6

Machines

It is unworthy of excellent people to lose hours like slaves in the labor of calculation which could safely be relegated to anyone else if machines were used.
Gottfried Wilhelm von Leibniz, 1685

Well let's be clear right from the start, I never have been interested in computing, and I'm still not interested in computing. What I'm interested in is computers. I'm an engineer, I define the computer right from square one as a device which was designed to facilitate the performance of mathematics, the greater part of which would be very much better not done.
F. C. Williams, engineer of the first stored-program computer

The first five chapters focused on ways to use language to describe procedures. Although finding ways to describe procedures succinctly and precisely would be useful even if we did not have machines to carry out those procedures, the tremendous practical value we gain from being able to describe procedures comes from the ability of computers to carry out those procedures astoundingly quickly, reliably, and inexpensively. As a very rough approximation, having a modern laptop gives an individual computing power roughly equivalent to having every living human on the planet working for you without ever making a mistake or needing a break.

In this chapter, we introduce computing machines. Computers are different from other machines in two key ways:

1. Whereas other machines amplify or extend our *physical* abilities, computers amplify and extend our *mental* abilities.
2. Whereas other machines are designed for a small set of tasks, computers can be *programmed* to perform many tasks. In fact, the simple computer model we present in this chapter is sufficient to perform *all* possible computations.

The next section gives a brief history of computing machines, from prehistoric calculating aids to the design of the first universal computers. Section 6.2 explains how machines can implement logic. Section 6.3 introduces a simple abstract model of a computing machine that is powerful enough to carry out any algorithm.

We provide only a very shallow introduction to how machines can implement computations. Our primary goal is not to convey the details of how to design and build an efficient computing machine (although that is certainly a worthy goal that is pursued in other computing courses), but to gain sufficient understanding of the properties nearly all conceivable computing machines share to be able to predict properties about the costs involved in carrying out a particular procedure. The following chapters use this to reason about the costs of various procedures. In later chapters, we use it to reason about the range of problems that can and cannot be solved by an mechanical computing machine (Chapter 13), and the set of problems that can be solved by conceivable computing machines in a reasonable amount of time (Chapter 18).

6.1 History of Computing Machines

The goal of early machines was to carry out some physical process with less effort than would be required by a human. These machines took physical things as inputs, performed physical actions on those things, and produced some physical output. For example, a cotton gin takes as input raw cotton, mechanically separates the cotton seed and lint, and produces the separated products as output.

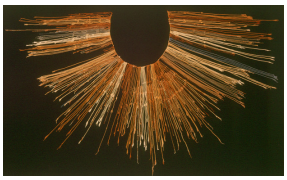
The first big leap toward computing machines was the invention of machines whose purpose is abstract rather than physical. Instead of producing physical things, these machines used physical things to *represent* information. The valuable output of the machine is not its physical effect, but how its physical output can be interpreted as information.

Our first example is not a machine, but using fingers to count. The base ten number system used by most human cultures reflecting our ten fingers for counting.[1] Successful shepherds needed to find ways to count higher than ten. Early shepherds did this by using stones to represent numbers, making the cognitive leap of using a physical stone to represent some quantity of sheep. A shepherd would count sheep by holding stones in his hand that represent the number of sheep.

More complex societies required more counting and more advanced calculating. For example, the Inca civilization in Peru used knots in collections of strings known as *khipu* to keep track of thousands of items for a hierarchical system of taxation. By tying different kinds of knots on different

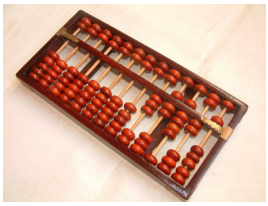
[1]Not all human cultures use base ten number systems. For example, many cultures including the Maya and Basque adopted base twenty systems counting both fingers and toes. This was natural in warm areas, where typical footwear left the toes uncovered.

strings and at different positions on the strings, the Inca accountants could represent thousands of different numbers.



Inca Khipu

Other societies developed machines for computing mathematical operations beyond just counting. An *abacus* is a device for performing calculations by moving beads on rods. Many cultures have developed forms of abaci, including the ancient Mesopotamians and Romans. The Chinese *suan pan* ("calculating plate") is an abacus with a beam subdividing the rods, typically with two beads above the bar (each representing 5), and five beads below the beam (each representing 1). An operator can compute addition, subtraction, multiplication, and division by following mechanical processes using an abacus.

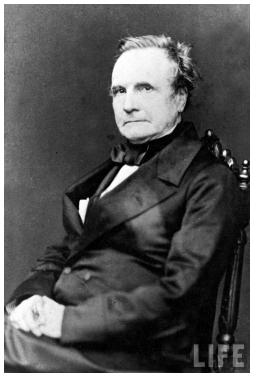


Suan Pan

All of the machines described so far require humans to move parts to perform calculations. As machine technology improved, automatic calculating machines could be built where the operator only needed to set up the inputs (and then turn a crack, push a lever, or use some external power source). The first known automatic calculating machine was built in Germany by Wilhelm Schickard in 1623, and could perform addition and subtraction mechanically, and multiplication and division with assistance from the operator. Other than as a historical artifact, however, Schickard's machine had little impact and the only machine burned in 1624.

The first automatic calculating machine to be widely demonstrated was the *Pascaline*, built by then nineteen-year old French mathematician Blaise Pascal (also responsible for Pascal's triangle from Excursion 5.1) to replace the tedious calculations he had to do to manage his father's accounts. The Pascaline had five wheels, each representing one digit of a number, linked by gears to perform addition with carries. The first machine capable of performing all four basic arithmetic operations (addition, subtraction, multiplication, and division) fully mechanically was built by Gottfried Wilhelm von Leibniz in 1694.

Over the following centuries, smaller, more reliable, and more complex mechanical calculating machines were developed. These machines could perform calculations on large numbers, but they could only perform one operation at a time. Hence, performing a series of calculations was a tedious and error-prone process in which a human operator had to set up the machine for each arithmetic operation, record the result, and reset the machine for the next calculation.



Charles Babbage

Life Magazine

The big breakthrough was the conceptual leap of programmability. A machine is *programmable* if its inputs not only control the values it operates on, but the operations it performs. Babbage was born in London in 1791 and studied mathematics at Cambridge. In the 1800s, calculations were done by looking up values in large books of mathematical and astronomical tables.

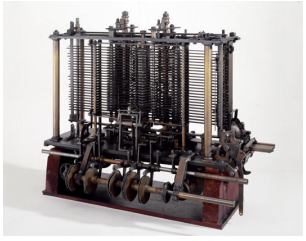
These tables were computed by hand, and often contained errors. Babbage and astronomer John Herschel examined one set of tables in 1821 and found it contained thousands of errors. The calculations were especially important for astronomical navigation, and when the values were incorrect a ship would miscalculate its position at sea (sometimes with tragic consequences).

We got nothing for our £17,000 but Mr. Babbage's grumblings. We should at least have had a clever toy for our money.
Richard Sheepshanks, *Letter to the Board of Visitors of the Greenwich Royal Observatory*, 1854

Babbage sought to develop a machine to mechanize the calculations needed to compute these tables. Starting in 1822, he designed a steam-powered machine known as the Difference Engine to compute polynomials needed for astronomical calculations using Newton's method of successive differences (a generalization of Heron's method from Example 4.4). The Difference Engine was never fully completed, and despite receiving funding from the Royal Astronomical Society responsible for producing the astronomical navigation tables, Babbage endured repeated difficulty securing continued funding from the government.

On two occasions I have been asked by members of Parliament, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
Charles Babbage

As Babbage gained experience with increasingly precise machining, he envisioned a more general calculating machine that could be programmed to perform any calculation. This new machine, the Analytical Engine, designed between 1833 and 1844, was the first general-purpose computer envisioned. One breakthrough in Babbage's design was to feed the machine's outputs back into its inputs. This meant the engine could perform calculations with an arbitrary number of steps by cycling outputs back through the machine.



Analytical Engine Mill

Science Museum, London

The Analytical Engine could be programmed using punch cards, based on the cards that were used by Jacquard looms. Each card could describe an instruction such as loading a number into a variable in the store, moving values, performing arithmetic operations on the values in the store, and, most interestingly, jumping forward and backwards in the instruction cards. The Analytical Engine supported conditional jumps where the jump would be taken depending on the state of a lever in the machine (this is essentially a simple form of the if-expression).

In 1842, Babbage visited Italy and described the Analytical Engine to Federico Luigi Conte di Menabrea, an Italian engineer, military officer, statesman, and mathematician who would later become Prime Minister of Italy. Menabrea published a description of the Analytical Engine in French. Ada Augusta Byron King (also known as Ada, Countess of Lovelace) translated the article into English. In writing the translation, she also added a series of notes to the article (that contained more text than the original article).

In the notes, Ada presented the first detailed program for the Analytical Engine, describing a program to compute Bernoulli numbers. Ada was the first to realize the importance and interest in creating the programs them-

selves, and envisioned how programs could be used to do much more than just calculate mathematical functions. This was the first computer program ever described, and Ada is recognized as the first computer programmer.

Ada Augusta Byron King

Despite Babbage's design, and Ada's vision, the Analytical Engine was never completed. It is unclear whether the main reason for the failure to build a working Analytical Engine was due to limitations of the mechanical components available at the time, or due to Babbage's inability to work with his engineer collaborator or to secure continued funding.

At any rate, the first working programmable computers would not appear for nearly a hundred years. Advances in electronics enables more reliable and faster components than the mechanical components used by Babbage, and the desperation brought on by World War II spurred the funding and efforts that led to working general-purpose computing machines.

With this store available, the next step was to build a computer around it. Tom Kilburn and I knew nothing about computers, but a lot about circuits. Professor Newman and Mr A. M. Turing knew a lot about computers and substantially nothing about electronics. They took us by the hand and explained how numbers could live in houses with addresses and how if they did they could be kept track of during a calculation.
F. C. Williams, engineer of the Manchester Small-Scale Experimental Machine

The remaining conceptual leap was treating the program itself as data, analogous to how we use procedures as inputs and outputs in Chapter 4. In Babbage's Analytical Engine, the program was the stack of cards and the data was the numbers stored in the machine. Thus, there was a big separation between programs and data, and no way for the machine to alter its program.

The idea of treating the program as just another kind of data the machine can process was developed in theory by Alan Turing in the 1930s (Section 6.3 of this chapter describes his model of computing), and first implemented by the Manchester Small-Scale Experimental Machine (built by a team at Victoria University in Manchester) in 1948. With this computer, and all general-purpose computers in use today, the program itself is stored in the machine's memory. Hence, the computer can modify and create new programs by writing into its own memory. It it this power to change its own program that makes stored-program computers so versatile.

Exercise 6.1. Babbage's design for the Analytical Engine called for a store holding 1000 variables, each of which is a 50-digit (decimal) number.

a. How many bits could the store of Babbage's Analytical Engine hold?

b. How does this compare to the Apollo Guidance Computer and to a modern cell phone?

6.2 Mechanizing Logic

George Boole

In this section, we give some insight into how machines can compute, starting with simple logical operations.

We use *Boolean logic*, in which there are two possible values: true (often denoted as 1), and false (often denoted as 0). Boolean logic is named for George Boole (1815-1864), a self-taught British mathematician who published *An investigation into the Laws of Thought, on Which are founded the Mathematical Theories of Logic and Probabilities* in 1854. Before Boole's work, logic was treated as part of philosophy, focused on natural language discourse. Boole's advance was to make logic a formal language, to which the tools of mathematics could be applied. The Boolean datatype in Scheme is based on Boolean logic and can represent the two Boolean values.

Our goal is to show how logical functions can be implemented mechanically. Modern computers use electrons to compute because they are small (more than a billion billion billion (10^{31}) electrons fit within the volume of a grain of sand), fast (approaching the speed of light), and cheap (more than a billion billion (10^{22}) electrons come out of a power outlet for less than a cent). They are also invisible and behave in somewhat mysterious ways, however, so we will instead consider how to compute with wine (or your favorite colored liquid). The basic notions of mechanical computation don't depend on the medium we use to compute, only on our ability to use it to represent values and to perform simple logical operations.

To implement logic using a machine, we need physical ways of representing the two possible values. One representation is to use a full bottle of wine to represent true, and an empty bottle of wine to represent false. So, to represent an input with value true, we will pour a bottle of wine in the input nozzle, and to represent an input with value false, we will pour an empty bottle of wine in the input (or equivalently, do nothing). Similarly, electronic computers typically use presence of voltage to represent true, and absence of voltage to represent false.

6.2.1 Implementing Logic

First, let's consider how to build an *and* function that performs logical conjunction. A logical *and* function takes two inputs, and produces one output. The output is true if both of the inputs are true; otherwise the output is false.

Using Scheme notation, we can write this as (*and A B*) where *and* is the func-

tion being applied to the two inputs, *A* and *B*.[2] We could define a Scheme procedure that performs logical conjunction using an if-expression:

```
(define (logical-and a b)
    (if a b false))
```

To design a mechanical implementation of the logical *and* function, we want a simpler definition that does not involve implementing something as complex as the Scheme if-expression. A different way to define a function is by using a table to show the corresponding output value for each possible pair of input values. This way of defining a function is limited to functions with a small number of possible inputs; we could not define addition on arbitrarily-large integers this way, since there are infinitely many possible different numbers that could be used as inputs. For functions in Boolean logic, there are only two possible values for each input (true and false) so it is reasonable to list the outputs for all possible inputs.

We call a table defining a Boolean function a *truth table*. If there is one input, the table needs two entries, showing the output value for each possible input. When there are two inputs, the table needs four entries, showing the output value for all possible combinations of the input values. The truth table for the logical *and* function is:

truth table

Inputs		**Output**
A	*B*	(*and A B*)
false	false	false
true	false	false
false	true	false
true	true	true

Our goal is to design a machine that implements the function described by the truth table: if both inputs are true (represented by full bottles of wine in our machine), the output should be true (that is, a full bottle of wine); if either input is false (an empty bottle), the output should be false (an empty bottle).

One way to do this is shown in Figure 6.1. Both inputs pour into a basin. The output nozzle is placed at a height corresponding to one bottle of wine in the collection basin, so the output bottle will fill (representing true), only if both inputs are true.

[2] Scheme provides a special form *and* that performs the same function as the logical *and* function. It is a special form, though, since the second input expression is not evaluated unless the first input expression evaluates to true.

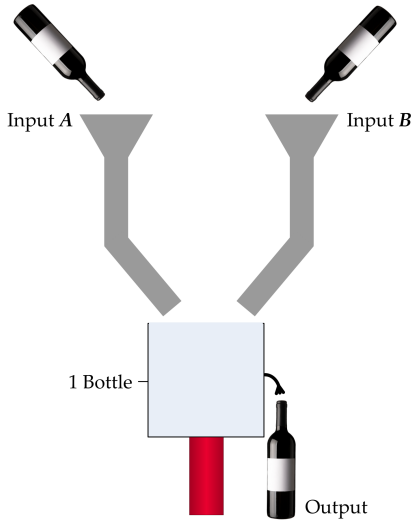


Figure 6.1. Computing *and* with wine.

The design in Figure 6.1 would work in theory, but probably not very well in practice. Some of the wine is likely to spill or get stuck in the pipes, so even when both inputs are true, the output might not be a full bottle of wine, but rather one that is only partially full. While the loss of wine is unfortunate, the real problem is for our representation. What should a $\frac{3}{4}$ full bottle of wine represent? What about a bottle that is half full?

digital abstraction

The solution is the *digital abstraction*. Although there are many different quantities of wine that could be in a bottle, regardless of the actual quantity the value is interpreted as only one of two possible values: true or false. If the bottle has more than a given threshold, say half full, it represents true; otherwise, it represents false. This means an infinitely large set of possible values are abstracted as meaning true, so it doesn't matter which of the values above half full it is.

The digital abstraction provides a transition between the continuous world of physical things and the logical world of discrete values. It is much easier to design computing systems around discrete values than around continuous values; by mapping a range of possible continuous values to just two discrete values, we give up a lot of information but gain in simplicity and reliability. Essentially all computing machines today operate on discrete values using the digital abstraction.

Implementing or. The logical *or* function takes two inputs, and outputs true if any of the inputs are true.[3]

[3]Scheme provides a special form *or* that implements the logical *or* function, similarly to

Here is the truth table for the *or* function:

Inputs		**Output**
A	*B*	(*or A B*)
false	false	false
true	false	true
false	true	true
true	true	true

Try to invent your own design for a machine that computes the *or* function before looking at our solution in Figure 6.2.

Implementing not. Next, consider the logical *not* function. The output of the *not* function is the opposite of the value of its input. It only takes one input, so its truth table only has two entries:

Input	**Output**
A	(*not A*)
false	true
true	false

Implementing a logical *not* is more difficult: the output needs to be the opposite of the input. Alas, it is not possible to produce a logical *not* without some other source of wine; it needs to create wine (to represent true) when there is none input (representing false). To implement the *not* function, we need the notion of a *source current* and a *clock*. The source current injects a bottle of wine on each clock tick. The clock ticks periodically, on each operation. The inputs need to be set up before the clock tick. When the clock ticks, a bottle of wine is sent through the source current, and the output is produced. Try to imagine your own design of a *not* function before looking at our solution in Figure 6.3.

6.2.2 Composing Operations

We have seen how to implement *and*, *or* and *not* using wine, but is that enough to perform interesting computations? In this subsection, we consider how simple logical functions can be combined to implement any logical function; in the following subsection, we see how this is enough to build basic arithmetic operations also.

We start by consider how to make a three-input conjunction function. The *and3* of three inputs is true if and only if all three inputs are true. One way

the *and* special form. If the first input evaluates to true, the second input is not evaluated and the value of the *or* expression is true.

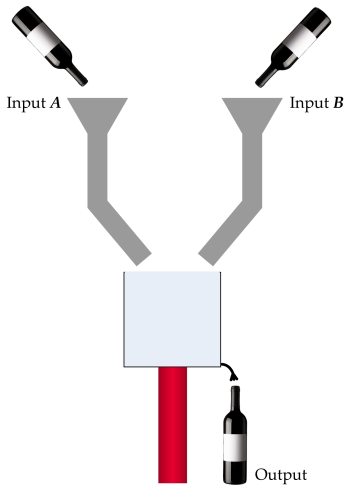


Figure 6.2. Computing *or* with wine.
Our *or* machine is similar to the *and* machine in design, except we move the output nozzle to the bottom of the basin, so if either input is true, the output will be true (when both inputs are true, some wine is spilled, which is a shame, of course, but this does not impact the logical operation).

to make the three-input *and3* is to follow the same idea as the two-input *and*, but make the basin with the output nozzle above the two bottle level (as shown in Figure 6.4).

Another way to implement a three-input *and3* is to compose two of the two-input *and* functions, similarly to how we composed procedures in Section 4.2. Building *and3* by composing two two-input *and* functions allows us to construct a three-input *and3* without needing to design any new structures, as shown in Figure 6.5. The output of the first *and* function is fed into the second *and* function as its first input; the third input is fed directly into the second *and* function as its second input. We could write this as:

(*and* (*and A B*) *C*)

Composing logical functions allows us to build new logical functions also. Consider the *xor* (exclusive or) function that takes two inputs, and has output true when exactly one of the inputs is true.

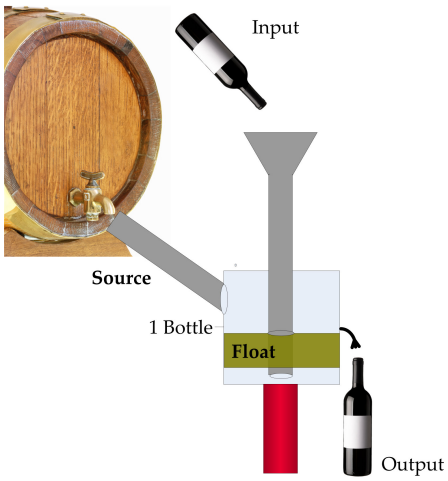


Figure 6.3. Computing *not* with wine.

Before the clock tick, the input is set. If the input is true, the float is lifted, blocking the source opening; if the input is false, the float rests on the bottom of the basin. When the clock ticks, the source wine is injected. If the float is up (because of the true input, the opening is blocked, and the output is empty (false). If the float is down (because of the false input), the opening is open, the source wine will pour across the float, filling the output (representing true). (This design assumes wine coming from the source does not leak under the float, which might be hard to build in a real system, but is reasonable to imagine.)

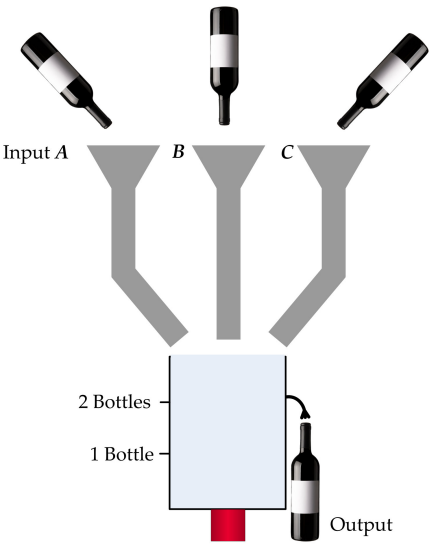


Figure 6.4. Computing the three-input and3.

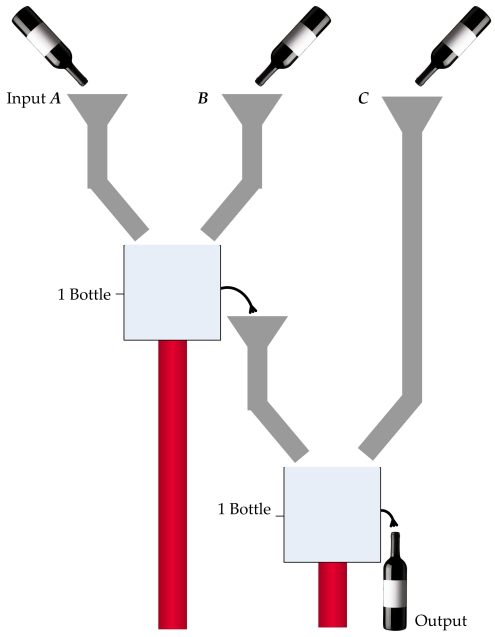


Figure 6.5. Computing *and3* by composing two *and* functions.

The truth-table for *xor* is:

Inputs		**Output**
A	*B*	(*xor A B*)
false	false	false
true	false	true
false	true	true
true	true	false

Can we build *xor* by composing the functions we already have (*and*, *or*, and *not*)?

The *xor* is similar to *or*, except for the result when both inputs are true. So, we could compute (*xor A B*) as:

(*and* (*or A B*) (*not* (*and A B*)))

Another approach is to observe that (*xor A B*) is true in only two situations: (1) when *A* is true and *B* is false, and (2) when *A* is false and *B* is true. Hence, we can compute (*xor A B*) by using the *or* function to combine these two situations:

(*or* (*and A* (*not B*)) (*and* (*not A*) *B*))

The first input to the *or* function is true only if the first situation holds, and the second input is true only if the second situation holds, so the *or* will be true only if at least one of the clauses is true. Thus, we can build an *xor* machine by composing the designs we already have for *and*, *or*, and *not*.

We can also compose different functions, for example, *not* and *and*. We can compose any pair of functions where the outputs for the first function are consistent with the input for the second function (because of mathematical notation, the functions are listed in reverse order: composing *not* and *and* means first perform *and*, and then perform *not* on the output of the *and*). This produces the logical function known as *nand*, described by the truth table below:

Inputs		**Output**
A	*B*	(*nand A B*)
false	false	true
true	false	true
false	true	true
true	true	false

In fact, we can implement all Boolean logic functions on two inputs using just the *nand* function. One way to prove this is to show how to build all logic functions using just *nand*. For example, we can implement *not* using *nand* where the one input to the *not* function is used for both inputs to the *nand* function:

$$(not\ A) \equiv (nand\ A\ A)$$

Now that we have shown how to implement *not* using *nand*, it is easy to see how to implement *and* using *nand*:

$$(and\ A\ B) \equiv (not\ (nand\ A\ B))$$

Implementing *or* is a bit trickier. Recall that *A or B* is true if any one of the inputs is true. But, *A nand B* is true if both inputs are false, and false if both inputs are true. To compute *or* using only *nand* functions, we need to invert both inputs:

$$(or\ A\ B) \equiv (nand\ (not\ A)\ (not\ B))$$

To complete the proof, we would need to show how to implement all the other Boolean logic functions. We omit the details here, but leave some of the other functions as exercises.

Exercise 6.2. Define a Scheme procedure, *logical-or*, that takes two inputs and outputs the logical or of those inputs.

Exercise 6.3. What is the meaning of composing *not* with itself? For example, (*not* (*not A*)).

Exercise 6.4. [⋆] Show how to implement the *xor* function using only *nand* functions.

Exercise 6.5. [⋆] Our definition of (*not A*) as (*nand A A*) assumes there is a way to produce two copies of a given input. Design a component for our wine machine that can do this. It should take one input, and produce two outputs, both with the same value as the input. (Hint: when the input is true, we need to produce two full bottles as outputs, so there must be a source similarly to the *not* component.)

Exercise 6.6. [⋆] The digital abstraction works fine as long as actual values stay close to the value they represent. But, if we continue to compute with the outputs of functions, the actual values will get increasingly fuzzy. For example, if the inputs to the *and3* function in Figure 6.5 are initially all $\frac{3}{4}$ full bottles (which should be interpreted as true), the basin for the first *and* function will fill to $1\frac{1}{2}$, so only $\frac{1}{2}$ bottle will be output from the first *and*. When combined with the third input, the second basin will contain $1\frac{1}{4}$ bottles, so only $\frac{1}{4}$ will spill into the output bottle. Thus, the output will represent false, even though all three inputs represent true. The solution to this problem is to use an *amplifier* to restore values to their full representations. Design a wine machine amplifier that takes one input and produces a strong representation of that input as its output. If that input represents true (any value that is half full or more), the amplifier should output true, but with a strong, full bottle representation. If that input represents false (any value that is less than half full), the amplifier should output a strong false value (completely empty).

6.2.3 Arithmetic

Not only is the *nand* function complete for Boolean logical functions, it is also enough to implement all discrete arithmetic functions. First, consider the problem of adding two one-bit numbers.

There are four possible pairs of inputs:

Inputs				**Output**	
A		B		r_1	r_0
0	+	0	=	0	0
0	+	1	=	0	1
1	+	0	=	0	1
1	+	1	=	1	0

We can compute each of the two output bits as a logical function of the two input bits. The right output bit is 1 if both input bits are 0 or both input bits are 1:

$$r_0 = (or\ (and\ (not\ A)\ (not\ B))\ (and\ A\ B))$$

More simply, we can observe that r_0 is 1 only when exactly one of A and B is 1. This is what the *xor* function computes, so:

$$r_0 = (xor\ A\ B)$$

The left output bit is 0 for all inputs except when both inputs are 1:

$$r_1 = (and\ A\ B)$$

Since we have already seen how to implement *and*, *or*, *xor*, and *not* using only *nand* functions, this means we can implement a one-bit adder using only *nand* functions.

Implementing an adder for larger numbers requires more logical functions. Consider adding two n-bit numbers:

$$\begin{array}{cccccccc} & & a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \\ + & & b_{n-1} & b_{n-2} & \cdots & b_1 & b_0 \\ \hline = & r_n & r_{n-1} & r_{n-2} & \cdots & r_1 & r_0 \end{array}$$

The algorithm most people (at least in North America) learn for adding decimal numbers in elementary school is to sum up the digits from right to left. If the result in one place is more than one digit, the additional tens are carried to the next digit. We use c_k to represent the carry digit in the k^{th} column.

$$\begin{array}{cccccccc} & c_n & c_{n-1} & c_{n-2} & \cdots & c_1 & \\ & & a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \\ + & & b_{n-1} & b_{n-2} & \cdots & b_1 & b_0 \\ \hline = & r_n & r_{n-1} & r_{n-2} & \cdots & r_1 & r_0 \end{array}$$

The algorithm for addition is:

- Initially, $c_0 = 0$.
- Then, repeat for each digit k from 0 to n:
  1. $v_1 v_0 = a_k + b_k + c_k$ (if there is no digit a_k or b_k use 0).
  2. $r_k = v_0$.
  3. If v has two digits, $c_{k+1} = v_1$; otherwise $c_{k+1} = 0$.

Note that this is an algorithm (perhaps the first interesting one most people learn): if followed correctly, it is guaranteed to produce the correct result, and to always finish, for any two input numbers.

Step 1 seems to require already knowing how to perform addition, since it uses $+$. But, the numbers added are one-digit numbers (and c_k is 0 or 1). Hence, there are a finite number of possible inputs for the addition in step 1: 10 decimal digits for $a_k \times 10$ decimal digits for $b_k \times 2$ possible values of c_k. We can memorize the 100 possibilities for adding two digits (or write them down in a table), and easily add one as necessary for the carry. Hence, computing this addition does not require a general addition algorithm, just a specialized method for adding one-digit numbers.

We can use the same algorithm to sum binary numbers, except it is simpler since there are only two binary digits. Without the carry bit, the result bit, r_k, is 1 if $(xor\ a_k\ b_k)$. If the carry bit is 1, the result bit should flip. So,

$$r_k = (xor\ (xor\ a_k\ b_k)\ c_k)$$

This is the same as adding $a_k + b_k + c_k$ base two and keeping only the right digit.

The carry bit is 1 if the sum of the input bits and previous carry bit is greater than 1. This happens when any two of the bits are 1:

$$c_{k+1} = (or\ (and\ a_k\ b_k)\ (and\ a_k\ c_k)\ (and\ b_k\ c_k))$$

As with elementary school decimal addition, we start with $c_0 = 0$, and proceed through all the bits from right to left.

We can propagate the equations through the steps to find a logical equation for each result bit in terms of just the input bits. First, we simplify the functions for the first result and carry bits based on knowing $c_0 = 0$:

$$r_0 = (xor\ (xor\ a_0\ b_0)\ c_0) = (xor\ a_0\ b_0)$$
$$c_1 = (or\ (and\ a_0\ b_0)\ (and\ a_0\ c_0)\ (and\ b_0\ c_0)) = (and\ a_0\ b_0)$$

Then, we can derive the functions for r_1 and c_2:

$$r_1 = (xor\ (xor\ a_1\ b_1)\ c_1) = (xor\ (xor\ a_1\ b_1)\ (and\ a_0\ b_0))$$
$$c_2 = (or\ (and\ a_1\ b_1)\ (and\ a_1\ c_1)\ (and\ b_1\ c_1))$$
$$= (or\ (and\ a_1\ b_1)\ (and\ a_1\ (and\ a_0\ b_0))\ (and\ b_1\ (and\ a_0\ b_0)))$$

As we move left through the digits, the terms get increasingly complex.

But, for any number of digits, we can always find functions for computing the result bits using only logical functions on the input bits. Hence, we can implement addition for any length binary numbers using only *nand* functions.

Using a similar strategy, we can also implement multiplication, subtraction, and division using only *nand* functions. We omit the details here, but the essential approach of breaking down our elementary school arithmetic algorithms into functions for computing each output bit will work for all of the arithmetic operations.

Exercise 6.7. Adding logically.

a. What is the logical formula for r_3?

b. Without simplification, how many functions will be composed to compute the addition result bit r_4?

c. [⋆] Is it possible to compute r_4 with fewer logical functions?

Exercise 6.8. [⋆] Show how to compute the result bits for binary multiplication of two 2-bit inputs using only logical functions.

Exercise 6.9. [⋆] Show how to compute the result bits for binary multiplication of two inputs of any length using only logical functions.

6.3 Modeling Computing

By composing the logic functions, we could build a wine computer to perform any Boolean function. And, we can perform any discrete arithmetic

function using only Boolean functions. For a useful computer, though, we need programmability. We would like to be able to make the inputs to the machine describe the logical functions that it should perform, rather than having to build a new machine for each desired function. We could, in theory, construct such a machine using wine, but it would be awfully complicated. Instead, we consider programmable computing machines abstractly.

Recall in Chapter 1, we defined a computer as a machine that can:

1. Accept input.
2. Execute a mechanical procedure.
3. Produce output.

So, our model of a computer needs to model these three things.

In real computers, input comes in many forms: typing on a keyboard, moving a mouse, packets coming in from the network, an accelerometer in the device, etc.

Figure 6.6. Sample input devices.

Keyboard, mouse, camera, touchscreen, and microphone.

For our model, we want to keep things as simple as possible, though. From a computational standpoint, it doesn't really matter how the input is collected. As seen in Chapter 1, so long as the input is a discrete we can represent any input with just a sequence of bits. Input devices like keyboards are clearly discrete: there are a finite number of keys, and each key could be assigned a unique number. Input from a pointing device like a mouse could be continuous, but we can always identify some minimum detected movement distance, and record the mouse movements as discrete numbers of move units and directions. Richer input devices like a camera or microphone can also produce discrete output by discretizing the input using a process similar to the image storage in Chapter 1. So, the information produces by any input device can be represented by a sequence of bits.

For real input devices, the time an event occurs is often crucial. For example, when playing a video game, it does not just matter than the mouse button was clicked, it matters *when* the click occurs. How can we model inputs where time matters using just our simple sequence of bits?

One way would be to divide time into discrete quanta (as discussed in the footnote on page 1-13, it is not know if time is actually discrete or continuous, but we can sensibly divide time into small enough quanta that the quantization is imperceptible to a human) and encode the input as zero or one events in each quanta. A more efficient way would be to add a timestamp to each input. The timestamps are just numbers (e.g., the number of milliseconds since the start time), so can be written done just as a sequence of bits.

Thus, we can model a wide range of complex input devices with just a finite sequence of bits. The input must be finite, since our model computer needs all the input before it starts processing. This means we our model is not a good model for computations where the input is infinite, such as a web server intended to keep running and processing new inputs (e.g., requests for a web page) forever. In practice, though, this isn't usually a big problem with our model since we can make the input finite by limiting the time the server is running in the model.

The finite sequence of bits could be modeled using a long, narrow, tape that is divided into squares, where each square contains one bit of the input.

We'll consider the output next. Output from computers effects the physical world in lots of very complex ways: displaying images on a screen, printing text on a printer, sending an encoded web page over a network, sending an electrical signal to an anti-lock brake to increase the braking pressure, etc.

Figure 6.7. Sample output devices.

Monitor, multi-screen display, printer, and speakers.

We don't attempt to model the physical impact of computer outputs; that would be far too complicated, but it is also one step beyond modeling the computation itself. Instead, we consider just the information content of the output. The information in a picture is the same whether it is presented as a sequence of bits or an image projected on a screen, its just less pleasant to look at as a sequence of bits. So, we can model the input just like we modeled the output: a sequence of bits written on a tape divided into squares.

Modeling the processing is trickier. We need a model that can capture every

possible mechanical procedure, but we want the model to be as simple as possible. One thing our model computer needs is a way to keep track of what it is doing. We can think of this like scratch paper: a human would not be able to do a long computation without keeping track of intermediate values on scratch paper, and a computer has the same need. In Babbage's Analytical Engine, this was called the *store*, and divided into a thousand variables, each of which could store a fifty decimal digit number. In the Apollo Guidance Computer, the working memory was divided into banks, each bank holding 1024 words. Each word was 15 bits (plus one bit for error correction). In current 32-bit processors, such as the x86, memory is divided into pages, each containing 1024 32-bit words.

For our model machine, we don't want to have arbitrary limits on the amount of working storage. So, we model the working storage with an infinitely long tape. Like the input and output tapes, it is divided into squares, and each square can contain one symbol. For our model computer, it is useful to think about having an infinitely long tape, but of course, no real computer has infinite amounts of working storage. We can, however, imagine continuing to add more memory to a real computer as needed until we have enough to solve a given problem, and adding more if we need to solve a larger problem.

So, our model involves three tapes: one for the input, one for the output, and one for the working tape. We can simplify the model by using a single tape for all three. At the beginning of the execution, the tape contains the input (which must be finite). As processing is done, the input is read and the tape is used as the working tape. Whatever is on the tape and the end of the execution is the output.

We also need a way for our model machine to interface with the tape. The simplest possible interface is to have a tape head that identifies a current square on the tape. The tape head can read the symbol in the current square, can write a symbol in the current square, and can move one square either left or right.

The final thing we need is a way to model actually doing the processing. In our model, this means controlling what the tape head does: at each step, it needs to decide what to write on the tape, and whether to move left or right, or to finish the execution. Babbage called the processor the *mill*; in modern computers, it is called the *central processing unit* (CPU).

In early computing machines, processing meant performing one of the basic arithmetic operations (addition, subtraction, multiplication, or division). We don't want to have to model anything as complex as multiplication in our model machine, however. In Section 6.2.3, we how simpler logical operations can be used to describe addition. The other arithmetic oper-

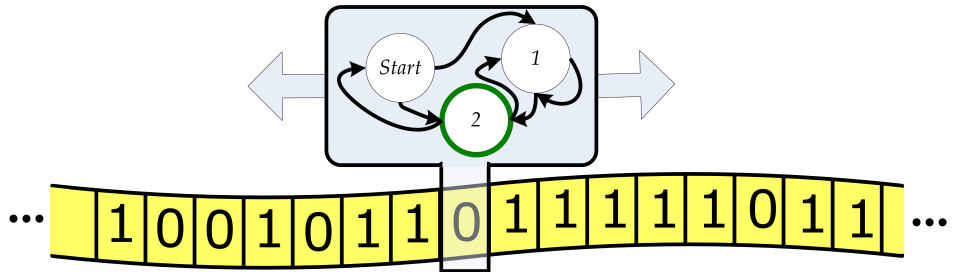


Figure 6.8. Turing Machine model.

ations can also be defined in terms of simpler logical operations, indeed, all computations can be broken down into compositions of simple logical operations. To carry out a complex operation as a composition of simple operations, we need a way to keep track of enough state to know what to do next. The machine state is just a number that keeps track of what the machine is doing. Unlike the tape, it is limited to a finite number. There are two reasons why the machine state number must be finite: first, we need to be able to write down the program for the machine by explaining what it should do in each state, which would be difficult if there were infinitely many states.

To control the machine, we need rules to control what the tape head does. We can think of each rule as a mapping from the current observed state of the machine to what to do next. The input for a rule is the symbol in the current tape square and the current state of the machine; the output of each rule is three things: the symbol to write on the current tape square, the direction for the tape head to move (left, right, or halt), and the new machine state. So, we can describe the program for the machine by listing the rules. For each machine state, we need a rule for each possible symbol on the tape.

Alan Turing

Image from Bletchley Park Ltd.

This abstract model of a computer was invented by Alan Turing in the 1930s and is known as a *Turing Machine*. Turing's model is depicted in Figure 6.8. An infinite tape divided into squares is used as the input, working storage, and output. A tape head can read the current square on the tape, write a symbol into the current tape square, and move left or right one position. The tape head keeps track of its internal state, and follows rules matching the current state and current tape square to determine what to do next. In the example at the end of this section, we describe a Turing Machine for solving a simple problem. It is quite tedious, and rarely necessary, to describe a Turing Machine in full detail, but instructive to do it once.

Turing's model is equivalent to the model we described earlier, but instead of using only bits as the symbols on the tape, Turing's model uses members

of any finite set of symbols, known as the *alphabet* of the tape. Allowing the tape alphabet to contain any set of symbols instead of just the two binary digits makes it easier to describe a Turing Machine that computes a particular function, but does not change the power of the model. That means, every computation that could be done with a Turing Machine using any alphabet set, could also be done by some Turing Machine using only the binary digits.

We could show this by describing an algorithm that takes in a description of a Turing Machine using an arbitrarily large alphabet, and produces a Turing Machine that uses only two symbols to simulate the input Turing Machine. As we saw in Chapter 1, we can map each of the alphabet symbols to a finite sequence of binary digits. Mapping the rules is more complex: since each original input symbol is now spread over several squares, we need extra states and rules to read the equivalent of one original input. For example, suppose our original machine uses 16 alphabet symbols, and we map each symbol to a 4-bit sequence. If the original machine used a symbol `X`, which we map to the sequence of bits `1011`, we would need four states for every state in the original machine that has a rule using `X` as input. These four states would read the `1`, `0`, `1`, `1` from the tape. The last state now corresponds to the state in the original machine when an `X` is read from the tape. To follow the rule, we also need to use four states to write the bit sequence corresponding to the original write symbol on the tape. Then, simulating moving one square left or right on the original Turing Machine, now requires moving four squares, so requires four more states. Hence, we may need 12 states for each edge on the original machine, but can simulate everything it does using only two symbols.

universal computing machine

The Turing Machine model is a *universal computing machine*. This means that every algorithm can be implemented by some Turing Machine. We call a system that can implement every algorithm *Turing complete*. A system that is Turing complete can simulate every possible Turing Machine. Chapter 13 explores more deeply what it means to simulate every possible Turing Machine, and the set of problems that can be solved by a Turing Machine.

Of course, any real machine is limited by the amount of space it has; as we saw in Chapter 1, the amount of information a machine can process is limited by its memory. If the machine does not have enough space to store 1000 bits, say, there is no way it can do a computation where the input requires 1000 bits to describe. For any physical machine, there is some limit on the number of bits it can store. Nevertheless, it is useful to think about most real machines as Turing Machines. The simplicity of the model, and its robustness, make it a useful way to think about computing even if it is not possible to really build a truly universal computing machine.

Although Turing's model is by far the most widely used model for computers today, Turing developed it in 1936, before anything resembling a modern computer existed. Turing did not develop his model as a model of an automatic computer, but instead as a model for what could be done by a human following mechanical rules. He devised the infinite tape to model the two-dimensional graph paper students used to perform arithmetic. He restricted the number of machine states to a finite number by arguing that the amount of information a human could keep in mind at one time was finite. Turing's model has proven to be remarkably robust. Despite being invented before anything resembling a modern computer existed, nearly every computing machine ever imagined or built can be modeled well using Turing's simple model.[4] The important thing about the model is that we can simulate any computer using a Turing Machine. Any step on any computer that operates using standard physics and be simulated with a finite number of steps on a Turing Machine. This means if we know how many steps it takes to solve some problem on a Turing Machine, the number of steps it takes on any other machine is at most some multiple of that number. Hence, if we can reason about the number of steps required for a Turing Machine to solve a given problem, then we can make strong and general claims about the number of steps it would take *any* standard computer to solve the problem. We will show this more convincingly in Chapter 13, but for now we assert it, and use it to reason about the cost of executing various procedures in the following chapter.

Alan Turing was born in London in 1912, and developed his computing model while at Cambridge in the 1930s. He developed the model to solve a famous problem posed by David Hilbert in 1928. The problem, known as the *Entscheidungsproblem* (German for "decision problem") asked for an algorithm that could determine the truth or falsehood of a mathematical statement. To solve the problem, Turing first needed a formal model of an algorithm. For this, he invented the Turing Machine model described above, and defined an algorithm as any Turing Machine that is guaranteed to eventually halt on any input. With the model, Turing was able to show that there are some problems that cannot be solved by *any* algorithm. We return to this in Chapter 13 and explain Turing's proof and examples of problems that cannot be solved.

After publishing his solution to the *Entscheidungsproblem* in 1936, Turing went to Princeton and studied with Alonzo Church (inventor of the Lambda calculus, on which Scheme is based). With the start of World War II, Turing joined the highly secret British effort to break Nazi codes at Bletchley Park. Turing was instrumental in breaking the Enigma code, used by the

[4] In Chapter 19, we describe a few machines that cannot be modeled well using Turing's model.

Bombe

Rebuilt at Bletchley Park

Nazi's to communicate with field units and submarines, and designed an electro-mechanical machine for searching possible keys to decrypt Enigma-encrypted messages. The machines, known as *bombes*, used logical operations to search the possible rotor settings on the Enigma to find the settings that were most likely to have generated an intercepted encrypted message. Bletchley Park was able to break thousands of Enigma messages during the war, and the Allies used the knowledge gained from them to avoid Nazi submarines and gain a tremendous tactical advantage.

After the war, Turing continued to make both practical and theoretical contributions to computer science. Among other things, he worked on designing general-purpose computing machines and published a paper *Intelligent Machinery*, speculating on the ability of computers to exhibit intelligence. Turing introduced a test for machine intelligence (now known as the Turing Test) based on a machines ability to answer questions indistinguishably from a human, and speculated that machines would be able to pass the test within 50 years (that is, by the year 2000). Turing also studied morphogenesis (how biological systems grow), including studying why Fibonacci numbers appear so often in plants.

In 1952, Turing's house was broken into, and Turing reported the crime to the police. The investigation revealed that Turing was a homosexual, which at the time was consider a crime in Britain. Turing did not attempt to hide his homosexuality, and was convicted and given a choice between serving time in prison and taking hormone treatments. He accepted the treatments, and has his security clearance revoked. In 1954, at the age of 41, Turing was found dead in an apparent suicide, with a cynide-laced partially-eaten apple next to him. The codebreaking effort at Bletchley Park was kept secret for many years after the war (Turing's report on Enigma was not declassified until 1996), so Turing never received public recognition for his contributions to the war effort.

Example 6.1: Balancing Parentheses. Here we define a Turing Machine that solves the problem of checking parentheses are well-balanced. For example, in a Scheme expression, every opening left parenthesis must have a corresponding closing right parenthesis. For example, `(()(()))()` is well-balanced, but `())))(()` is not (even though it has the same number of open and close parentheses). Our goal is to design a Turing Machine that takes as input a string of parentheses (with a `#` at the beginning and end to mark the endpoints) and produces as output a `1` on the tape if the input string is well-balanced, and a `0` otherwise. For this problem, the output is what is written in the square under the tape head; it doesn't matter what is left on the rest of the tape.

Our strategy is to find matching pairs of parentheses and cross them out

(by writing an `X` on the tape in place of the parenthesis). If all the parentheses are crossed out at the end, the input was well-balanced, so the machine writes a `1` as its output and halts. If not, the input was not well-balanced, and the machine write a `0` as its output and halts. The trick to the matching is to observe a closing parenthesis always matches the first open parenthesis found moving to the left from the closing parenthesis. The plan for the machine is to move the tape head to the right (without changing the input) until a closing parenthesis is found. Cross out that closing parenthesis by replacing it with an `X`, and move to the left until an open parenthesis is found. This matches the closing parenthesis, so it is replaced with an `X`. Then, continue to the right searching for the next closing parenthesis. If the end of the tape (marked with a `#`) is found when searching for a closing parenthesis, check the tape has no remaining open parenthesis.

Hence, we need three internal states: *LookForClosing*, in which the machine continues to the right until it finds a closing parenthesis (this is the start state); *LookForOpen*, in which the machine continues to the left until it finds the balancing open parenthesis; and *CheckTape*, in which the machine checks there are no unbalanced open parentheses on the tape starting from the right end of the tape and moving towards the left end. The full rules are:

State	Read	Next State	Write	Move	
LookForClosing	`)`	*LookForOpen*	`X`	←	*Found closing.*
LookForClosing	`(`	*LookForClosing*	`(`	→	*Keep looking.*
LookForClosing	`X`	*LookForClosing*	`X`	→	*Keep looking.*
LookForClosing	`#`	*CheckTape*	`#`	←	*End of tape.*
LookForOpen	`)`	-	`X`	Error	*Shouldn't happen.*
LookForOpen	`(`	*LookForClosing*	`X`	→	*Found open.*
LookForOpen	`X`	*LookForOpen*	`X`	←	*Keep looking.*
LookForOpen	`#`	-	`0`	Halt	*Reached beginning.*
CheckTape	`)`	-	`0`	Error	*Shouldn't happen.*
CheckTape	`(`	-	`0`	Halt	*Unbalanced open.*
CheckTape	`X`	*CheckTape*	`X`	←	*Keep checking.*
CheckTape	`#`	-	`1`	Halt	*Finished checking.*

Another way to depict a Turing Machine is to show the states and rules graphically. Each state is a node in the graph. For each rule, we draw an edge on the graph between the starting state and the next state, and label the edge with the read and write tape symbols (separated by a `/`), and move direction. Figure 6.9 shows the same Turing Machine as the rules above. When a read symbol in a given state indicates an error (such as when a `)` is encountered in the *LookForOpen* state), it is not necessary to draw an edge on the graph. If there is no outgoing edge for the current read symbol for the current state in the state graph, execution terminates with an error.

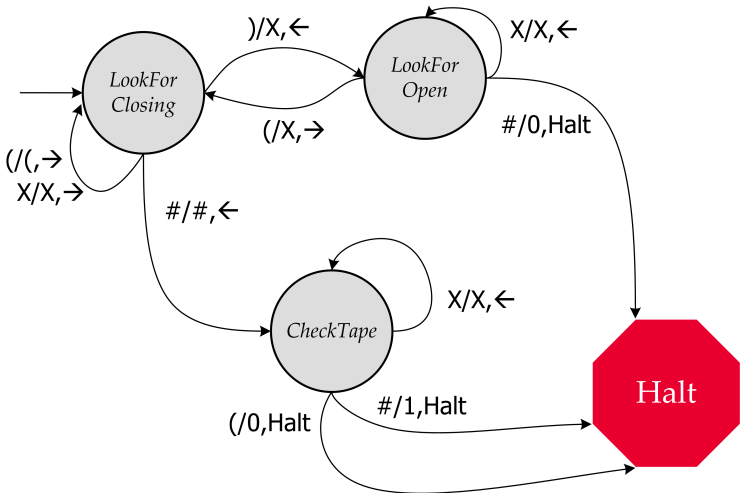


Figure 6.9. Turing Machine for checking balanced parentheses.

Exercise 6.10. Follow the rules to simulate what the Turing Machine will do on each input (assume the beginning and end of the input is marked with a #):

a.)

b. (

c. ()

d. *empty input*

e. (( ) (( ))) ( )

f. (( ))) (( )

Exercise 6.11. [⋆] Design a Turing Machine for adding two arbitrary-length binary numbers. The input is of the form $a_{n-1} \ldots a_1 a_0 + b_{m-1} \ldots b_1 b_0$ (with # markers at both ends) where each a_k and b_k is either 0 or 1. The output tape should contain bits that represent the sum of the two inputs.

6.4 Summary

The power of computers comes from their programmability. Universal computers can be programmed to execute any algorithm. The Turing Machine model provides a simple, abstract, model of a computing machine. Despite its simplicity, every algorithm can be implemented as a Turing Machine, and a Turing Machine can simulate any other reasonable computer.

As the first computer programmer, Ada deserves the last word:

It may be desirable to explain, that by the word operation, we mean any process which alters the mutual relation of two or more things, be this relation of what kind it may. This is the most general definition, and would include all subjects in the universe. In abstract mathematics, of course operations alter those particular relations which are involved in the considerations of number and space, and the results of operations are those peculiar results which correspond to the nature of the subjects of operation. But the science of operations, as derived from mathematics more especially, is a science of itself, and has its own abstract truth and value; just as logic has its own peculiar truth and value, independently of the subjects to which we may apply its reasonings and processes.. . .

The operating mechanism can even be thrown into action independently of any object to operate upon (although of course no result could then be developed). Again, it might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine. Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.

Ada Augusta Byron King, *Sketch of The Analytical Engine*,
Translator's Note A, 1843