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 {ε, *a*, *b*, *aa*, *bb*, *ab*, *ba*}. The length of ε is zero, of course: |ε| = 0. Be careful not to confuse the empty set, {}, with the empty string, ε, and note also that {} ≠ {ε}; the set {} contains nothing, while the set {ε} 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*ε = ε*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, …}, 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 = ε (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*: $(ab)^7 = ababababababab$.

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: {ε, *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*: {ε, *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 ∈ 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,

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

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

$\{x \in \{a, b\}^* \mid x$ contains one *a* and two *b*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 *b*s." 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. $\{x \in \{a, b, ..., z\}^* \mid x$ 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.