EXCERPTED FROM

STEPHEN WOLFRAM
A NEW KIND OF SCIENCE

SECTION 3.9

Register Machines

Register Machines

All of the various kinds of systems that we have discussed so far in this chapter can readily be implemented on practical computers. But none of them at an underlying level actually work very much like typical computers. Register machines are however specifically designed to be very simple idealizations of present-day computers.

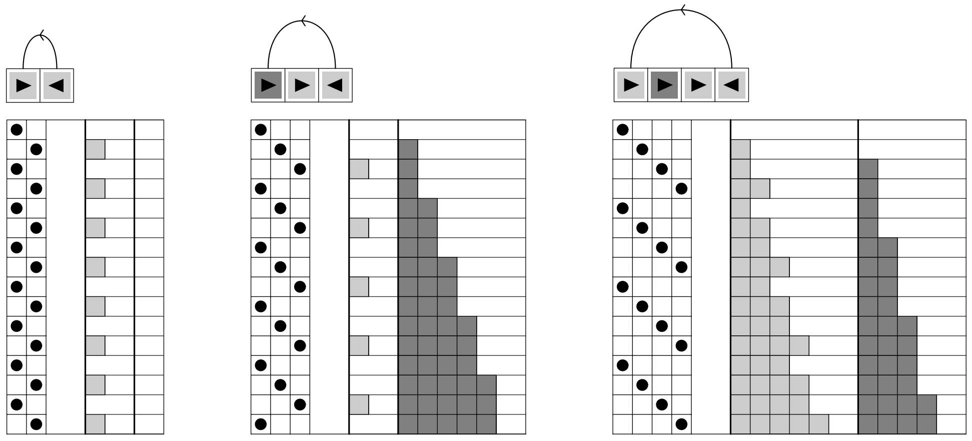
Under most everyday circumstances, the hardware construction of the computers we use is hidden from us by many layers of software. But at the lowest level, the CPUs of all standard computers have registers that store numbers, and any program we write is ultimately converted into a sequence of simple instructions that specify operations to be performed on these registers.

Most practical computers have quite a few registers, and support perhaps tens of different kinds of instructions. But as a simple idealization one can consider register machines with just two registers—each storing a number of any size—and just two kinds of instructions: "increments" and "decrement-jumps". The rules for such register machines are then idealizations of practical programs, and are taken to consist of fixed sequences of instructions, to be executed in turn.

Increment instructions are set up just to increase by one the number stored in a particular register. Decrement-jump instructions, on the other hand, do two things. First, they decrease by one the number in a particular register. But then, instead of just going on to execute the next instruction in the program, they jump to some specified other point in the program, and begin executing again from there.

Since we assume that the numbers in our registers cannot be negative, however, a register that is already zero cannot be decremented. And decrement-jump instructions are then set up so that if they are applied to a register containing zero, they just do essentially nothing: they leave the register unchanged, and then they go on to execute the next instruction in the program, without jumping anywhere.

This feature of decrement-jump instructions may seem like a detail, but in fact it is crucial—for it is what makes it possible for our register machines to take different paths depending on values in registers through the programs they are given.



Examples of simple register machines, set up to mimic the low-level operation of practical computers. The machines shown have two registers, whose values on successive steps are given on successive lines down the page. Each machine follows a fixed program given at the top. The program consists of a sequence of increment ▸ and decrement-jump ◂ instructions. Instructions that are shown as light gray boxes refer to the first register; those shown as dark gray boxes refer to the second one. On each line going down the page, the black dot on the left indicates which instruction in the program is being executed at the corresponding step. With the particular programs shown here, each machine just executes successive instructions in turn, jumping to the beginning again when it reaches the end of the program.

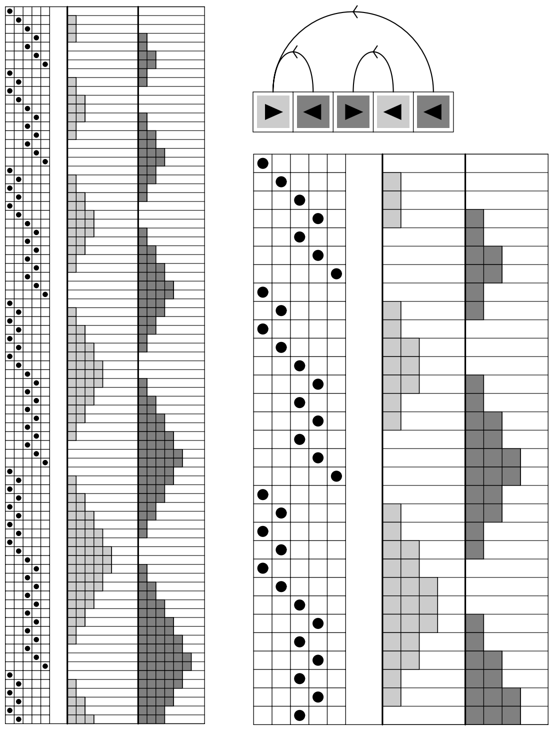
And with this setup, the pictures above show three very simple examples of register machines with two registers. The programs for each of the machines are given at the top, with ▸ representing an increment instruction, and ◂ a decrement-jump. The successive steps in the evolution of each machine are shown on successive lines down the page. The instruction being executed is indicated at each step by the position of the dot on the left, while the numbers in each of the two registers are indicated by the gray blocks on the right.

All the register machines shown start by executing the first instruction in their programs. And with the particular programs used here, the machines are then set up just to execute all the other instructions in their programs in turn, jumping back to the beginning of their programs whenever they reach the end.

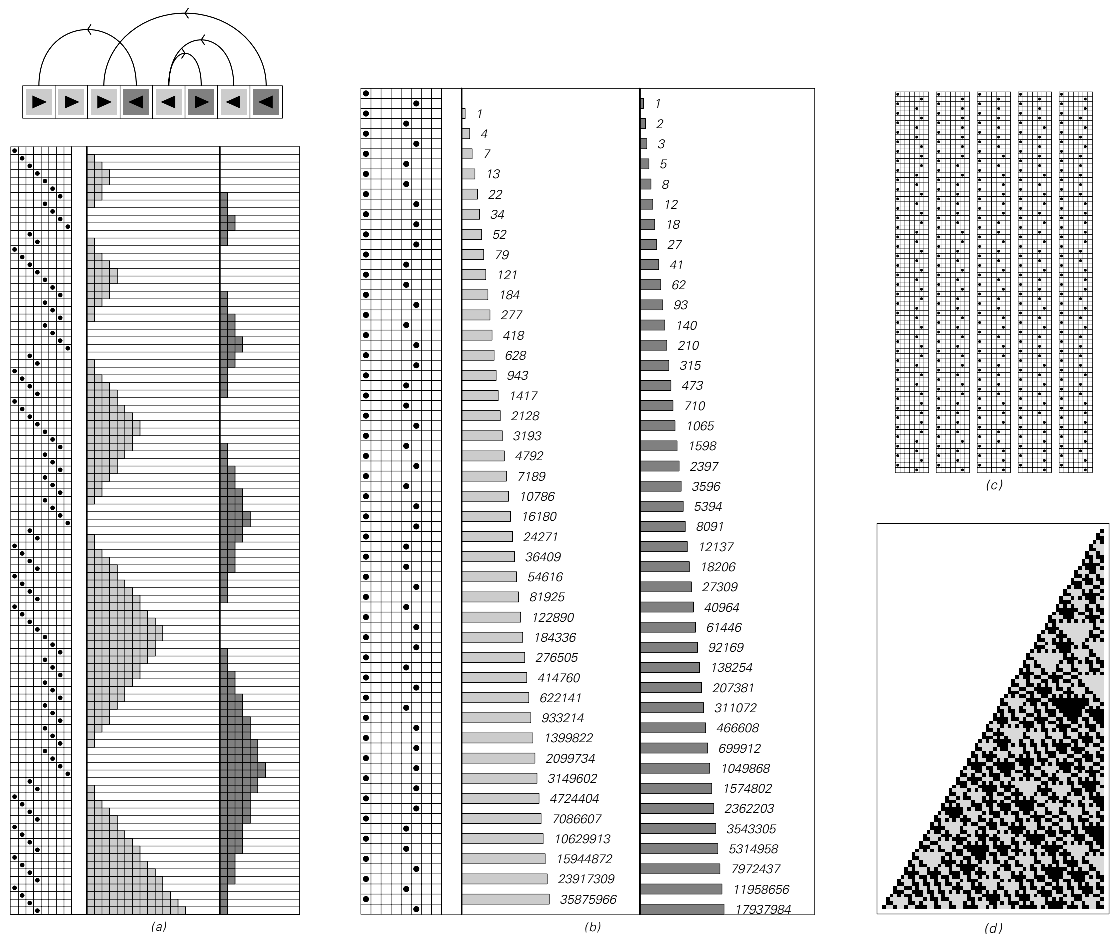
Both registers in each machine are initially zero. And in the first machine, the first register alternates between 0 and 1, while the second remains zero. In the second machine, however, the first register again alternates between 0 and 1, but the second register progressively grows. And finally, in the third machine both registers grow.

But in all these three examples, the overall behavior is essentially repetitive. And indeed it turns out that among the 10,552 possible register machines with programs that are four or fewer instructions long, not a single one exhibits more complicated behavior.

However, with five instructions, slightly more complicated behavior becomes possible, as the picture below shows. But even in this example, there is still a highly regular nested structure.


A register machine that shows nested rather than strictly repetitive behavior. The register machine has a program which is five instructions long. It turns out that this program is one of only two (which differ just by interchange of the first and second registers) out of the 248,832 possible programs with five instructions that yield anything other than strictly repetitive behavior.

And it turns out that even with up to seven instructions, none of the 276,224,376 programs that are possible lead to substantially more complicated behavior. But with eight instructions, 126 out of the 11,019,960,576 possible programs finally do show more complicated behavior. The next page gives an example.



A register machine whose behavior seems in some ways random. The program for this register machine is eight instructions long. Testing all 11,019,960,576 possible programs of length eight revealed just this and 125 similar cases of complex behavior. Part (b) shows the evolution in compressed form, with only those steps included at which either of the registers has just decreased to zero. The values of the nonzero registers are shown using a logarithmic scale. Part (c) shows the instructions that are executed for the first 400 times that one of the registers is decreased to zero. Finally, part (d) gives the successive values attained by the second register at steps where the first register has just decreased to zero. These values are given here as binary digit sequences. As discussed on page 122, the values can in fact be obtained by a simple arithmetic rule, without explicitly following each step in the evolution of the register machine. If one value is n, then the next value is $3n/2$ if n is even, and $(3n+1)/2$ if n is odd. The initial condition is $n = 1$.

Looking just at the ordinary evolution labelled (a), however, the system might still appear to have quite simple and regular behavior. But a closer examination turns out to reveal irregularities. Part (b) of the picture shows a version of the evolution compressed to include only those steps at which one of the two registers has just decreased to zero. And in this picture one immediately sees some apparently random variation in the instructions that are executed.

Part (c) of the picture then shows which instructions are executed for the first 400 times one of the registers has just decreased to zero. And part (d) finally shows the base 2 digits of the successive values attained by the second register when the first register has just decreased to zero. The results appear to show considerable randomness.

So even though it may not be as obvious as in some of the other systems we have studied, the simple register machine on the facing page can still generate complex and seemingly quite random behavior.

So what about more complicated register machines?

An obvious possibility is to allow more than two registers. But it turns out that very little is normally gained by doing this. With three registers, for example, seemingly random behavior can be obtained with a program that is seven rather than eight instructions long. But the actual behavior of the program is almost indistinguishable from what we have already seen with two registers.

Another way to set up more complicated register machines is to extend the kinds of underlying instructions one allows. One can for example introduce instructions that refer to two registers at a time, adding, subtracting or comparing their contents. But it turns out that the presence of instructions like these rarely seems to have much effect on either the form of complex behavior that can occur, or how common it is.

Yet particularly when such extended instruction sets are used, register machines can provide fairly accurate idealizations of the low-level operations of real computers. And as a result, programs for register machines are often very much like programs written in actual low-level computer languages such as C, BASIC, Java or assembler.

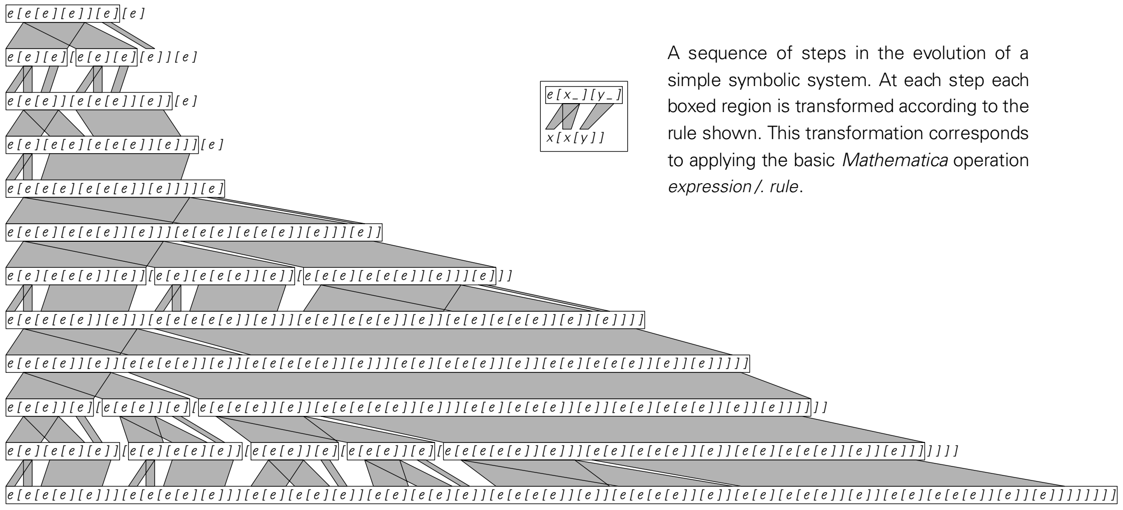
In a typical case, each variable in such a program simply corresponds to one of the registers in the register machine, with no arrays or pointers being allowed. And with this correspondence, our general results on register machines can also be expected to apply to simple programs written in actual low-level computer languages.

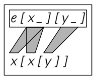
Practical details make it somewhat difficult to do systematic experiments on such programs. But the experiments I have carried out do suggest that, just as with simple register machines, searching through many millions of short programs typically yields at least a few that exhibit complex and seemingly random behavior.

Symbolic Systems

Register machines provide simple idealizations of typical low-level computer languages. But what about *Mathematica*? How can one set up a simple idealization of the transformations on symbolic expressions that *Mathematica* does? One approach suggested by the idea of combinators from the 1920s is to consider expressions with forms such as *e*[*e*[*e*][*e*]][*e*][*e*] and then to make transformations on these by repeatedly applying rules such as *e*[*x*_][*y*_] → *x*[*x*[*y*]], where *x*_ and *y*_ stand for any expression.

The picture below shows an example of this. At each step the transformation is done by scanning once from left to right, and applying the rule wherever possible without overlapping.


e[e[e][e]][e][e]
e[e][e][e[e][e][e]][e]
e[e[e]][e[e[e]][e]][e]
e[e][e[e][e[e[e]][e]]][e]
e[e[e[e][e[e[e]][e]]]][e]
e[e[e][e[e[e]][e]]][e[e[e][e[e[e]][e]]][e]]
e[e][e[e[e]][e]][e[e][e[e[e]][e]][e[e[e][e[e[e]][e]]][e]]]
e[e[e[e[e]][e]]][e[e[e[e[e]][e]]][e[e][e[e[e]][e]][e[e][e[e[e]][e]][e]]]]
e[e[e[e]][e]][e[e[e[e]][e]][e[e[e[e[e]][e]]][e[e][e[e[e]][e]][e[e][e[e[e]][e]][e]]]]]
e[e[e]][e][e[e[e]][e][e[e[e[e]][e]][e[e[e[e[e]][e]]][e[e][e[e[e]][e]][e[e][e[e[e]][e]][e]]]]]]
e[e][e[e][e]][e[e][e[e][e]][e[e[e]][e][e[e[e]][e][e[e[e[e]][e]]][e[e][e[e[e]][e]][e[e][e[e[e]][e]][e]]]]]]]
e[e[e[e][e]]][e[e[e[e][e]]][e[e][e[e][e]][e[e][e[e][e]][e[e[e[e]][e]][e[e[e[e]][e]][e[e][e[e[e]][e]][e[e][e[e[e]][e]][e]]]]]]]



A sequence of steps in the evolution of a simple symbolic system. At each step each boxed region is transformed according to the rule shown. This transformation corresponds to applying the basic *Mathematica* operation *expression /. rule*.