Engineering a Compiler

Manuscript for the Third Edition (EaC 3e)

Keith D. Cooper
Linda Torczon

Rice University
Houston, Texas

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 the three-part process that the compiler's front end uses to understand the input program. The scanner, or lexical analyzer, reads a stream of characters and produces a stream of words. The parser, or syntax analyzer, fits that stream of words to a grammatical model of the input language. The parser, in turn, invokes semantic elaboration routines to perform deeper analysis and to build structures that model the input program's actual meaning.

Syntactic category: a classification of words according to their grammatical usage

The scanner aggregates characters into words. For each word, it determines if the word is valid in the source language. For each valid word, it assigns the word 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 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 both syntax and recognizers. In Section 2.4, we show a set of constructions to convert a regular expression into a recognizer. Finally, in Section 2.5, we examine practical issues in scanner implementation, for both generated and hand-crafted scanners.

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, many 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 of words, each labelled with its *syntactic category*—equivalent to a word's *part of speech* in English. Each time it is called, the scanner produces a pair, ⟨*lexeme*,*category*⟩, where *lexeme* is the spelling of the word and *category* is its syntactic category. This pair is sometimes called a *token*.

Lexeme: the actual text for a word recognized by an FA

To find and classify words, the scanner applies a set of rules that describe the lexical structure of the input programming language, sometimes called its *microsyntax*. Microsyntax specifies how to group characters into words and, conversely, how to separate words that run together. (In the context of scanning, punctuation marks and other symbols are treated as separate words.)

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 and its potential parts of speech 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-60 and its descendants define an *identifier* as a single alphabetic character followed by zero or more alphanumeric characters. The identifier ends with the first non-alphanumeric 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`. The techniques used to implement a scanner must be capable of distinguishing individual words, such as keywords, without compromising the efficiency of recognizing the larger classes that contain them.

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 $\mathbf{O}(1)$ time per character, so they run in time proportional to the number of characters in the input stream.

A Few Words About Time

Chapter 2 focuses on the design and implementation of a compiler's scanner. As such, it deals with issues that arise at three different times: design time, build time, and compile time.

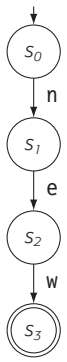
At *design time*, the compiler writer creates specifications for the microsyntax of the programming language (the spelling of words). The compiler writer chooses an implementation method and writes any code that is needed.

At *build time*, the compiler writer invokes tools that build the actual executable scanner. For a specification-driven scanner, this process will include the use of tools to convert the specification into code and the use of a compiler to turn that code into an executable image.

At *compile time*, the end user invokes the compiler to translate application code into executable code. The compiler includes the scanner; it uses the scanner to convert the application code from a

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

(a) Code



(b) Recognizer

FIGURE 2.1 Code Fragment to Recognize "new"

string of characters into a string of words, and to classify each of those words into a syntactic category or part of speech.

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 2.1a. 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 "`new`", then it should report an error. Because scanners rarely recognize only one word, we will leave this "error path" deliberately vague at this point.

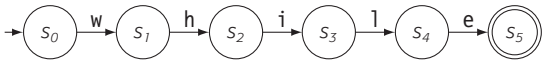
We assume that, once *NextChar* reaches the end of the input file, it always returns `eof`.

The code fragment performs one test per character. We can represent the code fragment using the simple transition diagram shown in panel (b). The transition diagram represents the recognizer. Each labelled circle represents an abstract state in the computation. The initial state, or start state, is s_0. State s_3 is an accepting state; the recognizer reaches s_3 only when the input is "`new`." The double circle denotes s_3 as an accepting state.

Arrows represent transitions from state to state based on input characters. 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 letter `n` takes the recognizer to s_1. The letter

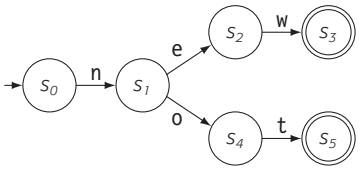
`o` does not match any edge leaving s_1. In the code, cases that do not match `new` *try something else*. In the recognizer, we can think of this action as a transition to an error state. When we draw the transition diagram of a recognizer, we usually omit transitions to the error state. Each state has an implicit 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 the recognizer starts in s_0 and reaches s_5, it has matched the word `while`. The corresponding code fragment would involve five nested *if-then-else* constructs.

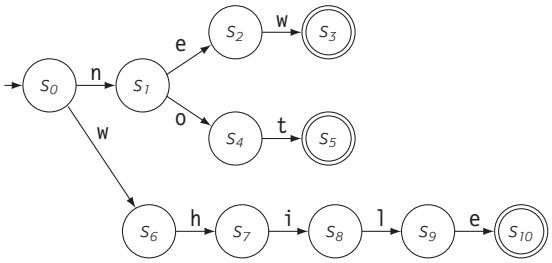
To recognize multiple words, the recognizer can have multiple edges that leave a given state. (In the code, these would become else clauses.) The straightforward recognizer for both `new` and `not` is:



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

Of course, the reader could renumber the recognizer's states without changing its "meaning." Renaming the states produces an equivalent recognizer.

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 the implicit error state, s_e.

The recognizer takes one transition for each input character. Assuming that we implement the recognizer efficiently, we should expect it to run in time proportional to the length of the input string.

2.2.1 A Formalism for Recognizers

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

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

- S is the finite set of states in the recognizer, including 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.

Putting the FA for `new` or `not` or `while` into this formalism yields:

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

δ is only partially specified. 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 error state. This quintuple is equivalent to the transition diagram; given one, we can easily re-create the other.

An FA accepts a string x if and only if, starting in s_0, the sequence of characters in x 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 $s_0 \xrightarrow{\mathtt{n}} s_1$, $s_1 \xrightarrow{\mathtt{e}} s_2$, and $s_2 \xrightarrow{\mathtt{w}} s_3$. Since $s_3 \in S_A$, and no input remains, the FA accepts `new`. For the input string `nut`, the behavior is different. On the letter `n`, the FA takes $s_0 \xrightarrow{\mathtt{n}} s_1$. On `u`, it takes $s_1 \xrightarrow{\mathtt{u}} s_e$. Once the FA enters s_e, it stays in s_e until it exhausts the input stream.

More formally, if the string x consists 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

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

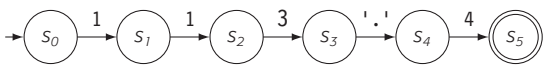
Intuitively, this definition corresponds to a repeated application of δ to a pair composed of some state s and input symbol x_i. The base case, $\delta(s_0, x_1)$, represents the FA's initial transition out of the start state, s_0, on the character x_1. The state $\delta(s_0, x_1)$ is then used as input to δ, along with x_2, which yields 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 $x_1\, x_2\, x_3 \ldots\ x_n$.

The FA can encounter a lexical error in the input. Some character x_j might take it into the error state s_e. Entry into s_e occurs because $x_1\, x_2\, x_3 \ldots\ x_j$ is not a valid prefix for any word in the language accepted by the FA. Alternatively, the FA can reach the end of its input while in a nonaccepting state. Either case indicates that the input string is not a word in the language.

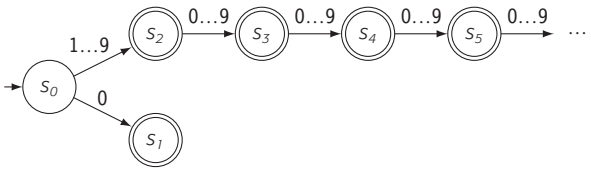
Consider the set of strings that cause an FA to halt in a nonaccepting state. If the FA passed through an accepting state on the way, then the string contains a prefix that is a valid word. As we will see in Section 2.4.5, scanners use this observation to find word boundaries.

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, we will 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?



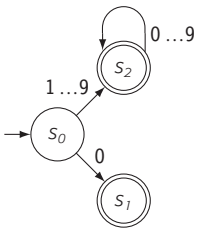
```
state ← s0;
while (state ≠ se and char ≠ eof) do
    char ← NextChar();
    state ← δ(state,char);
end;
if (state ∈ SA) then
    report acceptance;
    else
        report failure;
```

(a) Code to Interpret State Table

$$S = \{s_0, s_1, s_2, s_e\}$$
$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$\delta = \left\{ s_0 \xrightarrow{0} s_1, \quad s_0 \xrightarrow{1\ldots9} s_2, \quad s_2 \xrightarrow{0\ldots9} s_3 \right\}$$
$$S_A = \{s_1, s_2\}$$

(b) Formal Definition of the FA

FIGURE 2.2 A Recognizer for Unsigned Integers



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, violates the stipulation that S is finite.

Notice that 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. 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, as shown in the margin.

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 the code in Figure 2.2a. 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

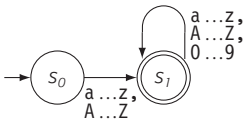
Changing the table allows the same basic code skeleton to implement other recognizers. Notice that this table has ample opportunity for compression. The columns for the digits *1* through *9* are identical, so they could be represented once, reducing the table to three columns: *0*, [*1* . . . *9*], and *other*. The code skeleton reports failure as soon as it enters s_e, so the table row for s_e is never used. It we elide the row for s_e, the table can be represented with three rows and three columns.

We will denote a range of characters with the first and last element, connected by an ellipsis, "…", as in 0…9.

Some systems and authors use a dash rather than an ellipsis. We use the ellipsis to avoid confusion with the minus sign.

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. It can be specified with the simple two-state FA shown in the margin. Many languages include designated special characters, such as _ and &, in the set of alphabetic characters.

FAs can be viewed as specifications for a recognizer. However, they are not particularly concise specifications. To simplify scanner construction, we need a concise notation to specify the lexical structure of words, and a way to turn such a specification into an FA and into code to implement the FA. The remaining sections of this chapter develop precisely those ideas.

The FA for unsigned integers raises the distinction between a syntactic category, such as "unsigned integers," and a specific word, such as 113. Any word that the FA recognizes is a member of the category. Thus, for 113, the scanner should return ⟨"113",unsigned integer⟩.

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. An 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, }

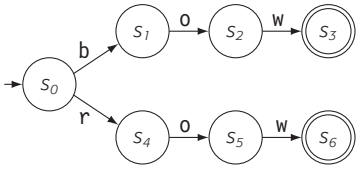
2.3 REGULAR EXPRESSIONS

The set of words accepted by a finite automaton, *F*, forms a language, denoted *L(F)*. The transition diagram of *F* specifies, in precise detail, how to spell every word in that language. Transition diagrams can be complex and non-intuitive. Thus, most systems use a notation called a *regular expression* (RE) to describe spelling. Any language described by an RE is considered a *regular language*.

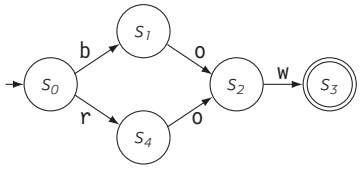
Regular expressions are equivalent to the FAs described in the previous section. (We will prove this with a construction in Section 2.4.) Simple recognizers have simple RE specifications.

- The language consisting of the single word `bow` can be described by an RE written as *bow*. Writing two characters next to each other implies that they are expected to appear in that order.
- The language consisting of the two words `bow` or `row` can be written as *bow* or *row*. To avoid possible misinterpretation of *or*, we write this using the symbol | to mean *or*. Thus, we write the RE as *bow* | *row*. We refer to | as an *alternation*.

For any given language, there may exist multiple REs that specify the language. For example, *bow* | *row* and (*b*|*r*) *ow* both specify the same language. The different REs, in turn, suggest different FAs, as shown below. The FAs shown in panels (a) and (b) accept the same language.



(a) FA suggested by *bow*|*row*



(b) FA suggested by *(b*|*r) ow*

Alternation is commutative, so *bow* | *row* describes the same language as *row* | *bow* or *(r* | *b) ow*.

To make this discussion concrete, consider some examples that occur in most programming languages. Punctuation marks, such as colons, semicolons, and various brackets, are 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:

```
:    ;    ?    =>    (    )    {    }    [    ]
```

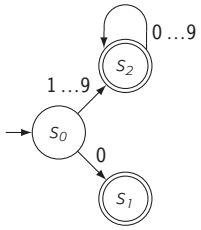
Similarly, keywords have simple REs.

```
if    while    this    integer    instanceof
```

To model syntactic categories that have large numbers of words, such

as integers or identifiers, we need a notation to capture the essence of the cyclic edge in an FA.

The FA for an unsigned integer, shown in the margin, 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:

Kleene Closure: The RE x^* designates zero or more occurrences of x.

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

Of course, we can write this RE more concisely as $0 \mid ([1\ldots9])([0\ldots9])^*$.

2.3.1 Formalizing the Notation

To work with regular expressions in a rigorous way, we need a formal definition. Assume that we have an alphabet, Σ. An RE describes a set of strings over the characters in Σ, plus an additional character ϵ that represents the empty string. The set of strings defined by an RE is called a *language*. We denote the language described by some RE r as $L(r)$.

An RE is built up from three basic operations:

1. *Alternation* The alternation, or union, of two sets of REs, r and s, denoted $r \mid s$, is $\{x \mid x \in L(r) \text{ or } x \in L(s)\}$.
2. *Concatenation* The concatenation of two REs r and s, denoted rs, contains all strings formed by prepending an string from $L(r)$ onto one from $L(s)$, or $\{xy \mid x \in L(r) \text{ and } y \in L(s)\}$.
3. *Closure* The Kleene closure of r, denoted r^*, is $\bigcup_{i=0}^{\infty} r^i$. $L(r^*)$ contains all strings that consist of zero or more words from $L(r)$.

For convenience, we sometimes use a *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 the finite and positive closures can be rewritten in terms of alternation, concatenation and Kleene closure, 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.

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 the QED text editor. From that early genesis, the notation has crept into many 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. (`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.

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

1. If $\mathtt{a} \in \Sigma$, then a is an RE denoting the set $\{\mathtt{a}\}$, and $L(a)$ is `a`.
2. If r and s are REs, denoting sets $L(r)$ and $L(s)$, respectively, then
   $r \mid s$ is an RE denoting the alternation of $L(r)$ and $L(s)$;
   rs is an RE denoting the concatenation of $L(r)$ and $L(s)$; and
   r^* is an RE denoting the Kleene closure of $L(r)$.
3. ϵ is an RE denoting the set that only contains the empty string.

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

2.3.2 Examples of Regular Expressions

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 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 _, %, $, or &, in identifiers.

If the language limits the length of an identifier, we can use a finite closure, as in $([A\ldots Z] \mid [a\ldots z])\ ([A\ldots Z] \mid [a\ldots z] \mid [0\ldots 9])^5$ for a six-character identifier.

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. The simpler specification $[0\ldots 9]^+$ admits a larger class of strings as integers.

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 admit 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 " $(\hat{}")^*$ ".

   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 " $(\hat{}(" \mid \backslash n))^*$ " will recognize a correctly formed C or C++ string and will take an error transition on a string that includes a newline.

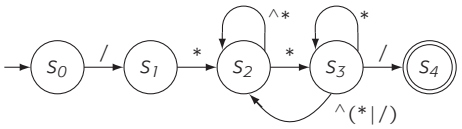
5. Comments appear in a number of forms. C++ and Java offer two ways of writing a comment. The delimiter // indicates a comment that runs to the end of the current input line. The RE for this style of comment is straightforward: // $(\hat{}\backslash n)^*$ \n.

   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 simple: /* $(\hat{}*)^*$ */. With *, the RE is more complex: /* $(\hat{}* \mid *^+\hat{}/)^*$ */. An FA that implement this RE follows.

Complement: The RE $\hat{}c$ specifies the set $\{\Sigma - c\}$, the complement of c with respect to Σ.

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

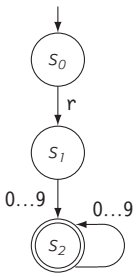
Escape sequence: Two or more characters that the scanner translates into another character. Escape sequences are used for characters that lack a glyph, such as newline or tab, and for ones that occur in the syntax, such as an open or close quote.



The relationship between this RE and this FA is less obvious than in many of the earlier examples.

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.

By contrast, Pascal used single-character comment delimiters: { and }, so a Pascal comment is just { ^}* }.

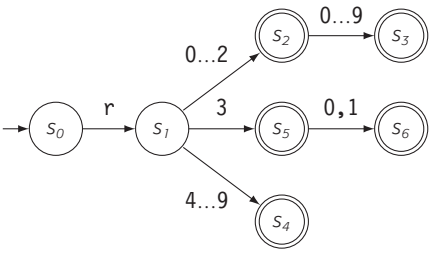


In many cases, tighter specifications lead to more complex regular expressions. Consider, for example, the register specifier in a typical assembly language. It consists of the letter r followed immediately by a small integer. In ILOC, which admits an unlimited set of register names, the RE might be $r[0\ldots9]^+$, which corresponds to the FA shown in the margin. The FA accepts r29 and rejects s29. It also accepts r99999 even though no modern processor has 100,000 registers.

On a typical processor, the set of register names is severely limited—say, to 32, 64, 128, or 256 registers. With a more complex RE, the scanner can check the validity of a register name. For example, the RE

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

specifies a much smaller language. It limits register numbers to the range [0,31] and allows an optional leading zero on single-digit register names. Thus, it accepts r0, r00, r01, and r31, but rejects r001, r32, and r99999. The corresponding FA looks like:



The alternative, of course, it to use the RE $r[0\ldots9]^+$ and to test the register number as an integer.

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 complexity of the RE or the number of states in the FA. More states need more space, but not more time. The build-time cost of *generating* the FA for a more complex RE may be larger, but the cost of operation remains one transition per character. A good implementation will have $O(1)$ cost per transition.

Can we improve the RE for a register name? 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 processors suddenly have 256 or 384 registers, enumeration may become tedious, too.

2.3.3 Closure Properties of REs

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.

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

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.

URL-filtering software relies on this property to build fast FA-based recognizers.

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. Closure ensures that the alternation is an RE and that 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 system-

Programming 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.

By contrast, the words in a programming language are almost always specified lexically. Thus, any string in $[1\ldots9][0\ldots9]^*$ is a positive integer. The RE $[a\ldots z]\,([a\ldots z]\,|\,[0\ldots9])^*$ 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.

atic 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 $L(\text{FA})$, called the complement of $L(\text{FA})$. To build the FA for the complement, we can swap the designation of accepting and nonaccepting states in the original FA. Since FAs and REs are equivalent, this result shows 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 an RE. The next section formalizes that relationship.

REVIEW QUESTIONS

1. A six-character identifier might be specified with a finite closure:

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

   Rewrite the specification using only the three basic RE operations: alternation, concatenation, and Kleene 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 character.

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 to transform an RE into an FA. Kleene's construction, presented in Section 2.6.1, builds an RE from any FA. Together, these constructions form a cycle, shown in Figure 2.3.

Taken together, the cycle of constructions proves that REs and FAs a have equivalent expressive power. That is, an RE can express any language recognizable with an NFA or a DFA, and an FA can recognize any language specifiable with an RE.

The constructions rely on both *nondeterministic* FAs, or NFAs, and *deterministic* FAs, or DFAs. Section 2.4.1 introduces this distinc-

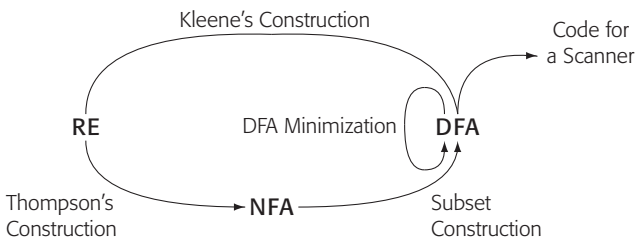


FIGURE 2.3 The Cycle of Constructions

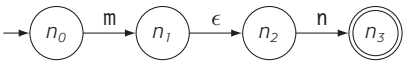
tion. Section 2.4.2 presents Thompson's construction, which builds an NFA directly from an RE. Section 2.4.3 presents the subset construction, which builds a DFA to simulate an NFA. Section 2.4.4 presents Hopcroft's algorithm for DFA minimization; an alternative minimization algorithm by Brozozowski appears in Section 2.6.3.

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 to simplify construction of FAs from REs. For example, assume that we have FAs for m and n, called FA_m and FA_m.

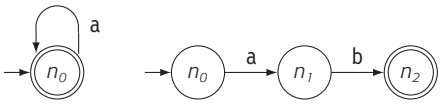
We can build an FA for mn by adding a transition on ϵ from the accepting state of FA_m to the initial state of FA_n, renumbering the states, and using FA_n's accepting state as the accepting state for the new FA.

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

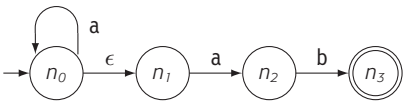


With an ϵ-transition, the definition of acceptance must change slightly to allow one or more ϵ-transitions between any two characters in the input string. For example, in n_1, the FA takes the transition $n_1 \xrightarrow{\epsilon} n_2$ without consuming an input character. The change is minor and intuitive. It does, however, mean that an FA with ϵ-transitions can take multiple transitions per input character.

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 n_0 on the letter a. It can take the transition $n_0 \xrightarrow{\text{a}} n_0$, or the two transitions $n_0 \xrightarrow{\epsilon} n_1$ and $n_1 \xrightarrow{\text{a}} n_2$. Which transition is correct? Consider the strings aab and ab. The FA should accept both strings. For aab, it should move: $n_0 \xrightarrow{\text{a}} n_0$, $n_0 \xrightarrow{\epsilon} n_1$, $n_1 \xrightarrow{\text{a}} n_2$, and $n_2 \xrightarrow{\text{b}} n_3$. For ab, it should move: $n_0 \xrightarrow{\epsilon} n_1$,

$n_1 \xrightarrow{\text{a}} n_2$, and $n_2 \xrightarrow{\text{b}} n_3$.

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 n_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 n_0 violates our notion of the behavior of a sequential algorithm. An FA that includes states such as n_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 and its clones are in some set of states. 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 $(S, \Sigma, \delta, n_0, S_A)$ 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 n_0 and ends in some n_m, $m \geq k$, such that the edge labels along the path match the input string, omitting ϵ's. 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

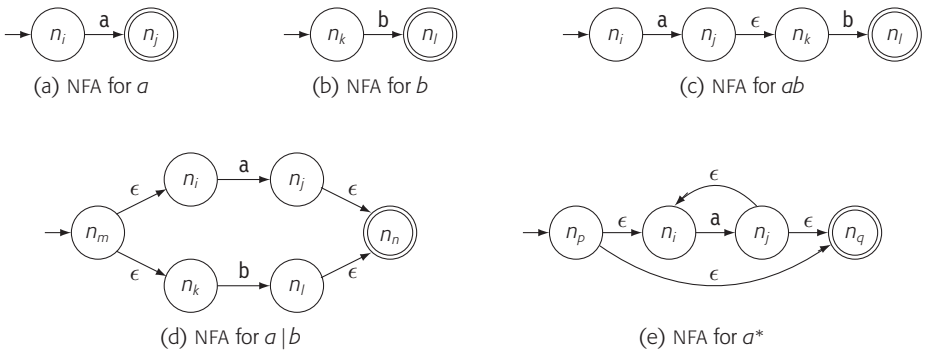
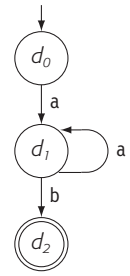


FIGURE 2.4 Trivial NFAs for Regular Expression Operators

some finite set of operating clones. The set of states that those clones occupy form a configuration of the NFA. Each configuration is a subset of N, the set of states of the NFA. The number of such subsets is finite. Thus, an NFA with N states produces at most $|2^N|$ configurations.

Powerset of N: the set of all subsets of N, denoted 2^N.

To simulate the behavior of the NFA, we need a DFA with a state for each configuration of the NFA. The DFA may have exponentially more states than the NFA. Still, the number of configurations and, therefore, DFA states is finite. The resulting DFA still makes one transition per input symbol, so it runs in time proportional to the length of the input string. Thus, the simulation of an NFA on a DFA has a potential space problem, but not a time problem.



Since NFAs and DFAs are equivalent, we can construct a DFA that recognizes a^*ab. The NFA that we saw earlier had two transitions out of n_0 on a. The obvious way to avoid this nondeterministic transition is to observe that a^*ab is equivalent to aa^*b. That RE suggests the DFA shown in the margin. The subset construction automates the transformation of an NFA into a DFA (See Section 2.6).

2.4.2 RE to NFA: Thompson's Construction

The first step in deriving a scanner from an RE constructs an NFA from the RE with *Thompson's construction*. The construction uses a simple, template-driven process to build up an NFA from smaller NFAs. It builds NFAs for individual symbols, $s \in \Sigma$, in the RE and applies transformations on the resulting NFAs to model the effects of concatenation, alternation, and closure. Figure 2.4 shows the trivial NFAs for the a and b, as well as the transformations to form ab, $a \mid b$, and a^* from NFAs for a and b. The transformations apply to arbitrary NFAs.

Figure 2.5 shows the steps that Thompson's construction takes to build an NFA from the RE $a\,(b \mid c)^*$. First, the construction builds trivial

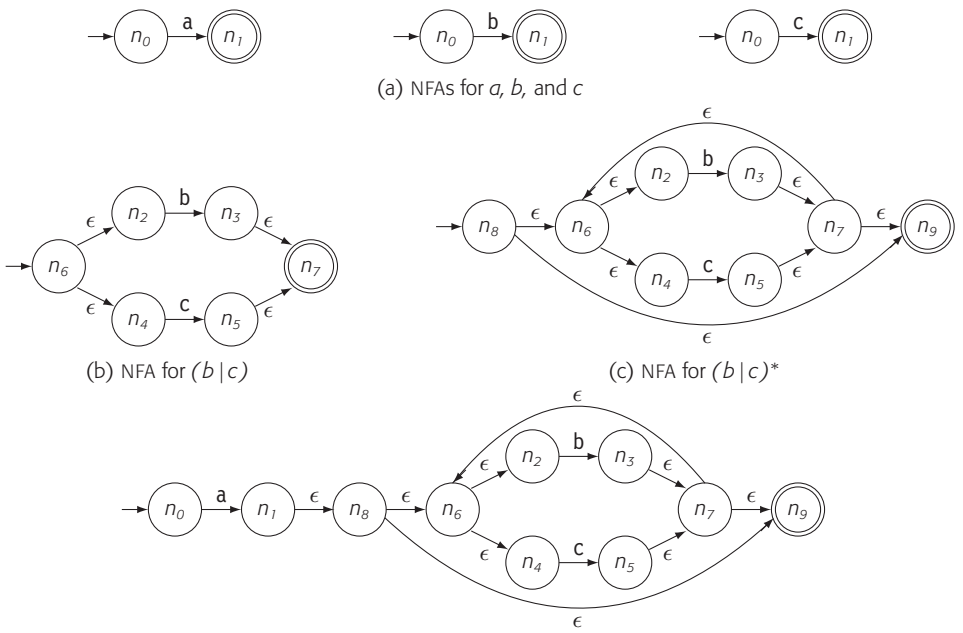


FIGURE 2.5 Applying Thompson's Construction to $a\,(b \mid c)^*$

NFAs for each character in the RE, shown in panel (a). It then applies the operators in precedence order. Next, it builds an NFA for the parenthetic expression, $(b \mid c)$, shown in panel (b). The closure is next, as shown in panel (c). Finally, it concatenates the NFA for a onto the front of the NFA for $(b \mid c)^*$, shown in panel (d). This simple process produces an NFA for an RE written in terms of the three basic operators.

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. 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.

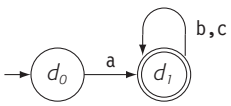


Figure 2.5d shows the NFA that Thompson's construction builds for $a\,(b \mid c)^*$. The combination of the subset construction and DFA minimization should transform the NFA from Figure 2.5d into a DFA similar to the one shown in the margin.

```
q0 ← FollowEpsilon( {n0} ) ;
Q ← q0;
WorkList ← {q0};

while (WorkList ≠ ∅) do
    remove q from WorkList;
    for each character c ∈ Σ do
        temp ← FollowEpsilon(Delta(q, c));
        if temp ∉ Q then
            add temp to both Q and WorkList;
        T[q, c] ← temp;
```

FIGURE 2.6 The Subset Construction

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 building a recognizer from an RE converts the NFA into an equivalent DFA. The algorithm to construct 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 builds a model that captures all of the valid configurations that the NFA can enter in response to input strings. Each configuration consists of one or more NFA states, all of which are reachable from the same collection of input strings. The construction builds that model as a set Q that contains sets of NFA states, each representing a valid configuration, and a set T of transitions that recall how each q_i was built.

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

To construct a DFA from the model, we create a DFA state d_i for each $q_i \in Q$. The DFA transitions follow directly from the transitions recorded in T. If the construction built q_j by considering how the NFA, in configuration q_i, would move on character c, then T contains the transition $q_i \xrightarrow{c} q_j$ and $\delta_N(d_i, c)$ should be d_j.

Figure 2.6 shows the algorithm. To build Q and T, the algorithm considers, for each $q_i \in Q$ and each $c \in \Sigma$, the NFA's behavior when it is in configuration q_i and its next character is c. It computes the NFA's next configuration, *temp*, and records the transition in T. Next, it checks if temp is already in Q. If not, it adds *temp* to both Q and *WorkList*.

Each set $q_i \in Q$ contains a set of core states and zero or more non-core states. The non-core states consist of all states reachable from a core state by following one or more ϵ-transitions. The model's initial configuration, q_0, has only one core state—the NFA's initial state n_0. To form q_0, the algorithm adds all of the NFA states that n_0 implies—that is, those reachable along paths of ϵ-transitions. In the algorithm, the function *FollowEpsilon*(s) expands a set s with its non-core elements.

The algorithm finds new valid configurations by iteratively examining the configurations $q_i \in Q$ and the characters $c \in \Sigma$. To compute the NFA configuration reachable from q_i on c, it applies δ_N to each $n_x \in q_i$. If $\delta_N(n_x, c) = n_y$ and n_y is not the error state, then n_y is a core state in the new configuration. In the algorithm, the function *Delta*(*s*,*c*) computes the core of a new configuration from *s* and *c*. The algorithm then uses *FollowEpsilon* to add the implied non-core elements.

The algorithm starts with a single configuration, q_0, constructed from n_0. It uses a worklist to track which $q \in Q$ must still be processed. It iterates over the worklist and the alphabet, simulating possible transitions in the NFA and checking whether or not the transition creates a new configuration. It halts when it exhausts the worklist.

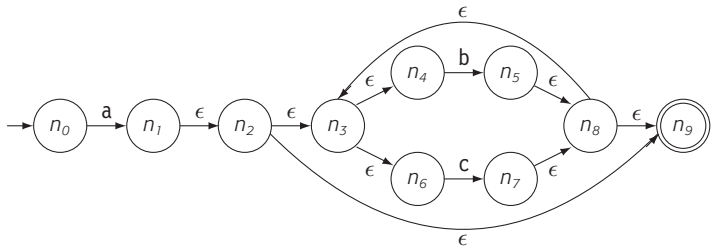
When the construction halts, Q contains all of the valid configurations of the NFA and T records all of the legal transitions between those configurations. Together, they represent a DFA that simulates the original NFA. To instantiate the DFA, we create a state d_i for each set $q_i \in Q$. If q_i contains an accepting state of the NFA—that is, some n_j such that $n_j \in N_A$—then the d_i that represents q_i is an accepting state—that is, $d_i \in D_A$. The DFA's transition function is built directly from T by mapping the sets in Q to their DFA states. Finally, q_0, the set constructed from n_0, becomes d_0, the DFA's initial state.

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—each q_i is a subset of 2^N (the powerset of N)—and each q_i appears on the worklist exactly once, the while loop must halt.

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, so that it 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 asymptotic running time of the DFA. If the data structures used to represent the DFA become sufficiently large, memory locality may become an issue that affects runtime performance. Fortunately, DFA minimization and table compression can mitigate these effects (See Sections 2.4.4 and 2.5.4).

Example

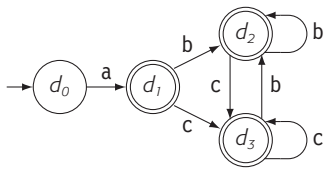
Figure 2.7a shows the NFA that Thompson's construction built for $a\,(b \mid c)^*$ with its states renumbered to read left-to-right. The table in Figure 2.7b sketches the steps of the subset construction on that NFA. The first column shows the name of the set in Q being processed in a given iteration 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



(a) Original NFA

			FollowEpsilon(Delta(q,x))		
Set Name	**DFA State**	**NFA States**	*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



(c) Resulting DFA

FIGURE 2.7 Applying the Subset Construction to the NFA from Figure 2.5

three columns show the result of applying *FollowEpsilon(Delta(q_i,x)* to the current set q_i and each character $x \in \Sigma$.

The algorithm takes the following steps:

1. The initialization sets q_0 to *FollowEpsilon*($\{n_0\}$), which is $\{n_0\}$.
2. The first iteration applies *Delta* and *FollowEpsilon* to q_0 with *a*, *b*, and *c*. Using *a* yields q_1, which contains six NFA states. Using *b* and *c* both produce the empty set.
3. The second iteration of the while loop examines q_1. Using *a* produces the empty set, while *b* yields q_2 and *c* yields q_3.
4. The third iteration of the while loop examines q_2. Using *a* produces the empty set, while *b* and *c* reconstruct q_2 and q_3.
5. The fourth iteration of the while loop examines q_3. Like the third iteration, it reconstructs q_2 and q_3. The algorithm halts because *WorkList* is empty.

Figure 2.7c shows the resulting DFA. The DFA states correspond to the sets in *Q*; the table and the transitions are taken from *T*. Each of q_1, q_2 and q_3 is an accepting state in the DFA because each contains n_9, the NFA's accepting state.

Fixed-Point Computations

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

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

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 never removes an element from *Q*.

We can view the while loop as a monotone increasing function *f*. It adds elements to *Q*, so the successive iterations produce iterates Q_i where each Q_{i+1} is larger than Q_i. (The comparison operator $\leq$ is $\subseteq$.) Since *Q* can grow to have 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.

A concern with fixed-point algorithms is the uniqueness of their results. For example, does the order in which the algorithm selects *q*

from the worklist affect the final set Q? In this algorithm, the monotone function applies the union operator to sets. Because set union is both commutative ($a \cup b = b \cup a$) and associative ($(a \cup b) \cup c = a \cup (b \cup c)$), the order in which the loop adds sets to Q does not change the final result. The subscripts assigned to specific $q_i \in Q$ may change with different orders of removal from the worklist, but the final Q will always contain the same sets of valid NFA configurations. The different possibilities for Q differ, at most, by the names of the q_i sets.

2.4.4 DFA to Minimal DFA

As the final step in the RE→DFA construction, we can employ an algorithm to minimize the number of states in the automaton. The subset construction can produce a DFA that has a large set of states. While the size of the DFA does not affect its asymptotic complexity, it does determine the recognizer's footprint 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 lowest level of cache memory, producing faster average accesses.

To reduce the size of the DFA and, thus, its transition table, the scanner generator can apply a DFA minimization algorithm. The best known and asymptotically fastest algorithm, Hopcroft's algorithm, constructs a minimal DFA from an arbitrary DFA by grouping together states into sets that are *equivalent*. Two DFA states are equivalent when they produce the same behavior on any input string. The algorithm finds the largest possible sets of equivalent states; each set becomes a state in the minimal DFA.

The algorithm constructs a *set partition*, $P = \{ p_1, p_2, p_3, \ldots, p_m \}$ of the DFA states. Each p_i contains a set of equivalent DFA states. More formally, it constructs a partition with the smallest number of sets, subject to the following two rules:

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

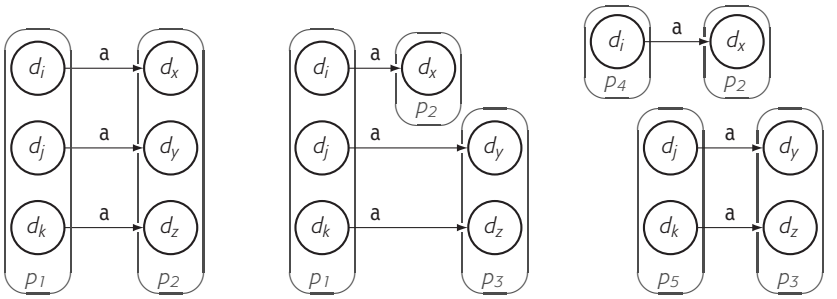
(1) $\forall\, c \in \Sigma$, if $d_i, d_j \in p_s$; $d_i \xrightarrow{c} d_x$; $d_j \xrightarrow{c} d_y$; and $d_x \in p_t$; then $d_y \in p_t$.

(2) If $d_i, d_j \in p_i$ and $d_i \in D_A$, then $d_j \in D_A$.

Rule 1 mandates that two states in the same set must, for every character $c \in \Sigma$, transition to states that are, themselves, members of a single set in the partition. Rule 2 rule states that any single set contains either accepting states or nonaccepting states, but not both.

These two properties not only constrain the final partition, P, but they also lead to a construction for P. The algorithm starts with the coarsest partition on behavior, $P_0 = \{ D_A, \{ D - D_A \} \}$. It then iteratively "refines" the partition until both properties hold true for each set in P. To refine the partition, the algorithm splits sets based on the transitions out of DFA states in the set.

P_0 divides D into accepting and non-accepting states, a fundamental difference in behavior specified by rule 2.



(a) a does not split p_1 (b) a splits p_1 (c) Sets from (b) after the split

FIGURE 2.8 Splitting a Set around a

Figure 2.8 shows how the algorithm uses transitions to split sets in the partition. In panel (a), all three DFA states in set p_1 have transitions to DFA states in p_2 on the input character a. Specifically, $d_i \xrightarrow{a} d_x$, $d_j \xrightarrow{a} d_y$, and $d_k \xrightarrow{a} d_z$. Since $d_i, d_j, d_k \in p_1$, and $d_x, d_y, d_z \in p_2$, sets p_1 and p_2 conform to rule 1. Thus, the states in p_1 are behaviorally equivalent on a, so a does not induce the algorithm to split p_1.

By contrast, panel (b) shows a situation where the character a induces a split in set p_1. As before, $d_i \xrightarrow{a} d_x$, $d_j \xrightarrow{a} d_y$, and $d_k \xrightarrow{a} d_z$, but $d_x \in p_2$ while $d_y, d_z \in p_3$. This situation violates rule 1, so a induces the algorithm to split p_1 into two sets, $p_4 = \{ d_i \}$ and $p_5 = \{ d_j, d_k \}$, shown in panel (c).

The algorithm, shown in Figure 2.9, builds on these ideas. Given an arbitrary DFA, it constructs a partition that represents the minimal DFA. To simplify the exposition, it uses two copies of the partition. At each stage, *Partition* holds the current approximation to the minimal DFA, while the algorithm builds the next approximation in *NextPartition*.

Starting with the largest possible sets and splitting them is an *optimistic* approach to building the sets. Optimism is discussed in Section 9.3.6 or [347].

To start, the algorithm constructs the coarsest partition consistent with rule 2, $\{ D_A, \{ D - D_A \} \}$. This choice has two consequences. First, since each set in the final partition is constructed by splitting a set in an earlier approximation, it ensures that no set in the final partition will contain both accepting and nonaccepting states. Second, by choosing the largest sets consistent with rule 2, it imposes the minimum constraints on the splitting process which, in turn, can lead to larger sets in the final partition. (Larger sets means fewer states in the final DFA.)

The algorithm operates from a worklist of states, starting with the initial partition $\{ D_A, \{ D - D_A \} \}$. It repeatedly picks a set s from the worklist and uses that set to refine the partition in *NextPartition* by splitting sets based on their transitions into s.

To identify states that must split because of a transition into set s on some character c, the algorithm inverts the transition function. It com-

```
Partition   ← {D_A, {D − D_A}}
Worklist    ← {D_A, {D − D_A}}

while( Worklist ≠ ∅ )
    select a set s from Worklist and remove it
    for each character c ∈ Σ
        Image ← {x | δ(x,c) ∈ s}
        for each set q ∈ Partition that has a state in Image
            q_1 ← q ∩ Image
            q_2 ← q − q_1
            if q_1 ≠ ∅ and q_2 ≠ ∅ then    // split q around s and c
                remove q from Partition
                Partition ← Partition ∪ q_1 ∪ q_2
                if q ∈ Worklist then       // and update the Worklist
                    remove q from Worklist
                    WorkList ← WorkList ∪ q_1 ∪ q_2
                else if |q_1| ≤ |q_2|
                    then WorkList ← Worklist ∪ q_1
                    else WorkList ← WorkList ∪ q_2
                if s = q then              // need another s
                    break
```

FIGURE 2.9 DFA Minimization Algorithm

putes the set of DFA states that can reach a state in set s on a transition labelled c and assigns that set to *Image*. It then systematically examines each set q that has a state in *Image* to see if *Image* induces a split in q. If *Image* divides q into non-empty sets q_1 and q_2, it removes q from both *Partition* and *NextPartition* and then adds both q_1 and q_2 to *NextPartition*.

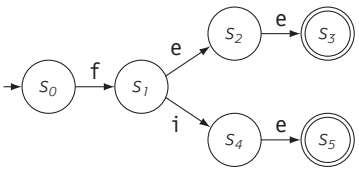
All that remains, in processing q with respect to c, is to update the worklist. If q is on the worklist, then the algorithm replaces q with both q_1 and q_2. The rationale is simple: q was on the worklist for some potential effect; that effect might be from some character other than c, so all of the DFA states in q need to be represented on the worklist.

If, on the other hand, q is not on the worklist, then the only effect that splitting q can have on other sets is to split them. Assume that some set r has transitions on letter e into q. Dividing q might create the need to split r into sets that transition to q_1 and q_2. In this case, either of q_1 or q_2 will induce the split, so the algorithm can choose between them. Using the smaller set will lead to faster execution; for example, computing *Image* takes time proportional to the size of the set.

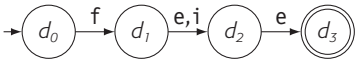
To construct the new DFA from the final *Partition*, we can create a state to represent each set $p_i \in$ *Partition* and add the appropriate transi-

Step	Partition on Entry	Worklist	s	c	Image	q	q_1	q_2	Action
0	p_0: $\{s_3, s_5\}$, p_1: $\{s_0, s_1, s_2, s_4\}$	p_0, p_1	—	—	—	—	—	—	—
1	p_0: $\{s_3, s_5\}$, p_1: $\{s_0, s_1, s_2, s_4\}$	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	*split* $p_1 \rightarrow p_2, p_3$
		p_2, p_3	p_0	f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
		p_2, p_3	p_0	i	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
2	p_0: $\{s_3, s_5\}$, p_2: $\{s_2, s_4\}$,	p_3	p_2	e	s_1	p_3	s_1	s_0	*split* $p_3 \rightarrow p_4, p_5$
	p_3: $\{s_0, s_1\}$	p_4, p_5	p_2	f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
		p_4, p_5	p_2	i	s_1	$\emptyset$	$\emptyset$	$\emptyset$	*none*
3	p_0: $\{s_3, s_5\}$, p_2: $\{s_2, s_4\}$,	p_5	p_4	e	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
	p_4: $\{s_1\}$, p_5: $\{s_0\}$	p_5	p_4	f	s_0	p_5	s_0	$\emptyset$	*none*
		p_5	p_4	i	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
4	p_0: $\{s_3, s_5\}$, p_2: $\{s_2, s_4\}$,	$\emptyset$	p_5	e	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
	p_4: $\{s_1\}$, p_5: $\{s_0\}$	$\emptyset$	p_5	f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*
		$\emptyset$	p_5	i	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*none*

(a) Iterations of Hopcroft's Algorithm on the Original DFA for "fee | fie"



(b) Original DFA for "fee | fie"



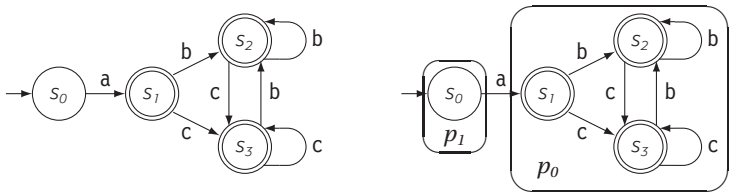
(c) Minimal DFA for "fee | fie"

FIGURE 2.10 Applying the DFA Minimization Algorithm

tions between these new representative states. For the state representing p_m, we add a transition to the state representing p_n on character c if some $d_j \in p_m$ has a transition on c to some $d_k \in p_n$. The construction ensures that, if $d_j \xrightarrow{c} d_k$, where $d_j \in p_m$ and $d_k \in p_n$, then every state in p_m has a similar transition on c to a state in p_n. If this condition did not hold, the algorithm would have split p_m around the transitions on c. The resulting DFA is minimal; the proof is beyond our scope.

Examples

As a first example, consider the DFA for *fee* | *fie* shown in Figure 2.10.b. Panel (a) shows the progress of Hopcroft's algorithm on this DFA.



(a) Original DFA (b) Initial Partition

FIGURE 2.11 DFA for $a\,(b \mid c)^*$

The first line, step 0, shows the initial configuration of the algorithm. Both *Partition* and *Worklist* contain two sets: $\{ D_A, \{D - D_A\} \}$. D_A is labelled p_0 while $\{D - D_A\}$ is labelled p_1.

The algorithm enters the while loop and removes p_0 from *Worklist*; it becomes *s*. The algorithm iterates over the characters in Σ, in the order `e`, `f`, and `i`. For `e`, p_0 splits p_1 into two sets: p_2: $\{s_0, s_1\}$ and p_3: $\{s_2, s_4\}$. The algorithm removes p_1 from *Partition* and adds p_2 and p_3. Next, it removes p_1 from *Worklist* and adds p_2 and p_3. For `f` and `i`, no edges enter p_0. Thus, *Image* is empty and no splits occur.

The second iteration proceeds in a similar fashion. It chooses $p2$. The character `e` splits p_3 and causes an update to both *Partition* and *Worklist*. For `f`, the *Image* set is empty. For `i`, the *Image* set contains s_i. Because the algorithm already split p_3 around *e*, this situation does not cause a split. It will, however, add a transition to the final DFA.

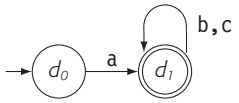
The third iteration chooses p_4 from the worklist. Both `e` and `i` produce empty *Image* sets. With `f`, the *Image* set contains s_0. Because p_5, which contains s_0, is a singleton set, it cannot be split. This situation, however, will add a transition to the final DFA.

The fourth and final iteration takes p_5 from the worklist. For each of `e`, `f`, and `i`, the *Image* set is empty. Thus, the iteration splits no sets. It adds no transitions. It finishes with an empty worklist. The resulting minimal DFA is shown Figure 2.10.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 2.11a. The initial partition is $\{ p_0{:}\{s_1, s_2, s_3\}, p_1{:}\{s_0\} \}$.

The algorithm first selects p_0 and examines each of `a`, `b`, and `c`. For `a`, *Image* contains s_0 which is in a singleton set, p_1. Thus, `a` introduces a transition for the final DFA, but no split. For both `b` and `c`, *Image* is $\{s_1, s_2, s_3\}$, which is exactly p_0. Thus, q_2 is empty and no splits occur.

Next the algorithm removes p_1 and examines each of `a`, `b`, and `c`. Since no transitions enter p_1, the *Image* set is empty for each letter. No further splits occur. The original two set partition is the final partition. The final DFA has two states, as shown in the margin. Recall that this is the DFA that we suggested a human would derive. After minimization, the automatic techniques produce the same result.



2.4.5 Using a DFA as a Scanner

The tools in the three previous sections provide an algorithmic path from an RE to a minimal DFA. As we saw in Figure 2.2, a DFA can be simulated with a simple table-driven skeleton. Taken together, these suggest that we can automate scanner construction by taking REs for all of the words in a programming language, combining them into a single RE, and using the resulting DFA as a scanner. Reality, however, is more complex. Scanners and DFAs differ in two critical ways that affect how we formulate and build RE-based scanners.

Model of Execution

A DFA reads all of its input and accepts the input if its last state is a final state. That is, a DFA tries to find one word. By contrast, a scanner reads enough input to find the next word in the input stream. The scanner leaves the input stream in a state from which it can find the next word.

This difference necessitates a new model of execution. Rather than exhausting the input stream, the scanner simulates the DFA until it hits an error—that is, until it is in some state d_i with input character c such that $\delta(d_i, c) = s_e$, the error state. We also define $\delta(d_j, \texttt{eof}) = s_e, \forall\, d_j \in D$.

If d_i is an accepting state, $d_i \in D_A$, the scanner has found a word. If d_i is not an accepting state, the scanner may have passed through such a state on its way to d_i. To determine if it did, the scanner must back up, one character at a time until it either reaches an accepting state or it exhausts the lexeme.

This scheme adds some work to the implementation. The scanner must either record states or invert δ. Either approach takes time proportional to the number of scanned characters. A character may be scanned multiple times; Section 2.5.1 shows a method for avoiding the worst case of this behavior.

Finding Syntactic Categories

A DFA returns a binary answer: it either accepts or rejects the input. By contrast, a scanner returns a token, ⟨*lexeme*,*category*⟩, that gives the spelling and syntactic category of the next word. It indicates an error with an invalid token.

$d_i \in D_A$ maps uniquely to a category, but one category may map to multiple d_is.

If we construct the DFA so that each final state maps to a single category, then the scanner can find the category with a simple table lookup. However, this scheme requires that we build the DFA in a way that preserves the mapping of final states to categories.

Most scanner generators take, as input, a list of REs, $r_1, r_2, \ldots, r_k$, each of which defines the spelling of some category. The obvious way to build a single DFA is to construct a single RE, $(r_1 \,|\, r_2 \,|\, \ldots \,|\, r_k)$, and construct a DFA from this RE. However, Thompson's construction

Identifying Keywords

Most programming languages reserve the keywords that identify critical parts of the syntax, words such as `if`, `then`, and `while`. In a typical scanner and parser, each keyword has its own syntactic category—a category with only one word. The compiler writer faces a choice: specify each keyword with its own rule, or fold keywords into the rule for identifiers and recognize them with some other mechanism. Either approach works and can lead to an efficient scanner.

With a separate rule for each keyword, the scanner can return the appropriate category using the same mechanism used for other categories, such as *number* or *identifier*. The extra rules may add minor cost to scanner generation, for the extra rules and extra states, and may produce a DFA with more states. However, since the process produces a DFA, the resulting scanner will still require $O(1)$ time per character.

As an alternative, most scanners build a table of all identifier names. This table serves as a start on the compiler's *symbol table* (see Section 4.4) and as a way to map identifier names into small integers so that they can be represented and compared efficiently. If the compiler writer pre-loads the symbol table with the keywords and their syntactic categories, the scanner will find the keywords as previously seen and categorized identifiers, and will return the appropriate category for each.

will immediately unify the final states, destroying the mapping from $d_i \in D_A$ to categories.

Instead, the scanner generator can build an NFA for each rule, using Thompson's construction. It can join those NFAs into a single DFA, with a new start state and ϵ-transitions, and use the subset construction to build a DFA that simulates the NFA. This process preserves the final states for each rule.

If two rules share a common prefix, the subset construction will merge their final states. When the subset construction merges final states, the scanner generator must decide which syntactic category it will return for that final state. In practice, scanner generators let the compiler writer specify a precedence among syntactic categories. The scanner generator assigns to the final state the category with the highest precedence.

This situation reveals an ambiguity in the specification. For example, a keyword such as `then` may also fit the rule for an identifier.

Both `flex` and `lex` assign higher precedence to the rule that appears first in the list of rules.

Minimization poses another challenge. Hopcroft's algorithm immediately combines all of the final states into a single partition, destroying the property that final states map to syntactic categories. If, however,

Hopcroft's algorithm splits partitions but never combines them.

the scanner generator constructs an initial partition that places the final states for each syntactic category in a distinct set in the final partition, then the rest of the algorithm will maintain that property.

The resulting DFA may be larger than the minimal DFA that results from grouping all final states into the same partition. However, the larger DFA has the property that the compiler needs: each final state maps to a specific syntactic category.

The Role of Whitespace

Programmers often refer to blanks and tabs, when used to format code, as *whitespace*. In most languages, whitespace has no intrinsic meaning. Scanners for these languages typically recognize and discard whitespace. The primary impact of whitespace arises from its absence in the regular expressions that define words in the language.

For example, the fact that the RE for an identifier or keyword name does not include blank or tab forces an RE-based scanner to separate `do` and `i` in a sentence such as:

```
do i = 1 to 100
```

For similar reasons, the RE for identifier does not contain `+`, `-`, `*`, or `/`. This fact ensures that "`a * b`" scans the same as "`a*b`".

Fortran In FORTRAN 66 blanks are not significant. That is, "`n a m e`" and "`name`" refer to the same identifier. This rule complicates scanning. The header of a FORTRAN do loop might read:

```
do 10 i = 1,100
```

where `10` is the label of the last statement in the loop body, `i` is the loop's index variable, and `i`'s value runs from 1 to 100. (The increment defaults to one unless specified.)

Of course, `do10i` is a valid variable name. To differentiate between these two statements:

```
do 10 i = 1
do 10 i = 1,100
```

a scanner must read beyond the `=` and `1` to the comma. The comma proves that the second statement is a loop header, and the scanner can separate `do10i` into three words, `do`, `10`, and `i`. Few, if any languages, have followed FORTRAN's example.

Python Python takes the opposite approach: not only are blanks significant, but the number of blanks at the start of a line determines the meaning of a Python program. Rather than using bracket constructs, such as { and } or `begin` and `end`, to indicate block structure, Python relies on changes in indentation.

A simple way of handling leading blanks in Python is to add a rule that recognizes an end-of-line followed by zero or more blanks. The scanner can then test the length of the lexeme. If its length is identical to the previous token in this category, it returns the result of calling the scanner again. If its length differs, the scanner can return a category indicating the start of a block or the end of a block, as appropriate.

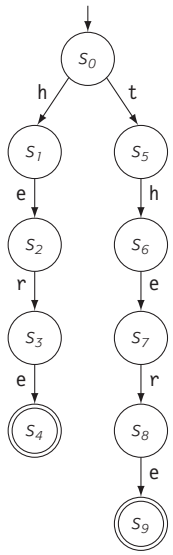
SECTION REVIEW

Given a regular expression, we can derive a minimal DFA to recognize the language specified by the RE in a two-step process: (1) apply Thompson's construction to build an NFA for the RE; (2) use Brzozowski's algorithm to construct the minimal DFA from that NFA. Brzozowski's algorithm employs the subset construction. This process produces an efficient recognizer for the language specified by a single RE.

To build a scanner that recognizes multiple categories of words, each specified by an RE, we can use the two-step process to build a minimal RE for each category, and then combine those DFAs into a single NFA by adding a new start state that has ϵ-transitions to the start states of the individual minimal DFAs. The subset construction will convert that NFA to a DFA that has distinct final states for each category.

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 DFA shown in the margin.



DFA for Review Question 2

2.5 IMPLEMENTING SCANNERS

Scanner generators apply the theory of formal languages directly to the problem of creating efficient tokenizers for programming languages. Using the techniques outlined in the previous section, these tools build DFA models of a programming language's microsyntax and covert those models into executable code.

This section discusses three implementation strategies for converting a DFA into executable code. The first two, table-driven scanners and direct-coded scanners, are products of *scanner generators*. The third strategy is to craft a custom scanner by hand. Each strategy can lead to a robust and efficient scanner.

Generated scanners operate by simulating a DFA, as described in Section 2.4.5. They begin in some initial state and take a series of transitions based on the current state and the current input character. When the current state has no valid transition for the input character, the scanner backs up until it finds an accepting state. If it cannot find an accepting state, it reports a lexical error.

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. All these strategies can produce scanners that use $O(1)$ time per character; the differences, however, can affect the constants in that equation. 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. Typically, table-driven scanners come from scanner generators. At design time, the compiler writer creates a set of REs. At build time, the scanner generator creates a set of tables that, when compiled with the skeleton scanner, implements the underlying DFA. Conceptually, it looks like:

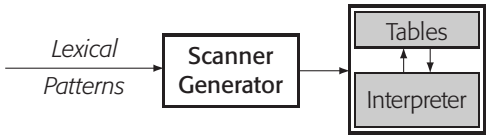


Figure 2.12 shows a table-driven scanner for the RE $r[0\ldots9]^+$, introduced in Section 2.3.2. Panel (a) shows the code for the skeleton scanner. Panels (b), (c), and (d) show the tables that encode the DFA, which is shown in panel (e). The code is more detailed than that shown in Figure 2.2 on page 32, but the basic operation is similar.

The skeleton scanner divides into four sections: initializations, a scanning loop to model the DFA's behavior, a roll back loop to find a final state, and a final section to interpret and report the results. The scanning loop repeats the two basic actions of a scanner: read a character and take a transition. The loop halts when the DFA enters the error state, s_e. The transition table δ represents the DFA's transition diagram. Identical columns in δ have been combined, so the scanner uses the table *CharClass* to map an input character into a column index in δ. The roll back loop uses a stack of states to revert the scanner to its most recent accepting state.

Compressed tables are discussed in Section 2.5.4. In the example, the entries for 0…9 have been combined into one column.

The function *NextChar* returns the next character in the input stream. A corresponding function, *RollBack*, moves the input stream back by one character (See Section 2.5.4).

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

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

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

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

(a) Code to Interpret the Tables

r	0...9	Other
Register	*Digit*	*Other*

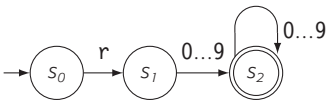
(b) The Classification Table, *CharClass*

	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

(c) The Transition Table, δ

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

(d) The Token Type Table, *Type*



(e) The Underlying DFA

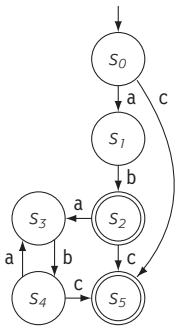
FIGURE 2.12 A Table-Driven Scanner for Register Names

When the scanning loop halts, *state* may contain a non-accepting state. In that case, the scanner backs up until it either finds an accepting state or it proves that none exists. In most languages, the amount of such roll-back will be limited. We can construct REs that will require quadratic roll-back on specific inputs. If the scanner must handle such an RE, an alternative implementation, such as the one described in the next subsection, should be used. In most programming languages, however, the amount of roll-back should be small.

Avoiding Excess Roll Back

Some regular expressions can cause the scanner in Figure 2.12.a to need quadratic roll-back. The problem arises from the 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 `abababababc`, a scanner built from the DFA will read all the characters and return the entire string as a



```
state ← s0;
lexeme ← "";
clear stack;
push(⟨bad, 1⟩);

while (state ≠ se) do
    if Failed[state,InputPos] then
        ⟨state, InputPos⟩ ← pop();
        truncate lexeme;
        break;
    char ← Input[InputPos];
    lexeme ← lexeme + char;
    if state ∈ SA then
        clear stack;
        push(⟨bad, 1⟩);
    push(⟨state, InputPos⟩);
    col ← CharClass[char];
    state ← δ[state, col];
    InputPos ← InputPos + 1;

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

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

This figure is ugly!

FIGURE 2.13 The Maximal Munch Scanner

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, roll-back can create $O(n^2)$ behavior.

The *maximal munch scanner* avoids this kind of 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 code tests each ⟨*state,input position*⟩ pair and breaks out of the scanning loop whenever a failed transition is attempted.

Figure 2.13 shows the maximal munch scanner. The scanner keeps a global counter, `InputPos`, to record position in the input stream. It uses a bit-array, `Failed`, to record dead-end transitions. `Failed` has a row for each state and a column for each character in the input stream. When the scanner must roll back the input, it marks the appropriate transitions in `Failed` to prevent it from taking the same dead-end path on subsequent invocations.

The code in Figure 2.13 runs on each call to the scanner. Before the first call to the scanner, both `Failed` and `InputPos` must be initialized. Every bit in `Failed` is set to `false`. `InputPos` is set to one.

Minor optimizations can reduce the size of `Failed`. For example, if the scanner uses a finite input buffer (see Section 2.5.4), the number of

columns in `Failed` can be reduced to the size of the input buffer.

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.

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
    char  ← NextChar();
    col   ← CharClass[char];
    state ← δ[state,col];
```

Here, `state` explicitly represents the DFA's current state and δ is a two dimensional array that represents the DFA's transition diagram. Identical columns in δ have been combined (See Section 2.5.4). `CharClass` is a vector that maps an input character to a column index in δ.

Reducing the Overhead of Table Lookup

For each character, the table-driven scanner accesses two arrays: δ and `CharClass`. While both lookups take $\mathbf{O}(1)$ time, these table lookups have constant-cost overheads that a direct-coded scanner may be able to avoid. To access the i^{th} element of `CharClass`, the code must compute its address, given by

```
@CharClass_0 + i × w
```

where `@CharClass`$_0$ is a constant related to the starting address of *CharClass* in memory and *w* is the size of an element of *CharClass*. The code then loads the column index found at that memory address.

Detailed discussion of the code to compute an address for an array element starts on page 361 in Section 7.5.

Next, the scanner locates the state in δ. Because δ has two dimen-

```
s_in : lexeme ← " ";
       clear stack;
       push( bad );
       goto s_0 ;

s_0 :  char ← NextChar();
       lexeme ← lexeme + char;
       push( s_0 );
       if (char = 'r')
          then goto s_1 ;
          else goto s_out

s_1 :  char ← NextChar();
       lexeme ← lexeme + char;
       push( s_1 );
       if ('0' ≤ char ≤ '9')
          then goto s_2
          else goto s_out

s_2 :  char ← NextChar();
       lexeme ← lexeme + char;
       clear stack;
       push( s_2 );
       if '0' ≤ char ≤ '9'
          then goto s_2
          else goto s_out

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

FIGURE 2.14 A Direct-Coded Scanner for $r\,[0 \ldots 9]^+$

sions, the address calculation for δ[state,col] is more complex:

$$@\delta_0 + (\text{state} \times \text{number of columns in } \delta + \text{col}) \times w$$

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 computes two addresses and performs two loads for each input character. Some of the speed improvement in a direct-coded scanner comes from reducing this overhead.

Replacing the Table-Driven Scanner's While Loop

The table-driven scanner represents the DFA state and transition diagram explicitly, so that it can use the same code to implement each state. By contrast, a direct-coded scanner represents the state and transition diagram implicitly. It uses a distinct and customized code fragment to implement each state. It emulates state-to-state transitions by branching to the appropriate code fragments.

Figure 2.14 shows a direct-coded scanner for $r\,[0 \ldots 9]^+$; it is equivalent to the table-driven scanner shown earlier in Figure 2.12. Execution begins at label s_{in}, which jumps to s_0, the code for DFA state s_0.

Consider the code for state s_1. It reads a character, concatenates it onto the current lexeme, and pushes s_1 onto its internal stack. If *char* is a digit, it jumps to state s_2. Otherwise, it jumps to state s_{out}. The code has

no complicated address calculations. It refers to a tiny set of values—*char*, *lexeme*, and *state*—that can be kept in registers. The other states have equally simple implementations.

A scanner generator can directly emit code similar to that shown in Figure 2.14. Each state has a couple of standard actions, followed by branching logic that implements the transitions out of the state. If some state has too many distinct outbound transitions, a clever implementation might build a small transition table and use a "computed" branch scheme—a table lookup into a table of labels. 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.

The code in Fig. 2.14 assumes that `truncate lexeme` and `Rollback` both handle gracefully any attempt to back up past the start of the lexeme.

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.

Code in the style of Figure 2.14 is often called *spaghetti code* in honor of its tangled control flow.

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.

We suspect that hand–coded scanners persist for one simple reason: they are small, simple programs that can be fun to implement.

The direct-coded scanner reduced the overhead of simulating the DFA; the hand-coded scanner offers a clever compiler writer additional opportunities to improve performance. The code along specific paths can be optimized.

For example, the scanner from Figure 2.14 implements a DFA that has just one accepting state. Thus, the entire stack mechanism for tracking accepting states can be eliminated. The transitions $s_0 \rightarrow s_{out}$ and $s_1 \rightarrow s_{out}$ can be replaced with code that simply returns *invalid*.

Similarly, the interface between the scanner and the parser can be improved. The table-driven and direct-coded scanners for $r[0...9]^+$ return the lexeme as a character string. If the parser has a syntactic category *register name*, the scanner might return the actual register number rather than the string that contains it—avoiding that conversion in the parser and eliminating multiple concatenations in the scanner.

2.5.4 Practical Implementation Issues

This section addresses two practical issues that arise in building a scanner: handling the input stream in a fashion that allows both efficient character-by-character scanning and rollback; and compressing the transition table so that it requires less space.

Buffering the Input Stream

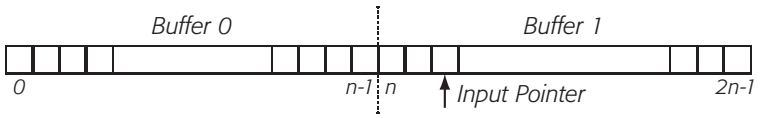
While character-by-character I/O leads to clean algorithms, the overhead of a function call per character is significant relative to the cost of simulating the DFA. To reduce the I/O cost per character, the compiler writer can use buffered I/O, where each read operation returns a longer string of characters, or buffer, and the scanner then indexes through the buffer. The scanner maintains a pointer into the buffer. Responsibility for filling the buffer, tracking the current location, and recognizing the end of file all fall 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 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. To back up in the input stream, 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 2.15. Each execution of `NextChar` loads a

```
NextChar() {
    Char ← Buffer[Input];
    if Char ≠ eof then
        Input ← (Input + 1) mod 2n;
        if (Input mod n = 0) then
            fill Buffer[Input : Input + n - 1];
            Fence ← (Input + n) mod 2n;
    return Char;
}
```

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

Initialize() {
    Input ← 0;
    Fence ← 0;
    fill Buffer[0 : n-1];
}
```

FIGURE 2.15 Implementing *NextChar* and *RollBack*

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. `Initialize` simply provides a known and consistent starting state.

As a 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 where n is the buffer size, `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.

If n is chosen to make the I/O efficient, say 2,048 or 4,096 bytes for each half of the buffer, that should provide enough rollback for real programs. The amount of rollback required by any particular input is bounded by the longest sequence of whitespace-free characters. Few programs contain identifiers with more than 2,048 characters.

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

(a) The Full Transition Table for *0 | [1...9] [0...9]**

δ	0	1...9	Other
s_0	s_1	s_2	s_e
s_1	s_e	s_e	s_e
s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e

(b) The Compressed Table

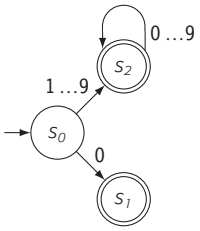
FIGURE 2.16 Transition-Table Compression Example

Compressing the Transition Table

The transition table for a DFA contains $|\text{states}| \cdot |\Sigma|$ entries. For a real programming language, both $|\text{states}|$ and $|\Sigma|$ can be large. When the table size grows larger than the size of the first-level cache, it may cause performance problems.

If the DFA is minimal, its rows cannot be identical.

The transition table for a programming language scanner often contains columns that are identical. Consider, for example, the DFA for *0 | [1...9] [0...9]**, introduced in in Section 2.2.2 and repeated in the margin. Figure 2.16.a shows the naive representation of the table. Panel (b) of that figure shows the same table, with the columns for the characters 1 through 9 compressed into a single column. The code skeletons in Sections 2.12 and 2.13 create one more opportunity: the row for s_e cannot be referenced and, thus, need not be represented.



To use a compressed table, the scanner must map actual characters into columns in the transition table. A simple and efficient way to implement this translation is with a classification table that maps an input character to a column index. The fundamental loop in a scanner can be abstracted to the following code, shown on the left.

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

```
while (state ≠ s_e) do
    char  ← NextChar();
    col   ← CharClass[char];
    state ← δ[state,col];
```

The corresponding code, with classification, is shown on the right. It adds one memory reference from `CharClass`; in return, the scanner can use a much smaller table representation.

Of course, the scanner generator must generate both the classifier and the compressed table. Figure 2.17 shows the obvious algorithm to find identical columns. It assumes a transition table, δ with *NumCols* columns and *NumRows* rows. When it finishes, if *MapTo[i]* $= j$, for $i \neq j$, then rows i and j are identical and can be compressed to a single row. We leave the construction of the classifier table and the compressed version of δ as an exercise for the reader (see problem 13.).

```
for i ← 1 to NumCols
    MapTo[i] ← i

for i ← 1 to NumCols
    if MapTo[i] = i then
        for j ← i to NumCols
            if MapTo[j] = j then
                same ← true
                for k ← 1 to NumRows
                    if δ(i,k) = δ(j,k) then
                        same ← false
                        break
                if same then
                    MapTo[j] ← i
```

FIGURE 2.17 Finding Identical Columns in δ

The algorithm, as shown, performs $O(|\Sigma|^2)$ comparisons in the worst case. Each comparison costs $O(|\text{states}|)$ time. We can reduce the quadratic term by keeping, for each column of δ, a population count of the non-error states in the column. Since identical columns must have identical counts, the algorithm can construct an index set and radix sort it based on the population count. Then, it need only compare two columns if the radix sort puts them in the same category. This strategy should lead to multiple groups, each of which requires $O(n^2)$ comparisons; however, each n should be smaller than $|\Sigma|$.

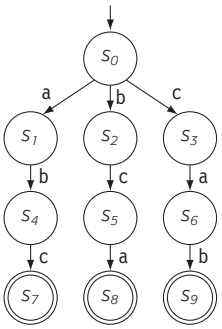
The count of non-error states is a simple signature for the column. More complex, and possibly more expensive, signatures can be implemented. In the best case, they reduce the cost of the comparisons to $O(1)$. For example, There is, however, a direct tradeoff between the cost of the comparison and the cost of computing the signature—an issue for the careful implementor to consider.

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 character, 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 in the margin, 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. Contrast the cost of this approach with using the DFA from question 1 above.

2.6 ADVANCED TOPICS

This section expands on the material in Section 2.4 for the interested reader. With Thompson's construction, we can build an NFA from an arbitrary regular expression. With the subset construction, we can build a DFA to simulate any NFA. Together, these show that DFAs are at least as powerful as REs. To complete the cycle of construction, shown in Figure 2.3 on page 42, we must show a construction that creates an RE that represents the set of words accepted by an arbitrary

$$
\begin{array}{l}
\textit{for } i = 0 \textit{ to } |D| - 1 \\
\quad \textit{for } j = 0 \textit{ to } |D| - 1 \\
\qquad R_{ij}^{-1} \leftarrow \{\, a \mid \delta(d_i, a) = d_j \,\} \\
\qquad \textit{if } (i = j) \textit{ then} \\
\qquad\quad R_{ij}^{-1} \leftarrow R_{ij}^{-1} \mid \{\, \epsilon \,\} \\
\textit{for } k = 0 \textit{ to } |D| - 1 \\
\quad \textit{for } i = 0 \textit{ to } |D| - 1 \\
\qquad \textit{for } j = 0 \textit{ to } |D| - 1 \\
\qquad\quad R_{ij}^{k} \leftarrow R_{ik}^{k-1} \left(R_{kk}^{k-1}\right)^{*} R_{kj}^{k-1} \mid R_{ij}^{k-1} \\
L \leftarrow \mid_{d_j \in D_A} R_{0j}^{|D|-1}
\end{array}
$$

FIGURE 2.18 Deriving a Regular Expression from a DFA

DFA. Section 2.6.1 sketches that construction, often called Kleene's algorithm.

Section 2.6.2 looks at an interesting subcase of the RE to DFA problem: closure-free regular expressions. The DFA for a closure-free RE is acyclic. Section 2.6.2 sketches an algorithm to build such a DFA directly and incrementally.

Finally, Section 2.4.4 presented Hopcroft's algorithm for DFA minimization. Section 2.6.3 describes an alternative algorithm by Brzozowski that reuses the subset construction. known and widely taught.

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_j, $\forall\, d_j \in D_A$. 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, $d_j \in D_A$. It constructs the path expressions for all paths by iterating over i, j, and k.

We use the notation R_{ij}^{k} to represent the regular expression that describes all paths from d_i to d_j that do not pass through a state numbered higher than d_k. Here, *through* means that the path both enters and leaves a state numbered higher than d_k. In a DFA with a transition $d_1 \rightarrow d_{16}$, the RE $R_{1,16}^{2}$ would be nonempty because the path enters d_{16} but does not pass through it.

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_{ij}^{0}.

Initially, the algorithm places all of the direct paths from d_i to d_j in R_{ij}^{-1}. It adds $\{ \epsilon \}$ to each expression where $i = j$. Over successive iterations, it builds up longer paths; it computes R_{ij}^{k} from R_{ij}^{k-1} by adding those paths that pass through d_k on their way from d_i to d_j. The algorithm computes this additional component as (1) the set of paths from d_i to d_k that pass through no state numbered higher than *k-1*, concatenated with (2) any paths from d_k to itself that pass through no state numbered higher than *k-1*, concatenated with (3) the set of paths from d_k to d_j that pass through no state numbered higher than *k-1*. The assignment in the inner loop

$$R_{ij}^{k} \leftarrow R_{ik}^{k-1} \left(R_{kk}^{k-1}\right)^{*} R_{kj}^{k-1} \mid R_{ij}^{k-1}$$

captures those paths and uses alternation to add them to the RE for the paths from R_{ij}^{k-1}. In this way, each iteration of the inner loop adds the paths that pass through d_k to R_{ij}^{k-1} to form R_{ij}^{k}.

When the k loop terminates, the various R_{ij}^{k} 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.

Of course, Kleene's algorithm computes expressions for many paths that are not needed. To derive an RE from a DFA, we only need the paths that start at d_0 and end at some accepting state d_j. Limiting the computation to just the paths of interest would not reduce the worst-case asymptotic complexity, but it might significantly reduce the total amount of work.

2.6.2 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

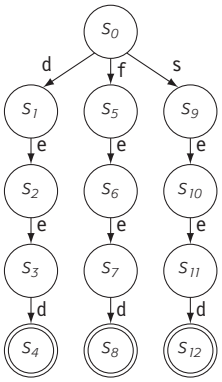
expressions. Such REs have the form $w_1 \mid w_2 \mid w_3 \mid \ldots \mid w_n$ where the individual words, w_i, are just concatenations of characters in the alphabet, Σ. These REs have the property that they produce DFAs with acyclic transition graphs.

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 in the margin. 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. The algorithm will build a DFA that is minimal with regard to prefixes of words in the language, similar to those produced by the subset construction (See the next subsection). A complete minimization would combine states (s_1, s_5, s_9), 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 four state DFA.



2.6.3 An Alternative DFA Minimization Algorithm

The subset construction converted an NFA to a DFA by systematically eliminating ϵ-transitions and combining paths in the NFA's transition

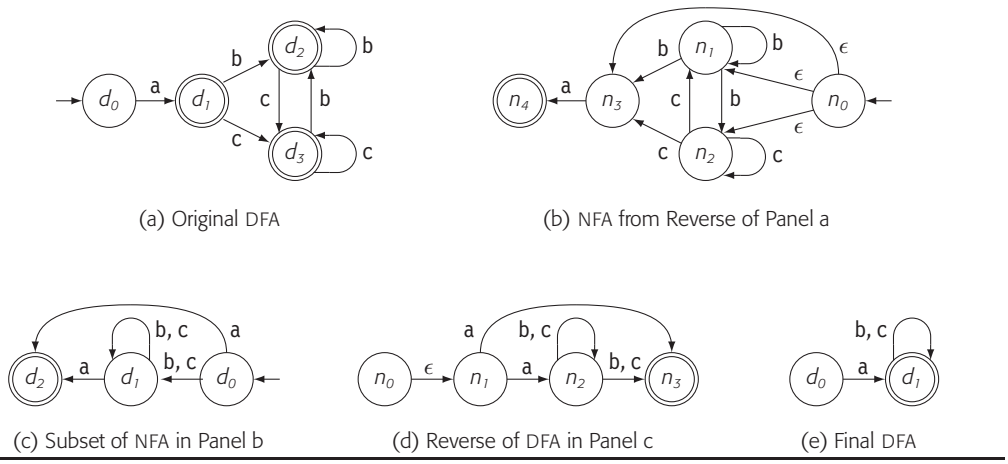


FIGURE 2.19 Applying Brzozowski's Algorithm to the DFA for $a\,(b \mid c)^*$

diagram. If we apply the subset construction to an NFA that has multiple paths from the start state for some prefix, the construction combines those paths into a single path. The resulting DFA has no duplicate prefixes. Brzozowski used this observation to devise an alternative minimization algorithm that directly constructs the minimal DFA from either an NFA or a DFA.

For NFAs built with Thompson's construction, reachability is not an issue. It can arise in minimizing an arbitrary NFA.

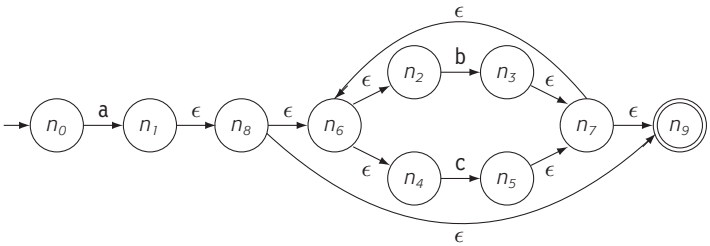
For an NFA n, let *reverse(n)* be the NFA obtained by reversing the direction of all the transitions, making the initial state into a final state, adding a new initial state, and connecting it to all of the states that were final states in n. Further, let *reachable(n)* be a function that returns the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

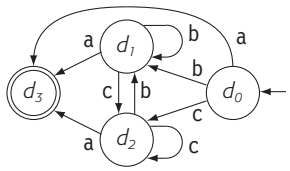
reachable(subset(reverse(reachable(subset(reverse(n)))))).

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.)

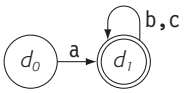
Figure 2.19 shows the steps that the algorithm takes to minimize the DFA for $a\,(b \mid c)^*$ produced in the previous two subsections. Panel (a) repeats the DFA from Figure 2.7. Applying *reverse* to this DFA produces the NFA shown in panel (b). Next, the algorithm applies *subset* to this NFA, to produce the DFA shown in panel (c). *Reverse* applied to panel (c) yields the NFA in panel (d). Applying *subset* to this NFA produces the final DFA shown in panel (e), which is minimal. Note



(a) NFA for $a(b \mid c)^*$



(b) Subset of Reverse of Panel a



(c) Subset of Reverse of Panel c

FIGURE 2.20 Brzozowski's Algorithm Applied to the NFA from Figure 2.5

that reachability did not play a role in this example.

Given that the subset construction can construct an exponentially large set of states, Brzozowski's algorithm has the potential to be expensive. Hopcroft's algorithm, described in Section 2.4.4 has asymptotic complexity of $O(|N|\,|\Sigma|\,\log_2(|N|)$, where N is the set of states in the input FA.

The tradeoff between the two algorithms is not straightforward. Studies of the running times of various FA minimization techniques suggest, however, that the actual running times depend on specific properties of the FA. In practice, Brzozowski's algorithm appears to perform reasonably well.

Furthermore, the implementation of Brzozowski's algorithm will almost certainly be simpler than that of Hopcroft's algorithm. Since Brzozowski's algorithm produces a DFA, it can be applied directly to the output of Thompson's construction, eliminating an extra application of the subset construction.

The first application of *subset* is, effectively, free.

Figure 2.20 shows the steps that the algorithm takes when applied directly to the NFA that Thompson's construction built for $a\,(b \mid c)^*$. Panel (a) shows the original NFA. Panel (b) shows the DFA constructed by applying *reverse* and then *subset* to the NFA. Panel c shows the final DFA. The two reversed NFAs are left as an exercise for the reader.

Use in a Scanner Generator Because Brzozowski's algorithm uses the subset construction, its first step combines all of the final states into

a single representative state. As described in Section 2.4.5, the scanner needs a DFA where each final state maps to one syntactic category. We can modify Hopcroft's algorithm to maintain this map; Brzozowski's algorithm has no similar fix.

Applying Brzozowski's algorithm to an NFA will produce a DFA.

To use Brzozowski's algorithm in a scanner generator, the tool would need to build an NFA for each rule and use Brzozowski's algorithm on the individual NFAs. It could then combine the individual DFAs with ϵ transitions and use the subset construction to produce a final DFA. As before, if this application of the subset construction merges final states, it must assign the new final state the syntactic category of the higher priority rule. The resulting DFA will not be minimal. It will, however, be smaller than the DFA produced without minimization.

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

Revise end of 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 [193, 231, 315].

Kleene [223] 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 [261]. 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 first describe the application of these ideas to automate scanner construction [206]. The subset construction derives from Rabin and Scott [291].

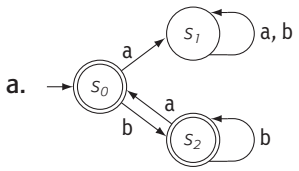
The DFA minimization algorithm in Section 2.4.4 is due to Brzozowski in 1962 [60]. Hopcroft's algorithm, in Section 2.4.4, appeared in 1971. It has found application to many different problems, including detecting when two program variables always have the same value [22]. Several authors have compared DFA minimization techniques and their performance [328, 344]. Many authors have looked at the construction and minimization of acyclic DFAs [112, 343, 345].

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 [273]. Several authors have examined tradeoffs in scanner implementation. Jones [207] 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 maximal munch scanner is due to Reps [297].

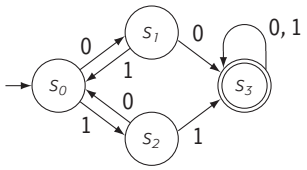
EXERCISES

Section 2.2

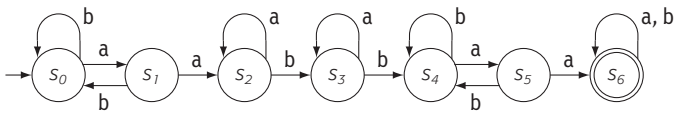
1. Describe informally the languages accepted by the following FAs:



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 $\Sigma = \{a, \text{b}, \text{c}, \text{d}\}$, 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

Section 2.4

7. Consider the three regular expressions:

$(ab \mid ac)^*$

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

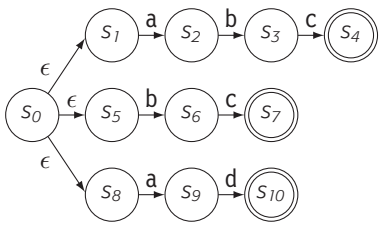
$(01 \mid 10 \mid 00)^*\ 11$

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

b. Convert the NFAs to DFAs.

c. Minimize the DFAs.

8. NEW: Apply Hopcroft's minimization algorithm to the NFA shown below.



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

Section 2.5

10. 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
11. For each of the DFAs in the previous exercise, build a direct-coded scanner.
12. This chapter describes several ways to implement a DFA. Another alternative would use mutually recursive functions to implement a scanner. Discuss the advantages and disadvantages of such an implementation.
13. NEW: Figure 2.17 shows an algorithm that discovers identical columns in the transition function, δ. Give an algorithm that constructs both the character classifier, a map from characters to column numbers, and the reduced form of δ.

    Assume that column i of δ corresponds to the i^{th} character in the alphabet, Σ, denoted Σ_i.
14. Figure 2.13 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, on average, uses less space.

Section 2.6

15. NEW: Apply Brzozowski's algorithm to the DFA from question 8.