EXCERPTED FROM

STEPHEN WOLFRAM

A NEW KIND OF SCIENCE

NOTES FOR CHAPTER 3:

The World of Simple Programs

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.

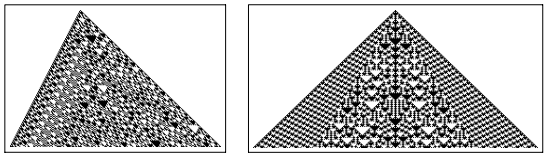
■ **Page 59 · Algebraic forms.** The rules here can be expressed in algebraic terms (see page 869) as follows:

- 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]*

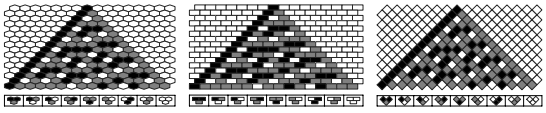
■ **Rule 45.** The center column of the pattern appears for practical purposes random, just as in rule 30. The left edge of the pattern moves 1 cell every 2 steps; the boundary between repetition and randomness moves on average 0.17 cells per step.

■ **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.

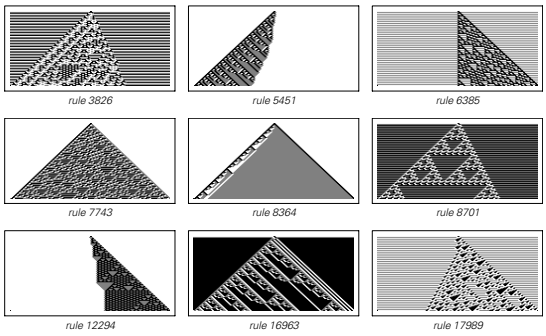
■ **Alternating colors.** The pictures below show rules 45 and 73 with the colors of cells on alternate steps reversed.



■ **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.)

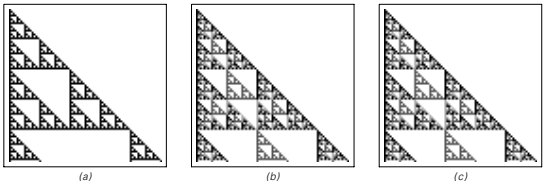
▪ **Rules based on algebraic systems.** If the values of cells are taken to be elements of some finite algebraic system, then one can set up a cellular automaton with rule

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

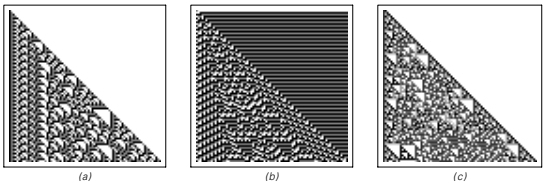
where f is the analog of multiplication for the system (see also page 1094). The pattern obtained after t steps is then given by

```
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

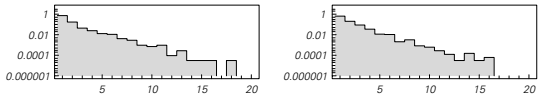
```
{{1, 1, 1} → {{0, 0, 0}, -1}, {1, 1, 0} → {{1, 0, 1}, -1},
 {1, 0, 1} → {{1, 1, 1}, 1}, {1, 0, 0} → {{1, 0, 0}, 1},
 {0, 1, 1} → {{0, 0, 0}, 1}, {0, 1, 0} → {{0, 1, 1}, -1},
 {0, 0, 1} → {{1, 0, 1}, 1}, {0, 0, 0} → {{1, 1, 1}, 1}}
```

and *MAStep* must be rewritten as

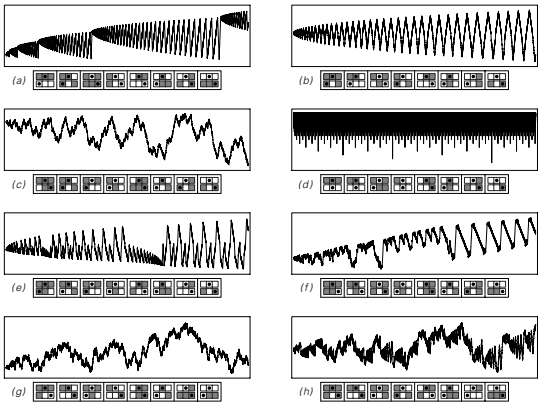
```
MAStep[rule_, {list_List, n_Integer}] /; 1 < n < Length[list] :=
  Apply[{Join[Take[list, {1, n - 2}], #1, Take[list, {n + 2, -1}]],
    n + #2} &, Replace[Take[list, {n - 1, n + 1}], rule]]
```

■ **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).

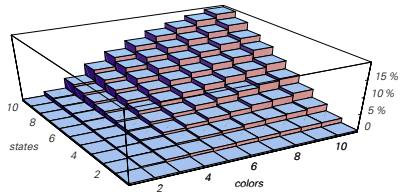
■ **Numbering scheme.** One can number Turing machines and get their rules using

```
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}]]
```

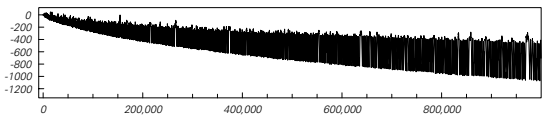
The examples on page 79 have numbers 3024, 982, 925, 1971, 2506 and 1953.

■ **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.)

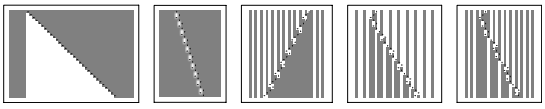


■ **Page 81 · Head motion.** The picture below shows the motion of the head for the first million steps. After about 20,000 steps, the width of the pattern produced grows at a rate close to $\sqrt{t}$.



■ **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

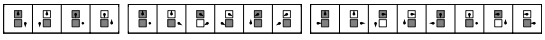
such as rule 110. What can happen is that with certain specific repetitive backgrounds, the head can move in a simple repetitive way, as shown in the pictures below for the Turing machine from page 81.



■ **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.

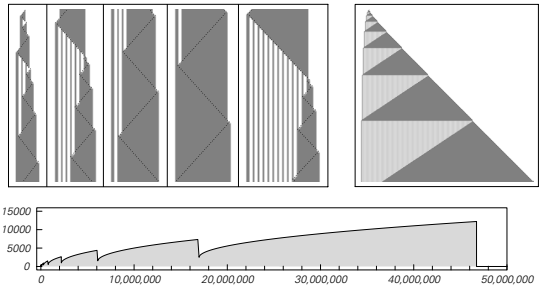
Since Turing's time, Turing machines have been extensively used as abstract models in theoretical computer science. But in almost no cases has the explicit behavior of simple Turing machines been considered. In the early 1960s, however, Marvin Minsky and others did work on finding the simplest Turing 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]
```

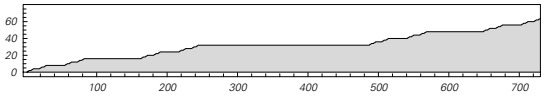
The sequence occurs many times in this book; it can for example be derived from a column of values in the rule 150 cellular automaton pattern discussed on page 885.

(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}
```

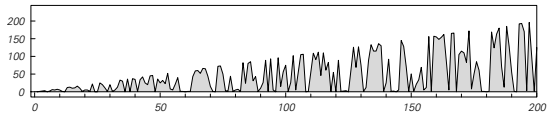
Fibonacci numbers appear to have first arisen in perhaps 200 BC in work by Pingala on enumerating possible patterns of

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

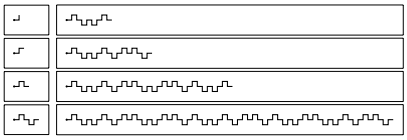
To treat examples such as case (c) where elements can subdivide into blocks of several different lengths one must generalize the notion of digit sequences. In base k a number is constructed from a digit sequence *a[r], …, a[1], a[0]* (with $0 \le a[i] < k$) according to *Sum[a[i] k^i, {i, 0, r}]*. But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

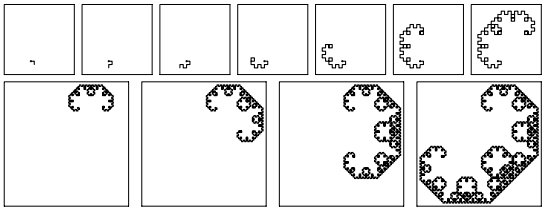
■ **Connections with square roots.** Substitution systems such as (c) above are related to projections of lines with quadratic irrational slopes, as discussed on page 904.

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

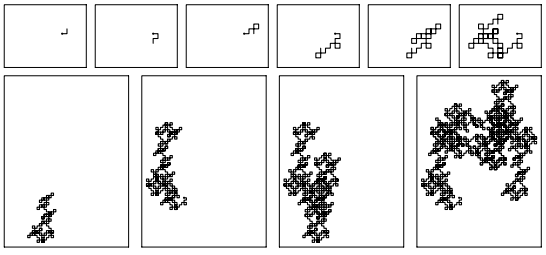
■ **Representation by paths.** An alternative to representing substitution systems by 1D sequences of black and white squares is to use 2D paths consisting of sequences of left and right turns. The paths obtained at successive steps for rule (b) above are shown below.



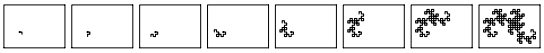
The pictures below show paths obtained with the rule *{1 → {1}, 0 → {0, 0, 1}}*, starting from *{0}*. Note the similarity to the 2D system shown on page 190.



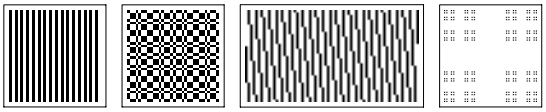
When the paths do not cross themselves, nested structure is evident. But in a case like the rule *{1 → {0, 0, 1}, 0 → {1, 0}}* starting with *{1}*, the presence of many crossings tends to hide such regularity, as in the pictures below.



■ **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.

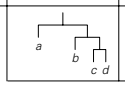
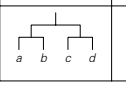
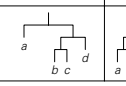
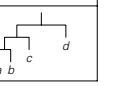
Symbolic Systems

■ **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*:

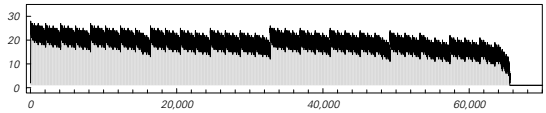
```
c[s_, 1] = s; c[s_, n_] := Flatten[
  Table[Outer[#1[#2] &, c[s, n - m], c[s, m]], {m, n - 1}]]
```

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}]]]
```

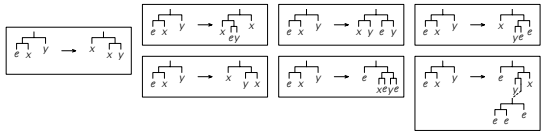
This quantity alternates between value *1* at position 2^j and value *j* at position $2^j - j + 1$. It reaches a fixed point as soon as the depth reaches 0. For initial conditions of size *n*, this occurs after at most *Sum[Nest[2^# &, 0, i] - 1, {i, n}] + 1* steps. (See also page 1145.)



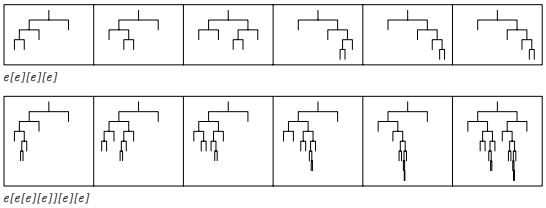
▪ **Other rules.** If only a single variable appears in the rule, then typically only nested behavior can be generated—though in an example like *e[x_][_] → e[x[e[e]][e]]* it can be quite complex. The left-hand side of each rule can consist of any expression; *e[e[x_]][y_]* and *e[e][x_[y_]]* are two 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

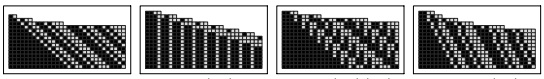
```
Module[{i = 1}, expr /. lhs :→ rhs /; i++ == 1]
```

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.)