CHAPTER 1

Fundamentals

Algebraists use the words group, ring, *and* field *in technical ways, while entomologists have precise definitions for common words like* bug *and* fly. *Although it can be slightly confusing to overload ordinary words like this, it's usually better than the alternative, which is to invent new words. So most specialized fields of study make the same choice, adding crisp, rigorous definitions for words whose common meaning is fuzzy and intuitive.*

The study of formal language is no exception. We use crisp, rigorous definitions for basic terms such as alphabet, string, *and* language.

1.1 Alphabets

Formal language begins with sets of symbols called alphabets:

> An *alphabet* is any finite set of symbols.

A typical alphabet is the set $\Sigma = \{a, b\}$. It is the set of two symbols, *a* and *b*. There is no semantics; we do not ascribe any meaning to *a* or to *b*. They are just symbols, which could as well be 0 and 1, or 空 and 手.

Different applications of formal languages use different alphabets. If you wanted to work with decimal numbers, you might use the alphabet {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. If you wanted to work with the contents of computer memories, you might use the alphabet {0, 1}. If you wanted to work with text files on a computer, you might use one of the standard machine-text alphabets, like ASCII or Unicode.

It is a convention in the study of formal languages, which is followed in this book, to use the first few Latin letters, like *a* and *b*, in most example alphabets. This helps you resist the urge to ascribe meaning to the symbols. When you see a string like 1000, you naturally think of it as representing the decimal number one thousand, while the string *abbb* does not tempt you to make any irrelevant interpretations. Because the symbols in the alphabet are uninterpreted, our results apply to all alphabets.

Another convention followed in the book is to use the symbol Σ to stand for the alphabet currently in use. When it is necessary to manipulate more than one alphabet at once, subscripts are used: Σ_1, Σ_2, and so on.

The empty set {} is a legal alphabet, but not usually an interesting one. (Some people use the special symbol $\varnothing$ for the empty set, but this book just uses {}.)

1.2 Strings

The symbols from an alphabet are put together into strings:

> A *string* is a finite sequence of zero or more symbols.

For example, *abbb* and 010 are strings. In formal languages, unlike most programming languages, strings are not written with quotation marks around them.

The length of a string is the number of symbols in it. To refer to the length of a string, bracket the string with vertical lines: $|abbb| = 4$.

When the symbols in a string are part of a particular alphabet Σ, we say that the string is "a string over Σ." In this usage, the word "over" means "built using symbols from." So, for example, the set of all strings of length 2 over the alphabet $\{a, b\}$ is the set of all strings of length 2 that can be built using the symbols *a* and *b*: {*aa*, *bb*, *ab*, *ba*}.

The special symbol ε is used to represent the string of length zero: the string of no symbols. Note here that ε is not a symbol in any alphabet we will use; it simply stands for the empty string, much as you would write `""` in some programming languages. For example, the set of all strings of length 2 or less over the alphabet $\{a, b\}$ is $\{\varepsilon, a, b, aa, bb, ab, ba\}$. The length of ε is zero, of course: $|\varepsilon| = 0$. Be careful not to confuse the empty set, $\{\}$, with the empty string, ε, and note also that $\{\} \neq \{\varepsilon\}$; the set $\{\}$ contains nothing, while the set $\{\varepsilon\}$ contains one thing: the empty string.

When describing languages and proving things about them, it is sometimes necessary to use variables that stand for strings, such as $x = abbb$. This is a natural concept in programming languages; in Java one writes `String x = "abbb"`, and it is clear from the syntax that `x` is the name of a variable and `abbb` are the characters making up the string to which `x` refers. In the notation used for describing formal languages there is not so much syntax, so you have to rely more on context and on naming conventions. The convention followed in the book is to use the last few Latin letters, like x, y, and z, as string variables, not as symbols in alphabets.

For example, the following definition of concatenation uses string variables.

The *concatenation* of two strings x and y is the string containing all the symbols of x in order, followed by all the symbols of y in order.

To refer to the concatenation of two strings, just write them right next to each other. For example, if $x = abc$ and $y = def$ then the concatenation of x and y is $xy = abcdef$. For any string x, we have $x\varepsilon = \varepsilon x = x$. So ε is the identity element for concatenation of strings, just as 0 is the identity element for addition of natural numbers and just as 1 is for multiplication of natural numbers.

Speaking of numbers, we'll denote the set of natural numbers, $\{0, 1, \ldots\}$, as $\mathcal{N}$. Any natural number $n \in \mathcal{N}$ can be used like an exponent on a string, denoting the concatenation of that string with itself, n times. Thus for any string x,

$x^0 = \varepsilon$ (that is, zero copies of x)
$x^1 = x$
$x^2 = xx$
$x^3 = xxx$

and, in general,

$$x^n = \underbrace{xx \cdots x}_{n \text{ times}}$$

When the alphabet does not contain the symbols (and), which is almost all the time, you can use parentheses to group symbols together for exponentiation. For example, $(ab)^7$ denotes the string containing seven concatenated copies of ab: $(ab)^7 = abababababababab$.

1.3 Languages

A *language* is a set of strings over some fixed alphabet.

A special notation is used to refer to the set of all strings over a given alphabet.

The *Kleene closure* of an alphabet Σ, written Σ^*, is the set of all strings over Σ.

For example, $\{a\}^*$ is the set of all strings of zero or more as: $\{\varepsilon, a, aa, aaa, ...\}$, and $\{a, b\}^*$ is the set of all strings of zero or more symbols, each of which is either a or b: $\{\varepsilon, a, b, aa, bb, ab, ba, aaa, ...\}$. This allows a more compact way of saying that x is a string over the alphabet Σ; we can just write $x \in \Sigma^*$. (Here the symbol $\in$ is used as in standard mathematical notation for sets and means "is an element of.")

Except for the special case of $\Sigma = \{\}$, the Kleene closure of any alphabet is an infinite set. Alphabets are finite sets of symbols, and strings are finite sequences of symbols, but a language is any set of strings—and most interesting languages are in fact infinite sets of strings.

Languages are often described using *set formers*. A set former is written like a set, but it uses the symbol |, read as "such that," to add extra constraints or conditions limiting the elements of the set. For example, $\{x \in \{a, b\}^* \mid |x| \leq 2\}$ is a set former that specifies the set of all strings x over the alphabet $\{a, b\}$, such that the length of x is less than or equal to 2. Thus,

$$\{x \in \{a, b\}^* \mid |x| \leq 2\} = \{\varepsilon, a, b, aa, bb, ab, ba\}$$

$$\{xy \mid x \in \{a, aa\} \text{ and } y \in \{b, bb\}\} = \{ab, abb, aab, aabb\}$$

$$\{x \in \{a, b\}^* \mid x \text{ contains one } a \text{ and two } b\text{s}\} = \{abb, bab, bba\}$$

$$\{a^n b^n \mid n \geq 1\} = \{ab, aabb, aaabbb, aaaabbbb, ...\}$$

That last example shows why set former notation is so useful: it allows you to describe infinite languages without trying to list the strings in them. Unless otherwise constrained, exponents in a set former are assumed to range over all of $\mathcal{N}$. For example,

$$\{(ab)^n\} = \{\varepsilon, ab, abab, ababab, abababab, ...\}$$

$$\{a^n b^n\} = \{\varepsilon, ab, aabb, aaabbb, aaaabbbb, ...\}$$

The set former is a very expressive tool for defining languages, but it lacks precision. In one example above we used an English-language constraint: "x contains one a and two bs." That's allowed—but it makes it easy to write set formers that are vague, ambiguous, or self-contradictory. For that matter, even if we stick to mathematical constraints like "$|x| \leq 2$" and "$n \geq 1$," we still have the problem of explaining exactly which mathematical constraints are

permitted and exactly what they mean. We won't try to do that—we'll continue to use set formers throughout this book, but we'll use them in an informal way, without any formal definition of what they mean. Our quest in subsequent chapters will be to find other tools for defining languages, formal tools that are precise and unambiguous.

Exercises

EXERCISE 1

Restate each of the following languages by listing its contents. For example, if the language is shown as $\{x \in \{a, b\}^* \mid |x| \leq 2\}$, your answer should be $\{\varepsilon, a, b, aa, bb, ab, ba\}$.

a. $\{x \in \{a, b, c\}^* \mid |x| \leq 2\}$
b. $\{xy \mid x \in \{a, aa\}$ and $y \in \{aa, aaa\}\}$
c. $\{\}^*$
d. $\{a^n \mid n$ is less than 20 and divisible by $3\}$
e. $\{a^n b^m \mid n < 2$ and $m < 3\}$

EXERCISE 2

List all strings of length 3 or less in each of the following languages:

a. $\{a\}^*$
b. $\{a, b\}^*$
c. $\{a^n b^n\}$
d. $\{xy \mid x \in \{a\}^*$ and $y \in \{b\}^*\}$
e. $\{a^n b^m \mid n > m\}$

EXERCISE 3

Many applications of formal language theory do associate meanings with the strings in a language. Restate each of the following languages by listing its contents:

a. $\{x \in \{0, 1\}^* \mid x$ is a binary representation, without unnecessary leading zeros, of a number less than $10\}$
b. $\{x \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \mid x$ is a decimal representation, without unnecessary leading zeros, of a prime number less than $20\}$
c. $\{x \in \{a, b, \ldots, z\}^* \mid x$ is a two-letter word in English$\}$

EXERCISE 4

Restate each of the following languages using set former notation.

a. the language of all strings over the alphabet $\{a, b\}$ that begin with a
b. the language of all even-length strings over the alphabet $\{a, b, c\}$
c. the language of strings consisting of zero or more copies of the string ba
d. the language of strings consisting of any number of as followed by the same number of bs, followed by the same number of cs

EXERCISE 5

This exercise illustrates one of the perils of set formers: their use can lead to logical contradictions. This example is related to Russell's Paradox, a famous contradiction in naive set theory discovered by Bertrand Russell in 1901, which led to an important body of work in mathematics.

A set former is itself a string, so one can define languages that contain set formers. For example, the set of all set formers that define the empty set would include such strings as "{}", "{x | $|x| < 0$}", and "{x | x is not equal to x}". (We put quotation marks around set formers considered as strings.)

a. Give an example of a string x that is a set former that defines a language that includes x itself.

b. Give an example of a string y that is a set former that defines a language that does not include y itself.

c. Consider r = "{set formers x | the language defined by x does not include x}". (Your set former y from Part b, above, is an example of a string in the language defined by r.) Show that assuming r is in the language defined by r leads to a contradiction. Show that assuming r is *not* in the language defined by r also leads to a contradiction.

CHAPTER 2

Finite Automata

One way to define a language is to construct an automaton—a kind of abstract computer that takes a string as input and produces a yes-or-no answer. The language it defines is the set of all strings for which it says yes.

The simplest kind of automaton is the finite automaton. The more complicated automata we discuss in later chapters have some kind of unbounded memory to work with; in effect, they will be able to grow to whatever size necessary to handle the input string they are given. But in this chapter, we begin with finite automata, and they have no such power. A finite automaton has a finite memory that is fixed in advance. Whether the input string is long or short, complex or simple, the finite automaton must reach its decision using the same fixed and finite memory.

2.1 Man, Wolf, Goat, and Cabbage

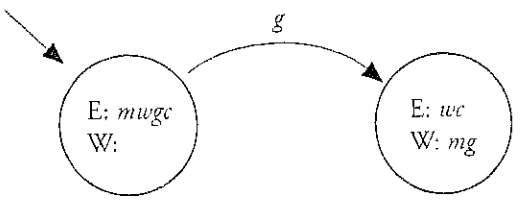
A man is traveling with a wolf, a goat, and a cabbage. He comes to the east bank of a river and wants to cross. A tiny rowboat is available, but it is so small that the man can take at most one of his possessions with him in the boat. If he leaves the goat and the cabbage alone together, the goat will eat the cabbage. If he leaves the wolf and the goat alone together, the wolf will eat the goat. How can the man get to the west bank without losing any of his possessions?

In this old riddle, there are really only four moves to choose from at each step:

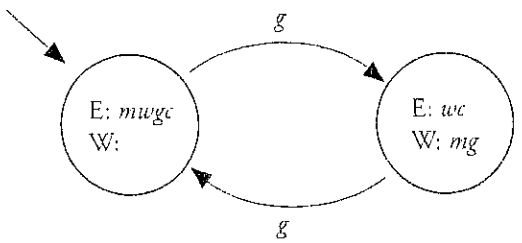
- Man crosses with wolf.
- Man crosses with goat.
- Man crosses with cabbage.
- Man crosses with nothing.

If we abbreviate these moves as *w* (wolf), *g* (goat), *c* (cabbage), and *n* (nothing), then we can represent a sequence of moves as a string over the alphabet {*w*, *g*, *c*, *n*}. A seven-move solution to the problem is represented by the string *gnwgcng*: first cross with the goat, then cross back with nothing, then cross with the wolf, then cross back with the goat (leaving the wolf on the west side), then cross with the cabbage (leaving the goat on the east side again), then cross back with nothing, and finally cross with the goat.

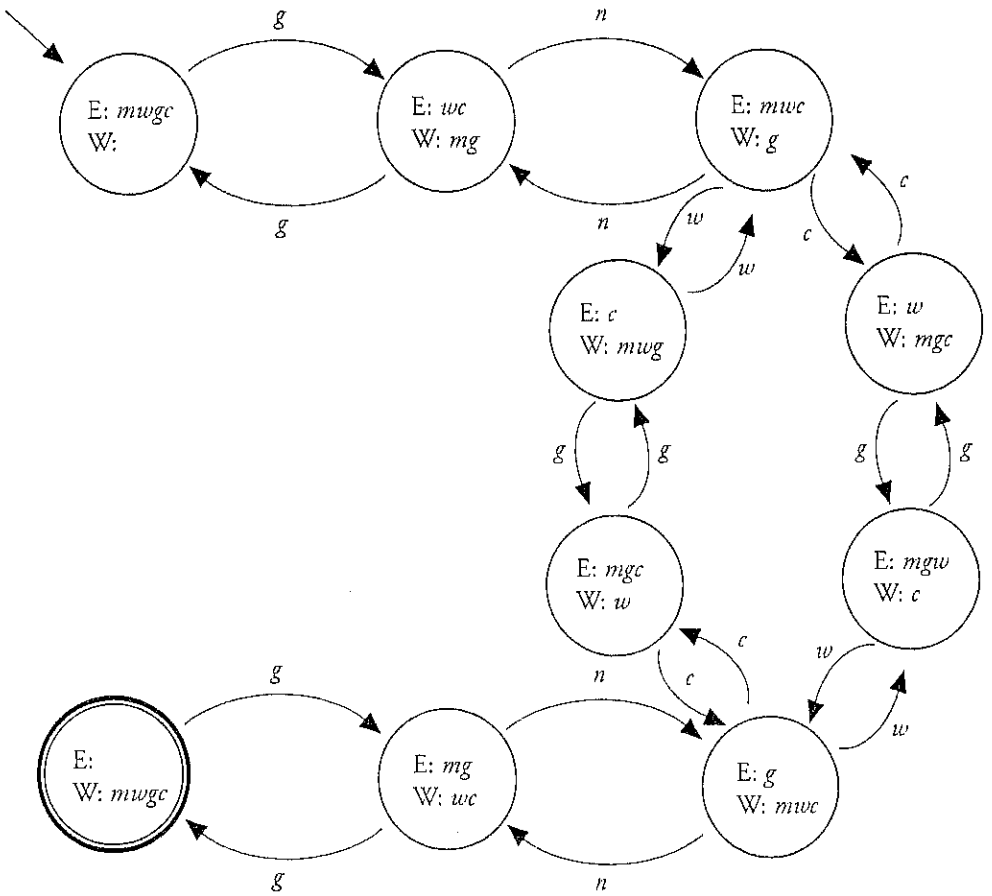
Each move is a *state transition*, taking the puzzle from one state to another state. This can be shown graphically. For instance, this illustration shows that if everything is on the east side of the river, and the *g* move is used, the new state has the wolf and cabbage on the east side and the man and goat on the west side (*m* represents the man):



What happens if the *g* move is used again? Then the man rows back to the east side with the goat, and the whole puzzle returns to the initial state. We include this transition as another arrow:



In this way, a state transition diagram can show all legal transitions and all reachable states, omitting those illegal states where something gets eaten:



An extra arrow indicates the start state, and a double circle indicates the state in which the diagram accepts that the solution is correct.

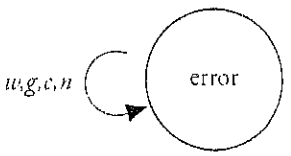
By starting in the start state and making one transition on each symbol in the string *gnwgcng*, you can check that this is indeed a minimum-length solution to the problem. The other minimum-length solution is *gncgwng*. There are longer solutions too, which involve repeated states. For example, the man can just row back and forth with the goat any number of times before completing the solution: *gggggggggncgwng*. In fact, there are infinitely many possible solutions, though only two minimum-length solutions. The language of strings representing legal solutions to the problem can be described with reference to the diagram above: $\{x \in \{w, g, c, n\}^* \mid$ starting in the start state and following the transitions of x ends up in the accepting state$\}$.

2.2 Not Getting Stuck

One problem with the previous diagram is that it gets stuck on many nonsolution strings. What happens, for example, if you follow the previous diagram with the string *gnwn*? The last move, the final *n* in that string, is illegal, since it leaves the goat and the wolf alone on the same side of the river. However, the diagram does not actually say what to do in that case; it just leaves no alternative. The string *gnwn* is not in the language of legal solutions, but the diagram shows this only by getting stuck when the final *n* is reached.

It is a useful property for such a diagram to be fully specified—to show exactly one transition from every state for every symbol in the alphabet, so that it never gets stuck. A fully specified diagram would show what to do on any string, all the way to the end. To fix the previous diagram so that it has this property, we need to give it one additional state: a state representing the fact that an error has been found in the solution.

In the man-wolf-goat-cabbage puzzle, errors are unrecoverable. Once eaten, the goat cannot be restored, so no string beginning with *gnwn* can be a solution to the puzzle, no matter what else follows. The error state needed for the diagram looks like this:



The transition arrow is labeled with all the symbols in the alphabet and simply returns to the same state. This shows that the state is a trap: once reached, it cannot be departed from. All the transitions that were unspecified in the original diagram can now be shown as explicit transitions to this error state. The resulting fully specified diagram is shown on the following page.

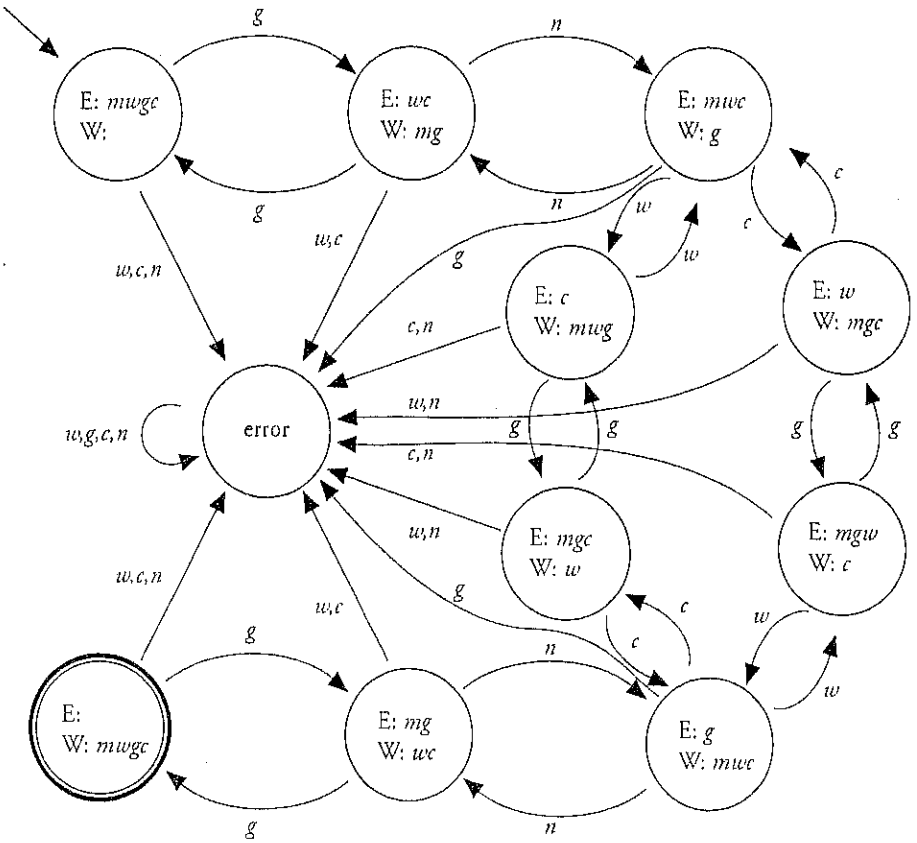
This more elaborate diagram can handle any string over the alphabet {*w*, *g*, *c*, *n*}. By starting in the start state and following the transitions on each character in the string, you end up in a final state. If that final state is the accepting state, the string is a solution; if it is any other state, the string is not a solution.

2.3 Deterministic Finite Automata

The man-wolf-goat-cabbage diagram is an example of a deterministic finite automaton (or DFA), a kind of abstract machine for defining languages.

Here is an informal definition of a DFA; the formal definition comes a little later.

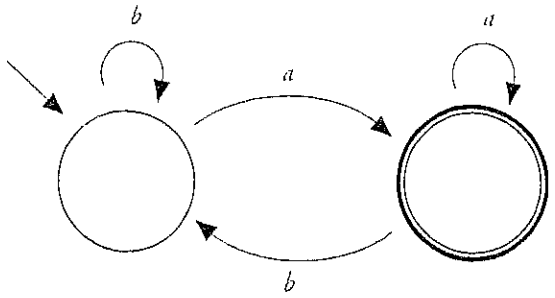
(Informal) A *deterministic finite automaton* or *DFA* is a diagram with a finite number of *states* represented by circles. An arrow points to one of the states, the unique *start state*; double circles mark any number of the states as *accepting*



states. From every state, for every symbol in the alphabet Σ, there is exactly one arrow labeled with that symbol going to another state (or back to the same state).

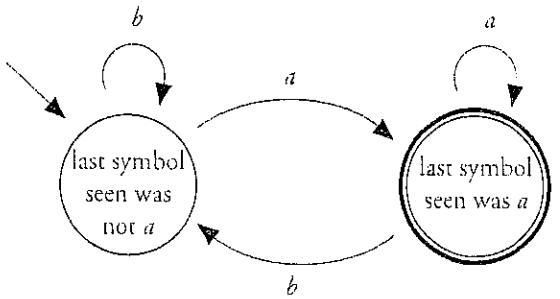
DFAs define languages, just as our man-wolf-goat-cabbage automaton defined the language of solutions to the puzzle. Given any string over its alphabet, a DFA can read the string and follow its state-to-state transitions. At the end of the string it is either in an accepting state or not. If it is an accepting state, we say that the machine accepts the string; otherwise we say that it rejects the string. The language defined by a DFA M is just the set of strings in Σ^* accepted by M.

For example, here is a DFA for $\{xa \mid x \in \{a, b\}^*\}$—that is, the language of strings over the alphabet $\{a, b\}$ that end in a:



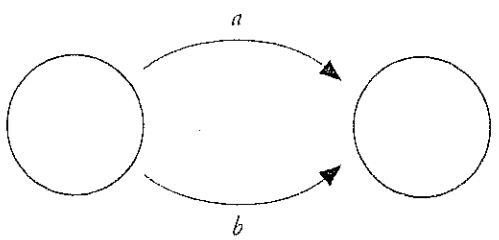
2.4

This diagram meets all the requirements for a DFA: it has a finite set of states with a single start state, and it has exactly one transition from every state on every symbol in the alphabet $\{a, b\}$. Unlike the man-wolf-goat-cabbage diagram, the states in this DFA are unlabeled. It does not hurt to label the states, if they have some meaning that you want to call attention to, or if you need to refer to the states by name for some reason. We could, for example, draw the DFA above like this:

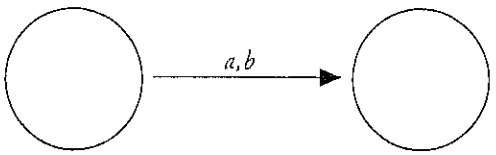


But the labels of states (unlike the labels on the arrows) have no impact on the behavior of the machine. They are rather like comments in a computer program.

There is one important convention you should know about before attempting the exercises at the end of the chapter. If a DFA has more than one arrow with the same source and destination states, like this:



then we usually draw it more compactly as a single arrow with a list of symbols as its label, like this:



These two forms mean exactly the same thing: they both show a state transition that the DFA can make on either a or b.

2.4 The 5-Tuple

A diagram of a DFA is reasonably easy for people to look at and understand. Mathematically, however, a diagram of a DFA is not very useful. To study DFAs with mathematical rigor—to prove things about them and the languages they accept—it is important to have a more abstract definition.

A DFA has five key parts, so we describe it mathematically as a 5-tuple.

A DFA M is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

Q is the finite set of states
Σ is the alphabet (that is, a finite set of symbols)
$\delta \in (Q \times \Sigma \to Q)$ is the transition function
$q_0 \in Q$ is the start state
$F \subseteq Q$ is the set of accepting states

The first part of a DFA is the set Q of states; this corresponds to the set of states drawn as circles in a DFA diagram. The different states in Q are usually referred to as q_i for different values of i; note that the definition implies that there must be at least one state in Q, namely the start state q_0.

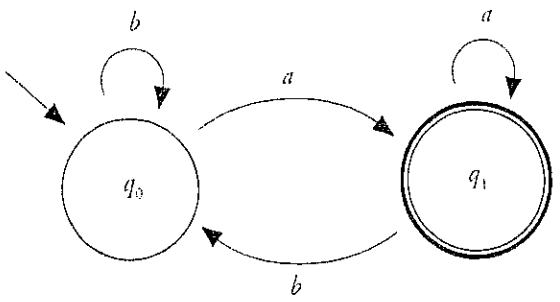
The alphabet Σ is the second part of a DFA. When you draw a diagram for a DFA, the alphabet is implicit; you can tell what it must be by looking at the labels on the arrows. For the formal definition of a DFA, however, the alphabet is explicit.

The third part of a DFA is the transition function δ. The definition says $\delta \in (Q \times \Sigma \to Q)$. This is the mathematical way of giving the type of the function, and it says that δ takes two inputs, a state from Q and a symbol from Σ, and produces a single output, another state in Q. If the DFA is in state q_i reading symbol a, then $\delta(q_i, a)$ is the state to go to next. Thus the transition function encodes all the information about the arrows in a DFA diagram.

The fourth part of the DFA is the start state q_0. There must be exactly one start state. The fifth and final part of the DFA is F, a subset of Q identifying the accepting states, those that are double-circled in the DFA diagram. There is nothing in the definition that prevents F from being $\{\}$—in that case there would be no accepting states, so the machine would reject all strings and the language would be $\{\}$. There is also nothing that prevents F from

being equal to Q—in that case all states would be accepting states, so the machine would accept all strings and the language would be Σ^*.

Consider for example this DFA for the language $\{xa \mid x \in \{a, b\}^*\}$:



Formally, the DFA shown is $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $F = \{q_1\}$, and the transition function δ is

$$\delta(q_0, a) = q_1$$
$$\delta(q_0, b) = q_0$$
$$\delta(q_1, a) = q_1$$
$$\delta(q_1, b) = q_0$$

A DFA is a 5-tuple: it is a mathematical structure with five parts given in order. The names given to those five parts in the definition above—Q, Σ, and so forth—are conventional, but the machine M could just as well have been defined as $M = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_1\})$, without naming all the parts. The important thing is to specify the five parts in the required order.

2.5 The Language Accepted by a DFA

A DFA's transition function δ says how the DFA makes a single move: one state-to-state transition, reading one symbol from an input string. Intuitively, we understand that the DFA makes a sequence of these moves, one for each symbol in the string, until the end of the string is reached. To capture this intuition about making a sequence of moves, an extended transition function δ^* can be defined.

Where the δ function takes a single symbol as its second parameter, the δ^* function takes a string of zero or more symbols. So for any string $x \in \Sigma^*$ and any state $q_i \in Q$, $\delta^*(q_i, x)$ should be the state the DFA ends up in, starting from q_i and making the sequence of δ-transitions on the symbols in x. There is a simple recursive definition for this δ^*:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, xa) = \delta(\delta^*(q, x), a)$$

The first line in the definition is the base case. It just says that when the input is the empty string, no transition is made: the final state is the same as the initial state. The second line

in the definition is the recursive case. It says that for a string xa, where x is any string and a is the final symbol in the string, you first make all the transitions for x, and then one final δ-transition using the final symbol a.

The definition above constructs an extended transition function δ^* from any basic transition function δ. Using these extended transition functions, it is possible to say exactly what it means for a DFA to accept a string:

> A string $x \in \Sigma^*$ is accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$ if and only if $\delta^*(q_0, x) \in F$.

That is, a string x is accepted if and only if the DFA, started in its start state and taken through the sequence of transitions on the symbols in x, ends up in one of its accepting states. We can also define $L(M)$, the language accepted by a DFA M:

> For any DFA $M = (Q, \Sigma, \delta, q_0, F)$, $L(M)$ denotes the language accepted by M, which is $L(M) = \{x \in \Sigma^* \mid \delta^*(q_0, x) \in F\}$.

A language that can be recognized by a DFA is called a *regular* language.

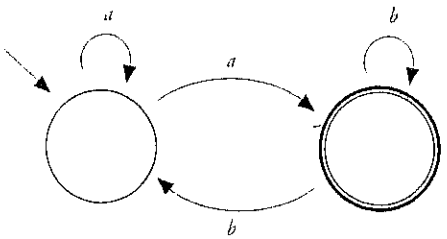
> A regular language is one that is $L(M)$ for some DFA M.

A direct way to prove that a language is regular is to give a DFA that accepts it. To prove that a language is *not* regular is generally much harder, and this is a topic that will be addressed later in this book.

Exercises

EXERCISE 1

For each of the following strings, say whether it is in the language accepted by this DFA:



a. *b*
b. ε
c. *ab*
d. *abba*
e. *ababaaaab*