6. States and Transitions

6.1 Introduction

This chapter introduces the idea of states and transitions. It talks about representing transitions by rules, noting the difference between deterministic and non-deterministic systems. It indicates how programs can be thought of in terms of states and transitions, and how ordinary imperative programs can be transformed into functional ones. It also discusses Turing machines, and a variety of related ideas. Most of these ideas are tied together with the thread of "functional programming" already introduced. Then in the next chapter we will see how these ideas play into "object-oriented programming", which is at the other end of the spectrum.

Among the most pervasive notions in computing is that of "state". We define the *state* of a computation as a set of information sufficient to determine the future possible behaviors. In other words, the state tells us how the system can change. It also can tell us something about what has happened in the past, if we know it to have been started in a particular initial state. It doesn't usually tell exactly what has happened, but rather conveys some *abstraction* of what has happened.

A typical computational system starts with the initial state that embodies the input to the computation, and continues until termination, at which point the output can be extracted from the final state.

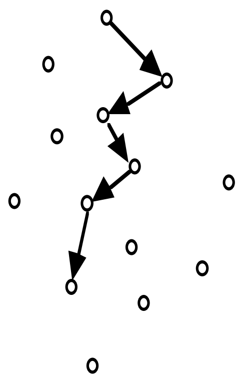


Figure 54: The progression of a system from state to state

In the context of a set of rewrite rules, the state determines the possible *end results* of the computation, if any. We can stop the computation at any point and resume it, so long as we are careful to record the state at the stopping point. This is of major importance, for example, in computer *operating systems,* which are sets of programs that control the usage of major resources, such as input-output devices, available on the computer. Operating systems frequently switch from one computation in progress to another, for purposes of best exploiting the available resources. The state of a program is saved in memory, enabling the system to restart the program later on. For example, if a program needs to wait for a critical resource to become available, the operating system will suspend that program and run another program meanwhile. The state concept is also useful when we wish to simulate a program's execution mentally or with pencil and paper. If we get tired, we can record the state and resume the activity later.

States in digital computation are a lot like the states we encounter in solving certain kinds of puzzles. Here the configuration of the puzzle completely determines the state. Many such puzzles can be solved using the techniques of graph searching, such as breadth-first search. These entail being able to detect whether we encountered the current state earlier, in which case we would not want to repeat the same set of moves, for that would lead to no progress.

Towers of Hanoi Example

In this famous puzzle, N discs of decreasing sizes are stacked on one of three spindles, the other two of which are initially empty. The problem is to move all N discs, one at a time, from the first spindle to the second, maintaining the constraint that at no time is a larger disc placed atop a small one.

The state in the case of this puzzle is the way that the disks are stacked on the spindles. The initial state for N = 4 would be as shown below.

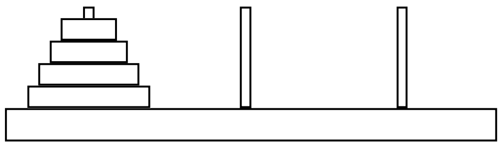


Figure 55: ***The Towers of Hanoi puzzle for N = 4, initial state.***

The desired final state is:

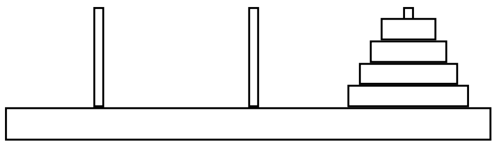


Figure 56: The Towers of Hanoi puzzle for N = 4, final state.

The following is an example of an illegal state, one that cannot occur during a solution, since a larger disk is atop a smaller one.

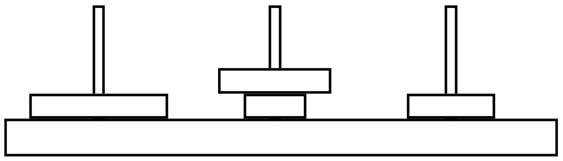


Figure 57: An illegal state in the Towers of Hanoi puzzle

To provide a tool for discussion of the solution to such puzzles, as well as for computational systems in general, we can exploit the mathematical concept of "binary relation" as described in the next section.

6.2 Transition Relations

To apply relations to the discussion of states, the set of pairs of states (s, s') such that one move will take the system from s to s' is called the **transition relation**. Typically we will represent a transition relation by =>. A single pair (s, s') such that s => s' is called a **transition**. We sometimes say that s' is a **successor** of s when there is a transition (s, s') and that s is a **predecessor** of s'.

Below we show some possible transitions between states for the Towers of Hanoi puzzle with only 2 disks. With this bird's-eye view of the states and transitions, it is possible to find a sequence of transitions that lead from any state to any other, and, in particular, to solve the puzzle. While many solutions are possible, there is a unique *shortest* solution represented by the path from top center to bottom left. Quite evidently, any deviation from this path adds more steps than are necessary.

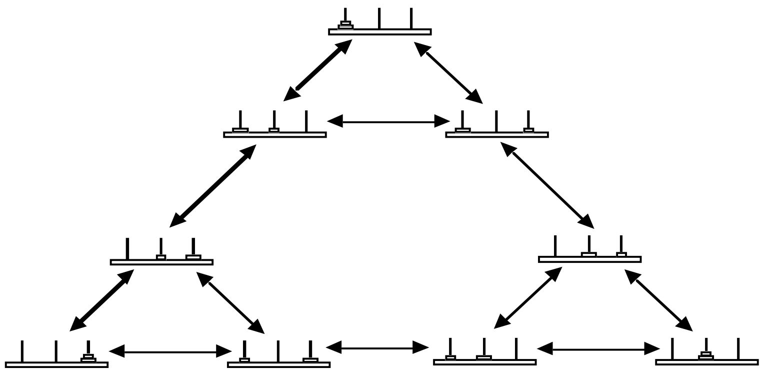


Figure 58: Bird's-eye view of possible state transitions in the Towers of Hanoi puzzle with 2 disks. The shortest solution is the sequence of transitions from the top state downward to the lower left.

Notice that in this particular puzzle, the transition relation happens to be *symmetric*, in that for each move there is an opposite move that returns the puzzle to the previous state. Accordingly, the state-transition graph can be represented as an undirected graph. Of course, this is not the case for puzzles in general. Many puzzles have one-way transition relations.

The scheme of constructing the state-transition diagram can be used to solve the puzzle for larger values of N as well. The only problem here is that we have a "combinatorial explosion" of states as we consider larger N. Fortunately, we can get an understanding of the state-transition structure of this particular puzzle without going to very large N. Examine the state-transition diagram above. Notice that there are three triangular patterns embedded within an overall triangular pattern. Consider the top three states. This triangular pattern shows the motions of the top disk with the bottom disk remaining fixed on the first spindle. Similarly, the lower-left triangle has the bottom disk fixed to the third spindle, and finally, the lower-right triangle has the bottom disk fixed to the second spindle. We could, therefore, have constructed this diagram as follows:

1. Diagram the states for the puzzle with a single disk. This diagram is just a triangle, with each vertex representing one of the three positions for the single disk.

2. For two disks, consider a similar triangle with the vertices representing the positions of the bottom disk. For each of these vertices, embed a single-disk triangle.

This construction is suggested by the following pattern:

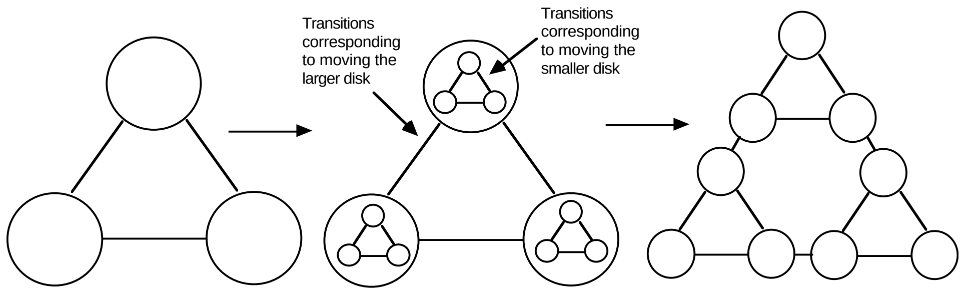

Transitions corresponding to moving the larger disk
Transitions corresponding to moving the smaller disk

Figure 59: Embedding pattern for constructing a 2-disk transition diagram from 1-disk transition diagrams

In this analysis, we are starting to think "recursively", an important skill we will continue to hone throughout this book. The same triangular pattern can be used to construct a diagram for N+1 disks from three N disk diagrams for any N. For example, the following diagram for 3 disks was obtained by taking three of the diagrams on the right-hand side above and placing them inside a 1-disk diagram.

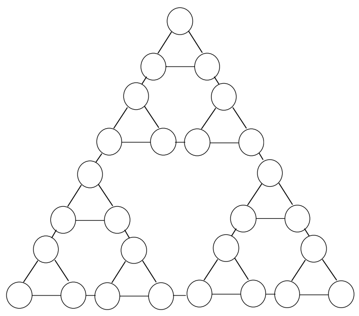


Figure 60: The state-transition diagram corresponding to the 3-disk structure

One thing this construction tells us is that every time we add a new disk, we triple the number of states that have to be considered. For an N-disk puzzle, there are thus 3^N states. While we are at it, we can make some other observations about these diagrams:

- The start and end points of the puzzle are always at the extreme vertices of the triangle.
- The shortest path from one of these extremities to another will always entail 2^N nodes, i.e. it takes 2^N - 1 moves to solve an N-disk puzzle.

Both of these assertions can be proved by induction. We could also continue this construction *ad infinitum* by repeatedly placing the 1-disk triangle inside the inner-most vertices of the diagram. In the limit, we would have a "self-similar system", sometimes called a *fractal*. The limiting case is self-similar because inside each of the outermost vertices we have contained an exact replica of the entire system. A related geometric construction is known as the *Sierpinski Gasket*.

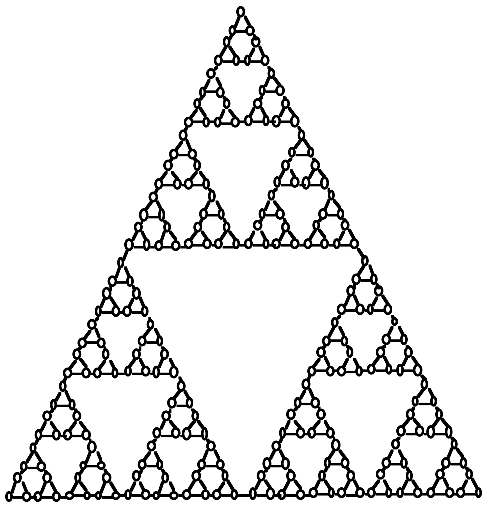


Figure 61: Approximation to a Sierpinski Gasket

Of course, we don't expect that all state-transition diagrams will generate comparable artwork.

Reachability Relations

From a transition relation ➪, we will have need for the accompanying **reachability relation** (also called the **reflexive transitive closure** of ➪), which will be represented ➡. The reachability relation ➡ means that we can get from one state to another by a series of zero or more intervening states. More precisely:

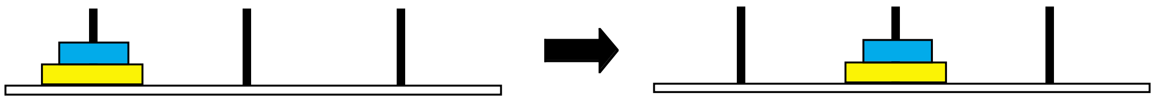
T_0 ➡ T_n

means that there are terms $T_1, T_2, \dots, T_{n-1}$ ($n \geq 0$) such that

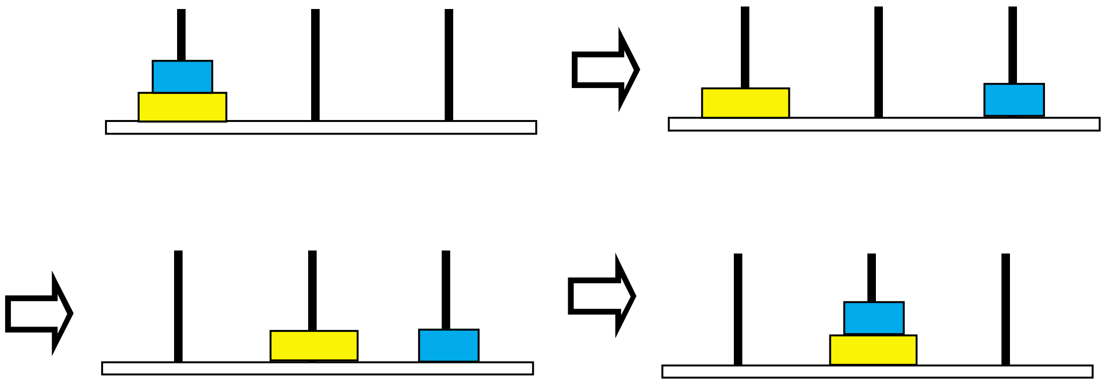
T_0 ➪ T_1 ➪ T_2 ➪ … ➪ T_{n-1} ➪ T_n

Example – Towers of Hanoi Reachability Relation

In the 2-disk Towers of Hanoi Puzzle, we can assert



based on the following series of individual transitions:



In later sections, we will have further occasion to use this notation in connection with computational systems.

6.3 Transition Rule Notation

The transitions for the Towers of Hanoi puzzle are determined by very explicit rules. The most succinct way we can state these is:

- A transition can involve moving only one disk
- A transition must result in a legal state.

These rules, then, characterize an infinite set of possible transitions (when we consider the puzzle for all values of N, the number of disks). In like fashion, a computational system will have transition rules. Unlike puzzles, computational rules will often be **deterministic**: that is, there is at most one successor to any state. A system, such as many puzzles, where for some state there are at least two possible successors, is called **non-deterministic**. Non-deterministic systems have some important technical uses outside of puzzles, as we shall see later. One of these uses is as *grammars*, ways of representing the syntax of programming languages.

A state-transition system is called

deterministic if every state has at most one successor.

non-deterministic if a state may more than one successor.

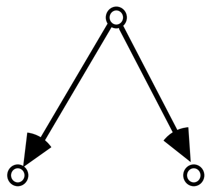


Figure 62: Scenario not occurring in a deterministic system

To codify the rules of the Towers of Hanoi puzzle, we can represent a state as a pattern

towers(L, M, N)

where L, M, and N represent the disks stacked on each tower. We will then be able to express rules for changing from one state to another in the form

towers(L, M, N) => towers(L', M', N').

where the primed towers are derived from the unprimed ones according to a particular pattern, or by

towers(L, M, N) => *Condition* ? towers(L', M', N').

in the case that the transition holds only if *Condition* on towers is satisfied. Note that these *look* a lot like rex rules. However, they end with a period rather than a semi-colon. This is because the rules are not actually in rex, but are implemented in a language called *Prolog*. A Prolog program is provided that can determine from such a set of rules whether the puzzle has any solutions, or a shortest solution. It does this by enumerating reachable states, and therefore will only work in the case that the number of such states is not too large.

To represent the disks on the towers themselves, we could number the disks 1, 2, 3, 4 and record the numbers on each of the three spindles as lists from top to bottom. Thus the initial state is

```
towers([1, 2, 3, 4],  [ ], [ ])
```

and the desired final state is

```
towers([ ], [ ], [1, 2, 3, 4])
```

(Note that this is not the state representation that we used in counting the number of states. There is no requirement that it be the same.) For the 2-disk puzzle, the state-transitions would be shown as follows, where we have omitted the outer towers(…) to keep the diagram from getting too cluttered:

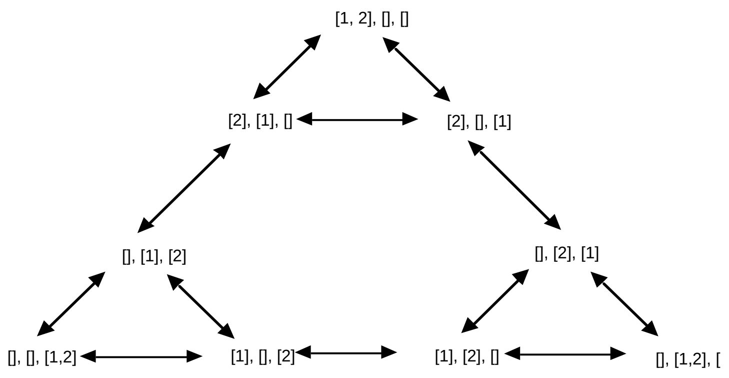


Figure 63: Coded state transitions for the Towers of Hanoi puzzle

For a fixed value of N, one could enumerate all of the states and transitions possible for an N-disk Towers of Hanoi problem. However, none of these enumerations expresses the general rules for transitions, because we don't yet have a way to describe manipulations on *arbitrary* lists. We now digress to describe a notation for presenting such rules. Then we will give a complete set of rules for the puzzle.

The transition rules are represented in one of the following forms:

S => S'.

or

S => C ? S'.

where S and S' are state forms and C is a *guard condition*. The condition states additional constraints governing when the use of the rule is possible. So the rules for moving a disk from the first spindle are:

```
towers([A | L], [], N) => towers(L, [A], N).                      % 12e

towers([A | L], M, []) => towers(L, M, [A]).                      % 13e

towers([A | L], [B | M], N) => A < B ? towers(L, [A, B | M], N). % 12n

towers([A | L], M, [B | N]) => A < B ? towers(L, M, [A, B | N]). % 13n
```

The characters on lines following `%` are comments. The first two rules govern movement of a disk from the first spindle to an empty spindle. The third and fourth rules govern movement of a disk from the first to a non-empty spindle, and require that the disk being moved is smaller than the disk already on top of that spindle. Here we are assuming that the disks are identified with numbers, so that `A < B` means that disk `A` is smaller than disk `B`. The naming of the rules is of the form *i j S*, where *i* and *j* are spindle numbers (1, 2, or 3) giving the number of the "from" and "to" spindles, and S is either `e` or `n`, designating whether the target spindle is empty or non-empty. For example, rule `12n` designates moves from spindle 1 to spindle 2, where spindle 2 is non-empty. The reason we have to separate the empty and non-empty cases is so that we can make the necessary comparison to determine whether a disk is being put atop a larger disk or not, in the case of a non-empty spindle.

There are eight more rules in two sets paralleling these, showing how disks can be moved from the second spindle and from the third spindle, respectively.

```
towers(N, [A | L], []) => towers(N, L, [A]).                      % 23e

towers([], [A | L], M) => towers([A], L, M).                      % 21e

towers(N, [A | L], [B | M]) => A < B ? towers(N, L, [A, B | M]). % 23n

towers([B | N], [A | L], M) => A < B ? towers([A, B | N], L, M). % 21n

towers([], N, [A | L]) => towers([A], N, L).                      % 31e

towers(M, [], [A | L]) => towers(M, [A], L).                      % 32e

towers([B | M], N, [A | L]) => A < B ? towers([A, B | M], N, L). % 31n

towers(M, [B | N], [A | L]) => A < B ? towers(M, [A, B | N], L). % 32n
```

There are thus twelve rules in all, six for each combination of ij to an empty spindle, and six to a non-empty spindle. Because there is no limit on the size of the lists, each transition rule represents an infinite set of transitions.

For example, in the previous figure, one series of state transitions (going downward and to the left) is the following:

```
towers([1, 2, 3, 4], [], [])
towers([2, 3, 4], [1], [])
towers([3, 4], [1], [2])
towers([3, 4], [], [1, 2])
towers([4], [3], [1, 2])
```

To see that the rules justify these transitions,

```
towers([1, 2, 3, 4], [], []) => towers([2, 3, 4], [1], [])  by 12e

towers([2, 3, 4], [1], []) => towers([3, 4], [1], [2])      by 23e

towers([3, 4], [1], [2]) => towers([3, 4], [], [1, 2])      by 23n

towers([3, 4], [], [1, 2]) => towers([4], [3], [1, 2])      by 12e
```

It should be emphasized that these rules by themselves do not *solve* the puzzle. They only articulate the legal transitions. However, these rules can be the input of a relatively simple program that does solve the puzzle without additional intellectual effort. Such a program will be further discussed in chapter *Computing Logically*. Later on in the current chapter, we will give a different set of rules for solving the puzzle directly, i.e. rules that *program* the solution.

Deterministic Solution of the Towers of Hanoi

This example makes use of recursive rules to solve the Towers of Hanoi puzzle. Let us try to discover a general method for solving this problem for N discs. We want to create a function that takes the number of discs N as an argument and returns a list of "instructions" for moving the discs. Each instruction will itself be a list, of the form:

```
["move disk ", D, " from ", A, " to ", B]
```

where D, A, and B are numbers of disks and spindles.

It is reasonable to try to get recursion to work for us. This would entail breaking an N disc problem down into lesser problems, e.g. N-1 disc problems. In order to move N discs from a spindle A to a spindle B, we might try the following:

> Move top disc from spindle A to spindle C.
>
> Move N-1 discs from spindle A to spindle B.
>
> Move top disc from spindle C to spindle B.

Unfortunately, this doesn't work. When the top disc has been moved to spindle C, it blocks the use of spindle C for subsequent moves from A, since in a legal state the top disc must be smaller than the other discs below it on A.

A different attempt would be to recursively move the top N-1 discs from A to C, move the bottom one disc to B, then move N-1 discs from C to B. This approach has the virtue that the bottom disc can be ignored while all of the other motion is taking place, since it must be larger than all of the other N-1 discs: no illegal states will be introduced.

This second attempt can be converted into a method, expressed recursively as follows:

```
To move N discs from a spindle A to a spindle B:

    If N = 0, there is nothing to do.

    If N > 0, then:

        move (N-1) discs from A to C, where C is the third spindle other
        than A and B;

        move one disc from A to B;

        move (N-1) discs from C to B.
```

> Since the first and third steps of the recursion can be done with the one disc in the second step unaffected, the desired constraint is maintained.

Obviously we are letting recursion work for us here, in fact with *two* recursive task calls to solve one task. We wish to give a set of rules that will solve this problem, in the sense that the ultimate result will be a list of pairs indicating single moves from one spindle to another. The spindles will be numbered 1, 2, 3. The discs will be numbered 1, 2, …, N to smallest to largest. We will represent a stack of discs to be moved as a list [d_0, d_1, …, d_N], smallest first.

We transcribe our rules into *rex* as follows, where `towers(L, A, B, C)` means move the stack of discs `L` from spindle `A` to `B`, with `C` as the other spindle (`A`, `B`, and `C` will vary from term to term).

```
towers([ ], A, B, C) => [ ];

towers([ D | L ], A, B, C ) =>
  append( towers(L, A, C, B), [move(D,A,B) | towers(L, C, B, A)] );
```

programmed rex solution rules for the Towers of Hanoi problem

Here `move(D, A, B)` rewrites to some term depending on how we wish to represent the move, e.g. we could use the rule

```
move(D, A, B ) => ["move disk ", D, " from ", A, " to ", B];
```

Since the exact form of move is not important, we shall not rewrite terms of the form `move(D, A, B)` further in the answers but just leave them as is. To complete the solution to our problem's specification, we need an interface function that takes the number of discs as an argument and calls *solve*. Given the number `N`, we need to create a stack of discs numbered [1, 2, …, N]. This can be accomplished by the function *range* given earlier. So the interface function, which we call tower, is

```
tower(N) => towers(range(1, N), a, b, c);
```

where a, b, and c are the names of the three spindles.

A different version of *solve* that eliminates the append use in favor of an accumulator argument (the last argument of `towers1` in this case) can be expressed as:

```
towers(L, A, B, C) => towers1(L, A, B, C, [ ]);

towers1( [ ], A, B, C, Moves) => Moves;

towers1( [D | L], A, B, C, Moves) =>
  towers1(L, A, C, B, [ move(D,A,B) | towers1(L, C, B, A, Moves)] );
```

programmed solution rules for the Towers of Hanoi problem using an accumulator

A Basic Counting Principle

In order to get an idea of the magnitude of certain problems such as those involving states, it is helpful to be able to count the sizes of sets without actually enumerating their members. One intuitive way to count is to describe a thought experiment that would, if actually conducted, enumerate the members and use it to determine the number of members, or at least an upper bound on the number of members.

Perhaps the simplest counting principle involves the notion of the **Cartesian product** of sets, designated by x. If A and B are two sets, then

$$A \times B$$

stands for the set of all *ordered pairs*, the first element drawn from A and the second drawn from B. Similarly, if N sets are mentioned, then

$$A_1 \times A_2 \times \ldots \times A_N$$

designates the set of all *ordered N-tuples*, the i^{th} component of which is drawn from A_i. For example, if A = {1, 2} and B = {a, b, c}, then A x B = {(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)}.

Let's use |S| to denote the size of S, i.e. the number of elements in S. Then the counting principle is:

$$| A_1 \times A_2 \times \ldots \times A_N | = | A_1| \ | A_2 | \ldots \ | A_N |$$

basic counting principle

where the juxtaposition on the right is ordinary numeric product. To describe this as a thought experiment, consider the case N = 2. Each element of $| A_1 \times A_2 |$ consists of a pair, the first element of which is drawn from A_1 and the second element of which is drawn from A_2. The experiment consists of enumerating all possible ways to construct such a pair. There are $| A_1 |$ ways to choose the first element in the pair. For each of those choices, there are $| A_2 |$ ways to choose the second element in the pair. Therefore we have a total of $| A_1| \ | A_2 |$ ways to choose pairs.

The extension of this argument to the general case is straightforward. We would use induction on N. For N = 1, there is nothing to prove. For N > 1, we can adopt the inductive hypothesis that

$$| A_1 \times A_2 \times \ldots \times A_{N-1} | = | A_1| \ | A_2| \ldots \ | A_{N-1}|$$

Now consider this as a single set and add on A_N. That is, $A_1 \times A_2 \times \ldots \times A_N$ has the same number of elements as

$$(A_1 \times A_2 \times \ldots \times A_{N-1}) \times A_N$$

(although these two sets are technically slightly different; why?). So perform the same argument that we did for N = 2 to conclude that

$$| A_1 \times A_2 \times \ldots \times A_{N-1} \times A_N | = | A_1| \, | A_2| \ldots | A_{N-1}| \, | A_N|$$

As a special case, consider the situation where each A_i is the same, say A. Then we have

$$\boxed{| A \times A \times \ldots \times A | = |A|^N}$$

where the superscript denotes raising |A| to the Nth power.

Example

The number of (legal) states in the N-disk Towers of Hanoi puzzle is 3^N. To see this, consider that each state is representable by an N-tuple over the set {1, 2, 3}. The ith element of the N-tuple is the tower on which the ith disk resides.

Exercises

The following are various puzzles used as test cases in computer science. For each puzzle, describe the legal states. Devise a linear notation for the states. Describe the possible state-transitions informally (you need not use the formal rule notation we presented, although it is worth trying to see if it will work. When necessary, just state guard conditions informally.). Sketch a portion of the state-transition diagrams.

Note that in some cases (e.g. peg solitaire) it is possible to give a single set of rules that fit many sizes of puzzle. You should do this whenever possible. In others (e.g. the N^2-1 puzzles), formulating a general rule for many sizes of puzzle appears more difficult; the rules seem to need to be tailored to the size of the puzzle. This is not to say it can't be done. In such cases, it is acceptable to use a small instance of the puzzle, e.g. the 8-puzzle.

1 •• The linear peg solitaire(N) puzzle (a different puzzle for each N). This puzzle is played on a board of 2N-1 holes in which N pegs of each of two colors are placed, as shown below. The objective is to interchange all the pegs of the two colors. A legal transition involves either (a) moving a peg forward into an adjacent hole in the direction of its intended home; (b) jumping a peg forward over another peg and into an adjacent hole.

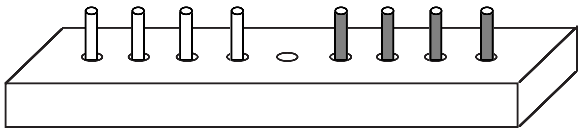


Figure 64: Peg solitaire: white pegs move or jump a peg only to the right. Colored pegs move or jump a peg only to the left.

The following shows another possible state of the puzzle:

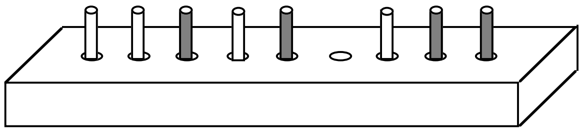


Figure 65: Peg solitaire after a few moves and jumps

The desired final state of the puzzle is as shown below:

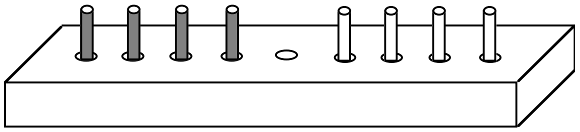


Figure 66: Desired final state of peg solitaire

[Hint: Model the sequence of pegs on either side of the hole as separate lists, constructing the left-hand list in the right-to-left peg direction.]

2 ••• Water jugs puzzles. There are several jugs of known capacities, each an integer number of units. One of the jugs is filled with water. The problem is to get a specified number of units of water into one of the jugs. A common example is that the jugs have capacities of 3, 5, and 8 liters, the 8 liter jug is the full one, and the objective is to obtain exactly 4 liters.

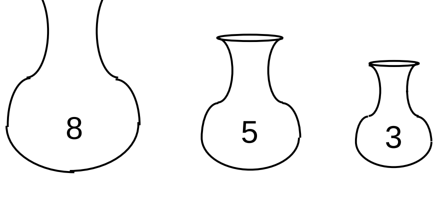


Figure 67: A water jug puzzle. The 8 liter jug is full. What transitions give us 4 liters in one jug?

3 ••• Give a rex program for generating a solution to the Towers of Hanoi puzzle given an arbitrary (legal) starting state.

4 ••• Suppose we wished to construct the state-diagram for a puzzle similar to the Towers of Hanoi, but with four towers instead of three. Describe the structure of the state-transition diagram.

5 ••• The N^2-1 puzzles (e.g. the 15 puzzle, the 8 puzzle, the 3 puzzle, etc.) This puzzle is played on an N-by-N board. There are N^2-1 tiles, numbered 1 through N^2-1. This leaves one space for a tile blank. The objective is to get from a given state of the puzzle to the state in which the tiles are all in order, by sliding blocks vertically or horizontally (not diagonally) onto the adjacent blank space.

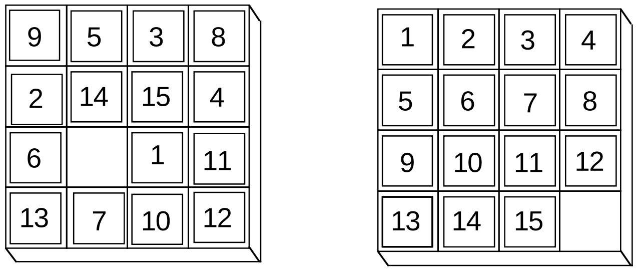


Figure 68: Two states of the 15 puzzle

Draw the complete state transition diagram for the 3 puzzle (played on a 2-by-2 board).

6 ••• The previous exercise introduced the N^2-1 puzzles. Express the 8-puzzle in the rule notation.

7 •• A puzzle (or its underlying transition relation) is called *strongly connected* if for any two states T_0 and T_1, it happens to be the case that $T_0 \Rightarrow T_1$. Is the 3-puzzle (described in the preceding problem, with N = 2) strongly connected?

8 ••• Give an argument that shows that the Towers of Hanoi puzzle, for any fixed number of disks, is strongly connected. (Hint: Use induction on the number of disks.)

9 ••• The Chinese rings puzzle. N rings are wired in a particular way through an elongated loop. Each loop is permanently attached to a wire, the opposite end of which is permanently connected to a bar. On its way from the bar to the ring, the wire passes through the ring on the immediate right. The objective is to remove the loop from all of the wires and their rings, so that the loop can be completely

separated from the rest of the puzzle. Give a set of rules that produce a list of moves for solving the puzzle deterministically.

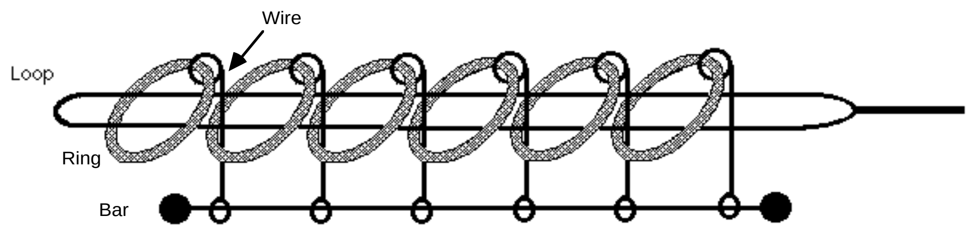


Figure 69: The Chinese rings with N = 6, before the loop has been removed from any ring

[Hint: Pursue an analysis similar to the deterministic solution to the Towers of Hanoi. This can be done purely based on the following analysis of the constraints of the puzzle: (a) The loop can be removed from the *leftmost* wire, or put back through it, in any state. (b) The loop can be removed from a non-leftmost wire w, or put back through w, provided that the wire w' to the immediate left of w goes through the loop and none of the wires to the left of w' go through the loop. So, for example, the first few states in the solution of the puzzle might be (numbering the wires 1, 2, 3, … left-to-right):

Take loop off wire 1.
Take loop off wire 3.
Put loop on wire 1.
Take loop off wire 2.
Take loop off wire 1.
Take loop off wire 5.
…

10 ••• What is the minimum number of moves required to solve the 6-ring puzzle?

11 ••• How many states are there in an N^2-1 puzzle (e.g. 15-puzzle) as described earlier), for a given N?

12 ••• How many states are there in an N-ring Chinese ring puzzle?

6.4 Rules for Assignment-Based Programs

In this section, we demonstrate how programs constructed in terms of sequences of assignment statements, loops, etc. can be recast as rule sets in rex, even though rex itself does not have an assignment construct. This demonstration is important in understanding the relationship between iteration and recursion.

Assignment-based programs use assignment to (i.e. changing the value of) various **program variables** to do their work. A program variable is an object, the value of which can change through an appropriate command. The form of an assignment will be assumed to be (using Java syntax)

Variable = *Expression*

meaning that the value of *Expression* is computed, then the value of *Variable* becomes that computed value. To give an equivalent rule-based program, we can use rule argument variables to play the role of assignable variables.

The *state* of an assignment-based program is a mapping from the names of variables to their values, in combination with the value of the instruction pointer (or "program counter") indicating the next instruction to be executed. To convert an assignment-based program to a rule-based one, associate a function name with each point before an instruction. Associate the arguments of the function with the program variables. We will then show how to derive appropriate rules.

Factorial Flowchart Example

Consider the following flowchart for a program that computes N-factorial. There are 6 places at which the instruction pointer can be, labeled 0..5. There are three program variables: `F`, `K`, and `N`. `N` is unchanging and represents the value for which the factorial is to be computed. `F` is supposed to contain the value of `N` factorial upon exit. `K` is used as an internal counter.

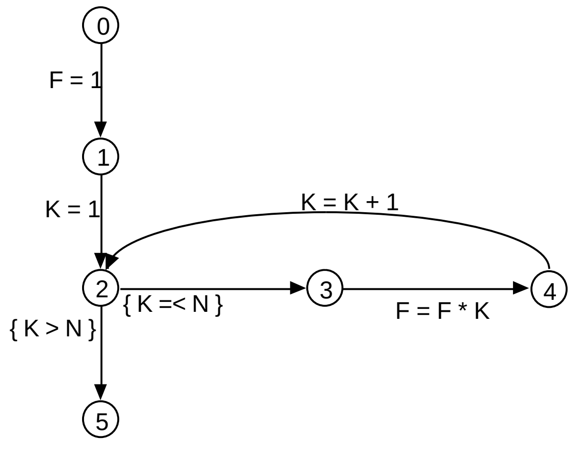


Figure 70: Flowchart for a factorial program.
The labels in braces {…} represent test conditions.
Labels of the form Var = Expression represent assignments.

We will first present the rules corresponding to the flowchart program, then indicate how they were derived. We let `fac` be the function being computed. We introduce symbols

f_0, …, f_5 as auxiliary functions. We use `arb` to indicate an *arbitrary* value (used for example to denote the value of an uninitialized assignable variable). The rules corresponding to the factorial flowchart are:

```
fac(N) => f0(N, arb, arb);

f0(N, K, F) => f1(N, K, 1);     // corresponds to assignment F = 1;

f1(N, K, F) => f2(N, 1, F);     // corresponds to assignment K = 1;

f2(N, K, F) => (K <= N) ? f3(N, K, F);

f2(N, K, F) => (K > N) ? f5(N, K, F);

f3(N, K, F) => f4(N, K, F*K);  // corresponds to assignment F = F*K;

f4(N, K, F) => f2(N, K+1, F);  // corresponds to assignment K = K+1;

f5(N, K, F) => F;
```

Let us verify that these rules give the correct computation of, say, `fac(4)`:

```
fac(4)             =>
f0(4, arb, arb)    =>
f1(4, arb, 1)      =>
f2(4, 1, 1)        =>
f3(4, 1, 1)        =>
f4(4, 1, 1)        =>
f2(4, 2, 1)        =>
f3(4, 2, 1)        =>
f4(4, 2, 2)        =>
f2(4, 3, 2)        =>
f3(4, 3, 2)        =>
f4(4, 3, 6)        =>
f2(4, 4, 6)        =>
f3(4, 4, 6)        =>
f4(4, 4, 24)       =>
f2(4, 5, 24)       =>
f5(4, 5, 24)       =>
24
```

Notice that the successive rewritten expressions correspond precisely to the *states* of the factorial flowchart in execution. A term of the form

$$f_i(N\text{-}value, K\text{-}value, F\text{-}value)$$

corresponds to the state in which the instruction pointer is at node i in the flowchart, and the values of variables *N*, *K*, and *F* are *N-value*, *K-value*, and *F-value* respectively.

McCarthy's Transformation Principle

Now we articulate the method for deriving rules from the flowchart. This method was first presented by Professor John McCarthy, so we call it "McCarthy's Transformation Principle". We have already indicated that there is one function name for each point in the flowchart.

> *McCarthy's Transformation Principle*:
>
> Every assignment-based program can be represented as a set of mutually-recursive functions without using assignment.

First we give the transformation rule when two points are connected by an assignment:

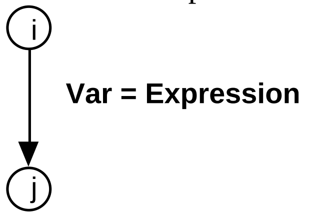


Figure 71: Assignment in the flowchart model

Corresponding to this connection we introduce a rule that rewrites f_i in terms of f_j. We identify the argument of f_i corresponding to the variable *Var* on the *lhs*. On the *rhs*, we replace that variable with *Expression*, to give the rule:

f_i(…, Var, …) => f_j(…, Expression, …);

The justification for this rule is as follows: The computation from point i will take the same course as the computation from point j would have taken with the value of *Var* changed to be the value of *Expression*.

Next we give the transformation rule when two points are connected by a test condition:

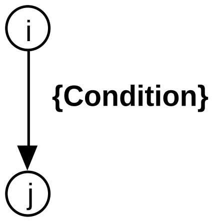


Figure 72: Conditional test in the flowchart model

In this case, the rule will be guarded, of the form

$f_i(\ldots) \Rightarrow$ Condition ? $f_j(\ldots)$;

The justification for this rule is: The computation from point i, in the case *Condition* is true, will take the same course as the computation from j. In most flowcharts, conditions come paired with their negations. Such a rule would usually be accompanied by a rule for the complimentary condition, recalling that ! means *not*

$f_i(\ldots) \Rightarrow$!Condition ? $f_k(\ldots)$;

where f_k will usually correspond to a different point than f_j. Alternatively, we could use the sequential listing convention to represent the negation implicitly:

$f_i(\ldots) \Rightarrow$ Condition ? $f_j(\ldots)$;
$f_i(\ldots) \Rightarrow f_k(\ldots)$;

Finally, to package the set of rules for external use, we introduce a rule that expresses the overall function in terms of the function at the entry point and one that expresses the function at the exit point to give the overall result. In the factorial example, these were:

```
fac(N) => f0(N, arb, arb);

f5(N, K, F) => F;
```

The rule-based model allows some other constructions not found in some flowcharts:

Parallel Assignment

With a single rule, we can represent the result of assigning to more than one variable at a time. Consider a rule such as

$f_i(\ldots, \text{Var}_1, \ldots \text{Var}_2, \ldots) \Rightarrow f_j(\ldots, \text{Expression}_1, \ldots \text{Expression}_2, \ldots)$;

The corresponding assignment statement would evaluate both Expression_1 and Expression_2, then assign the respective values to Var_1 and Var_2. This is not generally equivalent to two assignments in sequence, since each expression could entail use of the other variable. Some languages include a parallel assignment construction to represent this concept. For two variables a parallel assignment might appear as

$(\text{Var}_1, \text{Var}_2) = (\text{Expression}_1, \text{Expression}_2)$

Combined Condition and Assignment

With a single rule, we can represent both a conditional test and an assignment. Consider a rule such as

f_i(…, Var, …) => Condition ? f_j(…, Expression, …);

Here the computation of f_i is expressed in terms of f_j with a substituted value of *Expression* for *Var* only in case that *Condition* holds This is depicted in the flowchart as:

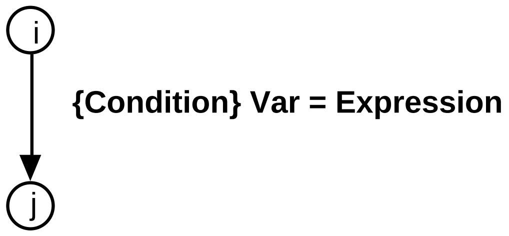


Figure 73: Combined condition and assignment in the flowchart model

List Reverse Example

One way to approach the problem of reversing a list by a recursive rule is to first give an assignment-based program for doing this, then use the McCarthy transformation. The idea of our assignment-based program is to loop repeatedly, in each iteration decomposing the remaining list to be reversed, moving the first symbol of that list to the result, then repeating on the remainder of the first list. This could be described by the following program, which is intentionally more "verbose" than necessary. An assignment of the form [A | M] = L decomposes the list L (assumed to be non-empty) into a first element A and a list of the rest of the elements M. This is pseudo-code, not a specific language.

```
L = list to be reversed;

R = [ ];
while( L != [ ] )
  {
  [A | M] = L;    // decompose L into first element A and rest M
  L = M;
  R = [A | R];    // compose list R
  }
assert R is reverse of original list
```

For example, here is a trace of the program through successive loop bodies:

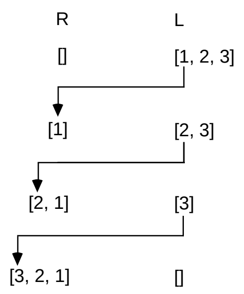


Figure 74: Flow of data in reversing a list

The flowchart version of this program could be expressed as follows:

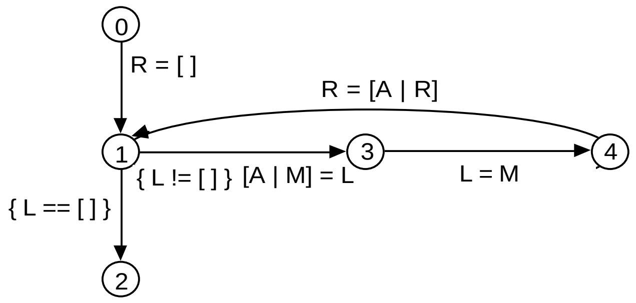


Figure 75: Flowchart for reversing a list

According to the McCarthy Transformation Principle, we can convert this to rewrite rules using the four program variables `L`, `R`, `M`, `A`.

```
f0(L) => f1(L, [ ], arb, arb);

f1(L, R, M, A) =>  L == [ ] ? f2(L, R, M, A);

f1(L, R, _, _) => L != [ ] ? L = [A | M], f3(L, R, M, A);

f3(L, R, M, A) => f4(M, R, M, A);

f4(L, R, M, A) => f1(L, [A | R], M, A);

f2(L, R, M, A) => R;
```

In the second rule for f_1,

```
f1(L, R, _, _) => L != [ ] ? L = [A | M], f3(L, R, M, A);
```

we use _ arguments to avoid confusion of the third and fourth arguments with the element `A` and list `M` decomposed from list `L`. We could simplify this further by using matching on the first argument to:

```
f1([A | M], R, _, _) => f3([A | M], R, M, A);
```

In the following section we will show why this set of rules is equivalent to:

```
f0(L) => f1(L, [ ]);

f1([ ], R) => R;

f1([A | L], R) => f1(L, [A | R]);
```

We can see that f1 is the same function as the two-argument `reverse` function described in *Low-Level Functional Programming*.

Program Compaction Principle

The rule-based presentation gives us a way to derive more compact representations of programs. When there is a single rule for an auxiliary function name, we can often combine the effect of that rule with any use of the name and eliminate the rule itself. For example, return to our rules for factorial:

```
fac(N) => f0(N, arb, arb);

f0(N, K, F) => f1(N, K, 1);          // corresponds to assignment F = 1;

f1(N, K, F) => f2(N, 1, F);          // corresponds to assignment K = 1;

f2(N, K, F) => (K <= N) ? f3(N, K, F);

f2(N, K, F) => (K > N) ? f5(N, K, F);

f3(N, K, F) => f4(N, K, F*K);        // corresponds to assignment F = F*K;

f4(N, K, F) => f2(N, K+1, F);        // (corresponds to assignment K = K+1;

f5(N, K, F) => F;
```

We see that `f3` is immediately rewritten in terms of `f4`. Thus anywhere that `f3` occurs could be translated into a corresponding occurrence of `f4` by first making a corresponding substitution of expressions for variables. In our rules, `f3` occurs only once, in the second rule for `f2`. We can substitute the *rhs* of the `f3` rule in the second rule for `f2` to get a replacement for the latter:

```
f2(N, K, F) => (K <= N) ? f4(N, K, F*K);
```

We also see that `f4` is immediately rewritten in terms of `f2`, so the rule could be further changed to:

```
f2(N, K, F) => (K <= N) ? f2(N, K+1, F*K);
```

We can similarly re-express `f0` in terms of `f2`, as follows:

```
f0(N, K, F) => f1(N, K, 1);     // original

f0(N, K, F) => f2(N, 1, 1);     // using rule f1(N, K, F) => f2(N, 1, F);
```

Finally, we can get rid of `f5` since `f5(N, K, F)` is immediately rewritten as `F`

```
f2(N, K, F) => (K > N) ? F;
```

and we can re-express `fac` directly in terms of `f2`. The resulting set of rules is:

```
fac(N) => f2(N, 1, 1);

f2(N, K, F) => (K <= N) ? f2(N, K+1, F*K);

f2(N, K, F) => (K > N) ? F;
```

This is obviously much more compact than the original, since it contains only one auxiliary function, `f2`. The corresponding compacted flowchart, using combined conditions and assignment and parallel assignment, could be shown as follows:

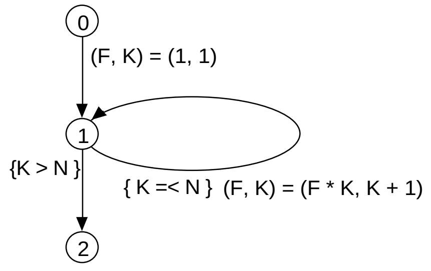


Figure 76: A compact factorial flowchart

List Reverse Compaction Example

Compact the simplified rules for the reverse program:

```
f0(L) => f1(L, [ ], arb, arb);

f1(L, R, M, A) =>  L == [ ] ? f2(L, R, M, A);

f1([A | M], R, _, _) => f3([A | M], R, M, A);

f3(L, R, M, A) => f4(M, R, M, A);

f4(L, R, M, A) => f1(L, [A | R], M, A);

f2(L, R, M, A) => R;
```

We can get rid of `f4` by compaction, giving us:

```
f3(L, R, M, A) => f1(M, [A | R], M, A);
```

We can get rid of `f3` by compaction, giving us:

```
f1([A | M], R, _, _) =>f1(M, [A | R], M, A);
```

We can get rid of `f2` by compaction, and move the guard into the rule, giving

```
f1([], R, M, A) =>  R;
```

The resulting system is:

```
f0(L) => f1(L, [ ], arb, arb);

f1([], R, M, A) =>  R;

f1([A | M], R, _, _) =>f1(M, [A | R], M, A);
```

We notice that the third and fourth arguments of `f1` after compaction never get used, so we simplify to:

```
f0(L) => f1(L, [ ]);

f1([], R) =>  R;

f1([A | M], R) =>f1(M, [A | R]);
```

Now note that `f1` is our earlier 2-argument `reverse` by a different name. So McCarthy's transformation in this case took us to an efficient functional program.

Exercises

1 •• Consider the following Java program fragment for computing `k` to the `n`th power, giving the result in `p`. Give a corresponding set of rules in uncompacted and compacted form.

```
p = 1;
c = 1;
while( c <= n )
  {
  p = p * k;
  c++;
  }
```

2 •• The Fibonacci function can be presented by the following rules:

```
fib(0) => 1;

fib(1) => 1;
fib(N) => fib(N-1) + fib(N-2);
```

However, using straight rewriting on these rules entails a lot of recomputation. [See for yourself by computing fib(5) using the rules.] We mentioned this earlier in a discussion of "caching". Give an assignment-based program that computes fib(N) "bottom up", then translate it to a corresponding set of rules for more efficient rewrite computation. [Hint: fib(N) is the Nth number in the series 1, 1, 2, 3, 5, 8, 13, 21, … wherein each number after the second is obtained by adding together the two preceding numbers.] Then give an equivalent set of rules in compacted form.

The use of a bottom-up computation to represent a computation that is most naturally expressed top-down and recursively is sometimes called the **"dynamic programming principle"**, following the term introduced by Richard Bellman, who explored this concept in various optimization problems.

3 ••• Try to devise a set of rules for computing the integer part of the square root of a number, using only addition and <= comparison, not multiplication or division. If you have trouble, try constructing an assignment-based solution first, then translating it. [Hint: Each square can be represented as the sum of consecutive odd integers. For example, 25 = 1 + 3 + 5 + 7 + 9.]

6.5 Turing Machines – Simple Rewriting Systems

We conclude this chapter with a different example that can be represented by rewriting rules, one of major interest in theoretical computer science. The main definitions, however, should be part of the cultural knowledge of every scientist, since they relate to certain problems having an unsolvable character, about which we shall say more in *Limitations of Computing.*

Turing machines **(TMs)** are named after the British mathematician Alan M. Turing, 1912-1954[†], who first proposed them as a model for universal computability. By "universal" we mean a model general enough to enable the representation of any computable function. Although they share this property with the general recursive functions, Turing machines are much more basic. For example, they do not assume a general matching capability. For that matter, they do not assume anything about numbers or arithmetic. Instead, everything, including matching, is done in terms of a finite set of symbols. Any numeric interpretation of those symbols is up to the user.

[†] For a biography, see the book by Hodges, *Alan Turing: The Enigma*. The title is a pun. Not only is Turing regarded as enigmatic, but also one of his principal contributions was a machine which helped break codes of the German encoding machine known as the Enigma during World War II.

Each Turing machine has a fixed or "wired-in" program, although through an encoding technique, it is possible to have a single machine that mimics the behavior of any other TM. The components of a TM are:

Finite-state controller (sequential circuit)

Movable read/write head

Unbounded storage tape:

On each tape cell, one symbol from a **fixed finite alphabet** can be written. One symbol in the alphabet is distinguished as "**blank**". The distinction of blank is that, in any given state, at most a finite set of the cells will contain other than blank.

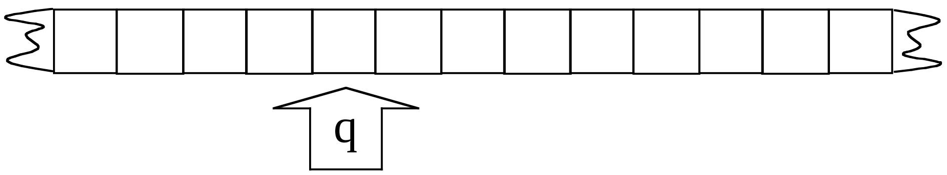


Figure 77: Depiction of a TM with controller in state q.

Since only a finite set of cells can be non-blank at any time, beyond a certain cell on either side of the tape, the cells contain only blanks. However, this boundary will generally change, and the non-blank area may widen.

The controller, in a given state (which includes the controller state and the tape state), will:

- read the symbol under the head
- write a new symbol (possibly the same as the one read)
- change state (possibly to the same state)
- move one cell left or right, or stay put

Program or Transition Function for a Turing Machine

A description of, or program for, a TM is a partial function, called the **transition function**:

States x Symbols $\rightarrow$ Symbols x Motions x States

where Motions = {left, right, none}

The symbol x here denotes the Cartesian product of sets.

In the present case, we are saying that the transition function of a TM is a partial function having a domain consisting of all pairs of the form

(State, Symbol)

and a co-domain consisting of all 3-tuples (triples) of the form

(Symbol, Motion, States)

Such a program could be given by a *state transition table*, with each row of the table listing one combination of five items (i.e. a 5-tuple or quintuple)

State, *Read*, *Written*, *Motion*, *NewState*

Such a 5-tuple means the following:

> If the machine's control is in *State* and the head is over symbol *Read*, then machine will write symbol *Written* over the current symbol, move the head as specified by *Motion*, and the next state of the controller will be *NewState*.

If no transition is specified for a given state/symbol pair, then the machine is said to *halt* in that state. The requirement that the program be a *partial function* is equivalent to saying the following:

> There is at most one row of the table corresponding to any given combination (*State*, *Read*).

The Partial Function Computed by a Turing Machine

The partial function computed by a TM is the tape transformation that takes place when the machine starts in a given initial state and eventually halts. Alternatively, the *partial function computed* by a TM is the transformation from initial tape to one of a finite set of halting combinations (state-symbol pairs). If the machine does not halt on a given input, then we say the partial function is *undefined* for this input.

Turing Machine Addition Example

We show the construction of a machine that adds 1 to its binary input numeral. It is assumed that the machine starts with the head to the immediate right of the least significant bit, which is on the right. The alphabet of symbols on the tape will be {0, 1, _}, where _ represents a blank cell.

```
_ _ _ _ _ 1 0 1 1 0 1 1 1 1 _ _ _ _ _ _
                ^
```

The state transitions for such a machine could be specified as:

State	Read	Written	Motion	NewState
start	_	_	left	add1
add1	0	1	right	end
add1	_	1	right	end
add1	1	0	left	add1
end	0	0	right	end
end	1	1	right	end

Below we give a trace of this Turing machine's action when started on a binary number 100111 representing decimal 39. The result is 101000, representing decimal 40. The first set of square brackets are around the symbol under the tape head. The second set of brackets shows the control state.

```
1 0 0 1 1 1 [_]  [start]
1 0 0 1 1 [1] _  [add1]
1 0 0 1 [1] 0 _  [add1]
1 0 0 [1] 0 0 _  [add1]
1 0 [0] 0 0 0 _  [add1]
1 0 1 [0] 0 0 _  [end]
1 0 1 0 [0] 0 _  [end]
1 0 1 0 0 [0] _  [end]
1 0 1 0 0 0 [_]  [end]
```

Note the importance of every Turing machine rule-set to be finite. Here we have an example of an infinite-state system with a finite representation.

The *partial function computed* by this machine can be thought of as a representation of the function f where f(x) = x+1, at least as long as the tape is *interpreted* as containing a binary-encoded number.

Universal Turing Machines

A universal Turing Machine (UTM) is a Turing Machine that, given an encoding of *any* Turing machine (into a fixed size alphabet) and an encoding of a tape for that machine, both on the tape of the UTM, simulates the behavior of the encoded machine and tape, transforming the tape as the encoded machine would and halting if, and only if, the encoded machine halts.

The set of all TMs has no bound to the number of symbols used as names of control states nor of tape symbols. However, we can encode an infinity of such symbols using only two symbols as follows: suppose that we name the control states q_0, q_1, q_2, … without loss of generality. Then we could use something like a series of i 1's to represent q_i. However, we need more than this simplistic encoding, since we will need to be able to mark certain of these symbols in the process of simulating the machine being presented. We will not go into detail here.

Exercises

1 •• Develop the rules for a Turing machine that erases its input, assuming that all of the input occurs between two blank cells and the head is initially positioned in this region.

2 •• Develop the rules for a Turing machine that detects whether its input is a palindrome (reads the same from left-to-right as from right-to-left). Assume the input string is a series of 0's and 1's, with blank cells on either end.

3 ••• Develop the rules for a Turing machine that detects whether one input string (called the "search string") is contained within a second input string. The two strings are separated by a single blank and the head is initially positioned at the right end of the search string.

4 ••• Develop a Turing machine that determines if one number is evenly divisible by a second. Assume the numbers are represented as a series of 1's (i.e. in 1-adic notation). The two numbers are separated by a single blank. Assume the head starts at the right end of the divisor.

5 •••• Develop a Turing machine that determines if a number is prime. Assume the number is represented in 1-adic notation. Use the machine in the previous problem as a subroutine.

6 ••• Present an argument that demonstrates that the composition of two functions that are computable by a Turing machine must itself be computable by a Turing machine. Assume that each function takes one argument string.

7 ••••• Develop the rules for a universal Turing machine.

6.6 Turing's Thesis

Turing's thesis, also called *Church's Thesis,* the *Church/Turing Thesis,* the *Turing Hypothesis, etc.,* is the following important assertion.

Every function that is intuitively computable is computable by some TM.

The reason for the assertion's importance is that it is one way of establishing a link between a formal rule-based computation model and what is intuitively computable. That is, we can describe a process in informal terms and as long the description is reasonably clear as a computational method, we know that it can be expressed more formally without actually doing so.

Turing's argument in support of this thesis follows[†]. Note that "computer" should be read as "person doing computation". The relation to a machine comes later in the passage.

> "Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, *i.e.* on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as 17 or 999999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same.
>
> The behavior of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be 'arbitrarily close' and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.
>
> Let us imagine the operations performed by the computer to be split up into "simple operations" which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer. We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind. The situation in regard to the squares whose symbols may be altered in this way is the

[†] from Turing's 1937 paper. I have made minor substitutions in extracting this selection from the context of the paper.

same as in regard to the observed squares. We may, therefore, without loss of generality, assume that the squares whose symbols are changed are always 'observed' squares.

Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognizable by the computer. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed square does not exceed a certain fixed amount. Let us say that each of the new observed squares is within L squares of an immediately previously observed square.

In connection with ‘immediate recognizability’, it may be thought that there are other kinds of squares which are immediately recognizable. In particular, squares marked by special symbols might be taken as immediately recognizable. Now if these squares are marked only by single symbols there can be only a finite number of them, and we should not upset our theory by adjoining these marked squares to the observed squares. If, on the other hand, they are marked by a sequence of symbols, we cannot regard the process of recognition as a simple process. This is a fundamental point and should be illustrated. In most mathematical papers the equations and theorems are numbered. Normally the numbers do not go beyond (say) 1000. It is, therefore, possible to recognize a theorem at a glance by its number. But if the paper was very long, we might reach Theorem 157767733443477; then, further on in the paper, we might find ‘… hence (applying Theorem 157767733443477) we have …'. In order to make sure which was the relevant theorem we should have to compare the two numbers figure by figure, possibly ticking the figures off in pencil to make sure of their not being counted twice. If in spite of this it is still thought that there are other ‘immediately recognizable’ squares, it does not upset my contention so long as these squares can be found by some process of which my type of machine is capable.

The simple operations must therefore include:

(a) Changes of the symbol on one of the observed squares.

(b) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.

It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:

(A) A possible change (a) of symbol together with a possible change of state of mind.

> (B) A possible change (b) of observed squares, together with a possible change of state of mind.
>
> The operation actually performed is determined, as has been suggested above, by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation is carried out.
>
> We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an internal state of the machine. The machine scans B squares corresponding to the B squares observed by the computer. In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than L squares from one of the other scanned squares. The move which is done, and the succeeding configuration, are determined by the scanned symbol and the internal state. The machines as defined here can be constructed to compute the same sequence computed by the computer."

Our definition is the now-customary one with B = 1 and L = 1. That is, there is one scanned square and the head changes position by at most one in a single move.

Evidence in Support of Turing's Thesis

Turing's thesis has withstood the test of time and many examinations. It is generally accepted as true, even though it can't ever be proven. To do so would require a formalization of an informal concept, computability. Then a similar argument would have to be presented to argue the correctness of this new definition, and so forth. The evidence for its validity include:

- Turing's argument itself
- No counter-examples to the thesis have ever been found, despite considerable effort.
- Many other models were developed independently, some of which were argued similarly to characterize computability, then later shown to be equivalent to the Turing model.

Some of these models are enumerated below:

Specific universal computing models

Each of these models is known to have computational power equivalent to the Turing machine model:

Turing machines (Turing)

We have discussed this model above.

general recursive functions (Kleene)
This model is very similar to the rex rewriting system, except that it concerned itself only with natural numbers, not lists.

Partial Recursive Functions (Goedel, Kleene)
This model is more rigidly structured than the general recursive function model. It also deals with computing on the natural numbers. The set of functions is defined using induction, a principle to be discussed in a subsequent section. Even though the defining scheme is a little different, the set of functions defined can be shown to be the same as the general recursive functions.

register machines (Shepherdson and Sturgis)
A register machine is a simple computer with a fixed number of registers, each of which can hold an arbitrary natural number. The program for a register machine is a sequence of primitive operations specifying incrementation and decrementation of a register and conditionally branching on whether a register is 0.

lambda-calculus (Church)
The lambda calculus is a calculus of functional forms based on Church's lambda notation. This was introduced earlier when we were discussing functions that can return functions as values.

phrase-structure grammars (Chomsky)
This model will be discussed in Computing Grammatically.

Markov algorithms (Markov)
Markov algorithms are simple rewriting systems similar to phrase-structure grammars.

tag systems (Post)
Tag systems are a type of rewriting system. They are equivalent to a kind of Turing machine with separate read- and write- heads that move in one direction only, with the tape in between. The name of the system apparently derives from using the model to analyze questions of whether the read-head ever "tags" (catches up with) the write-head.

Practical uses of Turing's Thesis

The "Turing Tarpit"

Suppose that you have invented a new computing model or programming language. One test that is usually applied to such a model is whether or not every computable partial function can be expressed in the model. One way to establish this is to show that any Turing machine can be simulated within the model. Henry Ledgard calls this scheme the "Turing Tarpit".

This principle relies on the acceptance of the Turing machine notion being universally powerful for computability. For most programming languages, another model, known as a **register machine**, a machine with two registers, each of which holds an arbitrary natural number, is simpler to simulate. In turn, a register machine can simulate a Turing machine. We indicate how this is done in a subsequent section.

The Informal-Description Principle

This principle relies on an informal acceptance of Turing's argument. In order to show a certain function is computable by a Turing machine, it suffices to give a verbal description of a computational process, even without the use of a formal model. If challenged, the purveyor of the argument can usually describe in sufficient detail how to carry out the computation on a Turing machine, although such details can quickly reach a laborious level. For many examples of the use of this principle, see a book such as that of Rogers, 1967.

6.7 Expressing Turing Machines as Rewrite Rules

Given two models that are both claimed to be universal with respect to computability, it is important to be able to express every partial function expressible in one model in terms of the other model and vice-versa. Here we show how Turing machine computations can be expressed in terms of rex rules. We generate rules that manipulate lists, and leave to the reader the small remaining task of showing that the lists can be encoded as numbers. In other words, we show in this section:

Every Turing-computable partial function is a general recursive function.

It suffices to give a method for creating rex rules from Turing machine rules. Thus, at the same time, we are using the Turing Tarpit principle to show:

rex is a universal computing language.

We must first show how the state of a Turing machine will be encoded. Recall that the complete state of a Turing machine consists of a control state, a tape, and a head position. We represent the complete state by the following four components:

- The control state
- A list of symbols to the left of the head, reading left-to-right.
- A list of symbols to the right of the head, reading right-to-left.
- The symbol under the head.

In particular, the list of symbols to the left of the head reads in the opposite direction from the list to the right of the head. The reason for this is so that we can use the list-constructing facilities of rex on the symbols around the head.

Recall that a Turing machine is specifiable by a set of 5-tuples:

State, *Read*, *Written*, *Motion*, *NewState*

where *State* and *NewState* refer to control states. For each 5-tuple, there will be one rex rule. The structure of the rule depends on whether *Motion* is left, right, or none. We describe each of these cases. Note that in the rewrite rules, the variables *State*, *Read*, *Written*, *Motion*, and *NewState* will be replaced by literal symbols according to the TM rules, whereas the variables *Left*, *Right*, *FirstLeft*, *FirstRight* will be left as rex variables. The partial function `tm` defined below mimics the state-transitions of the machine.

In the case that *Motion* is *none*:

```
           tm(State, Left, Read, Right)

      => tm(NewState, Left, Written, Right);
```

In the case that *Motion* is *right*:

```
             tm(State, Left, Read, [FirstRight | RestRight])

      => tm(NewState, [Written | Left], FirstRight, RestRight);
```

In the case that *Motion* is *left*:

```
            tm(State, [FirstLeft | RestLeft], Read, Right)

      => tm(NewState, RestLeft, FirstLeft, [Written | Right]);
```

Next, we need one pair of rules to handle the case where the head tries to move beyond the symbols at either end of the tape. In either case, we introduce a blank symbol into the tape. We will use ‘ ‘ to designate the blank symbol.

```
tm(State, [ ], Read, Right) => tm(State, [‘ ‘], Read, Right);

tm(State, Left, Read, [ ]) => tm(State, Left, Read, [‘ ‘]);
```

Finally, we need to provide a rule that will give an answer when the machine reaches a halting state. Recall that these are identified by combinations of states and control symbols for which no transition is specified. We want a rule that will return the tape contents for such a configuration. In order to this, we need to reverse the list representing the left tape and append it to the list representing the right. The two-argument `reverse` function, which appends the second argument to the reverse of the first, is ideally suited for this purpose:

For each *halting* combination: *State*, *Read*, include a rule:

```
tm(State, Left, Read, Right) => reverse(Left, [Read | Right]);
```

where, as before,

```
reverse([ ], M) => M;

reverse([A | L], M) => reverse(L, [A | M]);
```

Turing Machine Example in rex

We give the rex rules for the machine presented earlier that adds 1 to its binary input numeral. We show the original TM rules on the lines within /* … */ and follow each with the corresponding rex rule. Here we use {`‘0‘`,`‘1‘`, `' '`} for tape symbols (single quotes may be used for single characters) and {`“start”`, `“add1”`, `“end”`} for control states.

```
/* start        ' '       ' '       left    add1 */

tm(“start”, [FirstLeft | RestLeft], ' ', Right)
=> tm(“add1”, RestLeft, FirstLeft, [' ' | Right]);


/* add1         0       1       right   end */

tm(“add1”, Left, ‘0‘, [FirstRight | RestRight])
=> tm(“end”, [‘1‘ | Left], FirstRight, RestRight);


/* add1         ' '       1       right   end */

tm(“add1”, Left, ' ', [FirstRight | RestRight])
=> tm(“end”, [‘1‘ | Left], FirstRight, RestRight);


/* add1         1       0       left    add1 */
```

```
tm(“add1”, [FirstLeft | RestLeft], ‘1‘, Right)
=> tm(“add1”, RestLeft, FirstLeft, [‘0‘ | Right]);


/* end  0       0       right   end */

tm(“end”, Left, ‘0‘, [FirstRight | RestRight])
=> tm(“end”, [‘0‘ | Left], FirstRight, RestRight);


/* end  1       1       right   end */

tm(“end”, Left, ‘1‘, [FirstRight | RestRight])
=> tm(“end”, [‘1‘ | Left], FirstRight, RestRight);
```

As specified in the general scheme, we also add the following rules:

```
tm(end, Left, ' ', Right) => reverse(Left, [' ' | Right]);

reverse([ ], M) => M;

reverse([A | L], M) => reverse(L, [A | M]);
```

The following is a trace of the rewrites made by rex on the input tape `100111` representing decimal 39. As discussed in the text, the left list is the reverse of the tape, so the corresponding initial argument is `[‘1’, ‘1’, ‘1’, ‘0’, ‘0’, ‘1’]`.

```
tm(“start”,[‘1’,’1’,’1’,’0’,’0’,’1’], ' ', [ ])    =>
tm(“start”,[‘1’,’1’,’1’,’0’,’0’,’1’], ' ', [' ']) =>
tm(add1,   [‘1’,’1’,’0’,’0’,’1’], ‘1’, [' ',' ']) =>
tm(add1,   [‘1’,’0’,’0’,’1’], ’1’, [‘0’,' ',' ']) =>
tm(add1,   [‘0’,’0’,’1’], ’1’, [‘0’,’0’,' ',' ']) =>
tm(add1,   [‘0’,’1’], ’0’, [‘0’,’0’,’0’,' ',' ']) =>
tm(“end”,[‘1’,’0’,’1’], ’0’, [‘0’,’0’,' ',' '])  =>
tm(“end”,[‘0’,’1’,’0’,’1’], ’0’, [‘0’,' ',' '])  =>
tm(“end”,[‘0’,’0’,’1’,’0’,’1’], ’0’, [' ',' '])  =>
tm(“end”,[‘0’,’0’,’0’,’1’,’0’,’1’], ' ', [' '])  =>
reverse([‘0’,’0’,’0’,’1’,’0’,’1’], [' ',' '])    =>
[‘1’,’0’,’1’,’0’,’0’,’0’,' ',' ']
```

The result `101000` is indeed the representation of the number that would be 40 decimal.

Exercises

1 ••• Show that we can directly convert the function constructed above into a numeric one. [Hint: If the TM has an N-symbol tape alphabet, then lists can be viewed simply as N-adic numerals. Show how to compute, as general recursive functions, the *numeric* functions that extract the first symbol and the remaining symbols from a list.] Then show how each list rewriting rule can be recast as a numeric rewriting rule.

2 ••••• Show that every general recursive function is computable by a Turing machine**.**

3 ••• The following is a list of possible ideas for extending the "power" of the Turing machine notion. Give constructions that show that each can be reduced to the basic Turing machine definition as presented here.

1. Multiple heads on one tape. The transitions generally depend on the symbol under each tape head, and can write on all heads in one move. [Show that this model can be simulated by a single head machine by using "markers" to simulate the head positions.]

2. Multiple tapes, each with its own head.

3. Two dimensional tape. Head can move left, right, up, or down.

4. An infinite set of one-dimensional tapes, with the head being able to alternately select the next or previous tape in the series.

5. N-dimensional tape for N > 2.

6. Adding multiple registers to the control of the machine.

7. Adding some number of "counters", each of which can hold any natural number.

6.8 Chapter Review

• Define the following terms:

Cartesian product
deterministic
dynamic programming
McCarthy's transformation
parallel assignment
reachability
state
transition
transitive closure
Turing machine
Turing's thesis

• Demonstrate the application of McCarthy's transformation principle.

• Demonstrate how to convert a Turing machine program to a rex program.

6.9 Further Reading

•• Jon Barwise and John Etchemendy, *Turing's World 3.0*, CSLI Publications, Stanford California, 1993. [Includes Macintosh disk with visual Turing machine simulator.]

••• Alonzo Church, *The Calculi of Lambda Conversion*, Annals of Mathematical Studies, 6, Princeton University Press, Princeton, 1941. [Introduces the lambda calculus as a universal computing model. Expresses "Church's thesis".]

•• W.F. Clocksin and C. S. Mellish, *Programming in Prolog*, Third edition, Springer-Verlag, Berlin, 1987. [A readable introduction to Prolog.]

••• Rolf Herken (ed.), *The Universal Turing Machine, A Half-Century Survey*, Oxford University Press, Oxford, 1988. [A survey of some of the uses of Turing machines and related models.]

• A. Hodges, *Alan Turing: The Enigma*. Simon & Schuster, New York, 1983. [Turing's biography.]

•• R.M. Keller. *Formal verification of parallel programs*. Communications of the ACM, **19**, 7, 371-384 (July 1976). [Introduces the transition-system concept and applies it to parallel computing.]

••• S.C. Kleene, *Introduction to Metamathematics*, Van Nostrand, Princeton, 1952. [Introduces general recursive functions and related models, including partial recursive functions and Turing machines.]

••• Benoit Mandelbrot. *The Fractal Geometry of Nature*, W.H. Freeman and Co., San Francisco, 1982. [Discusses initial explorations of the idea of fractals.]

•• Zohar Manna, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974. [Examples using McCarthy's transformation and other models.]

•• J. McCarthy, *Towards a mathematical science of computation*, in C.M. Popplewell (ed.), Information Processing, Proceedings of IFIP Congress '62, pp. 21-28, North-Holland, Amsterdam, 1962. [The original presentation of "McCarthy's Transformation.]

•• Marvin Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, New Jersey, 1967. [Discusses Turing machines, including universal Turing machines, and other computational models, such as register machines.]

•••• H. Rogers, *Theory of recursive functions and effective computability*, McGraw-Hill, 1967. [More abstract and advanced treatment of partial recursive functions. Employs the informal description principle quite liberally.]

•• J.C. Shepherdson and H.E. Sturgis, *Computability of recursive functions*, Journal of the ACM, **10**, 217-255 (1963).

••• A.M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proc. London Mathematical Society, ser. 2, vol. 42, pp. 230-265 (1936-37), corrections, ibid., vol. 43, pp. 544-546 (1937). [The presentation of Turing's thesis. Moderate to difficult.]