“Good morning. I’d like to buy a cat.”
“Certainly sir. I’ve got a lovely terrier.”
“No, I want a cat really.”
“Oh, yeah, how about that?”
“No, that’s the terrier.”
“Well, it’s as near as dammit.”
“Well what do you mean? I want a cat.”
“Listen, tell you what. I’ll file its legs down a bit, take its snout out, stick a few wires through its cheeks. There you are, a lovely pussy cat.”
“It’s not a proper cat.”
“What do you mean?”
“It wouldn’t miaow.”
“Well it would howl a bit.”

We’re now ready for one of the mathematical centerpieces of this book, the study of algorithms that decide membership in languages using a constant amount of memory — that is, the amount of memory they use does not depend on the length of the input string. We’ll define a kind of abstract computer called a finite-state machine or a deterministic finite automaton (DFA), that will be able to represent any such algorithm. Then we’ll be able to prove two fundamental facts about such machines and the languages they decide:

- **The Myhill-Nerode Theorem** tells us whether any given language has a finite-state machine deciding it, based on a particular equivalence relation defined from the language. Furthermore, our proof will tell us the smallest possible finite-state machine deciding the language if one exists. In an Excursion, we’ll see how to take any finite-state machine and get the smallest possible one that decides the same language.

- **Kleene’s Theorem** says that a language has a finite-state machine deciding it if and only if it is regular. To prove this, we’ll present procedures that can take a finite-state machine and produce a regular expression for the same language, and vice versa. Along the way we’ll need to develop variations of the finite-state machine, such as the nondeterministic finite automaton (NFA). The proof will take up most of the chapter and use most of the concepts we’ve developed in the book.
14.1 Deterministic Finite Automata

14.1.1 The Definition of a DFA

Our most important computational problem involving a given formal language is to find a decision procedure — an algorithm that takes an arbitrary string as input and returns a boolean that tells whether the string is in the language. In this section our main concern will be the difficulty of decision procedures for various languages. In particular, can we define a decision procedure in such a way that no matter how big the input string is, we won’t use more than a certain amount of memory?

Finding a decision procedure is a simplified form of a problem that occurs quite often in computer science. For example, the first phase of operation of a compiler is lexical analysis — taking the input (a very long string of source code) and breaking it up into “tokens” such as identifiers and reserved words. A text editor or similar program is often asked to find the next occurrence of a particular string.

It turns out the difficulty of these operations can be reasonably well understood by looking at decision procedures, particularly for languages such as \( \Sigma^* \text{THEN} \Sigma^* \). A key question is how the difficulty of the problem scales when you increase the size of the input. If we want to know whether the string \( \text{THEN} \) occurs in a 50 megabyte file, we’re going to have to read the file from end to end. Reading a 100 megabyte file end to end is going to take about twice as long, but is that the only problem? Along with the time of the operation, we’re also going to be concerned with the space needed. Will we need more local variables to process the 100 megabyte file? If so, this could be bad, as we’d prefer to allocate our variables beforehand without knowing how long the input file is.

For many languages, we can design a decision procedure that works as follows. We read the input end to end, and as we go along we remember a constant amount of information. In Java or another programming language, this means that the algorithm’s internal storage consists only of variables that have a limited number of possible values, such as boolean variables or fixed-point variables restricted to specific ranges. When our procedure reaches the end of the input string, the information we’ve stored must allow us to determine whether that string is in the language. To reason mathematically about such decision procedures, we will put them in a special form called a deterministic finite automaton or DFA. We’ll see below that a DFA exists for a language if and only if there exists a decision procedure that uses constant memory and accesses its input sequentially.

**Definition:** A deterministic finite automaton or DFA consists of a finite set \( S \) of states, a particular initial state \( i \in S \), a set of final states \( F \subseteq S \), and a transition function \( \delta \) from \( S \times \Sigma \) to \( S \), where \( \Sigma \) is the input alphabet.

---

1. This practice is sometimes called “grepping” because of the Unix tool grep that does this.
2. Of course any Java variable is restricted to have only a finite number of values, such as the several billion possible values of an int. But we normally think of the amount of memory available to a constant-memory decision procedure as being relatively small.
3. “Automaton” is another Greek word — its plural is “automata”.

14-2
Given a DFA and an input string $w \in \Sigma^*$, we assign a behavior to the machine as follows. Informally, we read each letter of $w$ in order, changing state as determined by the transition function. If we are in state $s$ and we read an $a$, we go into state $\delta(s, a)$. We start in state $\iota$, and our eventual decision depends on where we end up after reading all of $w$. We accept the input (say it is in $L$) if we end up in a state in $F$, and reject the input (say it is not in $L$) otherwise. Formally, we define a behavior function $\delta^*$ from $S \times \Sigma^*$ to $S$ as follows, by induction on the string input:

- $\delta^*(s, \lambda) = s$ for any state $s$.
- $\delta^*(s, wa) = \delta(\delta^*(s, w), a)$ for any state $s$, string $w$, and letter $a$.
- The machine decides a language $L$ if $\delta^*(\iota, w) \in F$ if and only if $w \in L$.

Different presentations of this material vary as to the exact definition of a DFA or a finite-state machine. In particular, finite-state machines ("FSM's") are often defined so that on every time step they also produce an output. These machines are useful in modeling a number of real situations, but we're going to restrict ourselves to the mathematically simpler situation where there is no output (except for the final state) and our only task is to solve the decision problem for a language.

We usually represent a DFA by a picture that is a labeled directed multigraph (Figure 14-1). We have a node for each state, an edge labeled $a$ from node $i$ to node $j$ whenever $\delta(i, a) = j$, an arrow pointing to state $\iota$, and a double circle instead of a single circle around the name of a node that is in the final state set $F$. If there are two or more parallel edges from one node to another, we abbreviate this by a single edge with multiple labels.

---

4The typical example is a coin vending machine, that remembers how much money has been put in and then in some cases returns change or dispenses a product.
14.1.2 Examples of DFA’s

Among the simplest examples of a DFA is the following (Figure 14-2). This DFA inputs a string with alphabet $\Sigma = \{0, 1\}$ and determines whether the number of ones in the string is even. (We saw in Chapter 5 that one regular expression for this language is $0^* (10^* 10^*)^*.$) As we read the string, all we have to remember is whether we’ve seen an odd or even number of ones so far. So let our state set be $\{O, E\}$. Our start state is $E$ because zero is an even number, and our final state set is $\{E\}$ because we want to accept on final state $E$ and reject on final state $O$. To define the transition function, we must describe the behavior for each possible state and each possible input. On input 0 we stay in the same state, and on input 1 we go to the other state.

How could we prove that this DFA is correct? There is a general method, that depends on characterizing the strings that take the DFA to each state. Here it’s pretty simple — strings with an even number of ones wind up in $E$ and strings with an odd number wind up in $O$. We can prove this statement by induction on all strings:

**Lemma:** If $w \in \{0, 1\}^*$, then for the above DFA, $\delta^*(E, w)$ is equal to $E$ if the number of ones in $w$ is even and equal to $O$ otherwise.

**Proof:** We use induction on all strings $w$. For the base case, we need only observe that $\lambda$ has an even number of ones and that $\delta^*(E, \lambda)$ is defined to be $E$. The inductive case, where $w = va$, breaks down into four subcases depending on the input letter $a$ and the value of $\delta^*(E, v)$. If $a = 0$, then $\delta^*(E, v0) = \delta(\delta^*(E, v), 0)$ by the definition of $\delta^*$, and this is equal to $\delta^*(E, v)$ by the definition of $\delta$ for this DFA. By the inductive hypothesis $\delta^*(E, v)$ correctly indicates whether the number of ones in $v$ is odd or even, and this means that $\delta^*(E, v0)$ is also correct because it is the same as $\delta^*(E, v)$ and the two strings $v$ and $v0$ have the same number of ones.

The other case of $a = 1$ is similar. Here $\delta^*(E, v1)$ is defined to be $\delta(\delta^*(E, v), 1)$ which is the opposite state from $\delta^*(E, v)$. By the inductive hypothesis $\delta^*(E, v)$ tells whether the number of ones in $v$ is odd or even, so $\delta^*(E, v1)$ is correct because there is exactly one more one in $v1$. 

Here is another example of a DFA (with alphabet $\Sigma = \{0, 1\}$) (Figure 14-3). This one decides the language $\{w: \text{The next to last letter of } w \text{ exists and is 1}\}$. (This language is easily seen to be regular, with expression $(0 + 1)^* 1(0 + 1)$.) To decide this language, we have to know the next to last letter, which means also remembering the last letter. We’ll need four states, which we can call $\{00, 01, 10, 11\}$. The start state is 00, the final states are 10 and 11, and in general for any bits $a$, $b$, and $c$ we define $\delta(ab, c) = bc$. So, for example, on input 01110 we successively
compute $\delta^*(00, \lambda) = 00$, $\delta^*(00, 0) = 00$, $\delta^*(00, 01) = 01$, $\delta^*(00, 011) = 11$, $\delta^*(00, 0111) = 11$, and $\delta^*(00, 01110) = 10$, concluding that 01110 is in the language because 10 $\in F$.

### 14.1.3 DFA’s In Java

As we mentioned above, there is a DFA deciding a language if and only if there is any constant-memory decision procedure for the language that accesses the input sequentially and only once. To see this, first note that we can convert any such algorithm into a DFA as follows. At any given time, what the algorithm knows can be described by the contents of all the variables and by its position in the program. We may represent each such situation by a sequence consisting of the program counter followed by the value of each variable. There are only a fixed number of such sequences (we could calculate the number using the Product Rule from Chapter 6) and we will let the set of all possible sequences be our set of states. When the algorithm reads a letter and changes its memory, we can represent this by a transition of the DFA.

For the other direction, we can define a pseudo-Java class `DFA` whose objects represent deterministic finite automata, and ensure that the variables in each `DFA` object are restricted to a constant number of values:

```java
public class DFA
{
  // represents a DFA
  natural states; // how many states
  natural letters; // size of input alphabet
  natural start; // start state, is < "states"
  boolean [] isFinal = new boolean [states];
      // isFinal[i] means i is a final state
  natural [] [] delta = new natural [states] [letters];
      // transition function, entries < "states"

  natural getNext()
  { // returns letter number of next input character, code omitted
```
boolean inputDone() {
    // returns whether input has ended, code omitted
}

boolean decide () {
    // returns whether input string is in language of calling DFA
    natural current = start;
    while (!inputDone())
        current = delta [current, getNext()];
    return isFinal[current];}

The decide method has the variable current as its only internal storage, and this variable is always a natural less than states because the values of the array delta are each less than states. From the definitions, it is easy to see (and prove by induction) that current is always δ*(w), where w is the part of the input read so far. Hence the value returned by decide is true if and only if the input is in the language of the DFA.

Note the relationship between the recursive definition of δ* and the recursive definition of a path in the multigraph representing the machine. In fact, we can easily prove (see Problem 14.1.2) by induction on these two definitions that δ*(i, w) = j if and only if there is a path from node i to node j in the graph, such that the edges in the path are labeled, in order, by the letters of w.

14.1.4 Exercises

E14.1.1 (uses Java) Suppose that the instance fields of a DFA object have been initialized to the components of a particular DFA M with transition function δ and start state i. Prove by induction on all strings w that the value of the variable current in the method decide, after w has been read, is equal to δ*(i, w).

E14.1.2 Build a DFA that inputs a binary string and accepts it if and only if it represents an even number in binary notation. Describe your DFA in terms of its formal parts. What is δ*(s, 01110) for your DFA, if s is your start state?

E14.1.3 How many possible two-state DFA’s are there with input alphabet {0, 1}? How many three-state DFA’s?

E14.1.4 Let i and j be any two naturals, with j > 0. Describe a DFA that will determine whether the number of ones in the input is congruent to i modulo j.

E14.1.5 In Robert Adams’ novel Watership Down, the rabbit civilization has a number system in which “five” represents any number too big to be counted on an individual rabbit’s four feet. We call this “counting threshold five”. Describe a DFA that will count the number of ones in an input binary string “threshold five”, i.e., it should decide the language of strings such that a rabbit would say “there are five ones in the string”.

14-6
14.1.5 Problems

P14.1.1 Build a DFA that inputs a binary string and accepts it if and only if the natural it represents in binary notation is divisible by three. Describe it formally and determine $\delta^*(s, 01110)$, where $s$ is your start state.

P14.1.2 Let $M$ be a DFA with transition function $\delta$. Prove carefully, by induction for all strings $w$ over $M$’s alphabet, that $\delta^*(i, w) = j$ if and only if there is a path from node $i$ to node $j$ in the graph of $M$ such that the labels on the edges of the path, read in order, form $w$.

P14.1.3 Suppose that $M_1$ and $M_2$ are two DFA’s with the same input alphabet. We’ll refer to the state set, start state, final state set, and transition function of $M_1$ as $S_1$, $\iota_1$, $F_1$, and $\delta_1$ respectively, and similarly for $M_2$. We define the **product DFA** $M_1 \times M_2$ as follows. The state set is the direct product $S_1 \times S_2$, the set of ordered pairs $(s_1, s_2)$ with $s_1 \in S_1$ and $s_2 \in S_2$. The start state is the pair $(\iota_1, \iota_2)$ and the final state set is $F_1 \times F_2$. The new transition function takes a state $(s_1, s_2)$ and a letter $a$ to $(\delta_1(s_1, a), \delta_2(s_2, a))$. Prove that the product DFA decides the language $L(M_1) \cap L(M_2)$.

P14.1.4 Design a variant of the product DFA from Problem 14.1.3 that decides the language $L(M_1) \cup L(M_2)$. (Hint: You need change only the final state set.)

P14.1.5 (Suitable for an Excursion) Design a DFA to decide the “Fibonacci language” $(0+10)^*(\lambda+1)$, the language of strings that never contain 11 as a substring. Prove that your DFA is correct, by finding a characterization of the strings taking it to each state and proving by induction on strings that these characterizations are all correct.

P14.1.6 Prove that every language with a finite number of strings is the language of some DFA.

P14.1.7 Find the language decided by each of the two-state DFA’s counted in Exercise 14.1.3. (Hint: It is possible to solve many cases at once.)