has survived its first test. As in abducing a program, the theory predicts that the difficulty of deducing its consequences should depend on its K-complexity.

Experimental evidence

Experiments on solutions to rearrangements

If individuals use a partial means-ends analysis to solve rearrangements, then two factors should affect the difficulty of the task. The first factor is the number of moves in the solution, and the second factor is the total number of cars that move. The latter has a family resemblance to "relational complexity", i.e., the number of arguments in a relation or function, which also affects the difficulty of solving problems (Halford et al., 2010). However, the number of cars in a move is different: it concerns whether a single input refers to one or more

items. The two factors should both contribute to difficulty, leading individuals to take longer to solve a rearrangement and to make more than a minimal number of moves.

The first experiment to test these predictions for the solution of rearrangements allowed participants to move the cars on a computer-presented railway as in Fig. 1 (Khemlani et al., 2013, Experiment 1). These participants, as in all our studies, were naive in that they knew nothing about computer programming or its cognate disciplines. The participants had to solve all 24 possible rearrangements of trains containing four cars, and Table 2 collapses the results into 12 sorts. A rearrangement that requires only one move has a probability of being solved by chance of 0.125, because there are eight possible initial moves for a train of four cars (S1, S2, S3, S4, R1, R2, R3, R4) and only one of them solves the rearrangement. The next sort of rearrangement in the study calls for four moves, and it has a probability of being solved by chance of 0.0004 (1/8 × 1/8 × 1/6 × 1/6), taking into account only those moves that are possible after each move is carried out, and the chance probabilities are still smaller for the rearrangements with a greater number of moves. As Table 2 shows, the mean numbers of moves that the participants took to solve the rearrangements correlated with both the predicted number of moves and with the predicted total number of cars in moves, but participants often made unnecessary moves. The response times showed the same effects. The participants carried out two additional rearrangements in which they had to think aloud as they tackled them. Their remarks showed that they focused on the cars in the targetrearrangement, working backwards from the car at the front of this train. So, they were using some sort of partial means-ends analysis.

Given the relative ease of solving rearrangements, a subsequent study examined 10-year-old children's performance in solving four rearrangements of six cars (Bucciarelli et al., 2016, Experiment 1). Table 3 presents the results. The probability of solving these rearrangements by trial and error is remote, for example, the parity sort has a solution that calls for seven moves, and the probability of selecting this sequence by chance is less then one in a million.

As Table 3 shows, the participants made very few errors, but the rank order of their mean numbers of moves and times to solution correlated reliably with the number of moves in minimal solutions. The use of more complex rearrangements made it impossible to examine the total number of cars in moves independently from the number of moves. Like adults, the children tended to make redundant moves: all but one of the 20 children made at least one redundant move. The children differed from one another only in the time it took them to solve the rearrangements – a difference that correlated neither with age nor gender.

It is not surprising that individuals' solutions correlate with the number of moves in minimal solutions, or that their main shortcoming is to overlook parsimonious solutions. However, their number of moves and the differences from one rearrangement to another make a striking contrast with the predictions for the difficulty of the two other tasks in developing programs for rearrangements.

Experiments on abductions of programs

Five experiments tested participants' abductions of informal programs, and Table 4 summarizes their results. The standard procedure was for the participants to solve some preliminary rearrangements in order to become familiar with the railway, and then for them to formulate descriptions of programs for solving various other rearrangements. During this latter phase, they were not allowed to move the cars. To make a correct abduction of a program by chance is akin to the proverbial monkey typing a soliloquy from *Hamlet*: it calls for solving the rearrangement of six cars, which itself has a probability of about one in a million (see the previous section), for realizing that a loop of moves is needed for trains of any length, and then for discovering a correct loop. Hence, even the lowest percentages of correct programs in the experiments we report

Table 2 The mean numbers of moves from an experiment in which participants (N = 20) solved the 24 possible rearrangements of four cars (Experiment 1 in Khemlani et al., 2013). The means depended on the numbers in minimal solutions (means in right hand column) and on the total numbers of cars to be moved in minimal solutions (means in bottom row)

Number of moves in minimal solutions	Total number of cars moved in minimal solutions					Mean number of actual moves
	4	6	8	10	12	
1	1.00					1.00
4		4.33	4.68	4.55		4.48
5		5.50	5.20			5.43
6			6.51	6.63		6.56
7			7.90			7.90
8			8.28	8.48	8.55	8.41
Mean number of actual moves	1.00	4.92	6.51	6.90	8.55	

Table 3 The number of moves in minimal solutions to four sorts of rearrangement of trains of six cars, and the percentages of correct solutions that 10-year-old participants (N = 20) made, their mean numbers of moves and mean solution times (Experiment 1 in Bucciarelli et al., 2016)

Rearrangements	Minimal numbers of moves	Percentages of correct solutions	Mean numbers of actual moves	Mean solution times (s)
Palindrome:				
`abccba ⇒ aabbcc`	6	100	8.2	42
Parity sort:				
`abcdef ⇒ acebdf`	7	100	9.9	49
Faro shuffle:				
`acebdf ⇒ abcdef`	9	100	11.6	55
Reversal				
`abcdef ⇒ fedcba`	12	95	16.3	70

are better than those that would occur by chance. In Experiment 2 of Khemlani et al. (2013), the participants had a block of trials in which they had to formulate programs for trains of eight cars, and a block of trials in which they had to formulate programs for trains of any length. Two groups of participants carried out the blocks in counterbalanced orders. The percentages of accurate programs for trains of any length are shown in Table 4 for these two groups (columns i and ii): when they tackled trains of any length in the second block they were more accurate than when they tackled them in the first block (82% vs. 65%; Mann-Whitney test, $z = 1.70$; $p < 0.05$). So, a positive transfer appears to occur from a program for a train of fixed length to a program for a train of any length. In fact, both sorts of program tended to use loops—the latter are bound to do so, and participants preferred to use *while*-loops (82%) rather than *for*-loops (18%). Here is a typical protocol for the palindrome rearrangement (abccba ⇒ aabbcc), as a participant described it, but with the *while*-loop italicized:

> The cars on the left are letters A-Z and Z-A (sic) in order. Move all cars right of the last available letter to the side track. Move both copies of the last available letter to the right track. Then move one car from the side track back to the left track. Move the two rightmost cars from the left track to the right track. *Repeat this cycle until all cars have been 'paired' and moved to the right track.*

Bucciarelli et al. (2016) carried out a similar experiment with 10-year-old Italian children in which they had to describe programs for rearrangements of trains with six cars. They were able to do so (see column iii in Table 4), and they even used simple precursors to loops, such as, "I move B from the siding to the left track then to the right track, and *the same with C, D, E, and F*" (translated from the Italian). The children were not allowed to move the cars, but they made a striking number of gestures – some pointed to individual cars, and

Table 4 The percentages of accurate abductions of programs in nine experimental groups. The K-complexity of the programs is shown in terms of the number of words that the program mAbducer used in its Lisp *while*-loops and in its translations of them into English, and the minimal number of moves required to solve a re-arrangement for trains of six cars

			Percentages of accurate programs in nine groups								
Rearrangements	K-complexity Lisp (English)	Number of moves	i	ii	iii	iv	v	vi	vii	viii	ix
1. Swap adjacents	38 (37)	12								77	54
2. Reversal	40 (41)	12	89	91	81	84	78	91	83	71	11
3. Palindrome	42 (41)	6	44	27	90	78	66	83	83		
4. Make palindrome	44 (45)	6								63	0
5. 2 loop palindrome	48 (40)	12								54	3
6. Parity sort	54 (50)	7	56	9	43	38	31	83	83	63	6
7. Faro-in shuffle	79 (68)	9			14	59	34	75	75		

Note: i) Khemlani et al. (2013), Experiment 2, trains of any length in second block of trials, ii) trains of any length in first block of trials; iii) Bucciarelli et al. (2016), Experiment 2, iv) Experiment 3 with gestures allowed, v) Experiment 3 with no gestures allowed; vi) Mackiewicz (in preparation), with letters on cars, vii) with numbers on cars; viii) Bucciarelli et al. (2018), Experiment 2, for trains of six cars, and ix) for trains of any length

others mimicked moves. When they were prevented from gesturing in a subsequent study, their abduction of accurate programs fell by 13% in comparison with those in a group who were allowed to gesture (see the differences between columns iv and v in Table 4). Mackiewicz and his colleagues in a study of eye-movements (in preparation) tested adults in two groups, one with letters on the cars and one with numbers on the cars. Their results also corroborated the predicted trend (see columns vi and vii in Table 4). And Bucciarelli et al. (2018) called for 10-year-olds to formulate programs for trains of six cars and for trains of any length (see columns viii and ix in Table 4).

Each of these experiments corroborated the K-complexity trend of difficulty in abducing programs. Children's gestures bore out the theory's claim that people rely on a kinematic model of the track in order to formulate programs: the gestures helped the children to maintain an accurate model of the effects of moves.

To assess the experiments overall, we examined the proportions of differences within their means that matched the predicted trend from K-complexity (using the P statistic computed for tau in Kendall & Gibbons, 1990). Eight out of the nine groups in Table 4 yielded more than half such matches, and condition (vii) was a tie (Binomial test, $p<$ .0001). The size of the effect of K-complexity is evident in its correlations with difficulty in these nine groups: Kendall's tau, which ranges from -1.0 to +1.0, varied from .33 to 1.0 with a mean of .66. As Table 4 also suggests, the number of moves to solve a rearrangement for six cars did not correlate reliably with the difficulty of abducing a program: tau varied from -1.0 to +.33, with a mean of -0.39, and the difference between the taus for K-complexity and for number of moves was reliable (Mann-Whitney test, $z = 3.49$, $p<$ .00025). In sum, the complexity of programs rather than the number of moves in the rearrangements tended to predict the difficulty in abducing them.

Experiments on deductions from programs

Three experiments have examined the ability of participants to deduce the consequences of programs. The first study tested 43 adult Polish participants with deductions from programs in their native tongue for reversals, palindromes, parity sorts, and faro shuffles. They were translated into Polish from mAbducer's English *while*-loops, but expanded to clarify them and to ensure that all four descriptions had the same number of words (Khemlani et al., 2013, Experiment 3). The participants first watched a video that explained the railway and rearrangements. After two simple practice rearrangements, they carried out the experiment with access neither to the railway nor to paper and pencil. They had to deduce the consequences of the programs on a given train of six cars. There are 720 possible rearrangements of six cars and only one correct deduction, and so success by chance has $p<$ .0015). Table 5 shows the percentages of correct deductions. K-complexity predicted the reliable trend in accuracy, and it also predicted how long it took the participants to make the deductions.

A second study tested 30 10-year-old Italian children, who had to deduce the consequences for trains with five cars of two versions of three sorts of program: reversals, center palindromes, and parity sorts (Bucciarelli et al., 2018). One version was with *while*-loops, and the other version was without a loop and applied only to the five cars. The center palindrome is a version of the program for a palindrome but adapted for an odd number of cars in which there was a single central car, as in abcba (see Appendix 3). It has the same K-complexity as the palindrome. Table 5 shows the percentages of the children's correct deductions (solution by chance for five cars has $p<$ .009). Programs described without loops were easier than those described with loops—loops impose a greater load on working memory, because reasoners have to keep track of the *while*-condition. The versions with loops had the trend in difficulty that K-complexity predicts. The children differed in ability: two children made only correct deductions, and three children made no correct deductions. There was no reliable difference in accuracy between the sexes.

A third study tested 23 Polish students at the University of Social Sciences, Warsaw (Mackiewicz et al., 2016). They made deductions for trains of three lengths (four, six, and eight cars) from four programs: reversals, palindromes, parity sorts, and faro shuffles. Table 5 presents their overall percentages of correct deductions (chance successes have ps of less than .05, .0015, .000025, respectively). Once again, K-complexity correlated reliably with the accuracy of the deductions over the four sorts of rearrangement (mean tau = .75). In contrast, the number of moves to make the rearrangement did not correlate (mean tau = -.42) ; and the difference between the two sets of taus was reliable, Mann-Whitney test, $z = 2.17$, $p<$ .025). The participants differed in ability: the three most accurate participants made only correct deductions, but one participant made no correct deductions.

The cumulative results in Table 5 show that K-complexity is a reasonable proxy for the complexity of programs containing loops in that it predicts the difficulty of deducing their consequences quite well. These results make a striking contrast to those from solving rearrangements. In deductions, reversals were easiest, whereas they were the most difficult rearrangements to solve (see Table 3). So, what makes for difficulty in deducing the consequences of a program is, not the number of moves that it calls for (see Table 5), but its complexity, which includes such factors as the number of its instructions, both in loops and outside them (see Table 1). These factors increase the difficulty of simulating the program in a kinematic model of its effects, because they add to the load on working memory. And this difficulty is predicted from the

Table 5 The K-complexity of five programs, the minimal number of moves to make each of their rearrangements, and the percentages of correct deductions from them in three experiments

	K-complexity in Lisp (and in English)	Minimal number of moves	Percentages of correct deductions			
Rearrangement			i	ii without loops	iii with loops	iv
1. Reversal	41(40)	12	41	80	52	52
2. Palindrome	41(42)	6	35	-	-	48
3. Center palindrome	41(42)	8	-	36	32	-
4. Parity sort	50(54)	7	32	44	20	49
5. Faro shuffle	68(79)	9	23	-	-	41

Note. i) Experiment 3 in Khemlani et al. (2013), ii) Experiment 1 in Bucciarelli et al. (2018) for programs described without loops, iii) for programs described with loops, and iv) Mackiewicz et al. (2016) (n = 23)

number of words in Lisp programs with *while*-loops and in their translations into English.

Recursion in natural language

Our participants described their programs for primitive recursive functions in their native tongues of English, Italian, and Polish. These natural languages suffice to describe informal programs. An obvious connection between programs and languages concerns quantifiers. Our participants often used quantified assertions to describe moves, for example, "move *all the cars on the left side of the track* to the right track." To imagine the situations to which quantified assertions refer and to infer their consequences also depend on loops. Suppose, for example, that there are four people: Anne, Beth, Chuck, and Di, about whom it is true that Anne loves Beth, and that they all love anyone who loves someone. Most people can infer from a model of these two premises that:

Everyone loves Anne.

But, the quantified assertion can be used to make a loop of further updates to the model yielding the deductions: So, Beth loves Chuck; so, Di loves Chuck; . . . so, everyone loves Chuck; and so everyone loves everyone. In fact, individuals are much more likely to make the first step in the loop than to reach the final step (Cherubini & Johnson-Laird, 2004). Another connection between programs and quantified assertions is that if a program could determine whether any inference in the logic of quantifiers is valid or invalid, then a program could also determine whether or not any *while*-loop halts (see Boolos et al., 2007, Ch. 11).

Linguists have argued that recursion is the central component of natural language (e.g., Hauser et al., 2002). Chomsky (1957) first raised this question in relation to grammars. He argued for the inadequacy of *regular* grammars that finite-state computers could parse, that they need context-free rules and even rules sensitive to the grammatical context in which constituents occur – rules that demand linear-bounded computers to parse them. But, other linguists discovered that push-down computers sufficed to parse sentences and to compose their meanings from those of their constituents – tasks that had hitherto thought to demand linear-bounded computers (e.g., Gazdar et al., 1985). Indeed, grammars ought to yield syntactic analyses that enable the meanings of sentences to be composed in this way (e.g., Partee, 2014).

In English, sentences based on "respectively" introduce dependencies that cannot be handled with a push-down computer (Bar-Hillel & Shamir, 1960), for example:

Ann, Beth, and Cath love Ali, Ben, and Cam, respectively.

The interpretation of this sentence – whether semantic or pragmatic (Gazdar et al., 1985) – calls for the relation of *love* to hold from the first to the second member of each of these pairs:

Ann-Ali Beth-Ben Cath-Cam.

Readers should look again at the two sequences of proper nouns. Their rearrangement from the first sequence to the second sequence is an instance of our familiar friend, the faro shuffle:

```
Ann Beth Cath Ali Ben Cam ⇒ Ann Ali Beth Ben Cath Cam
```

As we showed earlier, such rearrangements in general call for a dynamic loop that moves items to and fro between two stacks, i.e., the siding and the left track of the railway computer. Its computation calls for the power of a linear-bounded computer with a working memory that is a proportion of the length of the input. Swiss-German, likewise, has grammatical relations between analogous sequences of case markings on

noun phrases and their respective verbs that cannot be handled in a context-free grammar and a push-down computer (Shieber, 1985).

In contrast, some theorists have argued against the hypothesis that recursion occurs in language. One sort of argument is: "Recursion is a mathematical self-calling function, and clearly there is no such thing in language" (Frath, 2014, p. 181). Which is akin to arguing that our participants did not devise programs to compute primitive recursive functions, because they didn't use *semi*-circular programs. Everett (2005) has claimed that Pirahã, a language of an Amazonian people, has a grammar that does not contain recursive rules, though the semantic processes of its speakers can be recursive (Everett & Gibson, 2019). The resulting controversy has been framed in terms or whether or not embedded structures occur in the grammar of Pirahã (see, e.g., Nevins et al., 2009; Sakel & Stapert, 2010). But, as we have shown, a function is primitive recursive if its computation necessitates a *for*-loop of two or more basic instructions. So, Everett is arguing, in effect, that parsing the language does not need loops.

Skepticism is warranted in some cases. If a colleague tells you that the song of a particular species of bird depends on a context-free grammar, then beware. The songs might be created from a regular grammar using at most a finite-state computer. However, such a system might also produce songs that the bird would never sing – an absence that might be difficult to discover. Greater computational power does not add new sorts of sequence to those that are well-formed, but rather prevents the construction of certain other sequences. But, it can construct more complex structures of the sort needed, say, for a compositional semantics.

A grammar can define a primitive recursive function. For example, the grammar to match left and right parentheses in algebraic expressions has these rules:

$$S \rightarrow (\quad)$$

$$S \rightarrow (S\ S)$$

The arrow in the first rule specifies that a well-formed string S can consist of a pair of matching parentheses. The second rule, which is circular, allows that S can consist of two instances of S within matching parentheses. Hence, a string such as: ((( )( )) ( )) is well-formed because it can be parsed using just the two preceding rules, and a push-down computer (see Chomsky, 1959). One moral of our investigations, however, is that such a program does not need to use the *semi*-circular grammar above. It can compute the same function using this loop instead:

```
While there is at least one parenthesis in the string:
   if the first parenthesis in the string is ')'
      then move it to the stack,
   otherwise if it is '('
      then delete it and the topmost item on the stack,
   otherwise halt.
If and only if the string and stack have no items in them
   then the original string is well-formed.
```

This parsing program exemplifies a general claim: grammars that are primitive-recursive with *semi*-circular rules are essential in the analysis of languages, but programs based on loops can compute the functions they describe. Human speakers are equipped with the equivalent of grammars for their native languages. But, it does not follow that this knowledge is represented in the brain in the form of a grammar: it could be embodied in programs. Another possibility is that rules universal to all languages are embodied in programs, and all information specific to the language is in the lexicon (Steedman, 2019).

A sensible psychological constraint on parsing natural language is that the time taken to parse a sentence should be at most proportional to some polynomial of the number of words in the sentence rather than an exponential function of it (see Johnson-Laird, 1983, Ch. 13, for a review of early efforts to devise such methods). No such constraint is plausible for the deliberations of our participants seeking to program a rearrangement – they may even fail the task. Likewise, the role of permutations in language and in rearrangements also differs. Steedman (2020) considers those permutations of a sequence of grammatical categories, as exemplified in: "These five young lads," that are also grammatical. Of course, not all 24 possible permutations of the four categories of these words are grammatical. Both within languages and over different languages, as Steedman shows, the number of permutations of a given number of categories that result in a grammatical order increases at much slower rate than the number of their possible permutations. This constraint follows from the assumptions built into his combinatory categorial grammar (Steedman, e.g., 2019), and so it is an excellent candidate for a linguistic universal. The constraint contrasts with the universal permutations that we investigated in our experiments: our participants, for example, solved with ease all rearrangements of trains containing four cars (see Table 2).

The various proposals about grammars (e.g., Stabler, 2004) and experiments on the learning of artificial grammars (Westphal-Fitch et al., 2018) have converged on similar results. Natural language has mildly context-sensitive rules of the sort in various systems, including tree-adjoining grammars (Joshi et al., 1975) and combinatory categorial grammars (Steedman, 2019). They can be parsed in a time proportional to a polynomial of the number of words in a sentence. In the

light of a refined hierarchy of computational power (Jäger & Rogers, 2012), what language and thought have in common is the need for a greater power than push-down computers and context-free grammars. Mild context-sensitivity can be handled with only slightly more power than a push-down computer, and so the power it needs falls within the power needed for thought. Speakers of a language understand its sentences rapidly, whereas their efforts to program rearrangements take much longer and may even fail.

Human working memory has a small finite capacity: it cannot cope with long inputs (see Christiansen & Chater, 2016). And, as we have seen, even the implications of a quantified assertion, such as: *They all love anyone who loves someone*, can be too burdensome to grasp in full. Deliberations about rearrangements call only for a memory bounded by the length of a train, which is less power than some linear-bounded computers. Other informal programs, for example, could call for cars to be added to a train. One sort might demand the copying of a train (the "copy" language, see Jäger & Rogers, 2012) as in:

$$\texttt{abbccc} \Rightarrow \texttt{abbcccabbccc}$$

Its computation calls for a linear-bounded computer with a working memory twice the length of the input. Yet, naïve individuals should be able to devise a program for this function. With no restrictions on the principles for rewriting inputs with additional symbols, they can need the power of a universal Turing machine (Post, 1946). Our conjecture is that the rearrangements that humans can solve require only the power of a linear-bounded computer. But, armed with a pencil and paper, or some other aid to memory, humans can even understand functions that they cannot compute or that cannot even be computed. There remains the eternal riddle of the innate roots of primitive recursion in languages (Berwick & Chomsky, 2016; Everaert et al., 2017), in plans and explanations (Johnson-Laird et al., 2004; Steedman, 2017), and in abductions and deductions. The riddle is insoluble at present because pertinent evidence is not at hand and may never be (Lewontin, 1998).

Discussion

This section starts with arguments that skeptics might make about the interpretation of our results, it then describes some of their pedagogical implications, and it concludes with answers to the three fundamental questions with which the article began.

The most general skeptical position about our investigations is that symbolic mental representations, such as mental models, do not exist (e.g., Ramsey, 2007), or that it is only the environment that constrains, affords, or situates intelligent behavior (e.g., Thelen & Smith, 1994). Whether mental representations play any causal role in thinking could be dubbed "Peirce's problem," because he was the first to refer to reasoning as akin to moving pictures in the mind (Peirce, 1931–1958, Vol. 4, paragraph 8). A more nuanced skepticism allows that mental representations exist, but that they have the form of grammatically structured strings of symbols in a language of thought (e.g., Pylyshyn, 2003), and that mental programs are formulated in such a language (Piantadosi et al., 2016). Mental models could therefore be epiphenomenal, and, as Pylyshyn argued, play no causal role in thinking. No cognitive scientists doubt that neuronal processes underlie mental life, just as electronic processes underlie digital computers executing programs. Indeed, phenomenologists have argued – as the late Paolo Bozzi often did (see also Bozzi, 1989) – that nothing other than neuronal events has a causal role, not even expressions in a language of thought. Our view, however, is that just as programmers find it expedient to devise high-level programming languages that can manipulate symbolic arrays, the brain has evolved to do so too. High-level representations play a causal role. Pertinent evidence includes our finding that children's gestures help them to abduce programs. Why don't they just imagine gesturing if all that matters are neuronal processes? Hence, until a theory making no use of kinematic models leads to alternative and corroborated predictions about how individuals devise programs, the controversy is not open to empirical resolution.

A more cogent criticism concerns Kolmogorov's measure of complexity, which depends on the number of words in programs in a standard language for programming. The measure yields finer predictions of difficulty than, say, computational power (see Table 1). Its trend predictions were borne out in our experiments on abducing programs (Table 4) and on deducing their consequences (Table 5). But, these experiments used only a handful of rearrangements, and so a legitimate criticism is that there could be others for which K-complexity makes erroneous predictions. Nevertheless, what remains secure is that naive individuals are able to program at least some primitive recursive functions.

Could our participants have abduced programs without themselves simulating loops of moves, and instead relied on some – as yet unspecified – short cut? When the great mathematician Gauss was a boy, he is said to have taken an astonishing short cut to avoid a cumbersome *for*-loop. To sum the numbers from 1 to 100, he took their mean of 50.5, and multiplied it by 100 to calculate the correct total of 5,050 (see Anderson et al., 2011, for experimental studies of a similar sort). No analogous short cut, however, yields programs for rearrangements, because they have to use the three basic instructions for moving cars to the siding (S), to left track (L), or to right track (R). So, we see no feasible alternative to participants having to simulate loops of instructions, which they then use in their programs for computing rearrangements.

Individual participants differed in ability, and not everyone was able to formulate a program for a rearrangement. And not every rearrangement was easy. Only a few exceptional individuals were able to abduce a correct program for a faro shuffle of trains, which splits a train in half, and interleaves the cars from the two halves (see columns vi and vii in Table 4). Yet, a common informal program that children in the West learn is how to lay place settings at a table. Given a supply of cutlery, they can make each setting of fork, knife, spoon, etc. A corresponding program given sets of cutlery rearranges them in the appropriate order. Like the faro shuffle, it calls for moving items between two stacks.

The differences from one individual to another in informal programming show that computer programming is a skill. Aptitude is needed to devise self-referential *semi*--circular programs (e.g., Rubio-Sanchez, 2017), and children have difficulty in coping with them (e.g., Dicheva & Close, 1996; Kurland & Pea, 1985). Yet, they do not have so much difficulty in devising loops of instructions. Other studies of novices devising programs both in formal programming languages (e.g., Anderson et al., 1988; Soloway & Spohrer, 2013) and in natural language (e.g., Good & Howland, 2017; Miller, 1981) inspired a search for tests to measure potential ability in programming (e.g., Bornat et al., 2008). Computational thinking ought to be taught in schools (Grover & Pea, 2013; Wing, 2008). But, no consensus exists about its nature (see, e.g., Bundy, 2007; Cetin & Dubinsky, 2017; Denning, 2017; Zhong et al., 2015). It should be independent of any particular programming language, and our present investigations imply that its roots lie in abducing programs for primitive recursive functions using loops of repeated instructions. The abduction of rearrangements may therefore reveal a person's potential as a computer programmer.

Conclusions

We began this article with three questions. We end it with our answers to them.

What is recursion? In its fundamental sense, recursion concerns the definitions of functions, and is of two sorts. *Primitive* recursion defines a function's value in a *semi*-circular way, i.e., its value for an input of zero, and its value for (n + 1) based on its self-referential value for n. *Minimization* is a more powerful recursion, but no-one is likely to be able to envisage it for more than tiny input values (see Appendix 1). From an appropriate set of basic functions, their composition and use in recursive definitions suffice – as far as anyone knows – to define any computable function. Programs for computing recursive functions can use the same sort of *semi*-circular formulations, but they don't need to. They can also be computed using *while*-loops; and primitive recursive functions can even be computed using *for*-loops. They include programs that rearrange the order of cars in trains using the railway computer (see Figs. 1 and 2).

How could naive individuals create informal programs to compute (primitive) recursive functions? According to the model theory and its implementation in the mAbducer program, they can use kinematic mental models to simulate the movements of cars on a railway. They solve rearrangements of the order of cars using *partial* means-ends analysis, which satisfies the goal in successive parts rather than dealing with it as a whole (cf. Newell, 1990). As the theory predicts, the greater the number of moves needed to solve a rearrangement, the more difficult its solution is to discover (see Tables 2 and 3). For instance, it is harder to reverse the order of cars in a train than to move all the cars in odd-numbered positions in the train in front of all the cars in its even-numbered positions (a parity sort). Given solutions to a primitive recursive rearrangement, individuals can in some cases abduce a program to solve it for trains of any length: they find a loop in its solutions. They never use *semi*-circular programs, but infer the conditions that must hold for the repetition of a *while*-loop or the required number of repetitions needed for a *for*-loop based on the length of the train. The difficulty of abductions depends, not on the number of moves they call for, but on their complexity of the required program (Table 1) for which Kolmogorov complexity – the number of words required in a program – seems to be a predictive metric (see Table 4). Ten-year-old children cope with a reversal of the order of the cars in a train, which calls for a working memory akin to a stack – a siding that accommodates no more than the number of cars in a train. More demanding is the faro shuffle that calls for two stacks – the siding and left track. Naive individuals can also deduce the consequences of rearrangement programs by simulating each instruction in a kinematic mental model, and again the difficulty of the task depends on the complexity of the program (see Table 5).

How does the computational power needed for informal programming relate to the power needed for natural language? Rearrangements are primitive recursive, but their programs differ in the power of the computer needed to carry them out – a matter that depends on the nature of the computer's working memory. The programs that our participants devised included those that swap the order of adjacent cars in a train and that use the siding only to store single cars (a finite-state computer), those that reverse the order of the cars and that use the siding as a stack-like memory (a push-down

computer), and those that make a faro shuffle of the cars and that use both the siding and left track as stack-like memories (a linear-bounded computer). None of these rearrangements or any others call for an unlimited length of track needed to compute a minimization: no cars are added or removed from a train in a rearrangement, and so working memory never has to exceed the length of a train.

Human beings appear to have a single working memory, albeit one with several components (e.g., Baddeley et al., 2019; Malmberg et al., 2019). It may have specialized linguistic components, but language and thought appear to share working memory. People can talk while they drive, but they are liable to shut up during difficult maneuvers. And, when people speak spontaneously, they are more likely to err in a concurrent tracking task akin to driving, when they produce a relative clause that is more demanding on working memory (Power, 1986). This interference supports the idea of a working memory in common. Many natural languages have mildly context-sensitive grammars that call for more power than a push-down computer, and some languages have grammars that demand rearrangements equivalent to a faro shuffle. The burden of our research is therefore that the working memory for thought is more than enough for language. And if long-term memory, writing, and other external devices, can aid working memory, individuals can even think about problems that call for maximal computational power.

Appendix 1. Minimization, the Ackermann function, and *while*-loops

Minimization is a more powerful way than primitive recursion to combine given functions. We can illustrate how it works with a function that outputs the largest whole number not larger than half a given integer, for example, 7 to an output of 3. Minimization in general maps to an output equal to the minimal value of an input to a given function for yielding an output of zero. So, this function is the minimization of y in the formula: x – 2y – 1, where x is the input value. *While*-loops can be used to compute any minimizations, and so this function can be computed in the following way:

```
Define no-more-than-half (x)

Set y = 0

While (x - 2y - 1) > 0, set y = (y + 1)

Output y.
```

If the input value of x equals 7, the program increases the value of y while the formula yields a value greater than 0:

$y = 0$, the formula yields 6
$y = 1$, the formula yields 4
$y = 2$, the formula yields 2
$y = 3$, the formula yields 0

So, now the program outputs 3, which is the highest integer not larger than half of 7. The definition above shows the existing arithmetical functions that are combined in the computation of this particular minimization.

In fact, not-more-than-half is primitive recursive – it doesn't require a minimization. But, other functions depend on 1it, because primitive recursion cannot define them. The best known is the Ackermann function (Ackermann, 1967/1928). It maps two input integers to an output integer, and these examples show that its output grows at a faster rate than exponential:

```
Ackermann (1, 1) = 3

Ackermann (0, 4) = 5

Ackermann (3, 0) = 5

Ackermann (2, 2) = 7

Ackermann (3, 3) = 61

Ackermann (4, 1) = 65533

Ackermann (4, 2) = 2^65536 - 3
```

The final output here is an astronomical number. The definition of the function depends in part on double *semi*-circles, for example:

```
Ackermann (1, 1) =

Ackermann (0, Ackermann (1, 0)) =

Ackermann (0, Ackermann (0, 1)) =

Ackermann (0, 2) = 3
```

Because its computations apply to decreasing values of its two inputs, it always has an output.

But, because of the doubly recursive calls when both inputs are greater than zero, it calls for great computational power equivalent to unlimited working memory (as in a Universal Turing machine). Minimization is arcane, but it can be computed using *while*-loops, in which a sequence of instructions is repeated while a particular condition holds. Hence, a program for computing the Ackermann function can be based on *while*-loops (Grossman & Zeitman, 1988). Minimization is hardly likely to play any role in using natural languages, and it is an open question whether the brain – even of calculating prodigies – ever computes functions that only minimization can define.

Appendix 2: The transformation of the railway into a universal Turing machine

The railway, as we have defined it in the text, cannot be used to compute addition, multiplication, subtraction, or division. However, granted a few modifications, the user can carry out programs for these arithmetical functions. There need to be two sorts of cars (labeled 1 or 0), and if need be, additional cars can be added to an existing train: so, the railway should never run out of cars, or of track for accommodating them. The rest is up to the user. She has to be able to carry out three actions:

1. To identify the car at a particular fixed location on left track as 1 or 0.
2. To remove a car or to add a new car at this location, where the conjunction of both these actions replaces one sort of car with the other sort,
3. To move the whole train one car to the left of this location or one car to the right of it– a move that may call for a lengthening of the track.

She also has to keep in mind the values of two other numbers, which are needed for programs. These numbers denote an abstract state of the railway prior to executing an instruction and after executing it. There are only finitely many states. Each instruction in the program consists of four numbers, which represent:

- the identity of the car at the particular location as 1 or 0.
- the number of the present state that the user is in.
- the number of the basic action for the user to carry out from the three described above.
- the number of the next state that the user is in after carrying out the instruction.

Any program consists of a set of these quadruples.

The program to add any two positive integers, such as 2 + 1, has the following instructions:

Present state	Symbol on car	Action carried out	Next state
1	1	Change car to 0	1
1	0	Move train one car to right	2
2	1	Move train one car to right	2
2	0	Change car to 1	3
3	1	Move train one car to left	3
3	0	Move train one car to right	4

Given a train for 2 + 1, i.e.: 01011, the effect of the program is as follows, where the arrow denotes the car that is scanned:

↓	Current state
01011	1
01010	1
01010	2
01010	2
01110	3
01110	3
01110	3
01110	4

At this point, the computation halts, because there are no instructions for state 4. The representation of numbers uses numerals such as 111 to denote 3. The numbers denoting states allow loops to occur when a state is revisited after a loop of other states.

These modifications use a train to replace the tape of a Turing machine (see Davis, 1958). Turing's great insight was that a particular program, such as the one above, for computing a certain function can itself be encoded as the sequence of digits representing each of its quadruples, and so a universal machine fed in a tape with this numeral and data can carry out the computations of the particular machine on the data. That is the origin of the programmable digital computer.

This modification to the railway puts it at the top of the Chomsky hierarchy. It can compute minimizations, such as the Ackermann function (see Appendix 1), though the extensions to the train and track would soon exhaust the world's supply of rolling stock and railways, and the demands on the user would soon become intractable.

Appendix 3: The eight programs in our studies

Table 6 The eight programs in our empirical investigations, the computational power needed to execute them (finite-state, push-down, or linear-bounded computers), mAbducer's *for*-loops and *while*-loops (in Lisp notation) and its English translations of *while*-loops, and the K-complexities of the *while*-loops and of their translations

Programs with *for*-loops	Programs with *while*-loops	Translations of *while*-loops
1. Swap adjacents abcdef ⇒ badcfe	K-complexity: 37 words	K-complexity: 38 words
Power: finite-state computer		
(defun swap(track)	(defun swap (track)	While there are more than zero cars on the left track,
(let* ((len (length (first track)))	(let ((len (length (first track))))	move one car to the siding,
(n-of-reps (+ (* 1/2 len) 0)))	(loop while (> (length (first track)) 0)	move one car to the right track,
(loop for i from 1 to n-of-reps	do (setf track (S 1 track))	move one car to the left track,
do (setf track (S 1 track))	(setf track (R 1 track))	move one car to the right track.
(setf track (R 1 track))	(setf track (L 1 track))	
(setf track (L 1 track))	(setf track (R 1 track)))	
(setf track (R 1 track)))	track))	
track))		
2. Reversal abcdef ⇒ fedbca	K-complexity: 41	K-complexity: 40 words
Power: push-down computers		
(defun reverse (track)	(defun reverse (track)	Move one less than the cars to the siding.
(let* ((len (length (first track)))	(let ((len (length (first track))))	Move one car to the right track.
(n-of-reps (+ (* 1 len) -1)))	(setf track (S (+ (* 1 len) -1) track))	While there are more than zero cars on the siding,
(setf track (S (+ (* 1 len) -1) track))	(setf track (R 1 track))	move one car to the left track,
(setf track (R 1 track))	(loop while (> (length (second track)) 0)	move one car to the right track.
(loop for i from 1 to n-of-reps	do (setf track (L 1 track))	
do (setf track (L 1 track))	(setf track (R 1 track)))	
(setf track (R 1 track)))	track))	
track))		
3. Palindrome abcdef ⇒ afbecd	K-complexity: 41	K-complexity: 42 words
Power: push-down computer		
(defun palind (track)	(defun palind (track)	Move one less than half the cars to the siding.
(let* ((len (length (first track)))	(let ((len (length (first track))))	Move two cars to the right track.
(n-of-reps (+ (* 1/2 len) -1)))	(setf track (S (+ (* 1/2 len) -1) track))	While there are more than zero cars on the left track,
(setf track (S (+ (* 1/2 len) -1) track))	(setf track (R 2 track))	move one car to the left track,
(setf track (R 2 track))	(loop while (> (length (first track)) 0)	move two cars to the right track.
(loop for i from 1 to n-of-reps	do (setf track (L 1 track))	
do (setf track (L 1 track))	(setf track (R 2 track)))	
(setf track (R 2 track)))	track))	
track))		
4. Center palindrome abcde ⇒ aebdc	K-complexity: 41	K-complexity: 42 words
Power: push-down computer		
(defun center-palind (track)	(defun center-palind (track)	Move half less than half the cars to the siding.
(let* ((len (length (first track)))	(let ((len (length (first track))))	Move one car to the right track.
(n-of-reps (+ (* 1/2 len) -1/2)))	(setf track (S (+ (* 1/2 len) -1/2) track))	While there are more than zero cars on the left track,
(setf track (S (+ (* 1/2 len) -1/2) track))	(setf track (R 1 track))	move one car to the left track,
(setf track (R 1 track))	(loop while (> (length (first track)) 0)	move two cars to the right track.
(loop for I from 1 to n-of-reps	do (setf track (L 1 track))	
do (setf track (L 1 track))	(setf track (R 2 track)))	
(setf track (R 2 track)))	track))	
track))		
5. Make palindrome abcdef ⇒ acefdb	K-complexity: 45 words	K-complexity: 44 words
Power: push-down computer		
(defun make-pal (track)	(defun make-pal (track)	Move two less than the cars to the siding.
(let* ((len (length (first track)))	(let ((len (length (first track))))	While there are more than zero cars on the siding,
(n-of-reps (+ (* 1/2 len) -1)))	(setf track (S (+ (* 1 len) -2) track))	move one car to the right track,
(setf track (S (+ (* 1 Len) -2) track))	(loop while (> (Length (Second track)) 0)	move two cars to the left track.
(loop for i from 1 to n-of-reps	do (setf track (R 1 track))	Move one more than half the cars to the right track.

Table 6 (continued)

Programs with *for*-loops	Programs with *while*-loops	Translations of *while*-loops
do (setf track (R 1 track))	(setf track (L 2 track)))	
(setf track (L 2 track)))	(setf track (R (+ (* 1/2 len) 1) track))	
(setf track (R (+ (* 1/2 len) 1) track))	track))	
track))		
6. Two loop palindrome	K-complexity: 40 words	K-complexity: 48 words
abcdef ⇒ cbafed		
Power: push-down computer		
(defun two-loops (track)	(defun two-loops (track)	While there are more than zero cars on the left track,
(let* ((len (length (first track)))	(loop while (> (length (first track)) 1)	move one car to the siding,
(n-of-reps (* 1/2 len)))	do (setf track (S 1 track))	move one car to the right track.
(loop for i from 1 to n-of-reps	(setf track (R 1 track)))	While there are more than zero cars on the siding,
do (setf track (S 1 track))	(loop while (> (length (second track)) 0)	move one car to the left track,
(setf track (R 1 track)))	do (setf track (L 1 track))	move one car to the right track.
(loop for i from 1 to n-of-reps	(setf track (R 1 track)))	
do (setf track (L 1 track))	track)	
(setf track (R 1 track)))		
track))		
7. Parity sort	K-complexity: 50	K-complexity: 54 words
abcdef ⇒ acebdf		
Power: push-down computer		
(defun parity (track)	(defun parity (track)	Move one car to right track.
(let* ((len (length (first track)))	(let ((len (length (first track))))	While there is more than one car on left track
(n-of-reps (+ (* 1/2 len) -1)))	(setf track (R 1 track))	move one car to siding,
(setf track (R 1 track))	(loop while (> (length (first track)) 1)	move one car to right track.
(loop for i from 1 to n-of-reps	do (setf track (S 1 track))	Move one less than half the number of cars in the train to left track.
do (setf track (S 1 track))	(setf track (R 1 track)))	Move half the number of cars in the train to right track.
(setf track (R 1 track)))	(setf track (L (+ (* 1/2 len) -1) track))	
(setf track (L (+ (* 1/2 len) -1) track))	(setf track (R (+ (* 1/2 len) 0) track))	
(setf track (R (+ (* 1/2 len) 0) track))	track))	
track))		
8. Faro-in	K-complexity: 68 words	K-complexity: 79 words
abcdef ⇒ adbecf		
Power: linear-bounded computer		
(defun faro-in (track)	(defun faro-in (track)	Set n-of-s, to one less than half the cars.
(let* ((len (length (first track)))	(let* ((len (length (first track)))	Set decrement-s to one.
(n-of-reps (+ (* 1/2 len) -1)))	(n-of-S (+ (* 1/2 len) -1))	Set n-of-l, to one less than half the cars.
(loop for i from 1 to n-of-reps	(decrement-S 1)	Set decrement-l to one.
do (setf track (R 1 track))	(n-of-L (+ (* 1/2 len) -1))	While there are more than two cars on the left track,
(setf track (S (+ (* 1 n-of-reps)	(decrement-L 1))	move one car to the right track,
(* i -1) 1) track))	(loop while (> (length (first track)) 2)	move n-of-s cars to the siding,
(setf track (R 1 track))	do (setf track (R 1 track))	move one car to the right track,
(setf track (L (+ (* i N-of-reps)	(setf track (S n-of-S track))	move n-of-l cars to the left track,
(* i -1) 1) track)))	(setf track (R 1 track))	take decrement-s from n-of-s,
(setf track (R 2 track))	(setf track (L n-of-L track))	take decrement-l from n-of-l.
track))	(setf n-of-S (- n-of-S decrement-S))	Move two cars to the right track.
	(setf n-of-L (- n-of-L decrement-L)))	
	(setf track (R 2 track))	
	track))	

Acknowledgements For their help and advice, we thank Geoff Goodwin, Philipp Koralus, David Lobina, Salvador Mascarenhas, Adam Moore, Marco Ragni, Mark Steedman, and Greg Trafton. We also thank Gary Lupyan, Mark Steedman, and three anonymous referees for their helpful criticisms of an earlier draft. The second author thanks Compagnia di San Paolo for support Grant "Smaile – simple methods for artificial intelligence learning and education." The third author thanks the Polish National Science Centre for support in Grant 2014/14/M/HS6/00916. We have no known conflicts of interest to disclose.

References

Ackermann, W. (1967/1928). On Hilbert's construction of the real numbers. In van Heijenoort, J. (Ed.) *From Frege to Gödel: A source book in mathematical logic, 1879-1931* (pp. 495-507.) Harvard University Press. (Originally published in 1928.)

Adams, R. (2011). *An early history of recursive functions and computability: from Gödel to Turing*. Docent Press. (An edited version of his 1983 Ph.D. thesis.)

Aho, A. V., & Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling,* Vol. 1: *Parsing*. Prentice-Hall.

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

Anderson, J. R., Betts, S., Ferris, J. L., & Fincham, J. M. (2011). Cognitive and metacognitive activity in mathematical problem solving: prefrontal and parietal patterns. *Cognitive, Affective, & Behavioral Neuroscience, 11*, 52-67.

Baddeley, A. D., Hitch, G. J., & Allen, R. J. (2019). From short-term store to multicomponent working memory: The role of the modal model. *Memory & Cognition*, *47*, 575-588.

Bar-Hillel, Y., & Shamir, E. (1960). Finite-state languages: formal representation and adequacy problems. *The Bulletin of the Research Council of Israel*, *8f*(3), 155-166.

Berwick, R. C., & Chomsky, N. (2016). *Why only us: Language and evolution*. MIT Press.

Bóna, M. (2012). *Combinatorics of permutations*. (2nd Ed.). Chapman & Hall.

Bóna, M. (2019). A survey of stack sortable permutations. Ch. 4 in *50 years of Combinatorics, Graph Theory, and Computing*. Chung, F. et al. (Eds.) (Pp. 55 in ebook). Chapman & Hall.

Boolos, G.S., Burgess, J.P., & Jeffrey, R.C. (2007). *Computability and logic*. (5th Ed.). Cambridge University Press.

Bornat, R., Dehnadi, S., & Simon (2008). Mental models, consistency and programming aptitude. *Proceedings of the Tenth Australasian Computing Education Conference* (ACE 2008), *78*, 53–61.

Bozzi, P. (1989). *Fenomenologia sperimentale*. Il Mulino.

Bucciarelli, M., Mackiewicz, R., Khemlani, S. S., & Johnson-Laird, P. N. (2016). Children's creation of algorithms: simulations and gestures. *Journal of Cognitive Psychology*, *28*, 297-318.

Bucciarelli, M., Mackiewicz, R., Khemlani, S. S., & Johnson-Laird, P. N. (2018). Simulation in children's conscious recursive reasoning. *Memory & Cognition, 46,* 1302-1314.

Bundy, A. (2007). Computational thinking is pervasive. *Journal of Scientific and Practical Computing, 1*, 67-69.

Cetin, I., & Dubinsky, E. (2017). Reflective abstraction in computational thinking. *Journal of Mathematical Behavior*, *47*, 70-80.

Chaitin, G.J. (1998). *The limits of mathematics*. Springer.

Cherubini, P., & Johnson-Laird, P. N. (2004). Does everyone love everyone? The psychology of iterative reasoning. *Thinking & Reasoning*, *10*, 31–53.

Chomsky, N. (1957). *Syntactic structures*. Mouton.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, *2*, 137-167.

Christiansen, M. H., & Chater, N. (2016). The now-or-never bottleneck: A fundamental constraint on language. *Behavioral and brain sciences*, *39*, 1-19.

Corballis, M. (2011). *The recursive mind: The origins of human language, thought, and civilization.* Princeton University Press.

Davis, M. (1958). *Computability and unsolvability*. McGraw-Hill.

Denning, P. J. (2017). Remaining trouble spots with computational thinking. Addressing unresolved questions concerning computational thinking. *Communications of the Association for Computing Machinery, 60*, 33-39.

Diaconis, P., Graham, R. L., & Kantor, W. M. (1983). The mathematics of perfect shuffles. *Advances in Applied Mathematics*, *4*, 175–196.

Dicheva, D., & Close, J. (1996). Mental models of recursion. *Journal of Educational Computing Research, 14*, 1-23.

Douven, I. (2017). Abduction. In: *The Stanford Encyclopedia of Philosophy* (Summer 2017), Zalta, E. N. (Ed.) https://plato.stanford.edu/archives/sum2017/entries/abduction/

Evans, J. S. B. (2008). Dual-processing accounts of reasoning, judgment, and social cognition. *Annual Review of Psychology*, *59*, 255-278.

Everaert, M. B., Huybregts, M. A., Berwick, R. C., Chomsky, N., Tattersall, I., Moro, A., & Bolhuis, J. J. (2017). What is language and how could it have evolved? *Trends in Cognitive Sciences*, *21*, 569-571.

Everett, D. L. (2005). Cultural constraints on grammar and cognition in Pirahã. *Current Anthropology*, *46*, 621-646.

Everett, D. L., & Gibson, E. (2019). Recursion across domains ed. by Luiz Amaral et al. *Language*, *95*, 777-790.

Ferrigno, S., Cheyette, S. J., Piantadosi, S. T., & Cantlon, J. F. (2020). Recursive sequence generation in monkeys, children, US adults, and native Amazonians. *Science Advances*, *6*, eaaz1002.

Frath P. (2014) There is no recursion in language. In: Lowenthal F., & Lefebvre L. (eds), *Language and Recursion*. (Pp. 181-191). Springer.

Garey, M., & Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman.

Gazdar, G., Klein, E., Pullum, G. K., & Sag, I. A. (1985). *Generalized phrase structure grammar*. Harvard University Press.

Gödel, K. (1967/1931). On formally undecidable propositions of *Principia Mathematica* and related systems. In van Heijenoort, J. (Ed.), *From Frege to Gödel: A source book in mathematical logic, 1879-1931* (pp. 596-616). Harvard University Press. (Originally published in 1931.)

Good, J., & Howland, K. (2017). Programming language, natural language? Supporting the diverse computational activities of novice programmers. *Journal of Visual Languages & Computing*, *39*, 78-92.

Grossman, J. W., & Zeitman, R. S. (1988). An inherently iterative computation of Ackermann ' s function. *Theoretical Computer Science*, *57*, 327-330.

Grover, S., & Pea, R. (2013). Computational thinking in k12: A review of the state of the field. *Educational Researcher, 42*, 38-43.

Halford, G. S., Wilson, W. H., & Phillips, S. (2010). Relational knowledge: the foundation of higher cognition. *Trends in Cognitive Sciences*, *14*, 497-505.

Hauser, M. D., Chomsky, N., & Fitch, W. T. (2002). The faculty of language: what is it, who has it, and how did it evolve? *Science*, *298*, 1569-1579.

Hegarty, M., Stieff, M., & Dixon, B. L. (2013). Cognitive change in mental models with experience in the domain of organic chemistry. *Journal of Cognitive Psychology*, *25*, 220–228.

Hopcroft, J.E., & Ullman, J.D. (1979). *Introduction to automata theory, languages, and computation*. Addison-Wesley.

Inhelder, B., and Piaget, J. (1958). *The Growth of Logical Thinking from Childhood to Adolescence*. Routledge & Kegan Paul.

Jäger, G., & Rogers, J. (2012). Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society (London), Series B*, *367*, 1956–1970.

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

Johnson-Laird, P. N. (2006). *How we reason*. Oxford University Press.

Johnson-Laird, P. N., Girotto, V., & Legrenzi, P. (2004). Reasoning from inconsistency to consistency. *Psychological Review*, *111*, 640-661.

Joshi, A. K., Levy, L. S., & Takahashi, M. (1975). Tree adjunct grammars. *Journal of Computer and System Sciences*, *10*, 136-163.

Kahneman, D. (2011). *Thinking fast and slow*. Farrar, Strauss, Giroux.

Kendall, M., & Gibbons, J.D. (1990). *Rank Correlation Methods*. 5th Ed. Oxford University Press.

Khemlani, S., & Johnson-Laird, P. N. (2021). Reasoning about properties: A computational theory. *Psychological Review*, in press.

Khemlani, S., Mackiewicz, R., Bucciarelli, M., & Johnson-Laird, P.N. (2013). Kinematic mental simulations in abduction and deduction. *Proceedings of the National Academy of Sciences of the United States of America, 110*, 16766-16771.

Khemlani, S., Orenes, I., & Johnson-Laird, P. N. (2014). The negations of conjunctions, conditionals, and disjunctions. *Acta Psychologica*, *151*, 1-7.

Khemlani, S., Lotstein, M., & Johnson-Laird, P. N. (2015). Naive probability: Model-based estimates of unique events. *Cognitive Science*, *39*, 1216–1258.

Knuth, D. (1997). *The art of computing*. Vol. 1: *Fundamental algorithms*. (3rd Ed.). Addison-Wesley.

Kolmogorov, A. (1965). Three approaches to the quantitative definition of information. *Problems of Information and Transmission, 1*, 1-7.

Kurland, D.M., & Pea, R.D. (1985). Children's mental models of recursive Logo programs. *Journal of Educational Computing Research, 1*, 235-243.

Lake, B. M., & Piantadosi, S. T. (2020). People infer recursive visual concepts from just a few examples. *Computational Brain & Behavior*, *3*, 54-65.

Lee, N. Y. L., & Johnson-Laird, P. N. (2013a). Strategic changes in problem solving. *Journal of Cognitive Psychology, 25,* 165-173.

Lee, N. Y. L., & Johnson-Laird, P. N. (2013b). A theory of reverse engineering and its application to Boolean systems. *Journal of Cognitive Psychology*, *25*, 365-389.

Lee, N. Y. L., Goodwin, G. P., & Johnson-Laird, P. N. (2008). The psychological problem of Sudoku. *Thinking & Reasoning*, 14, 342-364.

Lehmer, D. H. (1949). Methods in large-scale units. *Proceedings of a second symposium on large-scale digital calculating machinery* (pp. 141-146). Harvard University Press.

Lewontin, R. (1998). The evolution of cognition: Questions we will never answer. In Osherson, D. N., Scarborough, D., & Sternberg, S. (Eds). *An invitation to cognitive science*, *Vol. 4: Methods, models, and conceptual issues*. (Pp. 107–132). MIT Press.

Lowrie, T., Logan, T., & Hegarty, M. (2019). The influence of spatial visualization training on students' spatial reasoning and mathematics performance. *Journal of Cognition and Development*, *20*, 729-751.

Mackiewicz, R., Johnson-Laird, P., Khemlani, S., & Bucciarelli, M. (2016). Deductions from algorithms as mental simulations. Osf.io/bke3m.

Malmberg, K. J., Raaijmakers, J. G., & Shiffrin, R. M. (2019). 50 years of research sparked by Atkinson and Shiffrin (1968). *Memory & Cognition*, *47*, 561-574.

Manktelow, K. (2021). *Beyond Reasoning: The Life, Times and Work of Peter Wason, Pioneering Psychologist*. Routledge.

Marr, D. (1982). *Vision*. Freeman.

Melzak, Z. A. (1961). An informal arithmetical approach to computability and computation. *Canadian Mathematical Bulletin*, *4*, 279-293.

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

Miller, G. A., & Johnson-Laird, P. N. (1976). *Language and perception*. Belknap, Harvard University Press.

Nevins, A., Pesetsky, D., & Rodrigues, C. (2009). Pirahã exceptionality: A reassessment. *Language*, *85*, 355-404.

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

Nielsen, M., & Chuang, I. (2000). *Quantum computation and quantum information*. Cambridge University Press.

Oaksford, M., & Chater, N. (2020). New paradigms in the psychology of reasoning. *Annual Review of Psychology*, *71*, 12.1–12.26.

Partee, B. H. (2014). A brief history of the syntax-semantics interface in western formal linguistics. *Semantics-Syntax Interface*, *1*, 1-20.

Peirce, C. S. (1931-1958). *Collected papers of Charles Sanders Peirce*. (Vols. 1-8). Hartshorne, C., Weiss, P., & Burks, A. (Eds.). Harvard University Press.

Peirce, C. S. (1955). *Philosophical writings of Peirce*, Buchler, J. (Ed.). Dover.

Piantadosi, S. T., Tenenbaum, J. B., & Goodman, N. D. (2016). The logical primitives of thought: Empirical foundations for compositional cognitive models. *Psychological Review*, *123*, 392.

Pinker, S., & Jackendoff, R. (2005). The faculty of language: what's special about it? *Cognition, 95*, 201–236.

Post, E. (1946). A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, *52*, 264-268.

Power, M. J. (1986). A technique for measuring processes load during speech production. *Journal of Psycholinguistic Research, 15*, 371-382.

Prusinkiewicz, P., & Lindenmayer, A. (1990). *The algorithmic beauty of plants*. Springer-Verlag.

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

Ramsey, F. P. (1990/1926). Truth and probability. In: Philosophical papers. (Ed. Mellor, D. H.). Cambridge University Press. (Originally published 1926.)

Ramsey, W. M. (2007). *Representation reconsidered*. MIT Press.

Rogers, H. (1967). *Theory of recursive functions and effective computability*. McGraw-Hill.

Rubio-Sanchez, M. (2017). *Introduction to recursive programming*. CRC Press.

Sakel, J., & Stapert, E. (2010). Pirahã – in need of recursive syntax?. In Hulst, H. V. D. (Ed.), *Recursion and human language* (Pp. 3-16). De Gruyter.

Shieber, S. (1985). Evidence against the context-freeness of natural language. *Linguistics and Philosophy, 8*, 333–343.

Soloway, E., & Spohrer, J. C. (Eds.). (2013). *Studying the novice programmer*. Psychology Press.

Stabler, E. P. (2004). Varieties of crossing dependencies: structure dependence and mild context sensitivity. *Cognitive Science, 28*, 699–720.

Steedman, M. (2017). The emergence of language. *Mind & Language*, *32*, 579–590.

Steedman, M. (2019). Combinatory categorial grammar. In Kertész, A., Rákosi, E.M., & Rákosi, C. (Eds.) *Current Approaches to Syntax: A Comparative Handbook.* (Pp. 389–420). De Gruyter Mouton.

Steedman, M. (2020). A formal universal of natural language grammar. *Language*, *96*, 618-660.

Thelen, E., & Smith, L. B. (1994). *A dynamic systems approach to the development of cognition and action*. MIT Press.

Vicari, G., & Adenzato, M. (2014). Is recursion language-specific? Evidence of recursive mechanisms in the structure of intentional action. *Consciousness and Cognition*, *26*, 169-188.

Westphal-Fitch, G., Giustolisi, B., Cecchetto, C., Martin, J. S., & Fitch, W. (2018). Artificial grammar learning capabilities in an abstract visual task match requirements for linguistic syntax. *Frontiers in Psychology*, *9*, 1210.

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society, 366*, 3717-3725.

Zhong, B., Wang, Q., Chen, J., & Li, Y. (2015). An exploration of three-dimensional integrated assessment for computational thinking. *Journal of Educational Computing Research*, *53*, 562-590.