The Theory of Computation

Bernard M.E. Moret

Addison-Wesley-Longman, 1998

Solution Manual
Chapter 4, Text Problems

Exercise 4.1

The diagonalization in the halting argument of Figure 4.1 comes from running procedure P on arguments x and x. That is, we have, in spirit, tabulated all possible programs (well, almost: see below) and all possible inputs to form a semi-infinite table; then we run procedure P on the diagonal in this table—looking at the xth program run on the xth input.

The assumption that we are trying to disprove here is that P exists—and, in particular, that it runs on all possible programs and all possible inputs. What the argument shows is that P cannot even be fully defined on the diagonal, much less on the whole table—that is, that there cannot exist a procedure P that, for each x, decides whether or not program x run on input x halts.

We do this by sweeping down the diagonal and creating a new program Q on which P must fail to yield an answer. To place it in the perspective of Cantor's approach (creating something that should be in the enumeration but cannot be without causing a contradiction), we can look at a somewhat different table: the table of all programs x such that P correctly decides whether or not program x halts on input x. If P is what we claim it to be, then every single program must appear in the table; but we construct a particular program Q that cannot be in the table—P cannot decide whether Q halts when run on its own code without causing a contradiction. Q is thus the object that should be in the enumeration because of what the enumeration claims to be, but that cannot be in the enumeration without causing a contradiction.

Exercise 4.2

1. In order to decrement an unsigned number, we first need to move the head of the machine to the least significant bit of the number, at the right end of the string. We then start moving leftward; if we encounter

a 0, we change it into a 1 and loop; if we encounter a 1, we change it into a 0 and (almost) stop; we should not encounter a blank, since this would mean that we were attempting to decrement a value of 0. When we encountered a 1 and changed it into a 0, we should take care that it was not the leading 1, since we would then leave a number with a leading 0; we thus simply check the square to the left: if it is a blank, then we replace our 0 by a blank and stop with the head to the right, otherwise we enter a loop to move the head back to the beginning of the number.

2. We can do this in one pass by keeping track of carries in the finite-state control (or we can use the solution of (3) below and add the number to itself twice). In multiplying by 3, the carry value can vary between 0 and 2. So our machine moves its head to the least significant bit of the input, then starts moving left, in a state that corresponds to a carry of 0. Then, in general, we have the following possibilities:
   - If the carry is 0 and the scanned bit is 0, we write a 0 and keep the carry to 0; if the carry is 0 and the scanned bit is 1, we write a 1 and change the carry to 1.
   - If the carry is 1 and the scanned bit is 0, we write a 1 and change the carry to 0; if the carry is 1 and the scanned bit is 1, we write a 0 and change the carry to 2.
   - If the carry is 2 and the scanned bit is 0, we write a 0 and change the carry to 1; if the carry is 2 and the scanned bit is 1, we write a 1 and keep the carry to 2.

   Finally, if the scanned bit is blank, we simply write any nonzero carry leftward (i.e., we write a 1 if the carry is 1 and we write a 0, then a 1 to the left of it, if the carry is 2).

3. The numbers to be added are separated by some symbol, say a blank; for convenience, we assume that our alphabet includes a fourth symbol, say #. The basic idea is to read the same bit position in each number, add any outstanding carry, and write the new bit of the answer and update the carry, all moving from the least significant bit to the most significant bit—basically the standard longhand addition procedure taught to all schoolchildren. Since we are using binary numbers, the carry value is either 0 or 1; we use the finite-state control to "remember" the values of the carry and of the two scanned bits. To keep track of the current bit position, we simply overwrite the scanned squares with #, the fourth symbol. Since the answer is produced left to right, we write it to the left of the first input string, thereby avoiding the need to keep shifting the output string.

Our machine will have to do a lot of head motions, so it has two main blocks of states. One block entered after writing the new bit of the answer, simply move the head right until the least non-scanned bit of the second number is encountered—it goes through any 0s and 1s (the answer), one blank, any 0s and 1s (unscanned remainder of first number), zero or more #s, a blank, any 0s and 1s, and stops at a blank or a #, moving right by one. This block exists in two parallel tracks of states, one track for a carry of 0 and one track for a carry of 1. The other block takes it from there and moves left; it reads the current bit (which could be blank if all the second number has been scanned), replaces it by # if it is not a blank, moves left over any other 0s and 1s, one blank, and any #s, scans the next bit (again, could be blank), replaces it by # if not blank, moves left past any 0s and 1s, past one blank, and past any 0s and 1s, and writes the new bit—which is determined by the state, because our machine had several parallel state tracks for this block, corresponding to carry value (0 or 1), bit of second number (0, 1, or blank), and bit of first number (0, 1, or blank). If both bit values are blank, the carry is handled and the machine enters a final block, which will clean up the tape (removing the marked up input strings); otherwise, a new bit is written, the carry updated, and the machine moves right again.

Exercise 4.3

A 2-tape Turing machine can recognize the language of palindromes in linear time with three passes as follows. In the first pass, it copies the input string to its second tape; at the end of this pass, both tapes have the same string on it and both heads are at the end of the input string on both tapes. In the second pass, it moves its first head back to the beginning of the string, leaving its second head immobile; at the end of this pass, the head on the first tape is at the beginning of the input string and the head on the second tape is at the end of the input string. In the third pass, the machine scans both tapes, forward on the first tape and backward on the second tape, moving the two heads together, and verifying that the two symbols read are always equal to each other. Since each pass takes linear time, the entire procedure takes linear time.

On the other hand, a 1-tape Turing machine appears unable to carry out the same task in better than quadratic time: once it has scanned one symbol of the input string, it needs either to fetch the matching symbol (which, for at least one quarter of the symbols, is at least $n/2$ tape squares away, so that fetching just these $n/4$ matching symbols will takes at least $n^2/8$ moves, i.e., quadratic time) or to store the symbol just scanned in a place from which the matching symbol will be easily accessible (which appears to entail exactly the same problem: just moving past the end of the input string to copy the scanned symbol takes linear time per symbol and thus quadratic time overall).

Exercise 4.4

We will not give the details, because the construction is very similar to the decrement and increment instructions. A register transfer can cause an increase or a decrease (or no change) in the length of the string that encodes the contents of the target register. Thus we will need the left- and right-shift loops developed for the increment and decrement blocks. The easiest way to implement a transfer otherwise is to blank out the target string (replace its characters with some special symbol #), then start copying one bit at a time, encoding in the finite-state control the value of the bit to be copied. Copying is easiest if we have two special symbols, call them z and o to represent 0s and 1s that have been copied. Once the copy is complete (and each bit copy could cause a shift to expand room), we may have to run a shift to reduce the room (if the target string was larger before the copy), then replace the zs and os by 0s and 1s. The time taken is proportional to the square of the input size, since each bit copy could force a nearly complete traversal of the entire tape.

Exercise 4.5

This is again similar to the increment and decrement routines. We use the register transfer routine of Exercise 4.4 and the addition routine of Exercise 4.2, part (3). The catch here is in organizing the pieces of the operation. First, it is easiest to use the strategy of Exercise 4.2, part (3), and construct the result to the left of the entire input string; we can then think of it as a register, use the routine of Exercise 4.4 to transfer it to its rightful destination, and come back to erase it and leave the head at the beginning of the tape; not only does this approach simplify the addition, it also automatically handles properly the case where the target register is the same as one of the operands. Secondly, we need to handle the case of the addition calling for the tow operands to be the same—in the same register; again, we can provide a general solution by copying the second argument to the left of the input, thinking of it as a register; then the sum is computed to the left of *that* register; after transferring the sum to the correct register, we then erase both the sum and the copy of the second operand. Thus all we need is register transfer and addition; both take time proportional to the square of the space used.

Exercise 4.6

We need to show that a (deterministic) Turing machine that has three choices for its head movement (stay, move left, or move right) can be simulated by a (deterministic) Turing machine that has only two choices. The simulation is really very simple: we use extra control states and move the head left, then back to the right to simulate the absence of move. Thus, if the augmented Turning machine has a transition of the type

$$\delta(q_i, a) = (q_j, b, -)$$

where the dash denotes that the head stays put, our simulation uses the two transitions

$$\begin{cases} \delta(q_i, a) = ([q_i, a], b, L) \\ \delta([q_i, a], c) = (q_j, c, R) \quad \forall c \in \Sigma \end{cases}$$

while transitions that do move the head are the same in both machines. The notation $[q_i, a]$ denotes a new state for each pair q_i and a such that the augmented Turing machine makes a transition with stationary head when in state q_i reading input a. Our simulation runs in at most twice the number of steps of the original Turing machine, with exactly the same tape contents are all steps; if the Turing machine we are simulating uses an alphabet of k characters and has q control states, then our simulating machine has up to $(k+1)q$ control states.

Exercise 4.7

An easy solution to this problem is to use a brute-force approach. We simulate an arbitrary Turing machine; at the beginning, we place a special symbol (not part of the machine's alphabet) on the square to the left of the input; the simulation then simply runs the simulated machine with only one safeguard: whenever the special symbol is read (with the simulated machine expecting a blank), our simulating machine enters a block of states that shifts the entire tape to the right of the special symbol by one tape square to the right, thereby creating one new tape square to the immediate right of the special symbol. This new tape square is filled with a blank and the simulation continues from there. The right-shift block is similar to that used in the decrement routine in the text.

The simulation thus takes the same amount of space on the tape as the original machine did, but it may take a lot more time: if the original machine ran in $f(n)$ time and $g(n)$ space, the simulation also runs in $g(n)$ space, but takes $O(f(n) \cdot g(n))$ time.

Exercise 4.8

The equivalence has to be understood in somewhat loose terms, since our Turing machines compute functions, but our finite automata only decide membership in languages.o

In abstract terms, though, the answer is quite clear: our restricted TM *is* a finite-state machine! If we examine what defines the state of the computation carried out by such a machine, we see that it is the input string, the contents of the current tape square (which could be different), and the current control state. Whatever is to the left of the head is irrelevant, since the machine can never look at it again; what is to the right of the head is simply the rest of the input, just as in a finite-state machine. So the only apparent difference is that our restricted Turing machine could sit on the same square and alter it several times before moving on. But we have seen that a finite automaton can have ε transitions during which it does not consume an input symbol; and the number of transitions that can be taken by our Turing machine before moving to the next input symbol is a constant—it is bounded by the product of the number of control states by the number of tape symbols, so that the finite automaton can encode all of them in its finite-state control.

Thus, in the loose understanding mentioned above, the two models are indeed equivalent.

Exercise 4.9

The basic idea is quite simple: a write-once machine will simply copy its entire tape over to a new block of fresh tape whenever it needs to simulate a normal machine that alters a tape square. Of course, since the machine writes just once, it will never be able to clean up or reuse old blocks of tape, so that it will use much more tape than the original machine.

It remains to verify that we can copy a block of tape with just one write; our normal copy routine marks copied squares, but that constitutes a *second* writing of the square. So we can do it with a machine that writes each square at most twice. Can we do it with just one write? A bit of thought reveals that we can with a simple trick, but only if we assume that our machine need not write every square that it scans—it is allowed to leave the scanned square unchanged by not writing on it. This is not our standard model, but is needed for our purposes. Our new machine then uses only one out of two tape squares for data used directly in the computation ; the others are used for makers in the copying process. When copying to a fresh block of tape, the machine leaves every other square blank (they are blank to start with) in the new block; when copying that block, it writes a mark in the blank square corresponding to (adjacent to) the square being copied. Fresh blocks of tape are thus separated by at least two blanks or by a special symbol.

When simulating a conventional Turing machine that runs in $f(n)$ time and $g(n)$ space, our new machines takes $O(f(n) \cdot g(n))$ time and space.

Exercise 4.10

The transition function of such a machine is almost identical to that of normal Turing machine: the only difference is that the head movement must be specified as one of four directions rather than just as one of two. We do need some conventions for input and output, however; we assume that each takes the form of a string stored "horizontally" on the tape (on the same row, with the head stationed on the first (leftmost) square used by the string.

To simulate such a machine on a conventional Turing machine. we need to develop an encoding for the 2-dimensional tape. There are many ways to do this: we can use pairing, we can use prime power encoding, and many others. All will require that our simulating Turing machine, before simulating a single step of the 2-dimensional one, do some decoding of the encoded 2-dimensional tape. The one step is then simulated, then the results re-encoded and rewritten on the tape. Thus the simulation will run much more slowly than the original machine.

For instance, if we use a pairing function, to simulate a step where the 2D machine moves up on its grid, we need to compute the index on our 1D tape of the new pair $\langle x, y+1 \rangle$, then move right until this index is found, which may entail traversing a large fraction of the 1D tape several times, especially since we have to use some of that tape to carry out the calculations and since seeking an index is not that simple in a conventional Turing machine (easiest may be pushing a marker forward, coming back every time to decrement the index, an extremely costly way of doing the work).

Prime power encoding may be better: we encode a column of the 2D tape as the number $2^{i_1}3^{i_2}5^{i_3}\cdots$, where i_1, i_2, i_3 are the values in the range of rows from the first to the last nonzero entry in the column. Our 1D tape is then a list of these codes. Yet everything must be decoded and later re-encoded, which is quite time-consuming, especially as it must be done in some blank area of the tape, so that the value must first be copied beyond one end of the currently used tape.

Another, simpler solution is to appeal to Theorem 4.1 and simulate our 2-dimensional TM with a k-tape TM rather than just a single-tape machine. One approach is to consider at all times the smallest rectangle on the 3-dimensional tape that contains every square examined so far by the 2D machine. Consider this rectangle as an array of 1-dimensional rows; we just use one tape of a 2-tape TM to store this rectangle, row by row, while the second tape stores the current dimensions of the rectangle and the third tape is used for auxiliary computations. A move of the 2D machine that stays within the current rectangle is either a horizontal move, which is simply a move left or right by one square on the first tape of our 2-tape machine, or a vertical move, which corresponds to a m moves left or right on the first tape, where m is the horizontal size of the rectangle (stored on the second tape) and the counting is done on the third tape. If the move of the 2D machine takes it outside the current rectangle, we extend the rectangle. For a vertical move, this consists simply of adding a row of blanks either in front of or in back of the storage on the first tape, plus incrementing the number of rows on the second tape. For a horizontal move, every row must get a blank added in front or back of it, which we accomplish by our standard shifting mechanism, one at a time (again, we use the third tape for counting off rows), then we increase the number of columns on the second tape.

Exercise 4.11

We can use the addition to keep doubling the value stored in a register and register transfer to copy the result in a collection of registers. Consider the small RAM program below.

```
    R1 <- 4
    R2 <- 1
    R3 <- n      /* other registers are all 0 */
10: R2 <- R2 + R2
    R[R1] <- R2  /* R[R1] is the register indexed by R1 */
    inc R1
    dec R3
    jnz R3 10    /* jump to 10 if R3 is nonzero */
```

This program will store values 2, 4, 8, ... , 2^n in registers 4 through $n + 4$. When started, the program uses $\log n$ bits (in register 3); at completion, however, registers 4 through $n + 4$ use respectively 1, 2, 3, ... , n bits, for a total of $\Theta(n^2)$ bits. The running time is the constant time used in each loop times the number of loops and so is $\Theta(n)$. Thus this small program uses space at a rate that grows quadratically as a function of time.

Exercise 4.12

We need to charge more than unit time for some of the operations. The culprit, from what we have seen, is not the addition (which can only increase storage by one bit at a time, at least in asymptotic terms) nor the register transfer (which cannot increase storage at all, again in asymptotic terms), but the indirect addressing. In fact, if indirect addressing can be used with addition, we have enough power to recreate the situation of Exercise 4.11: it is enough to have the loop carry out the instructions

```
R[R1] <- R[R2] + R[R2]
inc R2
inc R1
```

With R1 exactly one larger than R2.

Thus we need a charging policy that penalizes the use of indirect addressing heavily enough to bring space use to a level commensurate with time use. The easiest way to do this is specify that the cost of indirect addressing is directly proportional to the space used in the register that is addressed (not the index register itself, but the register addressed through the index register). It seems somewhat of an arbitrary decision; more importantly, it is not always intuitive: for instance, why should it cost constant time to increment a named register, but logarithmic time to do the same on an indirectly addressed register?

Another solution is to impose a policy similar to that used for Turing machines: every bit costs, so we make the cost of every instruction be proportional to the length of its arguments and results (if any). This, of course, returns us to our earlier discussion and the apparently exaggerated cost of basic arithmetic in such a model.

Exercise 4.13

We prove the slightly stronger result that a 2-register machine can simulate a k-register machine.

Note that any general RAM has some fixed number of registers (all those named in the program). We encode all of the registers of the general RAM into the first register of our 2-register machine by using prime encoding: if the general RAM uses k registers and their current values are r_i, $1 \leq i \leq k$, the first register or our machine will contain the value $\prod_{i=1}^{k} p_i^{r_i}$, where p_i is the ith prime. The fundamental theorem of arithmetic tells us that such an encoding is one-to-one. The second register of our machine will be used for temporaries. To simulate an increment of register i, our machine simply multiplies its first register by p_i; it can do that by first setting the second register to zero, then entering a loop that carries out p_i increments of the second register (something that can be done by an *ad hoc* routine, since p_i is a known constant) for one decrement of the first, until the first register is zero, then entering another loop that copies the result from the second back into the first register. To simulate a decrement of register i, our machine divides its first register by p_i, which proceeds in the same spirit as the multiplication, except that we carry out p_i decrements of the first register for each increment of the second. (Note that the last routine can be modified to test for the presence of a remainder, indicating that the register was 0 before the decrement and should remain unchanged; we can then invert the operation completely to recover the original value.) To test register i for zero, we carry out the division by p_i, test whether the remainder is non-zero (indicating there was no factor of p_i in the encoding and hence that we had $r_i = 0$), then remultiply by p_i in order to maintain the original value. Jumps themselves are of course no problem. Thus we have a valid simulation of any given k-register RAM on a 2-register RAM, so that 2-register RAMs (and thus also 3-register RAMs) are universal models of computation.

Exercise 4.14

The model does not really differ at all from the conventional RAM; we just assume that all values are coded in unary. That being the case, there is no question at all that incrementation and decrementation take constant time, since they consist of adding one symbol to or removing one symbol from a string of identical symbols. Simulation of a binary RAM by a unary RAM is trivial, since we have the same set of instructions; if we lack addition, we can implement it with the program of Figure 4.5.

Exercise 4.15

Because our solution to Exercise 4.13 reduced the number of registers down to 2 rather than just down to 3, this result follows immediately from Exercises 4.13 (as enhance in our solution) and 4.14.

Exercise 4.16

Note that an off-line TM (also equipped with three tapes, except that the work tape is “normal”, i.e., can store any alphabet character in any quantity) can do anything that a standard TM can do: just copy the input to the work tape, simulate the standard TM on the work tape, then copy the work tape to the output tape. Thus we can simulate an off-line TM on our model; of course, we can also simulate a RAM or partial recursive functions! In our model, the work tape can be partitioned into four regions by the three pebbles, each region being composed of indistinguishable (empty) cells. The first and last regions (the two ends of the tape) are useless, but the two middle regions are bounded and can be used to count: we can store (in unary, by counting the number of empty cells between two pebbles) two natural integers, call them x and y. By shifting the pebbles left or right, we can increment or decrement either number and we can easily test either one for equality to zero. On the other hand, we can only add destructively (by zeroing one of the operands and replacing the other by the sum) and cannot multiply at all, so that we do not really have enough power to encode and decode. It would take at least 4 pebbles to enable us to do full arithmetic, including multiplication and division by a constant, while retaining values of operands—all of which is needed to encode and decode values. So we have the following alternatives: (i) show that the universal model of computation from which we start can be further restricted into a (still universal) model that is easier to simulate by our pebble machine; or (ii) define a somewhat more powerful version of our pebble machine, show that it can simulate an arbitrary TM/RAM, then show that our 3-pebble machine can simulate the more powerful pebble machine.

First, let us consider a two-register, unary-coded RAM (known to be universal from Exercise 4.15). Simulating such a RAM on our machine is easy: we use our two intervals to store (in unary) the contents of the two registers. The RAM can do one of three things: (i) increment a register and move to the next instruction—we increment the corresponding interval and change state; (ii) decrement its register and move to the next instruction—we decrement the corresponding interval and change state; and (iii) conditional jump to another instruction—we test an interval for equality to zero by checking whether its two limiting pebbles are adjacent, then change state as needed. The computation is initialized by "copying" the input (which is in unary in register 1) to the work tape (i.e., placing two pebbles to delimit a piece of work tape of the same size as the input); when the simulation is complete, the pebble machine "copies" its first interval to the output tape, that is, it writes one symbol on the output tape for each square between the first two pebbles. Thus we have a complete simulation of a two-register unary-coded RAM on our pebble machine.

Let us consider the alternative of using a slightly more powerful pebble TM. We shall allow ourselves 5 pebbles rather than just 3. With 5 pebbles, we have 4 intervals—4 numbers that can be incremented, decremented, and tested for 0. Now, if we encode the TM's tape in three parts (as we did when simulating a TM on a RAM: left-hand tape, symbol under the head, right-hand tape), we can store the three parts and still have one "scratch" counter. This allows us to manipulate the three parts as needed. Assuming that the original machine can store blanks and *a*s on its tape, we need division by 2, multiplication by 2, and parity test in order to simulate head moves and test for the character under the head. We can divide by two one of our "registers" by initializing the last register to zero (4th and 5th pebble adjacent), then decrementing our register by two for each increment of the scratch register by one; we then copy the result from the scratch register back in to the correct register by the same technique in reverse (increment the register for each decrement of the scratch register); multiplying by two is the same, but we increment twice for each decrement, then copy the result. Parity test is even easier: traverse the register "two squares at a time." These operations allow us to simulate the effect of an arbitrary transition of the original TM on our 5-pebble machine. It remains now to show how a 3-pebble machine can simulate a 5-pebble machine—or how to cut down from 4 registers to just 2 registers. Note that we need only simulate the specific 5-pebble TM produced above on our 3-pebble TM. In particular, the 5-pebble TM produced above always has all 5 pebbles on the work tape—it just shifts them left or right. In our simulation on the 3-pebble machine, we will keep the contents of the four registers of the 5-pebble machine encoded as a single number (in unary, naturally) in our first interval; the second will be used for temporaries. We proceed exactly as for the reduction of an arbitrary RAM to a two-register RAM described in Exercise 4.15: if the four registers of our 5-pebble machine contain values i, j, k, and l, we will encode them in our first interval as the single number $2^i \cdot 3^j \cdot 5^k \cdot 7^l$. Incrementation, decrementation, and test for zero are all as described earlier for the RAM simulation.

Exercise 4.17

Intuitively, the finite automaton can use ε-transitions on the input in order to work exclusively with the queue; indeed, we can think of the machine first copying the input into the queue and thereafter using only ε-transitions on the input. The queue itself is essentially like a circular tape—the finite automaton can examine any element of the queue by repeatedly removing the head and reinserting it at the tail, thereby simply rotating the queue by one position. The head of the queue is equivalent to the character under the head of the Turing machine—what the machine can check right now. Moving the head of the Turing machine in one direction is just the rotation we mentioned; to simulate a move in the other direction, we need to rotate the queue completely around, which means also identifying when the rotation is complete. To that end, we simply insert a marker (a symbol not otherwise used) into the queue and rotate until the marker appears at the head of the queue, at which time we remove it and stop the rotation. Moving past a blank at either end of the Turing machine tape corresponds simply to inserting an item in the queue. Thus our queue-based finite automaton can simulate a Turing machine and is therefore universal.

Exercise 4.18

Intuitively, we shall use one stack to simulate the portion of the tape to the left of the head and the other stack for the portion of the head to the right of the head, with the character under the head arbitrarily assigned to one stack. Moving the head of the Turing machine then corresponds to popping an item off one stack and pushing it (or some other value, as changed by the Turing machine's transition) onto the other stack. As in Exercise 4.17, we can think of the finite automaton first copying its input into one stack, then from that point on using only ε-transitions on the input and dealing exclusively with the two stacks. Moving past a blank one either end of the Turing machine tape corresponds to increasing storage in one stack (which contains everything) and having a single item in the other stack. We leave small details unspecified, but it is clear that our 2-stack machine can simulate a Turing machine and is therefore universal.

Exercise 4.19

The measures are fairly straightforward, since both machines have a completely uniform transition mechanism and a clear notion of storage. Each transition will be one step and count for one unit of time. The space used will be the largest queue size (or the largest combined stack size) used at any stage during the computation. Because our two machines can increase their use of storage by at most one cell at each step, they respect the standard relationship between space and time.

Exercise 4.20

1. Any rule of the type $x \longrightarrow cx + d$ can be applied indefinitely; if c and d are positive and we have $d \geq b$, then any rule of the type $x + b \longrightarrow cx + d$ can be applied indefinitely. A lot of specific rules also apply forever, such as the rule $2x - 1 \longrightarrow 2x + 1$, which produces successive odd numbers.
2. Any system with the single rule $ax \longrightarrow x$ eventually stops, having removed all factors of a. If a and b are relatively prime, then any system with the single rule $ax \longrightarrow bx$ eventually stops, having replaced every factor of a by a factor of b. The system of two rules

$$\begin{cases} 2x + 1 & \longrightarrow 2x \\ 2x & \longrightarrow x \end{cases}$$

   always stops, producing 1 from any nonzero starting number.
3. What we need to do is replace every factor of 3 by a factor of 2; we just saw how to do that with the simple system composed of the one rule $3x \longrightarrow 2x$.

4. This is much trickier than (3) above. Some experience writing production systems helps here. What we want to do is produce another n factors for 2 for each factor of 3, removing that factor when the n factors of 2 have been produced. However, we have to keep in mind that the rules are always scanned for applicability in the same order, so that, for instance, a rule that removes one factor of 3 without precaution, as in $3x \longrightarrow x$, will in fact get used over and over again, until all factors of 3 are gone. Removing only one factor takes some special care; we need to guard each rule by requiring that other factors be present, factors that we add in as needed and remove later. In particular, we want a single factor added in as a marker that we will remove when we remove the factor of 3; in spirit, we add the marking factor at the beginning of a loop and remove it at the end, enabling us to decrement the "loop counter" (the number of factors of 3) in a controlled fashion.

   In our solution we use 13 as the marking factor, 5 and 7 as stand-ins for 2, and 11 as a temporary replacement factor for 3. We accumulate all extra factors of 2 as factors of 5, so that we keep a count of n factors of 2 through all the loops—the 5s never get handled at all. Once we fall out of the last loop, we change all 5s back into 2s. The shift from 3s to 7s and back again into 3s and the shift of the n factors of 2 into 11s and back again into 2s are both used to sequence rules—we want to avoid going back to the first rules until the end of the transformation caused by one loop, so we change the factors and create additional rules that operate on the new factors. When we are ready to return to the beginning of the loop for a new iteration, we restore the original factors.

   A trick is used to introduce the single copy of the marking factor 13; obviously, we cannot just have a rule $x \longrightarrow 13x$, since that rule would be applicable forever. Instead, we set up 12 rules, each of the form $13x + i \longrightarrow 169x + 13i$, where I is between 1 and 12; such rules can only apply if our number is *not* a multiple of 13 and only one such rule can apply. (Our actual rule is a bit more complex, because it also requires that we have present at least one factor each of 2 and of two of 3, so all numbers are in fact multiplied by 18—that is, we have 12 rules of the form $234x + 18i \longrightarrow 3042x + 234i$.) The various guards account for the large numbers in the rules, but one can see what the rule actually does by dividing throughout by the guards—as indicated in the comments.

```
A.  234x     ->  4095x        guarded by 13, which is conserved;
                              also guarded by 3*3, also conserved;
```

```
                              replaces each 2 by a 5 and a 7 (2x -> 35 x)
                              the 7 is just a temporary stand-in for 2;
                              the 5s form one more copy of the 2s
B. 1365x     ->  5005x        guarded by 13, which is conserved;
                              also guarded by 5*7, also conserved;
                              replaces each 3 by an 11 (3x -> 11x)
C.   91x     ->    26x        guarded by 13, which is conserved;
                              replaces each 7 by a 2 (7x -> 2x)
D.  143x     ->      x        guarded by 13, which is removed;
                              removes one factor of 3 (in its 11 guise)
E.   11x     ->     3x        unguarded; converts all factors of 11
                              back into factors of 3

F.                            the block below adds a single factor of 13
                              (but only to a number that does not have one)
                              it is guarded by 2*3*3, which is conserved
    234x+18  ->  3042x+234    (13x+1 -> 169x+13)
    234x+36  ->  3042x+468    (13x+2 -> 169x+26)
    234x+54  ->  3042x+702    (13x+3 -> 169x+39)
    234x+72  ->  3042x+936    (13x+4 -> 169x+52)
    234x+90  ->  3042x+1170   (13x+5 -> 169x+65)
    234x+108 ->  3042x+1404   (13x+6 -> 169x+78)
    234x+126 ->  3042x+1638   (13x+7 -> 169x+91)
    234x+144 ->  3042x+1872   (13x+8 -> 169x+104)
    234x+162 ->  3042x+2106   (13x+9 -> 169x+117)
    234x+180 ->  3042x+2340   (13x+10-> 169x+130)
    234x+198 ->  3042x+2574   (13x+11-> 169x+143)
    234x+216 ->  3042x+2808   (13x+12-> 169x+156)

G.                            one gets here only when all but one fac-
tor of 3
                              have been handled and removed;
                              just replaces all factors of 5 by fac-
tors of 2
                              and removes the last 3, if any
      5x     ->      x
      3x     ->      x
```

The system works as follows—keep in mind that it always tries to apply rule A first, then rule B if rule A is not applicable, then rule C, etc. When started with a number of the form $2^n \cdot 3^m$, if we have $n \geq 1$ and $m \geq 2$, then rules A through D cannot initially apply, since they require a factor of 13; rule E does not apply (no factor of 11), but one of the rules in block F does apply, yielding the new number $2^n \cdot 3^m \cdot 13$. Now rule A applies, over and over, replacing every factor of 2 by a factor of $5 \cdot 7$, so that we obtain $3^m \cdot 5^n \cdot 7^n \cdot 13$, at which point rule A no longer applies. At this point, though, rule B applies, since we have a factor of $5 \cdot 7 \cdot 13$ (the guard) in our number; this rule applies over and over again, replacing every factor of 3 by a factor of 11, so that we obtain $5^n \cdot 7^n \cdot 11^m \cdot 13$. Now neither rule A nor rule B applies, but rule C applies and replaces a factor of 7 by a factor of 2; notice that this change alone does not make rule A applicable (it would need a factor of 9 as well), so that rule C applies over and over again and replaces every factor of 7 by a factor of 2; we thus now have $2^n \cdot 5^n \cdot 11^m \cdot 13$. (A full copy of the 2s has been made, represented by the 5s.) Now none of rules A, B, and C applies, but rule D applies; it removes the guard of 13 and also one factor of 11, yielding $2^n \cdot 5^n \cdot 11^{m-1}$. Because the guard of 13 has been removed, rules A through D now will not apply until the guard is re-established; so rule E applies over and over again and replaces every factor of 11 by a factor of 3, yielding $2^n \cdot 3^{m-1} \cdot 5^n$. We have completed the loop: the loop counter (the power of 3) has been decreased by 1 and a copy of the 2s has been made. Now rules A through E fail to apply, so we fall through to block F, which adds again a factor of 13, thereby enabling rule A, etc. Eventually, we get to an application of rule E yielding the number $2^n \cdot 3 \cdot 5^{n(m-1)}$, at which point block F also fails to apply, since it requires two factors of 3. We then fall through to rule G, which applies over and over and replaces every 5 by a 2, yielding $2^{mn} \cdot 3$ and then removes the last 3, finally yielding 2^{mn} and stopping, as desired! Also note that special cases, such as the numbers 1, 2, 3, and 6, are all properly handled: nothing at all applies to either 1 or 2, which are output unchanged (as should be) and only the very last rule applies to 3 and 6, which are changed into 1 and 2 (again as should be).

Exercise 4.21

What Exercise 4.20, part (4), demonstrated is that, when we can restrict the form of the numbers processed by a system of rules, we can control fairly easily the sequencing of the rules. Sequencing is of course crucial in simulating a RAM program or Turing machine with a Post system; the other crucial aspect is testing for the absence of something—but again, we showed how to do that in Exercise 4.20, part (4), with the block of rules that added a factor of 13 only to a number that did not have such a factor.

Say we want to simulate a RAM program. We shall encode everything with prime power encoding. That is, the initial state of the RAM (contents of registers) will be a number of the form $2^{i_1}3^{i_2}5^{i_3}\cdots$, where i_1 is the value of the first register, i_2 the value of the second register, etc. In addition, this value will be multiplied by a prime corresponding to the value of the program counter (we can number each line in the RAM program starting at 1)—since there are only a constant number of lines in any given program, this uses only a constant number of primes. Each instruction in the program will be encoded by a rule or a block of rules, guarded by the value of the program counter, so that it can only be applied when the program has reached that line. Thus rules corresponding to a RAM program instruction numbered, say 17, will in general be of the form $p_{17} \cdot (\ ) \longrightarrow p_{17} \cdot (\ )$, where p_{17} is the prime assigned to the 17th line of code in the RAM program. (Overall, then, we need at least one prime per line and one prime per register; we will also need some additional primes to play tricks of the type used in Exercise 4.20, part (4).) Our most elementary RAM has only three instructions: increment a register, decrement a register, and jump on zero to an address. Increment and decrement simply remove or add one factor and increase the program counter; for instance, the instruction at line 20

```
20:  inc R3
```

becomes the rule

$$p_{20} \cdot x \longrightarrow p_{21} \cdot 5 \cdot x$$

since 5 is the prime assigned to register 3. Decrement is just a bit trickier, because it appears we need a test for 0, since decrementing a 0 value leaves 0. However, one slick way of handling this problem is to encode a value of i in register, say 3, by the value 5^{i+1}, thereby giving ourselves an extra factor and making the test for the absence of a factor much simpler. The jump on zero is similar: we need a test for the absence of a factor; in both cases, the prime denoting the program counter is replaced by another, but which other depends on that test.

The details are somewhat long and not much different from what we did in Exercise 4.20, part (4), so we omit them here.

Solution Manual
Chapter 4, Additional Problems

Additional Problem 1

Additional Problem 2

Additional Problem 3