EXCERPTED FROM

STEPHEN WOLFRAM

A NEW KIND OF SCIENCE

CHAPTER 3

The World of Simple Programs

3 The World of Simple Programs

The Search for General Features

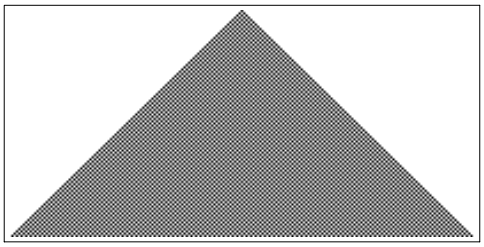
At the beginning of the last chapter we asked the basic question of what simple programs typically do. And as a first step towards answering this question we looked at several specific examples of a class of programs known as cellular automata.

The basic types of behavior that we found are illustrated in the pictures on the next page. In the first of these there is pure repetition, and a very simple pattern is formed. In the second, there are many intricate details, but at an overall level there is still a very regular nested structure that emerges.

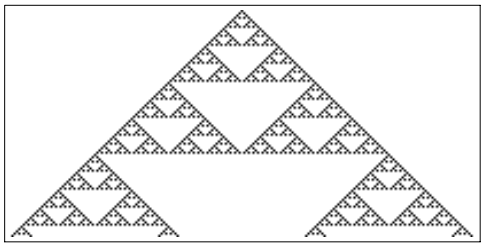
In the third picture, however, one no longer sees such regularity, and instead there is behavior that seems in many respects random. And finally in the fourth picture there is what appears to be still more complex behavior—with elaborate localized structures being generated that interact in complex ways.

At the outset there was no indication that simple programs could ever produce behavior so diverse and often complex. But having now seen these examples, the question becomes how typical they are. Is it only cellular automata with very specific underlying rules that produce such behavior? Or is it in fact common in all sorts of simple programs?

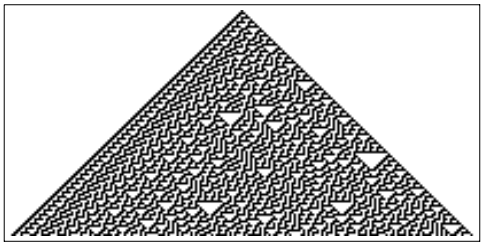
My purpose in this chapter is to answer this question by looking at a wide range of different kinds of programs. And in a sense my



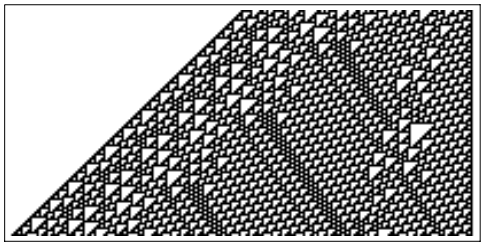
repetition (rule 250)



nesting (rule 90)



randomness (rule 30)



localized structures (rule 110)

Four basic examples from the previous chapter of behavior produced by cellular automata with simple underlying rules. In each case, the most obvious features that are seen are different. Note that all the pictures are shown on the same scale; the last picture appears coarser because the structures it contains are larger.

approach is to work like a naturalist—exploring and studying the various forms that exist in the world of simple programs.

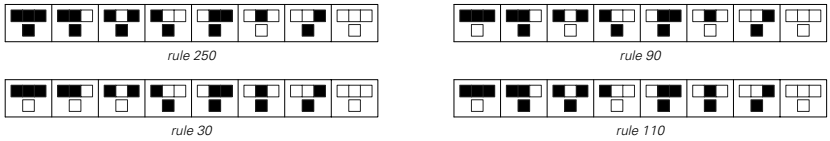
I start by considering more general cellular automata, and then I go on to consider a whole sequence of other kinds of programs—with underlying structures further and further away from the array of black and white cells in the cellular automata of the previous chapter.

And what I discover is that whatever kind of underlying rules one uses, the behavior that emerges turns out to be remarkably similar to the basic examples that we have already seen in cellular automata.

Throughout the world of simple programs, it seems, there is great universality in the types of overall behavior that can be produced. And in a sense it is ultimately this that makes it possible for me to construct the coherent new kind of science that I describe in this book—and to use it to elucidate a large number of phenomena, independent of the particular details of the systems in which they occur.

More Cellular Automata

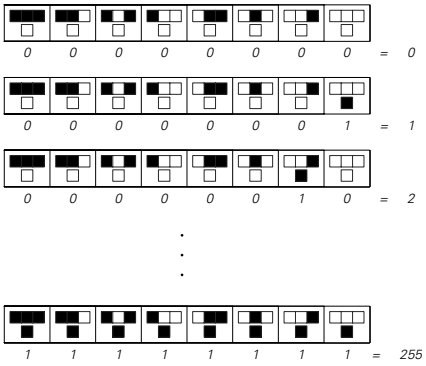
The pictures below show the rules used in the four cellular automata on the facing page. The overall structure of these rules is the same in each case; what differs is the specific choice of new colors for each possible combination of previous colors for a cell and its two neighbors.



The rules used for the four examples of cellular automata on the facing page. In each case, these specify the new color of a cell for each possible combination of colors of that cell and its immediate neighbors on the previous step. The rules are numbered according to the scheme described below.

There turn out to be a total of 256 possible sets of choices that can be made. And following my original work on cellular automata these choices can be numbered from 0 to 255, as in the picture below.

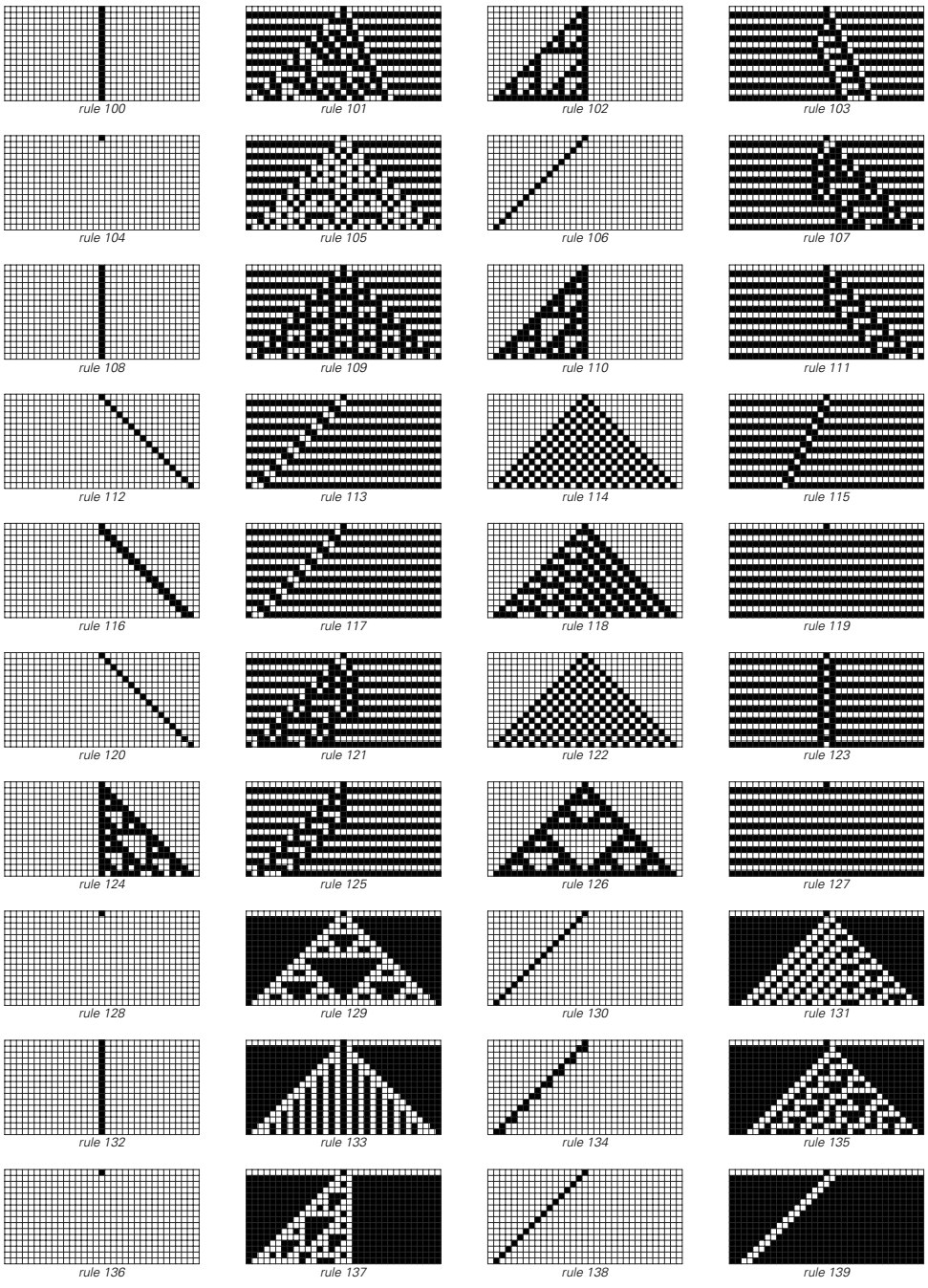
The sequence of 256 possible cellular automaton rules of the kind shown above. As indicated, the rules can conveniently be numbered from 0 to 255. The number assigned is such that when written in base 2, it gives a sequence of 0's and 1's that correspond to the sequence of new colors chosen for each of the eight possible cases covered by the rule.



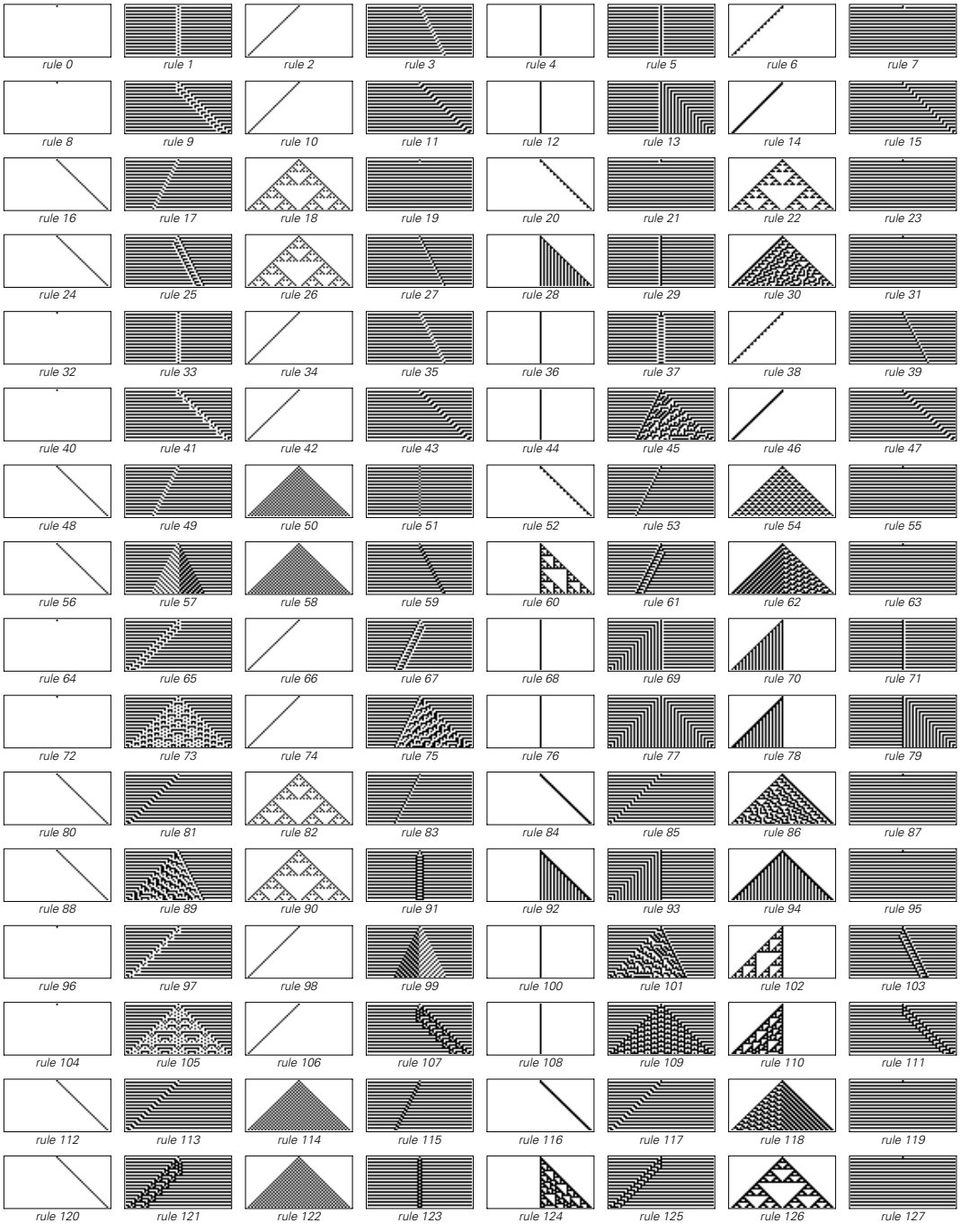
But so how do cellular automata with all these different rules behave? The next page shows a few examples in detail, while the following two pages show what happens in all 256 possible cases.

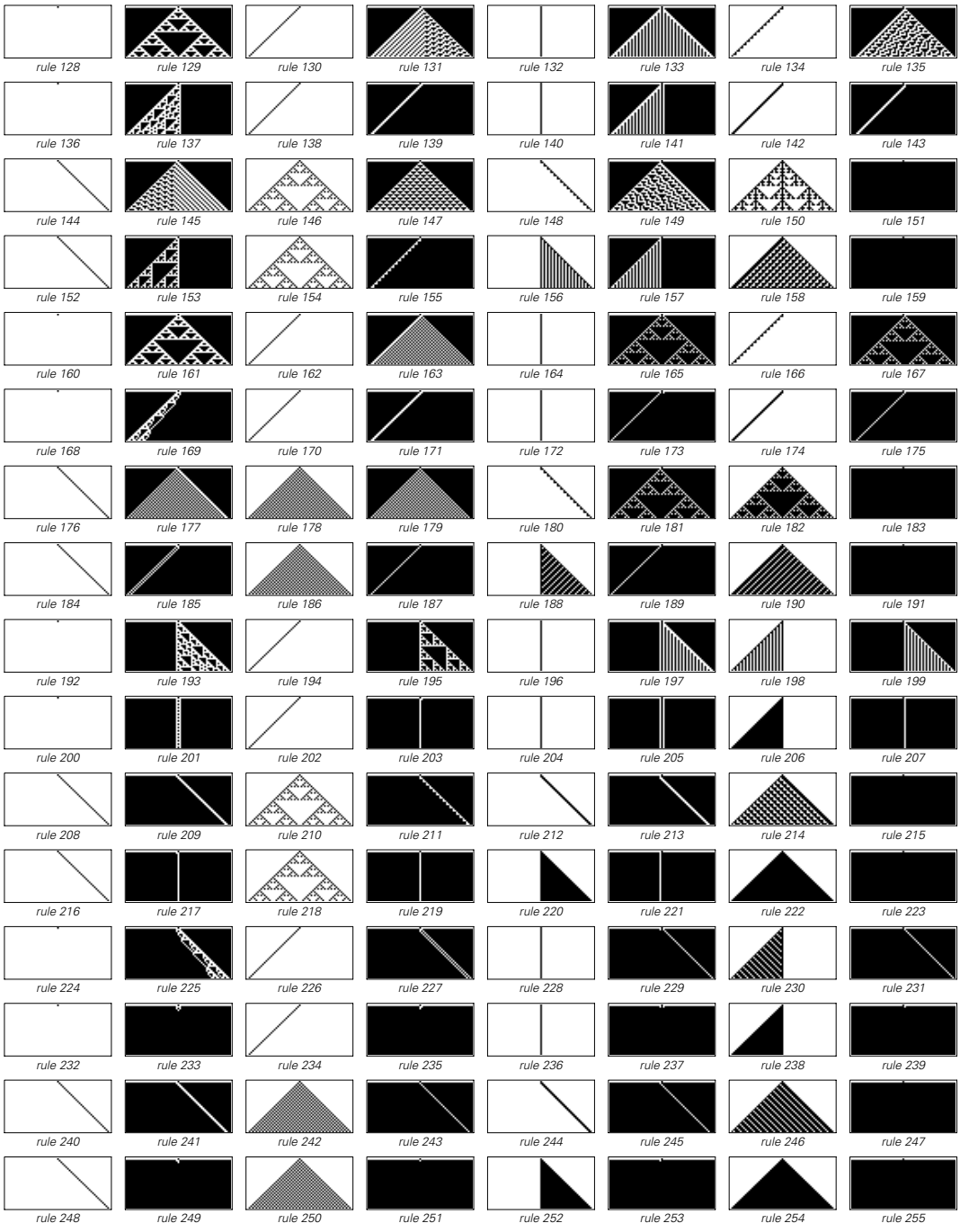
At first, the diversity of what one sees is a little overwhelming. But on closer investigation, definite themes begin to emerge.

In the very simplest cases, all the cells in the cellular automaton end up just having the same color after one step. Thus, for example, in



Evolution of cellular automata with a sequence of different possible rules, starting in all cases from a single black cell.


rule 0
rule 1
rule 2
rule 3
rule 4
rule 5
rule 6
rule 7
rule 8
rule 9
rule 10
rule 11
rule 12
rule 13
rule 14
rule 15
rule 16
rule 17
rule 18
rule 19
rule 20
rule 21
rule 22
rule 23
rule 24
rule 25
rule 26
rule 27
rule 28
rule 29
rule 30
rule 31
rule 32
rule 33
rule 34
rule 35
rule 36
rule 37
rule 38
rule 39
rule 40
rule 41
rule 42
rule 43
rule 44
rule 45
rule 46
rule 47
rule 48
rule 49
rule 50
rule 51
rule 52
rule 53
rule 54
rule 55
rule 56
rule 57
rule 58
rule 59
rule 60
rule 61
rule 62
rule 63
rule 64
rule 65
rule 66
rule 67
rule 68
rule 69
rule 70
rule 71
rule 72
rule 73
rule 74
rule 75
rule 76
rule 77
rule 78
rule 79
rule 80
rule 81
rule 82
rule 83
rule 84
rule 85
rule 86
rule 87
rule 88
rule 89
rule 90
rule 91
rule 92
rule 93
rule 94
rule 95
rule 96
rule 97
rule 98
rule 99
rule 100
rule 101
rule 102
rule 103
rule 104
rule 105
rule 106
rule 107
rule 108
rule 109
rule 110
rule 111
rule 112
rule 113
rule 114
rule 115
rule 116
rule 117
rule 118
rule 119
rule 120
rule 121
rule 122
rule 123
rule 124
rule 125
rule 126
rule 127


rule 128
rule 129
rule 130
rule 131
rule 132
rule 133
rule 134
rule 135
rule 136
rule 137
rule 138
rule 139
rule 140
rule 141
rule 142
rule 143
rule 144
rule 145
rule 146
rule 147
rule 148
rule 149
rule 150
rule 151
rule 152
rule 153
rule 154
rule 155
rule 156
rule 157
rule 158
rule 159
rule 160
rule 161
rule 162
rule 163
rule 164
rule 165
rule 166
rule 167
rule 168
rule 169
rule 170
rule 171
rule 172
rule 173
rule 174
rule 175
rule 176
rule 177
rule 178
rule 179
rule 180
rule 181
rule 182
rule 183
rule 184
rule 185
rule 186
rule 187
rule 188
rule 189
rule 190
rule 191
rule 192
rule 193
rule 194
rule 195
rule 196
rule 197
rule 198
rule 199
rule 200
rule 201
rule 202
rule 203
rule 204
rule 205
rule 206
rule 207
rule 208
rule 209
rule 210
rule 211
rule 212
rule 213
rule 214
rule 215
rule 216
rule 217
rule 218
rule 219
rule 220
rule 221
rule 222
rule 223
rule 224
rule 225
rule 226
rule 227
rule 228
rule 229
rule 230
rule 231
rule 232
rule 233
rule 234
rule 235
rule 236
rule 237
rule 238
rule 239
rule 240
rule 241
rule 242
rule 243
rule 244
rule 245
rule 246
rule 247
rule 248
rule 249
rule 250
rule 251
rule 252
rule 253
rule 254
rule 255

rules 0 and 128 all the cells become white, while in rule 255 all of them become black. There are also rules such as 7 and 127 in which all cells alternate between black and white on successive steps.

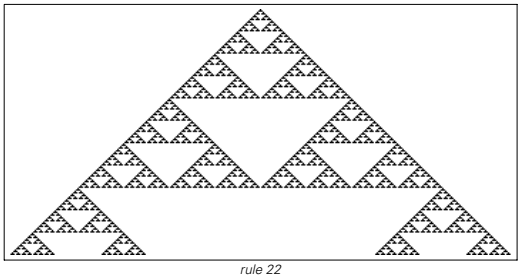
But among the rules shown on the last few pages, the single most common kind of behavior is one in which a pattern consisting of a single cell or a small group of cells persists. Sometimes this pattern remains stationary, as in rules 4 and 123. But in other cases, such as rules 2 and 103, it moves to the left or right.

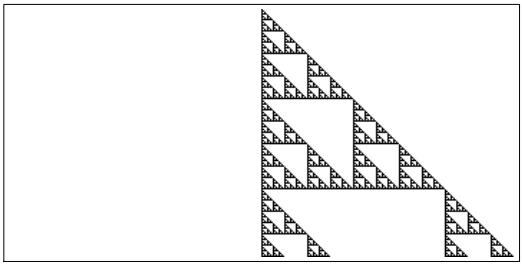
It turns out that the basic structure of the cellular automata discussed here implies that the maximum speed of any such motion must be one cell per step. And in many rules, this maximum speed is achieved—although in rules such as 3 and 103 the average speed is instead only half a cell per step.

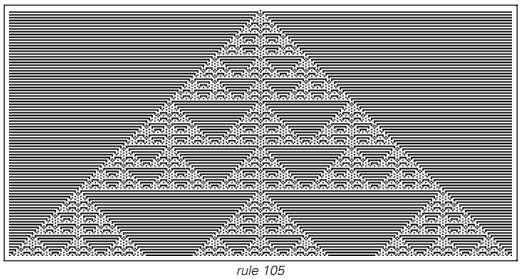
In about two-thirds of all the cellular automata shown on the last few pages, the patterns produced remain of a fixed size. But in about one-third of cases, the patterns instead grow forever. Of such growing patterns, the simplest kind are purely repetitive ones, such as those seen in rules 50 and 109. But while repetitive patterns are by a small margin the most common kind, about 14% of all the cellular automata shown yield more complicated kinds of patterns.

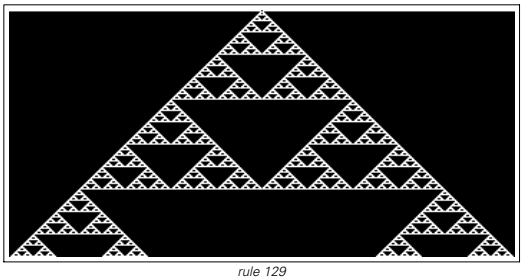
The most common of these are nested patterns, like those on the next page. And it turns out that although 24 rules in all yield such nested patterns, there are only three fundamentally different forms that occur. The simplest and by far the most common is the one exemplified by rules 22 and 60. But as the pictures on the next page show, other nested forms are also possible. (In the case of rule 225, the width of the overall pattern does not grow at a fixed rate, but instead is on average proportional to the square root of the number of steps.)

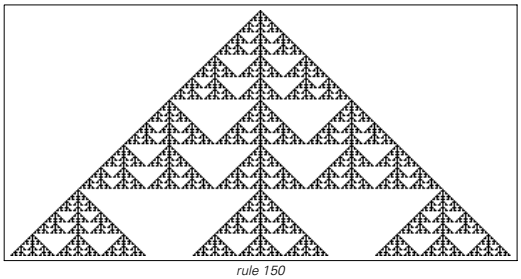
◀ The behavior of all 256 possible cellular automata with rules involving two colors and nearest neighbors. In each case, thirty steps of evolution are shown, starting from a single black cell. Note that some of the rules are related just by interchange of left and right or black and white (e.g. rules 2 and 16 or rules 126 and 129). There are 88 fundamentally inequivalent such elementary rules.

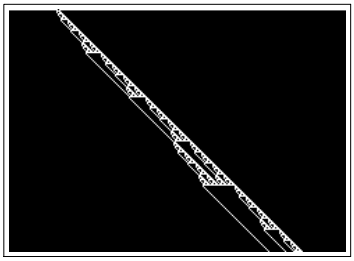

rule 22

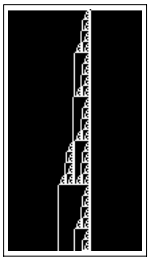

rule 60


rule 105


rule 129


rule 150

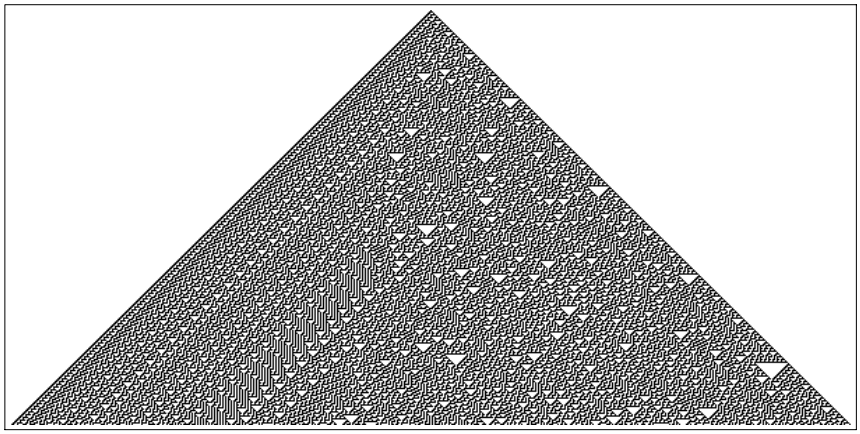

rule 225


rule 225 (shifted)

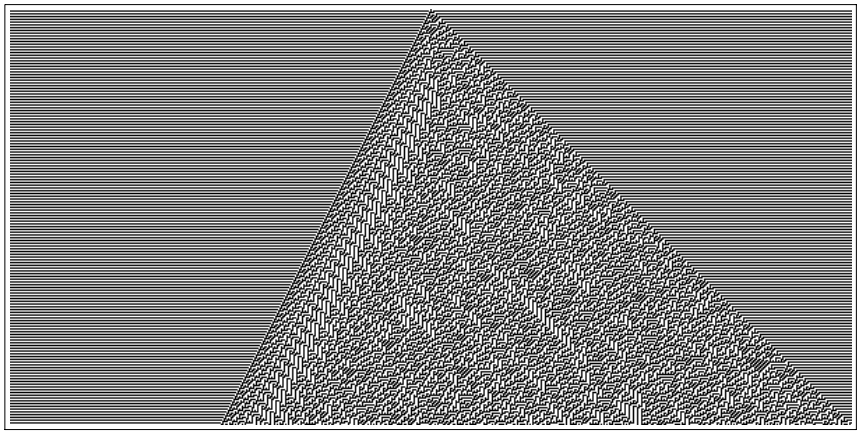
Examples of cellular automata that produce nested or fractal patterns. Rule 22—like rule 90 from page 26—gives a pattern with fractal dimension $Log[2, 3] \approx 1.59$; rule 150 gives one with fractal dimension $Log[2, 1 + \sqrt{5}\,] \approx 1.69$. The width of the pattern obtained from rule 225 increases like the square root of the number of steps.

Repetition and nesting are widespread themes in many cellular automata. But as we saw in the previous chapter, it is also possible for cellular automata to produce patterns that seem in many respects random. And out of the 256 rules discussed here, it turns out that 10 yield such apparent randomness. There are three basic forms, as illustrated on the facing page.

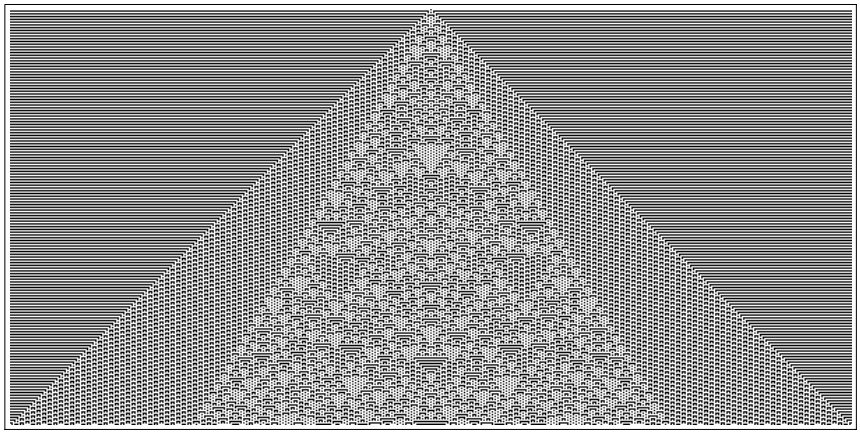
Examples of cellular automata that produce patterns with many apparently random features. Three hundred steps of evolution are shown, starting in each case from a single black cell. ▸



rule 30



rule 45



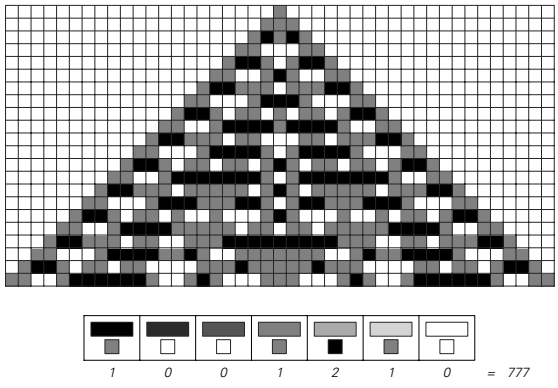
rule 73

Beyond randomness, the last example in the previous chapter was rule 110: a cellular automaton whose behavior becomes partitioned into a complex mixture of regular and irregular parts. This particular cellular automaton is essentially unique among the 256 rules considered here: of the four cases in which such behavior is seen, all are equivalent if one just interchanges the roles of left and right or black and white.

So what about more complicated cellular automaton rules?

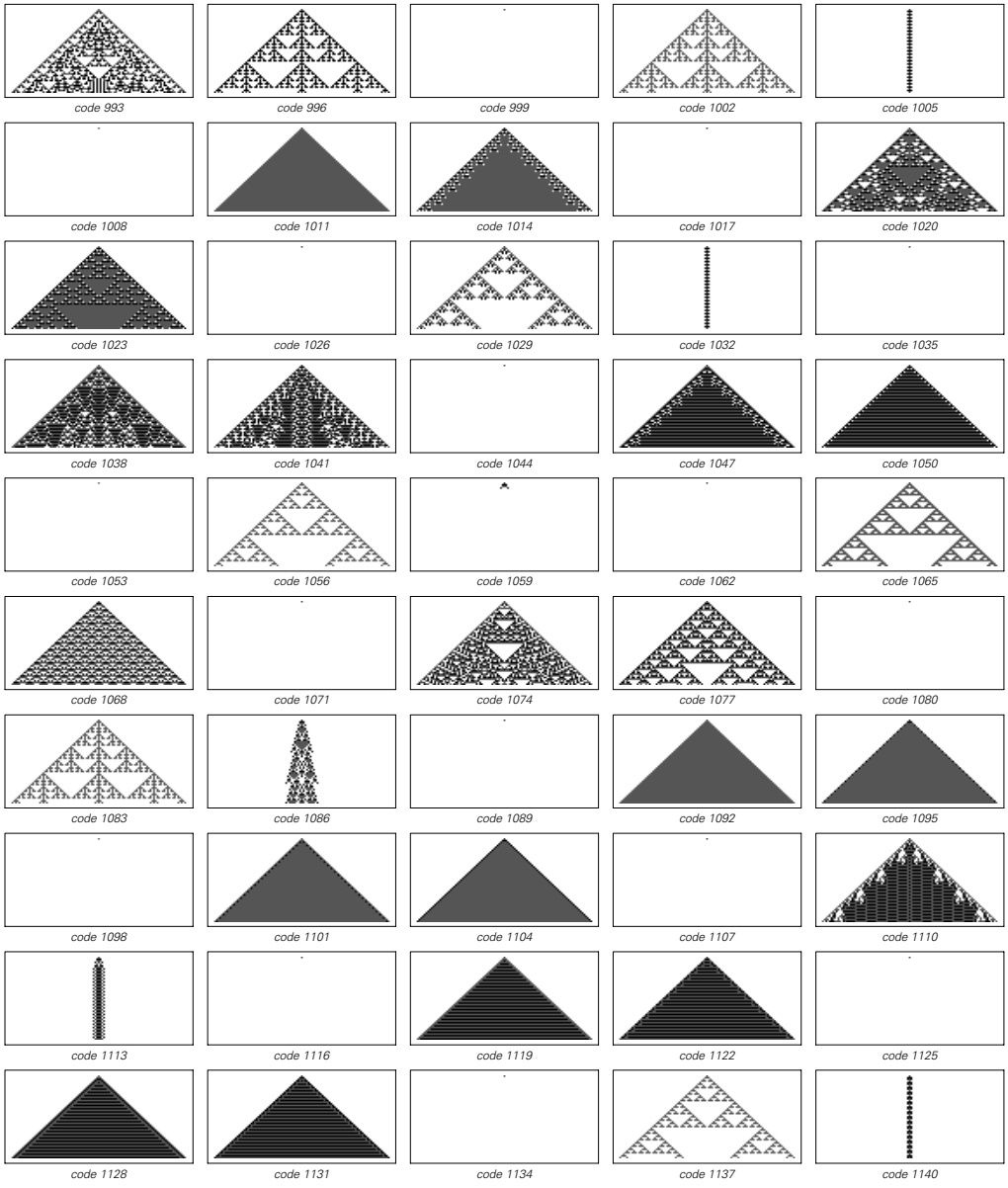
The 256 "elementary" rules that we have discussed so far are by most measures the simplest possible—and were the first ones I studied. But one can for example also look at rules that involve three colors, rather than two, so that cells can not only be black and white, but also gray. The total number of possible rules of this kind turns out to be immense—7,625,597,484,987 in all—but by considering only so-called "totalistic" ones, the number becomes much more manageable.

The idea of a totalistic rule is to take the new color of each cell to depend only on the average color of neighboring cells, and not on their individual colors. The picture below shows one example of how this works. And with three possible colors for each cell, there are 2187 possible totalistic rules, each of which can conveniently be identified by a code number as illustrated in the picture. The facing page shows a representative sequence of such rules.



Example of a totalistic cellular automaton with three possible colors for each cell. The rule is set up so that the new color of every cell is determined by the average of the previous colors of the cell and its immediate neighbors. With 0 representing white, 1 gray and 2 black, the rightmost element of the rule gives the result for average color 0, while the element immediately to its left gives the result for average color 1/3—and so on. Interpreting the sequence of new colors as a sequence of base 3 digits, one can assign a code number to each totalistic rule.

We might have expected that by allowing three colors rather than two we would immediately get noticeably more complicated behavior.

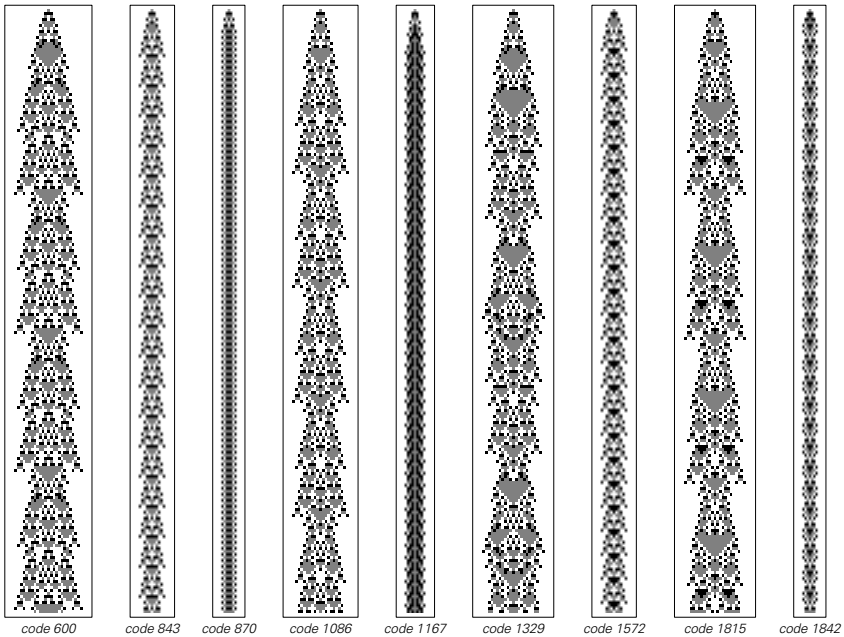


A sequence of totalistic cellular automata with three possible colors for each cell. Although their basic rules are more complicated, the cellular automata shown here do not seem to have fundamentally more complicated behavior than the two-color cellular automata shown on previous pages. Note that in the sequence of rules shown here, those that change the white background are not included. The symmetry of all the patterns is a consequence of the basic structure of totalistic rules.

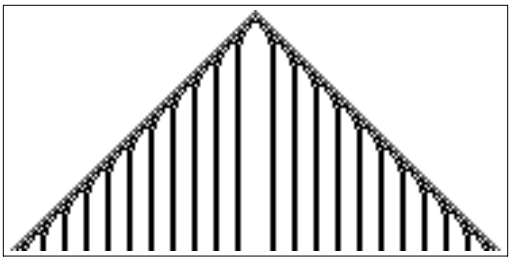
But in fact the behavior we see on the previous page is not unlike what we already saw in many elementary cellular automata a few pages back. Having more complicated underlying rules has not, it seems, led to much greater complexity in overall behavior.

And indeed, this is a first indication of an important general phenomenon: that at least beyond a certain point, adding complexity to the underlying rules for a system does not ultimately lead to more complex overall behavior. And so for example, in the case of cellular automata, it seems that all the essential ingredients needed to produce even the most complex behavior already exist in elementary rules.

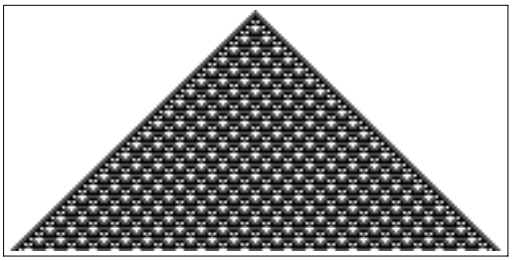
Using more complicated rules may be convenient if one wants, say, to reproduce the details of particular natural systems, but it does not add fundamentally new features. Indeed, looking at the pictures on the previous page one sees exactly the same basic themes as in elementary cellular automata. There are some patterns that attain a definite size, then repeat forever, as shown below, others that continue to grow, but have a repetitive form, as at the top of the facing page, and still others that produce nested or fractal patterns, as at the bottom of the page.



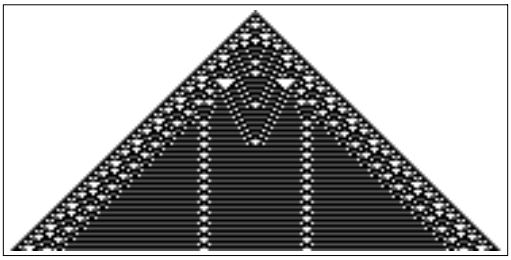
Examples of three-color totalistic rules that yield patterns which attain a certain size, then repeat forever. The maximum repetition period is found to be 78 steps, and is achieved by the rule with code number 1329. In the pictures shown here and on the following pages, the initial condition used contains a single gray cell.



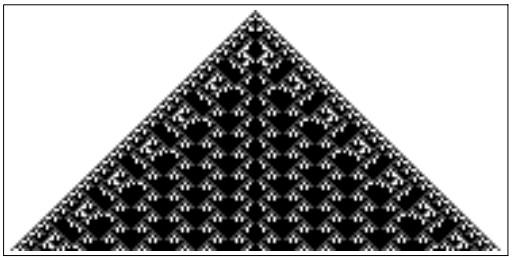
code 219



code 957

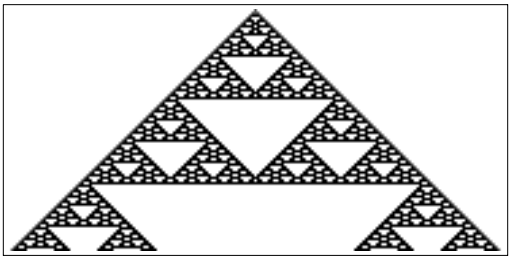


code 966

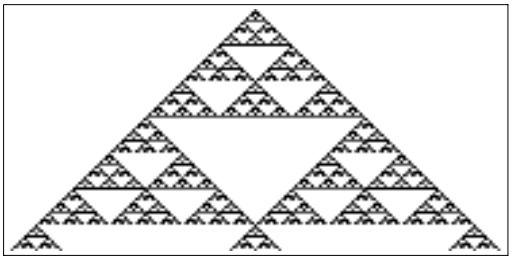


code 1884

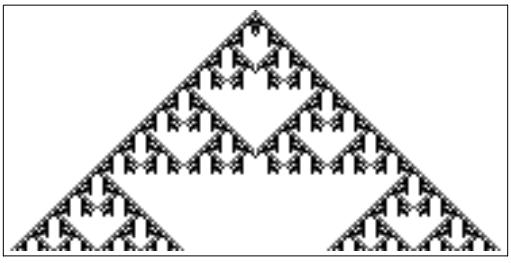
Examples of three-color totalistic rules that yield patterns which grow forever but have a fundamentally repetitive structure.



code 237



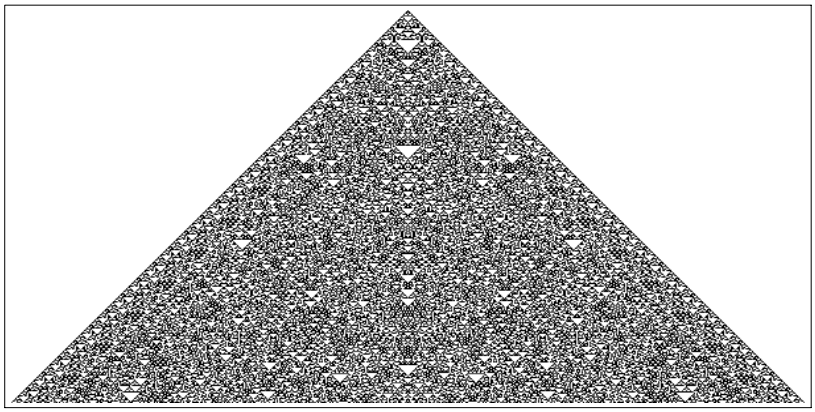
code 420



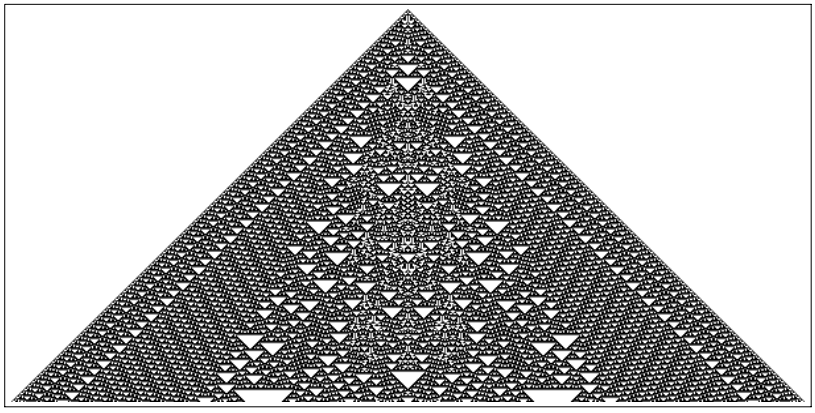
code 948

code 1749

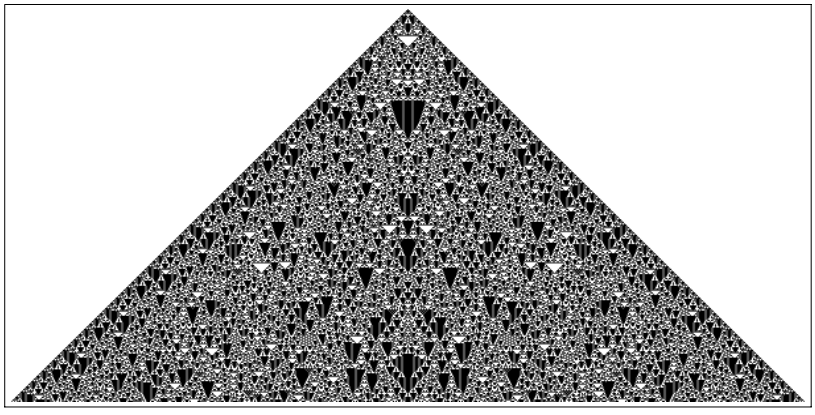
Examples of three-color totalistic rules which yield nested patterns. In most cases, these patterns have an overall form that is similar to what was found with two-color rules. But code 420, for example, yields a pattern with a slightly different structure.



code 177



code 912



code 2040

Examples of three-color totalistic rules that yield patterns with seemingly random features. Three hundred steps of evolution are shown in each case.

In detail, some of the patterns are definitely more complicated than those seen in elementary rules. But at the level of overall behavior, there are no fundamental differences. And in the case of nested patterns even the specific structures seen are usually the same as for elementary rules. Thus, for example, the structure in codes 237 and 948 is the most common, followed by the one in code 1749. The only new structure not already seen in elementary rules is the one in code 420—but this occurs only quite rarely.

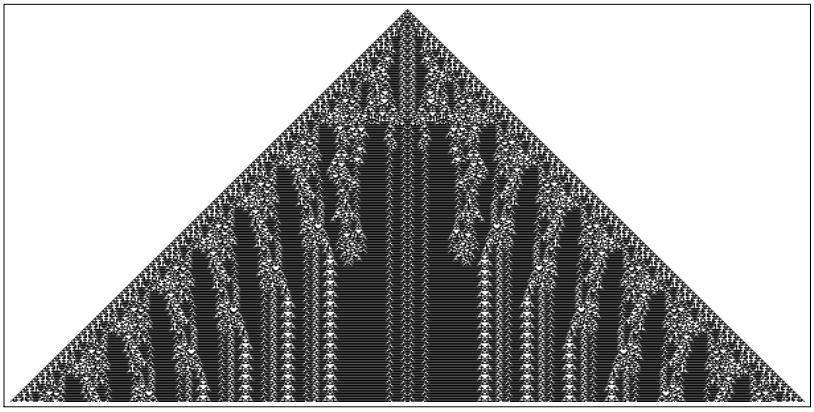
About 85% of all three-color totalistic cellular automata produce behavior that is ultimately quite regular. But just as in elementary cellular automata, there are some rules that yield behavior that seems in many respects random. A few examples of this are given on the facing page.

Beyond fairly uniform random behavior, there are also cases similar to elementary rule 110 in which definite structures are produced that interact in complicated ways. The next page gives a few examples. In the first case shown, the pattern becomes repetitive after about 150 steps. In the other two cases, however, it is much less clear what will ultimately happen. The following pages continue these patterns for 3000 steps. But even after this many steps it is still quite unclear what the final behavior will be.

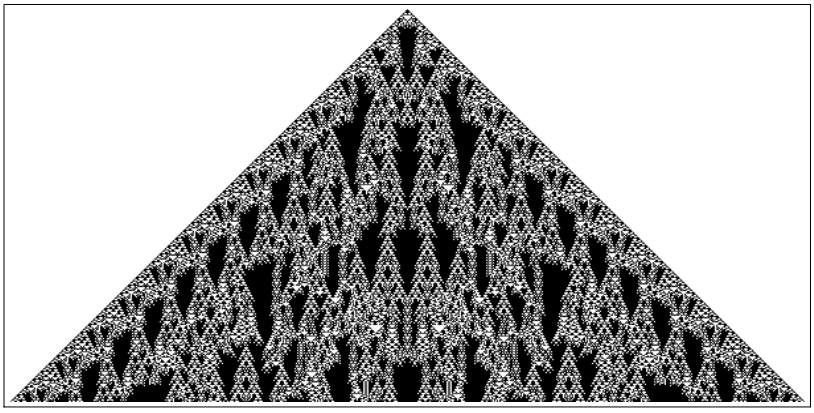
Looking at pictures like these, it is at first difficult to believe that they can be generated just by following very simple underlying cellular automaton rules. And indeed, even if one accepts this, there is still a tendency to assume that somehow what one sees must be a consequence of some very special feature of cellular automata.

As it turns out, complexity is particularly widespread in cellular automata, and for this reason it is fortunate that cellular automata were the very first systems that I originally decided to study.

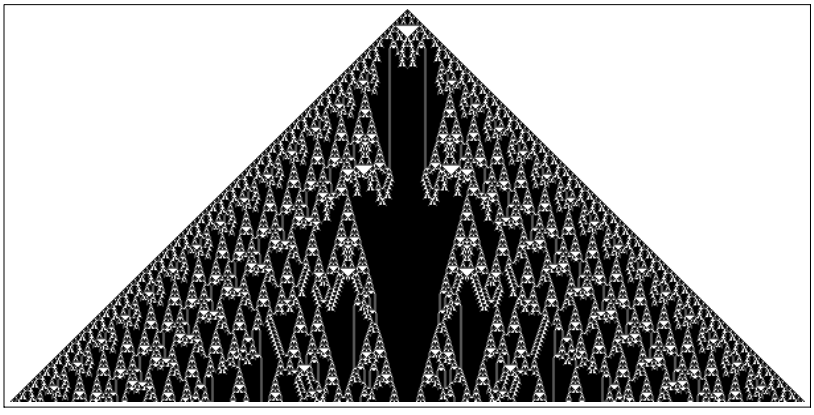
But as we will see in the remainder of this chapter, the fundamental phenomena that we discovered in the previous chapter are in no way restricted to cellular automata. And although cellular automata remain some of the very best examples, we will see that a vast range of utterly different systems all in the end turn out to exhibit extremely similar types of behavior.



code 1041

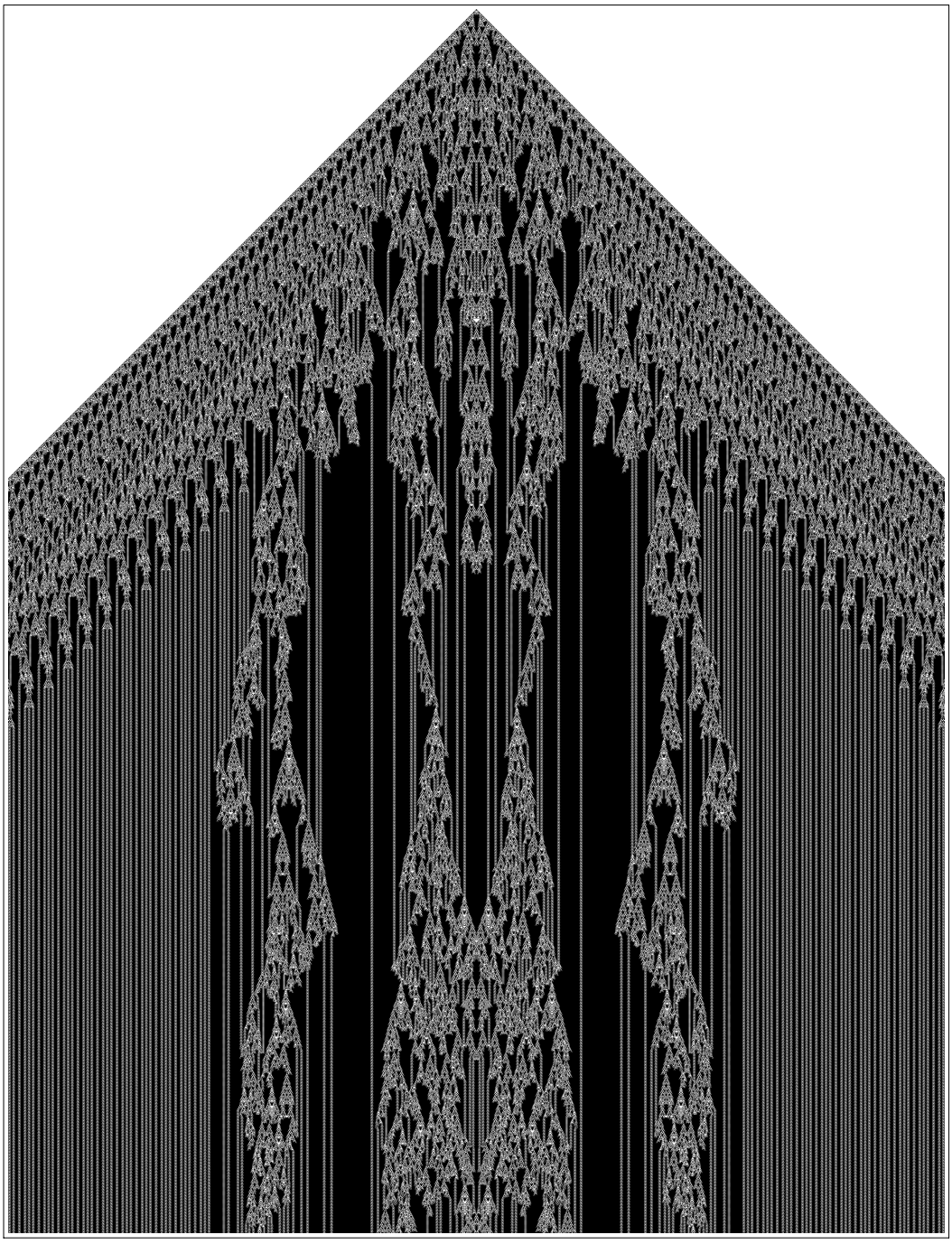


code 1635

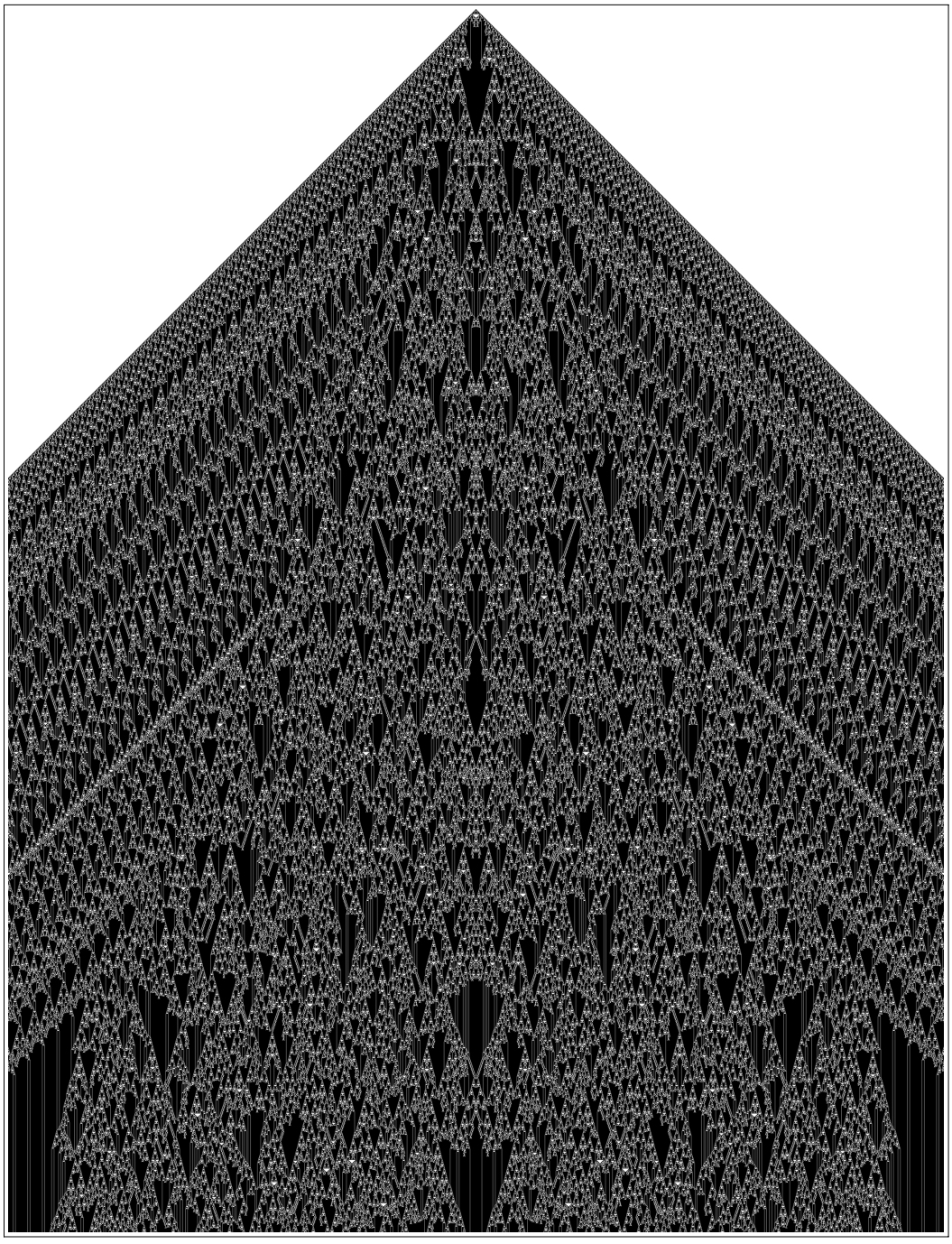


code 2049

Examples of three-color totalistic rules with highly complex behavior showing a mixture of regularity and irregularity. The partitioning into identifiable structures is similar to what we saw in rule 110 on page 32.

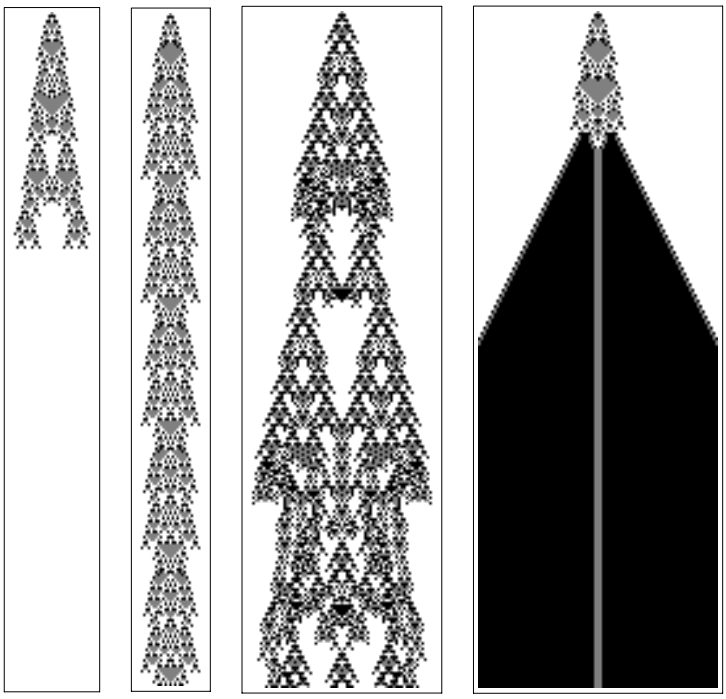


code 1635



code 2049

The pictures below show totalistic cellular automata whose overall patterns of growth seem, at least at first, quite complicated. But it turns out that after only about 100 steps, three out of four of these patterns have resolved into simple forms.

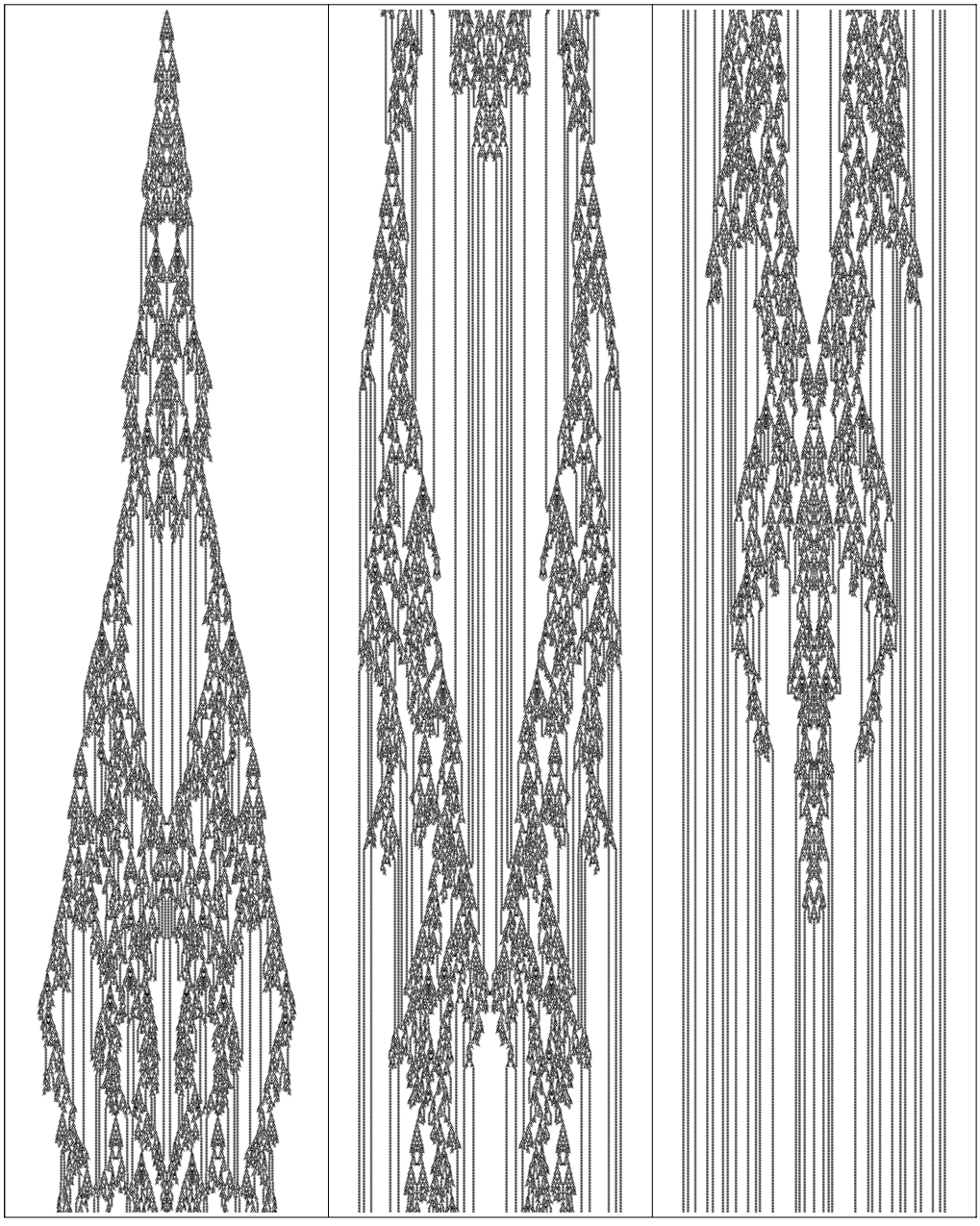


code 357 code 600 code 1599 code 2058

Examples of rules that yield patterns which seem to be on the edge between growth and extinction. For all but code 1599, the fate of these patterns in fact becomes clear after less than 100 steps. A total of 250 steps are shown here.

The one remaining pattern is, however, much more complicated. As shown on the next page, for several thousand steps it simply grows, albeit somewhat irregularly. But then its growth becomes slower. And inside the pattern parts begin to die out. Yet there continue to be occasional bursts of growth. But finally, after a total of 8282 steps, the pattern resolves into 31 simple repetitive structures.

◀ Three thousand steps in the evolution of the last two cellular automata from page 66. Despite the simplicity of their underlying rules, the final patterns produced show immense complexity. In neither case is it clear what the final outcome will be—whether apparent randomness will take over, or whether a simple repetitive form will emerge.



code 1599

Nine thousand steps in the evolution of the three-color totalistic cellular automaton with code number 1599. Starting from a single gray cell, each column corresponds to 3000 steps. The outcome of the evolution finally becomes clear after 8282 steps, when the pattern resolves into 31 simple repetitive structures.

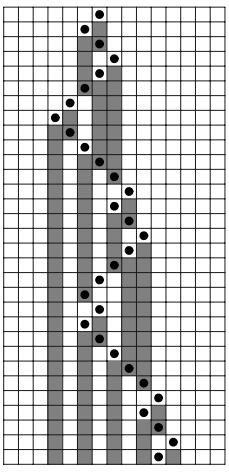
Mobile Automata

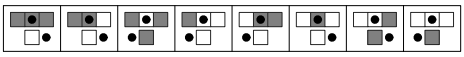
One of the basic features of a cellular automaton is that the colors of all the cells it contains are updated in parallel at every step in its evolution.

But how important is this feature in determining the overall behavior that occurs? To address this question, I consider in this section a class of systems that I call "mobile automata".

Mobile automata are similar to cellular automata except that instead of updating all cells in parallel, they have just a single "active cell" that gets updated at each step—and then they have rules that specify how this active cell should move from one step to the next.

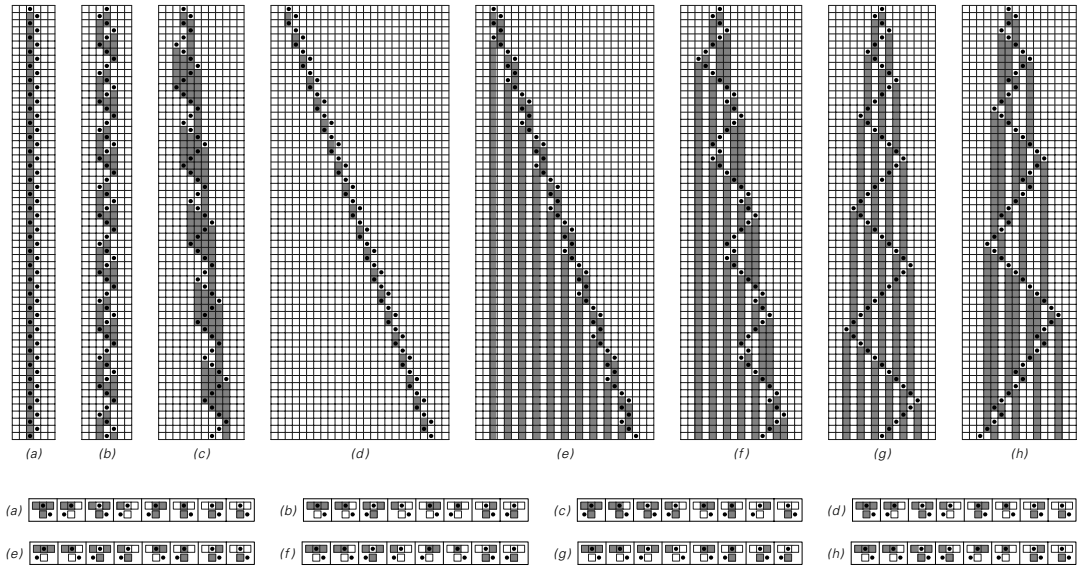
The picture below shows an example of a mobile automaton. The active cell is indicated by a black dot. The rule applies only to this active cell. It looks at the color of the active cell and its immediate neighbors, then specifies what the new color of the active cell should be, and whether the active cell should move left or right.





An example of a mobile automaton. Like a cellular automaton, a mobile automaton consists of a line of cells, with each cell having two possible colors. But unlike a cellular automaton, a mobile automaton has only one "active cell" (indicated here by a black dot) at any particular step. The rule for the mobile automaton specifies both how the color of this active cell should be updated, and whether it should move to the left or right. The result of evolution for a larger number of steps with the particular rule shown here is given as example (f) on the next page.

Much as for cellular automata, one can enumerate all possible rules of this kind; it turns out that there are 65,536 of them. The pictures at the top of the next page show typical behavior obtained with such rules. In cases (a) and (b), the active cell remains localized to a small region, and the behavior is very simple and repetitive. Cases (c) through (f) are similar,

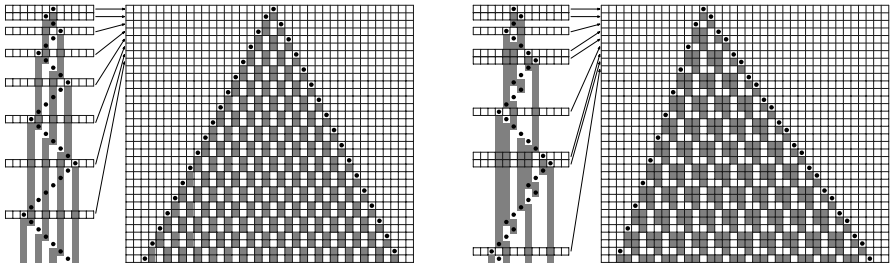


Examples of mobile automata with various rules. In cases (a) through (f) the motion of the active cell is purely repetitive. In cases (g) and (h) it is not. The width of the pattern in these cases after t steps grows roughly like $\sqrt{2t}$.

except that the whole pattern shifts systematically to the right, and in cases (e) and (f) a sequence of stripes is left behind.

But with a total of 218 out of the 65,536 possible rules, one gets somewhat different behavior, as cases (g) and (h) above show. The active cell in these cases does not move in a strictly repetitive way, but instead sweeps backwards and forwards, going progressively further every time.

The overall pattern produced is still quite simple, however. And indeed in the compressed form below, it is purely repetitive.

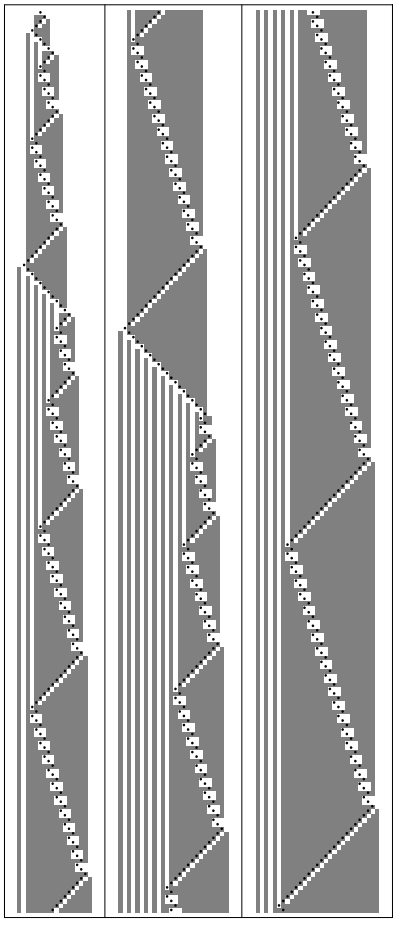


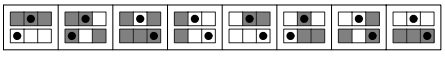
Compressed versions of the evolution of mobile automata (g) and (h) above, obtained by showing only those steps at which the active cell is further to the left or right than it has ever been before.

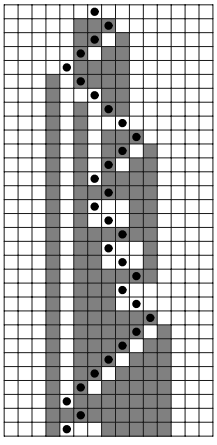
Of the 65,536 possible mobile automata with rules of the kind discussed so far it turns out that not a single one shows more complex behavior. So can such behavior then ever occur in mobile automata?

One can extend the set of rules one considers by allowing not only the color of the active cell itself but also the colors of its immediate neighbors to be updated at each step. And with this extension, there are a total of 4,294,967,296 possible rules.

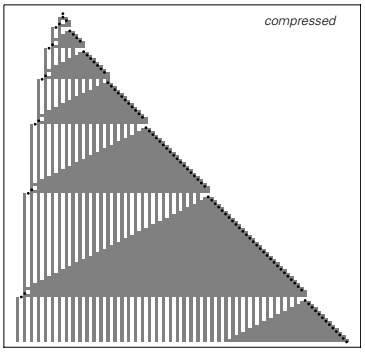
If one samples these rules at random, one finds that more than 99% of them just yield simple repetitive behavior. But once in every few thousand rules, one sees behavior of the kind shown below—that is not purely repetitive, but instead has a kind of nested structure.



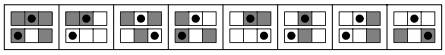




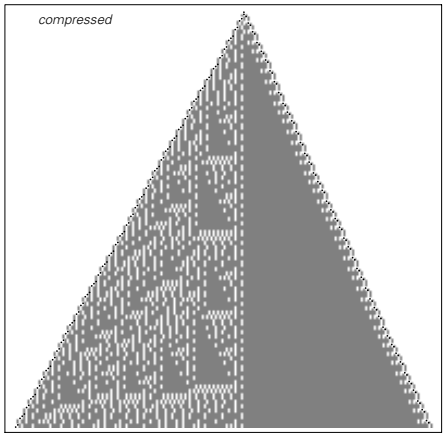
A mobile automaton with slightly more complicated rules that yields a nested pattern. Each column on the left shows 200 steps in the mobile automaton evolution. The compressed form of the pattern is based on a total of 8000 steps.

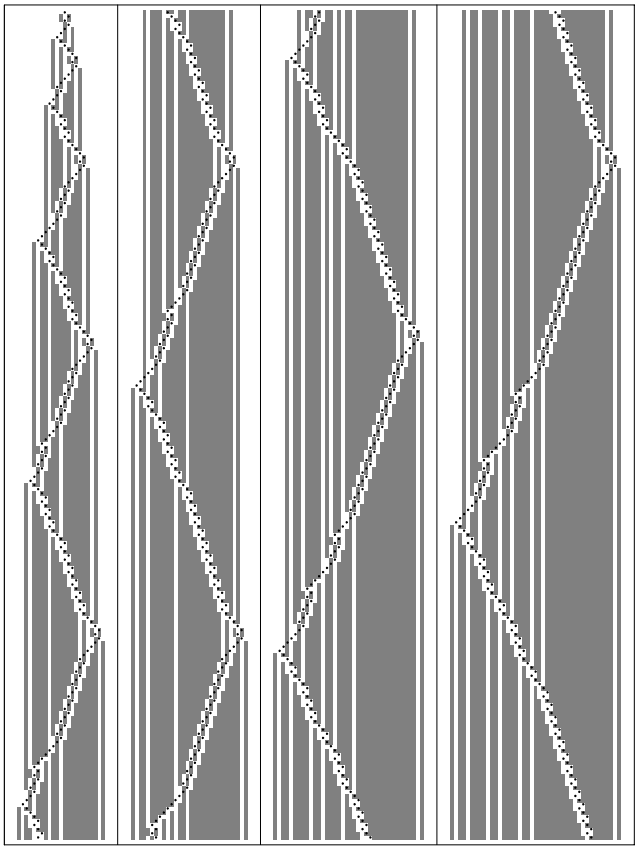


The overall pattern is nevertheless still very regular. But after searching through perhaps 50,000 rules, one finally comes across a rule of the kind shown below—in which the compressed pattern exhibits very much the same kind of apparent randomness that we saw in cellular automata like rule 30.

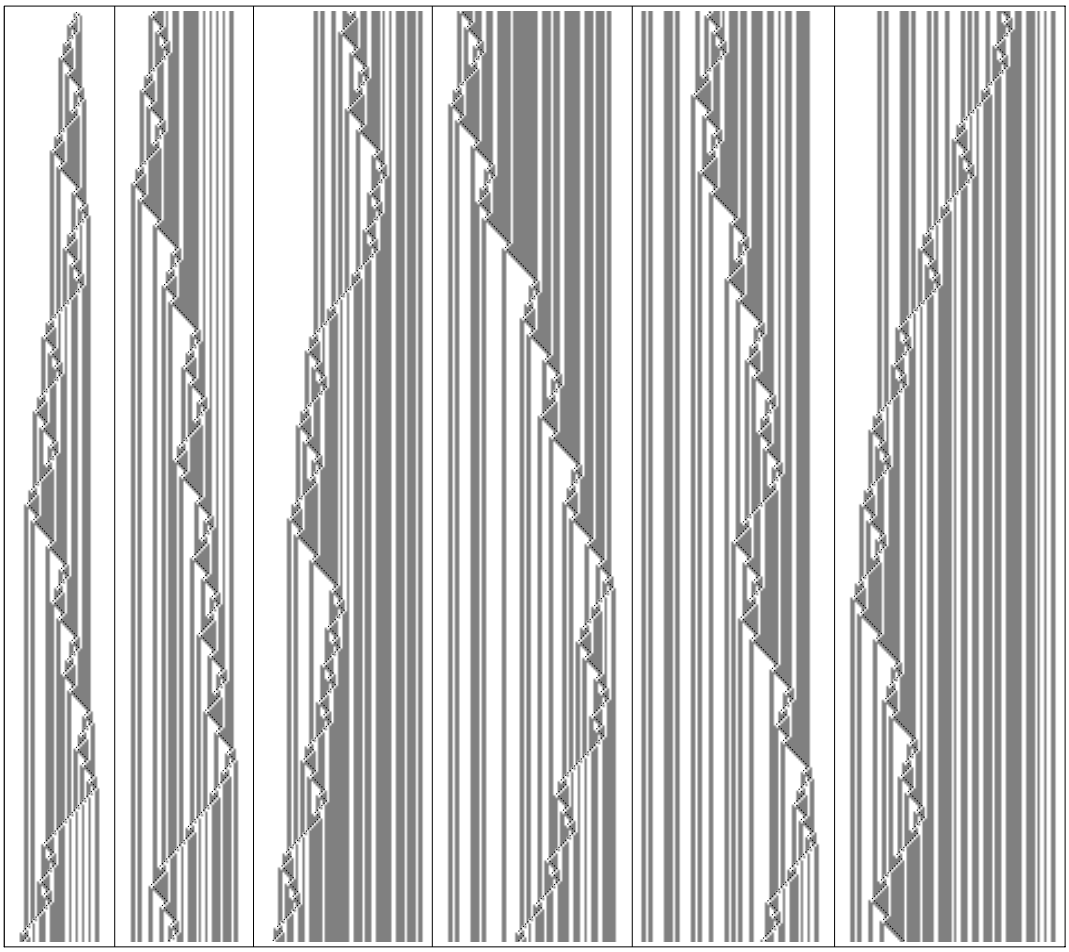


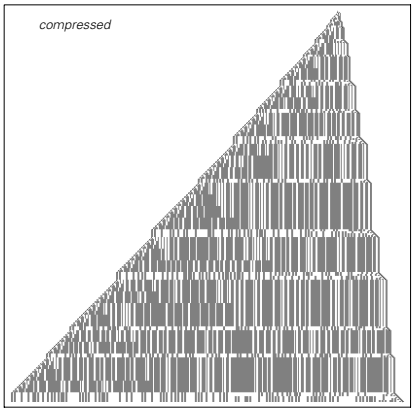
A mobile automaton that yields a pattern with seemingly random features. The motion of the active cell is still quite regular, as the picture on the right shows. But when viewed in compressed form, as below, the overall pattern of colors seems in many respects random. Each column on the right shows 200 steps of evolution; the compressed form below corresponds to 50,000 steps.

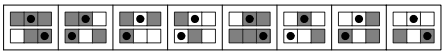




But even though the final pattern left behind by the active cell in the picture above seems in many respects random, the motion of the active cell itself is still quite regular. So are there mobile automata in which the motion of the active cell is also seemingly random? At first, I believed that there might not be. But after searching through a few million rules, I finally found the example shown on the facing page.







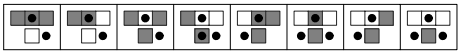
A mobile automaton in which the position of the active cell moves in a seemingly random way. Each column above shows 400 steps; the compressed form corresponds to 50,000 steps. It took searching through a few million mobile automata to find one with behavior as complex as what we see here.

Despite the fact that mobile automata update only one cell at a time, it is thus still possible for them to produce behavior of great complexity. But while we found that such behavior is quite common in cellular automata, what we have seen in this section indicates that it is rather rare in mobile automata.

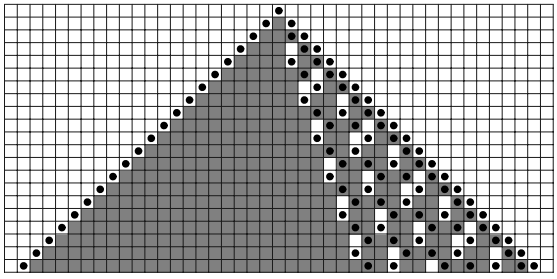
One can get some insight into the origin of this difference by studying a class of generalized mobile automata, that in a sense interpolate between ordinary mobile automata and cellular automata.

The basic idea of such generalized mobile automata is to allow more than one cell to be active at a time. And the underlying rule is then typically set up so that under certain circumstances an active cell can split in two, or can disappear entirely.

Thus in the picture below, for example, new active cells end up being created every few steps.

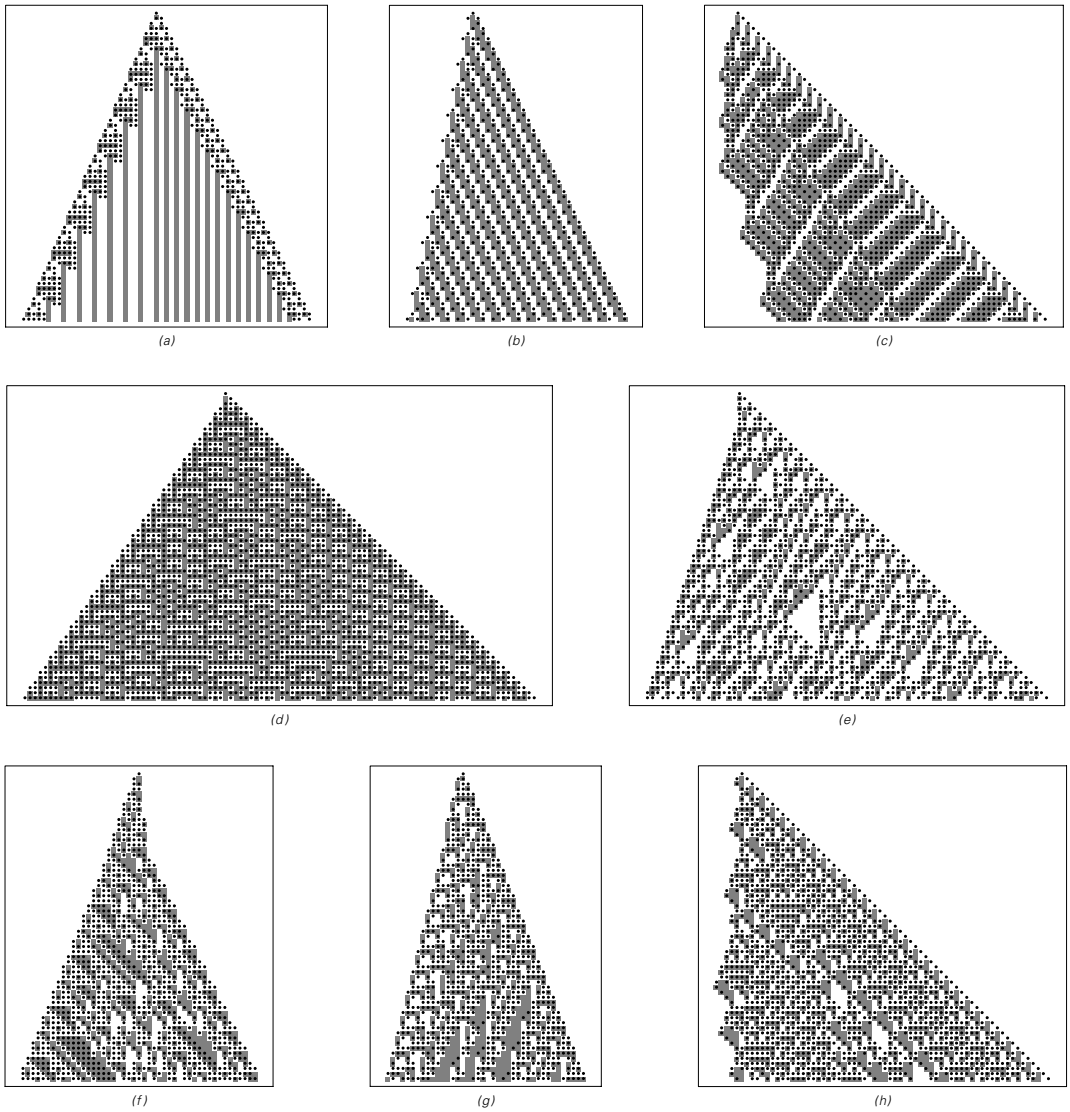


A generalized mobile automaton in which any number of cells can be active at a time. The rule given above is applied to every cell that is active at a particular step. In many cases, the rule specifies just that the active cell should move to the left or right. But in some cases, it specifies that the active cell should split in two, thereby creating an additional active cell.



If one chooses generalized mobile automata at random, most of them will produce simple behavior, as shown in the first few pictures on the facing page. But in a few percent of all cases, the behavior is much more complicated. Often the arrangement of active cells is still quite regular, although sometimes it is not.

But looking at many examples, a certain theme emerges: complex behavior almost never occurs except when large numbers of cells are active at the same time. Indeed there is, it seems, a significant correlation between overall activity and the likelihood of complex behavior. And this is part of why complex behavior is so much more common in cellular automata than in mobile automata.



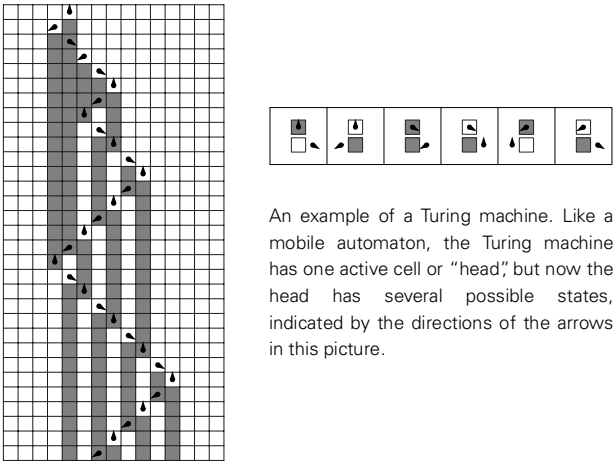
Examples of generalized mobile automata with various rules. In case (a), only a limited number of cells ever become active. But in all the other cases shown active cells proliferate forever. In case (d), almost all cells are active, and the system operates essentially like a cellular automaton. In the remaining cases somewhat complicated patterns of cells are active. Note that unlike in ordinary mobile automata, examples of complex behavior like those shown here are comparatively easy to find.

Turing Machines

In the history of computing, the first widely understood theoretical computer programs ever constructed were based on a class of systems now called Turing machines.

Turing machines are similar to mobile automata in that they consist of a line of cells, known as the "tape", together with a single active cell, known as the "head". But unlike in a mobile automaton, the head in a Turing machine can have several possible states, represented by several possible arrow directions in the picture below.

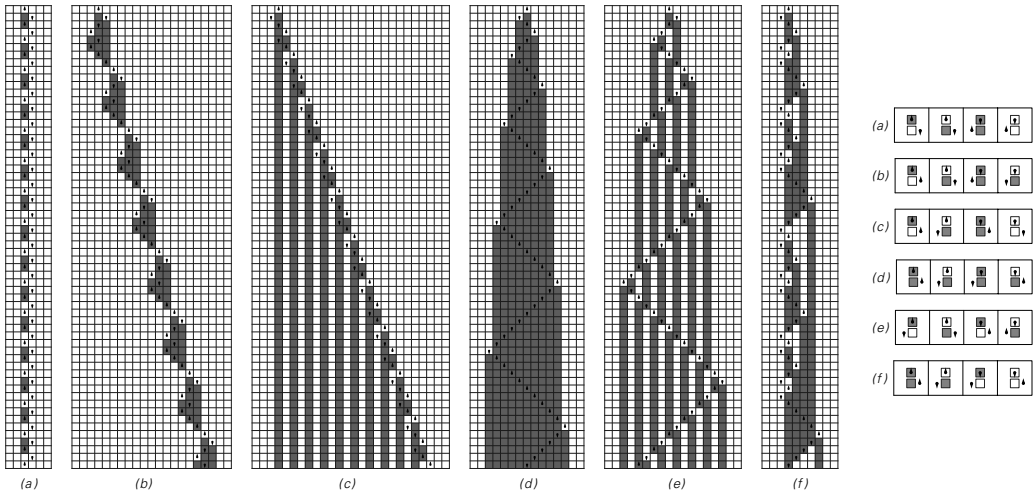
And in addition, the rule for a Turing machine can depend on the state of the head, and on the color of the cell at the position of the head, but not on the colors of any neighboring cells.



An example of a Turing machine. Like a mobile automaton, the Turing machine has one active cell or "head", but now the head has several possible states, indicated by the directions of the arrows in this picture.

Turing machines are still widely used in theoretical computer science. But in almost all cases, one imagines constructing examples to perform particular tasks, with a huge number of possible states and a huge number of possible colors for each cell.

But in fact there are non-trivial Turing machines that have just two possible states and two possible colors for each cell. The pictures on the facing page show examples of some of the 4096 machines of this kind. Both repetitive and nested behavior are seen to occur, though nothing more complicated is found.



Examples of Turing machines with two possible states for the head. There are a total of 4096 rules of this kind. Repetitive and nested patterns are seen, but nothing more complicated ever occurs.

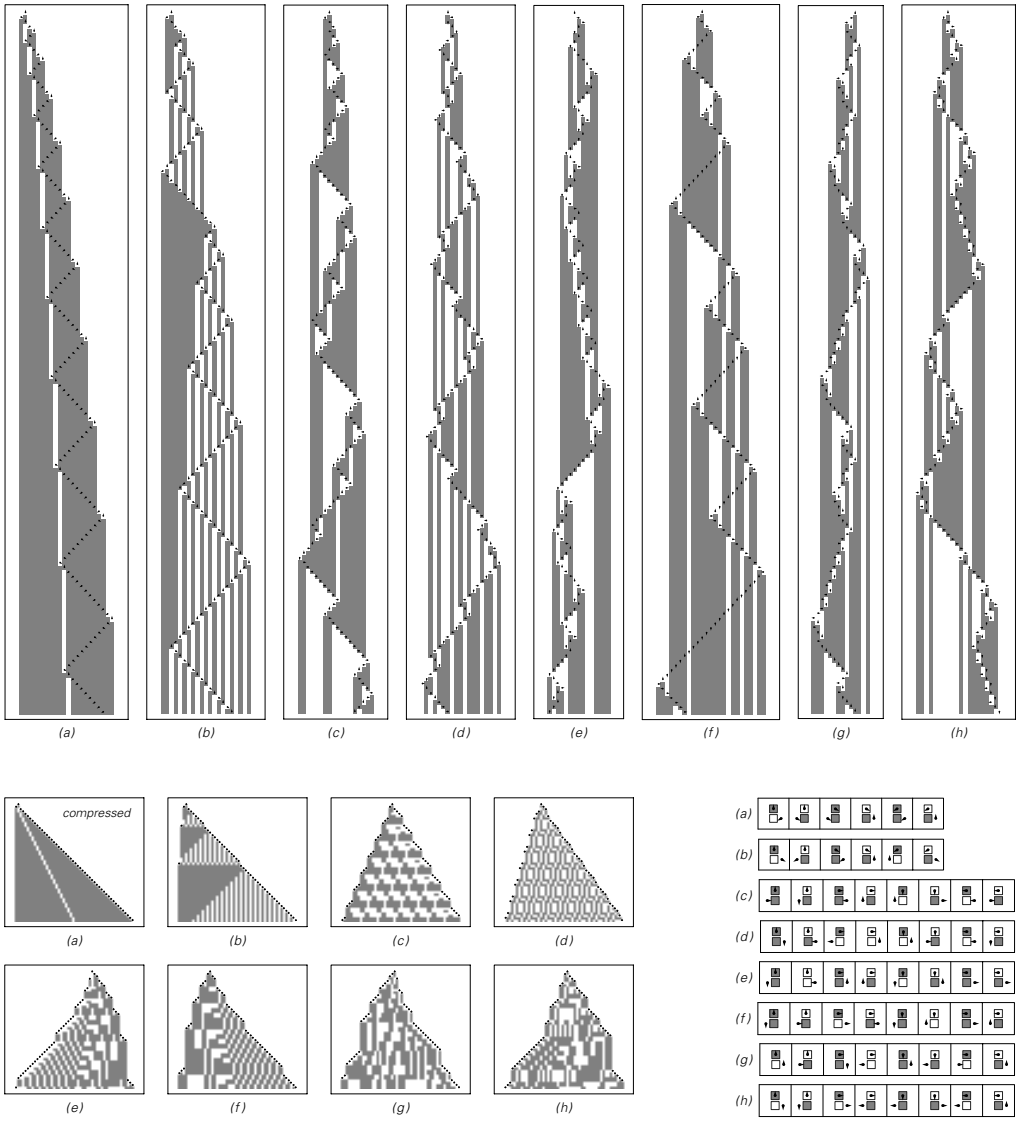
From our experience with mobile automata, however, we expect that there should be Turing machines that have more complex behavior.

With three states for the head, there are about three million possible Turing machines. But while some of these give behavior that looks slightly more complicated in detail, as in cases (a) and (b) on the next page, all ultimately turn out to yield just repetitive or nested patterns—at least if they are started with all cells white.

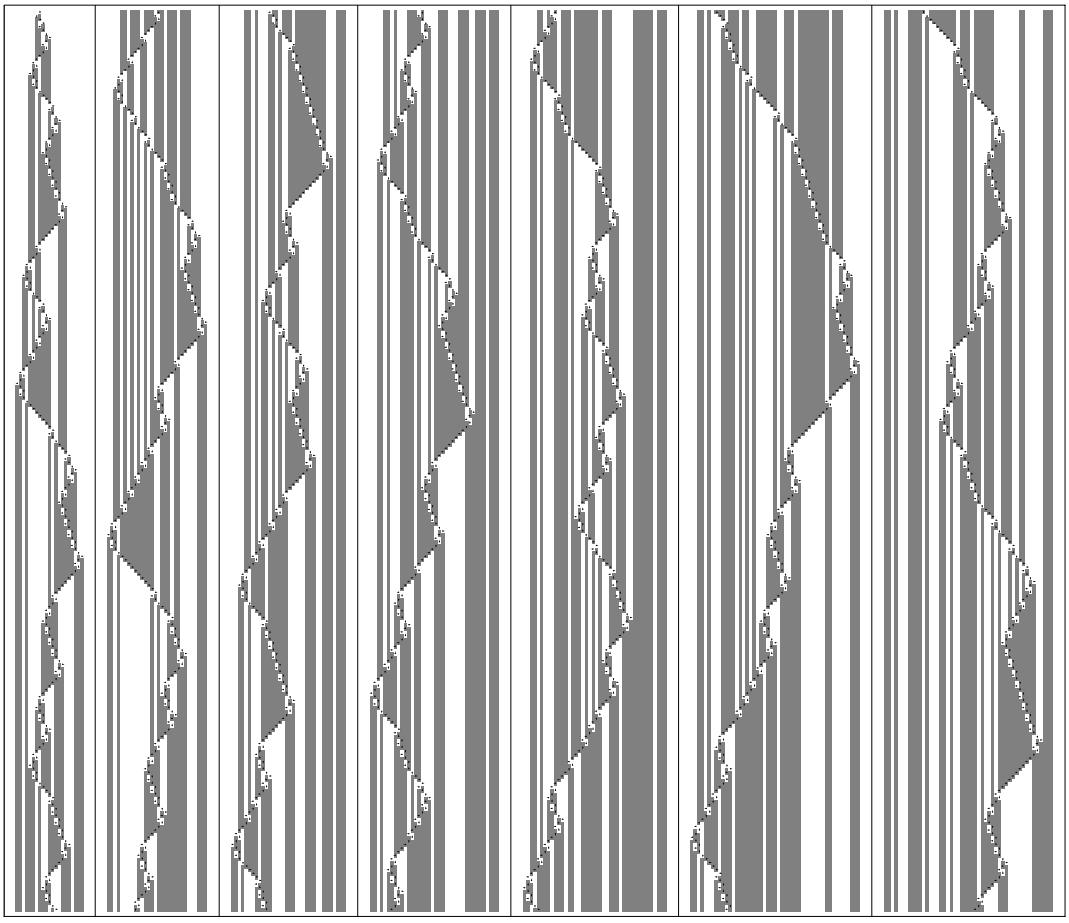
With four states, however, more complicated behavior immediately becomes possible. Indeed, in about five out of every million rules of this kind, one gets patterns with features that seem in many respects random, as in the pictures on the next two pages.

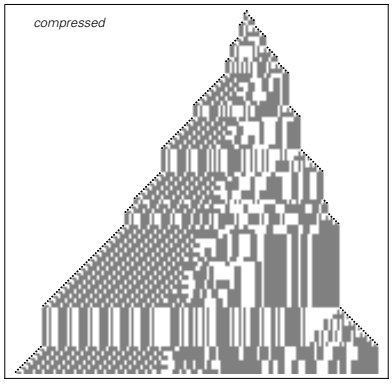
So what happens if one allows more than four states for the head? It turns out that there is almost no change in the kind of behavior one sees. Apparent randomness becomes slightly more common, but otherwise the results are essentially the same.

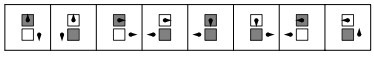
Once again, it seems that there is a threshold for complex behavior—that is reached as soon as one has at least four states. And just as in cellular automata, adding more complexity to the underlying rules does not yield behavior that is ultimately any more complex.



Examples of Turing machines with three and four possible states. With three possible states, only repetitive and nested patterns are ever ultimately produced, at least starting with all cells white. But with four states, more complicated patterns are generated. The top set of pictures show the first 150 steps of evolution according to various different rules, starting with the head in the first state (arrow pointing up), and all cells white. The bottom set of pictures show the evolution in each case in a compressed form. Each of these pictures includes the first 50 steps at which the head is further to the left or right than it has ever been before.







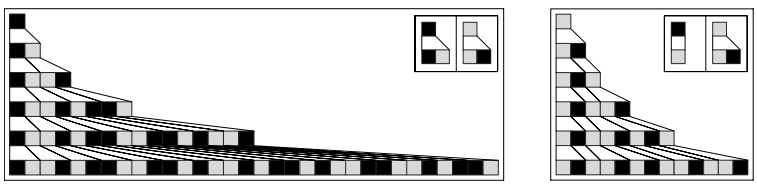
A Turing machine that exhibits behavior which seems in many respects random. The Turing machine has four possible states for its head, and two possible colors for each cell on its tape. It starts with all cells white, corresponding to a blank tape. Each column above shows 250 steps of evolution; the compressed form on the left corresponds to a total of 20,000 steps.

Substitution Systems

One of the features that cellular automata, mobile automata and Turing machines all have in common is that at the lowest level they consist of a fixed array of cells. And this means that while the colors of these cells can be updated according to a wide range of different possible rules, the underlying number and organization of cells always stays the same.

Substitution systems, however, are set up so that the number of elements can change. In the typical case illustrated below, one has a sequence of elements—each colored say black or white—and at each step each one of these elements is replaced by a new block of elements.

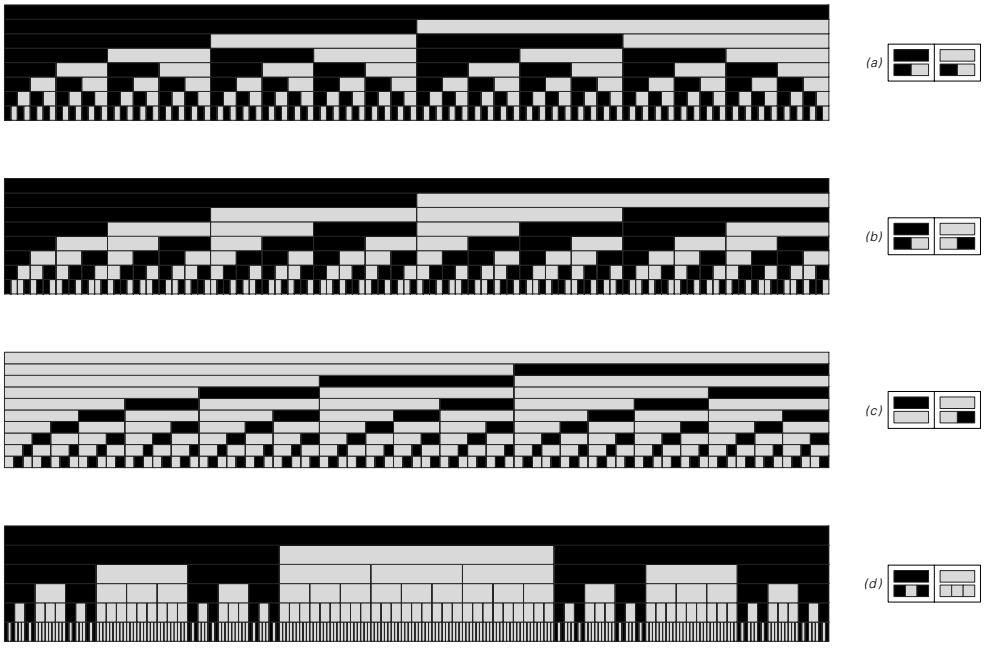
In the simple cases shown, the rules specify that each element of a particular color should be replaced by a fixed block of new elements, independent of the colors of any neighboring elements.



Examples of substitution systems with two possible kinds of elements, in which at every step each kind of element is replaced by a fixed block of new elements. In the first case shown, the total number of elements obtained doubles at every step; in the second case, it follows a Fibonacci sequence, and increases by a factor of roughly $(1+\sqrt{5})/2 \approx 1.618$ at every step. The two substitution systems shown here correspond to the second and third examples in the pictures on the following two pages.

And with these kinds of rules, the total number of elements typically grows very rapidly, so that pictures like those above quickly become rather unwieldy. But at least for these kinds of rules, one can make clearer pictures by thinking of each step not as replacing every element by a sequence of elements that are drawn the same size, but rather of subdividing each element into several that are drawn smaller.

In the cases on the facing page, I start from a single element represented by a long box going all the way across the picture. Then on successive steps the rules for the substitution system specify how each box should be subdivided into a sequence of shorter and shorter boxes.

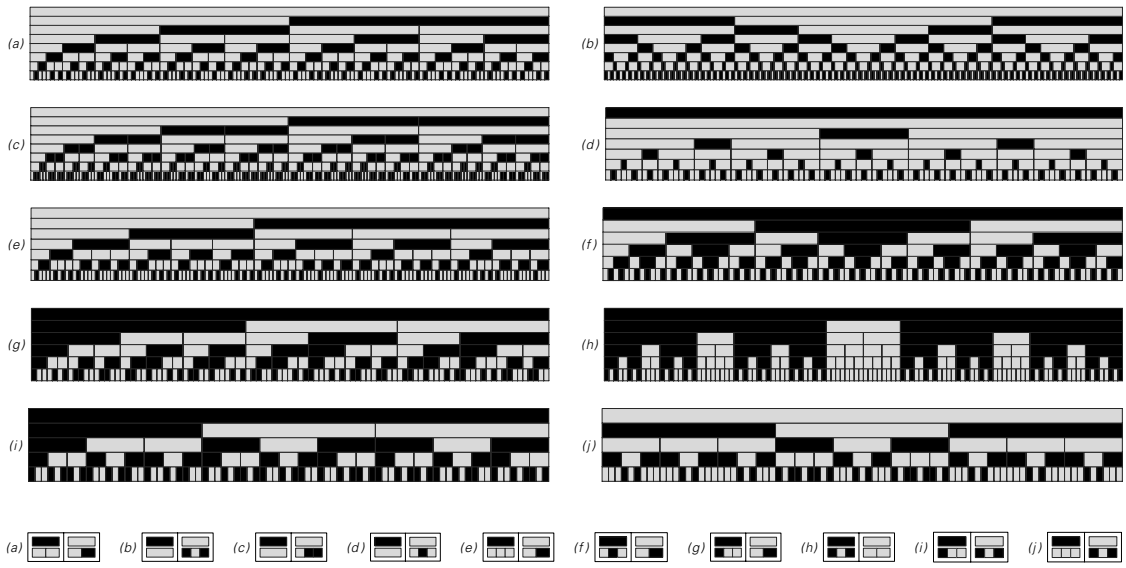


Examples of substitution systems in which every element is drawn as being subdivided into a sequence of new elements at each step. In all cases the overall patterns obtained can be seen to have a very regular nested form. Rule (b) gives the so-called Thue-Morse sequence, which we will encounter many times in this book. Rule (c) is related to the Fibonacci sequence. Rule (d) gives a version of the Cantor set.

The pictures at the top of the next page show a few more examples. And what we see is that in all cases there is obvious regularity in the patterns produced. Indeed, if one looks carefully, one can see that every pattern just consists of a collection of identical nested pieces.

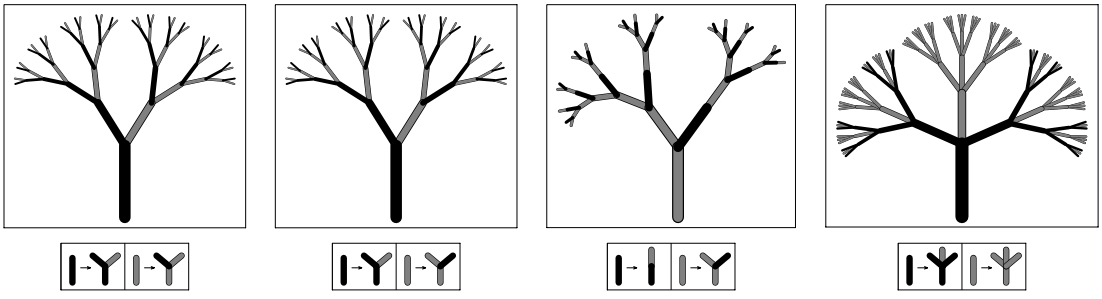
And ultimately this is not surprising. After all, the basic rules for these substitution systems specify that any time an element of a particular color appears it will always get subdivided in the same way.

The nested structure becomes even clearer if one represents elements not as boxes, but instead as branches on a tree. And with this setup the idea is to start from the trunk of the tree, and then at each step to use the rules for the substitution system to determine how every branch should be split into smaller branches.



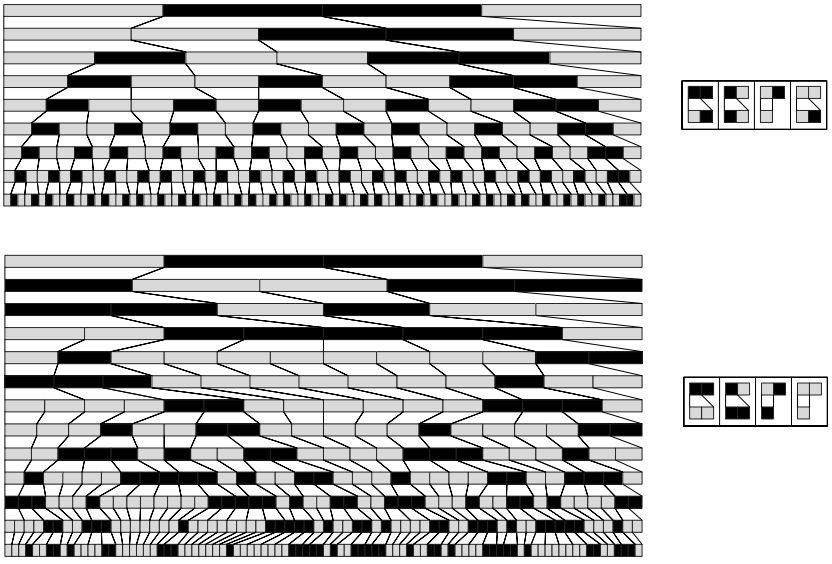
More examples of neighbor-independent substitution systems like those on the previous page. Each rule yields a different sequence of elements, but all of them ultimately have simple nested forms.

Then the point is that because the rules depend only on the color of a particular branch, and not on the colors of any neighboring branches, the subtrees that are generated from all the branches of the same color must have exactly the same structure, as in the pictures below.



The evolution of the same substitution systems as on the previous page, but now shown in terms of trees. Starting from the trunk at the bottom, the rules specify that at each step every branch of a particular color should split into smaller branches in the same way. The result is that each tree consists of a collection of progressively smaller subtrees with the same structure. On page 400 I will use similar systems to discuss the growth of actual trees and leaves.

To get behavior that is more complicated than simple nesting, it follows therefore that one must consider substitution systems whose rules depend not only on the color of a single element, but also on the color of at least one of its neighbors. The pictures below show examples in which the rules for replacing an element depend not only on its own color, but also on the color of the element immediately to its right.



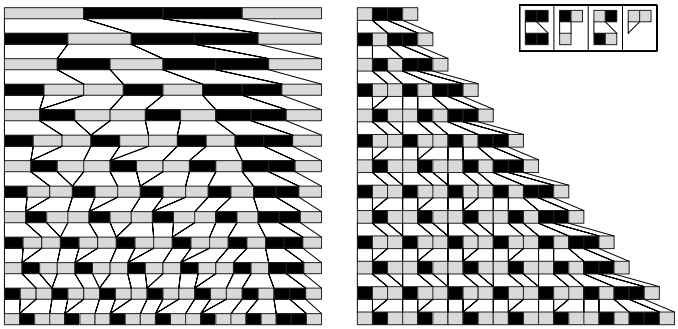
Examples of substitution systems whose rules depend not just on the color of an element itself, but also on the color of the element immediately to its right. Rules of this kind cannot readily be interpreted in terms of simple subdivision of one element into several. And as a result, there is no obvious way to choose what size of box should be used to represent each element in the picture. What I do here is simply to divide the whole width of the picture equally among all elements that appear at each step. Note that on every step the rightmost element is always dropped, since no rule is given for how to replace it.

In the first example, the pattern obtained still has a simple nested structure. But in the second example, the behavior is more complicated, and there is no obvious nested structure.

One feature of both examples, however, is that the total number of elements never decreases from one step to the next. The reason for this is that the basic rules we used specify that every single element should be replaced by at least one new element.

It is, however, also possible to consider substitution systems in which elements can simply disappear. If the rate of such disappearances is too large, then almost any pattern will quickly die out. And if there are too few disappearances, then most patterns will grow very rapidly.

But there is always a small fraction of rules in which the creation and destruction of elements is almost perfectly balanced.

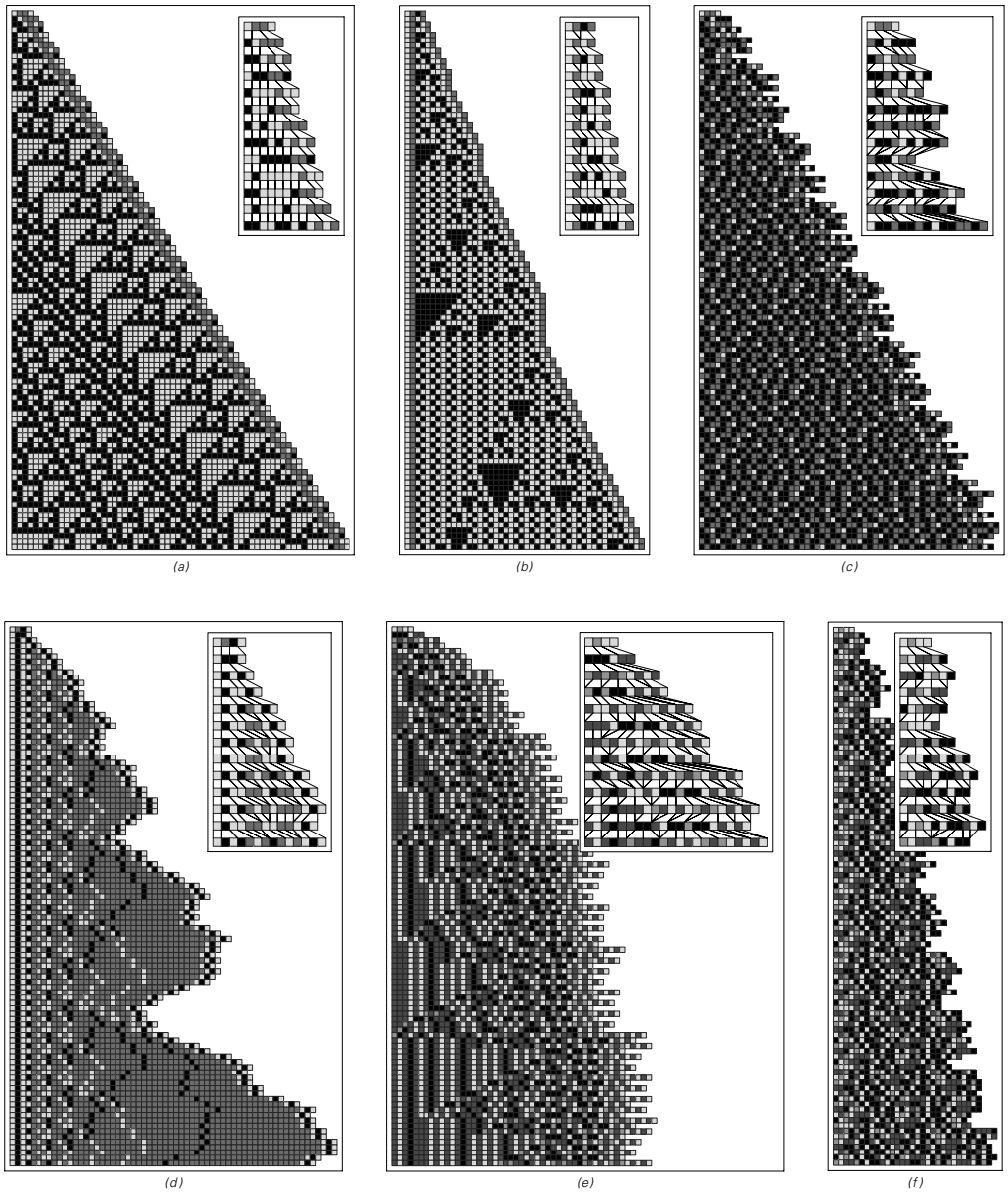


Two views of a substitution system whose rules allow both creation and destruction of elements. In the view on the left, the boxes representing each element are scaled to keep the total width the same, whereas on the right each box has a fixed size, as in our original pictures of substitution systems on page 82. The right-hand view shows that the rates of creation and destruction of elements are balanced closely enough that the total number of elements grows by only a fixed amount at each step.

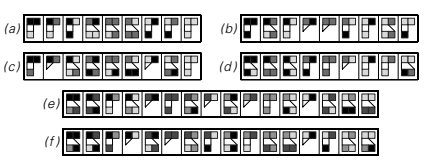
The picture above shows one example. The number of elements does end up increasing in this particular example, but only by a fixed amount at each step. And with such slow growth, we can again represent each element by a box of the same size, just as in our original pictures of substitution systems on page 82.

When viewed in this way, however, the pattern produced by the substitution system shown above is seen to have a simple repetitive form. And as it turns out, among substitution systems with the same type of rules, all those which yield slow growth also seem to produce only such simple repetitive patterns.

Knowing this, we might conclude that somehow substitution systems just cannot produce the kind of complexity that we have seen in systems like cellular automata. But as with mobile automata and with Turing machines, we would again be wrong. Indeed, as the pictures on the facing page demonstrate, allowing elements to have three or four colors rather than just two immediately makes much more complicated behavior possible.



Examples of substitution systems that have three and four possible colors for each element. The particular rules shown are ones that lead to slow growth in the total number of elements. Note that on each line in each picture, only the order of elements is ever significant: as the insets show, a particular element may change its position as a result of the addition or subtraction of elements to its left. Note that the pattern in case (a) does eventually repeat, while the one in case (b) eventually shows a nested structure.



As it turns out, the first substitution system shown works almost exactly like a cellular automaton. Indeed, away from the right-hand edge, all the elements effectively behave as if they were lying on a regular grid, with the color of each element depending only on the previous color of that element and the element immediately to its right.

The second substitution system shown again has patches that exhibit a regular grid structure. But between these patches, there are regions in which elements are created and destroyed. And in the other substitution systems shown, elements are created and destroyed throughout, leaving no trace of any simple grid structure. So in the end the patterns we obtain can look just as random as what we have seen in systems like cellular automata.

Sequential Substitution Systems

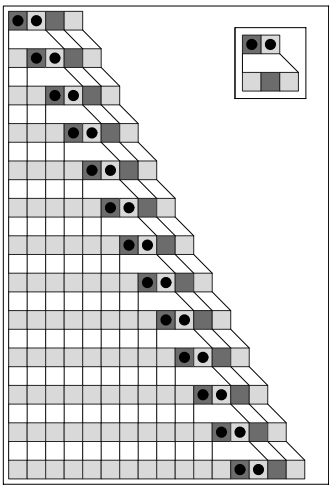
None of the systems we have discussed so far in this chapter might at first seem much like computer programs of the kind we typically use in practice. But it turns out that there are for example variants of substitution systems that work essentially just like standard text editors.

The first step in understanding this correspondence is to think of substitution systems as operating not on sequences of colored elements but rather on strings of elements or letters. Thus for example the state of a substitution system at a particular step can be represented by the string ABBBABA, where the A's correspond to white elements and the B's to black ones.

The substitution systems that we discussed in the previous section work by replacing each element in such a string by a new sequence of elements—so that in a sense these systems operate in parallel on all the elements that exist in the string at each step.

But it is also possible to consider sequential substitution systems, in which the idea is instead to scan the string from left to right, looking for a particular sequence of elements, and then to perform a replacement for the first such sequence that is found. And this setup is now directly analogous to the search-and-replace function of a typical text editor.

The picture below shows an example of a sequential substitution system in which the rule specifies simply that the first sequence of the form BA found at each step should be replaced with the sequence ABA.

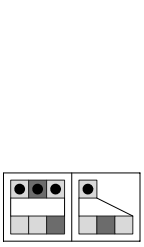


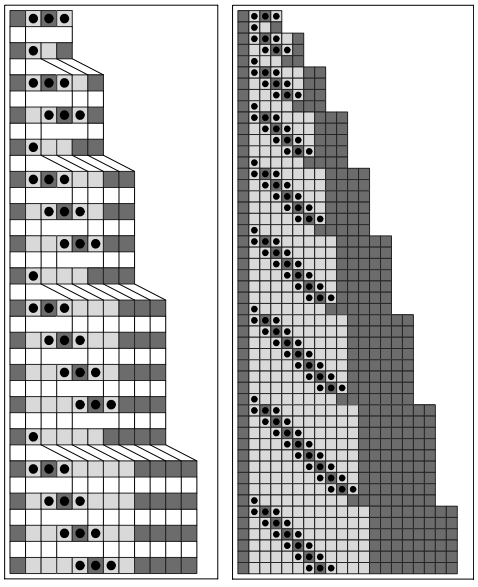
An example of a very simple sequential substitution system. The light squares can be thought of as corresponding to the element *A*, and the dark squares to the element *B*. At each step, the rule then specifies that the string which exists at that step should be scanned from left to right, and the first sequence *BA* that is found should be replaced by *ABA*. In the picture, the black dots indicate which elements are being replaced at each step. In the case shown, the initial string is *BABA*. At each step, the rule then has the effect of adding an *A* inside the string.

The behavior in this case is very simple, with longer and longer strings of the same form being produced at each step. But one can get more complicated behavior if one uses rules that involve more than just one possible replacement. The idea in this case is at each step to scan the string repeatedly, trying successive replacements on successive scans, and stopping as soon as a replacement that can be used is found.

The picture on the next page shows a sequential substitution system with rule {ABA → AAB, A → ABA} involving two possible replacements. Since the sequence ABA occurs in the initial string that is given, the first replacement is used on the first step. But the string BAAB that is produced at the second step does not contain ABA, so now the first replacement cannot be used. Nevertheless, since the string does contain the single element A, the second replacement can still be used.

Despite such alternation between different replacements, however, the final pattern that emerges is very regular. Indeed, if one allows only two possible replacements—and two possible elements—





A sequential substitution system whose rule involves two possible replacements. At each step, the whole string is scanned once to try to apply the first replacement, and is then scanned again if necessary to try to apply the second replacement.

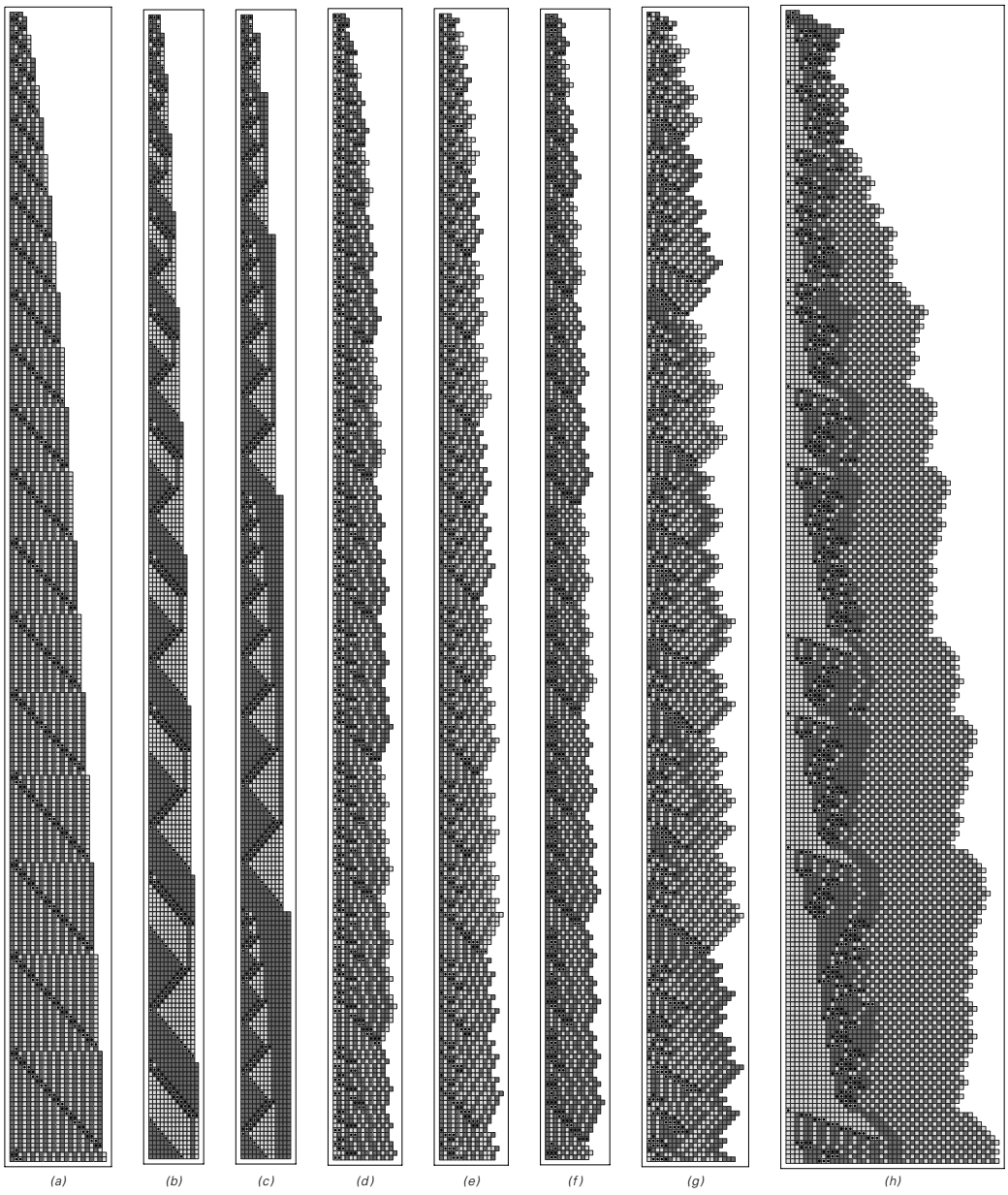
then it seems that no rule ever gives behavior that is much more complicated than in the picture above.

And from this one might be led to conclude that sequential substitution systems could never produce behavior of any substantial complexity. But having now seen complexity in many other kinds of systems, one might suspect that it should also be possible in sequential substitution systems.

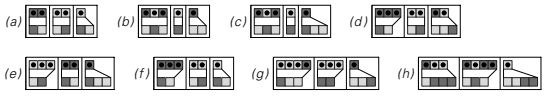
And it turns out that if one allows more than two possible replacements then one can indeed immediately get more complex behavior. The pictures on the facing page show a few examples. In many cases, fairly regular repetitive or nested patterns are still produced.

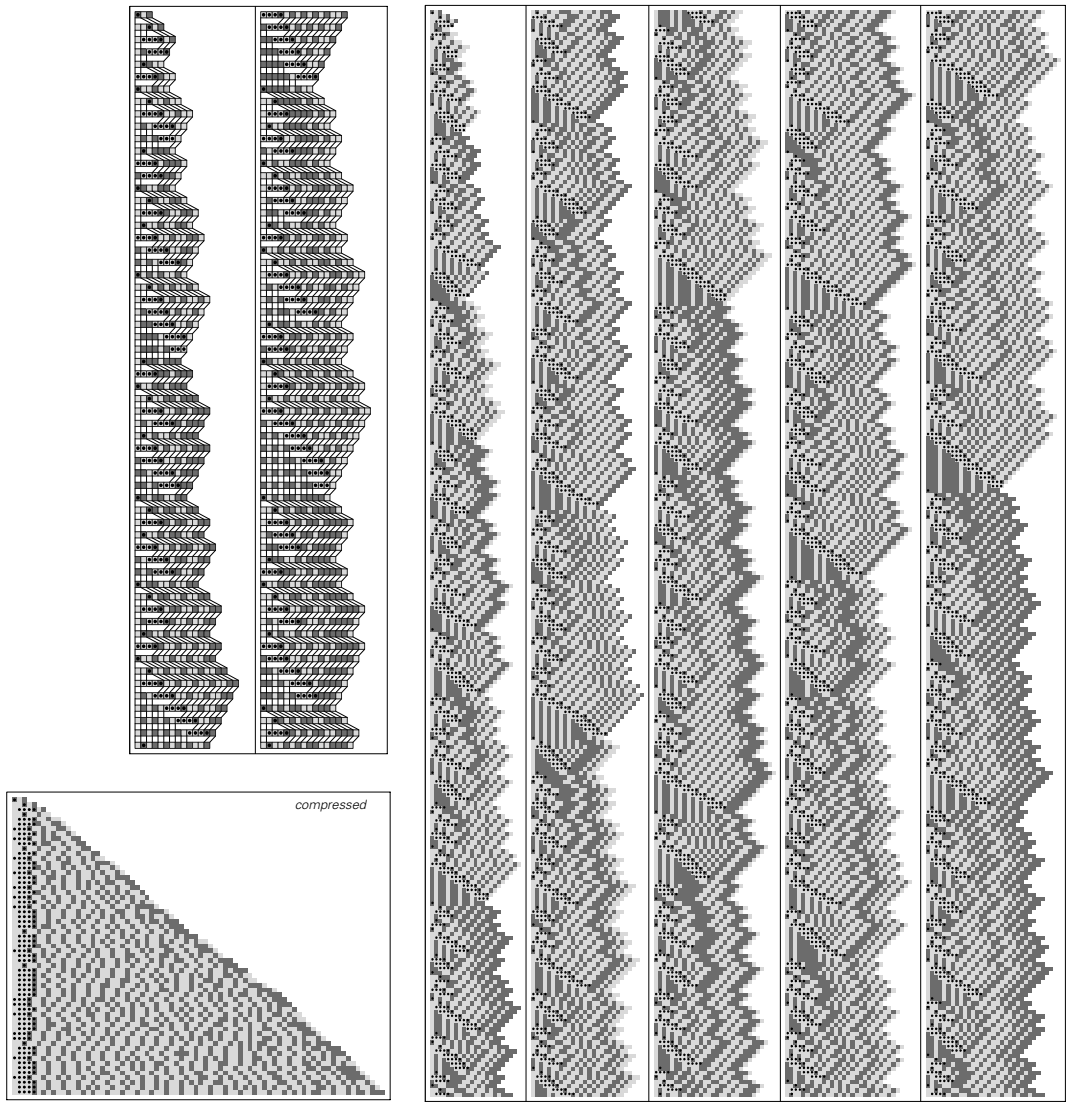
But about once in every 10,000 randomly selected rules, rather different behavior is obtained. Indeed, as the picture on the following page demonstrates, patterns can be produced that seem in many respects random, much like patterns we have seen in cellular automata and other systems.

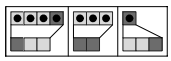
So this leads to the rather remarkable conclusion that just by using the simple operations available even in a very basic text editor, it is still ultimately possible to produce behavior of great complexity.



Examples of sequential substitution systems whose rules involve three possible replacements. In all cases, the systems are started from the initial string *BAB*. The black dots indicate the elements that are replaced at each step.







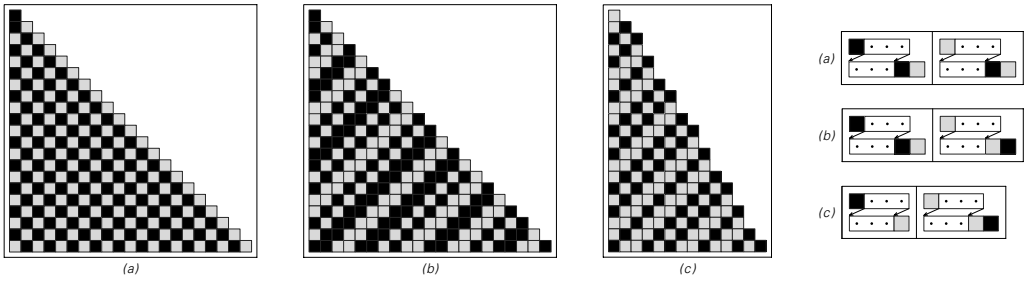
An example of a sequential substitution system that yields apparently random behavior. Each column on the right-hand side shows the evolution of the system for 250 steps. The compressed picture on the left is made by evolving for a million steps, but showing only steps at which the string becomes longer than it has ever been before. (The rule is the same as (g) on the previous page.)

Tag Systems

One of the goals of this chapter is to find out just how simple the underlying structure of a system can be while the system as a whole is still capable of producing complex behavior. And as one example of a class of systems with a particularly simple underlying structure, I consider here what are sometimes known as tag systems.

A tag system consists of a sequence of elements, each colored say black or white. The rules for the system specify that at each step a fixed number of elements should be removed from the beginning of the sequence. And then, depending on the colors of these elements, one of several possible blocks is tagged onto the end of the sequence.

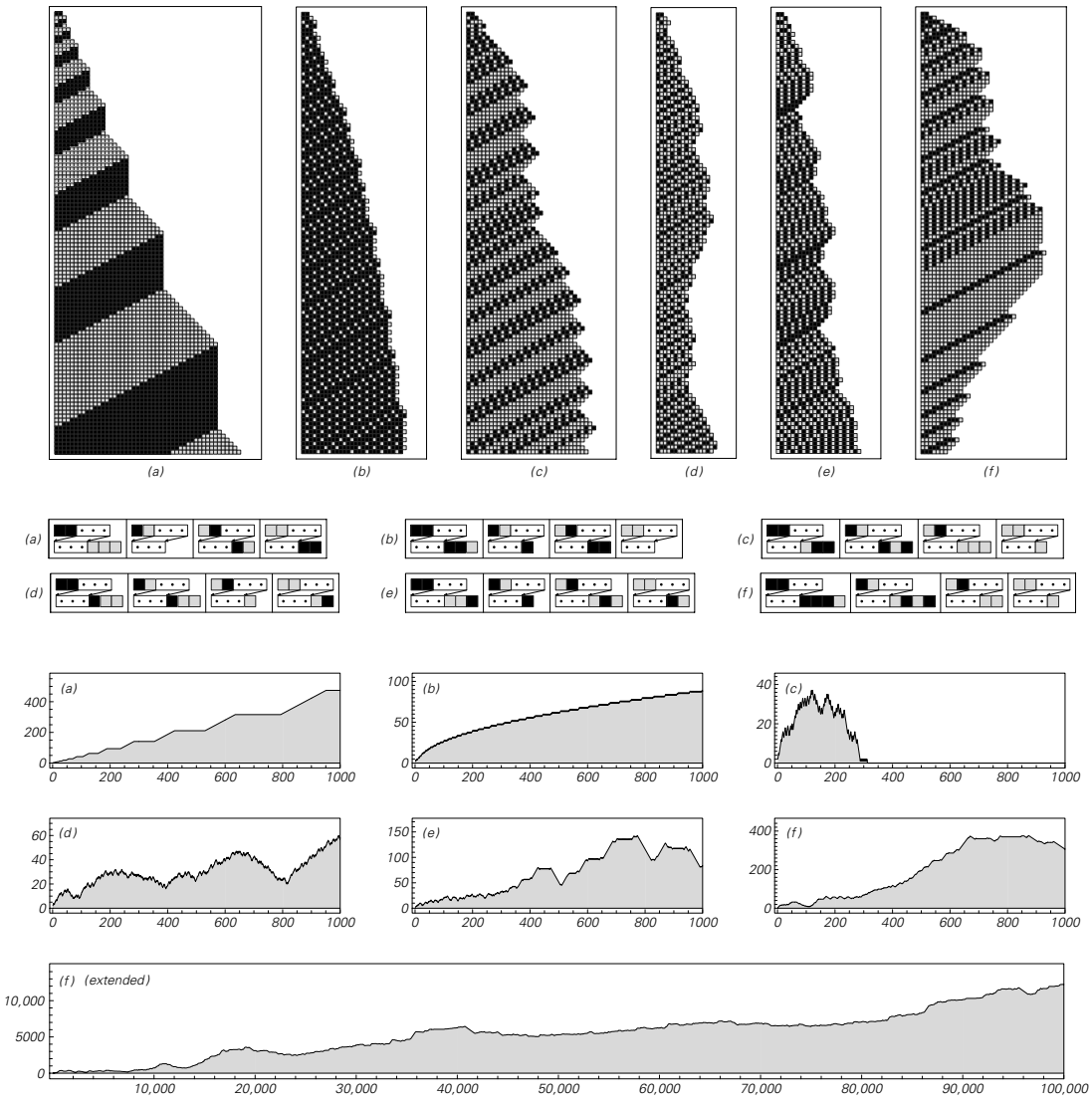
The pictures below show examples of tag systems in which just one element is removed at each step. And already in these systems one sometimes sees behavior that looks somewhat complicated.



Examples of tag systems in which a single element is removed from the beginning of the sequence at each step, and a new block of elements is added to the end of the sequence according to the rules shown. Because only a single element is removed at each step, the systems effectively just cycle through all elements, replacing each one in turn. And after every complete cycle, the sequences obtained correspond exactly to the sequences produced on successive steps in the first three ordinary neighbor-independent substitution systems shown on page 83.

But in fact it turns out that if only one element is removed at each step, then a tag system always effectively acts just like a slow version of a neighbor-independent substitution system of the kind we discussed on page 83. And as a result, the pattern it produces must ultimately have a simple repetitive or nested form.

If two elements are removed at each step, however, then this is no longer true. And indeed, as the pictures on the next page demonstrate, the behavior that is obtained in this case can often be very complicated.

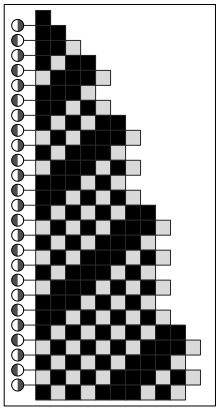


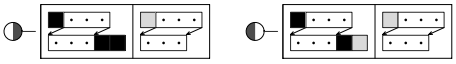
Examples of tag systems in which at each step two elements are removed from the beginning of the sequence and then, based on what these elements are, a specified block of new elements is added to the end of the sequence. (The three dots in the representation of each rule stand for the rest of the elements in the sequence.) The pictures at the top show the first hundred steps in evolution according to various rules starting from a pair of black elements. The plots show the total lengths of the sequences obtained in each case. Note that in case (c), all the elements are eventually removed from the sequence.

Cyclic Tag Systems

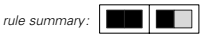
The basic operation of the tag systems that we discussed in the previous section is extremely simple. But it turns out that by using a slightly different setup one can construct systems whose operation is in some ways even simpler. In an ordinary tag system, one does not know in advance which of several possible blocks will be added at each step. But the idea of a cyclic tag system is to make the underlying rule already specify exactly what block can be added at each step.

In the simplest case there are two possible blocks, and the rule simply alternates on successive steps between these blocks, adding a block at a particular step when the first element in the sequence at that step is black. The picture below shows an example of how this works.



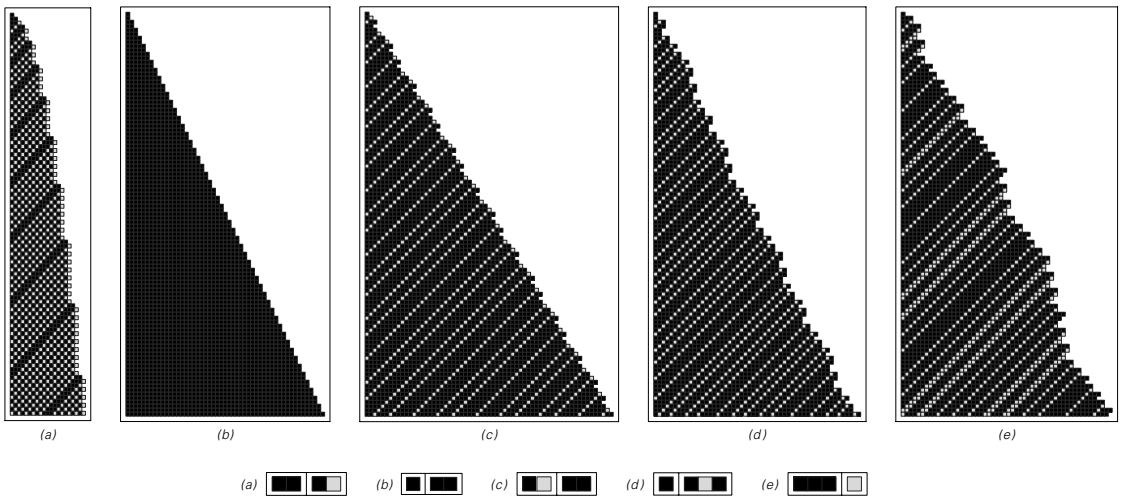


An example of a cyclic tag system. There are two cases in the rule, and these cases are used on alternate steps, as indicated by the circle icons on the left. In each case a single element is removed from the beginning of the sequence, and then a new block is added at the end whenever the element removed is black. The rule can be summarized just by giving the blocks to be used in each case, as shown below.

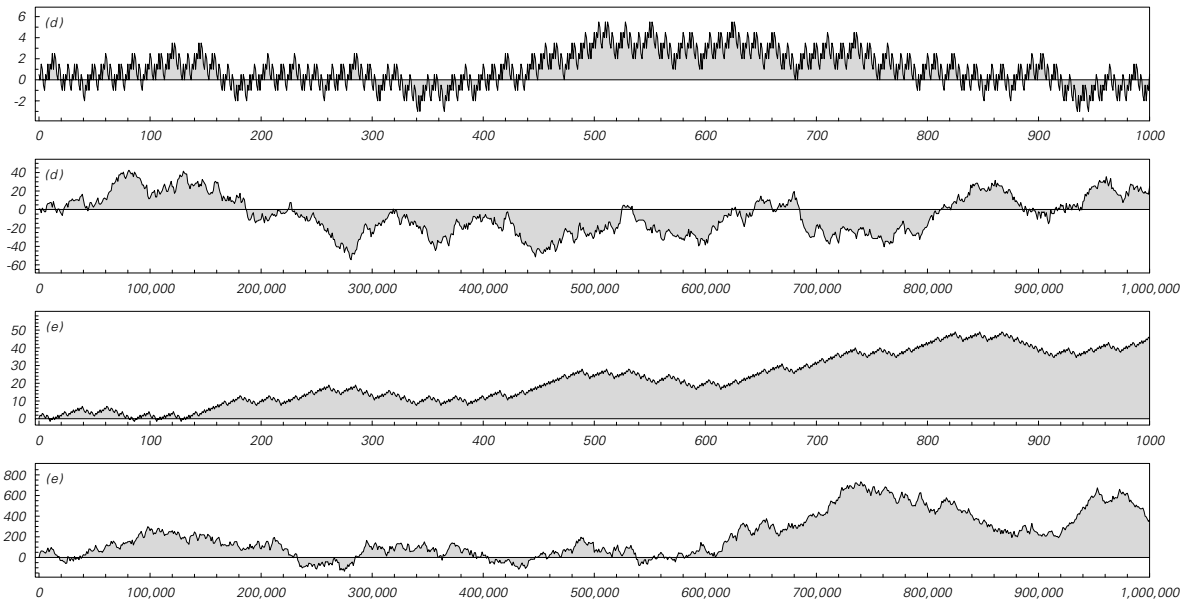


The next page shows examples of several cyclic tag systems. In cases (a) and (b) simple behavior is obtained. In case (c) the behavior is slightly more complicated, but if the pattern is viewed in the appropriate way then it turns out to have the same nested form as the third neighbor-independent substitution system shown on page 83.

So what about cases (d) and (e)? In both of these, the sequences obtained at successive steps grow on average progressively longer. But if one looks at the fluctuations in this growth, as in the plots on the next page, then one finds that these fluctuations are in many respects random.



Examples of cyclic tag systems. In each case the initial condition consists of a single black element. In case (c), alternate steps in the leftmost column (which in all cyclic tag systems determines the overall behavior) have the same nested form as the third neighbor-independent substitution system shown on page 83.



Fluctuations in the growth of sequences for cyclic tag systems (d) and (e) above. The fluctuations are shown with respect to growth at an average rate of half an element per step.

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.

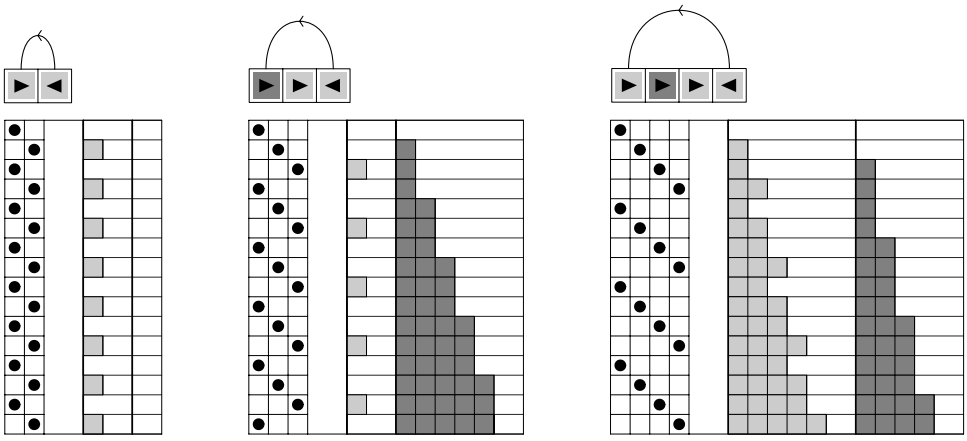
Under most everyday circumstances, the hardware construction of the computers we use is hidden from us by many layers of software. But at the lowest level, the CPUs of all standard computers have registers that store numbers, and any program we write is ultimately converted into a sequence of simple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

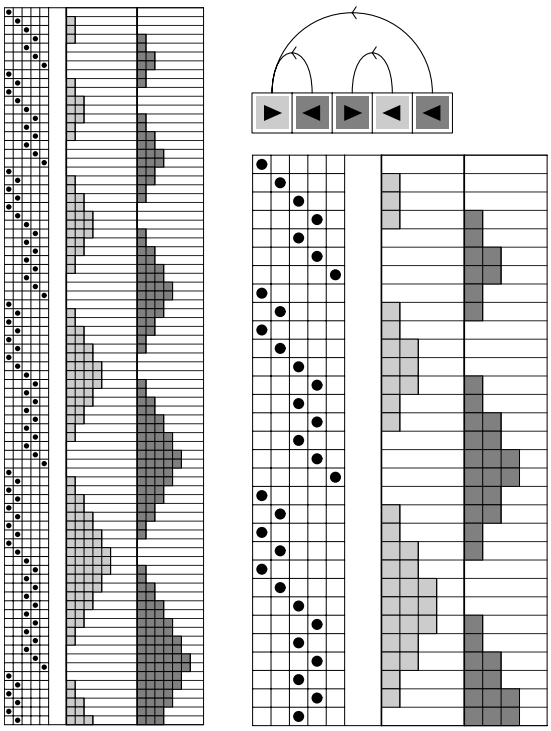
All the register machines shown start by executing the first instruction in their programs. And with the particular programs used here, the machines are then set up just to execute all the other instructions in their programs in turn, jumping back to the beginning of their programs whenever they reach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

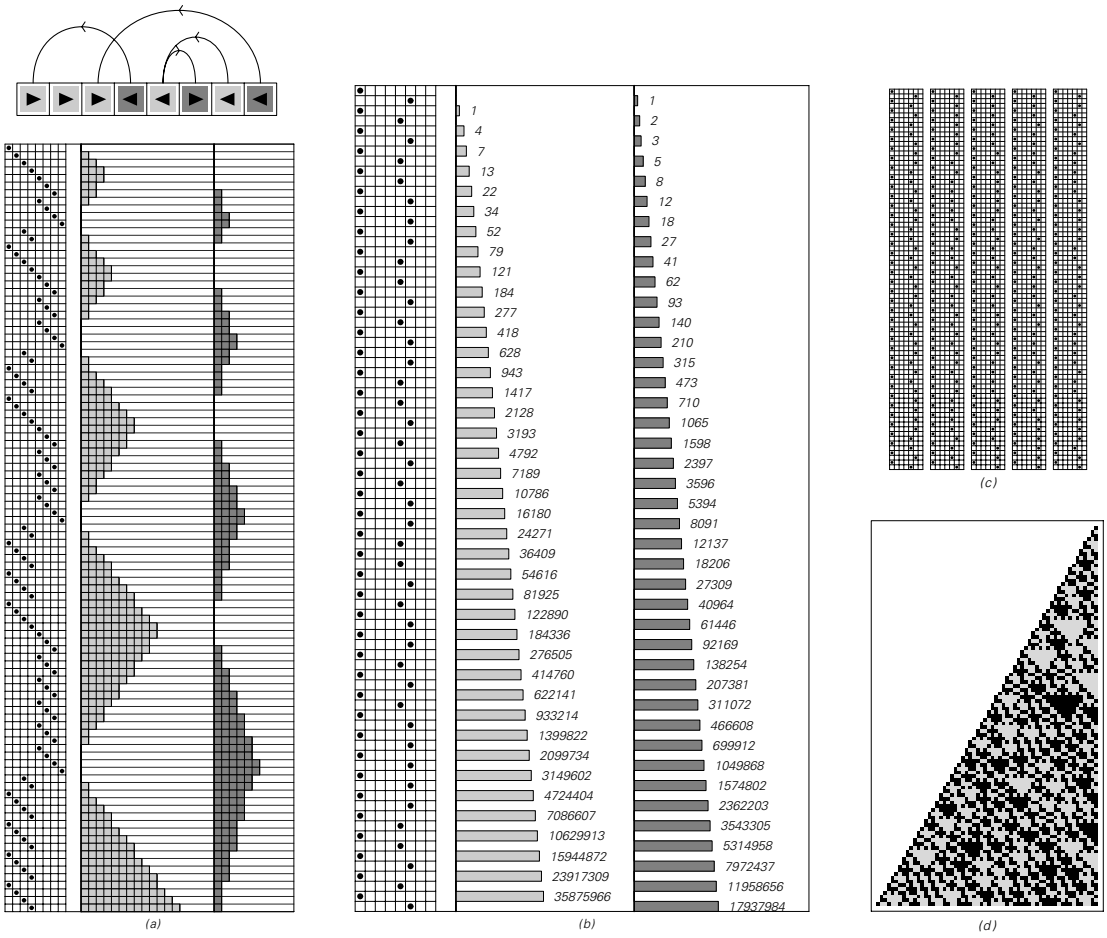
But in all these three examples, the overall behavior is essentially repetitive. And indeed it turns out that among the 10,552 possible register machines with programs that are four or fewer instructions long, not a single one exhibits more complicated behavior.

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.

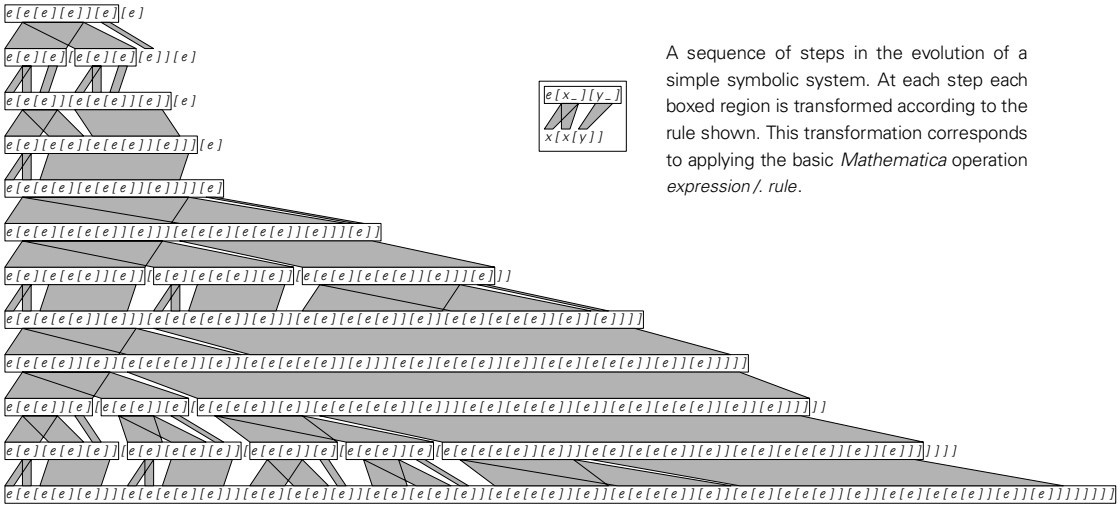
In a typical case, each variable in such a program simply corresponds to one of the registers in the register machine, with no arrays or pointers being allowed. And with this correspondence, our general results on register machines can also be expected to apply to simple programs written in actual low-level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

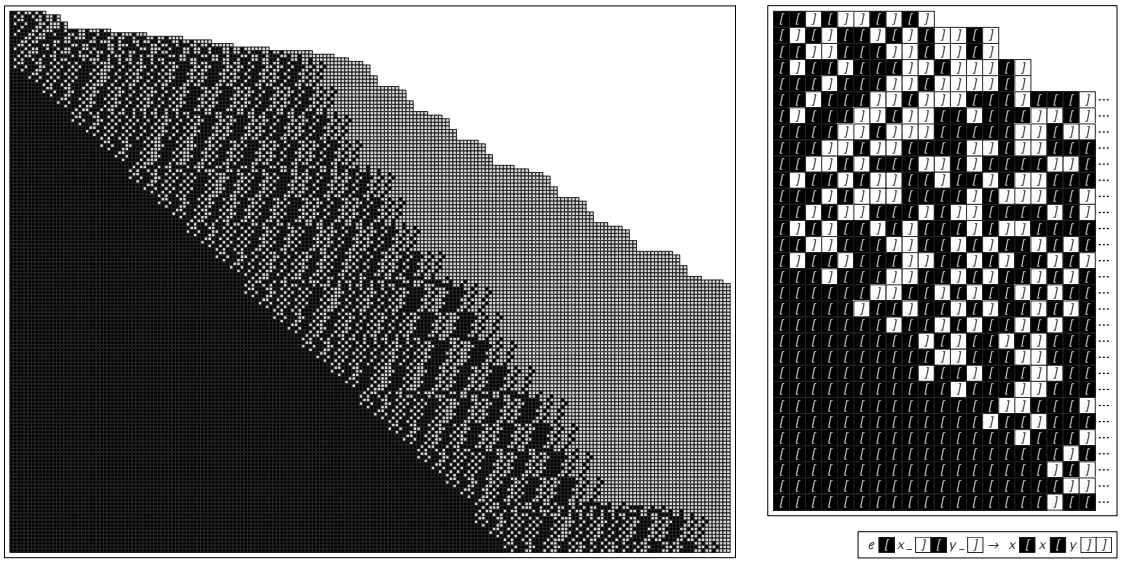
Register machines provide simple idealizations of typical low-level computer languages. But what about *Mathematica*? How can one set up a simple idealization of the transformations on symbolic expressions that *Mathematica* does? One approach suggested by the idea of combinators from the 1920s is to 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 $e[x\_][y\_] \to x[x[y]]$, 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.



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*.

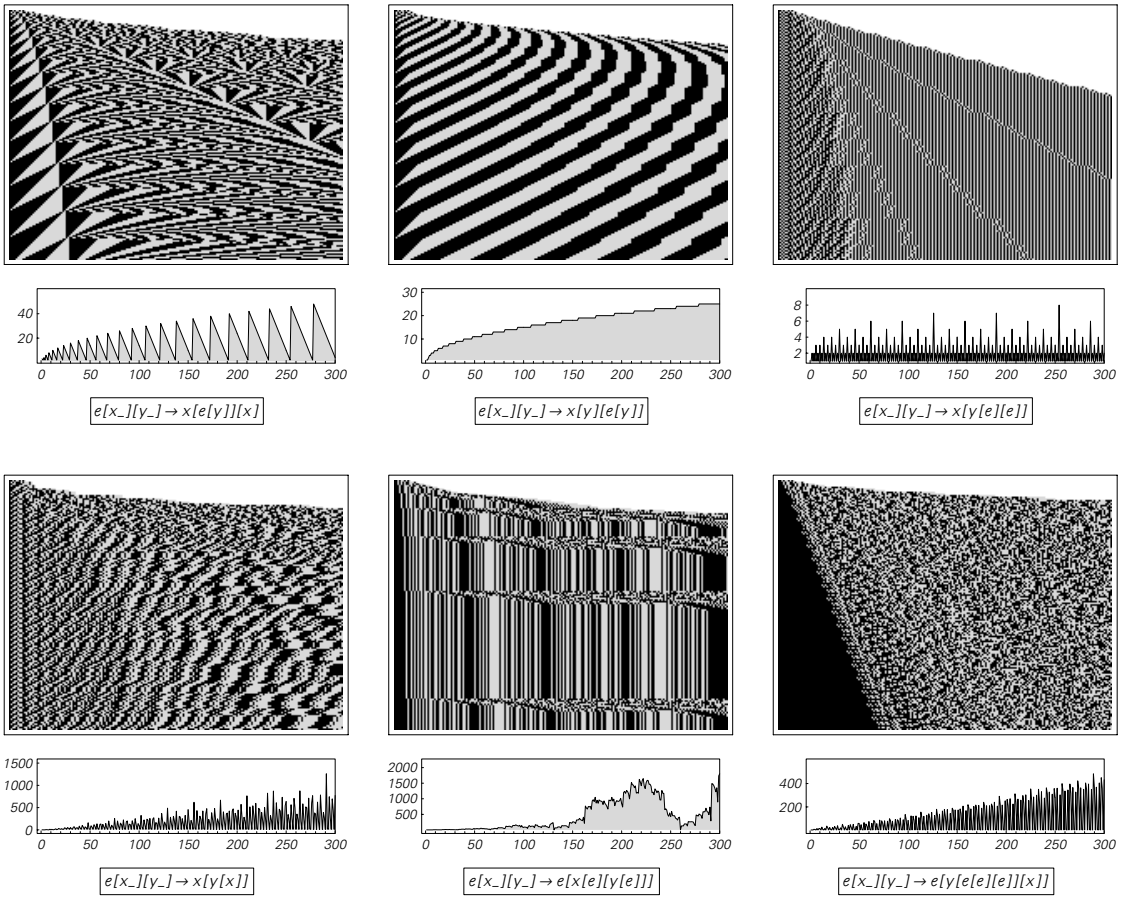
The structure of expressions like those on the facing page is determined just by their sequence of opening and closing brackets. And representing these brackets by dark and light squares respectively, the picture below shows the overall pattern of behavior generated.



More steps in the evolution on the previous page, with opening brackets represented by dark squares and closing brackets by light ones. In each case configurations wider than the picture are cut off on the right. For the initial condition from the previous page, the system evolves after 264 steps to a fixed configuration involving 256 opening brackets followed by 256 closing brackets. For the initial condition on the bottom right, the system again evolves to a fixed configuration, but now this takes 65,555 steps, and the configuration involves 65,536 opening and closing brackets. Note that the evolution rules are highly non-local, and are rather unlike those, say, in a cellular automaton. It turns out that this particular system always evolves to a fixed configuration, but for initial conditions of size n can take roughly n iterated powers of 2 (or $2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}$) to do so.

With the particular rule shown, the behavior always eventually stabilizes—though sometimes only after an astronomically long time.

But it is quite possible to find symbolic systems where this does not happen, as illustrated in the pictures below. Sometimes the behavior that is generated in such systems has a simple repetitive or nested form. But often—just as in so many other kinds of systems—the behavior is instead complex and seemingly quite random.



The behavior of various symbolic systems starting from the initial condition $e[e[e]][e][e]$. The plots at the bottom show the difference in size of the expressions obtained on successive steps.

Some Conclusions

In the chapter before this one, we discovered the remarkable fact that even though their underlying rules are extremely simple, certain cellular automata can nevertheless produce behavior of great complexity.

Yet at first, this seems so surprising and so outside our normal experience that we may tend to assume that it must be a consequence of some rare and special feature of cellular automata, and must not occur in other kinds of systems.

For it is certainly true that cellular automata have many special features. All their elements, for example, are always arranged in a rigid array, and are always updated in parallel at each step. And one might think that features like these could be crucial in making it possible to produce complex behavior from simple underlying rules.

But from our study of substitution systems earlier in this chapter we know, for example, that in fact it is not necessary to have elements that are arranged in a rigid array. And from studying mobile automata, we know that updating in parallel is also not critical.

Indeed, I specifically chose the sequence of systems in this chapter to see what would happen when each of the various special features of cellular automata were taken away. And the remarkable conclusion is that in the end none of these features actually matter much at all. For every single type of system in this chapter has ultimately proved capable of producing very much the same kind of complexity that we saw in cellular automata.

So this suggests that in fact the phenomenon of complexity is quite universal—and quite independent of the details of particular systems.

But when in general does complexity occur?

The examples in this chapter suggest that if the rules for a particular system are sufficiently simple, then the system will only ever exhibit purely repetitive behavior. If the rules are slightly more complicated, then nesting will also often appear. But to get complexity in the overall behavior of a system one needs to go beyond some threshold in the complexity of its underlying rules.

The remarkable discovery that we have made, however, is that this threshold is typically extremely low. And indeed in the course of this chapter we have seen that in every single one of the general kinds of systems that we have discussed, it ultimately takes only very simple rules to produce behavior of great complexity.

One might nevertheless have thought that if one were to increase the complexity of the rules, then the behavior one would get would also become correspondingly more complex. But as the pictures on the facing page illustrate, this is not typically what happens.

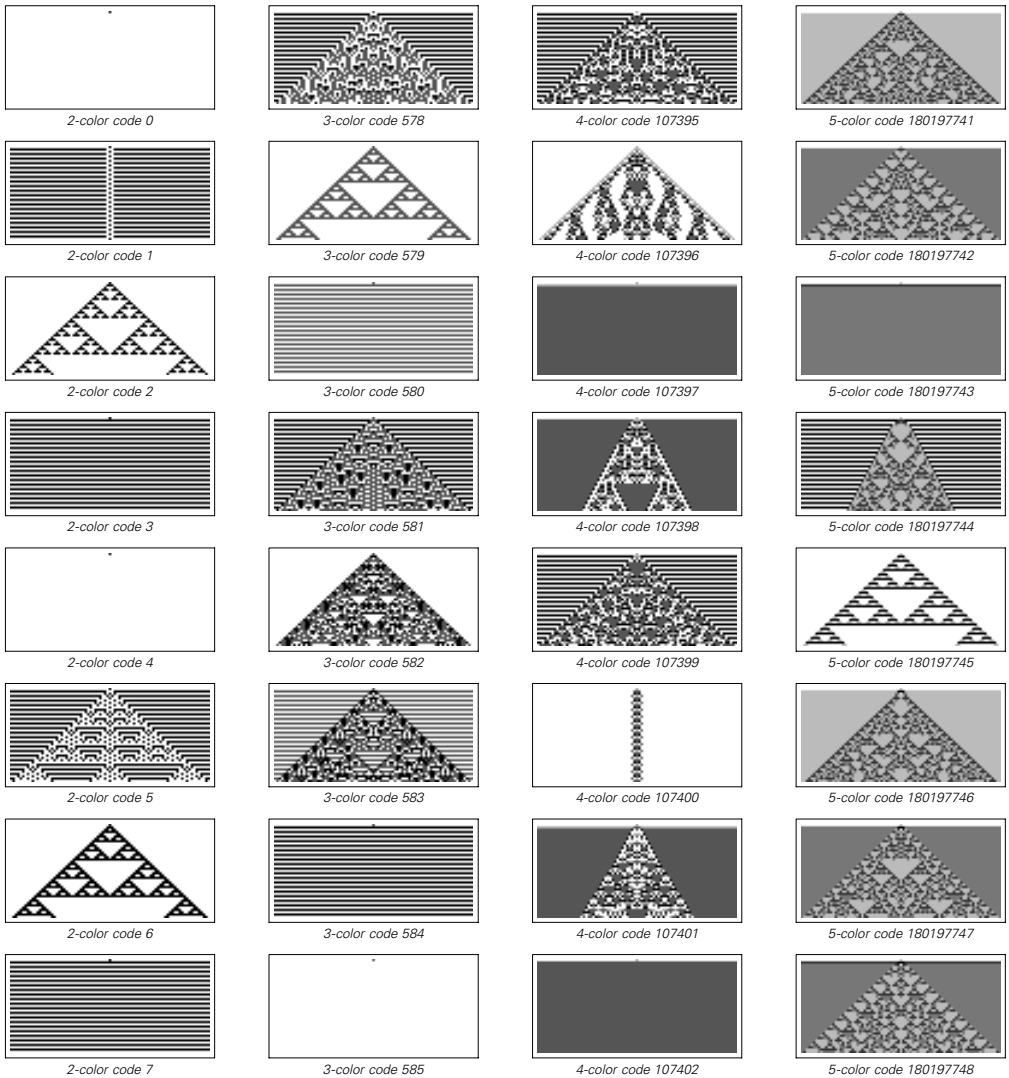
Instead, once the threshold for complex behavior has been reached, what one usually finds is that adding complexity to the underlying rules does not lead to any perceptible increase at all in the overall complexity of the behavior that is produced.

The crucial ingredients that are needed for complex behavior are, it seems, already present in systems with very simple rules, and as a result, nothing fundamentally new typically happens when the rules are made more complex. Indeed, as the picture on the facing page demonstrates, there is often no clear correlation between the complexity of rules and the complexity of behavior they produce. And this means, for example, that even with highly complex rules, very simple behavior still often occurs.

One observation that can be made from the examples in this chapter is that when the behavior of a system does not look complex, it tends to be dominated by either repetition or nesting. And indeed, it seems that the basic themes of repetition, nesting, randomness and localized structures that we already saw in specific cellular automata in the previous chapter are actually very general, and in fact represent the dominant themes in the behavior of a vast range of different systems.

The details of the underlying rules for a specific system can certainly affect the details of the behavior it produces. But what we have seen in this chapter is that at an overall level the typical types of behavior that occur are quite universal, and are almost completely independent of the details of underlying rules.

And this fact has been crucial in my efforts to develop a coherent science of the kind I describe in this book. For it is what implies that



Examples of cellular automata with rules of varying complexity. The rules used are of the so-called totalistic type described on page 60. With two possible colors, just 4 cases need to be specified in such rules, and there are 16 possible rules in all. But as the number of colors increases, the rules rapidly become more complex. With three colors, there are 7 cases to be specified, and 2187 possible rules; with five colors, there are 13 cases to be specified, and 1,220,703,125 possible rules. But even though the underlying rules increase rapidly in complexity, the overall forms of behavior that we see do not change much. With two colors, it turns out that no totalistic rules yield anything other than repetitive or nested behavior. But as soon as three colors are allowed, much more complex behavior is immediately possible. Allowing four or more colors, however, does not further increase the complexity of the behavior, and, as the picture shows, even with five colors, simple repetitive and nested behavior can still occur.

there are general principles that govern the behavior of a wide range of systems, independent of the precise details of each system.

And it is this that means that even if we do not know all the details of what is inside some specific system in nature, we can still potentially make fundamental statements about its overall behavior. Indeed, in most cases, the important features of this behavior will actually turn out to be ones that we have already seen with the various kinds of very simple rules that we have discussed in this chapter.

How the Discoveries in This Chapter Were Made

This chapter—and the last—have described a series of surprising discoveries that I have made about what simple programs typically do. And in making these discoveries I have ended up developing a somewhat new methodology—that I expect will be central to almost any fundamental investigation in the new kind of science that I describe in this book.

Traditional mathematics and the existing theoretical sciences would have suggested using a basic methodology in which one starts from whatever behavior one wants to study, then tries to construct examples that show this behavior. But I am sure that had I used this approach, I would not have got very far. For I would have looked only for types of behavior that I already believed might exist. And in studying cellular automata, this would for example probably have meant that I would only have looked for repetition and nesting.

But what allowed me to discover much more was that I used instead a methodology fundamentally based on doing computer experiments.

In a traditional scientific experiment, one sets up a system in nature and then watches to see how it behaves. And in much the same way, one can set up a program on a computer and then watch how it behaves. And the great advantage of such an experimental approach is that it does not require one to know in advance exactly what kinds of behavior can occur. And this is what makes it possible to discover genuinely new phenomena that one did not expect.

Experience in the traditional experimental sciences might suggest, however, that experiments are somehow always fundamentally imprecise.

For when one deals with systems in nature it is normally impossible to set up or measure them with perfect precision—and indeed it can be a challenge even to make a traditional experiment be at all repeatable.

But for the kinds of computer experiments I do in this book, there is no such issue. For in almost all cases they involve programs whose rules and initial conditions can be specified with perfect precision—so that they work exactly the same whenever and wherever they are run.

In many ways these kinds of computer experiments thus manage to combine the best of both theoretical and experimental approaches to science. For their results have the kind of precision and clarity that one expects of theoretical or mathematical statements. Yet these results can nevertheless be found purely by making observations.

Yet as with all types of experiments it requires considerable skill and judgement to know how to set up a computer experiment that will yield meaningful results. And indeed, over the past twenty years or so my own methodology for doing such experiments has become vastly better.

Over and over again the single most important principle that I have learned is that the best computer experiments are ones that are as simple and straightforward as possible. And this principle applies both to the structure of the actual systems one studies—and to the procedures that one uses for studying them.

At some level the principle of looking at systems with the simplest possible structure can be viewed as an abstract aesthetic one. But it turns out also to have some very concrete consequences.

For a start, the simpler a structure is, the more likely it is that it will show up in a wide diversity of different places. And this means that by studying systems with the simplest possible structure one will tend to get results that have the broadest and most fundamental significance.

In addition, looking at systems with simpler underlying structures gives one a better chance of being able to tell what is really responsible for any phenomenon one sees—for there are fewer features that have been put into the system and that could lead one astray.

At a purely practical level, there is also an advantage to studying systems with simpler structures; for these systems are usually easier to

implement on a computer, and can thus typically be investigated more extensively with given computational resources.

But an obvious issue with saying that one should study systems with the simplest possible structure is that such systems might just not be capable of exhibiting the kinds of behavior that one might consider interesting—or that actually occurs in nature.

And in fact, intuition from traditional science and mathematics has always tended to suggest that unless one adds all sorts of complications, most systems will never be able to exhibit any very relevant behavior. But the results so far in this book have shown that such intuition is far from correct, and that in reality even systems with extremely simple rules can give rise to behavior of great complexity.

The consequences of this fact for computer experiments are quite profound. For it implies that there is never an immediate reason to go beyond studying systems with rather simple underlying rules. But to absorb this point is not an easy matter. And indeed, in my experience the single most common mistake in doing computer experiments is to look at systems that are vastly more complicated than is necessary.

Typically the reason this happens is that one just cannot imagine any way in which a simpler system could exhibit interesting behavior. And so one decides to look at a more complicated system—usually with features specifically inserted to produce some specific form of behavior.

Much later one may go back and look at the simpler system again. And this is often a humbling experience, for it is common to find that the system does in fact manage to produce interesting behavior—but just in a way that one was not imaginative enough to guess.

So having seen this many times I now always try to follow the principle that one can never start with too simple a system. For at worst, one will just establish a lower limit on what is needed for interesting behavior to occur. But much more often, one will instead discover behavior that one never thought was possible.

It should however be emphasized that even in an experiment it is never entirely straightforward to discover phenomena one did not expect. For in setting up the experiment, one inevitably has to make assumptions about the kinds of behavior that can occur. And if it turns

out that there is behavior which does not happen to fit in with these assumptions, then typically the experiment will fail to notice it.

In my experience, however, the way to have the best chance of discovering new phenomena in a computer experiment is to make the design of the experiment as simple and direct as possible. It is usually much better, for example, to do a mindless search of a large number of possible cases than to do a carefully crafted search of a smaller number. For in narrowing the search one inevitably makes assumptions, and these assumptions may end up missing the cases of greatest interest.

Along similar lines, I have always found it much better to look explicitly at the actual behavior of systems, than to work from some kind of summary. For in making a summary one inevitably has to pick out only certain features, and in doing this one can remove or obscure the most interesting effects.

But one of the problems with very direct experiments is that they often generate huge amounts of raw data. Yet what I have typically found is that if one manages to present this data in the form of pictures then it effectively becomes possible to analyze very quickly just with one's eyes. And indeed, in my experience it is typically much easier to recognize unexpected phenomena in this way than by using any kind of automated procedure for data analysis.

It was in a certain sense lucky that one-dimensional cellular automata were the first examples of simple programs that I investigated. For it so happens that in these systems one can usually get a good idea of overall behavior just by looking at an array of perhaps 10,000 cells—which can easily be displayed in few square inches.

And since several of the 256 elementary cellular automaton rules already generate great complexity, just studying a couple of pages of pictures like the ones at the beginning of this chapter should in principle have allowed one to discover the basic phenomenon of complexity in cellular automata.

But in fact I did not make this discovery in such a straightforward way. I had the idea of looking at pictures of cellular automaton evolution at the very beginning. But the technological difficulty of producing these pictures made me want to reduce their number as

much as possible. And so at first I looked only at the 32 rules which had left-right symmetry and made blank backgrounds stay unchanged.

Among these rules I found examples of repetition and nesting. And with random initial conditions, I found more complicated behavior. But since I did not expect that any complicated behavior would be possible with simple initial conditions, I did not try looking at other rules in an attempt to find it. Nevertheless, as it happens, the first paper that I published about cellular automata—in 1983—did in fact include a picture of rule 30 from page 27, as an example of a non-symmetric rule. But the picture showed only 20 steps of evolution, and at the time I did not look carefully at it, and certainly did not appreciate its significance.

For several years, I did progressively more sophisticated computer experiments on cellular automata, and in the process I managed to elucidate many of their properties. But finally, when technology had advanced to the point where it became almost trivial for me to do so, I went back and generated some straightforward pages of pictures of all 256 elementary rules evolving from simple initial conditions. And it was upon seeing these pictures that I finally began to appreciate the remarkable phenomenon that occurs in systems like rule 30.

Seven years later, after I had absorbed some basic intuition from looking at cellular automata like rule 30, I resolved to find out whether similar phenomena also occurred in other kinds of systems. And the first such systems that I investigated were mobile automata.

Mobile automata in a sense evolve very slowly relative to cellular automata, so to make more efficient pictures I came up with a scheme for showing their evolution in compressed form. I then started off by generating pictures of the first hundred, then the first thousand, then the first ten thousand, mobile automata. But in all of these pictures I found nothing beyond repetitive and nested behavior.

Yet being convinced that more complicated behavior must be possible, I decided to persist, and so I wrote a program that would automatically search through large numbers of mobile automata. I set up various criteria for the search, based on how I expected mobile automata could behave. And quite soon, I had made the program search a million mobile automata, then ten million.

But still I found nothing.

So then I went back and started looking by eye at mobile automata with large numbers of randomly chosen rules. And after some time what I realized was that with the compression scheme I was using there could be mobile automata that would be discarded according to my search criteria, but which nevertheless still had interesting behavior. And within an hour of modifying my search program to account for this, I found the example shown on page 74.

Yet even after this, there were still many assumptions implicit in my search program. And it took some time longer to identify and remove them. But having done so, it was then rather straightforward to find the example shown on page 75.

A somewhat similar pattern has been repeated for most of the other systems described in this chapter. The main challenge was always to avoid assumptions and set up experiments that were simple and direct enough that they did not miss important new phenomena.

In many cases it took a large number of iterations to work out the right experiments to do. And had it not been for the ease with which I could set up new experiments using *Mathematica*, it is likely that I would never have gotten very far in investigating most of the systems discussed in this chapter. But in the end, after running programs for a total of several years of computer time—corresponding to more than a million billion logical operations—and creating the equivalent of tens of thousands of pages of pictures, I was finally able to find all of the various examples shown in this chapter and the ones that follow.

The World of Simple Programs

More Cellular Automata

▪ **Page 53 · Numbering scheme.** I introduced the numbering scheme used here in the 1983 paper where I first discussed one-dimensional cellular automata (see page 881). I termed two-color nearest-neighbor cellular automata "elementary" to reflect the idea that their rules are as simple as possible.

▪ **Page 55 · Rule equivalences.** The table below gives basic equivalences between elementary cellular automaton rules. In each block the second entry is the rule obtained by interchanging black and white, the third entry is the rule obtained by interchanging left and right, and the fourth entry the rule obtained by applying both operations. (The smallest rule number is given in boldface.) For a rule with number *n* the two operations correspond respectively to computing *1 – Reverse[list]* and *list[[{1, 5, 3, 7, 2, 6, 4, 8}]]* with *list = IntegerDigits[n, 2, 8]*.

▪ **Special rules.** Rule 51: complement; rule 170: left shift; rule 204: identity; rule 240: right shift. These rules only ever depend on one cell in each neighborhood.

▪ **Rule expressions.** The table on the next page gives Boolean expressions for each of the elementary rules. The expressions

0	255	**0**	255	**32**	251	**32**	251	64	253	**8**	239	96	249	**40**	235	**128**	254	**128**	254	**160**	250	**160**	250	192	252	**136**	238	224	248	**168**	234
1	127	**1**	127	**33**	123	**33**	123	65	125	**9**	111	97	121	**41**	107	129	**126**	129	**126**	161	**122**	161	**122**	193	124	137	**110**	225	120	169	**106**
2	191	16	247	**34**	187	48	243	66	189	**24**	231	98	185	**56**	227	**130**	190	144	246	**162**	186	176	242	194	188	**152**	230	226	**184**	**184**	226
3	63	17	119	**35**	59	49	115	67	61	**25**	103	99	**57**	**57**	99	131	**62**	145	118	163	**58**	177	114	195	**60**	153	102	227	**56**	185	98
4	223	**4**	223	**36**	219	**36**	219	68	221	**12**	207	100	217	**44**	203	**132**	222	**132**	222	**164**	218	**164**	218	196	220	**140**	206	228	216	**172**	202
5	95	**5**	95	**37**	91	**37**	91	69	93	**13**	79	101	89	**45**	75	133	**94**	133	**94**	165	**90**	165	**90**	197	92	141	**78**	229	88	173	**74**
6	159	20	215	**38**	155	52	211	70	157	**28**	199	102	153	**60**	195	**134**	158	148	214	166	**154**	180	210	198	**156**	**156**	198	230	**152**	188	194
7	31	21	87	39	**27**	53	83	71	**29**	**29**	71	103	**25**	61	67	135	**30**	149	86	167	**26**	181	82	199	**28**	157	70	231	**24**	189	66
8	239	64	253	**40**	235	96	249	**72**	237	**72**	237	**104**	233	**104**	233	**136**	238	192	252	**168**	234	224	248	**200**	236	**200**	236	**232**	**232**	**232**	**232**
9	111	65	125	**41**	107	97	121	**73**	109	**73**	109	**105**	**105**	**105**	**105**	137	**110**	193	124	169	**106**	225	120	201	**108**	201	**108**	233	**104**	233	**104**
10	175	80	245	**42**	171	112	241	**74**	173	88	229	**106**	169	120	225	**138**	174	208	244	**170**	**170**	240	240	202	**172**	216	228	234	**168**	248	224
11	47	81	117	**43**	**43**	113	113	75	**45**	89	101	107	**41**	121	97	139	**46**	209	116	171	**42**	241	112	203	**44**	217	100	235	**40**	249	96
12	207	68	221	**44**	203	100	217	**76**	205	**76**	205	**108**	201	**108**	201	**140**	206	196	220	**172**	202	228	216	**204**	**204**	**204**	**204**	236	**200**	236	**200**
13	79	69	93	**45**	75	101	89	**77**	**77**	**77**	**77**	109	**73**	109	**73**	141	**78**	197	92	173	**74**	229	88	205	**76**	205	**76**	237	**72**	237	**72**
14	143	84	213	**46**	139	116	209	**78**	141	92	197	**110**	137	124	193	**142**	**142**	212	212	174	**138**	244	208	206	**140**	220	196	238	**136**	252	192
15	**15**	85	85	47	**11**	117	81	79	**13**	93	69	111	**9**	125	65	143	**14**	213	84	175	**10**	245	80	207	**12**	221	68	239	**8**	253	64
16	247	**2**	191	48	243	**34**	187	80	245	**10**	175	112	241	**42**	171	144	246	**130**	190	176	242	**162**	186	208	244	**138**	174	240	240	**170**	**170**
17	119	**3**	63	49	115	**35**	59	81	117	**11**	47	113	113	**43**	**43**	145	118	131	**62**	177	114	163	**58**	209	116	139	**46**	241	112	171	**42**
18	183	**18**	183	**50**	179	**50**	179	82	181	**26**	167	114	177	**58**	163	**146**	182	**146**	182	**178**	**178**	**178**	**178**	210	180	**154**	166	242	176	186	**162**
19	55	**19**	55	**51**	**51**	**51**	**51**	83	53	**27**	39	115	49	59	**35**	147	**54**	147	**54**	179	**50**	179	**50**	211	52	155	**38**	243	48	187	**34**
20	215	**6**	159	52	211	**38**	155	84	213	**14**	143	116	209	**46**	139	148	214	**134**	158	180	210	166	**154**	212	212	**142**	**142**	244	208	174	**138**
21	87	**7**	31	53	83	39	**27**	85	85	**15**	**15**	117	81	47	**11**	149	86	135	**30**	181	82	167	**26**	213	84	143	**14**	245	80	175	**10**
22	151	**22**	151	**54**	147	**54**	147	86	149	**30**	135	118	145	**62**	131	**150**	**150**	**150**	**150**	182	**146**	182	**146**	214	148	158	**134**	246	144	190	**130**
23	**23**	**23**	**23**	55	**19**	55	**19**	87	21	31	**7**	119	17	63	**3**	151	**22**	151	**22**	183	**18**	183	**18**	215	20	159	**6**	247	16	191	**2**
24	231	66	189	**56**	227	98	185	88	229	**74**	173	120	225	**106**	169	**152**	230	194	188	**184**	226	226	**184**	216	228	202	**172**	248	224	234	**168**
25	103	67	61	**57**	99	99	**57**	89	101	75	**45**	121	97	107	**41**	153	102	195	**60**	185	98	227	**56**	217	100	203	**44**	249	96	235	**40**
26	167	82	181	**58**	163	114	177	**90**	165	**90**	165	**122**	161	**122**	161	**154**	166	210	180	186	**162**	242	176	218	**164**	218	**164**	250	**160**	250	**160**
27	39	83	53	59	**35**	115	49	91	**37**	91	**37**	123	**33**	123	**33**	155	**38**	211	52	187	**34**	243	48	219	**36**	219	**36**	251	**32**	251	**32**
28	199	70	157	**60**	195	102	153	92	197	**78**	141	124	193	**110**	137	**156**	198	198	**156**	188	194	230	**152**	220	196	206	**140**	252	192	238	**136**
29	71	71	**29**	61	67	103	**25**	93	69	79	**13**	125	65	111	**9**	157	70	199	**28**	189	66	231	**24**	221	68	207	**12**	253	64	239	**8**
30	135	86	149	**62**	131	118	145	**94**	133	**94**	133	**126**	129	**126**	129	158	**134**	214	148	190	**130**	246	144	222	**132**	222	**132**	254	**128**	254	**128**
31	**7**	87	21	63	**3**	119	17	95	**5**	95	**5**	127	**1**	127	**1**	159	**6**	215	20	191	**2**	247	16	223	**4**	223	**4**	255	**0**	255	**0**

rule 0 : 0
rule 1 : $\neg(p \vee q \vee r)$
rule 2 : $(\neg p) \wedge (\neg q) \wedge r$
rule 3 : $\neg(p \vee q)$
rule 4 : $(\neg(p \vee r)) \wedge q$
rule 5 : $\neg(p \vee r)$
rule 6 : $(\neg p) \wedge (q \veebar r)$
rule 7 : $\neg(p \vee (q \wedge r))$
rule 8 : $(\neg p) \wedge q \wedge r$
rule 9 : $\neg(p \vee (q \veebar r))$
rule 10 : $(\neg p) \wedge r$
rule 11 : $p \veebar (p \vee (\neg q) \vee r)$
rule 12 : $(p \wedge q) \veebar q$
rule 13 : $p \veebar (p \vee q \vee (\neg r))$
rule 14 : $p \veebar (p \vee q \vee r)$
rule 15 : $\neg p$
rule 16 : $p \wedge (\neg q) \wedge (\neg r)$
rule 17 : $\neg(q \vee r)$
rule 18 : $(p \veebar q \veebar r) \wedge (\neg q)$
rule 19 : $\neg((p \wedge r) \vee q)$
rule 20 : $(p \veebar q) \wedge (\neg r)$
rule 21 : $\neg((p \wedge q) \vee r)$
rule 22 : $p \veebar (p \wedge q \wedge r) \veebar q \veebar r$
rule 23 : $p \veebar ((p \veebar (\neg q)) \vee (q \veebar r))$
rule 24 : $(p \veebar q) \wedge (p \veebar r)$
rule 25 : $(p \wedge q \wedge r) \veebar q \veebar (\neg r)$
rule 26 : $p \veebar ((p \wedge q) \vee r)$
rule 27 : $p \veebar ((p \veebar (\neg q)) \vee r)$
rule 28 : $p \veebar ((p \wedge r) \vee q)$
rule 29 : $p \veebar ((p \veebar (\neg r)) \vee q)$
rule 30 : $p \veebar (q \vee r)$
rule 31 : $\neg(p \wedge (q \vee r))$
rule 32 : $p \wedge (\neg q) \wedge r$
rule 33 : $\neg((p \veebar q \veebar r) \vee q)$
rule 34 : $(\neg q) \wedge r$
rule 35 : $((\neg p) \vee q \vee r) \veebar q$
rule 36 : $(p \veebar q) \wedge (q \veebar r)$
rule 37 : $p \veebar (p \wedge q \wedge r) \veebar (\neg r)$
rule 38 : $((p \wedge q) \vee r) \veebar q$
rule 39 : $((p \veebar (\neg q)) \vee r) \veebar q$
rule 40 : $(p \veebar q) \wedge r$
rule 41 : $\neg((p \wedge q) \vee (p \veebar q \veebar r))$
rule 42 : $(p \wedge q \wedge r) \veebar r$
rule 43 : $p \veebar ((p \veebar r) \vee (p \veebar (\neg q)))$
rule 44 : $(p \wedge (q \vee r)) \veebar q$
rule 45 : $p \veebar (q \vee (\neg r))$
rule 46 : $(p \wedge q) \veebar (q \vee r)$
rule 47 : $(\neg p) \vee ((\neg q) \wedge r)$
rule 48 : $p \wedge (\neg q)$
rule 49 : $(p \vee q \vee (\neg r)) \veebar q$
rule 50 : $(p \vee q \vee r) \veebar q$
rule 51 : $\neg q$
rule 52 : $(p \vee (q \wedge r)) \veebar q$
rule 53 : $(p \vee (q \veebar (\neg r))) \veebar q$
rule 54 : $(p \vee r) \veebar q$
rule 55 : $\neg((p \vee r) \wedge q)$
rule 56 : $p \veebar ((p \vee r) \wedge q)$
rule 57 : $(p \vee (\neg r)) \veebar q$
rule 58 : $(p \vee (q \veebar r)) \veebar q$
rule 59 : $((\neg p) \wedge r) \vee (\neg q)$
rule 60 : $p \veebar q$
rule 61 : $p \veebar (p \vee q \vee r) \veebar (\neg q)$
rule 62 : $(p \wedge q) \veebar (p \vee q \vee r)$
rule 63 : $\neg(p \wedge q)$
rule 64 : $p \wedge q \wedge (\neg r)$
rule 65 : $\neg((p \veebar q) \vee r)$
rule 66 : $(p \veebar r) \wedge (q \veebar r)$
rule 67 : $p \veebar (p \wedge q \wedge r) \veebar (\neg q)$
rule 68 : $q \wedge (\neg r)$
rule 69 : $((\neg p) \vee q \vee r) \veebar r$
rule 70 : $((p \wedge r) \vee q) \veebar r$
rule 71 : $((p \veebar (\neg r)) \vee q) \veebar r$
rule 72 : $(p \wedge q) \veebar (q \wedge r)$
rule 73 : $\neg((p \wedge r) \vee (p \veebar q \veebar r))$
rule 74 : $(p \wedge (q \vee r)) \veebar r$
rule 75 : $p \veebar ((\neg q) \vee r)$
rule 76 : $(p \wedge q \wedge r) \veebar q$
rule 77 : $p \veebar ((p \veebar q) \vee (p \veebar (\neg r)))$
rule 78 : $p \veebar ((p \veebar q) \vee r)$
rule 79 : $(\neg p) \vee (q \wedge (\neg r))$
rule 80 : $p \wedge (\neg r)$
rule 81 : $(p \vee (\neg q) \vee r) \veebar r$
rule 82 : $(p \vee (q \wedge r)) \veebar r$
rule 83 : $(p \vee (q \veebar (\neg r))) \veebar r$
rule 84 : $(p \vee q \vee r) \veebar r$
rule 85 : $\neg r$
rule 86 : $(p \vee q) \veebar r$
rule 87 : $\neg((p \vee q) \wedge r)$
rule 88 : $p \veebar ((p \vee q) \wedge r)$
rule 89 : $(p \vee (\neg q)) \veebar r$
rule 90 : $p \veebar r$
rule 91 : $p \veebar (\neg(p \vee q \vee r)) \veebar r$
rule 92 : $(p \vee (q \veebar r)) \veebar r$
rule 93 : $\neg((p \vee (\neg q)) \wedge r)$
rule 94 : $(p \wedge r) \veebar (p \vee q \vee r)$
rule 95 : $\neg(p \wedge r)$
rule 96 : $p \wedge (q \veebar r)$
rule 97 : $\neg((p \veebar q \veebar r) \vee (q \wedge r))$
rule 98 : $((p \vee r) \wedge q) \veebar r$
rule 99 : $((\neg p) \vee r) \veebar q$
rule 100 : $((p \vee q) \wedge r) \veebar q$
rule 101 : $p \veebar (p \wedge q) \veebar (\neg r)$
rule 102 : $q \veebar r$
rule 103 : $(\neg(p \vee q \vee r)) \veebar q \veebar r$
rule 104 : $p \veebar (p \vee q \vee r) \veebar q \veebar r$
rule 105 : $p \veebar q \veebar (\neg r)$
rule 106 : $(p \wedge q) \veebar r$
rule 107 : $p \veebar (p \vee q \vee (\neg r)) \veebar q \veebar r$
rule 108 : $(p \wedge r) \veebar q$
rule 109 : $p \veebar (p \vee (\neg q) \vee r) \veebar q \veebar r$
rule 110 : $((\neg p) \wedge q \wedge r) \veebar q \veebar r$
rule 111 : $(\neg p) \vee (q \veebar r)$
rule 112 : $p \veebar (p \wedge q \wedge r)$
rule 113 : $p \veebar (\neg((p \veebar q) \vee (p \veebar r)))$
rule 114 : $((p \veebar q) \vee r) \veebar q$
rule 115 : $(p \wedge (\neg r)) \vee (\neg q)$
rule 116 : $(p \vee q) \veebar (q \wedge r)$
rule 117 : $(p \wedge (\neg q)) \vee (\neg r)$
rule 118 : $(p \vee q \vee r) \veebar (q \wedge r)$
rule 119 : $\neg(q \wedge r)$
rule 120 : $p \veebar (q \wedge r)$
rule 121 : $p \veebar ((\neg p) \vee q \vee r) \veebar q \veebar r$
rule 122 : $p \veebar (p \wedge (\neg q) \wedge r) \veebar r$
rule 123 : $\neg((p \veebar q \veebar r) \wedge q)$
rule 124 : $p \veebar (p \wedge q \wedge (\neg r)) \veebar q$
rule 125 : $(p \veebar q) \vee (\neg r)$
rule 126 : $(p \veebar q) \vee (p \veebar r)$
rule 127 : $\neg(p \wedge q \wedge r)$
rule 128 : $p \wedge q \wedge r$
rule 129 : $\neg((p \veebar q) \vee (p \veebar r))$
rule 130 : $(p \veebar q \veebar r) \wedge r$
rule 131 : $p \veebar (p \wedge q \wedge (\neg r)) \veebar (\neg q)$
rule 132 : $(p \veebar q \veebar r) \wedge q$
rule 133 : $p \veebar (p \wedge (\neg q) \wedge r) \veebar (\neg r)$
rule 134 : $(p \wedge (q \vee r)) \veebar q \veebar r$
rule 135 : $(\neg p) \veebar (q \wedge r)$
rule 136 : $q \wedge r$
rule 137 : $((\neg p) \vee q \vee r) \veebar q \veebar r$
rule 138 : $(p \wedge (\neg q) \wedge r) \veebar r$
rule 139 : $\neg((p \vee q) \veebar (q \wedge r))$
rule 140 : $((\neg p) \vee r) \wedge q$
rule 141 : $p \veebar ((p \veebar q) \vee (\neg r))$
rule 142 : $p \veebar ((p \veebar q) \vee (p \veebar r))$
rule 143 : $(\neg p) \vee (q \wedge r)$
rule 144 : $p \wedge (p \veebar q \veebar r)$
rule 145 : $((\neg p) \wedge q \wedge r) \veebar q \veebar (\neg r)$
rule 146 : $p \veebar ((p \vee r) \wedge q) \veebar r$
rule 147 : $(p \wedge r) \veebar (\neg q)$
rule 148 : $p \veebar ((p \vee q) \wedge r) \veebar q$
rule 149 : $(p \wedge q) \veebar (\neg r)$
rule 150 : $p \veebar q \veebar r$
rule 151 : $p \veebar (\neg(p \vee q \vee r)) \veebar q \veebar r$
rule 152 : $(p \vee q \vee r) \veebar q \veebar r$
rule 153 : $q \veebar (\neg r)$
rule 154 : $p \veebar (p \wedge q) \veebar r$
rule 155 : $(p \vee q \vee (\neg r)) \veebar q \veebar r$
rule 156 : $p \veebar (p \wedge r) \veebar q$
rule 157 : $(p \vee (\neg q) \vee r) \veebar q \veebar r$
rule 158 : $(p \veebar q \veebar r) \vee (q \wedge r)$
rule 159 : $\neg(p \wedge (q \veebar r))$
rule 160 : $p \wedge r$
rule 161 : $p \veebar (p \vee (\neg q) \vee r) \veebar r$
rule 162 : $(p \vee (\neg q)) \wedge r$
rule 163 : $((\neg p) \vee (q \veebar r)) \veebar q$
rule 164 : $p \veebar (p \vee q \vee r) \veebar r$
rule 165 : $p \veebar (\neg r)$
rule 166 : $(p \wedge q) \veebar q \veebar r$
rule 167 : $p \veebar (p \vee q \vee (\neg r)) \veebar r$
rule 168 : $(p \vee q) \wedge r$
rule 169 : $(\neg(p \vee q)) \veebar r$
rule 170 : r
rule 171 : $(\neg(p \vee q)) \vee r$
rule 172 : $(p \wedge (q \veebar r)) \veebar q$
rule 173 : $(p \veebar (\neg r)) \vee (q \wedge r)$
rule 174 : $((p \wedge q) \veebar q) \vee r$
rule 175 : $(\neg p) \vee r$
rule 176 : $p \wedge ((\neg q) \vee r)$
rule 177 : $p \veebar (\neg((p \veebar q) \vee r))$
rule 178 : $((p \veebar q) \vee (p \veebar r)) \veebar q$
rule 179 : $(p \wedge r) \vee (\neg q)$
rule 180 : $p \veebar q \veebar (q \wedge r)$
rule 181 : $p \veebar ((\neg p) \vee q \vee r) \veebar r$
rule 182 : $(p \wedge r) \vee (p \veebar q \veebar r)$
rule 183 : $(p \veebar q \veebar r) \vee (\neg q)$
rule 184 : $p \veebar (p \wedge q) \veebar (q \wedge r)$
rule 185 : $(p \wedge r) \vee (q \veebar (\neg r))$
rule 186 : $(p \wedge (\neg q)) \vee r$
rule 187 : $(\neg q) \vee r$
rule 188 : $p \veebar (p \wedge q \wedge r) \veebar q$
rule 189 : $(p \veebar q) \vee (p \veebar (\neg r))$
rule 190 : $(p \veebar q) \vee r$
rule 191 : $(\neg p) \vee (\neg q) \vee r$
rule 192 : $p \wedge q$
rule 193 : $p \veebar (p \vee q \vee (\neg r)) \veebar q$
rule 194 : $p \veebar (p \vee q \vee r) \veebar q$
rule 195 : $p \veebar (\neg q)$
rule 196 : $(p \vee (\neg r)) \wedge q$
rule 197 : $(\neg(p \vee (q \veebar r))) \veebar q$
rule 198 : $(p \wedge r) \veebar q \veebar r$
rule 199 : $p \veebar (p \vee (\neg q) \vee r) \veebar q$
rule 200 : $(p \vee r) \wedge q$
rule 201 : $(\neg(p \vee r)) \veebar q$
rule 202 : $(p \wedge (q \veebar r)) \veebar r$
rule 203 : $(p \veebar (\neg q)) \vee (q \wedge r)$
rule 204 : q
rule 205 : $(\neg(p \vee r)) \vee q$
rule 206 : $((\neg p) \wedge r) \vee q$
rule 207 : $\neg(p \wedge (\neg q))$
rule 208 : $p \wedge (q \vee (\neg r))$
rule 209 : $\neg((p \wedge q) \veebar (q \vee r))$
rule 210 : $p \veebar (q \wedge r) \veebar r$
rule 211 : $p \veebar ((\neg p) \vee q \vee r) \veebar q$
rule 212 : $((p \veebar q) \vee (p \veebar r)) \veebar r$
rule 213 : $(p \wedge q) \vee (\neg r)$
rule 214 : $(p \wedge q) \vee (p \veebar q \veebar r)$
rule 215 : $\neg((p \veebar q) \wedge r)$
rule 216 : $p \veebar ((p \veebar q) \wedge r)$
rule 217 : $(p \wedge q) \vee (q \veebar (\neg r))$
rule 218 : $p \veebar (p \wedge q \wedge r) \veebar r$
rule 219 : $(p \veebar r) \vee (p \veebar (\neg q))$
rule 220 : $(p \wedge (\neg r)) \vee q$
rule 221 : $q \vee (\neg r)$
rule 222 : $(p \veebar q \veebar r) \vee q$
rule 223 : $\neg(p \wedge (\neg q) \wedge r)$
rule 224 : $p \wedge (q \vee r)$
rule 225 : $p \veebar (\neg(q \vee r))$
rule 226 : $(p \wedge q) \veebar (q \wedge r) \veebar r$
rule 227 : $(p \wedge r) \vee (p \veebar (\neg q))$
rule 228 : $((p \veebar q) \wedge r) \veebar q$
rule 229 : $(p \wedge q) \vee (p \veebar (\neg r))$
rule 230 : $(p \wedge q \wedge r) \veebar q \veebar r$
rule 231 : $(p \veebar (\neg q)) \vee (q \veebar r)$
rule 232 : $(p \wedge q) \vee ((p \vee q) \wedge r)$
rule 233 : $p \veebar (p \wedge q \wedge r) \veebar q \veebar (\neg r)$
rule 234 : $(p \wedge q) \vee r$
rule 235 : $(p \veebar (\neg q)) \vee r$
rule 236 : $(p \wedge r) \vee q$
rule 237 : $(p \veebar (\neg r)) \vee q$
rule 238 : $q \vee r$
rule 239 : $(\neg p) \vee q \vee r$
rule 240 : p
rule 241 : $p \vee (\neg(q \vee r))$
rule 242 : $p \vee ((\neg q) \wedge r)$
rule 243 : $p \vee (\neg q)$
rule 244 : $p \vee (q \wedge (\neg r))$
rule 245 : $p \vee (\neg r)$
rule 246 : $p \vee (q \veebar r)$
rule 247 : $p \vee (\neg q) \vee (\neg r)$
rule 248 : $p \vee (q \wedge r)$
rule 249 : $p \vee (q \veebar (\neg r))$
rule 250 : $p \vee r$
rule 251 : $p \vee (\neg q) \vee r$
rule 252 : $p \vee q$
rule 253 : $p \vee q \vee (\neg r)$
rule 254 : $p \vee q \vee r$
rule 255 : 1

use the minimum possible number of operators; when there are several equivalent forms, I give the most uniform and symmetrical one. Note that ⊻ stands for *Xor*.

■ **Rule orderings.** The fact that successive rules often show very different behavior does not appear to be affected by using alternative orderings such as Gray code (see page 901.)

■ **Page 58 · Algebraic forms.** The rules here can be expressed in algebraic terms (see page 869) as follows:

- Rule 22: *Mod[p + q + r + p q r, 2]*
- Rule 60: *Mod[p + q, 2]*
- Rule 105: *Mod[1 + p + q + r, 2]*
- Rule 129: *Mod[1 + p + q + r + p q + q r + p r, 2]*
- Rule 150: *Mod[p + q + r, 2]*
- Rule 225: *Mod[1 + p + q + r + q r, 2]*

Note that rules 60, 105 and 150 are additive, like rule 90.

■ **Rule 150.** This rule can be viewed as an analog of rule 90 in which the values of three cells, rather than two, are added modulo 2. Corresponding to the result on page 870 for rule 90, the number of black cells at row t in the pattern from rule 150 is given by

```
Apply[Times, Map[(2^#+2 - (-1)^#+2)/3 &,
    Cases[Split[IntegerDigits[t, 2]], k : {(1)..} :> Length[k]]]]
```

There are a total of 2^m *Fibonacci[m + 2]* black cells in the pattern obtained up to step 2^m, implying fractal dimension $Log[2, 1 + \sqrt{5}]$. (See also page 956.)

The value at step t in the column immediately adjacent to the center is the nested sequence discussed on page 892 and given by *Mod[IntegerExponent[t, 2], 2]*. The cell at position n on row t turns out to be given by *Mod[GegenbauerC[n, -t, -1/2], 2]*, as discussed on page 612.

■ **Rule 225.** The width of the pattern after t steps varies between $Sqrt[3/2]\sqrt{t}$ (achieved when $t = 3 \times 2^{2n+1}$) and $Sqrt[9/2]\sqrt{t}$ (achieved when $t = 2^{2n+1}$). The pattern scales differently in the horizontal and vertical direction, corresponding to fractal dimensions *Log[2, 5]* and *Log[4, 5]* respectively. Note that with more complicated initial conditions rule 225 often no longer yields a regular nested pattern, as shown on page 951. The resulting patterns typically grow at a roughly constant average rate.

■ **Rule 22.** With more complicated initial conditions the pattern is often no longer nested, as shown on page 263.

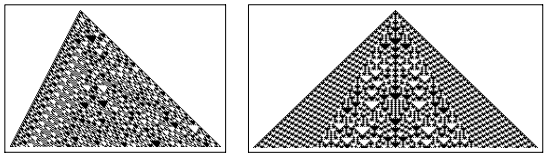
■ **Page 59 · Algebraic forms.** The rules here can be expressed in algebraic terms (see page 869) as follows:

- Rule 30: *Mod[p + q + r + q r, 2]*
- Rule 45: *Mod[1 + p + r + q r, 2]*
- Rule 73: *Mod[1 + p + q + r + p r + p q r, 2]*

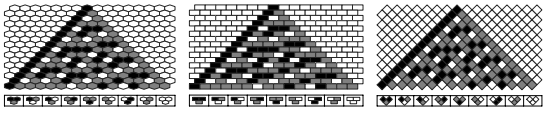
■ **Rule 45.** The center column of the pattern appears for practical purposes random, just as in rule 30. The left edge of the pattern moves 1 cell every 2 steps; the boundary between repetition and randomness moves on average 0.17 cells per step.

■ **Rule 73.** The pattern has a few definite regularities. The center column of cells is repetitive, alternating between black and white on successive steps. And in all cases black cells appear only in blocks that are an odd number of cells wide. (Any block in rule 73 consisting of an even number of black cells will evolve to a structure that remains fixed forever, as mentioned on page 954.) The more complicated central region of the pattern grows 4 cells every 7 steps; the outer region consists of blocks that are 12 cells wide and repeat every 3 steps.

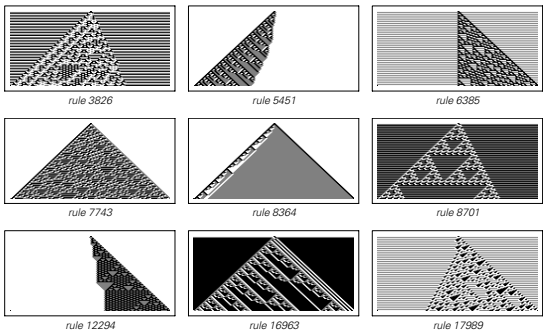
■ **Alternating colors.** The pictures below show rules 45 and 73 with the colors of cells on alternate steps reversed.



■ **Two-cell neighborhoods.** By having cells on successive steps be arranged like hexagons or staggered bricks, as in the pictures below, one can set up cellular automata in which the new color of each cell depends on the previous colors of two rather than three neighboring cells.



With k possible colors for each cell, there are a total of k^{k^2} possible rules of this type, each specified by a k^2-digit number in base k (7743 for the rule shown above). For $k = 2$, there are 16 possible rules, and the most complicated pattern obtained is nested like the rule 90 elementary cellular automaton. With $k = 3$, there are 19,683 possible rules, 1734 of which are fundamentally inequivalent, and many more complicated patterns are seen, as in the pictures at the top of the next page.



With *rule* given by *IntegerDigits[num, k, k^2]* a single step of evolution can be implemented as

```
CAStep[{k_, rule_}, a_List] := rule[[k^2 - RotateLeft[a] - k a]]
```

▪ **Page 60 · Numbers of rules.** Allowing k possible colors for each cell and considering r neighbors on each side, there are $k^{k^{2r+1}}$ possible cellular automaton rules in all, of which $k^{1/2\,k^{r+1}\,(1+k^r)}$ are symmetric, and $k^{1+(k-1)\,(2r+1)}$ are totalistic. (For $k = 2$, $r = 1$ there are therefore 256 possible rules altogether, of which 16 are totalistic. For $k = 2$, $r = 2$ there are 4,294,967,296 rules in all, of which 64 are totalistic. And for $k = 3$, $r = 1$ there are 7,625,597,484,987 rules in all, with 2187 totalistic ones.) Note that for $k > 2$, a particular rule will in general be totalistic only for a specific assignment of values to colors. I first introduced totalistic rules in 1983.

▪ **Implementation of general cellular automata.** With k colors and r neighbors on each side, a single step in the evolution of a general cellular automaton is given by

```
CAStep[CARule[rule_List, k_, r_], a_List] :=
  rule[[-1 - ListConvolve[k^Range[0, 2r], a, r + 1]]]
```

where *rule* is obtained from a rule number *num* by *IntegerDigits[num, k, k^(2r+1)]*. (See also page 927.)

▪ **Implementation of totalistic cellular automata.** To handle totalistic rules that involve k colors and nearest neighbors, one can add the definition

```
CAStep[TotalisticCARule[rule_List, 1], a_List] :=
  rule[[-1 - (RotateLeft[a] + a + RotateRight[a])]]
```

to what was given on page 867. The following definition also handles the more general case of r neighbors:

```
CAStep[TotalisticCARule[rule_List, r_Integer], a_List] :=
  rule[[-1 - Sum[RotateLeft[a, i], {i, -r, r}]]]
```

One can generate the representation of totalistic rules used by these functions from code numbers using

```
ToTotalisticCARule[num_Integer, k_Integer, r_Integer] :=
 TotalisticCARule[IntegerDigits[num, k, 1 + (k - 1) (2r + 1)], r]
```

▪ **Common framework.** The *Mathematica* built-in function *CellularAutomaton* discussed on page 867 handles general and totalistic rules in the same framework by using *ListConvolve[w, a, r + 1]* and taking the weights w to be respectively *k^Table[i - 1, {i, 2r + 1}]* and *Table[1, {2r + 1}]*.

▪ **Page 63 · Mod 3 rule.** Code 420 is an example of an additive rule, and yields a pattern corresponding to Pascal's triangle modulo 3, as discussed on page 870.

▪ **Compositions of cellular automata.** One way to construct more complicated rules is from compositions of simpler rules. One can, for example, consider each step applying first one elementary cellular automaton rule, then another. The result is in effect a $k = 2$, $r = 2$ rule. Usually the order in which the two elementary rules are applied will matter, and the overall behavior obtained will have no simple relationship to that of either of the individual rules. (See also page 956.)

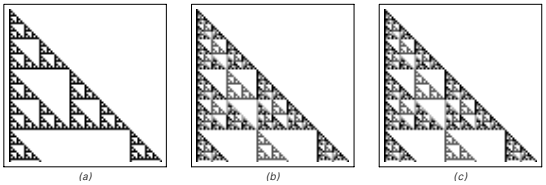
▪ **Rules based on algebraic systems.** If the values of cells are taken to be elements of some finite algebraic system, then one can set up a cellular automaton with rule

```
a[t_, i_] := f[a[t - 1, i - 1], a[t - 1, i]]
```

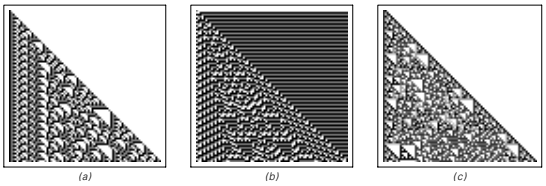
where f is the analog of multiplication for the system (see also page 1094). The pattern obtained after t steps is then given by

```
NestList[f[RotateRight[#], #] &, init, t]
```

The pictures below show results with f being *Times*, and cells having values (a) *{1, -1}*, (b) the unit complex numbers *{1, i, -1, -i}*, (c) the unit quaternions.



In general, with n elements f can be specified by an $n \times n$ "multiplication table". For $n = 2$, the patterns obtained are at most nested. Pictures (a) and (b) below however correspond to the $n = 3$ multiplication tables *{{1, 1, 3}, {3, 3, 2}, {2, 2, 1}}* and *{{3, 1, 3}, {1, 3, 1}, {3, 1, 2}}*. Note that for (b) the table is symmetric, corresponding to a commutative multiplication operation.



If f is associative (flat), so that *f[f[i, j], k] == f[i, f[j, k]]*, then the algebraic system is known as a semigroup. (See also

page 805.) With a single cell seed, no pattern more complicated than nested can be obtained in such a system. And with any seed, it appears to require a semigroup with at least six elements to obtain a more complicated pattern.

If *f* has an identity element, so that *f[1, i] == i* for all *i*, and has inverses, so that *f[i, j] == 1* for some *j*, then the system is a group. (See page 945.) If the group is Abelian, so that *f[i, j] == f[j, i]*, then only nested patterns are ever produced (see page 955). But it turns out that the very simplest possible non-Abelian group yields the pattern in (c) above. The group used is S_3, which has six elements and multiplication table

```
{{1, 2, 3, 4, 5, 6}, {2, 1, 5, 6, 3, 4}, {3, 4, 1, 2, 6, 5},
 {4, 3, 6, 5, 1, 2}, {5, 6, 2, 1, 4, 3}, {6, 5, 4, 3, 2, 1}}
```

The initial condition contains *{5, 6}* surrounded by *1*'s.

Mobile Automata

■ **Implementation.** The state of a mobile automaton at a particular step can conveniently be represented by a pair *{list, n}*, where *list* gives the values of the cells, and *n* specifies the position of the active cell (the value of the active cell is thus *list[[n]]*). Then, for example, the rule for the mobile automaton shown on page 71 can be given as

```
{{1, 1, 1} → {0, 1}, {1, 1, 0} → {0, 1},
 {1, 0, 1} → {1, -1}, {1, 0, 0} → {0, -1}, {0, 1, 1} → {0, -1},
 {0, 1, 0} → {0, 1}, {0, 0, 1} → {1, 1}, {0, 0, 0} → {1, -1}}
```

where the left-hand side in each case gives the value of the active cell and its left and right neighbors, while the right-hand side consists of a pair containing the new value of the active cell and the displacement of its position. (In analogy with cellular automata, this rule can be labelled *{35, 57}* where the first number refers to colors, and the second displacements.) With a rule given in this form, each step in the evolution of the mobile automaton corresponds to the function

```
MAStep[rule_, {list_List, n_Integer}] /; 1 < n < Length[list] :=
    Apply[{ReplacePart[list, #1, n], n + #2} &,
        Replace[Take[list, {n - 1, n + 1}], rule]]
```

The complete evolution for many steps can then be obtained with

```
MAEvolveList[rule_, init_List, t_Integer] :=
  NestList[MAStep[rule, #] &, init, t]
```

(The program will run more efficiently if *Dispatch* is applied to the rule before giving it as input.)

For the mobile automaton on page 73, the rule can be given as

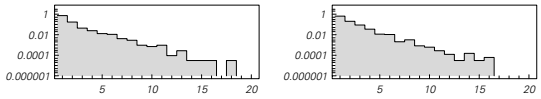
```
{{1, 1, 1} → {{0, 0, 0}, -1}, {1, 1, 0} → {{1, 0, 1}, -1},
 {1, 0, 1} → {{1, 1, 1}, 1}, {1, 0, 0} → {{1, 0, 0}, 1},
 {0, 1, 1} → {{0, 0, 0}, 1}, {0, 1, 0} → {{0, 1, 1}, -1},
 {0, 0, 1} → {{1, 0, 1}, 1}, {0, 0, 0} → {{1, 1, 1}, 1}}
```

and *MAStep* must be rewritten as

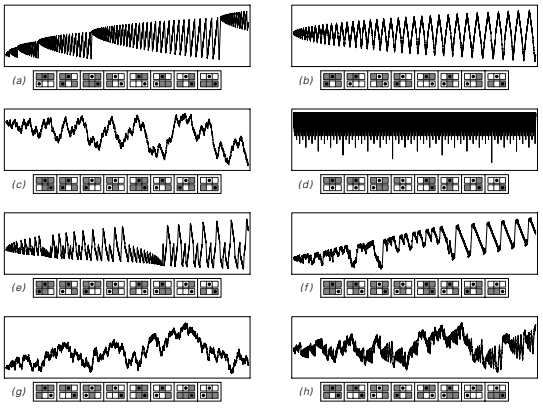
```
MAStep[rule_, {list_List, n_Integer}] /; 1 < n < Length[list] :=
  Apply[{Join[Take[list, {1, n - 2}], #1, Take[list, {n + 2, -1}]],
    n + #2} &, Replace[Take[list, {n - 1, n + 1}], rule]]
```

■ **Compressed evolution.** An alternative compression scheme for mobile automata is discussed on page 488.

■ **Page 72 · Distribution of behavior.** The pictures below show the distributions of transient and of period lengths for the 65,318 mobile automata of the type described here that yield ultimately repetitive behavior. Rule (f) has a period equal to the maximum of 16.



■ **Page 75 · Active cell motion.** The pictures below show the positions of the active cell for 20,000 steps of evolution in various mobile automata. (a), (b) and (c) correspond respectively to the rules on pages 73, 74 and 75. (c) has an outer envelope whose edges grow at rates $\{-1.5, 0.3\}\sqrt{t}$. (d) yields logarithmic growth as shown on page 496 (like Turing machine (f) on page 79). In most cases where the behavior is ultimately repetitive, transients and periods seem to follow the same approximate exponential distribution as in the note above. (g) however suddenly yields repetitive behavior with period 4032 after 405,941 steps. (h) does not appear to evolve to strict repetition or nesting, but does show progressively longer patches with fairly orderly behavior. (c) shows no obvious deviation from randomness in at least the first billion steps (after which the pattern it produces is 57,014 cells wide).



■ **Implementation of generalized mobile automata.** The state of a generalized mobile automaton at a particular step can be

specified by *{list, nlist}*, where *list* gives the values of the cells, and *nlist* is a list of the positions of active cells. The rule can be given by specifying a list of cases such as *{0, 0, 0}→{1, {1, -1}}*, where in each case the second sublist specifies the new relative positions of active cells. With this setup successive steps in the evolution of the system can be obtained from

```
GMAStep[rules_, {list_, nlist_}] := Module[{a, na}, {a, na} =
  Transpose[Map[Replace[Take[list, {# - 1, # + 1}], rules] &,
   nlist]]; {Fold[ReplacePart[#1, Last[#2], First[#2]] &,
   list, Transpose[{nlist, a}]], Union[Flatten[nlist + na]]}]
```

Turing Machines

■ **Implementation.** The state of a Turing machine at a particular step can be represented by the triple *{s, list, n}*, where *s* gives the state of the head, *list* gives the values of the cells, and *n* specifies the position of the head (the cell under the head thus has value *list[[n]]*). Then, for example, the rule for the Turing machine shown on page 78 can be given as

```
{{1, 0}→{3, 1, -1}, {1, 1}→{2, 0, 1}, {2, 0}→{1, 1, 1},
 {2, 1}→{3, 1, 1}, {3, 0}→{2, 1, 1}, {3, 1}→{1, 0, -1}}
```

where the left-hand side in each case gives the state of the head and the value of the cell under the head, and the right-hand side consists of a triple giving the new state of the head, the new value of the cell under the head and the displacement of the head.

With a rule given in this form, a single step in the evolution of the Turing machine can be implemented with the function

```
TMStep[rule_List, {s_, a_List, n_}] /; 1 ≤ n ≤ Length[a] :=
  Apply[{#1, ReplacePart[a, #2, n], n + #3} &,
   Replace[{s, a[[n]]}, rule]]
```

The evolution for many steps can then be obtained using

```
TMEvolveList[rule_, init_List, t_Integer] :=
  NestList[TMStep[rule, #] &, init, t]
```

An alternative approach is to represent the complete state of the Turing machine by *MapAt[{s, #} &, list, n]*, and then to use

```
TMStep[rule_, c_] := Replace[c,
  {a___, x_, h_List, y_, b___} :> Apply[{{a, x, #2, {#1, y}, b},
    {a, {#1, x}, #2, y, b}}[[#3]] &, h /. rule]]
```

The result of *t* steps of evolution from a blank tape can also be obtained from (see also page 1143)

```
s = 1; a[_] = 0; n = 0;
Do[{s, a[n], d} = {s, a[n]} /. rule; n += d, {t}]
```

■ **Number of rules.** With *k* possible colors for each cell and *s* possible states, there are a total of $(2sk)^{sk}$ possible Turing machine rules. Often many of these rules are immediately equivalent, or can show only very simple behavior (see page 1120).

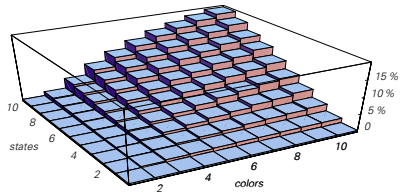
■ **Numbering scheme.** One can number Turing machines and get their rules using

```
Flatten[MapIndexed[{{1, -1}#2 + {0, k} → {1, 1, 2}
    Mod[Quotient[#1, {2 k, 2, 1}], {s, k, 2}] + {1, 0, -1} &,
  Partition[IntegerDigits[n, 2 s k, s k], k], {2}]]
```

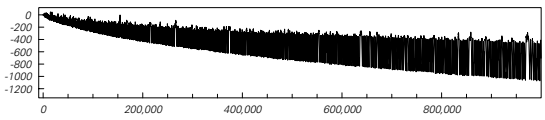
The examples on page 79 have numbers 3024, 982, 925, 1971, 2506 and 1953.

■ **Page 79 · Counter machine.** Turing machine (f) operates like a base 2 counter: at steps where its head is at the leftmost position, the colors of the cells correspond to the reverse of the base 2 digit sequences of successive numbers. All possible arrangements of colors are thus eventually produced. The overall pattern attains width *j* after $2^j - j$ steps.

■ **Page 80 · Distribution of behavior.** With 2 possible states and 2 possible colors for each cell, starting from a blank tape, the maximum repetition period obtained is 9 steps, and 12 out of the 4096 possible rules (or about 0.29%) yield non-repetitive behavior. With 3 states and 2 colors, the maximum period is 24, and about 0.37% of rules yield non-repetitive behavior, always nested. (Usually I have not found more complicated behavior in such rules even with initial conditions in which there are both black and white cells, though see page 761.) With 2 states and 3 colors, the maximum repetition period is again 24, about 0.65% of rules yield non-repetitive behavior, and the 14 rules discussed on page 709 yield more complex behavior. With more colors or more states, the percentage of rules that yield non-repetitive behavior steadily increases, as shown below, roughly like $0.28(s-1)(k-1)$. (Compare page 1120.)

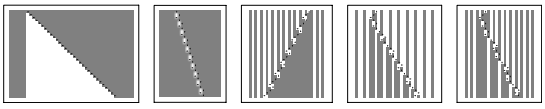


■ **Page 81 · Head motion.** The picture below shows the motion of the head for the first million steps. After about 20,000 steps, the width of the pattern produced grows at a rate close to $\sqrt{t}$.



■ **Localized structures.** Even when the overall behavior of a Turing machine is complicated, it is possible for simple localized structures to exist, much as in cellular automata

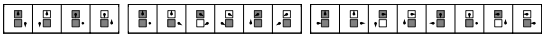
such as rule 110. What can happen is that with certain specific repetitive backgrounds, the head can move in a simple repetitive way, as shown in the pictures below for the Turing machine from page 81.



■ **History.** Turing machines were invented by Alan Turing in 1936 to serve as idealized models for the basic processes of mathematical calculation (see page 1128). As discussed on page 1110, Turing's main interest was in showing what his machines could in principle be made to do, not in finding out what simple examples of them actually did. Indeed, so far as I know, even though he had access to the necessary technology, Turing never explicitly simulated any Turing machine on a computer.

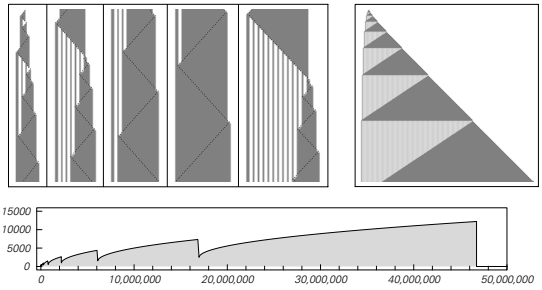
Since Turing's time, Turing machines have been extensively used as abstract models in theoretical computer science. But in almost no cases has the explicit behavior of simple Turing machines been considered. In the early 1960s, however, Marvin Minsky and others did work on finding the simplest Turing machines that could exhibit certain properties. Most of their effort was devoted to finding ingenious constructions for creating appropriate machines (see page 1119). But around 1961 they did systematically study all 4096 2-state 2-color machines, and simulated the behavior of some simple Turing machines on a computer. They found repetitive and nested behavior, but did not investigate enough examples to discover the more complex behavior shown in the main text.

As an offshoot of abstract studies of Turing machines, Tibor Radó in 1962 formulated what he called the Busy Beaver Problem: to find a Turing machine with a specified number of states that "keeps busy" for as many steps as possible before finally reaching a particular "halt state" (numbered 0 below). (A variant of the problem asks for the maximum number of black cells that are left when the machine halts.) By 1966 the results for 2, 3 and 4 states had been found: the maximum numbers of steps are 6, 21 and 107, respectively, with 4, 5 and 13 final black cells. Rules achieving these bounds are:



The result for 5 states is still unknown, but a machine taking 47,176,870 steps and leaving 4098 black cells was found by Heiner Marxen and Jürgen Buntrock in 1990. Its rule is:

The pictures below show (a) the first 500 steps of evolution, (b) the first million steps in compressed form and (c) the number of black cells obtained at each step. Perhaps not surprisingly for a system optimized to run as long as possible, the machine operates in a rather systematic and regular way. With 6 states, a machine is known that takes about 3.002×10^{1730} steps to halt, and leaves about 1.29×10^{865} black cells. (See also page 1144.)



Substitution Systems

■ **Implementation.** The rule for a neighbor-independent substitution system such as the first one on page 82 can conveniently be given as *{1→{1, 0}, 0→{0, 1}}*. And with this representation, the evolution for *t* steps is given by

```
SSEvolveList[rule_, init_List, t_Integer] :=
  NestList[Flatten[# /. rule] &, init, t]
```

where in the first example on page 82, the initial condition is *{1}*.

An alternative approach is to use strings, representing the rule by *{"B"→"BA", "A"→"AB"}* and the initial condition by *"B"*. In this case, the evolution can be obtained using

```
SSEvolveList[rule_, init_String, t_Integer] :=
  NestList[StringReplace[#, rule] &, init, t]
```

For a neighbor-dependent substitution system such as the first one on page 85 the rule can be given as

```
{{1, 1}→{0, 1}, {1, 0}→{1, 0}, {0, 1}→{0}, {0, 0}→{0, 1}}
```

And with this representation, the evolution for *t* steps is given by

```
SS2EvolveList[rule_, init_List, t_Integer] :=
  NestList[Flatten[Partition[#, 2, 1] /. rule] &, init, t]
```

where the initial condition for the first example on page 85 is *{0, 1, 1, 0}*.

■ **Page 83 · Properties.** The examples shown here all appear in quite a number of different contexts in this book. Note that each of them in effect yields a single sequence that gets progressively longer at each step; other rules make the colors of elements alternate on successive steps.

(a) *(Successive digits sequence)* The sequence produced is repetitive, with the element at position *n* being black for *n*

odd and white for n even. There are a total of 2^t elements after t steps. The complete pattern formed by looking at all the steps together has the same structure as the arrangement of base 2 digits in successive numbers shown on page 117.

(b) *(Thue-Morse sequence)* The color *s[n]* of the element at position n is given by *1 - Mod[DigitCount[n - 1, 2, 1], 2]*. These colors satisfy *s[n_] := If[EvenQ[n], 1 - s[n/2], s[(n + 1)/2]]* with *s[1] = 1*. There are a total of 2^t elements in the sequence after t steps. The sequence on step t can be obtained from *Nest[Join[#, 1 - #] &, {1}, t - 1]*. The number of black and white elements at each step is always the same. All four possible pairs of successive elements occur, though not with equal frequency. Runs of three identical elements never occur, and in general no block of elements can ever occur more than twice. The first 2^m elements in the sequence can be obtained from (see page 1081)

```
(CoefficientList[Product[1 - z^2^s, {s, 0, m - 1}], z] + 1)/2
```

The first n elements can also be obtained from (see page 1092)

```
Mod[CoefficientList[Series[(1 + Sqrt[(1 - 3 x)/(1 + x)])/
    (2 (1 + x)), {x, 0, n - 1}], x], 2]
```

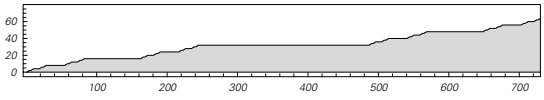
The sequence occurs many times in this book; it can for example be derived from a column of values in the rule 150 cellular automaton pattern discussed on page 885.

(c) *(Fibonacci-related sequence)* The sequence at step t can be obtained from *a[t_] := Join[a[t - 1], a[t - 2]]; a[1] = {0}; a[2] = {0, 1}*. This sequence has length *Fibonacci[t + 1]* (or approximately 1.618^{t+1}) (see note below). The color of the element at position n is given by *2 - (Floor[(n + 1) GoldenRatio] - Floor[n GoldenRatio])* (see page 904), while the position of the k^{th} white element is given by the so-called Beatty sequence *Floor[k GoldenRatio]*. The ratio of the number of white elements to black at step t is *Fibonacci[t - 1]/Fibonacci[t - 2]*, which approaches *GoldenRatio* for large t. For all $m \le$ *Fibonacci[t - 1]*, the number of distinct blocks of m successive elements that actually appear out of the 2^m possibilities is $m + 1$ (making it a so-called Sturmian sequence as discussed on page 1084).

(d) *(Cantor set)* The color of the element at position n is given by *If[FreeQ[IntegerDigits[n - 1, 3], 1], 1, 0]*, which turns out to be equivalent to

```
If[OddQ[n], Sign[Mod[Binomial[n - 1, (n - 1)/2], 3]], 0, 1]
```

There are 3^t elements after t steps, of which 2^t are black. The picture below shows the number of black cells that occur before position n. The resulting curve has a nested form, with envelope *n^Log[3, 2]*.



■ **Growth rates.** The total number of elements of each color that occur at each step in a neighbor-independent substitution system can be found by forming the matrix m where *m[[i, j]]* gives the number of elements of color *j + 1* that appear in the block that replaces an element of color *i + 1*. For case (c) above, *m = {{1, 1}, {1, 0}}*. A list that gives the number of elements of each color at step t can then be found from *init . MatrixPower[m, t]*, where *init* gives the initial number of elements of each color—*{1, 0}* for case (c) above. For large t, the total number of elements typically grows like λ^t, where λ is the largest eigenvalue of m; the relative numbers of elements of each color are given by the corresponding eigenvector. For case (c), λ is *GoldenRatio*, or $(1 + \sqrt{5})/2$. There are exceptional cases where $\lambda == 1$, so that the growth is not exponential. For the rule *{0 → {0, 1}, 1 → {1}}*, *m = {{1, 1}, {0, 1}}*, and the number of elements at step t starting with *{0}* is just t. For *{0 → {0, 1}, 1 → {1, 2}, 2 → {2}}*, *m = {{1, 1, 0}, {0, 1, 1}, {0, 0, 1}}*, and the number of elements starting with *{0}* is $(t^2 - t + 2)/2$. For neighbor-independent rules, the growth for large t must follow an exponential or an integer power less than the number of possible colors. For neighbor-dependent rules, any form of growth can in principle be obtained.

■ **Fibonacci numbers.** The Fibonacci numbers *Fibonacci[n]* (*f[n]* for short) can be generated by the recurrence relation

```
f[n_] := f[n] = f[n - 1] + f[n - 2]
f[1] = f[2] = 1
```

The first few Fibonacci numbers are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377. For large n the ratio *f[n]/f[n - 1]* approaches *GoldenRatio* or $(1 + \sqrt{5})/2 \simeq 1.618$.

Fibonacci[n] can be obtained in many ways:

- $(\text{GoldenRatio}^n - (-\text{GoldenRatio})^{-n})/\sqrt{5}$
- *Round[GoldenRatio^n/Sqrt[5]]*
- 2^{1-n} *Coefficient[(1 + Sqrt[5])^n, Sqrt[5]]*
- *MatrixPower[{{1, 1}, {1, 0}}, n - 1][[1, 1]]*
- *Numerator[NestList[1/(1 + #) &, 1, n]]*
- *Coefficient[Series[1/(1 - t - t^2), {t, 0, n}], t^(n-1)]*
- *Sum[Binomial[n - i - 1, i], {i, 0, (n - 1)/2}]*
- 2^{n-2} *- Count[IntegerDigits[Range[0, 2^(n-2)], 2], {___, 1, 1, ___}]*

A fast method for evaluating *Fibonacci[n]* is

```
First[Fold[f, {1, 0, -1}, Rest[IntegerDigits[n, 2]]]]
f[{a_, b_, s_}, 0] = {a (a + 2 b), s + a (2 a - b), 1}
f[{a_, b_, s_}, 1] = {-s + (a + b) (a + 2 b), a (a + 2 b), -1}
```

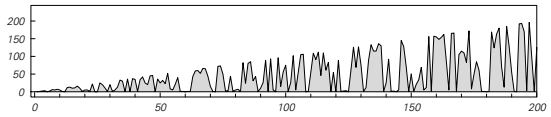
Fibonacci numbers appear to have first arisen in perhaps 200 BC in work by Pingala on enumerating possible patterns of

poetry formed from syllables of two lengths. They were independently discussed by Leonardo Fibonacci in 1202 as solutions to a mathematical puzzle concerning rabbit breeding, and by Johannes Kepler in 1611 in connection with approximations to the pentagon. Their recurrence relation appears to have been understood from the early 1600s, but it has only been in the past very few decades that they have in general become widely discussed.

For $m > 1$, the value of n for which m == *Fibonacci[n]* is *Round[Log[GoldenRatio, $\sqrt{5}$ m]]*.

The sequence *Mod[Fibonacci[n], k]* is always purely repetitive; the maximum period is $6k$, achieved when $k = 10\,5^m$ (compare page 975).

Mod[Fibonacci[n], n] has the fairly complicated form shown below. It appears to be zero only when n is of the form 5^m or $12q$, where q is not prime ($q > 5$).



The number *GoldenRatio* appears to have been used in art and architecture since antiquity. *1/GoldenRatio* is the default *AspectRatio* for *Mathematica* graphics. In addition:

- *GoldenRatio* is the solution to x == $1 + 1/x$ or x^2 == $x + 1$
- The right-hand rectangle in ▭ is similar to the whole rectangle when the aspect ratio is *GoldenRatio*
- *Cos[π/5] == Cos[36°] == GoldenRatio/2*
- The ratio of the length of the diagonal to the length of a side in a regular pentagon is *GoldenRatio*
- The corners of an icosahedron are at coordinates
  Flatten[Array[NestList[RotateRight, {0, (−1)^#1 GoldenRatio, (−1)^#2}, 3] &, {2, 2}], 2]
- *1 + FixedPoint[N[1/(1 + #), k] &, 1]* approximates *GoldenRatio* to k digits, as does *FixedPoint[N[Sqrt[1 + #], k] &, 1]*
- A successive angle difference of *GoldenRatio* radians yields points maximally separated around a circle (see page 1006).

■ **Lucas numbers.** Lucas numbers *Lucas[n]* satisfy the same recurrence relation *f[n_] := f[n − 1] + f[n − 2]* as Fibonacci numbers, but with the initial conditions *f[1] = 1; f[2] = 3*. Among the relations satisfied by Lucas numbers are:

- *Lucas[n_] := Fibonacci[n − 1] + Fibonacci[n + 1]*
- *GoldenRatio^n == (Lucas[n] + Fibonacci[n] $\sqrt{5}$)/2*

■ **Generalized Fibonacci sequences.** Any linear recurrence relation yields sequences with many properties in common with the Fibonacci numbers—though with *GoldenRatio* replaced by other algebraic numbers. The Perrin sequence *f[n_] := f[n − 2] + f[n − 3]; f[0] = 3; f[1] = 0; f[2] = 2* has the peculiar property that *Mod[f[n], n] == 0* mostly but not always only for n prime. (For more on recurrence relations see page 128.)

■ **Connections with digit sequences.** In a sequence generated by a neighbor-independent substitution system the color of the element at position n turns out always to be related to the digit sequence of the number n in an appropriate base. The basic reason for this is that as shown on page 84 the evolution of the substitution system always yields a tree, and the successive digits in n determine which branch is taken at each level in order to reach the element at position n. In cases (a) and (b) on pages 83 and 84, the tree has two branches at every node, and so the base 2 digits of n determine the successive left and right branches that must be taken. Given that a branch with a certain color has been reached, the color of the branch to be taken next is then determined purely by the next digit in the digit sequence of n. For case (b) on pages 83 and 84, the rule that gives the color of the next branch in terms of the color of the current branch and the next digit is *{{0, 0} → 0, {0, 1} → 1, {1, 0} → 1, {1, 1} → 0}*. In terms of this rule, the color of the element at position n is given by

Fold[Replace[{#1, #2}, rule] &, 1, IntegerDigits[n − 1, 2]]

The rule used here can be thought of as a finite automaton with two states. In general, the behavior of any neighbor-independent substitution system where each element is subdivided into exactly k elements can be reproduced by a finite automaton with k states operating on digit sequences in base k. The nested structure of the patterns produced is thus a direct consequence of the nesting seen in the patterns of these digit sequences, as shown on page 117.

Note that if the rule for the finite automaton is represented for example as *{{1, 2}, {2, 1}}* where each sublist corresponds to a particular state, and the elements of the sublist give the successor states with inputs *Range[0, k − 1]*, then the n^{th} element in the output sequence can be obtained from

Fold[rule[[#1, #2]] &, 1, IntegerDigits[n − 1, k] + 1] − 1

while the first k^m elements can be obtained from

Nest[Flatten[rule[[#]]] &, 1, m] − 1

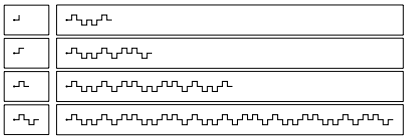
To treat examples such as case (c) where elements can subdivide into blocks of several different lengths one must generalize the notion of digit sequences. In base k a number is constructed from a digit sequence *a[r], …, a[1], a[0]* (with $0 \le a[i] < k$) according to *Sum[a[i] k^i, {i, 0, r}]*. But given a sequence of digits that are each 0 or 1, it is also possible for example to construct numbers according to

Sum[a[i] Fibonacci[i + 2], {i, 0, r}]. (As discussed on page 1070, this representation is unique so long as one does not allow any pairs of adjacent 1's in the digit sequence.) It then turns out that if one expresses the position *n* as a generalized digit sequence of this kind, then the color of the corresponding element in substitution system (c) is just the last digit in this sequence.

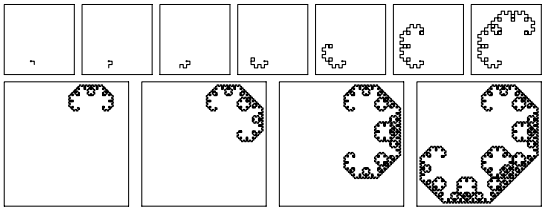
■ **Connections with square roots.** Substitution systems such as (c) above are related to projections of lines with quadratic irrational slopes, as discussed on page 904.

■ **Spectra of substitution systems.** See page 1080.

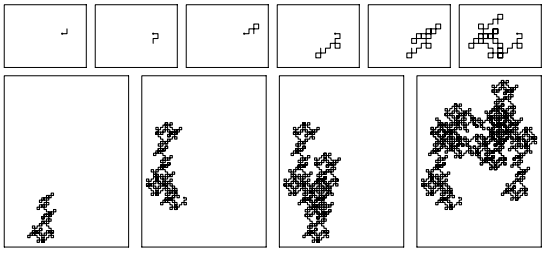
■ **Representation by paths.** An alternative to representing substitution systems by 1D sequences of black and white squares is to use 2D paths consisting of sequences of left and right turns. The paths obtained at successive steps for rule (b) above are shown below.



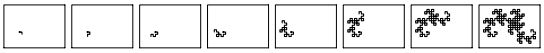
The pictures below show paths obtained with the rule *{1 → {1}, 0 → {0, 0, 1}}*, starting from *{0}*. Note the similarity to the 2D system shown on page 190.



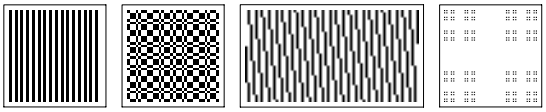
When the paths do not cross themselves, nested structure is evident. But in a case like the rule *{1 → {0, 0, 1}, 0 → {1, 0}}* starting with *{1}*, the presence of many crossings tends to hide such regularity, as in the pictures below.



■ **Paperfolding sequences.** The sequence of up and down creases in a strip of paper that is successively folded in half is given by a substitution system; after *t* steps the sequence turns out to be *NestList[Join[#, {0}, Reverse[1 - #]] &, {0}, t]*. The corresponding path (effectively obtained by making each crease a right angle) is shown below. (See page 189.)



■ **2D representations.** Individual sequences from 1D substitution systems can be displayed in 2D by breaking them into a succession of rows. The pictures below show results for the substitution systems on page 83. In case (b), with rows chosen to be 2^j elements in length, the leftmost column will always be identical to the beginning of the sequence, and in addition every interior element will be black exactly when the cell at the top of its column has the same color as the one at the beginning of its row. In case (c), stripes appear at angles related to *GoldenRatio*.



■ **Page 84 · Other examples.**

(a) *(Period-doubling sequence)* After *t* steps, there are a total of 2^t elements, and the sequence is given by *Nest[MapAt[1 - # &, Join[#, #], -1] &, {0}, t]*. It contains a total of *Round[2^t/3]* black elements, and if the last element is dropped, it forms a palindrome. The n^{th} element is given by *Mod[IntegerExponent[n, 2], 2]*. As discussed on page 885, the sequence appears in a vertical column of cellular automaton rule 150. The Thue-Morse sequence discussed on page 890 can be obtained from it by applying

```
1 - Mod[Flatten[Partition[FoldList[Plus, 0, list], 1, 2]], 2]
```

(b) The n^{th} element is simply *Mod[n, 2]*.

(c) Same as (a), after the replacement *1 → {1, 1}* in each sequence. Note that the spectra of (a) and (c) are nevertheless different, as discussed on page 1080.

(d) The length of the sequence at step *t* satisfies *a[t] == 2 a[t - 1] + a[t - 2]*, so that *a[t] = Round[(1 + √2)^(t-1)/2]* for *t > 1*. The number of white elements at step *t* is then *Round[a[t]/√2]*. Much like example (c) on page 83 there are *m + 1* distinct blocks of length *m*, and with *f = Floor[(1 - 1/√2)(# + 1/√2)] &* the n^{th} element of the sequence is given by *f[n + 1] - f[n]* (see page 903).

(e) For large t the number of elements increases like λ^t with $\lambda = (\sqrt{13} + 1)/2$; there are always λ times as many white elements as black ones.

(f) The number of elements at step t is $Round[(1 + \sqrt{2})^t/2]$, and the n^{th} element is given by $Floor[\sqrt{2}\,(n+1)] - Floor[\sqrt{2}\,n]$ (see page 903).

(g) The number of elements is the same as in (f).

(h) The number of black elements is 2^{t-1}; the total number of elements is $2^{t-2}\,(t+1)$.

(i) and (j) The total number of elements is 3^{t-1}.

■ **History.** In their various representations, 1D substitution systems have been invented independently many times for many different purposes. (For the history of fractals and 2D substitution systems see page 934.) Viewed as generators of sequences with certain combinatorial properties, substitution systems such as example (b) on page 83 appeared in the work of Axel Thue in 1906. (Thue's stated purpose in this work was to develop the science of logic by finding difficult problems with possible connections to number theory.) The sequence of example (b) was rediscovered by Marston Morse in 1917 in connection with his development of symbolic dynamics—and in finding what could happen in discrete approximations to continuous systems. Studies of general neighbor-independent substitution systems (sometimes under such names as sequence homomorphisms, iterated morphisms and uniform tag systems) have continued in this context to this day. In addition, particularly since the 1980s, they have been studied in the context of formal language theory and the so-called combinatorics of words. (Period-doubling phenomena also led to contact with physics starting in the late 1970s.)

Independent of work in symbolic dynamics, substitution systems viewed as generators of sequences were reinvented in 1968 by Aristid Lindenmayer under the name of L systems for the purpose of constructing models of branching plants (see page 1005). So-called 0L systems correspond to my neighbor-independent substitution systems; 1L systems correspond to the neighbor-dependent substitution systems on page 85. Work on L systems has proceeded along two quite different lines: modelling specific plant systems, and investigating general computational capabilities. In the mid-1980s, particularly through the work of Alvy Ray Smith, L systems became widely used for realistic renderings of plants in computer graphics.

The idea of constructing abstract trees such as family trees according to definite rules presumably goes back to antiquity. The tree representation of rule (c) from page 83 was for example probably drawn by Leonardo Fibonacci in 1202.

The first six levels of the specific pattern in example (a) on page 83 correspond exactly to the segregation diagram for the I Ching that arose in China as early as 2000 BC. Black regions represent yin and white ones yang. The elements on level six correspond to the 64 hexagrams of the I Ching. At what time the segregation diagram was first drawn is not clear, but it was almost certainly before 1000 AD, and in the 1600s it appears to have influenced Gottfried Leibniz in his development of base 2 numbers.

Viewed in terms of digit sequences, example (d) from page 83 was discussed by Georg Cantor in 1883 in connection with his investigations of the idea of continuity. General relations between digit sequences and sequences produced by neighbor-independent substitution systems were found in the 1960s. Connections of sequences such as (c) to algebraic numbers (see page 903) arose in precursors to studies of wavelets.

Paths representing sequences from 1D substitution systems can be generated by 2D geometrical substitution systems, as on page 189. The "C" curve shown on the facing page and on page 190 was for example described by Paul Lévy in 1937, and was rediscovered as the output of a simple computer program by William Gosper in the 1960s. Paperfolding or so-called dragon curves (as shown above) were discussed by John Heighway in the mid-1960s, and were analyzed by Chandler Davis, Donald Knuth and others. These curves have the property that they eventually fill space. Space-filling curves based on slightly more complicated substitution systems were already discussed by Giuseppe Peano in 1890 and by David Hilbert in 1891 in connection with questions about the foundations of calculus.

Sequences from substitution systems have no doubt appeared over the years as incidental features of great many pieces of mathematical work. As early as 1851, for example, Eugène Prouhet showed that if sequences of integers were partitioned according to sequence (b) on page 83, then sums of powers of these integers would be equal: thus $Apply[Plus, Flatten[Position[s, i]]^k]$ is equal for $i = 0$ and $i = 1$ if s is a sequence of the form (b) on page 83 with length 2^m, $m > k$. The optimal solution to the Towers of Hanoi puzzle invented in 1883 also turns out to be an example of a substitution system sequence.

Sequential Substitution Systems

■ **Implementation.** Sequential substitution systems can be implemented quite directly by using *Mathematica*'s standard

mechanism for applying transformation rules to symbolic expressions. Having made the definition

```
Attributes[s] = Flat
```

the state of a sequential substitution system at a particular step can be represented by a symbolic expression such as `s[1, 0, 1, 0]`. The rule on page 82 can then be given simply as

```
s[1, 0] → s[0, 1, 0]
```

while the rule on page 85 becomes

```
{s[0, 1, 0] → s[0, 0, 1], s[0] → s[0, 1, 0]}
```

The *Flat* attribute of `s` makes these rules apply not only for example to the whole sequence `s[1, 0, 1, 0]` but also to any subsequence such as `s[1, 0]`. (With `s` being *Flat*, `s[s[1, 0], 1, s[0]]` is equivalent to `s[1, 0, 1, 0]` and so on. A *Flat* function has the mathematical property of being associative.) And with this setup, `t` steps of evolution can be found with

```
SSSEvolveList[rule_, init_s, t_Integer] :=
  NestList[# /. rule &, init, t]
```

Note that as an alternative to having `s` be *Flat*, one can explicitly set up rules based on patterns such as `s[x___, 1, 0, y___] → s[x, 0, 1, 0, y]`. And by using rules such as `s[x___, 1, 0, y___] :> {s[x, 0, 1, 0, y], Length[s[x]]}` one can keep track of the positions at which substitutions are made. (*StringReplace* replaces all occurrences of a given substring, not just the first one, so cannot be used directly as an alternative to having a flat function.)

■ **Capabilities.** Even with the single rule `{s[1, 0] → s[0, 1]}`, a sequential substitution system can sort its initial conditions so that all 0's occur before all 1's. (See also page 1113.)

■ **Order of replacements.** For many sequential substitution systems the evolution effectively stops because a string is produced to which none of the replacements given apply. In most sequential substitution systems there is more than one possible replacement that can in principle apply at a particular step, so the order in which the replacements are tried matters. (Multiway systems discussed on page 497 are what result if all possible replacements are performed at each step.) There are however special sequential substitution systems (those with the so-called confluence property discussed on page 1036) in which in a certain sense the order of replacements does not matter.

■ **History.** Sequential substitution systems are closely related to the multiway systems discussed on page 938, and are often considered examples of production systems or string rewriting systems. In the form I discuss here, they seem to have arisen first under the name "normal algorithms" in the work of Andrei Markov in the late 1940s on computability and the idealization of mathematical processes. Starting in the 1960s text editors like TECO and ed used sequential substitution system rules, as have string-processing languages such as SNOBOL and perl. *Mathematica* uses an analog of sequential substitution system rules to transform general symbolic expressions. The fact that new rules can be added to a sequential substitution system incrementally without changing its basic structure has made such systems popular in studies of adaptive programming.

Tag Systems

■ **Implementation.** With the rules for case (a) on page 94 given for example by

```
{2, {{0, 0} → {1, 1}, {1, 0} → {}, {0, 1} → {1, 0}, {1, 1} → {0, 0, 0}}}
```

the evolution of a tag system can be obtained from

```
TSEvolveList[{n_, rule_}, init_, t_] := NestList[If[Length[#] <
    n, {}, Join[Drop[#, n], Take[#, n] /. rule]] &, init, t]
```

An alternative implementation is based on applying to the list at each step rules such as

```
{{0, 0, s___} → {s, 1, 1}, {1, 0, s___} → {s},
  {0, 1, s___} → {s, 1, 0}, {1, 1, s___} → {s, 0, 0, 0}}
```

There are a total of $((k^{r+1} - 1)/(k - 1))^{k^n}$ possible rules if blocks up to length r can be added at each step and k colors are allowed. For $r = 3$, $k = 2$ and $n = 2$ this is 50,625.

■ **Page 94 · Randomness.** To get some idea of the randomness of the behavior, one can look at the sequence of first elements produced on successive steps. In case (a), the fraction of black elements fluctuates around 1/2; in (b) it approaches 3/4; in (d) it fluctuates around near 0.3548, while in (e) and (f) it does not appear to stabilize.

■ **History.** The tag systems that I consider are generalizations of those first discussed by Emil Post in 1920 as simple idealizations of certain syntactic reduction rules in Alfred Whitehead and Bertrand Russell's *Principia Mathematica* (see page 1149). Post's tag systems differ from mine in that his allow the choice of block that is added at each step to depend only on the very first element in the sequence at that step (see however page 670). (The lag systems studied in 1963 by Hao Wang allow dependence on more than just the first element, but remove only the first element.) It turns out that in order to get complex behavior in such systems, one needs either to allow more than two possible colors for each element, or to remove more than two elements from the beginning of the sequence at each step. Around 1921, Post apparently studied all tag systems of his type that involve removal and addition of no more than two elements at each step, and he concluded that none of them produced complicated behavior. But then he looked at rules that

remove three elements at each step, and he discovered the rule *{3, {{0, _, _}→{0, 0}, {1, _, _}→{1, 1, 0, 1}}}*. As he noted, the behavior of this rule varies considerably with the initial conditions used. But at least for all the initial conditions up to length 28, the rule eventually just leads to behavior that repeats with a period of 1, 2, 6, 10, 28 or 40. With more than two colors, one finds that rules of Post's type which remove just two elements at each step can yield complex behavior, even starting from an initial condition such as *{0, 0}*. An example is *{2, {{0, _}→{2, 1}, {1, _}→{0}, {2, _}→{0, 2, 1, 2}}}*. (See also pages 1113 and 1141.)

Cyclic Tag Systems

■ **Implementation.** With the rules for the cyclic tag system on page 95 given as *{{1, 1}, {1, 0}}*, the evolution can be obtained from

```
CTEvolveList[rules_, init_, t_] :=
  Map[Last, NestList[CTStep, {rules, init}, t]]
CTStep[{{r_, s___}, {0, a___}}] := {{s, r}, {a}}
CTStep[{{r_, s___}, {1, a___}}] := {{s, r}, Join[{a}, r]}
CTStep[{u_, {}}] := {u, {}}
```

The leading elements on many more than t successive steps can be obtained directly from

```
CTList[rules_, init_, t_] :=
  Flatten[Map[Last, NestList[CTListStep, {rules, init}, t]]]
CTListStep[{rules_, list_}] :=
  {RotateLeft[rules, Length[list]], Flatten[rules[[
      Mod[Flatten[Position[list, 1]], Length[rules], 1]]]]}
```

■ **Page 95 · Generalizations.** The implementation above immediately allows cyclic tag systems which cycle through a list of more than two blocks. (With just one block the behavior is always repetitive.) Cyclic tag systems which allow any value for each element can be obtained by adding the rule

```
CTStep[{{r_, s___}, {n_, a___}}] :=
  {{s, r}, Flatten[{a, Table[r, {n}]}]}
```

The leading elements in this case can be obtained using

```
CTListStep[{rules_, list_}] :=
 {RotateLeft[rules, Length[list]], With[{n = Length[rules]},
    Flatten[Apply[Table[#1, {#2}] &, Map[Transpose[
        {rules, #}] &, Partition[list, n, n, 1, 0]], {2}]]]}
```

■ **Mechanical implementation.** Cyclic tag systems admit a particularly straightforward mechanical implementation. Black and white balls are kept in a trough as in the picture below. At each step the leftmost ball in the trough is released, and if this ball is black (as determined, for example, by size) a mechanism causes a new block of balls to be added at the right-hand end of the trough. This mechanism can work in several ways; typically it will involve a rotary element that determines which case of the rule to use at each step. Rule (e) from the main text allows a particularly simple supply of new balls. Note that the system will inevitably fail if the trough overflows with balls.

■ **Page 96 · Properties.** Assuming that black and white elements occur in an uncorrelated way, then the sequences in a cyclic tag system with n blocks should grow by an average of *Count[Flatten[rules], 1]/n - 1* elements at each step. With $n = 2$ blocks, this means that growth can occur only if the total number of black elements in both blocks is more than 3. Rules such as *{{1, 0}, {0, 1}}* and *{{1, 1}, {0}}* therefore yield repetitive behavior with sequences of limited length.

Note that if all blocks in a cyclic tag system with n blocks have lengths divisible by n, then one can tell in advance on which steps blocks will be added, and the overall behavior obtained must correspond to a neighbor-independent substitution system. The rules for the relevant substitution system may however depend on the initial conditions for the cyclic tag system.

Flatten[{1, 0, CTList[{{1, 0, 0, 1}, {0, 1, 1, 0}}, {0, 1}, t]}]

gives for example the Thue-Morse substitution system *{1→{1, 0}, 0→{0, 1}}*.

In example (a), the elements are correlated, so that slower growth occurs than in the estimate above. In example (c), the elements are again correlated: the growth is by an average of $(\sqrt{5}-1)/2 \approx 0.618$ elements at each step, and the first elements on alternate steps form the same nested sequence as obtained from the substitution system *{1→{1, 0}, 0→{1}}*. In example (d), the frequency of 1's among the first elements of sequence is approximately 3/4; *{0, 0}* never occurs, and the frequency of *{1, 1}* is approximately 1/2. In example (e), the frequency of 1's is again about 3/4, but now *{0, 0}* occurs with frequency 0.05, *{1, 1}* occurs with frequency 0.55, while *{0, 0, 0}* and *{0, 1, 0}* cannot occur.

■ **History.** Cyclic tag systems were studied by Matthew Cook in 1994 in connection with working on the rule 110 cellular automaton for this book. The sequence *{1, 2, 2, 1, 1, 2, …}* defined by the property *list == Map[Length, Split[list]]* was suggested as a mathematical puzzle by William Kolakoski in 1965 and is equivalent to

Join[{1, 2}, Map[First, CTEvolveList[{{1}, {2}}, {2}, t]]]

It is known that this sequence does not repeat, contains no more than two identical consecutive blocks, and has at least very close to equal numbers of 1's and 2's. Replacing 2 by 3 yields a sequence which has a fairly simple nested form.

Register Machines

■ **Implementation.** The state of a register machine at a particular step can be represented by the pair *{n, list}*, where *n* gives the position in the program of current instruction being executed (the "program counter") and *list* gives the values of the registers. The program for the register machine on page 99 can then be given as

```
{i[1], d[2, 1], i[2], d[1, 3], d[2, 1]}
```

where *i[_]* represents an increment instruction, and *d[_, _]* a decrement jump.

With this setup, the evolution of any register machine can be implemented using the functions (a typical initial condition is *{1, {0, 0}}*)

```
RMStep[prog_, {n_Integer, list_List}] := If[n > Length[prog],
    {n, list}, RMExecute[prog[[n]], {n, list}]]
RMExecute[i[r_], {n_, list_}] := {n + 1, MapAt[# + 1 &, list, r]}
RMExecute[d[r_, m_], {n_, list_}] :=
  If[list[[r]] > 0, {m, MapAt[# - 1 &, list, r]}, {n + 1, list}]
RMEvolveList[prog_, init : {_Integer, _List}, t_Integer] :=
  NestList[RMStep[prog, #] &, init, t]
```

The total number of possible programs of length *n* using *k* registers is $(k (1 + n))^n$. Note that by prepending suitable *i[r]* instructions one can effectively set up initial conditions with arbitrary values in registers.

■ **Halting.** It is sometimes convenient to think of register machines as going into a special halt state if they try to execute instructions beyond the end of their program. (See page 1137.) The fraction of possible register machines that do this starting from initial condition *{1, {0, 0}}* decreases steadily with program length *n*, reaching about 0.76 for *n = 8*. The most common number of steps before halting is always *n*, while the maximum numbers of steps for *n* up to 8 is *{1, 3, 5, 10, 16, 37, 215, 1280}* where in the last case this is achieved by

```
{i[1], d[2, 7], d[2, 1], i[2], i[2], d[1, 4], i[1], d[2, 3]}
```

■ **Page 101 · Extended instruction sets.** One can consider also including instructions such as

```
RMExecute[eq[r1_, r2_, m_], {n_, list_}] :=
  If[list[[r1]] == list[[r2]], {m, list}, {n + 1, list}]
RMExecute[add[r1_, r2_], {n_, list_}] :=
  {n + 1, ReplacePart[list, list[[r1]] + list[[r2]], r1]}
RMExecute[jmp[r1_], {n_, list_}] := {list[[r1]], list}
```

Note that by being able to add and subtract only 1 at each step, the register machines shown in the main text necessarily operate quite slowly: they always take at least *n* steps to build up a number of size *n*. But while extending the instruction set can increase the speed of operations, it does not appear to yield a much larger density of machines with complex behavior.

■ **History.** Register machines (also known as counter machines and program machines) are a fairly obvious idealization of practical computers, and have been invented in slightly different forms several times. Early uses of them were made by John Shepherdson and Howard Sturgis around 1959 and Marvin Minsky around 1960. Somewhat similar constructs were part of Kurt Gödel's 1931 work on representing logic within arithmetic (see page 1158).

■ **Page 102 · Random programs.** See page 1182.

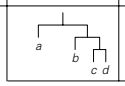
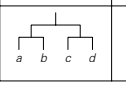
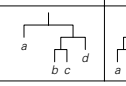
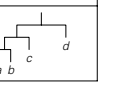
Symbolic Systems

■ **Implementation.** The evolution for *t* steps of the first symbolic system shown can be implemented simply by

```
NestList[# /. e[x_][y_] → x[x[y]] &, init, t]
```

■ **Symbolic expressions.** Expressions like *Log[x]* and *f[x]* that give values of functions are familiar from mathematics and from typical computer languages. Expressions like *f[g[x]]* giving compositions of functions are also familiar. But in general, as in *Mathematica*, it is possible to have expressions in which the head *h* in *h[x]* can itself be any expression—not just a single symbol. Thus for example *f[g][x]*, *f[g[h]][x]* and *f[g][h][x]* are all possible expressions. And these kinds of expressions often arise in *Mathematica* when one manipulates functions as a whole before applying them to arguments. ($\partial_{xx} f[x]$ for example gives *f″[x]* which is *Derivative[2][f][x]*.) (In principle one can imagine representing all objects with forms such as *f[x, y]* by so-called currying as *f[x][y]*, and indeed I tried this in the early 1980s in SMP. But although this can be convenient when *f* is a discrete function such as a matrix, it is inconsistent with general mathematical and other usage in which for example *Gamma[x]* and *Gamma[a, x]* are both treated as values of functions.)

■ **Representations.** Among the representations that can be used for expressions are:

functional	a[b[c[d]]]	a[b][c[d]]	a[b[c][d]]	a[b][c][d]
Polish	{∘, a, ∘, b, ∘, c, d}	{∘, ∘, a, b, ∘, c, d}	{∘, a, ∘, ∘, b, c, d}	{∘, ∘, ∘, a, b, c, d}
operator	a∘ (b∘ (c∘ d))	(a∘ b)∘ (c∘ d)	a∘ ((b∘ c)∘ d)	((a∘ b)∘ c)∘ d
tree	{a, {b, {c, d}}}	{{a, b}, {c, d}}	{a, {{b, c}, d}}	{{{a, b}, c}, d}
				

Typical transformation rules are non-local in all these representations. Polish representation (whose reverse form has been used in HP calculators) for an expression can be obtained using (see also page 1173)

```
Flatten[expr //. x_[y_] → {∘, x, y}]
```

The original expression can be recovered using

```
First[Reverse[list] //. {w___, x_, y_, ∘, z___} → {w, y[x], z}]
```

(Pictures of symbolic system evolution made with Polish notation differ in detail but look qualitatively similar to those made as in the main text with functional notation.)

The tree representation of an expression can be obtained using *expr //. x_[y_] → {x, y}*, and when each object has just one argument, the tree is binary, as in LISP.

If only a single symbol ever appears, then all that matters is the overall structure of an expression, which can be captured as in the main text by the sequence of opening and closing brackets, given by

```
Flatten[Characters[ToString[expr]] /.
  {"[" → 1, "]" → 0, "e" → {}}]
```

▪ **Possible expressions.** *LeafCount[expr]* gives the number of symbols that appear anywhere in an expression, while *Depth[expr]* gives the number of closing brackets at the end of its functional representation—equal to the number of levels in the rightmost branch of the tree representation. (The maximum number of levels in the tree can be computed from *expr /. _Symbol → 1 //. x_[y_] → 1 + Max[x, y]*.)

With a list *s* of possible symbols, *c[s, n]* gives all possible expressions with *LeafCount[expr] == n*:

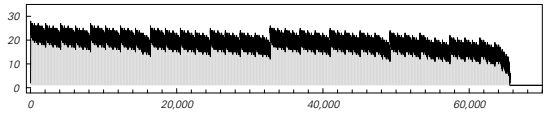
```
c[s_, 1] = s; c[s_, n_] := Flatten[
  Table[Outer[#1[#2] &, c[s, n - m], c[s, m]], {m, n - 1}]]
```

There are a total of *Binomial[2 n - 2, n - 1] Length[s]^n/n* such expressions. When *Length[s] == 1* the expressions correspond to possible balanced sequences of opening and closing brackets (see page 989).

▪ **Page 103 · Properties.** All initial conditions eventually evolve to expressions of the form *Nest[e, e, m]*, which then remain fixed. The quantity *expr //. {e → 0, x_[y_] → 2^x + y}* turns out to remain constant through the evolution, so this gives the final value of *m* for any initial condition. The maximum is *Nest[2^# &, 0, n]* (compare page 906), achieved for initial conditions of the form *Nest[#[e] &, e, n]*. (By analogy with page 1122 any *e* expression can be interpreted as a Church numeral *u = expr //. {e → 2, x_[y_] → y^x} = 2^(2^m)*, so that *expr[a][b]* evolves to *Nest[a, b, u]*.) During the evolution the rule can apply only to the inner part *FixedPoint[Replace[#, e[x_] → x] &, expr]* of an expression. The depth of this inner part for initial condition *e[e][e][e][e][e]* is shown below. For all initial conditions this depth seems at first to increase linearly, then to decrease in a nested way according to

```
FoldList[Plus, 0, Flatten[Table[
    {1, 1, Table[-1, {IntegerExponent[i, 2] + 1}]}, {i, m}]]]
```

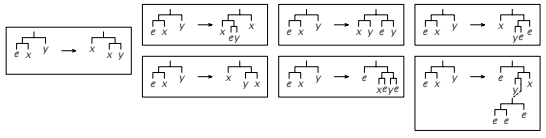
This quantity alternates between value *1* at position 2^j and value *j* at position $2^j - j + 1$. It reaches a fixed point as soon as the depth reaches 0. For initial conditions of size *n*, this occurs after at most *Sum[Nest[2^# &, 0, i] - 1, {i, n}] + 1* steps. (See also page 1145.)



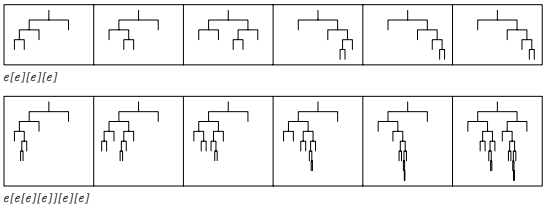
▪ **Other rules.** If only a single variable appears in the rule, then typically only nested behavior can be generated—though in an example like *e[x_][_] → e[x[e[e]][e]]* it can be quite complex. The left-hand side of each rule can consist of any expression; *e[e[x_]][y_]* and *e[e][x_[y_]]* are two possibilities. However, at least with small initial conditions it seems easier to achieve complex behavior with rules based on *e[x_][y_]*. Note that rules with no explicit *e*'s on the left-hand side always give trees with regular nested structures; *x_[y_] → x[y][x[y]]* (or *x_ → x[x]* in *Mathematica*), for example, yields balanced binary trees.

▪ **Long halting times.** Symbolic systems with rules of the form *e[x_][y_] → Nest[x, y, r]* always evolve to fixed points—though with initial conditions of size *n* this can take of order *Nest[r^# &, 0, n]* steps (see above). In general there will be symbolic systems where the number of steps to evolve to a fixed point grows arbitrarily rapidly with *n* (see page 1145), and indeed I suspect that there are even systems with quite simple rules where proving that a fixed point is always reached in a finite number of steps is beyond, for example, the axiom system for arithmetic (see page 1163).

▪ **Trees.** The rules given on pages 103 and 104 correspond to the transformations on trees shown below.



The first few steps in evolution from two initial conditions of the system on page 103 correspond to the sequences of trees below.



■ **Order dependence.** The operation *expr /. lhs → rhs* in *Mathematica* has the effect of scanning the functional representation of *expr* from left to right, and applying rules whenever possible while avoiding overlaps. (Standard evaluation in *Mathematica* is equivalent to *expr //. rules* and uses the same ordering, while *Map* uses a different order.) One can have a rule be applied only once using

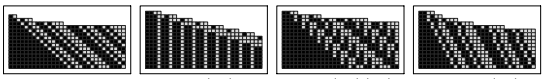
```
Module[{i = 1}, expr /. lhs :→ rhs /; i++ == 1]
```

Many symbolic systems (including the one on page 103) have the so-called Church-Rosser property (see page 1036) which implies that if a fixed point is reached in the evolution of the system, this fixed point will be the same regardless of the order in which rules are applied.

■ **History.** Symbolic systems of the general type I discuss here seem to have first arisen in 1920 in the work of Moses Schönfinkel on what became known as combinators. As discussed on page 1121 Schönfinkel introduced certain specific rules that he suggested could be used to build up functions defined in logic. Beginning in the 1930s there were a variety of theoretical studies of how logic and mathematics could be set up with combinators, notably by Haskell Curry. For the most part, however, only Schönfinkel's specific rules were ever used, and only rather specific forms of behavior were investigated. In the 1970s and 1980s there was interest in using combinators as a basis for compilation of functional programming languages, but only fairly specific situations of immediate practical relevance were considered. (Combinators have also been used as logic recreations, notably by Raymond Smullyan.)

Constructs like combinators appear to have almost never been studied in mainstream pure mathematics. Most likely the reason is that building up functions on the basis of the structure of symbolic expressions has never seemed to have much obvious correspondence to the traditional mathematical view of functions as mappings. And in fact even in mathematical logic, combinators have usually not been considered mainstream. Most likely the reason is that ever since the work of Bertrand Russell in the early 1900s it has generally been assumed that it is desirable to distinguish a hierarchy of different types of functions and objects—analogous to the different types of data supported in most programming languages. But combinators are set up not to have any restrictions associated with types. And it turns out that among programming languages *Mathematica* is almost unique in also having this same feature. And from experience with *Mathematica* it is now clear that having a symbolic system which—like combinators—has no built-in notion of types allows great generality and flexibility. (One can always set up the analog of types by having rules only for expressions whose heads have particular structures.)

■ **Operator systems.** One can generalize symbolic systems by having rules that define transformations for any *Mathematica* pattern. Often these can be thought of as one-way versions of axioms for operator systems (see page 1172), but applied only once per step (as */.* does), rather than in all possible ways (as in a multiway system)—so that the evolution is just given by *NestList[# /. rule &, init, t]*. The rule $x\_ \to x \circ x$ then for example generates a balanced binary tree. The pictures below show the patterns of opening and closing parentheses obtained from operator system evolution rules in a few cases.



■ **Network analogs.** The state of a symbolic system can always be viewed as corresponding to a tree. If a more general network is allowed then rules based on analogs of network substitution systems from page 508 can be used. (One can also construct an infinite tree from a general network by following all its possible paths, as on page 277, but in most cases there will be no simple way to apply symbolic system rules to such a tree.)

How the Discoveries in This Chapter Were Made

■ **Page 109 · Repeatability and numerical analysis.** The discrete nature of the systems that I consider in most of this book makes it almost inevitable that computer experiments on them will be perfectly repeatable. But if, as in the past, one tries to do computer experiments on continuous mathematical systems, then the situation can be different. For in such cases one must inevitably make discrete approximations for the underlying representation of numbers and for the operations that one performs on them. And in many practical situations, one relies for these approximations on "machine arithmetic"—which can differ from one computer system to another.

■ **Page 109 · Studying simple systems.** Over the years, I have watched with disappointment the continuing failure of most scientists and mathematicians to grasp the idea of doing computer experiments on the simplest possible systems. Those with physical science backgrounds tend to add features to their systems in an attempt to produce some kind of presumed realism. And those with mathematical backgrounds tend to add features to make their systems fit in with complicated and abstract ideas—often related to continuity—that exist in modern mathematics. The result of all this has been that remarkably few truly meaningful computer experiments have ended up ever being done.

▪ **Page 111 · The relevance of theorems.** Following traditional mathematical thinking, one might imagine that the best way to be certain about what could possibly happen in some particular system would be to prove a theorem about it. But in my experience, proofs tend to be subject to many of the same kinds of problems as computer experiments: it is easy to end up making implicit assumptions that can be violated by circumstances one cannot foresee. And indeed, by now I have come to trust the correctness of conclusions based on simple systematic computer experiments much more than I trust all but the simplest proofs.

▪ **Attitudes of mathematicians.** Mathematicians often seem to feel that computer experimentation is somehow less precise than their standard mathematical methods. It is true that in studying questions related to continuous mathematics, imprecise numerical approximations have often been made when computers are used (see above). But discrete or symbolic computations can be absolutely precise. And in a sense presenting a particular object found by experiment (such as a cellular automaton whose evolution shows some particular property) can be viewed as a constructive existence proof for such an object. In doing mathematics there is often the idea that proofs should explain the result they prove—and one might not think this could be achieved if one just presents an object with certain properties. But being able to look in detail at how such an object works will in many cases provide a much better understanding than a standard abstract mathematical proof. And inevitably it is much easier to find new results by the experimental approach than by the traditional approach based on proofs.

▪ **History of experimental mathematics.** The general idea of finding mathematical results by doing computational experiments has a distinguished, if not widely discussed, history. The method was extensively used, for example, by Carl Friedrich Gauss in the 1800s in his studies of number theory, and presumably by Srinivasa Ramanujan in the early 1900s in coming up with many algebraic identities. The Gibbs phenomenon in Fourier analysis was noticed in 1898 on a mechanical computer constructed by Albert Michelson. Solitons were rediscovered in experiments done around 1954 on an early electronic computer by Enrico Fermi and collaborators. (They had been seen in physical systems by John Scott Russell in 1834, but had not been widely investigated.) The chaos phenomenon was noted in a computer experiment by Edward Lorenz in 1962 (see page 971). Universal behavior in iterated maps (see page 921) was discovered by Mitchell Feigenbaum in 1975 by looking at examples from an electronic calculator. Many aspects of fractals were found by Benoit Mandelbrot in the 1970s using computer graphics. In the 1960s and 1970s a variety of algebraic identities were found using computer algebra, notably by William Gosper. (Starting in the mid-1970s I routinely did computer algebra experiments to find formulas in theoretical physics—though I did not mention this when presenting the formulas.) The idea that as a matter of principle there should be truths in mathematics that can only be reached by some form of inductive reasoning—like in natural science—was discussed by Kurt Gödel in the 1940s and by Gregory Chaitin in the 1970s. But it received little attention. With the release of *Mathematica* in 1988, mathematical experiments began to emerge as a standard element of practical mathematical pedagogy, and gradually also as an approach to be tried in at least some types of mathematical research, especially ones close to number theory. But even now, unlike essentially all other branches of science, mainstream mathematics continues to be entirely dominated by theoretical rather than experimental methods. And even when experiments are done, their purpose is essentially always just to provide another way to look at traditional questions in traditional mathematical systems. What I do in this book—and started in the early 1980s—is, however, rather different: I use computer experiments to look at questions and systems that can be viewed as having a mathematical character, yet have never in the past been considered in any way by traditional mathematics.

▪ **Page 113 · Practicalities.** The investigations described in this chapter were done using *Mathematica*, mostly in 1992. For larger searches, I sometimes created optimized C programs that were controlled via *MathLink* from within *Mathematica*—though with the versions of *Mathematica* that exist today this would now be unnecessary. For my very largest searches, I used *Mathematica* to dispatch programs to a large number of different computers on a network, then had the computers send me email whenever they found interesting results. (See also page 854.)