3

Scanning–Theory and Practice

In this chapter, we discuss the theoretical and practical issues involved in building a scanner. In Section 3.1, we give an overview of how a scanner operates. In Section 3.2, we introduce a declarative *regular expression* notation that is well-suited to the formal definition of tokens. In Section 3.3, the correspondence between regular expressions and *finite automata* is studied. Finite automata are especially useful because they are procedural in nature and can be directly executed to read characters and group them into tokens. As a case study, a well-known scanner generator, *Lex*, is considered in some detail in Section 3.4. Lex takes token definitions (in a declarative form—regular expressions) and produces a complete scanner subprogram, ready to be compiled and executed. Section 3.5 briefly considers other scanner generators.

In Section 3.6, we discuss the practical considerations needed to build a scanner and integrate it with the rest of the compiler. These considerations include anticipating the tokens and contexts that may complicate scanning, avoiding performance bottlenecks, and recovering from lexical errors. We conclude the chapter with Section 3.7, which explains how scanner generators, such as Lex, translate regular expressions into finite automata and how finite automata may be converted to equivalent regular expressions. Readers who prefer to view a scanner generator as simply a black box may skip this section. However, the material does serve to reinforce the concepts of regular expressions and finite automata introduced in earlier sections. The section also illustrates how finite automata can be built, merged, simplified, and even optimized.

3.1 Overview of a Scanner

The primary function of a scanner is to transform a character stream into a token stream. A scanner is sometimes called a **lexical analyzer**, or **lexer**. The names "scanner," "lexical analyzer," and "lexer" are used interchangeably. The ac scanner discussed in Chapter Chapter:global:two was simple and could easily be coded by any competent programmer. In this chapter, we develop a thorough and systematic approach to scanning that will allow us to create scanners for complete programming languages.

We introduce formal notations for specifying the precise structure of tokens. At first glance, this may seem unnecessary because of the simple token structure found in most programming languages. However, token structure can be more detailed and subtle than one might expect. For example, consider simple quoted strings in C, C++, and Java. The body of a string can be any sequence of characters except a quote character, which must be escaped. But is this simple definition really correct? Can a newline character appear in a string? In C it cannot, unless it is escaped with a backslash. Doing this avoids a "runaway string" that, lacking a closing quote, matches characters intended to be part of other tokens. While C, C++, and Java allow escaped newlines in strings, Pascal forbids them. Ada goes further still and forbids all unprintable characters (precisely because they are normally unreadable). Similarly, are null (zero-length) strings allowed? C, C++, Java, and Ada allow them, but Pascal forbids them. In Pascal, a string is a packed array of characters and zero-length arrays are disallowed.

A precise definition of tokens is necessary to ensure that lexical rules are clearly stated and properly enforced. Formal definitions also allow a language designer to anticipate design flaws. For example, virtually all languages allow fixed decimal numbers, such as 0.1 and 10.01. But should .1 or 10. be allowed? In C, C++, and Java, they are. But in Pascal and Ada they are not—and for an interesting reason. Scanners normally seek to match as many characters as possible so that, for example, ABC is scanned as one identifier rather than three. But now consider the character sequence 1..10. In Pascal and Ada, this should be interpreted as a range specifier (1 to 10). However, if we were careless in our token definitions, we might well scan 1..10 as two real literals, 1. and .10, which would lead to an immediate (and unexpected) syntax error. (The fact that two real literals *cannot* be adjacent is reflected in the **context-free grammar** (CFG), which is enforced by the parser, not the scanner.)

When a formal specification of token and program structure is given, it is possible to examine a language for design flaws. For example, we could analyze all pairs of tokens that can be adjacent to each other and determine whether the two if catenated might be incorrectly scanned. If so, a separator may be required. In the case of adjacent identifiers and reserved words, a blank space (whitespace) suffices to distinguish the two tokens. Sometimes, though, the lexical or program syntax might need to be redesigned. The point is that language design is far more involved than one might expect, and formal specifications allow flaws to be discovered before the design is completed.

All scanners, independent of the tokens to be recognized, perform much the same function. Thus writing a scanner from scratch means reimplementing components that are common to all scanners; this means a significant duplication of effort. The goal of a **scanner generator** is to limit the effort of building a scanner to that of specifying which tokens the scanner is to recognize. Using a formal notation, we tell the scanner generator what tokens we want recognized. It then is the generator's responsibility to produce a scanner that meets our specification. Some generators do not produce an entire scanner. Rather, they produce tables that can be used with a standard driver program, and this combination of generated tables and standard driver yields the desired custom scanner.

Programming a scanner generator is an example of **declarative programming**. That is, unlike in ordinary, or **procedural programming**, we do not tell a scanner generator *how* to scan but simply *what* to scan. This is a higher-level approach and in many ways a more natural one. Much recent research in computer science is directed toward declarative programming styles; examples are database query languages and Prolog, a "logic" programming language. Declarative programming is most successful in limited domains, such as scanning, where the range of implementation decisions that must be made automatically is limited. Nonetheless, a long-standing (and as yet unrealized) goal of computer scientists is to automatically generate an entire production-quality compiler from a specification of the properties of the source language and target computer.

Although our primary focus in this book is on producing correct compilers, performance is sometimes a real concern, especially in widely used "production compilers." Surprisingly, even though scanners perform a simple task, they can be significant performance bottlenecks if poorly implemented. This because scanners must wade through the text of a program character-by-character.

Suppose we want to implement a very fast compiler that can compile a program in a few seconds. We'll use 30,000 lines a minute (500 lines a second) as our goal. (Compilers such as "Turbo C++" achieve such speeds.) If an average line contains 20 characters, the compiler must scan 10,000 characters per second. On a 10 MIPS processor (10,000,000 instructions executed per second), even if we did nothing but scanning, we'd have only 1,000 instructions per input character to spend. But because scanning isn't the only thing a compiler does, 250 instructions per character is more realistic. This is a rather tight budget, considering that even a simple assignment takes several instructions on a typical processor. Although multi-MIPS processors are common these days and 30,000 lines per minute is an ambitious speed, clearly a poorly coded scanner can dramatically impact a compiler's performance.

3.2 Regular Expressions

Regular expressions are a convenient way to specify various simple (although possibly infinite) sets of strings. They are of practical interest because they can specify the structure of the tokens used in a programming language. In particular, you can use regular expressions to program a scanner generator.

Regular expressions are widely used in computer applications other than compilers. The Unix utility grep uses them to define search patterns in files. Unix shells allow a restricted form of regular expressions when specifying file lists for a command. Most editors provide a "context search" command that enables you to specify desired matches using regular expressions.

A set of strings defined by regular expressions is called a **regular set**. For purposes of scanning, a token class is a regular set whose structure is defined by a regular expression. A particular instance of a token class is sometimes called a **lexeme**; however, we simply call a string in a token class an *instance* of that token. For example, we call the string `abc` an identifier if it matches the regular expression that defines the set of valid identifier tokens.

Our definition of regular expressions starts with a finite character set, or **vocabulary** (denoted Σ). This vocabulary is normally the character set used by a computer. Today, the *ASCII* character set, which contains 128 characters, is very widely used. Java, however, uses the *Unicode* character set. This set includes all of the ASCII characters as well as a wide variety of other characters.

An empty, or null, string is allowed (denoted λ). This symbol represents an empty buffer in which no characters have yet been matched. It also represents an optional part of a token. Thus an integer literal may begin with a plus or minus, or, if it is unsigned, it may begin with λ.

Strings are built from characters in the character set Σ via *catenation* (that is, by joining individual characters to form a string). As characters are catenated to a string, it grows in length. For example, the string `do` is built by first catenating `d` to λ and then catenating `o` to the string `d`. The null string, when catenated with any string s, yields s. That is, $s\ \lambda \equiv \lambda\ s \equiv s$. Catenating λ to a string is like adding 0 to an integer—nothing changes.

Catenation is extended to sets of strings as follows. Let P and Q be sets of strings. The symbol $\in$ represents set membership. If $s_1 \in P$ and $s_2 \in Q$, then string $s_1 s_2 \in (P\ Q)$. Small finite sets are conveniently represented by listing their elements, which can be individual characters or strings of characters. Parentheses are used to delimit expressions, and |, the alternation operator, is used to separate alternatives. For example, D, the set of the ten single digits, is defined as $D = (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$. (In this text, we often use abbreviations such as $(0 \mid \ldots \mid 9)$ rather than enumerate a complete list of alternatives. The "..." symbol is *not* part of our regular expression notation.)

A **meta-character** is any punctuation character or regular expression operator. A meta-character must be quoted when used as an ordinary character in order to avoid ambiguity. (Any character or string may be quoted, but unnecessary quotation is avoided to enhance readability.) The characters (,), ' , *, +, and | are meta-characters. For example the expression ('(' | ')' | ; | ,) defines four single character tokens (left parenthesis, right parenthesis, semicolon, and comma) that we might use in a programming language. The parentheses are quoted to show they are meant to be individual tokens and not delimiters in a larger regular expression.

Alternation can be extended to sets of strings. Let P and Q be sets of strings. Then string $s \in (P \mid Q)$ if, and only if, $s \in P$ or $s \in Q$. For example, if LC is the set of lowercase letters and UC is the set of uppercase letters, then $(LC \mid UC)$ denotes the set of all letters (in either case).

Large (or infinite) sets are conveniently represented by operations on finite sets of characters and strings. Catenation and alternation may be used. A third operation, **Kleene closure**, as defined below, is also allowed. The operator * is the postfix *Kleene closure operator*. Here's an example. Let P be a set of strings. Then $P^\star$ represents all strings formed by the catenation of zero or more selections (possibly repeated) from P. (Zero selections are represented by λ.) For example, $LC^\star$ is the set of all words composed only of lowercase letters and of any length (including the zero-length word, λ).

Precisely stated, a string $s \in P^\star$ if, and only if, s can be broken into zero or more pieces: $s = s_1 s_2 ... s_n$ such that each $s_i \in P (n \geq 0, 1 \leq i \leq n)$. We explicitly allow $n = 0$ so that λ is always in $P^\star$.

Now that we've introduced the operators used in regular expressions, we can define regular expressions as follows.

- $\emptyset$ is a regular expression denoting the empty set (the set containing no strings). $\emptyset$ is rarely used but is included for completeness.

- λ is a regular expression denoting the set that contains only the empty string. This set is not the same as the empty set because it does contain one element.

- A string s is a regular expression denoting a set containing the single string s. If s contains meta-characters, s can be quoted to avoid ambiguity.

- If A and B are regular expressions, then $A \mid B$, $A\ B$, and $A^\star$ are also regular expressions. They denote, respectively, the alternation, catenation, and Kleene closure of the corresponding regular sets.

Each regular expression denotes a regular set. Any finite set of strings can be represented by a regular expression of the form $(s_1 \mid s_2 \mid ... \mid s_k)$. Thus the reserved words of ANSI C can be defined as (`auto` | `break` | `case` | ...).

The following additional operations are also useful. They are not strictly necessary because their effect can be obtained (perhaps somewhat clumsily) using the three standard regular operators (alternation, catenation, and Kleene closure).

- P^+, sometimes called **positive closure**, denotes all strings consisting of one or more strings in P catenated together: $P^\star = (P^+ \mid \lambda)$ and $P^+ = P\ P^\star$. For example, the expression $(0 \mid 1)^+$ is the set of all strings containing one or more bits.

- If A is a set of characters, Not(A) denotes $(\Sigma - A)$, that is, all *characters* in Σ not included in A. Since Not(A) can never be larger than Σ and Σ is finite, Not(A) must also be finite. It therefore is regular. Not(A) does not contain λ because λ is not a character (it is a zero-length string). As an example, Not(Eol) is the set of all characters excluding Eol (the end-of-line character; in Java or C, `\n`).

It is possible to extend Not() to strings, rather than just Σ. If S is a set of strings, we can define $\overline{S}$ to be ($\Sigma^\star$ - S), that is, the set of all strings except those in S. Although $\overline{S}$ is usually infinite, it also is regular if S is. (See Exercise 18.)

- If k is a constant, the set A^k represents all strings formed by catenating k (possibly different) strings from A. That is, A^k = (AAA ...) (k copies). Thus $(0 \mid 1)^{32}$ is the set of all bit strings exactly 32 bits long.

3.2.1 Examples

Next, we explore how regular expressions can be used to specify tokens. Let D be the set of the ten single digits and L be the set of all letters (52 in all). Then the following is true.

- A Java or C++, single-line comment that begins with // and ends with Eol can be defined as

$$Comment = // \ \mathsf{Not(Eol)}^\star \ \mathsf{Eol}$$

  This regular expression says that a comment begins with two slashes and ends at the *first* end-of-line. Within the comment, any sequence of characters is allowed that does not contain an end-of-line. (This guarantees that the first end-of-line we see ends the comment.)

- A fixed-decimal literal (for example, 12.345) can be defined as

$$Lit = D^+.D^+$$

  One or more digits must be on both sides of the decimal point, so this definition excludes .12 and 35.

- An optionally signed integer literal can be defined as

$$IntLiteral = ('+' \mid - \mid \lambda\ )\ D^+$$

  An integer literal is one or more digits preceded by a plus or minus or no sign at all (λ). So that the plus sign is not confused with the Kleene closure operator, it is quoted.

- A more complicated example is a comment delimited by ## markers, which allows single #'s within the comment body:

$$Comment2 = \#\# \ ((\# \mid \lambda) \ \ \mathsf{Not}(\#))^\star \ \#\#$$

  Any # that appears within this comment's body must be followed by a non-# so that a premature end of comment marker, ##, is not found.

All *finite* sets are regular. However, some but not all *infinite* sets are regular. For example, consider the set of balanced brackets of the form [[[. . .]]]. This set is defined formally as $\{[^m]^m \mid m \geq 1\}$. This is a set that is known not to be regular. The problem is that any regular expression that tries to define it either does not get all balanced nestings or includes extra, unwanted strings. (Exercise 14 proves this.)

It is easy to write a CFG that defines balanced brackets precisely. In fact, all regular sets can be defined by CFGs. Thus, the bracket example shows that CFGs are a more powerful descriptive mechanism than regular expressions. Regular expressions are, however, quite adequate for specifying token-level syntax. Moreover, for every regular expression we can create an efficient device, called a *finite automaton*, that recognizes exactly those strings that match the regular expression's pattern.

3.3 Finite Automata and Scanners

A **finite automation** (FA) can be used to recognize the tokens specified by a regular expression. An FA (plural: finite automata) is a simple, idealized computer that recognizes strings as belonging to regular sets. An FA consists of the following:

- A finite set of *states*
- A finite *vocabulary*, denoted Σ
- A set of *transitions* (or *moves*) from one state to another, labeled with characters in Σ
- A special state called the *start* state
- A subset of the states called the *accepting*, or *final*, states

These components of an FA can be represented graphically as shown in Figure 3.1.

An FA also can be represented graphically using a **transition diagram**, composed of the components shown in Figure 3.1. Given a transition diagram, we begin at the start state. If the next input character matches the label on a transition from the current state, we go to the state to which it points. If no move is possible, we stop. If we finish in an accepting state, the sequence of characters read forms a valid token; otherwise, a valid token has not seen. In the transition diagram shown in Figure 3.2, the valid tokens are the strings described by the regular expression $(ab(c)^\star)^\star$.

As an abbreviation, a transition may be labeled with more than one character (for example, Not(c)). The transition may be taken if the current input character matches any of the characters labeling the transition.

Reserving space for: finaut

Figure 3.1: The four parts of a finite automation.

Reserving space for: transdia

Figure 3.2: Transition diagram of an FA.

3.3.1 Deterministic Finite Automata

An FA that always has a *unique* transition (for a given state and character) is a **deterministic finite automation** (DFA). DFAs are easy to program and are often used to drive a scanner. A DFA is conveniently represented in a computer by a **transition table**. A transition table, T, is a two-dimensional array indexed by a DFA state and a vocabulary symbol. Table entries are either a DFA state or an error flag (often represented as a blank table entry). If we are in state s and read character c, then $T[s,c]$ will be the next state we visit, or $T[s,c]$ will contain an error flag indicating that c cannot extend the current token. For example, the regular expression

$$// \ \mathsf{Not(Eol)}^{\star}\mathsf{Eol}$$

which defines a Java or C++ single-line comment, might be translated as shown in Figure 3.3. The corresponding transition table is shown in Figure 3.4.

Reserving space for: transregexp

Figure 3.3: Translation of the regular expression // Not(Eol)*Eol.

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

Figure 3.4: Transition table of the regular expression // Not(Eol)*Eol.

A full transition table will contain one column for each character. To save space, *table compression* is sometimes utilized. In that case, only nonerror entries are explicitly represented in the table. This is done by using hashing or linked structures.

Any regular expression can be translated into a DFA that accepts (as valid tokens) the set of strings denoted by the regular expression. This translation can be done manually by a programmer or automatically by a scanner generator.

Coding the DFA

A DFA can be coded in one of two forms:

1. Table-driven
2. Explicit control

In the *table-driven* form, the transition table that defines a DFA's actions is explicitly represented in a run-time table that is "interpreted" by a driver program. In the *explicit control form*, the transition table that defines a DFA's actions appears implicitly as the control logic of the program. Typically, individual program statements correspond to distinct DFA states. For example, suppose *CurrentChar* is the current input character. End-of-file is represented by a special character value, Eof. Using the DFA for the Java comments illustrated previously, the two approaches would produce the programs illustrated in Figure 3.5 and 3.6.

```
/⋆ Assume $CurrentChar$ contains the first character to be scanned ⋆/
$State \leftarrow StartState$
while true do
    $NextState \leftarrow T[State, CurrentChar]$
    if $NextState$ = error
    then break
    $State \leftarrow NextState$
    READ( $CurrentChar$ )
if $State \in AcceptingStates$
then  /⋆ Return or process valid token ⋆/
else  /⋆ signal a lexical error ⋆/
```

Figure 3.5: Scanner driver interpreting a transition table.

```
/⋆ Assume $CurrentChar$ contains the first character to be scanned ⋆/
if $CurrentChar$ = '/'
then
    READ( $CurrentChar$ )
    if $CurrentChar$ = '/'
    then
        repeat
            READ( $CurrentChar$ )
        until $CurrentChar \in \{$ Eol, Eof $\}$
    else  /⋆ Signal a lexical error ⋆/
else  /⋆ Signal a lexical error ⋆/
if $CurrentChar$ = Eol
then  /⋆ Return or process valid token ⋆/
else  /⋆ Signal a lexical error ⋆/
```

Figure 3.6: Explicit control scanner.

The table-driven form is commonly produced by a scanner generator; it is token-independent. It uses a simple driver that can scan any token, provided the transition table is properly stored in T. The explicit control form may be produced automatically or by hand. The token being scanned is "hardwired" into the code. This form of a scanner is usually easy to read and often is more efficient, but it is specific to a single token definition.

Following are two more examples of regular expressions and their corresponding DFAs.

1. A FORTRAN-like real literal (which requires either digits on either or both sides of a decimal point or just a string of digits) can be defined as

$$RealLit = (D^{+}\ (\lambda\ |\ .\ ))\ |\ (D^{\star}\ .\ D^{+})$$

Reserving space for: iddia

Figure 3.7: DFA for FORTRAN-like real literal.

which corresponds to the DFA shown in Figure 3.7.

2. An identifier consisting of letters, digits, and underscores. It begins with a letter and allows no adjacent or trailing underscores. It may be defined as

   $$ID = L\ (L \mid D)^{\star}\ (\_(L \mid D)^{\star})^{+}.$$

   This definition includes identifiers such as sum or `unit_cost` but excludes `_one` and `two_` and `grand__total`. The corresponding DFA is shown in Figure 3.8.

Transducers

So far, we haven't saved or processed the characters we've scanned—they've been matched and then thrown away. It is useful to add an output facility to an FA; this makes the FA a **transducer**. As characters are read, they can be transformed and catenated to an output string. For our purposes, we limit the transformation operations to saving or deleting input characters. After a token is recognized, the transformed input can be passed to other compiler phases for further processing. We use the notation shown in Figure 3.9. For example, for Java and C++ comments, we might write the DFA shown in Figure 3.10. A more interesting example is given by Pascal-style quoted strings, according to the regular expression

$$(\text{" } (\text{Not(")} \mid \text{""})^{\star} \text{ "}).$$

A corresponding transducer might be as shown in Figure 3.11. The input `"""Hi"""` would produce output `"Hi"`.

Reserving space for: dfa

Figure 3.8: DFA for identifiers with underscores.

3.4 The Lex Scanner Generator

Next, as a case study in the design of scanner generation tools, we discuss a very popular scanner generator, Lex. Later, we briefly discuss several other scanner generators.

Lex was developed by M. E. Lesk and E. Schmidt of AT&T Bell Laboratories. It is used primarily with programs written in C or C++ running under the Unix operating system. Lex produces an entire scanner module, coded in C, that can be compiled and linked with other compiler modules. A complete description of Lex can be found in [LS75] and [LMB92]. Flex (see [Pax88]) is a widely used, freely distributed reimplementation of Lex that produces faster and more reliable scanners. Valid Lex scanner specifications may, in general, be used with Flex without modification.

The operation of Lex is illustrated in Figure 3.12. Here are the steps:

1. A scanner specification that defines the tokens to be scanned and how they are to be processed is presented to Lex.

2. Lex then generates a complete scanner coded in C.

3. This scanner is compiled and linked with other compiler components to create a complete compiler.

Using Lex saves a great deal of effort programming a scanner. Many low-level details of the scanner (reading characters efficiently, buffering them, matching characters against token definitions, and so on) need not be explicitly programmed. Rather, we can focus on the character structure of tokens and how they are to be processed.

Reserving space for: notat

Figure 3.9: Transducer notation.

The primary purpose of this section is to show how regular expressions and related information are presented to scanner generators. A good way to learn how to use Lex is to start with the simple examples presented here and then gradually generalize them to solve the problem at hand. To inexperienced readers, Lex's rules may seem unnecessarily complex. It is best to keep in mind that the key is always the specification of tokens as regular expressions. The rest is there simply to increase efficiency and handle various details.

3.4.1 Defining Tokens in Lex

Lex's approach to scanning is simple. It allows the user to associate regular expressions with commands coded in C (or C++). When input characters that match the regular expression are read, the command is executed. As a user of Lex, you don't need to tell it *how* to match tokens. You need only tell it *what* you want done when a particular token is matched.

Lex creates a file `lex.yy.c` that contains an integer function `yylex()`. This function is normally called from the parser whenever another token is needed. The value that `yylex()` returns is the token code of the token scanned by Lex. Tokens such as whitespace are deleted simply by having their associated command not return anything. Scanning continues until a command with a return in it is executed.

Figure 3.13 illustrates a simple Lex definition for the three reserved words—`f`, `i`, and `p`—of the ac language introduced in Chapter Chapter:global:two. When a string matching any of these three reserved keywords is found, the appropriate token code is returned. It is vital that the token codes that are returned when a token is matched are identical to those expected by the parser. If they are not, the parser won't "see" the same token sequence produced by the scanner. This will cause the parser to generate false syntax errors based on the incorrect token stream it sees.

Reserving space for: dfajava

Figure 3.10: The DFA for Java and C++ comments.

It is standard for the scanner and parser to share the definition of token codes to guarantee that consistent values are seen by both. The file `y.tab.h`, produced by the Yacc parser generator (see Chapter Chapter:global:seven), is often used to define shared token codes. A Lex token specification consists of three sections delimited by the pair %%. The general form of a Lex specification is shown in Figure 3.14.

In the simple example shown in Figure 3.13, we use only the second section, in which regular expressions and corresponding C code are specified. The regular expressions are simple single-character strings that match only themselves. The code executed returns a constant value representing the appropriate ac token.

We could quote the strings representing the reserved keywords (`f`, `i`, or `p`), but since these strings contain no delimiters or operators, quoting them is unnecessary. If you want to quote such strings to avoid any chance of misinterpretation, that's fine with Lex.

3.4.2 The Character Class

Our specification so far is incomplete. None of the other tokens in ac have been correctly handled, particularly identifiers and numbers. To do this, we introduce a useful concept: the **character class**. A character class is a set of characters treated identically in a token definition. Thus, in the definition of an ac identifier, all letters (except `f`, `i`, and `p`) form a class, since any of them can be used to form an identifier. Similarly in a number, any of the ten digits characters can be used.

A character class is delimited by `[` and `]`; individual characters are catenated without any quotation or separators. However `\`, `^`, `]`, and `-` must be escaped because of their special meanings in character classes. Thus [`xyz`] represents the class that can match a single `x`, `y`, or `z`. The expression `[\])]` represents the class

Reserving space for: dfatrans

Figure 3.11: The transducer for Pascal quoted strings.

Reserving space for: lexscan

Figure 3.12: The operation of the Lex scanner generator.

```
%%
f           { return(FLOATDCL); }
i           { return(INTDCL); }
p           { return(PRINT); }
%%
```

Figure 3.13: A Lex definiton for ac's reserved words.

```
declarations
%%
regular expression rules
%%
subroutine definitions
```

Figure 3.14: The structure of Lex definiton files.

Character Class	Set of Characters Denoted
`[abc]`	Three characters: `a`, `b`, and `c`
`[cba]`	Three characters: `a`, `b`, and `c`
`[a-c]`	Three characters: `a`, `b`, and `c`
`[aabbcc]`	Three characters: `a`, `b`, and `c`
`[^abc`	All characters except `a`, `b`, and `c`
`[\^\-\]]`	Three characters: `^`, `-`, and `]`
`[^]`	All characters
`"[abc]"`	Not a character class. This is an example of one five-character *string*: `[abc]`.

Figure 3.15: Lex character class definitions.

```
%%
[a-eghj-oq-z]       { return(ID); }
%%
```

Figure 3.16: A Lex definition for ac's identifiers.

that can match a single `]` or `)`. (The `]` is escaped so that it isn't misinterpreted as the end-of-character-class symbol.)

Ranges of characters are separated by a `-`; for example, `[x-z]` is the same as `[xyz]`. `[0-9]` is the set of all digits, and `[a-zA-Z]` is the set of all letters, both uppercase and lowercase. \ is the escape character; it is used to represent unprintables and to escape special symbols. Following C conventions, `\n` is the newline (that is, end-of-line), `\t` is the tab character, `\\` is the backslash symbol itself, and `\010` is the character corresponding to `10` in octal (base 8) form.

The `^` symbol complements a character class; it is Lex's representation of the Not() operation. For example, `[^xy]` is the character class that matches any single character except `x` and `y`. The `^` symbol applies to all characters that follow it in the character class definition, so `[^0-9]` is the set of all characters that aren't digits. `[^]` can be used to match all characters. (Avoid the use of `\0` in character classes because it can be confused with the null character's special use as the end-of-string terminator in C.) Figure 3.15 illustrates various character classes and the character sets they define.

Using character classes, we can easily define ac identifiers, as shown in Figure 3.16. The character class includes the range of characters, `a` to `e`, then `g` and `h`, and then the range `j` to `o`, followed by the range `q` to `z`. We can concisely represent the 23 characters that may form an ac identifier without having to enumerate them all.

```
%%
(" ")+                          { /* delete blanks */}
f                               { return(FLOATDCL); }
i                               { return(INTDCL); }
p                               { return(PRINT); }
[a-eghj-oq-z]                   { return(ID); }
([0-9]+)|([0-9]+"."[0-9]+)      { return(NUM);  }
"="                             { return(ASSIGN); }
"+"                             { return(PLUS); }
"-"                             { return(MINUS); }
%%
```

Figure 3.17: A Lex definition for ac's tokens.

3.4.3 Using Regular Expressions to Define Tokens

Tokens are defined using regular expressions. Lex provides the standard regular expression operators, as well as others. Catenation is specified by the juxtaposition of two expressions; no explicit operator is used. Thus `[ab][cd]`will match any of `ad`, `ac`, `bc`, or `bd`. Individual letters and numbers match themselves when outside of character class brackets. Other characters should be quoted (to avoid misinterpretation as regular expression operators.) For example, `while` (as used in C, C++, and Java) can be matched by the expressions `while`, `"while"`, or `[w][h][i][l][e]`.

Case *is* significant. The alternation operator is |. As usual, parentheses can be used to control grouping of subexpressions. Therefore, to match the reserved word `while` and allow any mixture of uppercase and lowercase (as required in Pascal and Ada), we can use

`(w|W)(h|H)(i|I)(l|L)(e|E)`

Postfix operators * (Kleene closure) and $^+$ (positive closure) are also provided, as is `?` (optional inclusion). For example, `expr?` matches `expr` zero times or once. It is equivalent to `(expr)` | λ and obviates the need for an explicit λ symbol. The character "." matches any single character (other than a newline). The character ^ (when used outside a character class) matches the beginning of a line. Similarly, the character $ matches the end of a line. Thus `^A.*` `e` $ could be used to match an entire line that begins with `A` and ends with `e`. We now define all of ac's tokens using Lex's regular expression facilities. This is shown in Figure 3.17.

Recall that a Lex specification of a scanner consists of three sections. The first, not used so far, contains symbolic names associated with character classes and regular expressions. There is one definition per line. Each definition line contains an identifier and a definition string, separated by a blank or tab. The { and } symbols signal the macro-expansion of a symbol defined in the first section. For example, in the definition

`Letter [a-zA-Z]`

```
%%
Blank                                   " "
Digits                                  [0-9]+
Non_f_i_p                               [a-eghj-oq-z]
%%
{Blank}+                                { /* delete blanks */}
f                                       { return(FLOATDCL); }
i                                       { return(INTDCL); }
p                                       { return(PRINT); }
{Non_f_i_p}                             { return(ID); }
{Digits}|({Digits}"."{Digits})          { return(NUM);  }
"="                                     { return(ASSIGN); }
"+"                                     { return(PLUS); }
"-"                                     { return(MINUS); }
%%
```

Figure 3.18: An alternative definition for ac's tokens.

the expression {Letter} expands to [a-zA-Z]. Symbolic definitions can often make Lex specifications easier to read, as illustrated in Figure 3.18.

In the first section can also include source code, delimited by %{ and %}, that is placed before the commands and regular expressions of section two. This source code may include statements, as well as variable, procedure, and type declarations that are needed to allow the commands of section two to be compiled. For example,

```
%{
#include "tokens.h"
%}
```

can include the definitions of token values returned when tokens are matched.

As shown earlier in the chapter, Lex's second section defines a table of regular expressions and corresponding commands in C. The first blank or tab not escaped or not part of a quoted string or character class is taken as the end of the regular expression. Thus you should avoid embedded blanks that are within regular expressions.

When an expression is matched, its associated command is executed. If an input sequence matches no expression, the sequence is simply copied verbatim to the standard output file. Input that is matched is stored in a global string variable yytext (whose length is yyleng). Commands may alter yytext in any way. The default size of yytext is determined by YYLMAX, which is initially defined to be 200. *All* tokens, even those that will be ignored like comments, are stored in yytext. Hence, you may need to redefine YYLMAX to avoid overflow. An alternative approach to scanning comments that is not prone to the danger of overflowing yytext involves the use of start conditions (see [LS75] or [LMB92]).

Flex, an improved version of Lex discussed in the next section, automatically extends the size of yytext when necessary. This removes the danger that a very long token may overflow the text buffer.

The content of yytext is overwritten as each new token is scanned. Therefore you must be careful if you return the text of a token by simply returning a pointer into yytext. You must copy the content of yytext (by using perhaps strcpy()) *before* the next call to yylex().

Lex allows regular expressions to overlap (that is, to match the same input sequences). In the case of overlap, two rules are used to determine which regular expression is matched:

1. The longest possible match is performed. Lex automatically buffers characters while deciding how many characters can be matched.

2. If two expressions match exactly the same string, the earlier expression (in order of definition in the Lex specification) is preferred.

Reserved words, for example, are often special cases of the pattern used for identifiers. So their definitions are placed before the expression that defines an identifier token. Often a "catch-all" pattern is placed at the very end of section two. It is used to catch characters that don't match any of the earlier patterns and hence are probably erroneous. Recall that "." matches any single character (other than a newline). It is useful in a catch-all pattern. However, avoid a pattern such as .* because it will consume all characters up to the next newline.

3.4.4 Character Processing Using Lex

Although Lex is often used to produce scanners, it is really a general-purpose character-processing tool, programmed using regular expressions. Lex provides no character-tossing mechanism because this would be too special-purpose. So we may need to process the token text (stored in yytext) before returning a token code. This is normally done by calling a subroutine in the command associated with a regular expression. The definitions of such subroutines may be placed in the final section of the Lex specification. For example, we might want to call a subroutine to insert an identifier into a symbol table before it is returned to the parser. For ac, the line

```
{Non_f_i_p}          {insert(yytext); return(ID);}
```

could do this, with insert defined in the final section. Alternatively, the definition of insert could be placed in a separate file containing symbol table routines. This would allow insert to be changed and recompiled without Lex's having to be rerun. (Some implementations of Lex generate scanners rather slowly.)

In Lex, end-of-file is not handled by regular expressions. A predefined EndFile token, with a token code of zero, is automatically returned when end-of-file is reached at the beginning of a call to yylex(). It is up to the parser to recognize the zero return value as signifying the EndFile token.

If more than one source file must be scanned, this fact is hidden inside the scanner mechanism. `yylex()` uses three user-defined functions to handle character-level I/O:

`input()`	Reads a single character, zero on end-of-file.
`output(c)`	Writes a single character to output.
`unput(c)`	Puts a single character back into the input to be reread.

When `yylex()` encounters end-of-file, it calls a user-supplied integer function named `yywrap()`. The purpose of this routine is to "wrap up" input processing. It returns the value `1` if there is no more input. Otherwise, it returns zero and arranges for `input()` to provide more characters.

The definitions for the `input()`, `output()`, `unput()`, and `yywrap()` functions may be supplied by the compiler writer (usually as C macros). Lex supplies default versions that read characters from the standard input and write them to the standard output. The default version of `yywrap()` simply returns `1`, thereby signifying that there is no more input. (The use of `output()` allows Lex to be used as a tool for producing stand-alone data "filters" for transforming a stream of data.)

Lex-generated scanners normally select the longest possible input sequence that matches some token definition. Occasionally this can be a problem. For example, if we allow FORTRAN-like fixed-decimal literals such as `1.` and `.10` and the Pascal subrange operator "..", then `1..10` will most likely be misscanned as two fixed-decimal literals rather than two integer literals separated by the subrange operator. Lex allows us to define a regular expression that applies only if some other expression immediately follows it. For example, `r/s` tells Lex to match regular expression `r` but only if regular expression `s` immediately follows it. `s` is *right-context*. That is, it isn't part of the token that is matched, but it must be present for r to be matched. Thus `[0-9]+/".."` would match an integer literal, but only if ".." immediately follows it. Since this pattern covers more characters than the one defining a fixed-decimal literal, it takes precedence. The longest match is still chosen, but the right-context characters are returned to the input so that they can be matched as part of a later token.

The operators and special symbols most commonly used in Lex are summarized in Figure 3.19. Note that a symbol sometimes has one meaning in a regular expression and an entirely different meaning in a character class (that is, within a pair of brackets). If you find Lex behaving unexpectedly, it's a good idea to check this table to be sure how the operators and symbols you've used behave. Ordinary letters and digits, as well as symbols not mentioned (such as `@`), represent themselves. If you're not sure whether a character is special, you can always escape it or make it part of a quoted string.

In summary, Lex is a very flexible generator that can produce a complete scanner from a succinct definition. The difficult part of working with Lex is learning its notation and rules. Once you've done this, Lex will relieve you of the many of chores of writing a scanner (for example, reading characters, buffering them, and deciding which token pattern matches). Moreover, Lex's notation for representing regular expressions is used in other Unix programs, most notably the grep pattern matching utility.

Lex can also transform input as a preprocessor, as well as scan it. It provides a number of advanced features beyond those discussed here. It does require that code segments be written in C, and hence it is not language-independent.

3.5 Other Scanner Generators

Lex is certainly the most widely known and widely available scanner generator because it is distributed as part of the Unix system. Even after years of use, it still has bugs, however, and produces scanners too slow to be used in production compilers. This section discussed briefly some of the alternatives to Lex, including Flex, JLex, Alex, Lexgen, GLA, and re2c.

It has been shown that Lex can be improved so that it is always faster than a handwritten scanner [Jac87]. This is done using *Flex*, a widely used, freely distributed Lex clone. It produces scanners that are considerably faster than the ones produced by Lex. It also provides options that allow the tuning of the scanner size versus its speed, as well as some features that Lex does not have (such as support for 8-bit characters). If Flex is available on your system, you should use it instead of Lex.

Lex also has been implemented in languages other than C. JLex [Ber97] is a Lex-like scanner generator written in Java that generates Java scanner classes. It is of particular interest to people writing compilers in Java. Alex [NF88] is an Ada version of Lex. Lexgen [AMT89] is an ML version of Lex.

An interesting alternative to Lex is GLA (Generator for Lexical Analyzers) [Gra88]. GLA takes a description of a scanner based on regular expressions and a library of common lexical idioms (such as "Pascal comment") and produces a *directly executable* (that is, not transition table-driven scanner written in C. GLA was designed with both ease of use and efficiency of the generated scanner in mind. Experiments show it to be typically twice as fast as Flex and only slightly slower than a trivial program that reads and "touches" each character in an input file. The scanners it produces are more than competitive with the best handcoded scanners.

Another tool that produces directly executable scanners is re2c [BC93]. The scanners it produces are easily adaptable to a variety of environments and yet scanning speed is excellent.

Scanner generators are usually included as parts of complete suites of compiler development tools. These suites are often available on Windows and Macintosh systems as well as on Unix systems. Among the most widely used and highly recommended of these are DLG (part of the PCCTS tools suite, [PDC89]), CoCo/R [Moe91], an integrated scanner/parser generator, and Rex [Gro89], part of the Karlsruhe Cocktail tools suite.

Symbol	Meaning in Regular Expressions	Meaning in Character Classes
(	matches with) to group subexpressions.	Represents itself.
)	matches with (to group subexpressions.	Represents itself.
[	Begins a character class.	Represents itself.
]	Represents itself.	Ends a character class.
{	Matches with } to signal macro-expansion.	Represents itself.
}	Matches with { to signal macro-expansion.	Represents itself.
"	Matches with " to delimit strings.	Represents itself.
\	Escapes individual characters. Also used to specify a character by its octal code.	Escapes individual characters. Also used to specify a character by its octal code.
.	Matches any one character except \n.	Represents itself.
|	Alternation (or) operator.	Represents itself.
*	Kleene closure operator (zero or more matches).	Represents itself.
+	Positive closure operator (one or more matches).	Represents itself.
?	Optional choice operator (one or more matches)	Represents itself.
/	Context sensitive matching operator.	Represents itself.
^	Matches only at the beginning of a line.	Complements the remaining characters in the class.
$	Matches only at the end of a line.	Represents itself.
-	Represents itself.	The range of characters operator.

Figure 3.19: Meaning of operators and special symbols in Lex.

3.6 Practical Considerations of Building Scanners

In this section, we discuss the practical considerations involved in building real scanners for real programming languages. As one might expect, the finite automaton model developed earlier in the chapter sometimes falls short and must be supplemented. Efficiency concerns must be addressed. In addition, some provision for error handling must be incorporated.

We discuss a number of potential problem areas. In each case, solutions are weighed, particularly in conjunction with the Lex scanner generator discussed in Section 3.4.

3.6.1 Processing Identifiers and Literals

In simple languages that have only global variables and declarations, the scanner commonly will immediately enter an identifier into the symbol table, if it is not already there. Whether the identifier is entered or is already in the table, a pointer to the symbol table entry is then returned from the scanner.

In block-structured languages, the scanner generally is not expected to enter or look up identifiers in the symbol table because an identifier can be used in many contexts (for example, as a variable, member of a class, or label). The scanner usually cannot know when an identifier should be entered into the symbol table for the current scope or when it should return a pointer to an instance from an earlier scope. Some scanners just copy the identifier into a private string variable (that can't be overwritten) and return a pointer to it. A later compiler phase, the type checker, then resolves the identifier's intended usage.

Sometimes a **string space** is used to store identifiers in conjunction with a symbol table (see Chapter Chapter:global:eight). A string space is an extendible block of memory used to store the text of identifiers. A string space eliminates frequent calls to memory allocators such as `new` or `malloc` to allocate private space for a string. It also avoids the space overhead of storing multiple copies of the same string. The scanner can enter an identifier into the string space and return a string space pointer rather than the actual text.

An alternative to a string space is a hash table that stores identifiers and assigns to each a unique **serial number**. A serial number is a small integer that can be used instead of a string space pointer. All identifiers that have the same text get the same serial number; identifiers with different texts get different serial numbers. Serial numbers are ideal indices into symbol tables (which need not be hashed) because they are small, contiguously assigned integers. A scanner can hash an identifier when it is scanned and return its serial number as part of the identifier token.

In some languages, such as C, C++, and Java, case is significant; in others, such as Ada and Pascal, it is not. When case is significant, identifier text must be stored or returned exactly as it was scanned. Reserved word lookup must distinguish between identifiers and reserved words that differ only in case. However, when case is insignificant, case differences in the spelling of an identifier or reserved word must be guaranteed to not cause errors. An easy way to do this is to put all tokens scanned as identifiers into a uniform case before they are returned or looked up in a reserved word table.

Other tokens, such as literals, require processing before they are returned. Integer and real (floating) literals are converted to numeric form and returned as part of the token. Numeric conversion can be tricky because of the danger of overflow or roundoff errors. It is wise to use standard library routines such as `atoi` and `atof` (in C) and `Integer.intValue` and `Float.floatValue` (in Java). For string literals, a pointer to the text of the string (with escaped characters expanded) should be returned.

The design of C contains a flaw that requires a C scanner to do a bit of special processing. Consider the character sequence `a (* b);`

This can be a call to procedure `a`, with `*b` as the parameter. If `a` has been declared in a `typedef` to be a type name, this character sequence also can be the declaration of an identifier `b` that is a pointer variable (the parentheses are not needed, but they are legal).

C contains no special marker that separates declarations from statements, so the parser will need some help in deciding whether it is seeing a procedure call or a variable declaration. One way to do this is for the scanner to create, while scanning and parsing, a table of currently visible identifiers that have been defined in `typedef` declarations. When an identifier in this table is scanned, a special `typeid` token is returned (rather than an ordinary `identifier` token). This allows the parser to distinguish the two constructs easily, since they now begin with different tokens.

Why does this complication exist in C? The `typedef` statement was not in the original definition of C in which the lexical and syntactic rules were established. When the `typedef` construct was added, the ambiguity was not immediately recognized (parentheses, after all, are rarely used in variable declarations). When the problem was finally recognized, it was too late, and the "trick" described previously had to be devised to resolve the correct usage.

Processing Reserved Words

Virtually all programming languages have symbols (such as `if` and `while`) that match the lexical syntax of ordinary identifiers. These symbols are called *keywords*. If the language has a rule that keywords may not be used as programmer-defined identifiers, then they are *reserved words*, that is, they are reserved for special use.

Most programming languages choose to make keywords reserved. This simplifies parsing, which drives the compilation process. It also makes programs more readable. For example, in Pascal and Ada, subprograms without parameters are called as `name;` (no parentheses required). But what if, for example, `begin` and `end` are not reserved and some devious programmer has declared procedures named `begin` and `end`? The result is a program whose meaning is not well-defined, as shown in the following example, which can be parsed in many ways:

```
begin
  begin;
  end;
  end;
  begin;
end
```

With careful design, you can avoid outright ambiguities. For example, in PL/I keywords are not reserved; procedures are called using an explicit `call` keyword. Nonetheless, opportunities for convoluted usage abound. Keywords may be used as variable names, allowing the following:

```
if if then else = then;
```

The problem with reserved words is that if they are too numerous, they may confuse inexperienced programmers, who may unknowingly choose an identifier name that clashes with a reserved word. This usually causes a syntax error in a program that "looks right" and in fact *would* be right if the symbol in question was not reserved. COBOL is infamous for this problem because it has several hundred reserved words. For example, in COBOL, `zero` is a reserved word. So is `zeros`. So is `zeroes`!

In Section 3.4.1, we showed how to recognize reserved words by creating distinct regular expressions for each. This approach was feasible because Lex (and Flex) allows more than one regular expression to match a character sequence, with the earliest expression that matches taking precedence. Creating regular expressions for each reserved word increases the number of states in the transition table that a scanner generator creates. In as simple a language as Pascal (which has only 35 reserved words), the number of states increases from 37 to 165 [Gra88]. With the transition table in uncompressed form and having 127 columns for ASCII characters (excluding null), the number of transition table entries increases from 4,699 to 20,955. This may not be a problem with modern multimegabyte memories. Still, some scanner generators, such as Flex, allow you to choose to optimize scanner size or scanner speed.

Exercise 18 establishes that any regular expression may be complemented to obtain all strings not in the original regular expression. That is, $\overline{A}$, the complement of A, is regular if A is. Using complementation of regular expressions, we can write a regular expression for nonreserved identifiers:

$(\overline{\overline{ident} \mid if \mid while \mid \ldots})$

That is, if we take the complement of the set containing reserved words and all nonidentifier strings, we get all strings that *are* identifiers, *excluding* the reserved words. Unfortunately, neither Lex nor Flex provides a complement operator for regular expressions (`^` works only on character sets).

We could just write down a regular expression directly, but this is too complex to consider seriously. Suppose `END` is the only reserved word and identifiers contain only letters. Then

$L \mid (LL) \mid ((LLL)L^{+}) \mid ((L -' E')L^{\star}) \mid (L(L -' N')L^{\star}) \mid (LL(L -' D')L^{\star})$

defines identifiers that are shorter or longer than three letters, that do not start with E, that are without N in position two, and so on.

Many hand-coded scanners treat reserved words as ordinary identifiers (as far as matching tokens is concerned) and then use a separate table lookup to detect them. Automatically generated scanners can also use this approach, especially if transition table size is an issue. After an apparent identifier is scanned, an exception table is consulted to see if a reserved word has been matched. When case is significant in reserved words, the exception lookup requires an exact match. Otherwise, the token should be translated to a standard form (all uppercase or lowercase) before the lookup.

An exception table may have any of various organizations. An obvious one is a sorted list of exceptions suitable for a binary search. A hash table also may be used. For example, the length of a token may be used as an index into a list of exceptions of the same length. If exception lengths are well-distributed, few comparisons will be needed to determine whether a token is an identifier or a reserved word. [Cic86] showed that perfect hash functions are possible. That is, each reserved word is mapped to a unique position in the exception table and no position in the table is unused. A token is either the reserved word selected by the hash function or an ordinary identifier.

If identifiers are entered into a string space or given a unique serial number by the scanner, then reserved words can be entered in advance. Then when what looks like an identifier is found to have a serial number or string space position *smaller* than the initial position assigned to identifiers, we know that a reserved word rather than an identifier has been scanned. In fact, with a little care we can assign initial serial numbers so that they match exactly the token codes used for reserved words. That is, if an identifier is found to have a serial number *s*, where *s* is less than the number of reserved words, then *s* must be the correct token code for the reserved word just scanned.

3.6.2 Using Compiler Directives and Listing Source Lines

Compiler directives and pragmas control compiler options (for example, listings, source file inclusion, conditional compilation, optimizations, and profiling). They may be processed either by the scanner or by subsequent compiler phases. If the directive is a simple flag, it can be extracted from a token. The command is then executed, and finally the token is deleted. More elaborate directives, such as Ada pragmas, have nontrivial structure and need to be parsed and translated like any other statement.

A scanner may have to handle source inclusion directives. These directives cause the scanner to suspend the reading of the current file and begin the reading and scanning of the contents of the specified file. Since an included file may itself contain an include directive, the scanner maintains a stack of open files. When the file at the top of the stack is completely scanned, it is popped and scanning resumes with the file now at the top of the stack. When the entire stack is empty, end-of-file is recognized and scanning is completed. Because C has a rather elaborate macro definition and expansion facility, macro processing and included files are typically handled by a preprocessing phase prior to scanning and parsing. The preprocessor, `cpp`, may in fact be used with languages other than C to obtain the effects of source file inclusion, macro processing, and so on.

Some languages (such as C and PL/I) include conditional compilation directives that control whether statements are compiled or ignored. Such directives are useful in creating multiple versions of a program from a common source. Usually, these directives have the general form of an `if` statement; hence, a conditional expression will be evaluated. Characters following the expression will then either be scanned and passed to the parser or be ignored until an `end if` delimiter is reached. If conditional compilation structures can be nested, a skeletal parser for the directives may be needed.

Another function of the scanner is to list source lines and to prepare for the possible generation of error messages. While straightforward, this requires a bit of care. The most obvious way to produce a source listing is to echo characters as they are read, using end-of-line characters to terminate a line, increment line counters, and so on. This approach has a number of shortcomings, however.

- Error messages may need to be printed. These should appear merged with source lines, with pointers to the offending symbol.
- A source line may need to be edited before it is written. This may involve inserting or deleting symbols (for example, for error repair), replacing symbols (because of macro preprocessing), and reformatting symbols (to prettyprint a program, that is, to print a program with text properly indented, `if-else` pairs aligned, and so on).
- Source lines that are read are not always in a one-to-one correspondence with source listing lines that are written. For example, in Unix a source program can legally be condensed into a single line (Unix places no limit on line lengths). A scanner that attempts to buffer entire source lines may well overflow buffer lengths.

In light of these considerations, it is best to build output lines (which normally are bounded by device limits) *incrementally* as tokens are scanned. The token image placed in the output buffer may not be an exact image of the token that was scanned, depending on error repair, prettyprinting, case conversion, or whatever else is required. If a token cannot fit on an output line, the line is written and the buffer is cleared. (To simplify editing, you should place source line numbers in the program's listing.) In rare cases, a token may need to be broken; for example, if a string is so long that its text exceeds the output line length.

Even if a source listing is not requested, each token should contain the line number in which it appeared. The token's position in the source line may also be useful. If an error involving the token is noted, the line number and position marker can used to improve the quality of error messages. by specifying where in the source file the error occurred. It is straightforward to open the source file and then list the source line containing the error, with the error message immediately below it. Sometimes, an error may not be detected until long after the line containing the error has been processed. An example of this is a `goto` to an undefined label. If such delayed errors are rare (as they usually are), a message citing a line number can be produced, for example, "Undefined label in statement 101." In languages that freely allow forward references, delayed errors may be numerous. For example, Java allows declarations of methods after they are called. In this case, a file of error messages keyed with line numbers can be written and later merged with the processed source lines to produce a complete source listing. Source line numbers are also required for reporting post-scanning errors in multipass compilers. For example, a type conversion error may arise during semantic analysis; associating a line number with the error message greatly helps a programmer understand and correct the error.

A common view is that compilers should just concentrate on translation and code generation and leave the listing and prettyprinting (but not error messages) to other tools. This considerably simplifies the scanner.

3.6.3 Terminating the Scanner

A scanner is designed to read input characters and partition them into tokens. When the end of the input file is reached, it is convenient to create an end-of-file pseudo-character.

In Java, for example, `InputStream.read()`, which reads a single byte, returns `-1` when end-of-file is reached. A constant, `Eof`, defined as `-1`, can be treated as an "extended" ASCII character. This character then allows the definition of an EndFile token that can be passed back to the parser. The EndFile token is useful in a CFG because it allows the parser to verify that the logical end of a program corresponds to its physical end. In fact, LL(1) parsers (discussed in Chapter Chapter:global:five) and LALR(1) parsers (discussed in Chapter Chapter:global:six) require an EndFile token.

What will happen if a scanner is called after end-of-file is reached? Obviously, a fatal error could be registered, but this would destroy our simple model in which the scanner always returns a token. A better approach is to continue to return the EndFile token to the parser. This allows the parser to handle termination cleanly, especially since the EndFile token is normally syntactically valid only after a complete program is parsed. If the EndFile token appears too soon or too late, the parser can perform error repair or issue a suitable error message.

3.6.4 Multicharacter Lookahead

We can generalize FAs to look ahead beyond the next input character. This feature is important for implementing a scanner for FORTRAN. In FORTRAN, the statement `DO 10 J = 1,100` specifies a loop, with index `J` ranging from `1` to `100`. In contrast, the statement `DO 10 J = 1.100` is an assignment to the variable `DO10J`. In FORTRAN, blanks are not significant except in strings. A FORTRAN scanner can determine whether the `O` is the last character of a `DO` token only after reading as far as the comma (or period). (In fact, the erroneous substitution of a "." for a "," in a FORTRAN `DO` loop once caused a 1960s-era space launch to fail!

Reserving space for: multichar

Figure 3.20: An FA that scans integer and real literals and the subrange operator.

Because the substitution resulted in a valid statement, the error was not detected until run-time, which in this case was *after* the rocket had been launched. The rocket deviated from course and had to be destroyed.)

We've already shown you a milder form of the extended lookahead problem that occurs in Pascal and Ada. Scanning, for example, `10..100` requires two-character lookahead after the `10`. Using the FA of Figure 3.20 and given `10..100`, we would scan three characters and stop in a nonaccepting state. Whenever we stop reading in a nonaccepting state, we can back up over accepted characters until an accepting state is found. Characters we back up over are rescanned to form later tokens. If no accepting state is reached during backup, we have a lexical error and invoke lexical error recovery.

In Pascal or Ada, more than two-character lookahead is not needed; this simplifies the buffering of characters to be rescanned. Alternatively, we can add a new accepting state to the previous FA that corresponds to a pseudotoken of the form $(D^+.)$. If this token is recognized, we strip the trailing "." from the token text and buffer it for later reuse. We then return the token code of an integer literal. In effect, we are simulating the effect of a context-sensitive match as provided by Lex's / operator.

Multiple character lookahead may also be a consideration in scanning *invalid* programs. For example, in C (and many other programming languages) `12.3e+q` is an invalid token. Many C compilers simply flag the entire character sequence as invalid (a floating-point value with an illegal exponent). If we follow our general scanning philosophy of matching the longest *valid* character sequence, the scanner could be backed up to produce four tokens. Since this token sequence

Buffered Token	Token Flag
`1`	Integer literal.
`12`	Integer literal.
`12.`	Floating-point literal.
`12.3`	Floating-point literal.
`12.3e`	Invalid (but valid prefix).
`12.3e+`	Invalid (but valid prefix).

Figure 3.21: Building the token buffer and setting token flags when scanning with a backup.

(`12.3`, `e`, `+`, `q`) is invalid, the parser will detect a syntax error when it processes the sequence. Whether we decide to consider this a lexical error or a syntax error (or both) is unimportant. Some phase of the compiler must detect the error.

It is not difficult to build a scanner that can perform general backup. This allows the scanner to operate correctly no matter how token definitions overlap. As each character is scanned, it is buffered and a flag is set indicating whether the character sequence scanned so far is a valid token (the flag might be the appropriate token code). If we are not in an accepting state and cannot scan any more characters, backup is invoked. We extract characters from the right end of the buffer and queue them for rescanning. This process continues until we reach a prefix of the scanned characters flagged as a valid token. This token is returned by the scanner. If no prefix is flagged as valid, we have a lexical error. (Lexical errors are discussed in Section 3.6.6.

Buffering and backup are essential in general-purpose scanners such as those generated by Lex. It is impossible to know in advance which regular expression pattern will be matched. Instead, the generated scanner (using its internal DFA) follows all patterns that are possible matches. If a particular pattern is found to be unmatchable, an alternative pattern that matches a shorter input sequence may be chosen. The scanner will back up to the longest input prefix that can be matched, saving buffered characters that will be matched in a later call to the scanner.

As an example of scanning with backup, consider the previous example of `12.3e+q`. Figure 3.21 shows how the buffer is built and flags are set. When the `q` is scanned, backup is invoked. The longest character sequence that is a valid token is `12.3`, so a floating-point literal is returned. `e+` is requeued so that it can be later rescanned.

3.6.5 Performance Considerations

Our main concern in this chapter is showing how to write correct and robust scanners. Because scanners do so much character-level processing, they can be a real performance bottleneck in production compilers. Hence, it is a good idea to consider how to increase scanning speed.

`System.out.println("Four score`	`and seven years ago,");`

Figure 3.22: An example of double buffering.

One approach to increasing scanner speed is to use a scanner generator such as Flex or GLA that is designed to generate fast scanners. These generators will incorporate many “tricks” that increase speed in nonobvious ways.

If you hand-code a scanner, a few general principles can increase scanner performance dramatically.

Try to block character-level operations whenever possible. It is usually better to do one operation on *n* characters rather than *n* operations on single characters. This is most apparent in reading characters. In the examples herein, characters are input one at a time, perhaps using Java's `InputStream.read` (or a C or C++ equivalent). Using single-character processing can be quite inefficient. A subroutine call can cost hundreds or thousands of instructions to execute—far too many for a single character. Routines such as `InputStream.read(buffer)` perform block reads, putting an entire block of characters directly into `buffer`. Usually, the number of characters read is set to the size of a disk block (512 or perhaps 1,024 bytes) so that an entire disk block can be read in one operation. If fewer than the requested number of characters are returned, we know we have reached end-of-file. An **end-of-file** (EOF) character can be set to indicate this.

One problem with reading blocks of characters is that the end of a block won't usually correspond to the end of a token. For example, near the end of a block may be found the beginning of a quoted string but not its end. Another read operation to get the rest of the string may overwrite the first part.

Double-buffering can avoid this problem, as shown in Figure 3.22. Input is first read into the left buffer and then into the right buffer, and then the left buffer is overwritten. Unless a token whose text we want to save is longer than the buffer length, tokens can cross a buffer boundary without difficulty. If the buffer size is made large enough (say 512 or 1,024 characters), the chance of losing part of a token is very low. If a token's length is near the buffer's length, we can extend the buffer size, perhaps by using Java-style `Vector` objects rather than arrays to implement buffers.

We can speed up a scanner not only by doing block reads, but also by avoiding unnecessary copying of characters. Because so many characters are scanned, moving them from one place to another can be costly. A block read enables direct reading into the scanning buffer rather than into an intermediate input buffer. As characters are scanned, we need not copy characters from the input buffer unless we recognize a token whose text must be saved or processed (an identifier or a literal). With care, we can process the token's text directly from the input buffer.

At some point, using a profiling tool such as `qpt`, `prof`, `gprof`, or `pixie` may allow you to find unexpected performance bottlenecks in a scanner.

3.6.6 Lexical Error Recovery

A character sequence that cannot be scanned into any valid token results in a **lexical error**. Although uncommon, such errors must be handled by a scanner. It is unreasonable to stop compilation because of what is often a minor error, so usually we try some sort of *lexical error recovery*. Two approaches come to mind:

1. Delete the characters read so far and restart scanning at the next unread character.

2. Delete the first character read by the scanner and resume scanning at the character following it.

Both approaches are reasonable. The former is easy to do. We just reset the scanner and begin scanning anew. The latter is a bit harder to do but also is a bit safer (because fewer characters are immediately deleted). Rescanning non-deleted characters can be implemented using the buffering mechanism described previously for scanner backup.

In most cases, a lexical error is caused by the appearance of some illegal character, which usually appears as the beginning of a token. In this case, the two approaches work equally well. The effects of lexical error recovery might well create a syntax error, which will be detected and handled by the parser. Consider `...for$tnight...`. The `$` would terminate scanning of `for`. Since no valid token begins with `$`, it would be deleted. Then `tnight` would be scanned as an identifier. The result would be `...for tnight...`, which will cause a syntax error. Such occurrences are unavoidable.

However, a good syntactic error-repair algorithm will often make some reasonable repair. In this case, returning a special warning token when a lexical error occurs can be useful. The semantic value of the warning token is the character string that is deleted to restart scanning. The warning token warns the parser that the next token is unreliable and that error repair may be required. The text that was deleted may be helpful in choosing the most appropriate repair.

Certain lexical errors require special care. In particular, runaway strings and comments should receive special error messages.

Handling Runaway Strings and Comments Using Error Tokens

In Java, strings are not allowed to cross line boundaries, so a runaway string is detected when an end-of-line character is reached within the string body. Ordinary recovery heuristics are often inappropriate for this error. In particular, deleting the first character (the double quote character) and restarting scanning will almost certainly lead to a cascade of further "false" errors because the string text is inappropriately scanned as ordinary input.

One way to catch runaway strings is to introduce an **error token**. An error token is not a valid token; it is never returned to the parser. Rather, it is a pattern for an error condition that needs special handling. We use an error token to represent a string terminated by an Eol rather than a double quote. For a valid string, in which internal double quotes and backslashes are escaped (and no other escaped characters are allowed), we can use

" (Not(" | Eol | \) | \" | \\ )* "

For a runaway string, we can use

" (Not(" | Eol | \) | \" | \\ )* Eol

When a runaway string token is recognized, a special error message should be issued. Further, the string may be repaired and made into a correct string by returning an ordinary string token with the opening double quote and closing Eol stripped (just as ordinary opening and closing double quotes are stripped). Note, however, that this repair may or may not be "correct." If the closing double quote is truly missing, the repair will be good. If it is present on a succeeding line, however, a cascade of inappropriate lexical and syntactic errors will follow until the closing double quote is finally reached.

Some PL/I compilers issue special warnings if comment delimiters appear within a string. Although such strings are legal, they almost always result from errors that cause a string to extend farther than was intended. A special string token can be used to implement such warnings. A valid string token is returned *and* an appropriate warning message is issued.

In languages such as C, C++, Java, and Pascal, which allow multiline comments, improperly terminated (that is, runaway) comments present a similar problem. A runaway comment is not detected until the scanner finds a close comment symbol (possibly belonging to some other comment) or until end-of-file is reached. Clearly, a special error message is required.

Consider the Pascal-style comments that begin with a { and end with a }. (Comments that begin and end with a pair of characters, such as `/*` and `*/` in Java, C, and C++, are a bit trickier to get right; see Exercise 6.)

Correct Pascal comments are defined quite simply: { Not(})* }

To handle comments terminated by Eof, the error token approach can be used: { Not(})* Eof

To handle comments closed by a close comment belonging to another comment (for example, {`...missing close comment...`{ `normal comment` }), we issue a warning (but not an error message; this form of comment is lexically legal). In particular, a comment containing an open comment symbol in its body is most probably a symptom of the kind of omission depicted previously. We therefore split the legal comment definition into two tokens. The one that accepts an open comment in its body causes a warning message to be printed ("`Possible unclosed comment`"). The result is three token definitions:

{ Not({ | })* } and { (Not({ | })* { Not({ | })*)$^+$ } and { Not(})* Eof

The first definition matches correct comments that do not contain an open comment in their bodies. The second matches correct, but suspect, comments that contain at least one open comment in their bodies. The final definition is an error token that matches a "runaway comment" terminated by end-of-file.

Single-line comments, found in Java and C++, are always terminated by an end-of-line character and so do not fall prey to the runaway comment problem. They do, however, require that each line of a multiline comment contain an open comment marker. Note, too, that we mentioned previously that balanced brackets cannot be correctly scanned using regular expressions and finite automata. A consequence of this limitation is that nested comments cannot be properly scanned using conventional techniques. This limitation causes problems when we want comments to nest, particularly when we "comment-out" a piece of code (which itself may well contain comments). Conditional compilation constructs, such as `#if` and `#endif` in C and C++, are designed to safely disable the compilation of selected parts of a program.

3.7 Regular Expressions and Finite Automata

Regular expressions are equivalent to FAs. In fact, the main job of a scanner generator program such as Lex is to transform a regular expression definition into an equivalent FA. It does this by first transforming the regular expression into a **nondeterministic finite automaton** (**NFA**). An NFA is a generalization of a DFA that allows transitions labeled with λ as well as multiple transitions from a state that have the same label.

After creating an NFA, a scanner generator then transforms the NFA into a DFA, introduced earlier in the chapter. Exactly how it does both of these steps is discussed a little later in this section.

An NFA, upon reading a particular input, need not make a unique (deterministic) choice of which state to visit. For example, as shown in Figure 3.23, an NFA is allowed to have a state that has two transitions (shown by the arrows) coming out of it, labeled by the same symbol. As shown in Figure 3.24, an NFA may also have transitions labeled with λ.

Transitions are normally labeled with individual characters in Σ, and although λ is a string (the string with no characters in it), it is definitely *not* a character. In the last example, when the FA is in the state at the left and the next input character is *a*, it may choose either to use the transition labeled *a* or to first follow the λ transition (you can always find λ wherever you look for it) and *then* follow an *a* transition. FAs that contain no λ transitions and that always have unique successor states for any symbol are *deterministic*.

The algorithm to make an FA from a regular expression proceeds in two steps. First, it transforms the regular expression into an NFA. Then it transforms the NFA into a DFA.

3.7.1 Transforming a Regular Expression into an NFA

Transforming a regular expression into an NFA is easy. A regular expression is built of the *atomic* regular expressions *a* (where *a* is a character in Σ) and λ by

Reserving space for: nfatwo

Figure 3.23: An NFA with two *a* transitions.

Reserving space for: nfalam

Figure 3.24: An NFA with a λ transition.

using the three operations AB, $A \mid B$, and $A^\star$. Other operations (such as A^+) are just abbreviations for combinations of these. As shown in Figure 3.25, NFAs for a and λ are trivial.

Now suppose we have NFAs for A and B and want one for $A \mid B$. We construct the NFA shown in Figure 3.26. The states labeled A and B were the accepting states of the automata for A and B; we create a new accepting state for the combined FA.

As shown in Figure 3.27, the construction of AB is even easier. The accepting state of the combined FA is the same as the accepting state of B.

Finally, the NFA for $A^\star$ is shown in Figure 3.28. The start state is an accepting state, so λ is accepted. Alternatively, we can follow a path through the FA for A one or more times so that zero or more strings that belong to A are matched.

Reserving space for: nfaalam

Figure 3.25: NFAs for a and λ.

Reserving space for: nfaab

Figure 3.26: An NFA for $A \mid B$.

Reserving space for: nfaabtwo

Figure 3.27: An NFA for AB.

3.7.2 Creating the DFA

The transformation from an NFA N to an equivalent DFA D works by what is sometimes called the **subset construction**. The subset construction algorithm is shown in Figure 3.29. The algorithm associates each state of D with a *set* of states of N. The idea is that D will be in state $\{x, y, z\}$ after reading a given input string if, and only if, N could be in *any* of the states x, y, or z, depending on the transitions it chooses. Thus D keeps track of all of the possible routes N might take and runs them simultaneously. Because N is a *finite* automaton, it has only a finite number of states. The number of subsets of N's states is also finite. This makes tracking various sets of states feasible.

The start state of D is the set of all states that N could be in without reading any input characters—that is, the set of states reachable from the start state of N following only λ transitions. Algorithm CLOSE, called from RECORDSTATE, in Figure 3.29 computes those states that can be reached after only λ transitions. Once the start state of D is built, we begin to create successor states.

To do this, we place each state S of D on a work list when it is created. For each state S on the work list and each character c in the vocabulary, we compute S's successor under c. S is identified with some set of N's states $\{n1, n2, \ldots\}$. We find all of the possible successor states to $\{n1, n2, \ldots\}$ under c and obtain a set $\{m1, m2, \ldots\}$. Finally, we include the λ-successors of $\{m1, m2, \ldots\}$. The resulting set of NFA states is included as a state in D, and a transition from S to it, labeled with c, is added to D. We continue adding states and transitions to D until all possible successors to existing states are added. Because each state

Reserving space for: nfaa

Figure 3.28: An NFA for $A^\star$.

corresponds to a (finite) subset of N's states, the process of adding new states to D must eventually terminate.

An accepting state of D is any set that contains an accepting state of N. This reflects the convention that N accepts if there is *any* way it could get to its accepting state by choosing the "right" transitions.

To see how the subset construction operates, consider the NFA shown in Figure 3.30. In the NFA in the figure, we start with state 1, the start state of N, and add state 2, its λ-successor. Hence, D's start state is $\{1, 2\}$. Under a, $\{1, 2\}$'s successor is $\{3, 4, 5\}$. State 1 has itself as a successor under b. When state 1's λ-successor, 2, is included, $\{1, 2\}$'s successor is $\{1, 2\}$. $\{3, 4, 5\}$'s successors under a and b are $\{5\}$ and $\{4, 5\}$. $\{4, 5\}$'s successor under b is $\{5\}$. Accepting states of D are those state sets that contain N's accepting state (5). The resulting DFA is shown in Figure 3.31.

It can be established that the DFA constructed by MAKEDETERMINISTIC is equivalent to the original NFA (see Exercise 20). What is not obvious is the fact that the DFA that is built can sometimes be *much* larger than the original NFA. States of the DFA are identified with sets of NFA states. If the NFA has n states, there are 2^n distinct sets of NFA states and hence the DFA may have as many as 2^n states. Exercise 16 discusses an NFA that actually exhibits this exponential blowup in size when it is made deterministic. Fortunately, the NFAs built from the kind of regular expressions used to specify programming language tokens do not exhibit this problem when they are made deterministic. As a rule, DFAs used for scanning are simple and compact.

When creating a DFA is impractical (either because of speed-of-generation

```
function MAKEDETERMINISTIC( $N$ ) : $DFA$
    $D.StartState \leftarrow$ RECORDSTATE( $\{ N.StartState \}$ )
    foreach $S \in WorkList$ do
        $WorkList \leftarrow WorkList - \{ S \}$
        foreach $c \in \Sigma$ do $D.T(S, c) \leftarrow$ RECORDSTATE( $\bigcup_{s \in S} N.T(s, c)$ )
    $D.AcceptStates \leftarrow \{ S \in D.States \mid S \cap N.AcceptStates \neq \emptyset \}$
end
function CLOSE( $S, T$ ) : $Set$
    $ans \leftarrow S$
    repeat
        $changed \leftarrow$ false
        foreach $s \in ans$ do
            foreach $t \in T(s, \lambda)$ do
                if $t \notin ans$
                then
                    $ans \leftarrow ans \cup \{ t \}$
                    $changed \leftarrow$ true
    until not $changed$
    return ($ans$)
end
function RECORDSTATE( $s$ ) : $Set$
    $s \leftarrow$ CLOSE( $s, N.T$ )
    if $s \notin D.States$
    then
        $D.States \leftarrow D.States \cup \{ s \}$
        $WorkList \leftarrow WorkList \cup \{ s \}$
    return ($s$)
end
```

Figure 3.29: Construction of a DFA D from an NFA N.

Reserving space for: nfasub

Figure 3.30: An NFA showing how subset construction operates.

Reserving space for: dfa2

Figure 3.31: DFA created for NFA of Figure 3.30.

or size concerns), an alternative is to scan using an NFA (see Exercise 17). Each possible path through an NFA can be tracked, and reachable accepting states can be identified. Scanning is slower using this approach, so it is usually used only when the construction of a DFA is not cost-effective.

3.7.3 Optimizing Finite Automata

We can improve the DFA created by MAKEDETERMINISTIC. Sometimes this DFA will have more states than necessary. For every DFA, there is a *unique* smallest (in terms of number of states) equivalent DFA. Suppose a DFA D has 75 states and there is a DFA D' with 50 states that accepts exactly the same set of strings. Suppose further that no DFA with fewer than 50 states is equivalent to D. Then D' is the only DFA with 50 states equivalent to D. Using the techniques discussed in this section, we can optimize D by replacing it with D'.

Some DFAs contain **unreachable states**, states that cannot be reached from the start state. Other DFAs may contain **dead states**, states that cannot reach any accepting state. It is clear that neither unreachable states nor dead states can participate in scanning any valid token. So we eliminate all such states as part of our optimization process.

We optimize the resulting DFA by merging states we know to be equivalent. For example, two accepting states that have no transitions out of them are equivalent. Why? Because they behave exactly the same way—they accept the string read so far but will accept no additional characters. If two states, s_1 and s_2, are equivalent, then all transitions to s_2 can be replaced with transitions to s_1. In effect, the two states are merged into one common state.

How do we decide what states to merge? We take a *greedy* approach and try the most optimistic merger. By definition, accepting and nonaccepting states are distinct, so we initially try to create only two states: one representing the merger of all accepting states and the other representing the merger of all nonaccepting states. Only two states is almost certainly too optimistic. In particular, all of the constituents of a merged state must agree on the same transition for each possible character. That is, for character c all of the merged states either must have no successor under c or must go to a single (possibly merged) state. If all constituents of a merged state do not agree on the transition to follow for some character, the merged state is split into two or more smaller states that *do* agree.

As an example, assume we start with the FA shown in Figure 3.32. Initially, we have a merged nonaccepting state $\{1, 2, 3, 5, 6\}$ and a merged accepting state $\{4, 7\}$. A merger is legal if, and only if, all constituent states agree on the same successor state for all characters. For example, states 3 and 6 would go to an accepting state when given character c; states 1, 2, and 5 would not, so a split must occur. We add an error state s_E to the original DFA that will be the successor state under any illegal character. (Thus reaching s_E becomes equivalent to detecting an illegal token.) s_E is not a real state. Rather, it allows us to assume that every state has a successor under every character. s_E is never merged with any real state.

Reserving space for: faaccept

Figure 3.32: Example FA before merging.

```
procedure SPLIT( $MergedStates$ )
    repeat
        $changed \leftarrow$ false
        foreach $S \in MergedStates, c \in \Sigma$ do
            $targets \leftarrow \bigcup_{s \in S}$ TARGETBLOCK( $s, c, MergedStates$ )
            if $|targets| > 1$
            then
                $changed \leftarrow$ true
                foreach $t \in targets$ do
                    $newblock \leftarrow \{ s \in S \mid$ TARGETBLOCK$( s, c, MergedStates ) = t \}$
                    $MergedStates \leftarrow MergedStates \cup \{ newblock \}$
                $MergedStates \leftarrow MergedStates - \{ S \}$
    until not $changed$
end
function TARGETBLOCK( $s, c, MergedStates$ ) : $MergedState$
    return $(B \in MergedStates \mid T(s, c) \in B)$
end
```

Figure 3.33: An algorithm to split FA states.

Algorithm SPLIT, shown in Figure 3.33, splits merged states whose constituents do not agree on a single successor state for a particular character. When SPLIT terminates, we know that the states that remain merged are equivalent in that they always agree on common successors.

Returning to the example, we initially have states $\{1, 2, 3, 5, 6\}$ and $\{4, 7\}$. Invoking SPLIT, we first observe that states 3 and 6 have a common successor under c and states 1, 2, and 5 have no successor under c (or, equivalently, they have the error state s_E). This forces a split that yields $\{1, 2, 5\}$, $\{3, 6\}$, and $\{4, 7\}$. Now, for character b states 2 and 5 go to the merged state $\{3, 6\}$, but state 1 does not, so another split occurs. We now have $\{1\}$, $\{2, 5\}$, $\{3, 6\}$, and $\{4, 7\}$. At this point, all constituents of merged states agree on the same successor for each input symbol, so we are done.

Once SPLIT is executed, we are essentially done. Transitions between merged states are the same as the transitions between states in the original DFA. That is, if there was a transition between states s_i and s_j under character c, there is now a transition under c from the merged state containing s_i to the merged state containing s_j. The start state is that merged state that contains the original start state. An accepting state is a merged state that contains accepting states (recall that accepting and nonaccepting states are never merged).

Returning to the example, the minimum state automaton we obtain is shown in Figure 3.34.

A proof of the correctness and optimality of this minimization algorithm can be found in most texts on automata theory, such as [HU79].

Reserving space for: minstate

Figure 3.34: The minimum state automaton obtained.

Reserving space for: newstates

Figure 3.35: An FA with new start and accepting states added.

Translating Finite Automata into Regular Expressions

So far, we have concentrated on the process of converting a given regular expression into an equivalent FA. This is the key step in Lex's construction of a scanner from a set of regular expression token patterns.

Since regular expressions, DFAs, and NFAs are interconvertible, it is also possible to derive for any FA a regular expression that describes the strings that the FA matches. In this section, we briefly discuss an algorithm that does this derivation. This algorithm is sometimes useful when you already have an FA you want to use but you need an regular expression to program Lex or to describe the FA's effect. This algorithm also helps you to see that regular expressions and FAs really are equivalent.

The algorithm we use is simple and elegant. We start with an FA and simplify it by removing states, one-by-one. Simplified FAs are equivalent to the original, except that transitions are now labeled with regular expressions rather than individual characters. We continue removing states until we have an FA with a single transition from the start state to a single accepting state. The regular expression that labels that single transition correctly describes the effect of the original FA.

Reserving space for: ttrans

Figure 3.36: The $T1$, $T2$, and $T3$ transformations.

To start, we assume our FA has a start state with no transitions into it and a single accepting state with no transitions out of it. If it doesn't meet these requirements, we can easily transform it by adding a new start state and a new accepting state linked to the original automaton with λ transitions. This is illustrated in Figure 3.35 using the FA we created with MAKEDETERMINISTIC in Section 3.7.2. We define three simple transformations, $T1$, $T2$, and $T3$, that will allow us to progressively simplify FAs. The first, illustrated in Figure 3.36(a), notes that if there are two different transitions between the same pair of states, with one transition labeled R and the other labeled S, then we can replace the two transitions with a new one labeled $R \mid S$. $T1$ simply reflects that we can choose to use either the first transition *or* the second.

Transformation $T2$, illustrated in Figure 3.36(b) allows us to *by pass* a state. That is, if state s has a transition to state r labeled X and state r has a transition to state u labeled Y, then we can go directly from state s to state u with a transition labeled XY.

Transformation $T3$, illustrated in Figure 3.36(c), is similar to transformation $T2$. It, too, allows us to bypass a state. Suppose state s has a transition to state r labeled X and state r has a transition to itself labeled Z, as well as a transition to state u labeled Y. We can go directly from state s to state u with a transition labeled $XZ^\star Y$. The $Z^\star$ term reflects that once we reach state r, we can cycle back into r zero or more times before finally proceeding to u.

We use transformations $T2$ and $T3$ as follows. We consider, in turn, each pair of predecessors and successors a state s has and use $T2$ or $T3$ to link a predecessor state directly to a successor state. In this case, s is no longer needed—all paths through the FA can bypass it! Since s isn't needed, we remove it. The FA is now simpler because it has one fewer states. By removing all states other than the start state and the accepting state (using transformation $T1$ when necessary), we will reach our goal. We will have an FA with only one transition, and the label on this transition will be the regular expression we want. FINDRE, shown in Figure 3.37, implements this algorithm. The algorithm begins by invoking AUGMENT, which introduces new start and accept states. The loop at Step **1** considers each state s of the FA in turn. Transformation $T1$ ensures that each pair of states is connected by at most one transition. This transformation is performed prior to processing s at Step **3**. State s is then eliminated by considering the cross-product of states with edges to and from s. For each such pair of states, transformation $T2$ or $T3$ is applied. State s is then removed at Step **2**, along with all edges to or from state s. When the algorithm terminates, the only states left are *NewStart* and *NewAccept*, introduced by AUGMENT. The regular expression for the FA labels the transition between these two states.

As an example, we find the regular expression corresponding to the FA in Section 3.7.2. The original FA, with a new start state and accepting state added, is shown in Figure 3.38(a). State 1 has a single predecessor, state 0, and a single successor, state 2. Using a $T3$ transformation, we add an arc directly from state 0 to state 2 and remove state 1. This is shown in Figure 3.38(b). State 2 has a single predecessor, state 0, and three successors, states 2, 4, and 5. Using three$T2$ transformations, we add arcs directly from state 0 to states 3, 4, and 5. State 2 is removed. This is shown in Figure 3.38(c).

State 4 has two predecessors, states 0 and 3. It has one successor, state 5. Using two $T2$ transformations, we add arcs directly from states 0 and 3 to state 5. State 4 is removed. This is shown in Figure 3.38(d). Two pairs of transitions are merged using $T1$ transformations to produce the FA in Figure 3.38(e). Finally, state 3 is bypassed with a $T2$ transformation and a pair of transitions are merged with a $T1$ transformation, as shown in Figure 3.38(f). The regular expression we obtain is

$$b^\star ab(a \mid b \mid \lambda) \mid b^\star aa \mid b^\star a.$$

By expanding the parenthesized subterm and then factoring a common term, we obtain

$$b^\star aba \mid b^\star abb \mid b^\star ab \mid b^\star aa \mid b^\star a \equiv b^\star a(ba \mid bb \mid b \mid a \mid \lambda).$$

Careful examination of the original FA verifies that this expression correctly describes it.

```
function FINDRE( $N$ ) : $RegExpr$
    $OrigStates \leftarrow N.States$
    call AUGMENT( $N$ )
    foreach $s \in OrigStates$ do                                          1
        call ELIMINATE( $s$ )
        $N.States \leftarrow N.States - \{ s \}$                           2
    /⋆ return the regular expression labeling the only remaining transition ⋆/
end
procedure ELIMINATE( $s$ )
    foreach $(x, y) \in N.States \times N.States \mid$ COUNTTRANS( $x, y$ ) $> 1$ do     3
        /⋆          Apply transformation T1 to $x$ and $y$          ⋆/
    foreach $p \in$ PREDS( $s$ ) $\mid p \neq s$ do
        foreach $u \in$ SUCCS( $s$ ) $\mid u \neq s$ do
            if $CountTrans(s, s) = 0$
            then  /⋆ Apply Transformation T2 to $p$, $s$, and $u$ ⋆/
            else  /⋆ Apply Transformation T3 to $p$, $s$, and $u$ ⋆/
end
function COUNTTRANS( $x, y$ ) : $Integer$
    return (number of transitions from $x$ to $y$)
end
function PREDS( $s$ ) : $Set$
    return ( $\{ p \mid (\exists\, a)(N.T(p, a) = s) \}$ )
end
function SUCCS( $s$ ) : $Set$
    return ( $\{ u \mid (\exists\, a)(N.T(s, a) = u) \}$ )
end
procedure AUGMENT( $N$ )
    $OldStart \leftarrow N.StartState$
    $NewStart \leftarrow$ NEWSTATE( )
    /⋆          Define $N.T(NewStart, \lambda) = \{ OldStart \}$          ⋆/
    $N.StartState \leftarrow NewStart$
    $OldAccepts \leftarrow N.AcceptStates$
    $NewAccept \leftarrow$ NEWSTATE( )
    foreach $s \in OldAccepts$ do
        /⋆          Define $N.T(s, \lambda) = \{ NewAccept \}$          ⋆/
    $N.AcceptStates \leftarrow \{ NewAccept \}$
end
```

Figure 3.37: An algorithm to generate a regular expression from an FA.

Reserving space for: findre

Figure 3.38: Finding a regular expression using FINDRE.

3.8 Summary

We have discussed three equivalent and interchangeable mechanisms for defining tokens: the regular expression, the deterministic finite automaton, and the nondeterministic finite automaton. Regular expressions are convenient for programmers because they allow the specification of token structure without regard for implementation considerations. Deterministic finite automata are useful in implementing scanners because they define token recognition simply and cleanly, on a character-by-character basis. Nondeterministic finite automata form a middle ground. Sometimes they are used for definitional purposes, when it is convenient to simply draw a simple automaton as a "flow diagram" of characters that are to be matched. Sometimes they are directly executed (see Exercise 17), when translation to deterministic finite automata is too costly or inconvenient. Familiarity with all three mechanisms will allow you to use the one best-suited to your needs.

Exercises

1. Assume the following text is presented to a C scanner:

```
main(){
    const float payment = 384.00;
    float bal;
    int month = 0;
    bal=15000;
    while (bal>0){
        printf("Month: %2d  Balance: %10.2f\n", month, bal);
        bal=bal-payment+0.015*bal;
        month=month+1;
    }
}
```

   What token sequence is produced? For which tokens must extra information be returned in addition to the token code?

2. How many lexical errors (if any) appear in the following C program? How should each error be handled by the scanner?

```
main(){
    if(1<2.)a=1.0else a=1.0e-n;
    subr('aa',"aaaaaa
               aaaaaa");
    /* That's all

}
```

3. Write regular expressions that define the strings recognized by the FAs in Figure 3.39.

4. Write DFAs that recognize the tokens defined by the following regular expressions.

   (a) $(a \mid (bc)^\star d)^+$

   (b) $((0 \mid 1)^\star (2 \mid 3)^+) \mid 0011$

   (c) $(a\ Not(a))^\star aaa$

5. Write a regular expression that defines a C-like, fixed-decimal literal with no superfluous leading or trailing zeros. That is, 0.0, 123.01, and 123005.0 are legal, but 00.0, 001.000, and 002345.1000 are illegal.

Reserving space for: regexpstr

Figure 3.39: FA for Exercise 3.

6. Write a regular expression that defines a C-like comment delimited by `/*` and `*/`. Individual `*`'s and `/`'s may appear in the comment body, but the pair `*/` may not.

7. Define a token class *AlmostReserved* to be those identifiers that are not reserved words but that would be if a single character were changed. Why is it useful to know that an identifier is "almost" a reserved word? How would you generalize a scanner to recognize *AlmostReserved* tokens as well as ordinary reserved words and identifiers?

8. When a compiler is first designed and implemented, it is wise to concentrate on correctness and simplicity of design. After the compiler is fully implemented and tested, you may need to increase compilation speed. How would you determine whether the scanner component of a compiler is a significant performance bottleneck? If it is, what might you do to improve performance (without affecting compiler correctness)?

9. Most compilers can produce a source listing of the program being compiled. This listing is usually just a copy of the source file, perhaps embellished with line numbers and page breaks. Assume you are to produce a prettyprinted listing.

   (a) How would you modify a Lex scanner specification to produce a prettyprinted listing?

   (b) How are compiler diagnostics and line numbering complicated when a prettyprinted listing is produced?

10. For most modern programming languages, scanners require little context information. That is, a token can be recognized by examining its text and

perhaps one or two lookahead characters. In Ada, however, additional context is required to distinguish between a single tic (comprising an attribute operator, as in `data'size`) and a tic-character-tic sequence (comprising a quoted character, as in `'x'`). Assume that a Boolean flag `can_parse_char` is set by the parser when a quoted character can be parsed. If the next input character is a tic, `can_parse_char` can be used to control how the tic is scanned. Explain how the `can_parse_char` flag can be cleanly integrated into a Lex-created scanner. The changes you suggest should not unnecessarily complicate or slow the scanning of ordinary tokens.

11. Unlike C, C++, and Java, FORTRAN generally ignores blanks and therefore may need extensive lookahead to determine how to scan an input line. We noted earlier a famous example of this: `DO 10 I = 1 , 10`, which produces seven tokens, in contrast with `DO 10 I = 1 .  10`, which produces three tokens.

    (a) How would you design a scanner to handle the extended lookahead that FORTRAN requires?

    (b) Lex contains a mechanism for doing lookahead of this sort. How would you match the identifier (`DO10I`) in this example?

12. Because FORTRAN generally ignores blanks, a character sequence containing n blanks can be scanned as many as 2^n different ways. Are each of these alternatives equally probable? If not, how would you alter the design you proposed in Exercise 11 to examine the most probable alternatives first?

13. You are to design the ultimate programming language, "Utopia 2010." You have already specified the language's tokens using regular expressions and the language's syntax using a CFG. Now you want to determine those token pairs that require whitespace to separate them (such as `else a`) and those that require extra lookahead during scanning (such as `10.0e-22`). Explain how you could use the regular expressions and CFG to automatically find all token pairs that need special handling.

14. Show that the set $\{[^k]^k \mid k \geq 1\}$ is not regular. *Hint*: Show that no fixed number of FA states is sufficient to exactly match left and right brackets.

15. Show the NFA that would be created for the following expression using the techniques of Section 3.7:

    $(ab^\star c) \mid (abc^\star)$

    Using MAKEDETERMINISTIC, translate the NFA into a DFA. Using the techniques of Section 3.7.3, optimize the DFA you created into a minimal state equivalent.

16. Consider the following regular expression:

    $(0 \mid 1)^\star 0(0 \mid 1)(0 \mid 1)(0 \mid 1) \ldots (0 \mid 1)$

Display the NFA corresponding to this expression. Show that the equivalent DFA is *exponentially* bigger than the NFA you presented.

17. Translation of a regular expression into an NFA is fast and simple. Creation of an equivalent DFA is slower and can lead to a much larger automaton. An interesting alternative is to scan using NFAs, thus obviating the need to ever build a DFA. The idea is to mimic the operation of the CLOSE and MAKEDETERMINISTIC routines (as defined in Section 3.7.2) while scanning. A set of possible states, rather than a single current state, is maintained. As characters are read, transitions from each state in the current set are followed, thereby creating a new set of states. If any state in the current set is final, the characters read will comprise a valid token.

    Define a suitable encoding for an NFA (perhaps a generalization of the transition table used for DFAs) and write a scanner driver that can use this encoding by following the set-of-states approach outlined previously. This approach to scanning will surely be slower than the standard approach, which uses DFAs. Under what circumstances is scanning using NFAs attractive?

18. Assume e is any regular expression. $\overline{e}$ represents the set of all strings not in the regular set defined by e. Show that $\overline{e}$ is a regular set.

    Hint: If e is a regular expression, there is an FA that recognizes the set defined by e. Transform this FA into one that will recognize $\overline{e}$.

19. Let *Rev* be the operator that reverses the sequence of characters within a string. For example, $Rev(abc) = cba$. Let R be any regular expression. $Rev(R)$ is the set of strings denoted by R, with each string reversed. Is $Rev(R)$ a regular set? Why?

20. Prove that the DFA constructed by MAKEDETERMINISTIC in Section 3.7.2 is equivalent to the original NFA. To do so, you must show that an input string can lead to a final state in the NFA if, and only if, that same string will lead to a final state in the corresponding DFA.

21. You have scanned an integer literal into a character buffer (perhaps `yytext`). You now want to convert the string representation of the literal into numeric (`int`) form. However, the string may represent a value too large to be represented in `int` form. Explain how to convert a string representation of an integer literal into numeric form with full overflow checking.

22. Write Lex regular expressions (using character classes if you wish) that match the following sets of strings:

    (a) The set of all unprintable ASCII characters (those before the blank and the very last character)

    (b) The string `["""]` (that is, a left bracket, three double quotes, and a right bracket)

(c) The string $x^{12,345}$ (your solution should be far less than 12,345 characters in length)

23. Write a Lex program that examines the words in an ASCII file and lists the ten most frequently used words. Your program should ignore case and should ignore words that appear in a predefined "don't care" list.

    What changes in your program are needed to make it recognize singular and plural nouns (for example, cat and cats) as the same word? How about different verb tenses (walk versus walked versus walking)?

24. Let *Double* be the set of strings defined as $\{s \mid s = ww\}$. *Double* contains only strings composed of two identical repeated pieces. For example, if you have a vocabulary of the ten digits 0 to 9, then the following strings (and many more!) are in *Double*: 11, 1212, 123123, 767767, 98769876, ...

    Assume you have a vocabulary consisting only of the single letter a. Is *Double* a regular set? Why?

    Assume you now have a vocabulary consisting of the two letters, a and b. Is Double a regular set? Why?

25. Let $Seq(x, y)$ be the set of all strings (of length 1 or more) composed of alternating x's and y's. For example, $Seq(a, b)$ contains a, b, ab, ba, aba, bab, $abab$, $baba$, and so on.

    Write a regular expression that defines $Seq(x, y)$.

    Let S be the set of all strings (of length 1 or more) composed of a's, b's, and c's that start with an a and in which no two adjacent characters are equal. For example, S contains a, ab, abc, $abca$, $acab$, $acac$, ... but not c, aa, abb, $abcc$, aab, cac, ... Write a regular expression that defines S. You may use $Seq(x, y)$ within your regular expression if you wish.

26. Let *AllButLast* be a function that returns all of a string but its last character. For example, $AllButLast(abc) = ab$. $AllButLast(\lambda)$ is undefined. Let R be any regular expression that does not generate λ. $AllButLast(R)$ is the set of strings denoted by R, with *AllButLast* applied to each string. Thus $AllButLast(a^{+}b) = a^{+}$. Show that $AllButLast(R)$ is a regular set.

27. Let F be any NFA that contains λtransitions. Write an algorithm that transforms F into an equivalent NFA F' that contains no λtransitions.

    Note: You need not use the subset construction, since you're creating a NFA, not a DFA.

28. Let s be a string. Define $Insert(s)$ to be the function that inserts a # into each possible position in s. If s is n characters long, $Insert(s)$ returns a set of $n + 1$ strings (since there are $n + 1$ places, a # may be inserted in a string of length n).

For example, $Insert(abc) = \{ \#abc, a\#bc, ab\#c, abc\# \}$. *Insert* applied to a set of strings is the union of *Insert* applied to members of the set. Thus $Insert(ab, de) = \{ \#ab, a\#b, ab\#, \#de, d\#e, de\# \}$.

Let R be any regular set. Show that $Insert(R)$ is a regular set.

Hint: Given an FA for R, construct one for $Insert(R)$.

29. Let D be any deterministic finite automaton. Assume that you know D contains exactly n states and that it accepts at least one string of length n or greater. Show that D must also accept at least one string of length $2n$ or greater.

Bibliography

[AMT89] Andrew W. Appel, James S. Mattson, and David R. Tarditi. *A lexical analyzer generator for Standard ML*. Princeton University, 1989. distributed with SML/NJ software.

[BC93] Peter Bumbulis and Donald D. Cowan. Re2c: a more versatile scanner generator. *ACM Letters on Programming Languages and Systems*, 2(1-4):70–84, 1993.

[Ber97] Elliot Berk. Jlex: A lexical analyzer generator for java. Princeton University, available at `http://www.cs.princeton.edu/ appel/-modern/java/JLex/manual.html`, 1997.

[Cic86] R.J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23:17–19, 1986.

[Gra88] Robert W. Gray. γ-gla: A generator for lexical analyzers that programmers can use. *Summer Usenix Conference Proceedings*, 1988.

[Gro89] J. Grosch. Efficient generation of lexical analysers. *Software – Practice and Experience*, 19:1089–1103, 1989.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Jac87] Van Jacobsen. Tuning unix lex, or it's not true what they say about lex. *Winter Usenix Conference Proceedings*, pages 163–164, 1987.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex and yacc*. O'Reilly and Associates, 1992.

[LS75] M.E. Lesk and E. Schmidt. Lex—a lexical analyzer generator. *Unix Programmer's Manual 2*, 1975.

[Moe91] H. Moessenboeck. A generator for production quality compilers. *Proc. of the 3rd int. workshop on compiler compilers*, 1991. Lecture Notes in Computer Science 477.

[NF88] T. Nguyen and K. Forester. Alex – an ada lexical analysis generator. Arcadia Document UCI-88-17, University of California, Irvine, 1988.

[Pax88] V. Paxson. Flex man pages. In ftp distribution from `//ftp.ee.lbl.gov`, 1988.

[PDC89] T.J. Parr, H.G. Dietz, and W.E. Cohen. *PCCTS Reference Manual*. Purdue University, 1989.