Chapter 2

Scanners

■ CHAPTER OVERVIEW

The scanner's task is to transform a stream of characters into a stream of words in the input language. Each word must be classified into a syntactic category, or "part of speech." The scanner is the only pass in the compiler to touch every character in the input program. Compiler writers place a premium on speed in scanning, in part because the scanner's input is larger, in some measure, than that of any other pass, and, in part, because highly efficient techniques are easy to understand and to implement.

This chapter introduces regular expressions, a notation used to describe the valid words in a programming language. It develops the formal mechanisms to generate scanners from regular expressions, either manually or automatically.

Keywords: Scanner, Finite Automaton, Regular Expression, Fixed Point

2.1 INTRODUCTION

Scanning is the first stage of a three-part process that the compiler uses to understand the input program. The scanner, or lexical analyzer, reads a stream of characters and produces a stream of words. It aggregates characters to form words and applies a set of rules to determine whether or not each word is legal in the source language. If the word is valid, the scanner assigns it a syntactic category, or part of speech.

The scanner is the only pass in the compiler that manipulates every character of the input program. Because scanners perform a relatively simple task, grouping characters together to form words and punctuation in the source language, they lend themselves to fast implementations. Automatic tools for scanner generation are common. These tools process a mathematical

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00002-5

description of the language's lexical syntax and produce a fast recognizer. Alternatively, many compilers use hand-crafted scanners; because the task is simple, such scanners can be fast and robust.

Conceptual Roadmap

This chapter describes the mathematical tools and programming techniques that are commonly used to construct scanners—both generated scanners and hand-crafted scanners. The chapter begins, in Section 2.2, by introducing a model for *recognizers*, programs that identify words in a stream of characters. Section 2.3 describes *regular expressions*, a formal notation for specifying syntax. In Section 2.4, we show a set of constructions to convert a regular expression into a recognizer. Finally, in Section 2.5 we present three different ways to implement a scanner: a table-driven scanner, a direct-coded scanner, and a hand-coded approach.

Recognizer
a program that identifies specific words in a stream of characters

Both generated and hand-crafted scanners rely on the same underlying techniques. While most textbooks and courses advocate the use of generated scanners, most commercial compilers and open-source compilers use hand-crafted scanners. A hand-crafted scanner can be faster than a generated scanner because the implementation can optimize away a portion of the overhead that cannot be avoided in a generated scanner. Because scanners are simple and they change infrequently, many compiler writers deem that the performance gain from a hand-crafted scanner outweighs the convenience of automated scanner generation. We will explore both alternatives.

Overview

A compiler's scanner reads an input stream that consists of characters and produces an output stream that contains words, each labelled with its *syntactic category*—equivalent to a word's part of speech in English. To accomplish this aggregation and classification, the scanner applies a set of rules that describe the lexical structure of the input programming language, sometimes called its *microsyntax*. The microsyntax of a programming language specifies how to group characters into words and, conversely, how to separate words that run together. (In the context of scanning, we consider punctuation marks and other symbols as words.)

Syntactic category
a classification of words according to their grammatical usage

Microsyntax
the lexical structure of a language

Western languages, such as English, have simple microsyntax. Adjacent alphabetic letters are grouped together, left to right, to form a word. A blank space terminates a word, as do most nonalphabetic symbols. (The word-building algorithm can treat a hyphen in the midst of a word as if it were an alphabetic character.) Once a group of characters has been aggregated together to form a potential word, the word-building algorithm can determine its validity with a dictionary lookup.

Most programming languages have equally simple microsyntax. Characters are aggregated into words. In most languages, blanks and punctuation marks terminate a word. For example, Algol and its descendants define an *identifier* as a single alphabetic character followed by zero or more alphanumeric characters. The identifier ends with the first nonalphanumeric character. Thus, `fee` and `f1e` are valid identifiers, but `12fum` is not. Notice that the set of valid words is specified by rules rather than by enumeration in a dictionary.

In a typical programming language, some words, called *keywords* or *reserved words*, match the rule for an identifier but have special meanings. Both `while` and `static` are keywords in both C and Java. Keywords (and punctuation marks) form their own syntactic categories. Even though `static` matches the rule for an identifier, the scanner in a C or Java compiler would undoubtedly classify it into a category that has only one element, the keyword `static`. To recognize keywords, the scanner can either use dictionary lookup or encode the keywords directly into its microsyntax rules.

Keyword
a word that is reserved for a particular syntactic purpose and, thus, cannot be used as an identifier

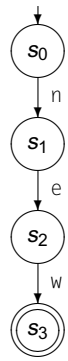
The simple lexical structure of programming languages lends itself to efficient scanners. The compiler writer starts from a specification of the language's microsyntax. She either encodes the microsyntax into a notation accepted by a scanner generator, which then constructs an executable scanner, or she uses that specification to build a hand-crafted scanner. Both generated and hand-crafted scanners can be implemented to require just **O**(1) time per character, so they run in time proportional to the number of characters in the input stream.

2.2 RECOGNIZING WORDS

The simplest explanation of an algorithm to recognize words is often a character-by-character formulation. The structure of the code can provide some insight into the underlying problem. Consider the problem of recognizing the keyword `new`. Assuming the presence of a routine *`NextChar`* that returns the next character, the code might look like the fragment shown in Figure 2.1. The code tests for `n` followed by `e` followed by `w`. At each step, failure to match the appropriate character causes the code to reject the string and "try something else." If the sole purpose of the program was to recognize the word `new`, then it should print an error message or return failure. Because scanners rarely recognize only one word, we will leave this "error path" deliberately vague at this point.

The code fragment performs one test per character. We can represent the code fragment using the simple transition diagram shown to the right of the code. The transition diagram represents a recognizer. Each circle represents an abstract state in the computation. Each state is labelled for convenience.

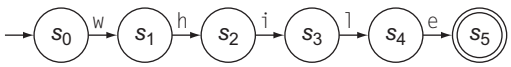
```
c ← NextChar();
if (c = 'n')
   then begin;
      c ← NextChar();
      if (c = 'e')
          then begin;
            c ← NextChar();
            if (c = 'w')
                then report success;
                else try something else;
          end;
          else try something else;
   end;
   else try something else;
```



■ **FIGURE 2.1** Code Fragment to Recognize "new".

The initial state, or start state, is s_0. We will always label the start state as s_0. State s_3 is an accepting state; the recognizer reaches s_3 only when the input is `new`. Accepting states are drawn with double circles, as shown in the margin. The arrows represent transitions from state to state based on the input character. If the recognizer starts in s_0 and reads the characters `n`, `e`, and `w`, the transitions take us to s_3. What happens on any other input, such as `n`, `o`, and `t`? The `n` takes the recognizer to s_1. The `o` does not match the edge leaving s_1, so the input word is not `new`. In the code, cases that do not match `new` *try something else*. In the recognizer, we can think of this action as a transition to an error state. When we draw the transition diagram of a recognizer, we usually omit transitions to the error state. Each state has a transition to the error state on each unspecified input.

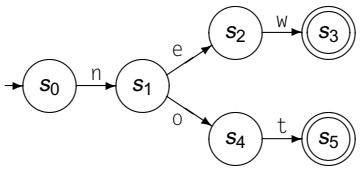
Using this same approach to build a recognizer for `while` would produce the following transition diagram:



If it starts in s_0 and reaches s_5, it has identified the word `while`. The corresponding code fragment would involve five nested *if-then-else* constructs.

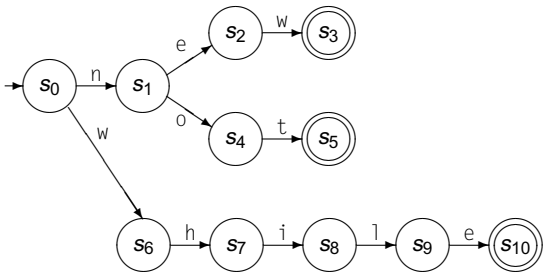
To recognize multiple words, we can create multiple edges that leave a given state. (In the code, we would begin to elaborate the *do something else* paths.)

One recognizer for both `new` and `not` might be



The recognizer uses a common test for `n` that takes it from s_0 to s_1, denoted $s_0 \xrightarrow{n} s_1$. If the next character is `e`, it takes the transition $s_1 \xrightarrow{e} s_2$. If, instead, the next character is `o`, it makes the move $s_1 \xrightarrow{o} s_4$. Finally, a `w` in s_2, causes the transition $s_2 \xrightarrow{w} s_3$, while a `t` in s_4 produces $s_4 \xrightarrow{t} s_5$. State s_3 indicates that the input was `new` while s_5 indicates that it was `not`. The recognizer takes one transition per input character.

We can combine the recognizer for `new` or `not` with the one for `while` by merging their initial states and relabeling all the states.



State s_0 has transitions for `n` and `w`. The recognizer has three accepting states, s_3, s_5, and s_{10}. If any state encounters an input character that does not match one of its transitions, the recognizer moves to an error state.

2.2.1 A Formalism for Recognizers

Transition diagrams serve as abstractions of the code that would be required to implement them. They can also be viewed as formal mathematical objects, called *finite automata*, that specify recognizers. Formally, a finite automaton (FA) is a five-tuple $(S, \Sigma, \delta, s_0, S_A)$, where

Finite automaton
a formalism for recognizers that has a finite set of states, an alphabet, a transition function, a start state, and one or more accepting states

- S is the finite set of states in the recognizer, along with an error state s_e.
- Σ is the finite alphabet used by the recognizer. Typically, Σ is the union of the edge labels in the transition diagram.

- $\delta(s,c)$ is the recognizer's transition function. It maps each state $s \in S$ and each character $c \in \Sigma$ into some next state. In state s_i with input character c, the FA takes the transition $s_i \xrightarrow{c} \delta(s_i,c)$.
- $s_0 \in S$ is the designated start state.
- S_A is the set of accepting states, $S_A \subseteq S$. Each state in S_A appears as a double circle in the transition diagram.

As an example, we can cast the FA for *new* or *not* or *while* in the formalism as follows:

$$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$$

$$\Sigma = \{\texttt{e}, \texttt{h}, \texttt{i}, \texttt{l}, \texttt{n}, \texttt{o}, \texttt{t}, \texttt{w}\}$$

$$\delta = \left\{ \begin{array}{lllll} s_0 \xrightarrow{\texttt{n}} s_1, & s_0 \xrightarrow{\texttt{w}} s_6, & s_1 \xrightarrow{\texttt{e}} s_2, & s_1 \xrightarrow{\texttt{o}} s_4, & s_2 \xrightarrow{\texttt{w}} s_3, \\ s_4 \xrightarrow{\texttt{t}} s_5, & s_6 \xrightarrow{\texttt{h}} s_7, & s_7 \xrightarrow{\texttt{i}} s_8, & s_8 \xrightarrow{\texttt{l}} s_9, & s_9 \xrightarrow{\texttt{e}} s_{10} \end{array} \right\}$$

$$s_0 = s_0$$

$$S_A = \{s_3, s_5, s_{10}\}$$

For all other combinations of state s_i and input character c, we define $\delta(s_i,c) = s_e$, where s_e is the designated error state. This quintuple is equivalent to the transition diagram; given one, we can easily re-create the other. The transition diagram is a picture of the corresponding FA.

An FA accepts a string x if and only if, starting in s_0, the sequence of characters in the string takes the FA through a series of transitions that leaves it in an accepting state when the entire string has been consumed. This corresponds to our intuition for the transition diagram. For the string `new`, our example recognizer runs through the transitions $s_0 \xrightarrow{\texttt{n}} s_1$, $s_1 \xrightarrow{\texttt{e}} s_2$, and $s_2 \xrightarrow{\texttt{w}} s_3$. Since $s_3 \in S_A$, and no input remains, the FA accepts `new`. For the input string `nut`, the behavior is different. On `n`, the FA takes $s_0 \xrightarrow{\texttt{n}} s_1$. On `u`, it takes $s_1 \xrightarrow{\texttt{u}} s_e$. Once the FA enters s_e, it stays in s_e until it exhausts the input stream.

More formally, if the string x is composed of characters $x_1\, x_2\, x_3 \ldots x_n$, then the FA $(S, \Sigma, \delta, s_0, S_A)$ accepts x if and only if

$$\delta(\delta(\ldots\delta(\delta(\delta(s_0, x_1), x_2), x_3)\ldots, x_{n-1}), x_n) \in S_A.$$

Intuitively, this definition corresponds to a repeated application of δ to a pair composed of some state $s \in S$ and an input symbol x_i. The base case, $\delta(s_0, x_1)$, represents the FA's initial transition, out of the start state, s_0, on the character x_1. The state produced by $\delta(s_0, x_1)$ is then used as input, along with x_2, to δ to produce the next state, and so on, until all the input has been

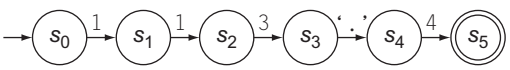
consumed. The result of the final application of δ is, again, a state. If that state is an accepting state, then the FA accepts $x_1\, x_2\, x_3 \ldots x_n$.

Two other cases are possible. The FA might encounter an error while processing the string—that is, some character x_j might take it into the error state s_e. This condition indicates a lexical error; the string $x_1\, x_2\, x_3 \ldots x_j$ is not a valid prefix for any word in the language accepted by the FA. The FA can also discover an error by exhausting its input and terminating in a nonaccepting state other than s_e. In this case, the input string is a proper prefix of some word accepted by the FA. Again, this indicates an error. Either kind of error should be reported to the end user.

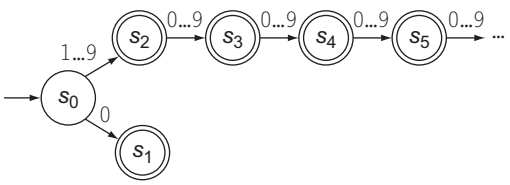
In any case, notice that the FA takes one transition for each input character. Assuming that we can implement the FA efficiently, we should expect the recognizer to run in time proportional to the length of the input string.

2.2.2 Recognizing More Complex Words

The character-by-character model shown in the original recognizer for `not` extends easily to handle arbitrary collections of fully specified words. How could we recognize a number with such a recognizer? A specific number, such as 113.4, is easy.



To be useful, however, we need a transition diagram (and the corresponding code fragment) that can recognize any number. For simplicity's sake, let's limit the discussion to unsigned integers. In general, an integer is either zero, or it is a series of one or more digits where the first digit is from one to nine, and the subsequent digits are from zero to nine. (This definition rules out leading zeros.) How would we draw a transition diagram for this definition?



The transition $s_0 \xrightarrow{0} s_1$ handles the case for zero. The other path, from s_0 to s_2, to s_3, and so on, handles the case for an integer greater than zero. This path, however, presents several problems. First, it does not end, violating the stipulation that S is finite. Second, all of the states on the path beginning with s_2 are equivalent, that is, they have the same labels on their output transitions and they are all accepting states.

```
char ← NextChar( );
state ← s0 ;

while (char ≠ eof and state ≠ se) do
   state ← δ(state,char);
   char ← NextChar( );
end;

if (state ∈ SA)
    then report acceptance;
    else report failure;
```

$$S = \{s_0, s_1, s_2, s_e\}$$

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\delta = \left\{ \begin{array}{ll} s_0 \xrightarrow{0} s_1, & s_0 \xrightarrow{1\text{-}9} s_2 \\ s_2 \xrightarrow{0\text{-}9} s_2, & s_1 \xrightarrow{0\text{-}9} s_e \end{array} \right\}$$

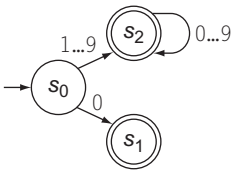
$$S_A = \{s_1, s_2\}$$

■ **FIGURE 2.2** A Recognizer for Unsigned Integers.

Lexeme
the actual text for a word recognized by an FA

This FA recognizes a class of strings with a common property: they are all unsigned integers. It raises the distinction between the class of strings and the text of any particular string. The class "unsigned integer" is a syntactic category, or part of speech. The text of a specific unsigned integer, such as `113`, is its *lexeme*.

We can simplify the FA significantly if we allow the transition diagram to have cycles. We can replace the entire chain of states beginning at s_2 with a single transition from s_2 back to itself:



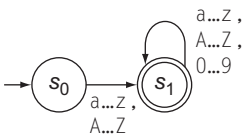
This cyclic transition diagram makes sense as an FA. From an implementation perspective, however, it is more complex than the acyclic transition diagrams shown earlier. We cannot translate this directly into a set of nested *if-then-else* constructs. The introduction of a cycle in the transition graph creates the need for cyclic control flow. We can implement this with a *while* loop, as shown in Figure 2.2. We can specify δ efficiently using a table:

δ	0	1	2	3	4	5	6	7	8	9	Other
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

Changing the table allows the same basic code skeleton to implement other recognizers. Notice that this table has ample opportunity for compression.

The columns for the digits `1` through `9` are identical, so they could be represented once. This leaves a table with three columns: `0`, `1 . . . 9`, and *other*. Close examination of the code skeleton shows that it reports failure as soon as it enters s_e, so it never references that row of the table. The implementation can elide the entire row, leaving a table with just three rows and three columns.

We can develop similar FAs for signed integers, real numbers, and complex numbers. A simplified version of the rule that governs identifier names in Algol-like languages, such as C or Java, might be: *an identifier consists of an alphabetic character followed by zero or more alphanumeric characters*. This definition allows an infinite set of identifiers, but can be specified with the simple two-state FA shown to the left. Many programming languages extend the notion of "alphabetic character" to include designated special characters, such as the underscore.



FAs can be viewed as specifications for a recognizer. However, they are not particularly concise specifications. To simplify scanner implementation, we need a concise notation for specifying the lexical structure of words, and a way of turning those specifications into an FA and into code that implements the FA. The remaining sections of this chapter develop precisely those ideas.

SECTION REVIEW

A character-by-character approach to scanning leads to algorithmic clarity. We can represent character-by-character scanners with a transition diagram; that diagram, in turn, corresponds to a finite automaton. Small sets of words are easily encoded in acyclic transition diagrams. Infinite sets, such as the set of integers or the set of identifiers in an Algol-like language, require cyclic transition diagrams.

Review Questions

Construct an FA to accept each of the following languages:

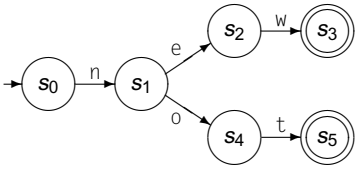
1. A six-character identifier consisting of an alphabetic character followed by zero to five alphanumeric characters
2. A string of one or more pairs, where each pair consists of an open parenthesis followed by a close parenthesis
3. A Pascal comment, which consists of an open brace, {, followed by zero or more characters drawn from an alphabet, Σ, followed by a close brace, }

2.3 REGULAR EXPRESSIONS

The set of words accepted by a finite automaton, $\mathcal{F}$, forms a language, denoted $L(\mathcal{F})$. The transition diagram of the FA specifies, in precise detail, that language. It is not, however, a specification that humans find intuitive. For any FA, we can also describe its language using a notation called a *regular expression* (RE). The language described by an RE is called a *regular language*.

Regular expressions are equivalent to the FAs described in the previous section. (We will prove this with a construction in Section 2.4.) Simple recognizers have simple RE specifications.

- The language consisting of the single word `new` can be described by an RE written as *new*. Writing two characters next to each other implies that they are expected to appear in that order.
- The language consisting of the two words `new` or `while` can be written as *new* or *while*. To avoid possible misinterpretation of *or*, we write this using the symbol | to mean *or*. Thus, we write the RE as *new* | *while*.
- The language consisting of `new` or `not` can be written as *new* | *not*. Other REs are possible, such as *n(ew* | *ot)*. Both REs specify the same pair of words. The RE *n(ew* | *ot)* suggests the structure of the FA that we drew earlier for these two words.



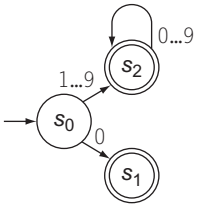
To make this discussion concrete, consider some examples that occur in most programming languages. Punctuation marks, such as colons, semicolons, commas, and various brackets, can be represented by their character representations. Their REs have the same "spelling" as the punctuation marks themselves. Thus, the following REs might occur in the lexical specification for a programming language:

: ; ? => ( ) { } [ ]

Similarly, keywords have simple REs.

if while this integer instanceof

To model more complex constructs, such as integers or identifiers, we need a notation that can capture the essence of the cyclic edge in an FA.



The FA for an unsigned integer, shown at the left, has three states: an initial state s_0, an accepting state s_1 for the unique integer zero, and another accepting state s_2 for all other integers. The key to this FA's power is the transition from s_2 back to itself that occurs on each additional digit. State s_2 folds the specification back on itself, creating a rule to derive a new unsigned integer from an existing one: add another digit to the right end of the existing number. Another way of stating this rule is: *an unsigned integer is either a zero, or a nonzero digit followed by zero or more digits.* To capture the essence of this FA, we need a notation for this notion of "zero or more occurrences" of an RE. For the RE x, we write this as x^*, with the meaning "zero or more occurrences of x." We call the * operator *Kleene closure*, or *closure* for short. Using the closure operator, we can write an RE for this FA:

$$0 \mid (1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)\ (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*.$$

2.3.1 Formalizing the Notation

To work with regular expressions in a rigorous way, we must define them more formally. An RE describes a set of strings over the characters contained in some alphabet, Σ, augmented with a character ϵ that represents the empty string. We call the set of strings a *language*. For a given RE, r, we denote the language that it specifies as $L(r)$. An RE is built up from three basic operations:

1. *Alternation* The alternation, or union, of two sets of strings, R and S, denoted $R \mid S$, is $\{x \mid x \in R \text{ or } x \in S\}$.
2. *Concatenation* The concatenation oftwo sets R and S, denoted RS, contains all strings formed by prepending an element of R onto one from S, or $\{xy \mid x \in R \text{ and } y \in S\}$.
3. *Closure* The Kleene closure of a set R, denoted R^*, is $\bigcup_{i=0}^{\infty} R^i$. This is just the union of the concatenations of R with itself, zero or more times.

For convenience, we sometimes use a notation for *finite closure*. The notation R^i denotes from one to i occurrences of R. A finite closure can be always be replaced with an enumeration of the possibilities; for example, R^3 is just $(R \mid RR \mid RRR)$. The *positive closure*, denoted R^+, is just RR^* and consists of one or more occurrences of R. Since all these closures can be rewritten with the three basic operations, we ignore them in the discussion that follows.

Finite closure
For any integer i, the RE R^i designates one to i occurrences of R.

Positive closure
The RE R^+ denotes one or more occurrences of R, often written as $\bigcup_{i=1}^{\infty} R^i$.

Using the three basic operations, alternation, concatenation, and Kleene closure, we can define the set of REs over an alphabet Σ as follows:

1. If $a \in \Sigma$, then a is also an RE denoting the set containing only a.
2. If r and s are REs, denoting sets $L(r)$ and $L(s)$, respectively, then

REGULAR EXPRESSIONS IN VIRTUAL LIFE

Regular expressions are used in many applications to specify patterns in character strings. Some of the early work on translating REs into code was done to provide a flexible way of specifying strings in the "find" command of a text editor. From that early genesis, the notation has crept into many different applications.

Unix and other operating systems use the asterisk as a wildcard to match substrings against file names. Here, * is a shorthand for the RE Σ^*, specifying zero or more characters drawn from the entire alphabet of legal characters. (Since few keyboards have a Σ key, the shorthand has stayed with us.) Many systems use **?** as a wildcard that matches a single character.

The `grep` family of tools, and their kin in non-Unix systems, implement regular expression pattern matching. (In fact, `grep` is an acronym for global regular-expression pattern match and print.)

Regular expressions have found widespread use because they are easily written and easily understood. They are one of the techniques of choice when a program must recognize a fixed vocabulary. They work well for languages that fit within their limited rules. They are easily translated into an executable form, and the resulting recognizer is fast.

$r \mid s$ is an RE denoting the union, or alternation, of $L(r)$ and $L(s)$,
rs is an RE denoting the concatenation of $L(r)$ and $L(s)$, respectively, and
r^* is an RE denoting the Kleene closure of $L(r)$.

3. ϵ is an RE denoting the set containing only the empty string.

To eliminate any ambiguity, parentheses have highest precedence, followed by closure, concatenation, and alternation, in that order.

As a convenient shorthand, we will specify ranges of characters with the first and the last element connected by an ellipsis, "...". To make this abbreviation stand out, we surround it with a pair of square brackets. Thus, [*0...9*] represents the set of decimal digits. It can always be rewritten as (*0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*).

2.3.2 Examples

The goal of this chapter is to show how we can use formal techniques to automate the construction of high-quality scanners and how we can encode the microsyntax of programming languages into that formalism. Before proceeding further, some examples from real programming languages are in order.

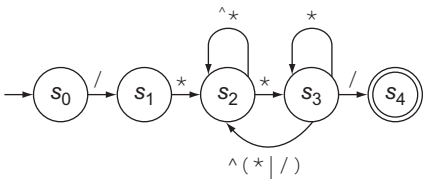
1. The simplified rule given earlier for identifiers in Algol-like languages, an alphabetic character followed by zero or more alphanumeric characters, is just $([A \ldots Z] \mid [a \ldots z])\ ([A \ldots Z] \mid [a \ldots z] \mid [0 \ldots 9])^*$. Most languages also allow a few special characters, such as the underscore (_), the percent sign (%), or the ampersand (&), in identifiers.
   If the language limits the maximum length of an identifier, we can use the appropriate finite closure. Thus, identifiers limited to six characters might be specified as $([A \ldots Z] \mid [a \ldots z])\ ([A \ldots Z] \mid [a \ldots z] \mid [0 \ldots 9])^5$. If we had to write out the full expansion of the finite closure, the RE would be much longer.
2. An unsigned integer can be described as either zero or a nonzero digit followed by zero or more digits. The RE $0 \mid [1 \ldots 9]\,[0 \ldots 9]^*$ is more concise. In practice, many implementations admit a larger class of strings as integers, accepting the language $[0 \ldots 9]^+$.
3. Unsigned real numbers are more complex than integers. One possible RE might be $(0 \mid [1 \ldots 9]\,[0 \ldots 9]^*)\ (\epsilon \mid .\,[0 \ldots 9]^*)$ The first part is just the RE for an integer. The rest generates either the empty string or a decimal point followed by zero or more digits.
   Programming languages often extend real numbers to scientific notation, as in $(0 \mid [1 \ldots 9]\,[0 \ldots 9]^*)\ (\epsilon \mid .\,[0 \ldots 9]^*)\ E\ (\epsilon \mid + \mid -)$ $(0 \mid [1 \ldots 9]\,[0 \ldots 9]^*)$.
   This RE describes a real number, followed by an E, followed by an integer to specify the exponent.
4. Quoted character strings have their own complexity. In most languages, any character can appear inside a string. While we can write an RE for strings using only the basic operators, it is our first example where a complement operator simplifies the RE. Using complement, a character string in C or Java can be described as "(^")* ".
   C and C++ do not allow a string to span multiple lines in the source code—that is, if the scanner reaches the end of a line while inside a string, it terminates the string and issues an error message. If we represent newline with the escape sequence `\n`, in the C style, then the RE "(^(" | `\n`))* " will recognize a correctly formed string and will take an error transition on a string that includes a newline.
5. Comments appear in a number of forms. C++ and Java offer the programmer two ways of writing a comment. The delimiter `//` indicates a comment that runs to the end of the current input line. The RE for this style of comment is straightforward: `//` (^`\n`)* `\n`, where `\n` represents the newline character.
   Multiline comments in C, C++, and Java begin with the delimiter `/*` and end with `*/`. If we could disallow * in a comment, the RE would be

Complement operator
The notation ^{c} specifies the set $\{\Sigma - c\}$, the complement of c with respect to Σ.

Complement has higher precedence than *, |, or $^+$.

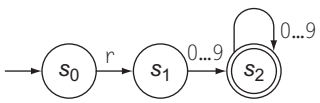
Escape sequence
Two or more characters that the scanner translates into another character. Escape sequences are used for characters that lack a glyph, such as newline or tab, and for ones that occur in the syntax, such as an open or close quote.

simple: /* (^*)* */. With *, the RE is more complex: /* (^* | *+ ^/)* */. An FA to implement this RE follows.



The correspondence between the RE and this FA is not as obvious as it was in the examples earlier in the chapter. Section 2.4 presents constructions that automate the construction of an FA from an RE. The complexity of the RE and FA for multiline comments arises from the use of multi-character delimiters. The transition from s_2 to s_3 encodes the fact that the recognizer has seen a * so that it can handle either the appearance of a / or the lack thereof in the correct manner. In contrast, Pascal uses single-character comment delimiters: { and }, so a Pascal comment is just { ^}* }.

Trying to be specific with an RE can also lead to complex expressions. Consider, for example, that the register specifier in a typical assembly language consists of the letter r followed immediately by a small integer. In ILOC, which admits an unlimited set of register names, the RE might be $r[0\ldots9]^+$, with the following FA:



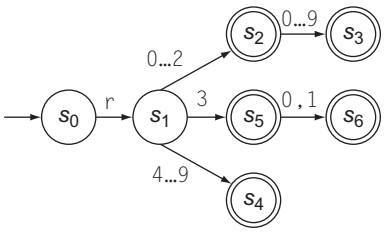
This recognizer accepts r29, and rejects s29. It also accepts r99999, even though no currently available computer has 100,000 registers.

On a real computer, however, the set of register names is severely limited—say, to 32, 64, 128, or 256 registers. One way for a scanner to check validity of a register name is to convert the digits into a number and test whether or not it falls into the range of valid register numbers. The alternative is to adopt a more precise RE specification, such as:

$$r\,(\,[0\ldots2]\,([0\ldots9]\,|\,\epsilon)\,|\,[4\ldots9]\,|\,(3\,(0\,|\,1\,|\,\epsilon))\,)$$

This RE specifies a much smaller language, limited to register numbers 0 to 31 with an optional leading 0 on single-digit register names. It accepts

`r0`, `r00`, `r01`, and `r31`, but rejects `r001`, `r32`, and `r99999`. The corresponding FA looks like:



Which FA is better? They both make a single transition on each input character. Thus, they have the same cost, even though the second FA checks a more complex specification. The more complex FA has more states and transitions, so its representation requires more space. However, their operating costs are the same.

This point is critical: the cost of operating an FA is proportional to the length of the input, not to the length or complexity of the RE that generates the FA. More complex REs may produce FAs with more states that, in turn, need more space. The cost of generating an FA from an RE may also rise with increased complexity in the RE. But, the cost of FA operation remains one transition per input character.

Can we improve our description of the register specifier? The previous RE is both complex and counterintuitive. A simpler alternative might be:

r0 | *r00* | *r1* | *r01* | *r2* | *r02* | *r3* | *r03* | *r4* | *r04* | *r5* | *r05* | *r6* | *r06* | *r7* | *r07* | *r8* | *r08* | *r9* | *r09* | *r10* | *r11* | *r12* | *r13* | *r14* | *r15* | *r16* | *r17* | *r18* | *r19* | *r20* | *r21* | *r22* | *r23* | *r24* | *r25* | *r26* | *r27* | *r28* | *r29* | *r30* | *r31*

This RE is conceptually simpler, but much longer than the previous version. The resulting FA still requires one transition per input symbol. Thus, if we can control the growth in the number of states, we might prefer this version of the RE because it is clear and obvious. However, when our processor suddenly has 256 or 384 registers, enumeration may become tedious, too.

2.3.3 Closure Properties of REs

Regular languages
Any language that can be specified by a regular expression is called a *regular language*.

Regular expressions and the languages that they generate have been the subject of extensive study. They have many interesting and useful properties. Some of these properties play a critical role in the constructions that build recognizers from REs.

PROGRAMMING LANGUAGES VERSUS NATURAL LANGUAGES

Lexical analysis highlights one of the subtle ways in which programming languages differ from natural languages, such as English or Chinese. In natural languages, the relationship between a word's representation—its spelling or its pictogram—and its meaning is not obvious. In English, *are* is a verb while *art* is a noun, even though they differ only in the final character. Furthermore, not all combinations of characters are legitimate words. For example, *arz* differs minimally from *are* and *art*, but does not occur as a word in normal English usage.

A scanner for English could use FA-based techniques to recognize potential words, since all English words are drawn from a restricted alphabet. After that, however, it must look up the prospective word in a dictionary to determine if it is, in fact, a word. If the word has a unique part of speech, dictionary lookup will also resolve that issue. However, many English words can be classified with several parts of speech. Examples include *buoy* and *stress*; both can be either a noun or a verb. For these words, the part of speech depends on the surrounding context. In some cases, understanding the grammatical context suffices to classify the word. In other cases, it requires an understanding of meaning, for both the word and its context.

In contrast, the words in a programming language are almost always specified lexically. Thus, any string in [*1*. . . *9*][*0*. . . *9*]* is a positive integer. The RE [*a*. . . *z*]([*a*. . . *z*]|[*0*. . . *9*])* defines a subset of the Algol identifiers; *arz*, *are* and *art* are all identifiers, with no lookup needed to establish the fact. To be sure, some identifiers may be reserved as keywords. However, these exceptions can be specified lexically, as well. No context is required.

This property results from a deliberate decision in programming language design. The choice to make spelling imply a unique part of speech simplifies scanning, simplifies parsing, and, apparently, gives up little in the expressiveness of the language. Some languages have allowed words with dual parts of speech—for example, PL/I has no reserved keywords. The fact that more recent languages abandoned the idea suggests that the complications outweighed the extra linguistic flexibility.

Regular expressions are closed under many operations—that is, if we apply the operation to an RE or a collection of REs, the result is an RE. Obvious examples are concatenation, union, and closure. The concatenation of two REs x and y is just xy. Their union is $x \mid y$. The Kleene closure of x is just x^*. From the definition of an RE, all of these expressions are also REs.

These closure properties play a critical role in the use of REs to build scanners. Assume that we have an RE for each syntactic category in the source language, $a_0, a_1, a_2, \ldots, a_n$. Then, to construct an RE for all the valid words in the language, we can join them with alternation as $a_0 \mid a_1 \mid a_2 \mid \ldots \mid a_n$. Since REs are closed under union, the result is an RE. Anything that we can

do to an RE for a single syntactic category will be equally applicable to the RE for all the valid words in the language.

Closure under union implies that any finite language is a regular language. We can construct an RE for any finite collection of words by listing them in a large alternation. Because the set of REs is closed under union, that alternation is an RE and the corresponding language is regular.

Closure under concatenation allows us to build complex REs from simpler ones by concatenating them. This property seems both obvious and unimportant. However, it lets us piece together REs in systematic ways. Closure ensures that *ab* is an RE as long as both *a* and *b* are REs. Thus, any techniques that can be applied to either *a* or *b* can be applied to *ab*; this includes constructions that automatically generate a recognizer from REs.

Regular expressions are also closed under both Kleene closure and the finite closures. This property lets us specify particular kinds of large, or even infinite, sets with finite patterns. Kleene closure lets us specify infinite sets with concise finite patterns; examples include the integers and unbounded-length identifiers. Finite closures let us specify large but finite sets with equal ease.

The next section shows a sequence of constructions that build an FA to recognize the language specified by an RE. Section 2.6 shows an algorithm that goes the other way, from an FA to an RE. Together, these constructions establish the equivalence of REs and FAs. The fact that REs are closed under alternation, concatenation, and closure is critical to these constructions.

The equivalence between REs and FAs also suggests other closure properties. For example, given a complete FA, we can construct an FA that recognizes all words *w* that are not in *L*(FA), called the complement of *L*(FA). To build this new FA for the complement, we can swap the designation of accepting and nonaccepting states in the original FA. This result suggests that REs are closed under complement. Indeed, many systems that use REs include a complement operator, such as the ^ operator in `lex`.

Complete FA
an FA that explicitly includes all error transitions

SECTION REVIEW

Regular expressions are a concise and powerful notation for specifying the microsyntax of programming languages. REs build on three basic operations over finite alphabets: alternation, concatenation, and Kleene closure. Other convenient operators, such as finite closures, positive closure, and complement, derive from the three basic operations. Regular expressions and finite automata are related; any RE can be realized in an FA and the language accepted by any FA can be described with RE. The next section formalizes that relationship.

Review Questions

1. Recall the RE for a six-character identifier, written using a finite closure.

   $$([A\ldots Z] \mid [a\ldots z])\ ([A\ldots Z] \mid [a\ldots z] \mid [0\ldots 9])^5$$

   Rewrite it in terms of the three basic RE operations: alternation, concatenation, and closure.

2. In PL/I, the programmer can insert a quotation mark into a string by writing two quotation marks in a row. Thus, the string

   ```
   The quotation mark, ", should be typeset in italics
   ```

   would be written in a PL/I program as

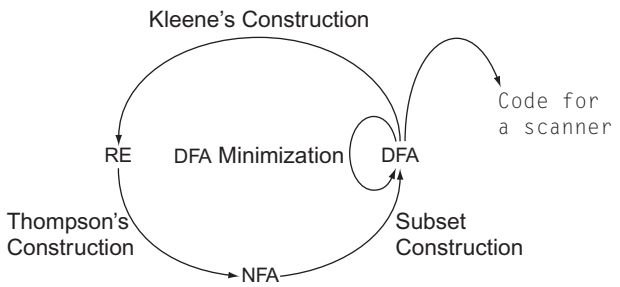
   ```
   "The quotation mark, "", should be typeset in italics."
   ```

   Design an RE and an FA to recognize PL/I strings. Assume that strings begin and end with quotation marks and contain only symbols drawn from an alphabet, designated as Σ. Quotation marks are the only special case.

2.4 FROM REGULAR EXPRESSION TO SCANNER

The goal of our work with finite automata is to automate the derivation of executable scanners from a collection of REs. This section develops the constructions that transform an RE into an FA that is suitable for direct implementation and an algorithm that derives an RE for the language accepted by an FA. Figure 2.3 shows the relationship between all of these constructions.

To present these constructions, we must distinguish between *deterministic* FAs, or DFAs, and *nondeterministic* FAs, or NFAs, in Section 2.4.1. Next,

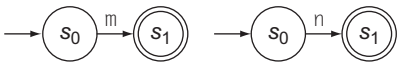


■ **FIGURE 2.3** The Cycle of Constructions.

we present the construction of a deterministic FA from an RE in three steps. Thompson's construction, in Section 2.4.2, derives an NFA from an RE. The subset construction, in Section 2.4.3, builds a DFA that simulates an NFA. Hopcroft's algorithm, in Section 2.4.4, minimizes a DFA. To establish the equivalence of REs and DFAs, we also need to show that any DFA is equivalent to an RE; Kleene's construction derives an RE from a DFA. Because it does not figure directly into scanner construction, we defer that algorithm until Section 2.6.1.

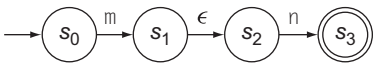
2.4.1 Nondeterministic Finite Automata

Recall from the definition of an RE that we designated the empty string, ϵ, as an RE. None of the FAs that we built by hand included ϵ, but some of the REs did. What role does ϵ play in an FA? We can use transitions on ϵ to combine FAs and form FAs for more complex REs. For example, assume that we have FAs for the REs m and n, called FA_m and FA_n, respectively.

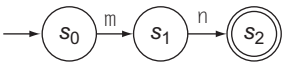


We can build an FA for mn by adding a transition on ϵ from the accepting state of FA_m to the initial state of FA_n, renumbering the states, and using FA_n's accepting state as the accepting state for the new FA.

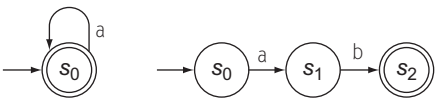
ϵ-transition
a transition on the empty string, ϵ, that does not advance the input



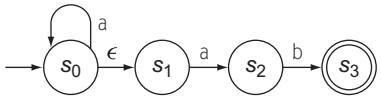
With an ϵ-transition, the definition of acceptance must change slightly to allow one or more ϵ-transitions between any two characters in the input string. For example, in s_1, the FA takes the transition $s_1 \xrightarrow{\epsilon} s_2$ without consuming any input character. This is a minor change, but it seems intuitive. Inspection shows that we can combine s_1 and s_2 to eliminate the ϵ-transition.



Merging two FAs with an ϵ-transition can complicate our model of how FAs work. Consider the FAs for the languages a^* and ab.



We can combine them with an ϵ-transition to form an FA for a^*ab.



The ϵ transition, in effect, gives the FA two distinct transitions out of s_0 on the letter a. It can take the transition $s_0 \xrightarrow{a} s_0$, or the two transitions $s_0 \xrightarrow{\epsilon} s_1$ and $s_1 \xrightarrow{a} s_2$. Which transition is correct? Consider the strings aab and ab. The DFA should accept both strings. For aab, it should move $s_0 \xrightarrow{a} s_0$, $s_0 \xrightarrow{\epsilon} s_1$, $s_1 \xrightarrow{a} s_2$, and $s_2 \xrightarrow{b} s_3$. For ab, it should move $s_0 \xrightarrow{\epsilon} s_1$, $s_1 \xrightarrow{a} s_2$, and $s_2 \xrightarrow{b} s_3$.

Nondeterministic FA
an FA that allows transitions on the empty string, ϵ, and states that have multiple transitions on the same character

As these two strings show, the correct transition out of s_0 on a depends on the characters that follow the a. At each step, an FA examines the current character. Its state encodes the left context, that is, the characters that it has already processed. Because the FA must make a transition before examining the next character, a state such as s_0 violates our notion of the behavior of a sequential algorithm. An FA that includes states such as s_0 that have multiple transitions on a single character is called a *nondeterministic finite automaton* (NFA). By contrast, an FA with unique character transitions in each state is called a *deterministic finite automaton* (DFA).

Deterministic FA
A DFA is an FA where the transition function is single-valued. DFAs do not allow ϵ-transitions.

To make sense of an NFA, we need a set of rules that describe its behavior. Historically, two distinct models have been given for the behavior of an NFA.

1. Each time the NFA must make a nondeterministic choice, it follows the transition that leads to an accepting state for the input string, if such a transition exists. This model, using an omniscient NFA, is appealing because it maintains (on the surface) the well-defined accepting mechanism of the DFA. In essence, the NFA guesses the correct transition at each point.
2. Each time the NFA must make a nondeterministic choice, the NFA clones itself to pursue each possible transition. Thus, for a given input character, the NFA is in a specific set of states, taken across all of its clones. In this model, the NFA pursues all paths concurrently.
   At any point, we call the specific set of states in which the NFA is active its *configuration*. When the NFA reaches a configuration in which it has exhausted the input and one or more of the clones has reached an accepting state, the NFA accepts the string.

Configuration of an NFA
the set of concurrently active states of an NFA

In either model, the NFA $(S, \Sigma, \delta, s_0, S_A)$ accepts an input string $x_1\ x_2\ x_3 \ldots x_k$ if and only if there exists at least one path through the transition diagram that starts in s_0 and ends in some $s_k \in S_A$ such that the edge labels along the path

match the input string. (Edges labelled with ϵ are omitted.) In other words, the i^{th} edge label must be x_i. This definition is consistent with either model of the NFA's behavior.

Equivalence of NFAs and DFAs

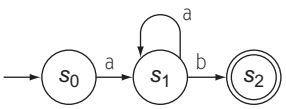
NFAs and DFAs are equivalent in their expressive power. Any DFA is a special case of an NFA. Thus, an NFA is at least as powerful as a DFA. Any NFA can be simulated by a DFA—a fact established by the subset construction in Section 2.4.3. The intuition behind this idea is simple; the construction is a little more complex.

Consider the state of an NFA when it has reached some point in the input string. Under the second model of NFA behavior, the NFA has some finite set of operating clones. The number of these configurations can be bounded; for each state, the configuration either includes one or more clones in that state or it does not. Thus, an NFA with n states produces at most $|\Sigma|^n$ configurations.

To simulate the behavior of the NFA, we need a DFA with a state for each configuration of the NFA. As a result, the DFA may have exponentially more states than the NFA. While S_{DFA}, the set of states in the DFA, might be large, it is finite. Furthermore, the DFA still makes one transition per input symbol. Thus, the DFA that simulates the NFA still runs in time proportional to the length of the input string. The simulation of an NFA on a DFA has a potential space problem, but not a time problem.

Powerset of *N*
the set of all subsets of N, denoted 2^N

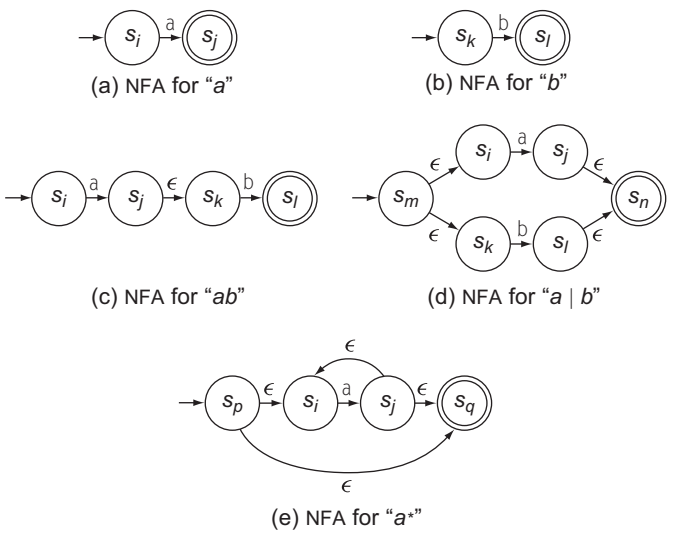
Since NFAs and DFAs are equivalent, we can construct a DFA for a^*ab:



It relies on the observation that a^*ab specifies the same set of words as aa^*b.

2.4.2 Regular Expression to NFA: Thompson's Construction

The first step in moving from an RE to an implemented scanner must derive an NFA from the RE. *Thompson's construction* accomplishes this goal in a straightforward way. It has a template for building the NFA that corresponds to a single-letter RE, and a transformation on NFAs that models the effect of each basic RE operator: concatenation, alternation, and closure. Figure 2.4

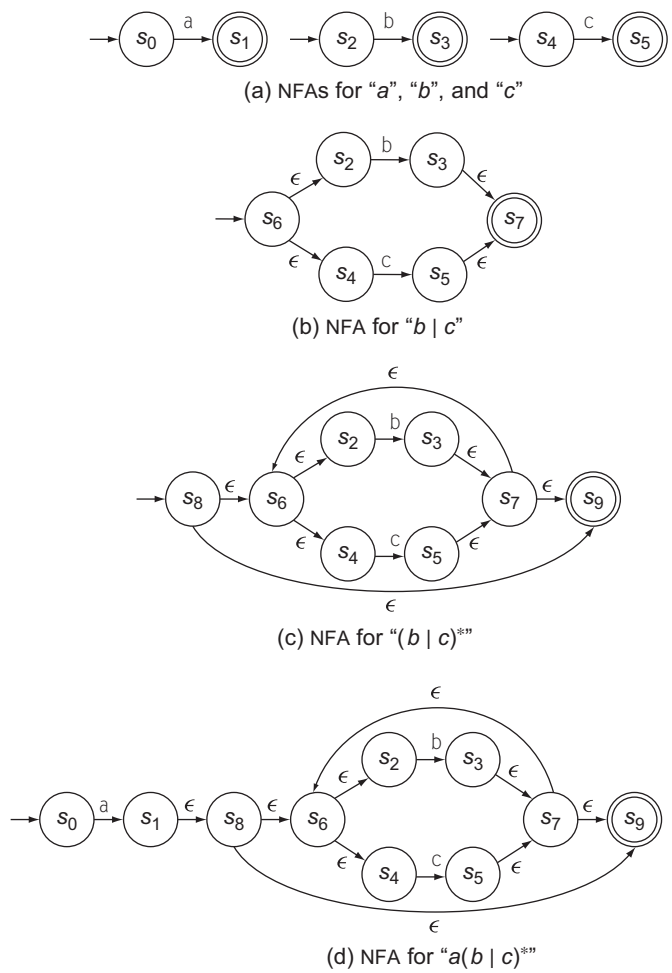


■ **FIGURE 2.4** Trivial NFAs for Regular Expression Operators.

shows the trivial NFAs for the REs a and b, as well as the transformations to form NFAs for the REs ab, $a|b$, and a^* from the NFAs for a and b. The transformations apply to arbitrary NFAs.

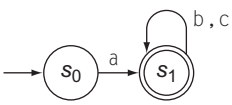
The construction begins by building trivial NFAs for each character in the input RE. Next, it applies the transformations for alternation, concatenation, and closure to the collection of trivial NFAs in the order dictated by precedence and parentheses. For the RE $a(b|c)^*$, the construction would first build NFAs for a, b, and c. Because parentheses have highest precedence, it next builds the NFA for the expression enclosed in parentheses, $b|c$. Closure has higher precedence than concatenation, so it next builds the closure, $(b|c)^*$. Finally, it concatenates the NFA for a to the NFA for $(b|c)^*$.

The NFAs derived from Thompson's construction have several specific properties that simplify an implementation. Each NFA has one start state and one accepting state. No transition, other than the initial transition, enters the start state. No transition leaves the accepting state. An ϵ-transition always connects two states that were, earlier in the process, the start state and the accepting state of NFAs for some component REs. Finally, each state has at most two entering and two exiting ϵ-moves, and at most one entering and one exiting move on a symbol in the alphabet. Together, these properties simplify the representation and manipulation of the NFAs. For example, the construction only needs to deal with a single accepting state, rather than iterating over a set of accepting states in the NFA.



■ **FIGURE 2.5** Applying Thompson's Construction to $a(b|c)^*$.

Figure 2.5 shows the NFA that Thompson's construction builds for $a(b|c)^*$. It has many more states than the DFA that a human would likely produce, shown at left. The NFA also contains many ϵ-moves that are obviously unneeded. Later stages in the construction will eliminate them.

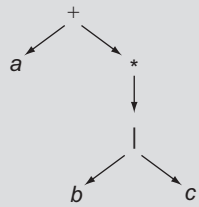


2.4.3 NFA to DFA: The Subset Construction

Thompson's construction produces an NFA to recognize the language specified by an RE. Because DFA execution is much easier to simulate than NFA execution, the next step in the cycle of constructions converts the NFA built

REPRESENTING THE PRECEDENCE OF OPERATORS

Thompson's construction must apply its three transformations in an order that is consistent with the precedence of the operators in the regular expression. To represent that order, an implementation of Thompson's construction can build a tree that represents the regular expression and its internal precedence. The RE $a(b|c)^*$ produces the following *tree*:



where + represents concatenation, | represents alternation, and * represents closure. The parentheses are folded into the structure of the tree and, thus, have no explicit representation.

The construction applies the individual transformations in a postorder walk over the tree. Since transformations correspond to operations, the postorder walk builds the following sequence of NFAs: a, b, c, $b|c$, $(b|c)^*$, and, finally, $a(b|c)^*$. Chapters 3 and 4 show how to build expression trees.

by Thompson's construction into a DFA that recognizes the same language. The resulting DFAs have a simple execution model and several efficient implementations. The algorithm that constructs a DFA from an NFA is called the *subset construction*.

The subset construction takes as input an NFA, $(N, \Sigma, \delta_N, n_0, N_A)$. It produces a DFA, $(D, \Sigma, \delta_D, d_0, D_A)$. The NFA and the DFA use the same alphabet, Σ. The DFA's start state, d_0, and its accepting states, D_A, will emerge from the construction. The complex part of the construction is the derivation of the set of DFA states D from the NFA states N, and the derivation of the DFA transition function δ_D.

Valid configuration
configuration of an NFA that can be reached by some input string

The algorithm, shown in Figure 2.6, constructs a set Q whose elements, q_i are each a subset of N, that is, each $q_i \in 2^N$. When the algorithm halts, each $q_i \in Q$ corresponds to a state, $d_i \in D$, in the DFA. The construction builds the elements of Q by following the transitions that the NFA can make on a given input. Thus, each q_i represents a valid configuration of the NFA.

The algorithm begins with an initial set, q_0, that contains n_0 and any states in the NFA that can be reached from n_0 along paths that contain only

```
q0 ← ε-closure({n0});
Q ← q0;
WorkList ← {q0};

while (WorkList ≠ ∅ ) do
   remove q from WorkList;

   for each character c ∈ Σ do
      t ← ε-closure(Delta(q,c));
      T[q,c] ← t;

      if t ∉ Q then
         add t to Q and to WorkList;
   end;
end;
```

■ **FIGURE 2.6** The Subset Construction.

ϵ-transitions. Those states are equivalent since they can be reached without consuming input.

To construct q_0 from n_0, the algorithm computes ϵ-`closure(`n_0`)`. It takes, as input, a set S of NFA states. It returns a set of NFA states constructed from S as follows: ϵ-`closure` examines each state $s_i \in S$ and adds to S any state reachable by following one or more ϵ-transitions from s_i. If S is the set of states reachable from n_0 by following paths labelled with `abc`, then ϵ-`closure(`S`)` is the set of states reachable from n_0 by following paths labelled `abc`ϵ^*. Initially, `Q` has only one member, q_0 and the `WorkList` contains q_0.

The algorithm proceeds by removing a set q from the worklist. Each q represents a valid configuration of the original NFA. The algorithm constructs, for each character c in the alphabet Σ, the configuration that the NFA would reach if it read c while in configuration q. This computation uses a function `Delta(`q,c`)` that applies the NFA's transition function to each element of q. It returns $\cup_{s \in q_i} \delta_N(s,c)$.

The while loop repeatedly removes a configuration q from the worklist and uses `Delta` to compute its potential transitions. It augments this computed configuration with any states reachable by following ϵ-transitions, and adds any new configurations generated in this way to both `Q` and the worklist. When it discovers a new configuration `t` reachable from q on character c, the algorithm records that transition in the table `T`. The inner loop, which iterates over the alphabet for each configuration, performs an exhaustive search.

Notice that `Q` grows monotonically. The while loop adds sets to `Q` but never removes them. Since the number of configurations of the NFA is bounded and

each configuration only appears once on the worklist, the while loop must halt. When it halts, `Q` contains all of the valid configurations of the NFA and `T` holds all of the transitions between them.

`Q` can become large—as large as $|2^N|$ distinct states. The amount of nondeterminism found in the NFA determines how much state expansion occurs. Recall, however, that the result is a DFA that makes exactly one transition per input character, independent of the number of states in the DFA. Thus, any expansion introduced by the subset construction does not affect the running time of the DFA.

From Q to D

When the subset construction halts, it has constructed a model of the desired DFA, one that simulates the original NFA. Building the DFA from `Q` and `T` is straightforward. Each $q_i \in$ `Q` needs a state $d_i \in D$ to represent it. If q_i contains an accepting state of the NFA, then d_i is an accepting state of the DFA. We can construct the transition function, δ_D, directly from `T` by observing the mapping from q_i to d_i. Finally, the state constructed from q_0 becomes d_0, the initial state of the DFA.

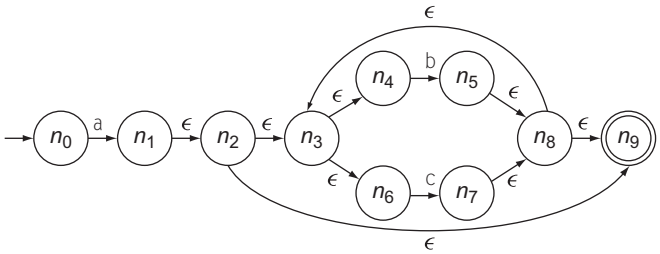
Example

Consider the NFA built for $a(b|c)^*$ in Section 2.4.2 and shown in Figure 2.7a, with its states renumbered. The table in Figure 2.7b sketches the steps that the subset construction follows. The first column shows the name of the set in `Q` being processed in a given iteration of the while loop. The second column shows the name of the corresponding state in the new DFA. The third column shows the set of NFA states contained in the current set from `Q`. The final three columns show results of computing the ϵ-`closure` of `Delta` on the state for each character in Σ.

The algorithm takes the following steps:

1. The initialization sets q_0 to ϵ-`closure(`$\{n_0\}$`)`, which is just n_0. The first iteration computes ϵ-`closure(Delta(`q_0,a`))`, which contains six NFA states, and ϵ-`closure(Delta(`q_0,b`))` and ϵ-`closure(Delta(`q_0,c`))`, which are empty.
2. The second iteration of the while loop examines q_1. It produces two configurations and names them q_2 and q_3.
3. The third iteration of the while loop examines q_2. It constructs two configurations, which are identical to q_2 and q_3.
4. The fourth iteration of the while loop examines q_3. Like the third iteration, it reconstructs q_2 and q_3.

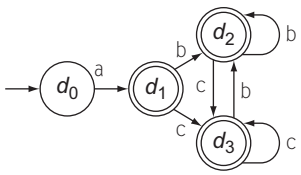
Figure 2.7c shows the resulting DFA; the states correspond to the DFA states from the table and the transitions are given by the `Delta` operations that



(a) NFA for "$a(b \mid c)^*$" (With States Renumbered)

Set Name	DFA States	NFA States	ϵ-closure(Delta(q,*))		
			a	b	c
q_0	d_0	n_0	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	– none –
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3

(b) Iterations of the Subset Construction



(a) Resulting DFA

■ **FIGURE 2.7** Applying the Subset Construction to the NFA from Figure 2.5.

generate those states. Since the sets q_1, q_2 and q_3 all contain n_9 (the accepting state of the NFA), all three become accepting states in the DFA.

Fixed-Point Computations

The subset construction is an example of a *fixed-point computation*, a particular style of computation that arises regularly in computer science. These

Monotone function
a function f on domain D is *monotone* if, $\forall x, y \in D, x \leq y \Rightarrow f(x) \leq f(y)$

computations are characterized by the iterated application of a monotone function to some collection of sets drawn from a domain whose structure is known. These computations terminate when they reach a state where further iteration produces the same answer—a "fixed point" in the space of successive iterates. Fixed-point computations play an important and recurring role in compiler construction.

Termination arguments for fixed-point algorithms usually depend on known properties of the domain. For the subset construction, the domain D is 2^{2^N}, since $Q = \{q_0, q_1, q_2, \ldots, q_k\}$ where each $q_i \in 2^N$. Since N is finite, 2^N and 2^{2^N} are also finite. The while loop adds elements to Q; it cannot remove an element from Q. We can view the while loop as a monotone increasing function f, which means that for a set x, $f(x) \geq x$. (The comparison operator $\geq$ is $\supseteq$.) Since Q can have at most $|2^N|$ distinct elements, the while loop can iterate at most $|2^N|$ times. It may, of course, reach a fixed point and halt more quickly than that.

Computing ϵ-closure Offline

An implementation of the subset construction could compute ϵ-`closure()` by following paths in the transition graph of the NFA as needed. Figure 2.8 shows another approach: an offline algorithm that computes ϵ-`closure(`$\{n\}$`)` for each state n in the transition graph. The algorithm is another example of a fixed-point computation.

For the purposes of this algorithm, consider the transition diagram of the NFA as a graph, with nodes and edges. The algorithm begins by creating a set E for each node in the graph. For a node n, $E(n)$ will hold the current

```
for each state n ∈ N do
    E(n) ← {n};
end;
WorkList ← N;
while (WorkList ≠ ∅) do
  remove n from WorkList;
  t ← {n} ∪ ⋃_{n →ε p ∈ δ_N} E(p);
  if t ≠ E(n)
      then begin;
         E(n) ← t;
         WorkList ← WorkList ∪ {m | m →ε n ∈ δ_N};
      end;
end;
```

■ **FIGURE 2.8** An Offline Algorithm for ϵ-closure.

approximation to ϵ-`closure(n)`. Initially, the algorithm sets $E(n)$ to $\{n\}$, for each node n, and places each node on the worklist.

Each iteration of the while loop removes a node n from the worklist, finds all of the ϵ-transitions that leave n, and adds their targets to $E(n)$. If that computation changes $E(n)$, it places n's predecessors along ϵ-transitions on the worklist. (If n is in the ϵ-closure of its predecessor, adding nodes to $E(n)$ must also add them to the predecessor's set.) This process halts when the worklist becomes empty.

Using a bit-vector set for the worklist can ensure that the algorithm does not have duplicate copies of a node's name on the worklist.

See Appendix B.2.

The termination argument for this algorithm is more complex than that for the algorithm in Figure 2.6. The algorithm halts when the worklist is empty. Initially, the worklist contains every node in the graph. Each iteration removes a node from the worklist; it may also add one or more nodes to the worklist.

The algorithm only adds a node to the worklist if the E set of its successor changes. The $E(n)$ sets increase monotonically. For a node x, its successor y along an ϵ-transition can place x on the worklist at most $|E(y)| \leq |N|$ times, in the worst case. If x has multiple successors y_i along ϵ-transitions, each of them can place x on the worklist $|E(y_i)| \leq |N|$ times. Taken over the entire graph, the worst case behavior would place nodes on the worklist $k \cdot |N|$ times, where k is the number of ϵ-transitions in the graph. Thus, the worklist eventually becomes empty and the computation halts.

2.4.4 DFA to Minimal DFA: Hopcroft's Algorithm

As a final refinement to the RE→DFA conversion, we can add an algorithm to minimize the number of states in the DFA. The DFA that emerges from the subset construction can have a large set of states. While this does not increase the time needed to scan a string, it does increase the size of the recognizer in memory. On modern computers, the speed of memory accesses often governs the speed of computation. A smaller recognizer may fit better into the processor's cache memory.

To minimize the number of states in a DFA, $(D, \Sigma, \delta, d_0, D_A)$, we need a technique to detect when two states are equivalent—that is, when they produce the same behavior on any input string. The algorithm in Figure 2.9 finds equivalence classes of DFA states based on their behavior. From those equivalence classes, we can construct a minimal DFA.

Set partition
A *set partition* of S is a collection of nonempty, disjoint subsets of S whose union is exactly S.

The algorithm constructs a set partition, $P = \{p_1, p_2, p_3, \ldots p_m\}$, of the DFA states. The particular partition, P, that it constructs groups together DFA states by their behavior. Two DFA states, $d_i, d_j \in p_s$, have the same behavior in response to all input characters. That is, if $d_i \xrightarrow{c} d_x$, $d_j \xrightarrow{c} d_y$, and $d_i, d_j \in p_s$,

```
T ← {D_A,  { D − D_A } };
P ← ∅
while (P ≠ T) do
    P ← T;
    T ← ∅;
    for each set p ∈ P do
        T ← T ∪ Split(p);
    end;
end;
```

```
Split(S) {
    for each c ∈ Σ do
        if c splits S into s1 and s2
            then return {s1, s2};
    end;
    return S;
}
```

■ **FIGURE 2.9** DFA Minimization Algorithm.

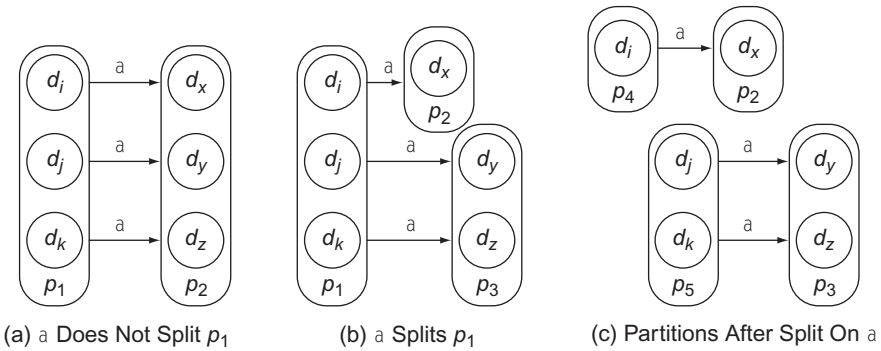
then d_x and d_y must be in the same set p_t. This property holds for every set $p_s \in P$, for every pair of states $d_i, d_j \in p_s$, and for every input character, c. Thus, the states in p_s have the same behavior with respect to input characters and the remaining sets in P.

To minimize a DFA, each set $p_s \in P$ should be as large as possible, within the constraint of behavioral equivalence. To construct such a partition, the algorithm begins with an initial rough partition that obeys all the properties *except* behavioral equivalence. It then iteratively refines that partition to enforce behavioral equivalence. The initial partition contains two sets, $p_0 = D_A$ and $p_1 = \{D - D_A\}$. This separation ensures that no set in the final partition contains both accepting and nonaccepting states, since the algorithm never combines two partitions.

The algorithm refines the initial partition by repeatedly examining each $p_s \in P$ to look for states in p_s that have different behavior for some input string. Clearly, it cannot trace the behavior of the DFA on every string. It can, however, simulate the behavior of a given state in response to a single input character. It uses a simple condition for refining the partition: a symbol $c \in \Sigma$ must produce the same behavior for every state $d_i \in p_s$. If it does not, the algorithm splits p_s around c.

This splitting action is the key to understanding the algorithm. For d_i and d_j to remain together in p_s, they must take equivalent transitions on each character $c \in \Sigma$. That is, $\forall\, c \in \Sigma$, $d_i \xrightarrow{c} d_x$ and $d_j \xrightarrow{c} d_y$, where $d_x, d_y \in p_t$. Any state $d_k \in p_s$ where $d_k \xrightarrow{c} d_z$, $d_z \notin p_t$, cannot remain in the same partition as d_i and d_j. Similarly, if d_i and d_j have transitions on c and d_k does not, it cannot remain in the same partition as d_i and d_j.

Figure 2.10 makes this concrete. The states in $p_1 = \{d_i, d_j, d_k\}$ are equivalent if and only if their transitions, $\forall\, c \in \Sigma$, take them to states that are, themselves, in an equivalence class. As shown, each state has a transition on `a`: $d_i \xrightarrow{a} d_x$, $d_j \xrightarrow{a} d_y$, and $d_k \xrightarrow{a} d_z$. If d_x, d_y, and d_z are all in the same set in



■ **FIGURE 2.10** Splitting a Partition around a.

the current partition, as shown on the left, then d_i, d_j, and d_k should remain together and a does not split p_1.

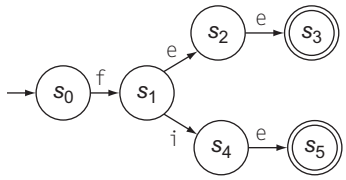
On the other hand, if d_x, d_y, and d_z are in two or more different sets, then a splits p_1. As shown in the center drawing of Figure 2.10, $d_x \in p_2$ while d_y and $d_z \in p_3$, so the algorithm must split p_1 and construct two new sets $p_4 = \{d_i\}$ and $p_5 = \{d_j, d_k\}$ to reflect the potential for different outcomes with strings that begin with the symbol a. The result is shown on the right side of Figure 2.10. The same split would result if state d_i had no transition on a.

To refine a partition P, the algorithm examines each $p \in P$ and each $c \in \Sigma$. If c splits p, the algorithm constructs two new sets from p and adds them to T. (It could split p into more than two sets, all having internally consistent behavior on c. However, creating one consistent state and lumping the rest of p into another state will suffice. If the latter state is inconsistent in its behavior on c, the algorithm will split it in a later iteration.) The algorithm repeats this process until it finds a partition where it can split no sets.

To construct the new DFA from the final partition p, we can create a single state to represent each set $p \in P$ and add the appropriate transitions between these new representative states. For the state representing p_l, we add a transition to the state representing p_m on c if some $d_j \in p_l$ has a transition on c to some $d_k \in p_m$. From the construction, we know that if d_j has such a transition, so does every other state in p_l; if this were not the case, the algorithm would have split p_l around c. The resulting DFA is minimal; the proof is beyond our scope.

Examples

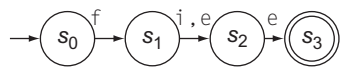
Consider a DFA that recognizes the language *fee* | *fie*, shown in Figure 2.11a. By inspection, we can see that states s_3 and s_5 serve the same purpose. Both



(a) DFA for "*fee* | *fie*"

Step	Current Partition	Examines		
		Set	Char	Action
0	$\{\{s_3,s_5\},\{s_0,s_1,s_2,s_4\}\}$	—	—	—
1	$\{\{s_3,s_5\},\{s_0,s_1,s_2,s_4\}\}$	$\{s_3,s_5\}$	*all*	*none*
2	$\{\{s_3,s_5\},\{s_0,s_1,s_2,s_4\}\}$	$\{s_0,s_1,s_2,s_4\}$	e	*split* $\{s_2,s_4\}$
3	$\{\{s_3,s_5\},\{s_0,s_1\},\{s_2,s_4\}\}$	$\{s_0,s_1\}$	f	*split* $\{s_1\}$
4	$\{\{s_3,s_5\},\{s_0\},\{s_1\},\{s_2,s_4\}\}$	*all*	*all*	*none*

(b) Critical Steps in Minimizing the DFA



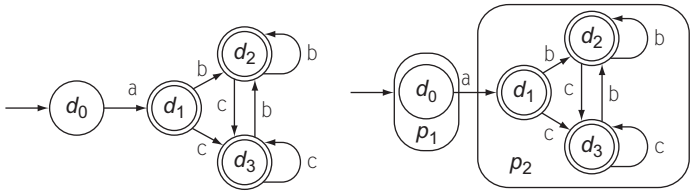
(c) The Minimal DFA (States Renumbered)

■ **FIGURE 2.11** Applying the DFA Minimization Algorithm.

are accepting states entered only by a transition on the letter e. Neither has a transition that leaves the state. We would expect the DFA minimization algorithm to discover this fact and replace them with a single state.

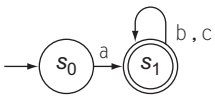
Figure 2.11b shows the significant steps that occur in minimizing this DFA. The initial partition, shown as step 0, separates accepting states from nonaccepting states. Assuming that the while loop in the algorithm iterates over the sets of P in order, and over the characters in $\Sigma = \{\texttt{e},\texttt{f},\texttt{i}\}$ in order, then it first examines the set $\{s_3,s_5\}$. Since neither state has an exiting transition, the state does not split on any character. In the second step, it examines $\{s_0,s_1,s_2,s_4\}$; on the character e, it splits $\{s_2,s_4\}$ out of the set. In the third step, it examines $\{s_0,s_1\}$ and splits it around the character f. At that point, the partition is $\{\{s_3,s_5\}, \{s_0\}, \{s_1\}, \{s_2,s_4\}\}$. The algorithm makes one final pass over the sets in the partition, splits none of them, and terminates.

To construct the new DFA, we must build a state to represent each set in the final partition, add the appropriate transitions from the original DFA, and designate initial and accepting state(s). Figure 2.11c shows the result for this example.



■ **FIGURE 2.12** DFA for $a(b|c^*)$.

As a second example, consider the DFA for $a\,(b \mid c)^*$ produced by Thompson's construction and the subset construction, shown in Figure 2.12a. The first step of the minimization algorithm constructs an initial partition $\{\{d_0\}, \{d_1, d_2, d_3\}\}$, as shown on the right. Since p_1 has only one state, it cannot be split. When the algorithm examines p_2, it finds no transitions on a from any state in p_2. For both b and c, each state has a transition back into p_2. Thus, no symbol in Σ splits p_2, and the final partition is $\{\{d_0\}, \{d_1, d_2, d_3\}\}$.



The resulting minimal DFA is shown in Figure 2.12b. Recall that this is the DFA that we suggested a human would derive. After minimization, the automatic techniques produce the same result.

This algorithm is another example of a fixed-point computation. P is finite; at most, it can contain $|D|$ elements. The while loop splits sets in P, but never combines them. Thus, $|P|$ grows monotonically. The loop halts when some iteration splits no sets in P. The worst-case behavior occurs when each state in the DFA has different behavior; in that case, the while loop halts when P has a distinct set for each $d_i \in D$. This occurs when the algorithm is applied to a minimal DFA.

2.4.5 Using a DFA as a Recognizer

Thus far, we have developed the mechanisms to construct a DFA implementation from a single RE. To be useful, a compiler's scanner must recognize all the syntactic categories that appear in the grammar for the source language. What we need, then, is a recognizer that can handle all the REs for the language's microsyntax. Given the REs for the various syntactic categories, $r_1, r_2, r_3, \ldots, r_k$, we can construct a single RE for the entire collection by forming $(r_1 \mid r_2 \mid r_3 \mid \ldots \mid r_k)$.

If we run this RE through the entire process, building an NFA, constructing a DFA to simulate the NFA, minimizing it, and turning that minimal DFA into executable code, the resulting scanner recognizes the next word that matches one of the r_i's. That is, when the compiler invokes it on some input, the

scanner will examine characters one at a time and accept the string if it is in an accepting state when it exhausts the input. The scanner should return both the text of the string and its syntactic category, or part of speech. Since most real programs contain more than one word, we need to transform either the language or the recognizer.

At the language level, we can insist that each word end with some easily recognizable delimiter, like a blank or a tab. This idea is deceptively attractive. Taken literally, it requires delimiters surrounding all operators, as +, -, (,), and the comma.

At the recognizer level, we can change the implementation of the DFA and its notion of acceptance. To find the longest word that matches one of the REs, the DFA should run until it reaches the point where the current state, *s*, has no outgoing transition on the next character. At that point, the implementation must decide which RE it has matched. Two cases arise; the first is simple. If *s* is an accepting state, then the DFA has found a word in the language and should report the word and its syntactic category.

If *s* is not an accepting state, matters are more complex. Two cases occur. If the DFA passed through one or more accepting states on its way to *s*, the recognizer should back up to the most recent such state. This strategy matches the longest valid prefix in the input string. If it never reached an accepting state, then no prefix of the input string is a valid word and the recognizer should report an error. The scanners in Section 2.5.1 implement both these notions.

As a final complication, an accepting state in the DFA may represent several accepting states in the original NFA. For example, if the lexical specification includes REs for keywords as well as an RE for identifiers, then a keyword such as `new` might match two REs. The recognizer must decide which syntactic category to return: identifier or the singleton category for the keyword `new`.

Most scanner-generator tools allow the compiler writer to specify a priority among patterns. When the recognizer matches multiple patterns, it returns the syntactic category of the highest-priority pattern. This mechanism resolves the problem in a simple way. The `lex` scanner generator, distributed with many Unix systems, assigns priorities based on position in the list of REs. The first RE has highest priority, while the last RE has lowest priority.

As a practical matter, the compiler writer must also specify REs for parts of the input stream that do not form words in the program text. In most programming languages, blank space is ignored, but every program contains it. To handle blank space, the compiler writer typically includes an RE that matches blanks, tabs, and end-of-line characters; the action on accepting

blank space is to invoke the scanner, recursively, and return its result. If comments are discarded, they are handled in a similar fashion.

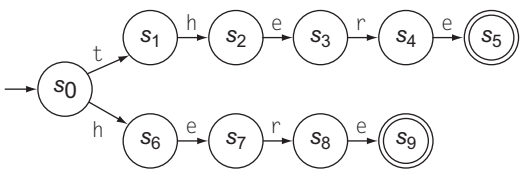
SECTION REVIEW

Given a regular expression, we can derive a minimal DFA to recognize the language specified by the RE using the following steps: (1) apply Thompson's construction to build an NFA for the RE; (2) use the subset construction to derive a DFA that simulates the behavior of the RE; and (3) use Hopcroft's algorithm to identify equivalent states in the DFA and construct a minimal DFA. This trio of constructions produces an efficient recognizer for any language that can be specified with an RE.

Both the subset construction and the DFA minimization algorithm are fixed-point computations. They are characterized by repeated application of a monotone function to some set; the properties of the domain play an important role in reasoning about the termination and complexity of these algorithms. We will see more fixed-point computations in later chapters.

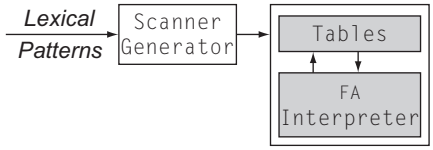
Review Questions

1. Consider the RE *who* | *what* | *where*. Use Thompson's construction to build an NFA from the RE. Use the subset construction to build a DFA from the NFA. Minimize the DFA.
2. Minimize the following DFA:



2.5 IMPLEMENTING SCANNERS

Scanner construction is a problem where the theory of formal languages has produced tools that can automate implementation. For most languages, the compiler writer can produce an acceptably fast scanner directly from a set of regular expressions. The compiler writer creates an RE for each syntactic category and gives the REs as input to a scanner generator. The generator constructs an NFA for each RE, joins them with ϵ-transitions, creates a corresponding DFA, and minimizes the DFA. At that point, the scanner generator must convert the DFA into executable code.



■ **FIGURE 2.13** Generating a Table-Driven Scanner.

This section discusses three implementation strategies for converting a DFA into executable code: a table-driven scanner, a direct-coded scanner, and a hand-coded scanner. All of these scanners operate in the same manner, by simulating the DFA. They repeatedly read the next character in the input and simulate the DFA transition caused by that character. This process stops when the DFA recognizes a word. As described in the previous section, that occurs when the current state, *s*, has no outbound transition on the current input character.

If *s* is an accepting state, the scanner recognizes the word and returns a lexeme and its syntactic category to the calling procedure. If *s* is a nonaccepting state, the scanner must determine whether or not it passed through an accepting state on the way to *s*. If the scanner did encounter an accepting state, it should roll back its internal state and its input stream to that point and report success. If it did not, it should report the failure.

These three implementation strategies, table driven, direct coded, and hand coded, differ in the details of their runtime costs. However, they all have the same asymptotic complexity—constant cost per character, plus the cost of roll back. The differences in the efficiency of well-implemented scanners change the constant costs per character but not the asymptotic complexity of scanning.

The next three subsections discuss implementation differences between table-driven, direct-coded, and hand-coded scanners. The strategies differ in how they model the DFA's transition structure and how they simulate its operation. Those differences, in turn, produce different runtime costs. The final subsection examines two different strategies for handling reserved keywords.

2.5.1 Table-Driven Scanners

The table-driven approach uses a skeleton scanner for control and a set of generated tables that encode language-specific knowledge. As shown in Figure 2.13, the compiler writer provides a set of lexical patterns, specified

```
NextWord()
  state ← s0 ;
  lexeme ← " ";
  clear stack;
  push(bad);

  while (state≠se) do
    NextChar(char);
    lexeme ← lexeme + char;
    if state ∈ SA
        then clear stack;
    push(state);
    cat ← CharCat[char];
    state ← δ[state,cat];
  end;

  while(state ∉ SA and
        state≠bad) do
    state ← pop();
    truncate lexeme;
    RollBack();
  end;

  if state ∈ SA
    then return Type[state];
    else return invalid;
```

r	0, 1, 2, ..., 9	EOF	**Other**
Register	*Digit*	*Other*	*Other*

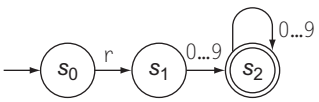
The Classifier Table, *CharCat*

	Register	***Digit***	***Other***
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

The Transition Table, δ

s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

The Token Type Table, *Type*



The Underlying DFA

■ **FIGURE 2.14** A Table-Driven Scanner for Register Names.

as regular expressions. The scanner generator then produces tables that drive the skeleton scanner.

Figure 2.14 shows a table-driven scanner for the RE $r\,[0\ldots9]^+$, which was our first attempt at an RE for ILOC register names. The left side of the figure shows the skeleton scanner, while the right side shows the tables for $r\,[0\ldots9]^+$ and the underlying DFA. Notice the similarity between the code here and the recognizer shown in Figure 2.2 on page 32.

The skeleton scanner divides into four sections: initializations, a scanning loop that models the DFA's behavior, a roll back loop in case the DFA overshoots the end of the token, and a final section that interprets and reports the results. The scanning loop repeats the two basic actions of a scanner: read a character and simulate the DFA's action. It halts when the DFA enters the

error state, s_e. Two tables, `CharCat` and δ, encode all knowledge about the DFA. The roll back loop uses a stack of states to revert the scanner to its most recent accepting state.

The skeleton scanner uses the variable `state` to hold the current state of the simulated DFA. It updates `state` using a two-step, table-lookup process. First, it classifies `char` into one of a small set of categories using the `CharCat` table. The scanner for $r[0\ldots9]^+$ has three categories: *Register, Digit*, or *Other*. Next, it uses the current state and the character category as indices into the transition table, δ.

This two-step translation, character to category, then state and category to new state, lets the scanner use a compressed transition table. The tradeoff between direct access into a larger table and indirect access into the compressed table is straightforward.A complete table would eliminate the mapping through `CharCat`, but would increase the memory footprint of the table. The uncompressed transition table grows as the product of the number of states in the DFA and the number of characters in Σ; it can grow to the point where it will not stay in cache.

For small examples, such as $r[0\ldots9]^+$, the classifier table is larger than the complete transition table. In a realistically sized example, that relationship should be reversed.

With a small, compact character set, such as ASCII, `CharCat` can be represented as a simple table lookup. The relevant portions of `CharCat` should stay in the cache. In that case, table compression adds one cache reference per input character. As the character set grows (e.g. Unicode), more complex implementations of `CharCat` may be needed. The precise tradeoff between the per-character costs of both compressed and uncompressed tables will depend on properties of both the language and the computer that runs the scanner.

To provide a character-by-character interface to the input stream, the skeleton scanner uses a macro, `NextChar`, which sets its sole parameter to contain the next character in the input stream. A corresponding macro, `RollBack`, moves the input stream back by one character. (Section 2.5.3 looks at `NextChar` and `RollBack`.)

If the scanner reads too far, `state` will not contain an accepting state at the end of the first while loop. In that case, the second while loop uses the state trace from the stack to roll the state, lexeme, and input stream back to the most recent accepting state. In most languages, the scanner's overshoot will be limited. Pathological behavior, however, can cause the scanner to examine individual characters many times, significantly increasing the overall cost of scanning. In most programming languages, the amount of roll back is small relative to the word lengths. In languages where significant amounts of roll back can occur, a more sophisticated approach to this problem is warranted.

Avoiding Excess Roll Back

Some regular expressions can produce quadratic calls to roll back in the scanner shown in Figure 2.14. The problem arises from our desire to have the scanner return the longest word that is a prefix of the input stream.

Consider the RE $ab \mid (ab)^* c$. The corresponding DFA, shown in the margin, recognizes either ab or any number of occurrences of ab followed by a final c. On the input string `abababab`c, a scanner built from the DFA will read all the characters and return the entire string as a single word. If, however, the input is `abababab`, it must scan all of the characters before it can determine that the longest prefix is `ab`. On the next invocation, it will scan `ababab` to return `ab`. The third call will scan `abab` to return `ab`, and the final call will simply return `ab` without any roll back. In the worst, case, it can spend quadratic time reading the input stream.

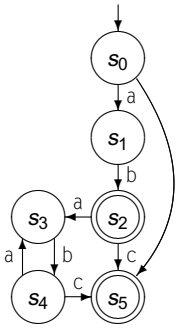


Figure 2.15 shows a modification to the scanner in Figure 2.14 that avoids this problem. It differs from the earlier scanner in three important ways. First, it has a global counter, `InputPos`, to record position in the input stream. Second, it has a bit-array, `Failed`, to record dead-end transitions as the scanner finds them. `Failed` has a row for each state and a column for each position in the input stream. Third, it has an initialization routine that

```
NextWord()
  state ← s0 ;
  lexeme ← " ";
  clear stack;
  push(⟨bad, bad⟩);

  while (state≠se) do
    NextChar(char);
    InputPos ← InputPos + 1;
    lexeme ← lexeme + char;
    if Failed[state,InputPos]
        then break;
    if state ∈ SA
        then clear stack;
    push(⟨state,InputPos⟩);
    cat ← CharCat[char];
    state ← δ[state,cat];
  end;

  while(state ∉ SA and state≠bad ) do
    Failed[state,InputPos] ← true;
    ⟨state,InputPos⟩ ← pop();
    truncate lexeme;
    RollBack();
  end;

  if state ∈ SA
    then return TokenType[state];
    else return bad;


InitializeScanner()
    InputPos=0;
    for each state s in the DFA do
        for i=0 to |input stream| do
            Failed[s,i] ← false;
        end;
    end;
```

■ **FIGURE 2.15** The Maximal Munch Scanner.

must be called before `NextWord()` is invoked. That routine sets `InputPos` to zero and sets `Failed` uniformly to false.

This scanner, called the *maximal munch scanner*, avoids the pathological behavior by marking dead-end transitions as they are popped from the stack. Thus, over time, it records specific ⟨*state*,*input position*⟩ pairs that cannot lead to an accepting state. Inside the scanning loop, the first while loop, the code tests each ⟨*state*,*input position*⟩ pair and breaks out of the scanning loop whenever a failed transition is attempted.

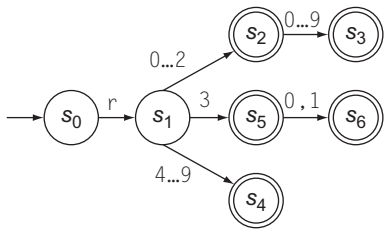
Optimizations can drastically reduce the space requirements of this scheme. (See, for example, Exercise 16 on page 82.) Most programming languages have simple enough microsyntax that this kind of quadratic roll back cannot occur. If, however, you are building a scanner for a language that can exhibit this behavior, the scanner can avoid it for a small additional overhead per character.

Generating the Transition and Classifier Tables

Given a DFA, the scanner generator can generate the tables in a straightforward fashion. The initial table has one column for every character in the input alphabet and one row for each state in the DFA. For each state, in order, the generator examines the outbound transitions and fills the row with the appropriate states. The generator can collapse identical columns into a single instance; as it does so, it can construct the character classifier. (Two characters belong in the same class if and only if they have identical columns in δ.) If the DFA has been minimized, no two rows can be identical, so row compression is not an issue.

Changing Languages

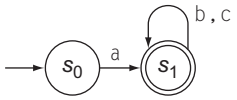
To model another DFA, the compiler writer can simply supply new tables. Earlier in the chapter, we worked with a second, more constrained specification for ILOC register names, given by the RE: $r([0\ldots2]([0\ldots9]\,|\,\epsilon)\,|\,[4\ldots9]\,|\,(3\,(0\,|\,1\,|\,\epsilon))\,)$. That RE gave rise to the following DFA:



Because it has more states and transitions than the RE for $r[0\ldots9]^+$, we should expect a larger transition table.

	r	0,1	2	3	4...9	Other
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e

As a final example, the minimal DFA for the RE $a\,(b|c)^*$ has the following table:



Minimal DFA

	a	b,c	Other
s_0	s_1	s_e	s_e
s_1	s_e	s_1	s_e

Transition Table

The character classifier has three classes: `a`, `b` or `c`, and all other characters.

2.5.2 Direct-Coded Scanners

To improve the performance of a table-driven scanner, we must reduce the cost of one or both of its basic actions: read a character and compute the next DFA transition. Direct-coded scanners reduce the cost of computing DFA transitions by replacing the explicit representation of the DFA's state and transition graph with an implicit one. The implicit representation simplifies the two-step, table-lookup computation. It eliminates the memory references entailed in that computation and allows other specializations. The resulting scanner has the same functionality as the table-driven scanner, but with a lower overhead per character. A direct-coded scanner is no harder to generate than the equivalent table-driven scanner.

The table-driven scanner spends most of its time inside the central while loop; thus, the heart of a direct-coded scanner is an alternate implementation of that while loop. With some detail abstracted, that loop performs the following actions:

```
while (state ≠ s_e) do
  NextChar(char);
  cat ← CharCat[char];
  state ← δ[state,cat];
end;
```

REPRESENTING STRINGS

The scanner classifies words in the input program into a small set of categories. From a functional perspective, each word in the input stream becomes a pair ⟨*word*,*type*⟩, where *word* is the actual text that forms the word and *type* represents its syntactic category.

For many categories, having both *word* and *type* is redundant. The words +, ×, and `for` have only one spelling. For identifiers, numbers, and character strings, however, the compiler will repeatedly use the *word*. Unfortunately, many compilers are written in languages that lack an appropriate representation for the *word* part of the pair. We need a representation that is compact and offers a fast equality test for two words.

A common practice to address this problem has the scanner create a single hash table (see Appendix B.4) to hold all the distinct strings used in the input program. The compiler then uses either the string's index in this "string table" or a pointer to its stored image in the string table as a proxy for the string. Information derived from the string, such as the length of a character constant or the value and type of a numerical constant, can be computed once and referenced quickly through the table. Since most computers have storage-efficient representations for integers and pointers, this reduces the amount of memory used internally in the compiler. By using the hardware comparison mechanisms on the integer or pointer proxies, it also simplifies the code used to compare them.

Notice the variable `state` that explicitly represents the DFA's current state and the tables `CharCat` and δ that represent the DFA's transition diagram.

Overhead of Table Lookup

For each character, the table-driven scanner performs two table lookups, one in `CharCat` and another in δ. While both lookups take $\mathbf{O}(1)$ time, the table abstraction imposes constant-cost overheads that a direct-coded scanner can avoid. To access the i^{th} element of `CharCat`, the code must compute its address, given by

$$@CharCat_0 + i \times w$$

Detailed discussion of code for array addressing starts on page 359 in Section 7.5.

where $@CharCat_0$ is a constant related to the starting address of `CharCat` in memory and w is the number of bytes in each element of `CharCat`. After computing the address, the code must load the data found at that address in memory.

Because δ has two dimensions, the address calculation is more complex. For the reference δ(state,cat), the code must compute

$$@\delta_0 + (state \times number\ of\ columns\ in\ \delta + cat) \times w$$

where $@\delta_0$ is a constant related to the starting address of δ in memory and w is the number of bytes per element of δ. Again, the scanner must issue a load operation to retrieve the data stored at this address.

Thus, the table-driven scanner performs two address computations and two load operations for each character that it processes. The speed improvements in a direct-coded scanner come from reducing this overhead.

Replacing the Table-Driven Scanner's While Loop

Rather than represent the current DFA state and the transition diagram explicitly, a direct-coded scanner has a specialized code fragment to implement each state. It transfers control directly from state-fragment to state-fragment to emulate the actions of the DFA. Figure 2.16 shows a direct-coded scanner

```
s_init : lexeme ← " ";
         clear stack;
         push(bad);
         goto s0 ;

s0 :     NextChar(char);
         lexeme ← lexeme + char;
         if state ∈ SA
             then clear stack;
         push(state);
         if (char = 'r')
            then goto s1 ;
            else goto sout ;

s1 :     NextChar(char);
         lexeme ← lexeme + char;
         if state ∈ SA
             then clear stack;
         push(state);
         if ('0' ≤ char ≤ '9')
             then goto s2 ;
             else goto sout ;

s2 :     NextChar(char);
         lexeme ← lexeme + char;
         if state ∈ SA
             then clear stack;
         push(state);
         if '0' ≤ char ≤ '9'
            then goto s2 ;
            else goto sout

sout :   while (state ∉ SA and
                state ≠ bad) do
            state ← pop();
            truncate lexeme;
            RollBack();
         end;
         if state ∈ SA
            then return Type[state];
            else return invalid;
```

■ **FIGURE 2.16** A Direct-Coded Scanner for $r[0...9]^+$.

for $r\,[0\ldots9]^+$; it is equivalent to the table-driven scanner shown earlier in Figure 2.14.

Consider the code for state s_1. It reads a character, concatenates it onto the current word, and advances the character counter. If `char` is a digit, it jumps to state s_2. Otherwise, it jumps to state s_{out}. The code requires no complicated address calculations. The code refers to a tiny set of values that can be kept in registers. The other states have equally simple implementations.

The code in Figure 2.16 uses the same mechanism as the table-driven scanner to track accepting states and to roll back to them after an overrun. Because the code represents a specific DFA, we could specialize it further. In particular, since the DFA has just one accepting state, the stack is unneeded and the transitions to s_{out} from s_0 and s_1 can be replaced with `report failure`. In a DFA where some transition leads from an accepting state to a nonaccepting state, the more general mechanism is needed.

A scanner generator can directly emit code similar to that shown in Figure 2.16. Each state has a couple of standard assignments, followed by branching logic that implements the transitions out of the state. Unlike the table-driven scanner, the code changes for each set of RES. Since that code is generated directly from the RES, the difference should not matter to the compiler writer.

Code in the style of Figure 2.16 is often called *spaghetti code* in honor of its tangled control flow.

Of course, the generated code violates many of the precepts of structured programming. While small examples may be comprehensible, the code for a complex set of regular expressions may be difficult for a human to follow. Again, since the code is generated, humans should not need to read or debug it. The additional speed obtained from direct coding makes it an attractive option, particularly since it entails no extra work for the compiler writer. Any extra work is pushed into the implementation of the scanner generator.

Classifying Characters

The continuing example, $r\,[0\ldots9]^+$, divides the alphabet of input characters into just four classes. An `r` falls in class *Register*. The digits `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, and `9` fall in class *Digit*, the special character returned when `NextChar` exhausts its input falls in class *EndOfFile*, and anything else falls in class *Other*.

Collating sequence
the "sorting order" of the characters in an alphabet, determined by the integers assigned each character

The scanner can easily and efficiently classify a given character, as shown in Figure 2.16. State s_0 uses a direct test on '`r`' to determine if `char` is in *Register*. Because all the other classes have equivalent actions in the DFA, the scanner need not perform further tests. States s_1 and s_2 classify

`char` into either *Digit* or anything else. They capitalize on the fact that the digits `0` through `9` occupy adjacent positions in the ASCII collating sequence, corresponding to the integers 48 to 57.

In a scanner where character classification is more involved, the translation-table approach used in the table-driven scanner may be less expensive than directly testing characters. In particular, if a class contains multiple characters that do not occupy adjacent slots in the collating sequence, a table lookup may be more efficient than direct testing. For example, a class that contained the arithmetic operators +, -, *, \, and ^ (43, 45, 42, 48, and 94 in the ASCII sequence) would require a moderately long series of comparisons. Using a translation table, such as `CharCat` in the table-driven example, might be faster than the comparisons if the translation table stays in the processor's primary cache.

2.5.3 Hand-Coded Scanners

Generated scanners, whether table-driven or direct-coded, use a small, constant amount of time per character. Despite this fact, many compilers use hand-coded scanners. In an informal survey of commercial compiler groups, we found that a surprisingly large fraction used hand-coded scanners. Similarly, many of the popular open-source compilers rely on hand-coded scanners. For example, the *flex* scanner generator was ostensibly built to support the *gcc* project, but *gcc 4.0* uses hand-coded scanners in several of its front ends.

The direct-coded scanner reduced the overhead of simulating the DFA; the hand-coded scanner can reduce the overhead of the interfaces between the scanner and the rest of the system. In particular, a careful implementation can improve the mechanisms used to read and manipulate characters on input and the operations needed to produce a copy of the actual lexeme on output.

Buffering the Input Stream

While character-by-character I/O leads to clean algorithmic formulations, the overhead of a procedure call per character is significant relative to the cost of simulating the DFA in either a table-driven or a direct-coded scanner. To reduce the I/O cost per character, the compiler writer can use buffered I/O, where each read operation returns a longer string of characters, or buffer, and the scanner then indexes through the buffer. The scanner maintains a pointer into the buffer. Responsibility for keeping the buffer filled and tracking the current location in the buffer falls to `NextChar`. These operations can

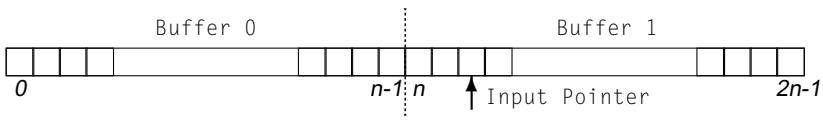
be performed inline; they are often encoded in a macro to avoid cluttering the code with pointer dereferences and increments.

The cost of reading a full buffer of characters has two components, a large fixed overhead and a small per-character cost. A buffer and pointer scheme amortizes the fixed costs of the read over many single-character fetches. Making the buffer larger reduces the number of times that the scanner incurs this cost and reduces the per-character overhead.

Using a buffer and pointer also leads to a simple and efficient implementation of the `RollBack` operation that occurs at the end of both the generated scanners. To roll the input back, the scanner can simply decrement the input pointer. This scheme works as long as the scanner does not decrement the pointer beyond the start of the buffer. At that point, however, the scanner needs access to the prior contents of the buffer.

Double buffering
A scheme that uses two input buffers in a modulo fashion to provide bounded roll back is often called *double buffering*.

In practice, the compiler writer can bound the roll-back distance that a scanner will need. With bounded roll back, the scanner can simply use two adjacent buffers and increment the pointer in a modulo fashion, as shown below:



To read a character, the scanner increments the pointer, modulo *2n* and returns the character at that location. To roll back a character, the program decrements the input pointer, modulo *2n*. It must also manage the contents of the buffer, reading additional characters from the input stream as needed.

Both `NextChar` and `RollBack` have simple, efficient implementations, as shown in Figure 2.17. Each execution of `NextChar` loads a character, increments the `Input` pointer, and tests whether or not to fill the buffer. Every `n` characters, it fills the buffer. The code is small enough to be included inline, perhaps generated from a macro. This scheme amortizes the cost of filling the buffer over `n` characters. By choosing a reasonable size for `n`, such as 2048, 4096, or more, the compiler writer can keep the I/O overhead low.

`Rollback` is even less expensive. It performs a test to ensure that the buffer contents are valid and then decrements the input pointer. Again, the implementation is sufficiently simple to be expanded inline. (If we used this implementation of `NextChar` and `RollBack` in the generated scanners, `RollBack` would need to truncate the final character away from `lexeme`.)

```
Char ← Buffer[Input];
Input ← (Input+1) mod 2n;

if (Input mod n = 0)
    then begin;
       fill Buffer[Input:Input+n-1];
       Fence ← (Input+n) mod 2n;
   end;

return Char;
```

Implementing *NextChar*

```
Input ← 0;
Fence ← 0;
fill Buffer[0:n];
```

Initialization

```
if (Input = Fence)
   then signal roll back error;
Input ← (Input-1) mod 2n;
```

Implementing *RollBack*

■ **FIGURE 2.17** Implementing *NextChar* and *RollBack*.

As a natural consequence of using finite buffers, *RollBack* has a limited history in the input stream. To keep it from decrementing the pointer beyond the start of that context, *NextChar* and *RollBack* cooperate. The pointer *Fence* always indicates the start of the valid context. *NextChar* sets *Fence* each time it fills a buffer. *RollBack* checks *Fence* each time it tries to decrement the *Input* pointer.

After a long series of *NextChar* operations, say, more than *n* of them, *RollBack* can always back up at least *n* characters. However, a sequence of calls to *NextChar* and *RollBack* that work forward and backward in the buffer can create a situation where the distance between *Input* and *Fence* is less than *n*. Larger values of *n* decrease the likelihood of this situation arising. Expected backup distances should be a consideration in selecting the buffer size, *n*.

Generating Lexemes

The code shown for the table-driven and direct-coded scanners accumulated the input characters into a string *lexeme*. If the appropriate output for each syntactic category is a textual copy of the lexeme, then those schemes are efficient. In some common cases, however, the parser, which consumes the scanner's output, needs the information in another form.

For example, in many circumstances, the natural representation for a register number is an integer, rather than a character string consisting of an 'r' and a sequence of digits. If the scanner builds a character representation, then somewhere in the interface, that string must be converted to an integer. A typical way to accomplish that conversion uses a library routine, such as `atoi` in the standard C library, or a string-based I/O routine, such as

sscanf. A more efficient way to solve this problem would be to accumulate the integer's value one digit at a time.

In the continuing example, the scanner could initialize a variable, *RegNum*, to zero in its initial state. Each time that it recognized a digit, it could multiply *RegNum* by 10 and add the new digit. When it reached an accepting state, *RegNum* would contain the needed value. To modify the scanner in Figure 2.16, we can delete all statements that refer to *lexeme*, add *RegNum* ← *0;* to s_{init}, and replace the occurrences of goto s_2 in states s_1 and s_2 with:

```
begin;
  RegNum ← RegNum × 10 + (char - '0');
  goto s2;
end;
```

where both *char* and *'0'* are treated as their ordinal values in the ASCII collating sequence. Accumulating the value this way likely has lower overhead than building the string and converting it in the accepting state.

For other words, the lexeme is implicit and, therefore, redundant. With singleton words, such as a punctuation mark or an operator, the syntactic category is equivalent to the lexeme. Similarly, many scanners recognize comments and white space and discard them. Again, the set of states that recognize the comment need not accumulate the lexeme. While the individual savings are small, the aggregate effect is to create a faster, more compact scanner.

This issue arises because many scanner generators let the compiler writer specify actions to be performed in an accepting state, but do not allow actions on each transition. The resulting scanners must accumulate a character copy of the lexeme for each word, whether or not that copy is needed. If compile time matters (and it should), then attention to such minor algorithmic details leads to a faster compiler.

2.5.4 Handling Keywords

We have consistently assumed that keywords in the input language should be recognized by including explicit REs for them in the description that generates the DFA and the recognizer. Many authors have proposed an alternative strategy: having the DFA classify them as identifiers and testing each identifier to determine whether or not it is a keyword.

This strategy made sense in the context of a hand-implemented scanner. The additional complexity added by checking explicitly for keywords causes

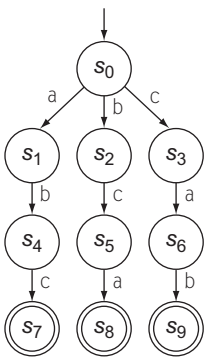
a significant expansion in the number of DFA states. This added implementation burden matters in a hand-coded program. With a reasonable hash table (see Appendix B.4), the expected cost of each lookup should be constant. In fact, this scheme has been used as a classic application for *perfect hashing*. In perfect hashing, the implementor ensures, for a fixed set of keys, that the hash function generates a compact set of integers with no collisions. This lowers the cost of lookup on each keyword. If the table implementation takes into account the perfect hash function, a single probe serves to distinguish keywords from identifiers. If it retries on a miss, however, the behavior can be much worse for nonkeywords than for keywords.

If the compiler writer uses a scanner generator to construct the recognizer, then the added complexity of recognizing keywords in the DFA is handled by the tools. The extra states that this adds consume memory, but not compile time. Using the DFA mechanism to recognize keywords avoids a table lookup on each identifier. It also avoids the overhead of implementing a keyword table and its support functions. In most cases, folding keyword recognition into the DFA makes more sense than using a separate lookup table.

SECTION REVIEW

Automatic construction of a working scanner from a minimal DFA is straightforward. The scanner generator can adopt a table-driven approach, wherein it uses a generic skeleton scanner and language-specific tables, or it can generate a direct-coded scanner that threads together a code fragment for each DFA state. In general, the direct-coded approach produces a faster scanner because it has lower overhead per character.

Despite the fact that all DFA-based scanners have small constant costs per characters, many compiler writers choose to hand code a scanner. This approach lends itself to careful implementation of the interfaces between the scanner and the I/O system and between the scanner and the parser.



Review Questions

1. Given the DFA shown to the left, complete the following:
   a. Sketch the character classifier that you would use in a table-driven implementation of this DFA.
   b. Build the transition table, based on the transition diagram and your character classifier.
   c. Write an equivalent direct-coded scanner.

2. An alternative implementation might use a recognizer for $(a|b|c)\,(a|b|c)\,(a|b|c)$, followed by a lookup in a table that contains the three words `abc`, `bca`, and `cab`.
   a. Sketch the DFA for this language.
   b. Show the direct-coded scanner, including the call needed to perform keyword lookup.
   c. Contrast the cost of this approach with those in question 1 above.
3. What impact would the addition of transition-by-transition actions have on the DFA-minimization process? (Assume that we have a linguistic mechanism of attaching code fragments to the edges in the transition graph.)

2.6 ADVANCED TOPICS

2.6.1 DFA to Regular Expression

The final step in the cycle of constructions, shown in Figure 2.3, is to construct an RE from a DFA. The combination of Thompson's construction and the subset construction provide a constructive proof that DFAs are at least as powerful as REs. This section presents Kleene's construction, which builds an RE to describe the set of strings accepted by an arbitrary DFA. This algorithm establishes that REs are at least as powerful as DFAs. Together, they show that REs and DFAs are equivalent.

Consider the transition diagram of a DFA as a graph with labelled edges. The problem of deriving an RE that describes the language accepted by the DFA corresponds to a path problem over the DFA's transition diagram. The set of strings in $L(\text{DFA})$ consists of the set of edge labels for every path from d_0 to d_i, $\forall\ d_i \in D_A$. For any DFA with a cyclic transition graph, the set of such paths is infinite. Fortunately, REs have the Kleene closure operator to handle this case and summarize the complete set of subpaths created by a cycle.

Figure 2.18 shows one algorithm to compute this path expression. It assumes that the DFA has states numbered from 0 to $|D| - 1$, with d_0 as the start state. It generates an expression that represents the labels along all paths between two nodes, for each pair of nodes in the transition diagram. As a final step, it combines the expressions for each path that leaves d_0 and reaches some accepting state, $d_i \in D_A$. In this way, it systematically constructs the path expressions for all paths.

The algorithm computes a set of expressions, denoted R_{ij}^k, for all the relevant values of i, j, and k. R_{ij}^k is an expression that describes all paths through the transition graph from state i to state j, without going through a state

```
for i = 0 to |D| − 1
    for j = 0 to |D| − 1
        R_ij^-1 = { a | δ(d_i, a) = d_j }
        if (i = j) then
            R_ij^-1 = R_ij^-1 | { ε }
for k = 0 to |D|−1
    for i = 0 to |D |−1
        for j = 0 to |D|−1
            R_ij^k = R_ik^(k−1) (R_kk^(k−1))* R_kj^(k−1) | R_ij^(k−1)
L = |_{s_j ∈ D_A}  R_0j^(|D|−1)
```

■ **FIGURE 2.18** Deriving a Regular Expression from a DFA.

numbered higher than k. Here, *through* means both entering and leaving, so that $R^2_{1,16}$ can be nonempty if an edge runs directly from 1 to 16.

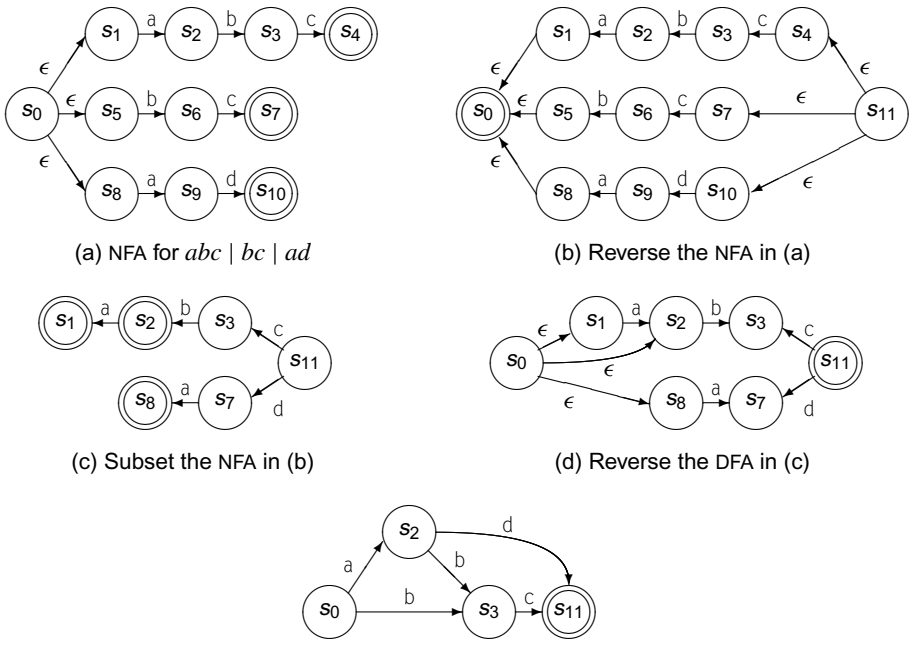
Initially, the algorithm places all of the direct paths from i to j in R^{-1}_{ij}, with $\{\epsilon\}$ added to R^{-1}_{ij} if $i = j$. Over successive iterations, it builds up longer paths to produce R^k_{ij} by adding to R^{k-1}_{ij} the paths that pass through k on their way from i to j. Given R^{k-1}_{ij}, the set of paths added by going from $k-1$ to k is exactly the set of paths that run from i to k using no state higher than $k-1$, concatenated with the paths from k to itself that pass through no state higher than $k-1$, followed by the paths from k to j that pass through no state higher than $k-1$. That is, each iteration of the loop on k adds the paths that pass through k to each set R^{k-1}_{ij} to produce R^k_{ij}.

Traditional statements of this algorithm assume that node names range from 1 to n, rather than from 0 to $n-1$. Thus, they place the direct paths in R^0_{ij}.

When the k loop terminates, the various R^k_{ij} expressions account for all paths through the graph. The final step computes the set of paths that start with d_0 and end in some accepting state, $d_j \in d_A$, as the alternation of the path expressions.

2.6.2 Another Approach to DFA Minimization: Brzozowski's Algorithm

If we apply the subset construction to an NFA that has multiple paths from the start state for some prefix, the construction will group the states involved in those duplicate prefix paths together and will create a single path for that prefix in the DFA. The subset construction always produces DFAs that have no duplicate prefix paths. Brzozowski used this observation to devise an alternative DFA minimization algorithm that directly constructs the minimal DFA from an NFA.



■ **FIGURE 2.19** Minimizing a DFA with Brzozowski's Algorithm.

For an NFA n, let *reverse(n)* be the NFA obtained by reversing the direction of all the transitions, making the initial state into a final state, adding a new initial state, and connecting it to all of the states that were final states in n. Further, let *reachable(n)* be a function that returns the set of states and transitions in n that are reachable from its initial state. Finally, let *subset(n)* be the DFA produced by applying the subset construction to n.

Now, given an NFA n, the minimal equivalent DFA is just

reachable(subset(reverse(reachable(subset(reverse(n)))))).

The inner application of *subset* and *reverse* eliminates duplicate suffixes in the original NFA. Next, *reachable* discards any states and transitions that are no longer interesting. Finally, the outer application of the triple, *reachable, subset*, and *reverse*, eliminates any duplicate prefixes in the NFA. (Applying *reverse* to a DFA can produce an NFA.)

The example in Figure 2.19 shows the steps of the algorithm on a simple NFA for the RE *abc | bc | ad*. The NFA in Figure 2.19a is similar to the one that Thompson's construction would produce; we have removed the ϵ-transitions that "glue" together the NFAs for individual letters. Figure 2.19b

shows the result of applying *reverse* to that NFA. Figure 2.19c depicts the DFA that *subset* constructs from the *reverse* of the NFA. At this point, the algorithm applies *reachable* to remove any unreachable states; our example NFA has none. Next, the algorithm applies *reverse* to the DFA, which produces the NFA in Figure 2.19d. Applying *subset* to that NFA produces the DFA in Figure 2.19e. Since it has no unreachable states, it is the minimal DFA for *abc* | *bc* | *cd*.

This technique looks expensive, because it applies *subset* twice and we know that *subset* can construct an exponentially large set of states. Studies of the running times of various FA minimization techniques suggest, however, that this algorithm performs reasonably well, perhaps because of specific properties of the NFA produced by the first application of *reachable (subset(reverse(n)))*. From a software-engineering perspective, it may be that implementing *reverse* and *reachable* is easier than debugging the partitioning algorithm.

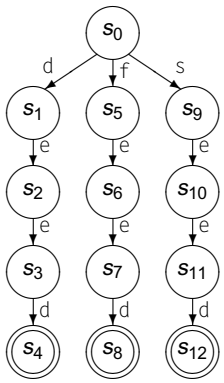
2.6.3 Closure-Free Regular Expressions

One subclass of regular languages that has practical application beyond scanning is the set of languages described by closure-free regular expressions. Such REs have the form $w_1 \mid w_2 \mid w_3 \mid \ldots \mid w_n$ where the individual words, w_i, are just concatenations of characters in the alphabet, Σ. These REs have the property that they produce DFAs with acyclic transition graphs.

These simple regular languages are of interest for two reasons. First, many pattern recognition problems can be described with a closure-free RE. Examples include words in a dictionary, URLs that should be filtered, and keys to a hash table. Second, the DFA for a closure-free RE can be built in a particularly efficient way.

To build the DFA for a closure-free RE, begin with a start state s_0. To add a word to the existing DFA, the algorithm follows the path for the new word until it either exhausts the pattern or finds a transition to s_e. In the former case, it designates the final state for the new word as an accepting state. In the latter, it adds a path for the new word's remaining suffix. The resulting DFA can be encoded in tabular form or in direct-coded form (see Section 2.5.2). Either way, the recognizer uses constant time per character in the input stream.

In this algorithm, the cost of adding a new word to an existing DFA is proportional to the length of the new word. The algorithm also works incrementally; an application can easily add new words to a DFA that is in use. This property makes the acyclic DFA an interesting alternative for



implementing a perfect hash function. For a small set of keys, this technique produces an efficient recognizer. As the number of states grows (in a direct-coded recognizer) or as key length grows (in a table-driven recognizer), the implementation may slow down due to cache-size constraints. At some point, the impact of cache misses will make an efficient implementation of a more traditional hash function more attractive than incremental construction of the acyclic DFA.

The DFAs produced in this way are not guaranteed to be minimal. Consider the acyclic DFA that it would produce for the REs *deed, feed*, and *seed*, shown to the left. It has three distinct paths that each recognize the suffix *eed*. Clearly, those paths can be combined to reduce the number of states and transitions in the DFA. Minimization will combine states (s_2, s_6, s_{10}), states (s_3, s_7, s_{11}), and states (s_4, s_8, s_{12}) to produce a seven state DFA.

The algorithm builds DFAs that are minimal with regard to prefixes of words in the language. Any duplication takes the form of multiple paths for the same suffix.

2.7 CHAPTER SUMMARY AND PERSPECTIVE

The widespread use of regular expressions for searching and scanning is one of the success stories of modern computer science. These ideas were developed as an early part of the theory of formal languages and automata. They are routinely applied in tools ranging from text editors to web filtering engines to compilers as a means of concisely specifying groups of strings that happen to be regular languages. Whenever a finite collection of words must be recognized, DFA-based recognizers deserve serious consideration.

The theory of regular expressions and finite automata has developed techniques that allow the recognition of regular languages in time proportional to the length of the input stream. Techniques for automatic derivation of DFAs from REs and for DFA minimization have allowed the construction of robust tools that generate DFA-based recognizers. Both generated and hand-crafted scanners are used in well-respected modern compilers. In either case, a careful implementation should run in time proportional to the length of the input stream, with a small overhead per character.

■ CHAPTER NOTES

Originally, the separation of lexical analysis, or scanning, from syntax analysis, or parsing, was justified with an efficiency argument. Since the cost

of scanning grows linearly with the number of characters, and the constant costs are low, pushing lexical analysis from the parser into a separate scanner lowered the cost of compiling. The advent of efficient parsing techniques weakened this argument, but the practice of building scanners persists because it provides a clean separation of concerns between lexical structure and syntactic structure.

Because scanner construction plays a small role in building an actual compiler, we have tried to keep this chapter brief. Thus, the chapter omits many theorems on regular languages and finite automata that the ambitious reader might enjoy. The many good texts on this subject can provide a much deeper treatment of finite automata and regular expressions, and their many useful properties [194, 232, 315].

Kleene [224] established the equivalence of REs and FAs. Both the Kleene closure and the DFA to RE algorithm bear his name. McNaughton and Yamada showed one construction that relates REs to NFAs [262]. The construction shown in this chapter is patterned after Thompson's work [333], which was motivated by the implementation of a textual search command for an early text editor. Johnson describes the first application of this technology to automate scanner construction [207]. The subset construction derives from Rabin and Scott [292]. The DFA minimization algorithm in Section 2.4.4 is due to Hopcroft [193]. It has found application to many different problems, including detecting when two program variables always have the same value [22].

The idea of generating code rather than tables, to produce a direct-coded scanner, appears to originate in work by Waite [340] and Heuring [189]. They report a factor of five improvement over table-driven implementations. Ngassam et al. describe experiments that characterize the speedups possible in hand-coded scanners [274]. Several authors have examined tradeoffs in scanner implementation. Jones [208] advocates direct coding but argues for a structured approach to control flow rather than the spaghetti code shown in Section 2.5.2. Brouwer et al. compare the speed of 12 different scanner implementations; they discovered a factor of 70 difference between the fastest and slowest implementations [59].

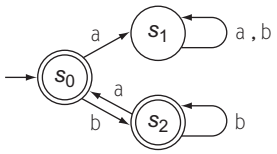
The alternative DFA minimization technique presented in Section 2.6.2 was described by Brzozowski in 1962 [60]. Several authors have compared DFA minimization techniques and their performance [328, 344]. Many authors have looked at the construction and minimization of acyclic DFAs [112, 343, 345].

■ EXERCISES

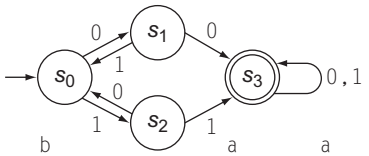
Section 2.2

1. Describe informally the languages accepted by the following FAs:

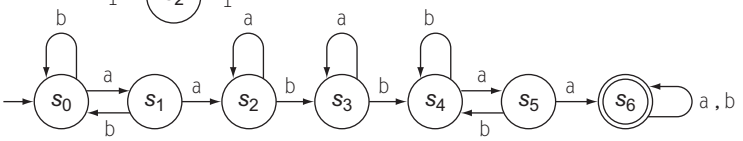
a.



b.



c.



2. Construct an FA accepting each of the following languages:
- **a.** $\{w \in \{a, b\}^* \mid w$ starts with 'a' and contains '$baba$' as a substring$\}$
- **b.** $\{w \in \{0, 1\}^* \mid w$ contains '111' as a substring and does not contain '00' as a substring$\}$
- **c.** $\{w \in \{a, b, c\}^* \mid$ in w the number of 'a's modulo 2 is equal to the number of 'b's modulo 3$\}$

3. Create FAs to recognize (a) words that represent complex numbers and (b) words that represent decimal numbers written in scientific notation.

Section 2.3

4. Different programming languages use different notations to represent integers. Construct a regular expression for each one of the following:
- **a.** Nonnegative integers in C represented in bases 10 and 16.
- **b.** Nonnegative integers in VHDL that may include underscores (an underscore cannot occur as the first or last character).
- **c.** Currency, in dollars, represented as a positive decimal number rounded to the nearest one-hundredth. Such numbers begin with the character $, have commas separating each group of three digits to the left of the decimal point, and end with two digits to the right of the decimal point, for example, $8,937.43 and $7,777,777.77.

Hint
Not all the specifications describe regular languages.

5. Write a regular expression for each of the following languages:
- **a.** Given an alphabet $\Sigma = \{0, 1\}$, L is the set of all strings of alternating pairs of 0s and pairs of 1s.

b. Given an alphabet $\Sigma = \{0, 1\}$, L is the set of all strings of 0s and 1s that contain an even number of 0s or an even number of 1s.

c. Given the lowercase English alphabet, L is the set of all strings in which the letters appear in ascending lexicographical order.

d. Given an alphabet $\Sigma = \{a, b, c, d\}$, L is the set of strings *xyzwy*, where *x* and *w* are strings of one or more characters in Σ, *y* is any single character in Σ, and *z* is the character z, taken from outside the alphabet. (Each string xyzwy contains two words *xy* and *wy* built from letters in Σ. The words end in the same letter, *y*. They are separated by *z*.)

e. Given an alphabet $\Sigma = \{+, -, \times, \div, (,), \texttt{id}\}$, L is the set of algebraic expressions using addition, subtraction, multiplication, division, and parentheses over `id`s.

6. Write a regular expression to describe each of the following programming language constructs:

a. Any sequence of tabs and blanks (sometimes called *white space*)

b. Comments in the programming language C

c. String constants (without escape characters)

d. Floating-point numbers

7. Consider the three regular expressions: Section 2.4

$(ab \mid ac)^*$

$(0 \mid 1)^* \; 1100 \;\; 1^*$

$(01 \mid 10 \mid 00)^* \;\; 11$

a. Use Thompson's construction to construct an NFA for each RE.

b. Convert the NFAs to DFAs.

c. Minimize the DFAs.

8. One way of proving that two REs are equivalent is to construct their minimized DFAs and then compare them. If they differ only by state names, then the REs are equivalent. Use this technique to check the following pairs of REs and state whether or not they are equivalent.

a. $(0 \mid 1)^*$ and $(0^* \mid 10^*)^*$

b. $(ba)^+ \;\; (a^* \;\; b^* \mid a^*)$ and $(ba)^* \;\; ba^+ \;\; (b^* \mid \epsilon)$

9. In some cases, two states connected by an ϵ-move can be combined.

a. Under what set of conditions can two states connected by an ϵ-move be combined?

b. Give an algorithm for eliminating ϵ-moves.

c. How does your algorithm relate to the ϵ-closure function used to implement the subset construction?

10. Show that the set of regular languages is closed under intersection.

11. The DFA minimization algorithm given in Figure 2.9 is formulated to enumerate all the elements of `P` and all of the characters in Σ on each iteration of the while loop.

a. Recast the algorithm so that it uses a worklist to hold the sets that must still be examined.

b. Recast the `Split` function so that it partitions the set around all of the characters in Σ.

c. How does the expected case complexity of your modified algorithms compare to the expected case complexity of the original algorithm?

Section 2.5

12. Construct a DFA for each of the following C language constructs, and then build the corresponding table for a table-driven implementation for each of them:

a. Integer constants

b. Identifiers

c. Comments

13. For each of the DFAs in the previous exercise, build a direct-coded scanner.

14. This chapter describes several styles of DFA implementations. Another alternative would use mutually recursive functions to implement a scanner. Discuss the advantages and disadvantages of such an implementation.

15. To reduce the size of the transition table, the scanner generator can use a character classification scheme. Generating the classifier table, however, seems expensive. The obvious algorithm would require $\mathbf{O}(|\Sigma|^2 \cdot |states|)$ time. Derive an asymptotically faster algorithm for finding identical columns in the transition table.

16. Figure 2.15 shows a scheme that avoids quadratic roll back behavior in a scanner built by simulating a DFA. Unfortunately, that scheme requires that the scanner know in advance the length of the input stream and that it maintain a bit-matrix, `Failed`, of size $|states| \times |input|$. Devise a scheme that uses less space than the scheme shown in Figure 2.15. How does your technique affect the size of the `Failed` table in cases where the worst case input does not occur?